Information Extraction from the Web: Techniques and Applications

Alexander Yates

A dissertation submitted in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy

University of Washington

2007

Program Authorized to Offer Degree: Computer Science & Engineering

University of Washington
Graduate School

This is to certify that I have examined this copy of a doctoral dissertation by

Alexander Yates

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Chair of the Supervisory Committee:

_____

Oren Etzioni

Reading Committee:

_____

Oren Etzioni

_____

Pedro Domingos

_____

Daniel Weld

Date: _____

Signature_____

Date_____

University of Washington

Abstract

Information Extraction from the Web: Techniques and Applications

Alexander Yates

Chair of the Supervisory Committee:
Professor Oren Etzioni
Computer Science & Engineering

Web Information Extraction (WIE) systems have recently been able to extract massive quantities of relational data from online text. This has opened the possibility of achieving an elusive goal in Artificial Intelligence (AI): broad-coverage domain knowledge. AI systems depend to a great extent on having knowledge about the domains in which they operate, and such knowledge is typically expensive to enter into the system. Furthermore, the knowledge must be entered for every different domain in which an application is to operate. The Web contains knowledge about all kinds of different domains, but in a format that is not readily usable by AI systems. WIE promises to bridge the gap between the Web and AI.

Natural Language Processing is an example of an area in AI in which knowledge can make a dramatic difference in the performance of an application. Understanding or interpreting language depends on the ability to understand the words used in a domain. The meanings, usages, and syntactic properties of words, and the relative frequency with which certain words are used, are necessary pieces of information for effective language processing, and much of this information can be extracted from text. In one case study, this thesis examines methods for using extracted information in improving a particular kind of language processing tool, a parser.

Before information extraction can become broadly useful, however, more research must be done to improve the quality of the extracted information. A number of factors affect the quality, including correctness, importance or relevance, and the sophistication of meaning

representation. The second case study in this thesis investigates a method for resolving synonyms in extracted information. This technique changes the meaning representation of extractions from one that relates words or names to one that relates entities to one another.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to thank my advisor, Prof. Oren Etzioni, for his wisdom and patient guidance during my six years at the University of Washington. I have benefited from the advice of many people in the Department of Computer Science and Engineering, especially my labmates and Profs. Daniel Weld and Pedro Domingos. Thanks to all of them. Most of all, I would like to thank my parents for their constant love and support.

Chapter 1

INFORMATION EXTRACTION FROM THE WEB: TASKS AND APPLICATIONS

In recent years researchers have extracted huge amounts of structured information from large corpora like the Web. Novel systems like KnowItAll, KnowItNow, TextRunner, and others have demonstrated that it is possible to process huge corpora efficiently, and at the same time extract millions of facts with high accuracy. [42, 41, 43, 17, 8, 111, 2, 104, 16] As the field grows, it promises to provide vast resources of information organized into structured formats.

The large collections of knowledge produced by such systems give rise to new challenges and opportunities. In particular, they provide the opportunity to create domain-independent knowledge-based systems. However, several technical challenges stand in the way. Extracted information is only part of the way towards being fully machine-processable; much more research is required to convert text into logical representations of knowledge. Second, building a knowledge-based system using extracted information requires figuring out how to incorporate the extracted knowledge into complicated systems. And third, it requires building scalable solutions to dealing with the large amounts of knowledge. This thesis investigates techniques for meeting these challenges.

1.1 Information Extraction from the Web

Information Extraction is the process of retrieving structured information from unstructured text. Web Information Extraction (WIE) systems differ significantly from traditional systems in both methods and goals. Whereas traditional IE systems focus on squeezing as much juice as possible from small corpora, WIE systems focus on domain independent extraction from relatively simple sentences, and rely on the redundancy of the Web to provide

large quantities of information.

WIE systems face several new challenges because they must cope with the huge scale of the Web efficiently. Numerous authors contribute to the Web, so WIE systems must deal with contrasting linguistic styles and differing levels of linguistic proficiency, which can cause many language processing tools to degrade in performance. And most research in information extraction focuses on extracting instances for a set of predicates of interest, which depends on the domain. [55, 89, 108, 3, 2, 14, 43] To achieve domain independence, and thus to be able to extract information from arbitrary documents on the Web, WIE systems must somehow minimize or eliminate the number of inputs required per domain or predicate.

Several methodological innovations have helped make WIE possible and practical. Turney [116, 113] was the first to recognize that Web search engines could cheaply provide hitcounts for a statistical computation, Pointwise Mutual Information for Information Retrieval (PMI-IR), that is useful for information extraction. The KnowItAll system [42] extracts instances of any given class from the Web, using the PMI-IR statistics as evidence for validating extractions.

Hearst pioneered the use of generic patterns for information extraction [50]. Many systems since then have improved on the basic notion of pattern-based extraction by scaling it up to the Web and learning new patterns during the extraction process. [55, 89, 3, 43]. This form of learning, known as *bootstrapping*, can greatly increase the recall of extraction systems and enable them to specialize to a domain.

Bootstrapping can help reduce the cost of extraction for a new domain by automatically creating and learning new patterns for the relations in the domain, but it cannot determine the relations. In another advance for WIE, newer systems [8, 104] perform *Open Information Extraction*, in which all possible relations of interest are extracted from a corpus at the same time as the instances of those relations. This greatly increases the speed of WIE and the scale of the output.

1.2 Applications of Information Extraction

For information extraction, the Web is worth the trouble because it opens the possibility of achieving an elusive goal in Artificial Intelligence (AI): broad-coverage domain knowledge. AI systems depend to a great extent on having knowledge about the domains in which they operate, and such knowledge is typically expensive to enter into the system. Furthermore, the knowledge must be entered for every different domain in which an application is to operate. The Web contains knowledge about all kinds of different domains, but in a format that is not readily usable by AI systems. WIE promises to bridge the gap between the Web and AI.

The output of information extraction systems has been applied in several different types of applications, including Question-Answering [4, 75, 71] and review and opinion mining [116, 113, 92, 53, 58]. Typically, these applications are search-oriented, where a human user is a significant part of the process and can help to overcome mistakes or noise in the extraction process. Furthermore, the text-like extractions can be integrated into such applications relatively easily.

Several systems have gone further afield in applying information extraction output. Pentney et al. [90] use the results of information extraction as background knowledge to help determine what household activity a human subject might be performing, given observations from RFID tags about what objects the subject has touched. Several Open Mind [109, 110] projects have used the Open Mind entries in applications like word sense disambiguation [69], determining affect in text [70], and even robot navigation and control [47]. In these systems, however, the information extraction process is highly constrained and tailored to the domain of interest, so that the output is cleaner and more relevant to the domain.

The Cyc project [46] aims to build a large database of common-sense world knowledge, and has been used in a number of research and industrial knowledge-based applications. Most Cyc data is entered manually, but recent work has looked at populating part of the database automatically from the Web. [74] The segment of the database that is populated automatically must still be segregated from the rest, in order to ensure that the cleaner, manually-entered data is not polluted by the automatically extracted data.

1.2.1 Using Extracted Information in Natural Language Processing

Natural Language Processing (NLP) is an example of an area in AI in which knowledge can make a dramatic difference in the performance of an application. Understanding or interpreting language depends on the ability to understand the words used in a domain. The meanings, usages, and syntactic properties of words, and the relative frequency with which certain words are used, are necessary pieces of information for effective language processing, and much of this information can be extracted from text. The first case study in this thesis examines the application of WIE in improving a language processing tool.

One facet of language understanding that has particularly interested researchers in NLP is *parsing*, or the process of identifying the grammatical structure of a sentence. Statistical parsers, tools that automatically parse sentences, have been used as the backbone for a number of different NLP applications, including Question-Answering [62, 75, 82], Natural Language Interfaces [93, 124], and Information Extraction [20, 80]. Unfortunately, statistical parsers have relatively low accuracy, especially on texts (like Web text) that differ significantly in style and substance from the text they are trained on. [45]

The WOODWARD system applies the techniques and knowledge from WIE to statistical parsing, improving the performance of the widely-used Collins parser [29, 28] by 20%. [121] WOODWARD works by eliminating semantically implausible parses produced by a parser. The semantic plausibility of a sentence is judged by how close the semantic form of the sentence is to knowledge extracted from the Web.

WOODWARD introduces a novel method for incorporating extracted knowledge into a complex, statistical system. The *semantic filters* it employs require very small amounts of training data, and yet they are able to leverage the size and content of the Web to significantly boost the performance of an important NLP system.

A promising aspect of this avenue of research is the potential for positive feedback between information extraction and natural language processing. As systems like WOODWARD develop, they boost the performance of NLP tools using WIE. WIE, in turn, depends on NLP tools to help make correct extractions. [42] There is the potential for parsers that have been improved by WIE to yield better WIE systems in turn.

Chapter 3 describes the WOODWARD system in full, and presents experiments that demonstrate its ability to improve the Collins parser.

1.3 Extensions to Basic Information Extraction

While the use of extracted information in applications has begun to grow, much work remains to be done to make the quality of information more suitable to applications. Several salient factors affect the quality of IE, including:

1. correctness.

2. importance or relevance — it would be easy to generate an unlimited number of correct facts, simply by generating `isAnInteger(n)` for all integers n; what matters more is that the information is relevant and useful.

3. sophistication of meaning representation — whether the intended referent of object strings are identified or not; or whether the implied quantification in complex expressions is identified.

Most WIE researchers have focused on improving the correctness, or the precision and recall, of extraction (*e.g.*, [43, 39]). State-of-the-art WIE systems currently can produce large quantities of relational extractions with high precision. However, extractions differ from the full-blown First-Order Logic representations or other advanced representation languages that AI reasoning systems understand. For example, extractions may not distinguish between collective and distributive readings; they may not properly treat universal, existential, or other quantifiers; they may use the same string to represent several logically distinct objects or relations; and they may use several different strings to represent logically equivalent objects or relations. The translation from natural language to logic is extremely difficult, and these are some of the remaining unsolved sub-problems in that translation. But without solving these problems, the promise of WIE may be unfulfilled, as the extractions remain disconnected from the AI reasoning systems that stand to benefit so much from them.

Several projects have addressed one aspect of extending the meaning representation by handling polysemy in information extraction, although the problem is far from solved. [65, 19] Others have looked at handling polysemy in unsupervised settings [87, 125], which is similar to the WIE setting. Polysemy is a difficult problem, both to solve and to measure, and more research is needed to make such systems more accurate for Web text.

Other dimensions in which information extraction has been extended involve extracting meta-information about relations. The VerbOcean project [21] measures verb pairs in five ways: similarity, strength (*e.g.*, `kill` is stronger than `wound`), antonymy, enablement, and temporal sequentiality. Several systems have considered how to measure the relative strength of adjectival relations, such as that `huge` is stronger than `big`. [49, 48, 92]

1.3.1 Synonym Resolution for Information Extraction from the Web

Synonym Resolution is one important aspect of the problem of extending information extraction systems. The second case study in this thesis describes the RESOLVER system, which leverages WIE's large number of extractions and their relational structure to merge synonymous strings into clusters of co-referential names. RESOLVER can cluster both object strings and relation strings with high accuracy, and without the use of manually-labeled training data. Furthermore, it operates efficiently on the large data sets produced by WIE systems. [122]

RESOLVER uses a novel probabilistic model to measure the distributional similarity between terms. It then uses a novel clustering algorithm, which places efficient limits on the number of comparisons made between strings, to cluster strings in a set of extractions such that each cluster contains strings with high similarity to one another. Chapter 4 describes the RESOLVER system in detail, and presents experiments which demonstrate that RESOLVER outperforms other attempts at measuring similarity and clustering synonymous terms.

1.4 Contributions of the Thesis

This thesis investigates the following hypothesis: that *it is possible to resolve synonyms in Web-scale information extraction, and to use the extracted knowledge to improve natural*

language processing tasks.

Contributions of the thesis include:

1. an empirical demonstration that information extracted from the Web can improve natural language parsers, tools that are of particular interest to the natural language processing and computational linguistics communities.

2. a new technique for incorporating extracted information into natural language technology, via *semantic filters*.

3. a state of the art system for synonym resolution in large collections of extracted information.

4. a new, unsupervised probabilistic model for determining the probability that two strings are synonyms, given a set of extractions for each.

5. a fast clustering algorithm designed for synonym resolution, which provides a new upper bound on the number of pairwise comparisons required between strings.

The rest of the thesis is organized as follows. Chapter 2 discusses Web Information Extraction in depth, and describes two existing systems for performing WIE, KnowItAll and TextRunner. Chapter 3 presents the Woodward system for improving natural language parsers using Web-based semantic filters. Chapter 4 describes the Resolver system for synonym resolution on the Web. Chapter 5 concludes.

Chapter 2

WEB INFORMATION EXTRACTION SYSTEMS

This chapter describes two Web Information Extraction systems, KnowItAll [41, 43, 42] and TextRunner [8], which form the basis for the work for this thesis. Woodward and Resolver both adapt techniques and ideas from KnowItAll, and they both use TextRunner extractions as a source of knowledge. The next section describes KnowItAll, with an emphasis on ideas related to future chapters. Section 2.2 presents TextRunner, and focuses on its ability to extract massive quantities of information from unlabeled text without requiring manual identification of the relations to be extracted.

2.1 The KnowItAll *System*[1]

2.1.1 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. KnowItAll is 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.

Initial experiments with KnowItAll have focused on a sub-problem of information extraction, building lists of named entities found on the Web, such as instances of the

[1]This description of KnowItAll is a condensed version of [42].

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 here 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 [50], KnowItAll utilizes a set of eight domain-independent extraction patterns to *generate* candidate facts.[2] 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 [115], KnowItAll leverages existing Web search engines to compute these statistics efficiently.[3] 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 [40] 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 [41].

The two main contributions of KnowItAll covered here are:

---

[2]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".

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

1. It demonstrates 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. It shows that Web-based mutual information statistics can be effective in validating the output of an information extraction system.

Additions to KnowItAll have demonstrated ways to greatly increase the recall of the baseline KnowItAll system [43, 42], but the description below focuses on the core system.

The remainder of this section is organized as follows. It begins with an overview of KnowItAll's central design decisions. Sections 2.1.3 through 2.1.7 provide details about the key components of KnowItAll. Experiments in Section 2.1.8 demonstrate the ability of KnowItAll to extract information from the Web with high precision. Section 2.1.9 discusses related work, and Section 2.1.10 presents conclusions.

2.1.2 Overview of KnowItAll

The only domain-specific input to KnowItAll is a set of predicates that specify KnowItAll's focus (*e.g.*, Figure 2.6). While 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 2.1) 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 2.2 and pseudocode in Figure 2.3 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

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

Figure 2.1: **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.**

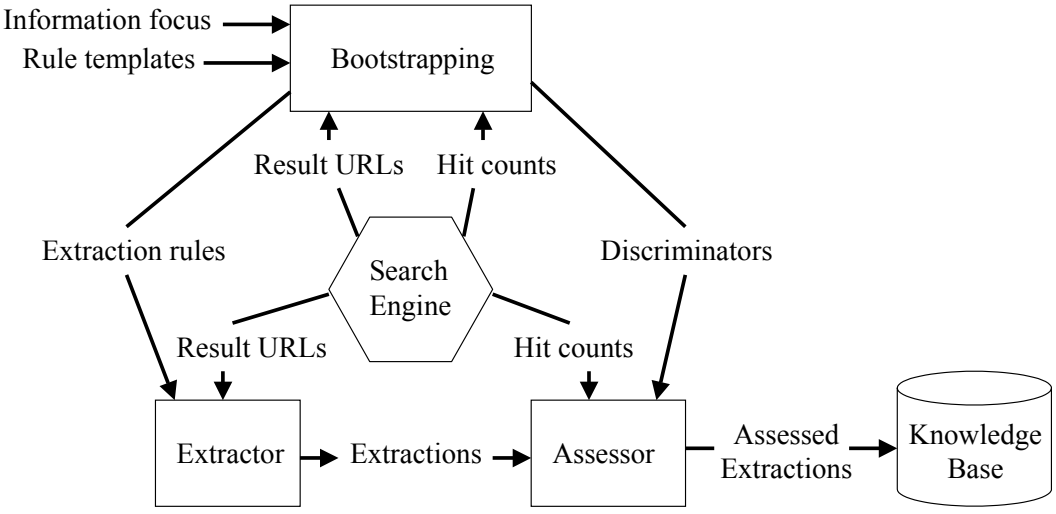


Figure 2.2: **Flowchart of the main components in KnowItAll.** Bootstrapping creates extractions rules and "discriminators" automatically with no hand-tagged training. Extractor fetches Web pages and applies extraction rules, then Assessor computes the probability of correctness before inserting into the Knowledge Base.

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 2.3: **High-level pseudocode for** KnowItAll**. (See Figure 2.10 for pseudocode of Bootstrap(I,T).)**

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.[4] This assessment process is an extension of Turney's PMI-IR algorithm [115].

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.

2.1.3 Extraction Rules and Discriminators

KnowItAll automatically creates a set of extraction rules for each predicate, as described in Section 2.1.4. 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 2.4, 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

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

arbitrary phrases.

The Extractor uses regular expressions based on part-of-speech tags from the Brill tagger [13] 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 2.4: **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 2.4. 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 2.1.5. A BNF description of the rule language is given in Figure 2.8. 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 2.9.

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 2.1.6. Figure 2.5 shows one of several possible discriminators that

can be used for the predicate `City` and for the binary predicate `CeoOf(Person,Company)`.

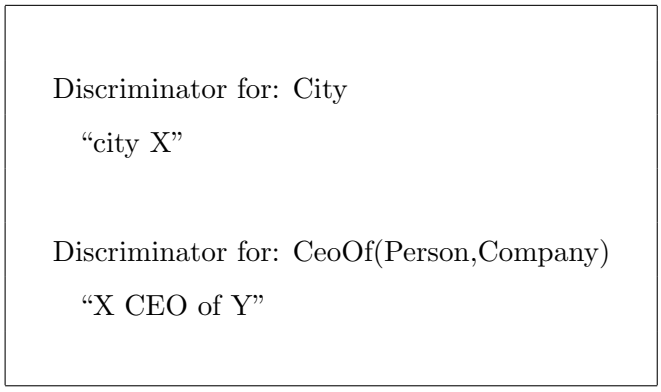


Figure 2.5: **When the discriminator for `City` is used to validate the extraction "Paris", the Assessor finds hit counts for the search query phrase "city Paris". Similarly, the discriminator for `CeoOf` validates Jeff Bezos as CEO of Amazon with the search query, "Jeff Bezos CEO of Amazon".**

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

2.1.4 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 2.6 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.

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

Figure 2.6: **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.**

The first step of Bootstrapping uses a set of domain-independent generic extraction patterns (*e.g.* Figure 2.1). The pattern in Figure 2.1 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 2.4 in Section 2.1.3) and a similar rule for the label “town”. The class-specific extraction patterns are:

“cities such as ” NPList

“towns such as ” NPList

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 2.7. Some of our rule templates are adapted from Marti Hearst's hyponym patterns [50] 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.

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 2.7: **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.

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 2.1.7.

2.1.5 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."

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 2.1) 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 2.4. 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 2.4 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 2.8.

The rule in Figure 2.4 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 2.9 shows a rule that finds instances 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 2.4 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

```
<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 2.8: **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.

results, and 3) apply the Extractor to sentences on each downloaded page. For robustness and scalability KnowItAll queries multiple different search engines.

2.1.6 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 *point-wise 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 [115]. 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`).[5] 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".

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 2.9: **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.**

[5] 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 pattern-learning techniques. [43, 42]

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

$$\text{PMI}(I, D) = \frac{|\text{Hits}(D + I)|}{|\text{Hits}(I)|} \tag{2.1}$$

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 2.2. 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.

$$P(\phi|f_1, f_2, \ldots f_n) = \frac{P(\phi) \prod_i P(f_i|\phi)}{P(\phi) \prod_i P(f_i|\phi) + P(\neg\phi) \prod_i P(f_i|\neg\phi)} \tag{2.2}$$

Our method to turn a PMI score into the conditional probabilities needed for Equation 2.2 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

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 2.10: **Pseudocode for Bootstrapping.**

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 [35] 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.

2.1.7 Training Discriminators

In order to estimate the probabilities $P(f_i|\phi)$ and $P(f_i|\neg\phi)$ needed in Equation 2.2, 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 below, 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 have been sufficient to produce correct seeds for all the classes we have experimented with thus far.

2.1.8 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 2.1.6.

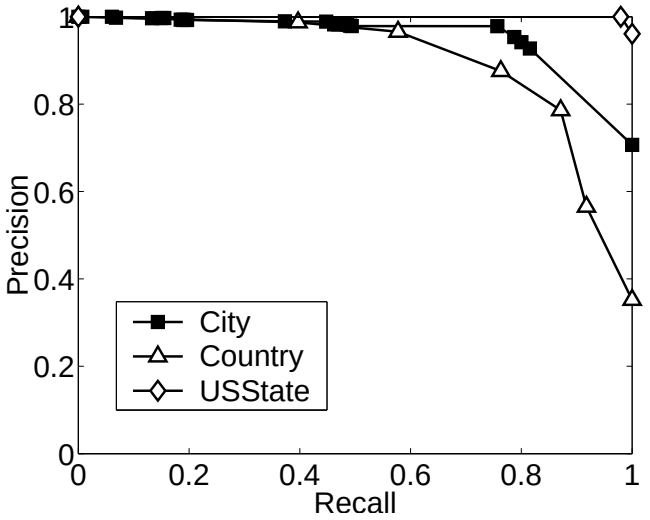
We use the standard metrics of *precision* and *recall* to measure KnowItAll's performance. At each probability p assigned by the Assessor, we count the number of correct extractions at or above probability p. This is done by first comparing the extracted instances automatically with an external knowledge 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.

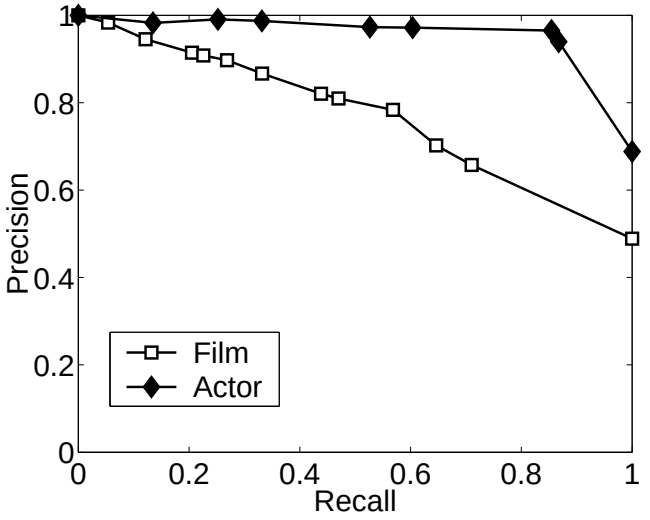
Experiments 4 and 5 show precision and recall at the end of running KnowItAll for four days. Each point on the curves shows the precision and recall for extractions with probability at or above a given level. The curve for `City` has precision 0.98 at recall 0.76, then drops to precision 0.71 at recall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



Experiment 1: **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 2: **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.**

at recall 0.87.

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.

2.1.9 Related Work

One of KnowItAll's main contributions is adapting Turney's PMI-IR algorithm [115, 116, 117] 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.* [71] 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 [118] 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. Un-

like IE systems that use supervised learning techniques such as *hidden Markov models* (HMMs) [44], rule learning [108, 18, 23], maximum entropy [85], or Conditional Random Fields [76], 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 [11], which alternately learns using two orthogonal view of the data. Jones *et al.* [55] gives a good overview of methods used in bootstrap learning. IE systems that use bootstrapping include [99, 1, 14, 86, 27, 24]. 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 [62] and further articulated in AskMSR [75]. 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.

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 [34], 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 [31]. 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. Finally, KnowItAll also shares the motivation of Schubert's project [102], which seeks to derive general world knowledge from texts. However, Schubert and his colleagues have focused on highly-structured texts such as WordNet and the Brown corpus whereas KnowItAll has focused on the Web.

2.1.10 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 as 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.

KnowItAll 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. However, several drawbacks to the KnowItAll architecture, including the inherent slow speed in using search engine queries and the need for users to specify the relations of interest, have led to a new system and architecture for Web Information Extraction, described next.

2.2 TextRunner and Open Information Extraction[6]

2.2.1 Introduction and Motivation

Traditional information extraction systems have focused on satisfying precise, narrow, pre-specified requests from small, homogeneous corpora. In contrast, the TextRunner system demonstrates a new kind of information extraction, called *Open Information Extraction* (Open IE), in which the system makes a single, data-driven pass over the entire corpus and extracts a large set of relational tuples, without requiring *any* human input. [8] TextRunner is a fully-implemented, highly scalable example of Open IE. TextRunner's extractions are indexed, allowing a fast query mechanism.

Systems designed for extracting instances of particular relations on small, homogeneous corpora have trouble with the new requirements of Open IE. KnowItAll, which is designed for unsupervised Web IE, can handle the heterogeneity of the Web without requiring manually labeled training examples. However, even KnowItAll still requires manual inputs in the form of target relations, and it is significantly slowed down by its search engine queries. Open IE has even more stringent automation requirements, and aims for much greater scalability.

OIE marks a significant departure from traditional information extraction in two main ways:

Automation and Scalability: The first step in automating IE came with the use of machine learning techniques to replace the early, brittle, hand-crafted rule-based systems. [98, 31] These systems were able to extract information in specialized domains by learning patterns from manually labeled data. Later techniques showed how bootstrapping methods could do the same with much less seed data, or starting with a few hand-crafted extraction patterns. [14, 3, 96] Nevertheless, all of these systems require some manual input for every relation, and potentially a substantial effort to create hand-tagged data. Open IE seeks to do away with any amount of manual effort that is specific to a relation. This greater degree of automation also entails better scalability,

[6]This description of TextRunner is based on Banko *et al.*'s description in [8] and Yates *et al.*'s description in [123].

since an Open IE system is free to extract information about all relations in a corpus, without the prohibitive manual effort required by traditional IE systems.

Corpus Heterogeneity: Previous work has used a wide variety of statistical methods for information extraction, including kernel-based methods [15], maximum-entropy models [56], graphical models [100, 32], and co-occurrence statistics [68, 22]. However, they all tend to be united in the use of heavyweight linguistic tools, such as named-entity recognizers, parsers, or dependency parsers. These tools tend to perform well when trained and tested on the same genre or style of text, such as the newswire text in the Penn Treebank corpus. However, they tend to have difficulty when applied to highly heterogeneous text like that found on the Web. [45] Named entities on the Web can fall into many more categories than the traditional Person, Location, and Organization categories identified by named entity recognizers, so these systems have particular difficulty on Web text. [37]

The TextRunner system, a fully-automated, highly-scalable, and efficient Open Information Extraction system, demonstrates the feasibility and benefits of Open IE. The main contributions are:

1. A demonstration that Open IE can work, using novel techniques such as an extraction mechanism that extracts relations at the same time that it extracts arguments to the relations.

2. A comparison between TextRunner and KnowItAll that shows that TextRunner can extract the same amount of information per relation as KnowItAll, but with a 33% lower error rate and with far less CPU time spent on each relation.

3. An analysis of TextRunner's 11 million highest probability extractions from a corpus of 9 million Web documents, which provides a better understanding of the type and quality of information TextRunner is able to extract.

2.2.2 Overview of the TextRunner *System*

Much like KnowItAll, TextRunner is divided into extraction and assessment modules, with a separate phase for training the extractor that has similarities to the KnowItAll bootstrapping paradigm. However, each module is fundamentally different from the KnowItAll analog in order to support Open IE. The Single-Pass Extractor module reads through the entire corpus once, processing each sentence separately and efficiently, and produces extractions as it goes. Its only input is the corpus. The Assessor module uses the set of extractions to make judgments about the probability that each extraction is correct. Again, it requires only the extracted data as input; it does not require search engine counts, like KnowItAll, or manually labeled examples, like many traditional information extraction systems. The bootstrapping module, called a *Self-Supervised Classifier*, trains the extractor using an unlabeled corpus, a parser, and a set of general heuristics about which subtrees of a parse provide reliably good extractions. Figure 2.11 shows the architecture of the full TextRunner system.

In the following sections, we present the main modules of TextRunner in detail. Section 2.2.3 presents the Single-Pass Extractor and the TextRunner data model. Section 2.2.4 shows how the Self-Supervised Classifier is trained. Section 2.2.5 illustrates the Assessor module. Section 2.2.7 analyzes TextRunner's efficiency in extraction, both theoretically and in practice. Section 2.2.8 experimentally compares TextRunner with KnowItAll, and Section 2.2.9 gives an overall evaluation of TextRunner's extraction accuracy. Section 2.2.11 concludes.

2.2.3 Single Pass Extractor

The TextRunner Extractor makes a single pass over all documents, processing each sentence in three stages. First, it tags the sentence with part-of-speech tags and noun phrase chunks. Second, for each pair of noun phrases that are not too far apart, and subject to several other constraints, it identifies candidate extractions. It then applies a classifier described below to determine whether or not to extract the relationship. If the classifier deems the extraction trustworthy, a tuple of the form `t = (`e_i`,` r_j`,` e_k`)` is extracted, where

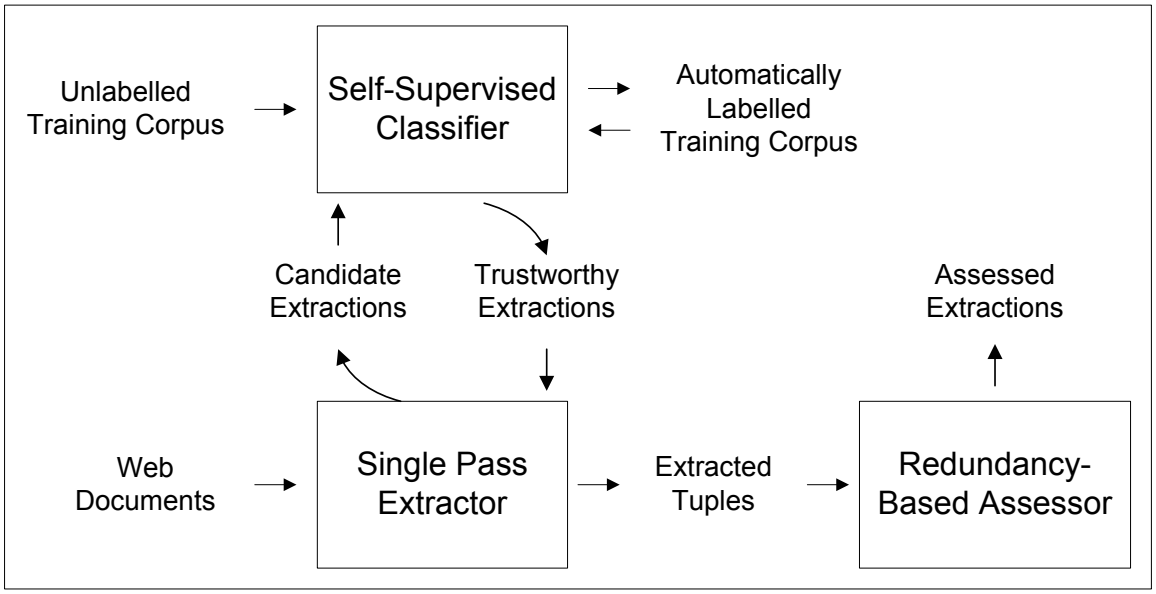


Figure 2.11: **Flowchart of the main components in** TEXTRUNNER**.** The Self-Supervised Classifier automatically labels a set of extractions, and then trains itself from it. The Extractor uses the classifier to find high-quality extractions from unlabeled Web text. The Assessor assigns probabilities to each extraction.

e_i, e_k are entities and r_j is the relation between them. For example, given the sentence "Everybody knows Edison invented the light bulb," TEXTRUNNER can extract the tuple `(Edison, invented, light bulb)`.

The first phase of TEXTRUNNER's extraction process, tagging and chunking, uses lightweight, maximum-entropy models [94] from the OpenNLP Toolkit.[7] It explicitly avoids the use of a parser or dependency parser, which would make the processing time super-linear in the length of the sentence. Furthermore, the tagging and chunking are easier to transport to other languages than are parsers, and they are more robust in the face of the diversity of language on the Web.

The second step of the extraction process finds candidate tuples. For each pair of noun chunks in the same sentence, TEXTRUNNER creates a candidate tuple using the pair of noun chunks as the argument strings, and the text between them as the relation string.

[7] http://opennlp.sourceforge.net/

The relation string is heuristically cleaned up by removing stopwords and non-essential adverbial or prepositional phrases.

The third step analyzes the candidate tuples and decides whether or not they are trustworthy extractions by applying the Self-Supervised Classifier. This classifier is a standard Naïve Bayes model whose features are part-of-speech tags and chunk tags from the sentence that a candidate tuple is taken from, plus some features that describe the length of the extracted strings. A partial list of the features used for the classifier is shown in Figure 2.12. As its name suggests, the classifier is trained using an automatic labeling process, which is described in the next section.

Whether or not the relation string contains POS sequence `(VBD, IN)`
The number of tokens in the relation string
The number of stop words in the relation string
Whether argument 1 is a proper noun
Whether argument 2 is a proper noun
The POS tag to the left of argument 1
The POS tag to the right of argument 2

Figure 2.12: **Example features used by the Self-Supervised Classifier in** TextRunner**.**

Finally, all extractions are sorted and merged such that the final set of extractions contains only one copy of every extraction, together with a count of how often each was extracted.

2.2.4 Self-Supervised Classifier

Just as KnowItAll uses a bootstrapping process to train itself (see Section 2.1.4), the Self-Supervised Classifier module in TextRunner trains a classifier for predicting whether or not an extraction is good or not by automatically labeling a set of candidate extractions. The classifier for TextRunner, however, is much faster than KnowItAll's Assessor, since it does not rely on any search engine queries. Because of its speed, it is called directly by

The path connects two constituents that each span an argument to the relation.
The path is no longer than a certain maximum length.
The path does not cross a sentence-like boundary (S, SQ, SBAR, SINV constituents).
Neither argument consists solely of a pronoun.

Figure 2.13: **Example constraints on paths through a parse tree that span likely extractions. The Self-Supervised Classifier in TextRunner uses these constraints to automatically label an extraction as being correct or incorrect.**

the inner loop of the Single-Pass Extractor: as a candidate tuple is extracted, the classifier determines which candidates should be part of the final set of extractions.

The Self-Supervised Classifier is able to train itself because it is able to automatically label a set of example extractions taken from an unlabeled training corpus. To generate labeled examples, it parses a set of sentences using an unlexicalized parser. [59] For each pair of base noun phrases (e_i, e_j), the system finds a minimal path through the parse tree that connects them, and creates a relation string and candidate extraction from the words spanned. It then applies a set of heuristic constraints to the path through the parse tree, and labels the candidate extraction correct if all constraints are met; otherwise, it is labeled incorrect. The constraints capture general notions about what type of grammatical relations indicate that there is likely to be a semantic relation; some example constraints are listed in Figure 2.13.

Full parsing is an expensive operation to apply to the Web. However, this startup process can be applied to a small sample of sentences, and the parser is then never used during the extraction process.

Once a labeled training set is created, the Naïve Bayes model can be trained in the standard way, by counting the frequency of each feature for both correct and incorrect extractions. The trained classifier is language-specific, but it contains no relation-specific or lexical features. As a result, it can be applied to arbitrary relations, and to arbitrary domains.

2.2.5 Redundancy-Based Assessor

While the Self-Supervised Classifier can determine which extractions are likely to be correct with high accuracy, there is another source of unused evidence in the extracted data that can be exploited to improve the accuracy further. The Redundancy-Based Assessor, based on the URNS model [38], uses global counts in the data to assign each extraction a probability of correctness. The probabilities can then be used to rank or filter the extractions.

The URNS model is an unsupervised model for determining the correctness of extractions based on the redundancy of extracted information. If an extraction is seen multiple times in different sentences, the model assigns it a higher probability than if it is seen only a small number of times. The exact probability score depends, however, on the total number of extractions. In previous experiments, it is shown to assign far more accurate probabilities than the noisy-or model or PMI-based techniques. [38] The model is summarized below.

URNS models information extraction as a process of drawing labeled balls from an urn, with replacement. The labels denote the strings being extracted; in TEXTRUNNER's case, each ball is labeled with a tuple. Because some tuples are more likely to be extracted than others, URNS allows labels to be repeated on multiple balls. There are two observed parameters in the URNS model, the number of times k that a label is extracted, and the total number of draws n from the urn. Given these two counts and several parameters that describe the frequency of correct and incorrect labels in the urn, the URNS model can assign an exact probability that a particular label is a correct extraction.

The parameters required to describe an urn completely are:

- C, the set of distinct correct labels (extractions).
- E, the set of distinct incorrect or error labels.
- $num(l)$, the number of balls in the urn with the label l, for every $l \in C \cup E$.

For a set of labels L, let $|L|$ denote the number of unique labels in L. Let $num(L)$ be the multi-set giving the number of balls for every label in L. Let $|num(L)|$ be the total number

of balls for every label in L, or $|num(L)| = \sum_{l \in L} num(l)$. Finally, let $T = |num(C \cup E)|$, the total number of balls in the urn. The URNS model is given by the following expression:

$$P(x \in C | x \text{ appears } k \text{ times in } n \text{ draws}) = \frac{\sum_{r \in num(C)} \left(\frac{r}{T}\right)^k \left(1 - \frac{r}{T}\right)^{n-k}}{\sum_{r' \in num(C \cup E)} \left(\frac{r'}{T}\right)^k \left(1 - \frac{r'}{T}\right)^{n-k}} \quad (2.3)$$

Following extraction, TEXTRUNNER counts the number of sentences from which each tuple was extracted. After the tuples are sorted, the assessor also calculates the number of times each relation string is extracted. These two counts are used as the observed variables k and n to assess the probability score for an extraction. The other parameters are estimated from unlabeled data, as described by Downey *et al.*[38]

2.2.6 Search Interface

TEXTRUNNER builds an inverted index of the extracted tuples, and spreads it across a cluster of machines. This architecture supports fast, interactive, and powerful relational queries. Users enter keywords, and TEXTRUNNER quickly returns the set of extractions matching the query. For example, a query for "Newton" will return tuples like *(Newton, invented, calculus)*.

The current interface to TEXTRUNNER is available over the Web at http://www.cs.washington.edu/research/textrunner/. Figure 2.14 shows a screenshot of the search page, and Figure 2.15 shows an example results page, the top extractions for the predicate `invented`.

2.2.7 Analysis of Scalability

The theoretical performance of TEXTRUNNER is better than KNOWITALL's and other information extraction systems because it is not a function of R, the number of relations being extracted. TEXTRUNNER's Single-Pass Extractor runs in time $O(D)$, where D is the number of documents in the corpus. Sorting, counting, and assessing the extractions collectively take time $O(T \log T)$, where T is the number of extracted tuples. In contrast, a traditional information extraction system must process the entire corpus every time it is given a new set of relations, so it takes time $O(R \cdot D)$. Since R can be quite large on the Web, this results in a significant advantage for TEXTRUNNER and Open Information Extraction.

Figure 2.14: **Screenshot of the TextRunner Search interface, available over the Web at http://www.cs.washington.edu/research/textrunner/. In this screenshot, a user has entered the keyword `invented` into the search field.**

Figure 2.15: **The top extractions returned by TextRunner for the keyword `invented`. Extractions are grouped by the predicate and first argument; each second argument belonging to the same group is listed on the same line.**

TextRunner is fast in practice as well. It took less than 68 CPU hours to extract tuples from 162 million sentences in a test corpus of 9 million Web documents. The process is easily parallelized, and took only 4 hours to run on a cluster of 20 machines. An additional 5 hours is needed to sort and assess the extracted tuples. TextRunner spends about 0.036 seconds per sentence during the extraction phase, more than 80 times faster than the speed of dependency parsers measured on this corpus. They took 3 seconds per sentence, on average.

2.2.8 Comparison with KnowItAll

The experiment below demonstrates that in a head-to-head comparison between KnowItAll and TextRunner, TextRunner substantially reduces the error rate while extracting almost identical numbers of a facts for a pre-selected set of relations. At the same time, TextRunner uses significantly less time per relation, and extracts orders of magnitude more relations, than KnowItAll.

For this experiment, both KnowItAll and TextRunner are restricted to a test corpus of 9 million Web documents. Since KnowItAll is not an Open IE system, it requires a set of predicates as input. The experiment compares KnowItAll and TextRunner on a set of ten predicates that were randomly selected from those predicates that appeared in at least 1,000 sentences in the test corpus. The ten predicates are listed in Figure 2.16.

The results for each system are summarized in Table 2.1. Both systems find almost the same number of instances for these ten predicates, but TextRunner's error rate is 33% lower than KnowItAll's. Moreover, KnowItAll took 63 CPU hours to extract instances for these ten relations, or 6.3 hours per relation. In contrast, TextRunner took 85 CPU hours to extract instances for *all* relations it could find in the corpus. Although it is difficult to say exactly how many distinct, correct relations are contained in the TextRunner extractions, it is certainly several orders of magnitude more than the ten relations processed by KnowItAll.

(<proper noun>, acquired, <proper noun>)

(<proper noun>, graduated from, <proper noun>)

(<proper noun>, is author of, <proper noun>)

(<proper noun>, is based in, <proper noun>)

(<proper noun>, studied, <noun phrase>)

(<proper noun>, studied at, <proper noun>)

(<proper noun>, was developed by, <proper noun>)

(<proper noun>, was formed in, <year>)

(<proper noun>, was founded by, <proper noun>)

(<proper noun>, worked with, <proper noun>)

Figure 2.16: **The ten predicates used by** KnowItAll **and** TextRunner **in their head-to-head comparison.**

Table 2.1: **A head-to-head comparison between** KnowItAll **and** TextRunner **on a predefined set of 10 predicates.** TextRunner **has a 33% lower error rate, while extracting an almost identical number of extractions.**

	Average Error Rate	Correct Extractions
TextRunner	12%	11,476
KnowItAll	18%	11,631

2.2.9 Analysis of TextRunner*'s Extractions*

To convey the quality and type of extractions that TextRunner extracts, the following analysis classifies TextRunner's extractions in four dimensions:

1. Correctness — whether the extraction has the same truth value as conveyed by the sentence it was extracted from.

2. Abstractness — whether the truth of the extraction is grounded in particular entities (a *concrete* extraction), or the extraction is general and underspecified (an *abstract* extraction).

3. Well-formedness of the relation — relations are judged to be well-formed if there exists some pair of arguments for which the relation holds. For example, `invented` is a well-formed relation, but `of securing` in the extraction `(demands, of securing, border)` is not.

4. Well-formedness of the arguments — arguments to a relation r are said to be well-formed for r if they are of the correct type. For example, `(29, dropped, instruments)` does not have well-formed arguments.

TextRunner extracts 60.5 million tuples from the 9 million page test corpus. To narrow down the analysis, only a subset of high-quality extractions are examined. This subset contains all extractions for which the following criteria are met:

1. The Assessor assigns a probability ≥ 0.8.

2. The extraction's relation appears in at least 10 distinct sentences in the corpus.

3. The relation does not appear in the top 0.1% of all relations, ranked according to the number of sentences they appear in. These high-frequency relations have a tendency to be overly vague, such as `has` or `is`.

The resulting subset contains 11.3 million tuples, with 278,085 distinct relation strings.

A random sample of 400 of these high quality extractions was manually labeled according to the criteria above, and the results were extrapolated to the whole set. Figure 2.17 shows the results of the analysis. 9.3 million, or 82%, of the tuples contain well-formed relations. Of these, 84% (7.8 million tuples) also contain well-formed arguments. Overall, 80.4% of the well-formed tuples are correct. A large fraction (87%) of the well-formed tuples are abstract, with the remaining 1 million tuples being concrete. Concrete tuples have a higher accuracy on average than abstract tuples: 88.1% against 79.2%.

These results indicate that TextRunner is extracting a large quantity of high-quality extractions, both abstract and concrete. Some applications, such as ontology learning and pattern mining, might benefit more from the abstract facts. Other applications, such as Question-Answering and Search will probably benefit more from the concrete facts. And of course applications will likely limit the set of relations to the domain of interest. TextRunner extracts everything available to it in the corpus, and allows the application to decide which subset to examine.

2.2.10 Previous Work in Large Scale and Open Information Extraction

The bulk of previous information extraction work uses hand-labeled data or hand-crafted patterns to enable relation-specific extraction from small, homogeneous corpora, as explained above. Large-scale extraction is beginning to gain some interest in recent work. [88]

One recent system eliminates some of the problems with the domain-dependence of information extraction systems, but falls short of open information extraction. Sekine [103] proposes a paradigm for "on-demand" information extraction, which aims to eliminate customization involved with adapting IE systems to new topics. Using unsupervised learning methods, the system automatically creates patterns and performs extraction based on a topic that has been specified by a user.

Shinyama and Sekine [104] describe an approach to "unrestricted relation discovery" that does away with many of the requirements for human input. However, it requires

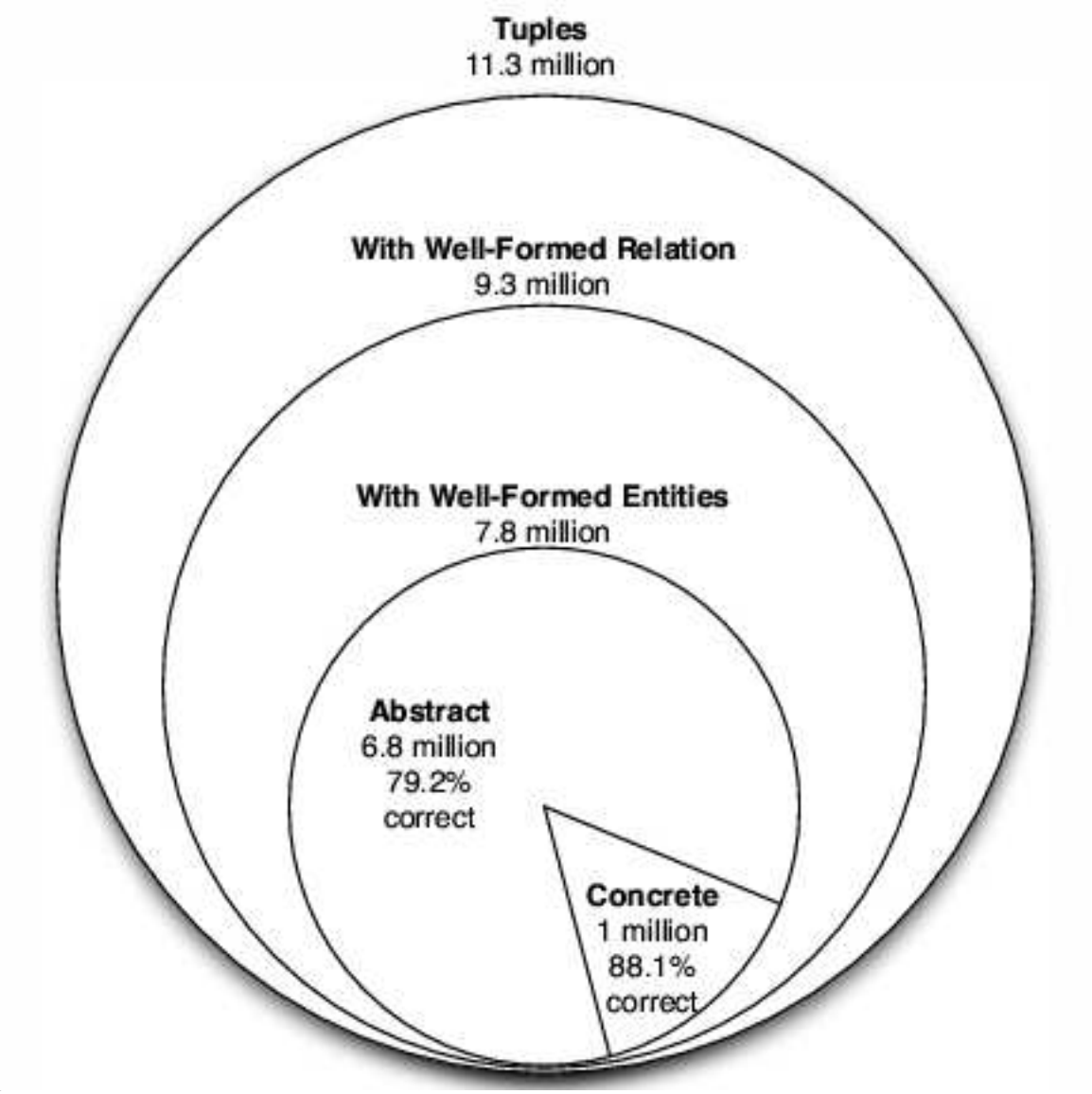


Figure 2.17: **Analysis of 11.3 million high-quality extractions from TextRunner. Overall, 80.4% of the 7.8 million well-formed tuples are correct. 82% of all 11.3 million tuples contain well-formed relations, and 84% of those contain well-formed arguments as well.**

clustering of the documents used for extraction, and thus scales in quadratic time in the number of documents, as compared with TextRunner's running time of $O(D + T \log T)$. It does not scale to the size of the Web. Furthermore, their use of heavyweight language processing tools, such as parsers and coreference resolution systems, may cause problems with scalability. In contrast, TextRunner uses lightweight tagging and chunking.

2.2.11 Conclusions

Taken together, KnowItAll and TextRunner provide a framework for information extraction at an unprecedented scale. But whereas KnowItAll is able to extract information at a Web scale for manually specified relations, TextRunner is able to extract far more information from the same amount of text at lower error rates. A large part of TextRunner's improvement is because it is an Open Information Extraction system, extracting relations on the fly as it extracts arguments. This ability enables it to extract information for far more relations than KnowItAll, even while it is extracting almost the same amount of information per relation.

These two systems demonstrate novel techniques and ideas for large-scale information extraction. The next two chapters show what can be done with the extracted knowledge.

Chapter 3

USING EXTRACTED INFORMATION TO IMPROVE NATURAL LANGUAGE PARSING

3.1 Introduction

Semantic processing of text in applications such as question answering or information extraction frequently relies on statistical parsers. Unfortunately, the efficacy of state-of-the-art parsers can be disappointingly low. For example, we found that the Collins parser correctly parsed just 54% of the list and factoid questions from TREC 2004 (that is, 54% of the parses had 100% precision and 100% recall on labeled constituents). Similarly, this parser produced 45% correct parses on a subset of 100 sentences from section 23 of the Penn Treebank.

Although statistical parsers continue to improve their efficacy over time, progress is slow, particularly for Web applications where training the parsers on a "representative" corpus of hand-tagged sentences is not an option. Because of the heterogeneous nature of text on the Web, such a corpus would be exceedingly difficult to generate.

In response, this paper investigates the possibility of detecting parser errors by using semantic information obtained from the Web. Our fundamental hypothesis is that *incorrect parses often result in wildly implausible semantic interpretations of sentences, which can be detected automatically in certain circumstances.* Consider, for example, the following sentence from the Wall Street Journal: "That compares with per-share earnings from continuing operations of 69 cents." The Collins parser yields a parse that attaches "of 69 cents" to "operations," rather than "earnings." By computing the mutual information between "operations" and "cents" on the Web, we can detect that this attachment is unlikely to be correct.

Our WOODWARD system detects parser errors as follows. First, it maps the tree produced by a parser to a *relational conjunction* (RC), a logic-based representation language that we

describe in Section 3.3. Second, WOODWARD employs four distinct methods for analyzing whether a conjunct in the RC is likely to be "reasonable" as described in Section 3.4.

Our approach makes several assumptions. First, if the sentence is absurd to begin with, then a correct parse could be deemed incorrect. Second, we require a corpus whose content overlaps at least in part with the content of the sentences to be parsed. Otherwise, much of our semantic analysis is impossible.

In applications such as Web-based question answering, these assumptions are quite natural. The questions are *about* topics that are covered extensively on the Web, and we can assume that most questions link verbs to nouns in reasonable combinations. Likewise, when using parsing for information extraction, we would expect our assumptions to hold as well.

Our contributions are as follows:

1. We introduce *Web-based semantic filtering*—a novel, domain-independent method for detecting and discarding incorrect parses.

2. We describe four techniques for analyzing relational conjuncts using semantic information obtained from the Web, and assess their efficacy both separately and in combination.

3. We find that WOODWARD can filter good parses from bad on TREC 2004 questions for a reduction of 67% in error rate. On a harder set of sentences from the Penn Treebank, the reduction in error rate is 20%.

The remainder of this chapter is organized as follows. We give an overview of related work in Section 3.2. Section 3.3 describes the semantic interpreter, including our RC representation. Section 3.4 illustrates the four Web-based classifiers that constitute the WOODWARD system. Section 3.5 presents our experiments and results, and section 3.6 concludes and gives ideas for future work.

3.2 Related Work

The problem of detecting parse errors is most similar to the idea of parse reranking. Collins [26] describes statistical techniques for reranking alternative parses for a sentence. Implicitly,

a reranking method detects parser errors, in that if the reranking method picks a new parse over the original one, it is classifying the original one as less likely to be correct. Collins uses syntactic and lexical features and trains on the Penn Treebank; in contrast, WOODWARD uses semantic features derived from the web. See section 3.5 for a comparison of our results with Collins'.

Several systems produce a semantic interpretation of a sentence on top of a parser. For example, Bos et al. [12] build semantic representations from the parse derivations of a CCG parser, and the English Resource Grammar (ERG) [112] provides a semantic representation using minimal recursion semantics. Toutanova et al. also include semantic features in their parse selection mechanism, although it is mostly syntax-driven. The ERG is a hand-built grammar and thus does not have the same coverage as the grammar we use. We also use the semantic interpretations in a novel way, checking them against semantic information on the Web to decide if they are plausible.

NLP literature is replete with examples of systems that produce semantic interpretations and use semantics to improve understanding. Several systems in the 1970s and 1980s used hand-built augmented transition networks or semantic networks to prune bad semantic interpretations. More recently, people have tried incorporating large lexical and semantic resources like WordNet, FrameNet, and PropBank into the disambiguation process. Allen [5] provides an overview of some of this work and contains many references. Our work focuses on using statistical techniques over large corpora, reducing the need for hand-built resources and making the system more robust to changes in domain.

Numerous systems, including Question-Answering systems like MULDER [61], PiQASso [6], and Moldovan et al.'s QA system [81], use parsing technology as a key component in their analysis of sentences. In part to overcome incorrect parses, Moldovan et al.'s QA system requires a complex set of relaxation techniques. These systems would greatly benefit from knowing when parses are correct or incorrect. Our system is the first to suggest using the output of a QA system to classify the input parse as good or bad.

Several researchers have used pointwise mutual information (PMI) over the Web to help make syntactic and semantic judgments in NLP tasks. Volk [119] uses PMI to resolve preposition attachments in German. Lapata and Keller [63] use web counts to resolve

preposition attachments, compound noun interpretation, and noun countability detection, among other things. And Markert et al. [73] use PMI to resolve certain types of anaphora. We use PMI as just one of several techniques for acquiring information from the Web.

3.3 Semantic Interpreter

The semantic interpreter aims to make explicit the relations that a sentence introduces, and the arguments to each of those relations. More specifically, the interpreter identifies the main verb relations, preposition relations, and semantic type relations in a sentence; identifies the number of arguments to each relation; and ensures that for every argument that two relations share in the sentence, they share a variable in the logical representation. Given a sentence and a Penn-Treebank-style parse of that sentence, the interpreter outputs a conjunction of First-Order Logic predicates. We call this representation a *relational conjunction* (RC). Each relation in an RC consists of a relation name and a tuple of variables and string constants representing the arguments of the relation. As an example, Figure 3.1 contains a sentence taken from the TREC 2003 corpus, parsed by the Collins parser. The parse for this sentence is incorrect: the parser treats `producing` as the main verb, with `are` as an auxiliary, whereas `are` is actually the main verb and `producing` is part of a phrase describing *states*. Figure 3.2 shows the correct RC for this sentence and the RC derived automatically from the incorrect parse.

3.3.1 Algorithm

The basic algorithm for converting a parse to a semantic representation consists of a recursive traversal through the parse tree. At each parse node, the algorithm applies a test to see if the node is a relation. If the parse constituent passes the test, the algorithm looks for a name of the relation, identifies all parse constituents that constitute arguments to the relation, builds a predicate from these pieces, and adds the new predicate to its stack. We now describe each of these operations in more detail.

The relation test: In principle, every clause, prepositional phrase, and base noun phrase will be a relation, but the Treebank-style annotation makes this test more complicated. For clauses, we say that any `SBAR` or `SBARQ` nodes are relations, as are `S`, `SQ`, or

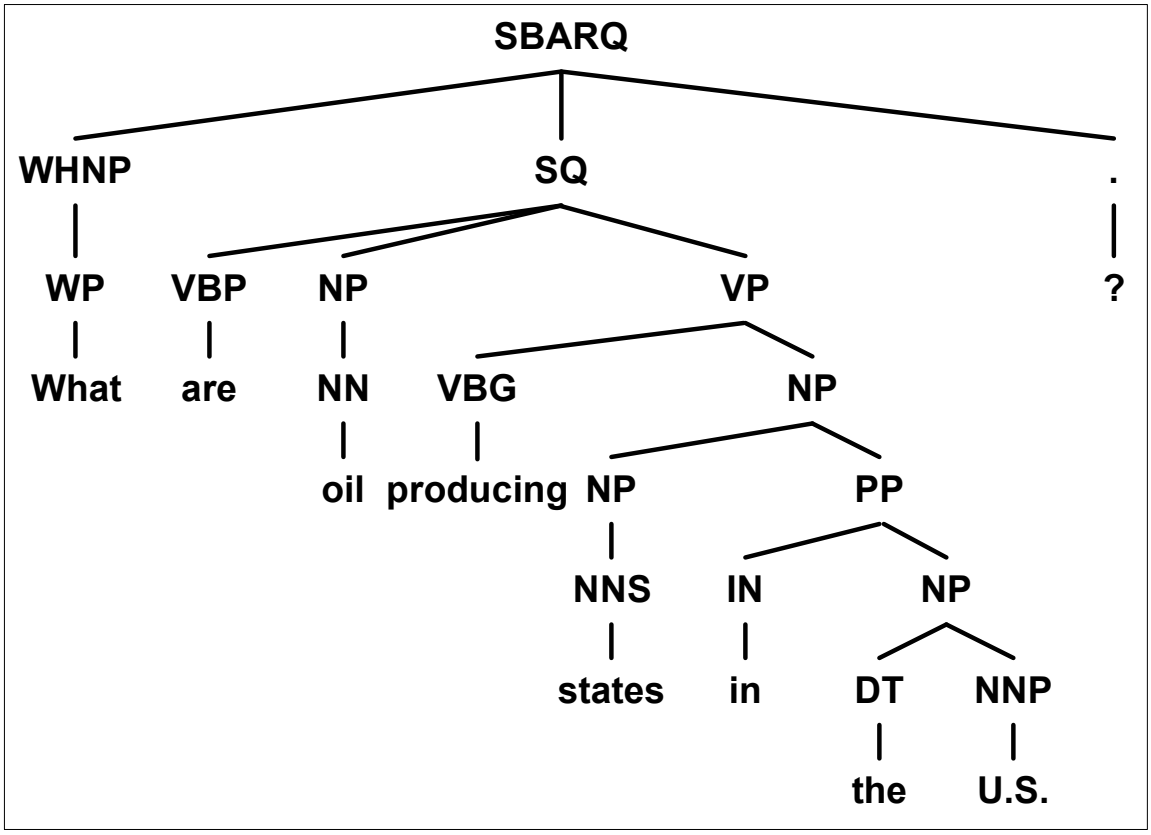


Figure 3.1: **An *incorrect* Collins Parse of a TREC question. The parser treats `producing` as the main verb in the clause, rather than `are`.**

1. What(NP1) ∧ **are(VP1, NP1, NP2)** ∧ states(NP2) ∧ **producing(VP2, NP2, NP4)** ∧ oil(NP4) ∧ in(PP1, NP2, U.S.)
2. What(NP1) ∧ states(NP2) ∧ **producing(VP1, NP2, NP1, NP4)** ∧ oil(NP4) ∧ in(PP1, NP2, U.S.)

Figure 3.2: **Example relational conjunctions. The first RC is the correct one for the sentence "What are oil producing states in the U.S.?" The second is the RC derived from the Collins parse in Figure 3.1. Differences between the two RCs appear in bold.**

SINV nodes that do not have a SBAR or SBARQ as parent. To catch participle phrases, VP nodes that have parent nodes which are not VP or any of the five S-type nodes are treated as relations. All prepositional phrases are treated as relations.

For noun phrases, we distinguish between *base noun phrases* — those that have part of speech-level head constituents — and *general noun phrases*. We treat base noun phrases as relations, but not general noun phrases.

In the example parse given in Figure 3.1, six nodes would be identified as relation nodes: SBARQ, WHNP, PP, and three of the four NP nodes. The NP node spanning `states in the U.S.` would not be considered a relation, as it is not a base noun phrase.

Finding a relation name: For prepositional phrases, we take the head word of the phrase. For noun phrases, we take the sequence of all words whose parents are immediate children of the base noun phrase, except for determiners, to be the name of the relation. For verb relations, the algorithm tries to identify the main verb phrase of the clause. That is, it searches for the verb phrase below the clause that does not contain any VP children. If such a node exists, the head of that phrase is taken to be the relation name. If it does not exist (as in, for example, the clause `himself intelligent` in the sentence `He considers himself intelligent`), the algorithm sets the relation name to be the verb `be`.

Returning to our example, the relation name for SBARQ would be identified as `producing`; for the noun phrases they would be `What`, `oil`, `states`, and `U.S.`, respectively; and for the PP, it would be `in`.

Finding arguments to a relation: In most cases, arguments to a prepositional phrase are easily identified as the parent constituent and the constituent following the preposition. However, when there are movement phenomena, which are naturally prevalent in the TREC data set, the preposition object needs to be recovered. As a simple heuristic, our algorithm takes the parent of the first SBAR or SBARQ above the PP to be the moved object.

Clause relations may take any number of arguments. Our algorithm identifies any child of a VP or S constituent inside the clause, except for auxiliary verbs and prepositions, as an argument. In addition, an SBAR clause whose parent is not a VP or another clause will take the parent as an argument as well, unless the parent is used for a gapped preposition. In our example, the SBARQ clause has argument constituents WHNP, the NP headed by `oil`, and

the NP spanning `states in the U.S.`

Base noun phrases take only one argument, which is the smallest noun phrase containing the base noun phrase whose parent is not a noun phrase. In our example, the NP spanning `states in the U.S.` corresponds to the argument to `states`, which conveniently helps the algorithm identify that this is the same argument to `producing` and `in`.

Putting the pieces together: The above three pieces constitute the major part of the interpreter, although there are a number of special cases to handle things like possessive noun phrases, proper nouns, adjective phrases, etc. The system can construct a predicate from the relation name and unique identifiers for the constituents that represent its arguments.

3.4 Semantic Filtering

This section describes semantic filtering as implemented in the WOODWARD system. Given the RC representation of a parsed sentence as supplied by the Semantic Interpreter, we test the parse using four web-based methods. Fundamentally, the methods all share the underlying principle that some form of co-occurrence of terms in the vast Web corpus can help decide whether a proposed relationship is semantically plausible.

Traditional statistical parsers also use co-occurrence of lexical heads as features for making parse decisions. We expand on this idea in two ways: first, we use a corpus several orders of magnitude larger than the tagged corpora traditionally used to train statistical parses, so that the fundamental problem of data sparseness is ameliorated. Second, we search for targeted patterns of words to help judge specific properties, like the number of complements to a verb. We now describe each of our techniques in more detail.

3.4.1 A PMI-Based Filter

A number of authors have demonstrated important ways in which search engines can be used to uncover semantic relationships, especially Turney's notion of *pointwise mutual information* (PMI) based on search-engine hits counts [114]. WOODWARD's PMI-Based Filter (PBF) uses PMI scores as features in a learned filter for predicates. Following Turney, we

use the formula below for the PMI between two terms $t1$ and $t2$:

$$PMI(t1, t2) = log\left(\frac{P(t1 \wedge t2)}{P(t1)P(t2)}\right) \tag{3.1}$$

We use PMI scores to judge the semantic plausibility of an RC conjunct as follows. We construct a number of different phrases, which we call *discriminator phrases*, from the name of the relation and the head words of each argument. For example, the prepositional attachment "operations of 65 cents" would yield phrases like "operations of" and "operations of * cents". (The '*' character is a wildcard in the Google interface; it can match any single word.) We then collect hitcounts for each discriminator phrase, as well as for the relation name and each argument head word, and compute a PMI score for each phrase, using the phrase's hitcount as the numerator in Equation 3.1. Given a set of such PMI scores for a single relation, we apply a learned classifier to decide if the PMI scores justify calling the relation implausible.

This classifier (as well as all of our other ones) is trained on a set of sentences from TREC and the Penn Treebank; our training and test sets are described in more detail in Section 3.5. We parsed each sentence automatically using Daniel Bikel's implementation of the Collins parsing model,[1] trained on sections 2–21 of the Penn Treebank, and then applied our semantic interpreter algorithm to come up with a set of relations. We labeled each relation by hand for correctness. Correct relations are positive examples for our classifier, incorrect relations are negative examples (and likewise for all of our other classifiers). We used the LIBSVM software package[2] to learn a Gaussian-kernel support vector machine model from the PMI scores collected for these relations. We can then use the classifier to predict if a relation is correct or not depending on the various PMI scores we have collected.

Because we require different discriminator phrases for preposition relations and verb relations, we actually learn two different models. After extensive experimentation, we used two patterns for verbs, and six for prepositions. Table 3.1 lists the patterns used. We use the PMI scores from the argument whose PMI values add up to the lowest value as the

[1] http://www.cis.upenn.edu/∼dbikel/software.html

[2] http://www.csie.ntu.edu.tw/∼cjlin/libsvm/.

features for a verb relation, with the intuition being that the relation is correct only if every argument to it is valid.

Table 3.1: **Patterns used by the PMI Filter. Each pattern is instantiated with verb, noun, or preposition head words, and verbs may be conjugated to fit the corresponding noun. The last two preposition patterns are targeted toward verb attachments. The "*" character is a wildcard for the Google search engine; it matches any single word, or possible a short sequence of words.**

Verb Patterns	Preposition Patterns
noun verb	*arg*1 *prep*
(or *verb noun* for non-subjects)	*arg*1 *prep* * *arg*2
noun * *verb*	*arg*1 *prep* the *arg*2
(or *verb* * *noun* for non-subjects)	*arg*1 * *arg*2
	*arg*1 it *prep arg*2
	*arg*1 them *prep arg*2

3.4.2 The Verb Arity Sampling Test

In our training set from the Penn Treebank, 13% of the time the Collins parser chooses too many or too few arguments to a verb. In this case, checking the PMI between the verb and each argument independently is insufficient, and there is not enough data to find hitcounts for the verb and all of its arguments at once. We therefore use a different type of filter in order to detect these errors, which we call the Verb Arity Sampling Test (VAST).

Instead of testing a verb to see if it can take a *particular* argument, we test if it can take a certain *number* of arguments. The verb predicate *producing(VP1, NP1, NP2, NP3)* in interpretation 2 of figure 3.2, for example, has too many arguments. To check if this predicate can actually take three noun phrase arguments, we first construct an extraction pattern with the following properties:

1. The pattern contains the verb and two noun phrases.

2. The noun phrases are likely to be arguments to the verb in any sentence matching the pattern.

3. The pattern is likely to be followed by a noun phrase for verbs that are capable of taking three arguments.

4. The pattern is unlikely to be followed by a noun phrase for verbs that cannot take three arguments.

5. The pattern is likely to be relatively common, and match a large number of sentences on the Web.

One example of a pattern that fits these criteria is "which it *verb*" (*e.g.*, "which it is producing"). Note that "which" and "it" are very common words, and in this phrase they are very likely to form arguments to the verb.

Next we take a sample of sentences containing this phrase from the Web. Since the words "which" and "it" are so common, for most verbs there will be thousands of matching sentences on the Web. The last step in VAST is to find out how many times in the sample sentences the verb "producing" is followed by a noun phrase. For verbs like "producing", there will be very few sentences that produce an extraction (mostly temporal noun phrases like "next week", or else extraction errors). But for verbs like "give" or "name", which can in fact accept three noun phrase arguments, there will be significantly more extracted noun phrases (*e.g.*, "which it gives them" occurs roughly 46,000 times on the Web, whereas "which it is producing them" does not occur at all.).

We now provide a more detailed description of the algorithm. For a given verb phrase, VAST counts the number of noun phrase arguments. The Collins parser also marks clause arguments as being essential by annotating them differently. VAST counts these as well, and considers the sum of the noun and clause arguments as the number of *essential arguments*. If the verb is passive and the number of essential arguments is one, or if the verb is active and the number of essential arguments is two, VAST performs no check. We call these *strictly transitive* verb relations. If the verb is passive and there are two essential arguments, or

if the verb is active and there are three, it performs the ditransitive check below. If the verb is active and there is one essential argument, it does the intransitive check described below. We call these two cases collectively *nontransitive* verb relations. In both cases, the checks produce a single real-valued score, and we use a linear kernel SVM to identify an appropriate threshold such that predicates above the threshold have the correct arity.

The ditransitive check begins by querying Google for two hundred documents containing the phrase "which it *verb*" or "which they *verb*". It downloads each document and identifies the sentences containing the phrase. It then POS-tags and NP-chunks the sentences using a maximum entropy tagger and chunker. It filters out any sentences for which the word "which" in the pattern is preceded by a preposition. Finally, if there are enough sentences remaining (more than ten), it counts the number of sentences in which the verb is directly followed by a noun phrase chunk. It then calculates the ditransitive score for verb v as the ratio of the number of extractions E to the number of filtered sentences F:

$$ditransitiveScore(v) = \frac{E}{F} \tag{3.2}$$

The intransitive check performs a very similar set of operations. It fetches up to two hundred sentences matching the phrases "but it *verb*" or "but they *verb*", tags and chunks them, and extracts noun phrases that directly follow the verb. It calculates the intransitive score for verb v using the number of extractions E and sentences S as:

$$intransitiveScore(v) = 1 - \frac{E}{S} \tag{3.3}$$

3.4.3 TextRunner *Filter*

TextRunner, an information extraction system described in Section 2.2, embodies a trade-off with the PMI method for checking the validity of a relation. [8] Its structure provides a much more natural search for the purpose of verifying a semantic relationship, since it has already arranged Web text into predicates and arguments. It is also much faster than querying a search engine like Google, both because we have local access to it and because commercial search engines tightly limit the number of queries an application may issue per day. On the other hand, the TextRunner index is at present still about two orders of magnitude smaller than Google's search index.

The TextRunner semantic filter checks the validity of an RC conjunct in a natural way: it asks TextRunner for the number of tuples that match the argument heads and relation name of the conjunct being checked. Since TextRunner predicates only have two arguments, we break the conjunct into trigrams and bigrams of head words, and average over the hitcounts for each. For predicate $P(A_1, \ldots, A_n)$ with $n \geq 2$, the score becomes

$$\begin{aligned} &TextRunnerScore = \\ &\quad \frac{1}{n-1} \sum_{i=2}^{n} hits(A_1, P, A_i) \\ &\quad + \frac{1}{n}(hits(A_1, P,) + \sum_{i=2}^{n} hits(, P, A_i)) \end{aligned}$$

As with PBF, we learn a threshold for good predicates using the LIBSVM package.

3.4.4 Question Answering Filter

When parsing questions, an additional method of detecting incorrect parses becomes available: use a question answering (QA) system to find answers. If a QA system using the parse can find an answer to the question, then the question was probably parsed correctly.

To test this theory, we implemented a lightweight, simple, and fast QA system that directly mirrors the semantic interpretation. It relies on TextRunner and KnowItNow [17] to quickly find possible answers, given the *relational conjunction* (RC) of the question. KnowItNow is a state of the art Information Extraction system that uses a set of domain independent patterns to efficiently find hyponyms of a class.

We formalize the process as follows: define a question as a set of variables X_i corresponding to noun phrases, a set of noun type predicates $T_i(X_i)$, and a set of relational predicates $P_i(X_{i1}, ..., X_{ik})$ which relate one or more variables and constants. The conjunction of type and relational predicates is precisely the RC.

We define an answer as a set of values for each variable that satisfies all types and predicates

$$ans(x_1, ..., x_n) = \bigwedge_i T_i(x_i) \wedge \bigwedge_j P_j(x_{j1}, ..., x_{jk})$$

The algorithm is as follows:

1. Compute the RC of the question sentence.

2. $\forall i$ find instances of the class T_i for possible values for X_i, using KnowItNow.

3. $\forall j$ find instances of the relation predicate $P_j(x_{j1}, ..., x_{jk})$. We use TextRunner to efficiently find objects that are related by the predicate P_j.

4. Return all tuples that satisfy $ans(x_1, ..., x_n)$

The QA semantic filter runs the Question Answering algorithm described above. If the number of returned answers is above a threshold (1 in our case), it indicates the question has been parsed correctly. Otherwise, it indicates an incorrect parse. This differs from the TextRunner semantic filter in that it tries to find subclasses and instances, rather than just argument heads. It makes a similar efficiency trade-off when compared to PMI: these checks are much faster than repeated queries to Google, but it is more likely to have problems with sparse data.

As an example, consider the question "What buildings has Frank Gehry designed?" Interpreting the question gives type predicates $Frank\_Gehry(X)$, $building(Y)$ and relation predicate $designed(X, Y)$. The system first tries to find instances of the types by querying KnowItNow. KnowItNow searches for patters like "buildings such as NP", and "NP and other Frank Gehries", since it does not know if a type predicate corresponds to a general class, or to a named instance. KnowItNow will find many subclasses and instances of buildings (e.g. 'pavilions', 'the Wright Guggenheim Museum', etc.) but, unsurprisingly, no instances of Frank Gehry. After stemming the heads, 'pavilion', 'museum', etc. are possible values for Y, and the only possible value for X is 'Gehry'. The system then finds as many instances of the $designed(X, Y)$ predicate as possible. This returns, among other tuples, `(MIT's Artificial Intelligence Lab, designed, Roomba)` and `(Gehry, designed, Pritzker Pavilion)`. After stemming, we look for answers common to all predicates. This results in *(X,Y)* tuples such as `(Gehry, pavilion)` and `(Gehry, museum)`. They clearly are not precise answers or sufficient for a real-world QA system, but they do indicate that the question might be answerable as parsed, and thus probably parsed correctly.

Table 3.2: **Summary of the five filters.**

Filter	**Method**
PBF	Checks PMI between relation and argument
VAST	Checks nontrans. verb relations for arity
TextRunner	Checks for known examples of the relation extracted from the Web
QA	Checks if answers to question exist on Web
Woodward	1. For questions, first runs the QA filter 2. For preposition relations, runs PBF 3. For nontransitive verbs, runs VAST 4. For transitive verbs, runs TextRunner

3.4.5 The Woodward *Filter*

Each of the above semantic filters has its strengths and weaknesses. On our training data, TextRunner had the most success of any of the methods on classifying verb relations that did not have arity errors. Because of sparse data problems, however, it was less successful than PMI on preposition relations. The QA system had the interesting property that when it predicted an interpretation was correct, it was always right; however, when it made a negative prediction, its results were mixed.

Woodward combines the four semantic filters in a way that draws on each of their strengths. First, it checks if the sentence is a question that does not contain prepositions. If so, it runs the QA module, and returns true if that module does.

After trying the QA module, Woodward checks each predicate in turn. If the predicate is a preposition relation, it uses PBF to classify it. For nontransitive verb relations, it uses VAST. For strictly transitive verb relations, it uses TextRunner. Woodward accepts the RC if every relation is predicted to be correct; otherwise, it rejects it. Table 3.2 summarizes all of our semantic filters.

3.5 Experiments

In our experiments we tested the ability of WOODWARD to detect bad parses. Our experiments proceeded as follows: we parsed a set of sentences, ran the semantic interpreter on them, and labeled each parse and each relation in the resulting RCs for correctness. We then extracted all of the necessary information from the Web and TEXTRUNNER. We divided the sentences into a training and test set, and trained the filters on the labeled RCs from the training sentences. Finally, we ran each of the filters and WOODWARD on the test set to predict which parses were correct. We report the results below, but first we describe our datasets and tools in more detail.

3.5.1 Datasets and Tools

Because question-answering is a key application, we began with data from the TREC question-answering track. We split the data into a training set of 61 questions (all of the TREC 2002 and TREC 2003 questions), and a test set of 55 questions (all list and factoid questions from TREC 2004). We preprocessed the questions to remove parentheticals (this affected 3 training questions and 1 test question). We removed 12 test questions because the Collins parser did not parse them as questions,[3] and that error was too easy to detect. 25 training questions had the same error, but we left them in to provide more training data.

We used the Penn Treebank as our second data set. Training sentences were taken from section 22, and test sentences from section 23. Because PBF is time-consuming, we took a subset of 100 sentences from each section to expedite our experiments. We extracted from each section the first 100 sentences that did not contain conjunctions, and for which all of the errors, if any, were contained in preposition and verb relations.

For our parser, we used Bikel's implementation of the Collins parsing model, trained on sections 2-21 of the Penn Treebank. We only use the top-ranked parse for each sentence. For the TREC data only, we first POS-tagged each question using Ratnaparkhi's MXPOST tagger. [95] We judged each of the TREC parses manually for correctness, but scored the Treebank parses automatically.

[3]That is, the root node was neither SBARQ nor SQ.

Table 3.3: **Accuracy of the filters on three relation types in the TREC 2004 questions and WSJ data: nontransitive verb relations, transitive verb relations, and preposition relations.**

Relation Type	num. correct	num. incorrect	PBF acc.	VAST acc.	TextRunner acc.
Nontrans. Verb	41	35	0.54	0.66	0.52
Other Verb	126	68	0.72	N/A	0.73
Preposition	183	58	0.73	N/A	0.76

3.5.2 Results and Discussion

Our semantic interpreter was able to produce the appropriate RC for every parsed sentence in our data sets, except for a few minor cases. Two idiomatic expressions in the WSJ caused the semantic interpreter to find noun phrases outside of a clause to fill gaps that were not actually there. And in several sentences with infinitive phrases, the semantic interpreter did not find the extracted subject of the infinitive expression. It turned out that none of these mistakes caused the filters to reject correct parses, so we were satisfied that our results mainly reflect the performance of the filters, rather than the interpreter.

In Table 3.3 we report the accuracy of our first three filters on the task of predicting whether a relation in an RC is correct. We break these results down into three categories for the three types of relations we built filters for: strictly transitive verb relations, nontransitive verb relations, and preposition relations. Since the QA filter works at the level of an entire RC, rather than a single relation, it does not apply here. These results show that the trends on the training data mostly held true: VAST was quite effective at verb arity errors, and TextRunner narrowly beat PBF on the remaining verb errors. However, on our training data PBF narrowly beat TextRunner on preposition errors, and the reverse was true on our test data.

Our QA filter predicts whether a full parse is correct with an accuracy of 0.76 on the 17 TREC 2004 questions that had no prepositions. The Collins parser achieves the same level of accuracy on these sentences, so the main benefit of the QA filter for Woodward is that it never misclassifies an incorrect parse as a correct one, as was observed on the training

set. This property allows WOODWARD to correctly predict a parse is correct whenever it passes the QA filter.

Classification accuracy is important for good performance, and we report it to show how effective each of WOODWARD's components is. However, it fails to capture the whole story of a filter's performance. Consider a filter that simply predicts that every sentence is incorrectly parsed: it would have an overall accuracy of 55% on our WSJ corpus, not too much worse than WOODWARD's classification accuracy of 66% on this data. [4] However, such a filter would be useless because it filters out every correctly parsed sentence.

Let the *filtered set* be the set of sentences that a filter predicts to be correctly parsed. The performance of a filter is better captured by two quantities related to the filtered set: first, how "pure" the filtered set is, or how many good parses it contains compared to bad parses; and second, how wasteful the filter is in terms of losing good parses from the original set. We measure these two quantities using metrics we call *filter precision* and *filter recall*. Filter precision is defined as the ratio of correctly parsed sentences in the filtered set to total sentences in the filtered set. Filter recall is defined as the ratio of correctly parsed sentences in the filtered set to correctly parsed sentences in the unfiltered set. Note that these metrics are quite different from the labeled constituent precision/recall metrics that are typically used to measure statistical parser performance. Filter precision and recall depend on the number of correctly parsed sentences in a set of sentences, whereas constituent precision and recall depend on the number of correct constituents in the parse of a single sentence.

Table 3.4 shows our overall results for filtering parses using WOODWARD. We compare against a baseline model that predicts every sentence is parsed correctly. WOODWARD outperforms this baseline in precision and F1 measure on both of our data sets.

Collins [26] reports a decrease in error rate of 13% over his original parsing model (the same model as used in our experiments) by performing a discriminative reranking of parses. Our WSJ test set is a subset of the set of sentences used in Collins' experiments, so our results are not directly comparable, but we do achieve a roughly similar decrease in error

[4] 66% is the accuracy of classifying whether a full parse is correct according to the algorithm in section 3.4.5. This is not the accuracy of classifying individual relations, as shown in Table 3.3.

Table 3.4: **Performance of WOODWARD on different data sets. Parser efficacy reports the percentage of sentences that the Collins parser parsed correctly. See the text for a discussion of our baseline and the precision and recall metrics. We weight precision and recall equally in calculating F1. Reduction in error rate (red. err.) reports the relative decrease in error (error calculated as 1 − F1) over baseline.**

			Baseline			WOODWARD			
	sents.	parser eff.	filter prec.	filter rec.	F1	filter prec.	filter rec.	F1	red. err.
trec	43	54%	0.54	1.0	0.70	0.82	1.0	0.90	**67%**
wsj	100	45%	0.45	1.0	0.62	0.58	0.88	0.70	**20%**

rate (20%) when we use our filtered precision/recall metrics. We also measured the labeled constituent precision and recall of both the original test set and the filtered set, and found a decrease in error rate of 37% according to this metric (corresponding to a jump in F1 from 90.1 to 93.8). Note that in our case, the error is reduced by throwing out bad parses, rather than trying to fix them. The 17% difference between the two decreases in error rate is probably due to the fact that WOODWARD is more likely to detect the worse parses in the original set, which contribute a proportionally larger share of error in labeled constituent precision/recall in the original test set.

WOODWARD performs significantly better on the TREC questions than on the Penn Treebank data. One major reason is that there are far more clause adjuncts in the Treebank data, and adjunct errors are intrinsically harder to detect. Consider the Treebank sentence: "The S&P pit stayed locked at its 30-point trading limit as the Dow average ground to its final 190.58 point loss Friday." The parser incorrectly attaches the clause beginning "as the Dow . . . " to "locked", rather than to "stayed." Our current methods aim to use key words in the clause to determine if the attachment is correct. However, with such clauses there is no single key word that can allow us to make that determination. We anticipate that as the paradigm matures we and others will design filters that can use more of the information in the clause to help make these decisions.

3.6 Conclusions and Future Work

Given a parse of a sentence, WOODWARD constructs a representation that identifies the key semantic relationships implicit in the parse. It then utilizes a set of Web-based sampling techniques to check whether these relationships are plausible. If any of the relationships is highly implausible, WOODWARD concludes that the parse is incorrect. WOODWARD successfully detects common errors in the output of the Collins parser including verb arity errors as well as preposition and verb attachment errors. While more extensive experiments are clearly necessary, our results suggest that the paradigm of *Web-based semantic filtering* could substantially improve the performance of statistical parsers.

In future work, we hope to further validate this paradigm by constructing additional semantic filters that detect other types of errors. We also plan to use semantic filters such as WOODWARD to build a large-scale corpus of automatically-parsed sentences that has higher accuracy than can be achieved today. Such a corpus could be used to re-train a statistical parser to improve its performance. Beyond that, we plan to embed semantic filtering into the parser itself. If semantic filters become sufficiently accurate, they could rule out enough erroneous parses that the parser is left with just the correct one.

Chapter 4

SYNONYM RESOLUTION ON THE WEB

4.1 Introduction

Web Information Extraction (WIE) systems extract *assertions* that describe a relation and its arguments from Web text (e.g., `(is capital of, D.C., United States)`). WIE systems can extract hundreds of millions of assertions containing millions of different strings from the Web (*e.g.*, the TEXTRUNNER system [8]). WIE systems often extract assertions that describe the same real-world object or relation using different names. For example, a WIE system might extract `(is capital city of, Washington, U.S.)`, which describes the same relationship as above but contains a different name for the relation and each argument.

Synonyms are prevalent in text, and the Web corpus is no exception. Our data set of two million assertions extracted from a Web crawl contained over a half-dozen different names each for the `United States` and `Washington, D.C.`, and three for the `is capital of` relation. The top 80 most commonly extracted objects had an average of 2.9 extracted names per entity, and several had as many as 10 names. The top 100 most commonly extracted relations had an average of 4.9 synonyms per relation.

We refer to the problem of identifying synonymous object and relation names as Synonym Resolution (SR).[1] [122] Previous techniques for SR have focused on one particular aspect of the problem, either objects or relations. In addition, the techniques either depend on a large set of training examples, or are tailored to a specific domain by assuming knowledge of the domain's schema. Due to the number and diversity of the relations extracted, these techniques are not feasible for WIE systems. Schemata are not available for the Web, and hand-labeling training examples for each relation would require a prohibitive manual effort.

In response, we present RESOLVER, a novel, domain-independent, unsupervised synonym

[1]Ironically, SR has a number of synonyms in the literature, including Entity Resolution, Record Linkage, and Deduplication.

resolution system that applies to both objects and relations. RESOLVER clusters coreferential names together using a probabilistic model informed by string similarity and the similarity of the assertions containing the names. The key questions answered by RESOLVER include:

1. *Is it possible to effectively cluster strings into sets of synonyms using nothing more than a set of automatically-extracted facts about them?* Experiments below include an empirical demonstration that RESOLVER can resolve objects with 78% precision and 68% recall, and relations with 90% precision and 35% recall.

2. *Can synonym resolution be scaled to millions of distinct strings? to the size of the Web?* RESOLVER provides a scalable clustering algorithm that runs in time $O(KN\ log\ N)$ in the number of extractions N and maximum number of synonyms per word, K, without discarding any potentially matching pair, under exceptionally weak assumptions about the data.

3. *Can unsupervised synonym resolution be formalized in a probabilistic model, and is there a practical benefit to doing so?* RESOLVER provides an unsupervised, generative probabilistic model for predicting whether two object or relation names co-refer, and experiments show that this significantly outperforms previous metrics for distributional similarity.

4. Certain relations, especially functions and inverse functions, provide especially strong evidence for and against synonymy. *Is it possible to use the special properties of functions and inverse functions to improve the precision of a synonym resolution algorithm?* Several extensions to the RESOLVER system show that the precision of object merging can be improved by 3% using functions, and experiments on artificial data show that with more suitable data, the increase could be as high as 26%.

5. RESOLVER is capable of determining both object and relation synonyms independently. It is natural to wonder if the process for merging objects could benefit from synonym relationships discovered between relations, and *vice versa*. Unfortunately, on

TEXTRUNNER data, this kind of mutual recursion between object and relation synonym resolution does not help. We therefore investigate: *under what conditions does* RESOLVER*'s synonym resolution improve when object and relation merging depend on one another?* Experiments on artificial data demonstrate what types of data sets are appropriate for mutual recursion between object and relation merging, and they show that mutual recursion could improve recall by as much as 16% (with a small benefit for precision) on suitable data.

The next section describes the problem of synonym resolution formally and introduces notation and terminology that will be used throughout. Section 4.3 discusses previous work in SR. Section 4.4 introduces RESOLVER's probabilistic model for SR. Section 4.5 describes RESOLVER's clustering algorithm. Section 4.6 presents experiments with the basic RESOLVER system that compare its performance with the performance of previous work in synonym resolution. Section 4.7 describes several extensions to the basic RESOLVER system, together with experiments illustrating the gains in precision and recall. Section 4.8 develops the clustering algorithm further, and demonstrates improvements through experiments. Finally, section 4.9 discusses conclusions and areas for future work.

4.2 Formal Synonym Resolution Problem

An SR system for WIE takes a set of extractions as input and returns a set of clusters, with each cluster containing coreferential object strings or relation strings. More precisely, the input is a data set D containing extracted *assertions* of the form $a = (r, o_1, \ldots, o_n)$, where r is a relation string and each o_i is an object string representing the arguments to the relation. Throughout this work, all assertions are assumed to be binary, so $n = 2$.

The output of an SR system is a *clustering*, or set of clusters, of the strings in D. Let S be the set of all distinct strings in D. A clustering of S is a set $C \subset 2^S$ such that all the clusters in C are distinct, and they cover the whole set:

$$\bigcup_{c \in C} = S$$

$$\forall c_1, c_2 \in C.\, c_1 \wedge c_2 = \emptyset$$

Each cluster in the output clustering constitutes the system's conjecture that all strings inside the cluster are synonyms, and no string outside that cluster is a synonym of any string in the cluster.

4.2.1 The Single-Sense Assumption

The formal representation of SR makes an important simplifying assumption: it is assumed that every string belongs to exactly one cluster. In language, however, strings often have multiple meanings; *i.e.,* they are *polysemous.* Polysemous strings cannot be adequately represented using a clustering in which each string belongs to exactly one cluster. Addressing this shortcoming of the representation language is an important challenge for future work.

As an example of the representational trouble posed by polysemy, consider the name "President Roosevelt." In certain contexts, this name is synonymous with "President Franklin D. Roosevelt," and in other contexts it is synonymous with "President Theodore Roosevelt." However, "President Franklin D. Roosevelt" is never synonymous with "President Theodore Roosevelt." There is no clustering of the three names, using the notion of clustering described above, such that all synonymy relationships are accurately represented.

Others have described alternate kinds of clustering that take polysemy into account. For example, "soft clustering" allows a string to be assigned to as many different clusters as it has senses. One variation on this idea is to assign a probability distribution to every string, describing the prior probability that the string belongs in each cluster [67, 91]. Both of these representations capture only prior information about strings. That is, they represent the idea that a particular string can belong to a cluster, or the probability that it belongs to a cluster, but not whether a particular instance of the string actually does belong to a cluster. A third type of clustering, the most explicit representation, stores each instance of a string separately. Each string instance is assigned to the cluster that is most appropriate for the instance's context. Word sense disambiguation systems that assign senses from WordNet [79] implicitly use this kind of clustering (*e.g.*, [54, 106]).

Existing SR systems all make the single-sense assumption or very similar assumptions. A full treatment of the problem needs to address polysemy, and it remains an important

challenge for the field.

4.2.2 Subproblems in Synonym Resolution

The SR problem can be divided into two subproblems: first, how to measure the similarity, or probability of synonymy, between pairs of strings in S; and second, how to form clusters such that all of the elements in each cluster have high similarity to one another, and relatively low similarity to elements in other clusters.

RESOLVER uses a generative, probabilistic model for finding the similarity between strings. For strings s_i and s_j, let $R_{i,j}$ be the random variable for the event that s_i and s_j refer to the same entity. Let $R^t_{i,j}$ denote the event that $R_{i,j}$ is true, and $R^f_{i,j}$ denote the event that it is false. Let D_x denote the set of extractions in D which contain string x. Given D and S, the first subtask of SR is to find $P(R_{i,j}|D_{s_i}, D_{s_j})$ for all pairs s_i and s_j. The second subtask takes S and the probability scores for pairs of strings from S as input. Its output is a clustering of S.

Sections 4.4 and 4.5 cover RESOLVER's solutions to each subtask respectively. First, however, we discuss previous work related to Synonym Resolution.

4.3 Previous Work

The Discovery of Inference Rules from Text (DIRT) algorithm [68] addresses a piece of the unsupervised SR problem. DIRT is a heuristic method for finding synonymous relations, or "inference rules." DIRT uses a dependency parser and mutual information statistics over a corpus to identify relations that have similar sets of arguments. In contrast, our algorithm provides a formal probabilistic model that applies equally well to relations and objects. Sections 4.4.2 contains a fuller description of the differences between the two methods, and section 4.6 describes experiments which show RESOLVER's superior performance in precision and recall over DIRT.

RESOLVER's method of determining the similarity between two strings is an example of a broad class of metrics called *distributional similarity* metrics [66], but it has significant advantages over traditional distributional similarity metrics for the SR task. All of these metrics are based on the underlying assumption, called the Distributional Hypothesis, that

"Similar objects appear in similar contexts" [52]. Previous distributional similarity metrics, however, have been designed for comparing words based on terms appearing in the same document, rather than extracted properties. This has two important consequences: first, extracted properties are by nature sparser because they appear only in a narrow window around words and because they consist of longer strings (at the very least, pairs of words); second, each property that is shared by two strings is much more meaningful than if they simply appeared near a word in a document because the extraction mechanism is designed to find meaningful relationships. RESOLVER's metric is designed to take advantage of the relational model provided by Web Information Extraction. Section 4.4.2 describes the difference between the Cosine Similarity Metric, an example of a traditional distributional similarity metric, and RESOLVER's metric more fully, and experiments in Section 4.6 demonstrate that RESOLVER outperforms the Cosine Similarity Metric [101].

There are many unsupervised approaches for object resolution in databases, but unlike our algorithm these approaches depend on a known, fixed, and generally very small schema. Ravikumar and Cohen [97] present an unsupervised approach to object resolution using Expectation-Maximization on a hierarchical graphical model. Several other recent approaches leverage domain-specific information and heuristics for object resolution. For example, many [36, 9, 10] rely on evidence from observing which strings appear as arguments to the same relation simultaneously (*e.g.*, co-authors of the same publication). While this is useful information when resolving authors in the citation domain, it is extremely rare to find relations with similar properties in extracted assertions. None of these approaches applies to the problem of resolving relations. See [120] for a survey of this area.

One promising new approach to clustering in a relational domain is the Multiple Relational Clusterings (MRC) algorithm. [60] This approach, though not specific to synonym resolution, can find synonyms in a set of unlabeled, relational extractions without domain-specific heuristics. The approach is quite recent, and so far no detailed experimental comparison has been conducted. One preliminary experiment showed that MRC could attain a higher precision clustering than RESOLVER using only string similarity as its evidence, but MRC had lower recall than RESOLVER, so the experiment was inconclusive.

Several supervised learning techniques make entity resolution decisions [57, 78, 105],

but of course these systems depend on the availability of training data, and often on a significant number of labeled examples per relation of interest. These approaches also depend on complex probabilistic models and learning algorithms, and they currently do not scale to the amounts of data extracted from the Web. Previous systems are tested on at most around ten thousand examples, compared with millions or hundreds of millions of extractions from WIE systems such as TextRunner.

Coreference resolution systems (e.g., [64, 84]), like SR systems, try to merge references to the same object (typically pronouns, but potentially other types of noun phrases). This problem differs from the SR problem in several ways: first, it deals with unstructered text input, possibly with syntactic annotation, rather than relational input. Second, it deals only with resolving objects. Finally, it requires local decisions about strings; that is, the same word may appear twice in a text and refer to two different things, so each occurrence of a word must be treated separately.

Resolver and most SR systems built for databases tend to use relational features. Previous systems, including coreference resolution systems as well as named-entity resolution systems [72, 7], have also investigated bag-of-words models for synonym resolution. While bag-of-words models give a broader context and might perhaps be better suited for handling polysemy than relational models, each feature tends to provide less evidence for or against synonymy than in a relational model, where the features have been pre-selected by an information extraction system.

The PASCAL Recognising Textual Entailment Challenge proposes the task of recognizing when two sentences entail one another, and many authors have submitted responses to this challenge [33]. Synonym resolution is a subtask of this problem. Our task differs significantly from the textual entailment task in that it has no labeled training data, and its input is in the form of relational extractions rather than raw text.

Resolver's probabilistic model is partly inspired by the ball-and-urns abstraction of information extraction presented by Downey et al. [38] Resolver's task and probability model are different from theirs, but many of the same modeling assumptions (such as the independence of extractions) are made in both cases to simplify the derivation of the models.

4.4 Models for String Comparisons

Our probabilistic model provides a formal, rigorous method for resolving synonyms in the absence of training data. It has two sources of evidence: the similarity of the strings themselves (*i.e.*, edit distance) and the similarity of the assertions they appear in. This second source of evidence is sometimes referred to as *distributional similarity.* [52]

Section 4.4.1 presents a simple model for predicting whether a pair of strings co-refer based on string similarity. Section 4.4.2 then presents a model called the Extracted Shared Property (ESP) Model for predicting whether a pair of strings co-refer based on their distributional similarity. Finally, a method is presented for combining these models to come up with an overall prediction for coreference decisions between two clusters of strings.

4.4.1 String Similarity Model

Many objects appear with multiple names that are substrings, acronyms, abbreviations, or other simple variations of one another. Thus string similarity can be an important source of evidence for whether two strings co-refer. RESOLVER's probabilistic String Similarity Model (SSM) assumes a similarity function $\text{sim}(s_1, s_2)$: $STRING \times STRING \rightarrow [0, 1]$. The model sets the probability of s_1 co-referring with s_2 to a smoothed version of the similarity:

$$P(R^t_{i,j}|\text{sim}(s_1, s_2)) = \frac{\alpha * \text{sim}(s_1, s_2) + 1}{\alpha + \beta}$$

As α increases, the probability estimate transitions from $1/\beta$ to the value of the similarity function. The particular choice of α and β make little difference to RESOLVER's results, so long as they are chosen such that the resulting probability can never be one or zero. In the experiments below, $\alpha = 20$ and $\beta = 5$. The Monge-Elkan string similarity function [83] is used for objects, and the Levenshtein string edit-distance function is used for relations [25].

4.4.2 The Extracted Shared Property Model

The Extracted Shared Property Model (ESP) outputs the probability that two strings co-refer based on the similarity of the extracted assertions in which they appear. For example, if the extractions `(Newton, invented, calculus)` and `(Leibniz, invented, calculus)` both

appeared in the data, then `Newton` and `Leibniz` would be judged to have similar contexts in the extracted data.

More formally, let a pair of strings (r, s) be called a *property* of an object string o if there is an assertion $(r, o, s) \in D$ or $(r, s, o) \in D$. A pair of strings (s_1, s_2) is an *instance* of a relation string r if there is an assertion $(r, s_1, s_2) \in D$. Equivalently, the property $p = (r, s)$ *applies* to o, and the instance $i = (s_1, s_2)$ *belongs* to r. The ESP model outputs the probability that two strings co-refer based on how many properties (or instances) they share.

As an example, consider the strings `Mars` and `Red Planet`, which appear in our data 659 and 26 times respectively. Out of these extracted assertions, they share four properties. For example, `(lacks, Mars, ozone layer)` and `(lacks, Red Planet, ozone layer)` both appear as assertions in our data. The ESP model determines the probability that `Mars` and `Red Planet` refer to the same entity after observing k, the number of properties that apply to both, n_1, the total number of extracted properties for `Mars`, and n_2, the total number of extracted properties for `Red Planet`.

ESP models the extraction of assertions as a generative process, much like the URNS model [38]. For each string s_i, a certain number, P_i, of properties of the string are written on balls and placed in an urn. Extracting n_i assertions that contain s_i amounts to selecting a subset of size n_i from these labeled balls.[2] Properties in the urn are called *potential properties* to distinguish them from extracted properties.

To model coreference decisions, ESP uses a pair of urns, containing P_i and P_j balls respectively, for the two strings s_i and s_j. Some subset of the P_i balls have the exact same labels as an equal-sized subset of the P_j balls. Let the size of this subset be $S_{i,j}$. The ESP model assumes that coreferential strings share as many potential properties as possible, though only a few of the potential properties will be extracted for both. For non-coreferential strings, the number of shared potential properties is a strict subset of the potential properties of each string. Thus if $R_{i,j}$ is true then $S_{i,j} = \min(P_i, P_j)$, and if $R_{i,j}$

[2]Unlike the URNS model, balls are drawn without replacement. The TextRunner data contains only one mention of any extraction, so drawing without replacement tends to model the data more accurately.

is false then $S_{i,j} < \min(P_i, P_j)$.

The ESP model makes several simplifying assumptions in order to make probability predictions. As is suggested by the ball-and-urn abstraction, it assumes that each ball for a string is equally likely to be selected from its urn. Because of data sparsity, almost all properties are very rare, so it would be difficult to get a better estimate for the prior probability of selecting a particular potential property. Second, balls are drawn from one urn independent of draws from any other urn. And finally, it assumes that without knowing the value of k, every value of $S_{i,j}$ is equally likely, since we have no better information.

Given these assumptions, we can derive an expression for $P(R^t_{i,j})$. The derivation is sketched below; see Appendix A for a complete derivation. First, note that there are $\binom{P_i}{n_i}\binom{P_j}{n_j}$ total ways of extracting n_i and n_j assertions for s_i and s_j. Given a particular value of $S_{i,j}$, the number of ways in which n_i and n_j assertions can be extracted such that they share exactly k is given by

$$\text{Count}(k, n_i, n_j | P_i, P_j, S_{i,j}) = \binom{S_{i,j}}{k} \sum_{r,s \geq 0} \binom{S_{i,j}-k}{r+s} \binom{r+s}{r} \binom{P_i - S_{i,j}}{n_i - (k+r)} \binom{P_j - S_{i,j}}{n_j - (k+s)} \tag{4.1}$$

By our assumptions,

$$P(k | n_i, n_j, P_i, P_j, S_{i,j}) = \frac{\text{Count}(k, n_i, n_j | P_i, P_j, S_{i,j})}{\binom{P_i}{n_i}\binom{P_j}{n_j}} \tag{4.2}$$

Let $P_{\min} = \min(P_i, P_j)$. The result below follows from Bayes' Rule and our assumptions above:

Proposition 1 *If two strings s_i and s_j have P_i and P_j potential properties (or instances), and they appear in extracted assertions D_i and D_j such that $|D_i| = n_i$ and $|D_j| = n_j$, and they share k extracted properties (or instances), the probability that s_i and s_j co-refer is:*

$$P(R^t_{i,j} | D_i, D_j, P_i, P_j) = \frac{P(k | n_i, n_j, P_i, P_j, S_{i,j} = P_{\min})}{\sum_{k \leq S_{i,j} \leq P_{\min}} P(k | n_i, n_j, P_i, P_j, S_{i,j})} \tag{4.3}$$

Substituting equation 4.2 into equation 4.3 gives us a complete expression for the probability we are looking for.

Note that the probability for $R_{i,j}$ depends on two hidden parameters, P_i and P_j. Since in unsupervised SR there is no labeled data to estimate these parameters from, these parameters are tied to the number of times the respective strings s_i and s_j are extracted: $P_i = N \times n_i$. The experimental methods in Section 4.6 explains how the parameter N is set.

Appendix B illustrates a technique for calculating the ESP model efficiently.

4.4.3 Comparison of ESP with Other Distributional Similarity Metrics

The Discovery of Inference Rules from Text (DIRT) [68] method is the most similar previous work to the ESP model in its goals, but the metric itself is very different. DIRT operates over triples of extracted strings and produces similarity scores for relations, without requiring manually labeled training data, by comparing the distributions of one relation's arguments to another's. Unlike ESP, DIRT uses a dependency parser to extract its triples, but the similarity metric may be isolated from the extraction mechanism and compared with ESP separately.

The most significant difference between the DIRT similarity metric and the ESP model is that the DIRT metric compares the x arguments from one relation to the x arguments of the other, and then compares the y arguments from one relation to the y arguments of the other, and finally combines the scores. In contrast, ESP compares the (x, y) argument pairs of one relation to the (x, y) pairs of the other. While the DIRT metric has the advantage that it is more likely to find matches between two relations in sparse data, it has the disadvantage that the matches it does find are not necessarily strong evidence for synonymy. In effect, it is capturing the intuition that synonyms have the same argument types for their domains and ranges, but it is certainly possible for non-synonyms to have the same property. Antonyms are an obvious example. Synonyms are not defined by their domains and ranges, but rather by the mapping between them, and ESP better captures the similarity in this mapping, as demonstrated in experiments below (Section 4.6).

Most traditional distributional similarity metrics operate over context vectors, rather than extracted triples. The ESP model treats the extracted triples much like context vectors:

each string has a binary vector of properties, ones representing properties that apply to the string and zeros representing those that do not.

However, unlike traditional distributional similarity metrics, ESP does not weigh each dimension in the context vector independently. Cosine Similarity Metric (CSM), for example, determines how similar two context vectors are in each dimension, and then adds the scores up for all dimensions. In contrast, the score assigned to a dimension in ESP depends on how many other dimensions have matching contexts. As the number of matching contexts grows, the weight for each additional matching context also grows. The effect is that ESP has much lower similarity scores than CSM for small numbers of matching contexts, and much higher scores for larger numbers of matching contexts. Experiments in Section 4.6 demonstrate the improvement in performance of ESP over CSM.

CSM's similarity score for a particular dimension depends on how often the string is extracted with that context, and how often the context appears with other strings. The basic ESP model does not take those two pieces of evidence into account, although Section 4.7.3 describes an extension to the basic model that does.

4.4.4 Combining the Evidence

For each potential coreference relationship $R_{i,j}$, RESOLVER considers two pieces of probabilistic evidence. Let $E^e_{i,j}$ be the evidence for ESP, and let $E^s_{i,j}$ be the evidence for SSM. Our method for combining the two uses the Naïve Bayes assumption that each piece of evidence is conditionally independent, given the coreference relation:

$$P(E^s_{i,j}, E^e_{i,j}|R_{i,j}) = P(E^s_{i,j}|R_{i,j})P(E^e_{i,j}|R_{i,j}) \tag{4.4}$$

Given this simplifying assumption, we can combine the evidence to find the probability of a coference relationship by applying Bayes' Rule to both sides (we omit the i, j indices for brevity):

$$P(R^t|E^s, E^e) = \frac{P(R^t|E^s)P(R^t|E^e)(1 - P(R^t))}{\sum_{i\in\{t,f\}} P(R^i|E^s)P(R^i|E^e)(1 - P(R^i))} \tag{4.5}$$

4.4.5 Comparing Clusters of Strings

Our algorithm merges clusters of strings with one another, using one of the above models. However, these models give probabilities for coreference decisions between two individual strings, not two clusters of strings.

We follow the work of Snow et al. [107] in incorporating transitive closure constraints in probabilistic modeling, and make the same independence assumptions. The benefit of this approach is that the calculation for merging two clusters depends only on coreference decisions between individual strings, which can be calculated independently.

Let a *clustering* be a set of coreference relationships between pairs of strings such that the coreference relationships obey the transitive closure property. We let the probability of a set of assertions D given a clustering C be:

$$P(D|C) = \prod_{R^t_{i,j} \in C} P(D_i \cup D_j | R^t_{i,j}) \times \prod_{R^f_{i,j} \in C} P(D_i \cup D_j | R^f_{i,j}) \tag{4.6}$$

The metric used to determine if two clusters should be merged is the likelihood ratio, or the probability for the set of assertions given the merged clusters over the probability given the original clustering. Let C' be a clustering that differs from C only in that two clusters in C have been merged in C', and let ΔC be the set of coreference relationships in C' that are true, but the corresponding ones in C are false. This metric is given by:

$$P(D|C')/P(D|C) = \frac{\prod_{R^t_{i,j} \in \Delta C} P(R^t_{i,j}|D_i \cup D_j)(1 - P(R^t_{i,j}))}{\prod_{R^t_{i,j} \in \Delta C}(1 - P(R^t_{i,j}|D_i \cup D_j))P(R^t_{i,j})} \tag{4.7}$$

The probability $P(R^t_{i,j}|D_i \cup D_j)$ may be supplied by the SSM, ESP, or combination model. In our experiments, we let the prior for the SSM model be 0.5. For the ESP and combined models, we set the prior to $P(R^t_{i,j}) = \frac{1}{\min(P_1,P_2)}$.

4.5 RESOLVER*'s Clustering Algorithm*

Synonym Resolution for the Web requires a clustering algorithm that can scale to huge numbers of strings in a sparse, high-dimensional space. Those requirements are difficult for any clustering algorithm. On the other hand, very few words have more than a handful of

synonyms, so clusters tend to be quite small. Greedy agglomerative approaches are well-suited to this type of clustering problem, since they start with the smallest possible clusters and merge them as needed.

The RESOLVER clustering algorithm is a version of greedy agglomerative clustering. The main difference between it and standard Agglomerative Clustering is that it can handle the sparse, high-dimensional space and huge numbers of elements required by SR. The algorithm iteratively merges clusters of co-referential names, making each iteration in time $O(N\ log\ N)$ in the number of extracted assertions. The algorithm requires only basic assumptions about which strings to compare. Previous work on speeding up clustering algorithms for SR has either required far stronger assumptions, or else it has focused on heuristic methods that remain, in the worst case, $O(N^2)$ in the number of distinct objects.

Our algorithm, a greedy agglomerative clustering method, is outlined in Figure 4.1. It begins by calculating similarity scores between pairs of strings, in steps 1-3. After the scores for all comparisons are made, each string is assigned its own cluster. Then the scores are sorted and the best cluster pairs are merged until no pair of clusters has a score above threshold. The novel part of the algorithm, step 3, compares pairs of strings that share the same property or instance, so long as no more than Max strings share that same property or instance. The property index limits the number of comparisons made between strings, and it is the reason for the algorithm's improved efficiency, as explained below.

This algorithm compares every pair of clusters that have the potential to be merged, assuming two properties of the data. First, it assumes that pairs of clusters with no shared properties are not worth comparing. Since the number of shared properties is a key source of evidence for our approach, these clusters almost certainly will not be merged, even if they are compared, so the assumption is quite reasonable. Second, the approach assumes that clusters sharing only properties that apply to very many strings (more than Max) need not be compared. Since properties shared by many strings provide little evidence that the strings are coreferential, this assumption is reasonable for SR. We use $Max = 50$ in our experiments. Less than 0.1% of the properties are thrown out using this cutoff.

$S :=$ set of all strings

For each property or instance p,

    $S_p := \{s \in S | s \text{ has property } p\}$

1. $Scores := \{\}$
2. Build index mapping properties (and instances)

    to strings with those properties (instances)

3. For each property or instance p:

    If $|S_p| <$ Max:

        For each pair $\{s1, s2\} \subset S_p$:

            Add mergeScore$(s1, s2)$ to $Scores$

4. Repeat until no merges can be performed:

    Sort $Scores$

    $UsedClusters := \{\}$

    While score of top clusters c_1, c_2 is above Threshold:

        Skip if either is in $UsedClusters$

        Merge c_1 and c_2

        Add c_1, c_2 to $UsedClusters$

    Recalculate merge scores involving merged clusters as in Steps 1-3

Figure 4.1: RESOLVER**'s Clustering Algorithm**

4.5.1 Algorithm Analysis

Let D be the set of extracted assertions. The following analysis shows that one iteration of merges takes time $O(|D|\ log\ |D|)$. Let NC be the number of comparisons between strings in step 3. To simplify the analysis, we consider only those properties that contain a relation string and an argument 1 string. Let A be the set of all such properties. NC is linear in N:[3]

$$\begin{aligned} NC &= \sum_{p \in A} \frac{|S_p| \times (|S_p| - 1)}{2} \\ &\leq \frac{(Max - 1)}{2} \times \sum_{p \in A} |S_p| \\ &= \frac{(Max - 1)}{2} \times |D| \end{aligned}$$

Note that this bound is quite loose because most properties apply to only a few strings. Step 4 requires time $O(|D|\ log\ |D|)$ to sort the comparison scores and perform one iteration of merges. If the largest cluster has size K, in the worst case the algorithm will take K iterations. In our experiments, the algorithm never took more than 9 iterations.

4.5.2 Relation to other speed-up techniques

The merge/purge algorithm [51] assumes the existence of a particular attribute such that when the data set is sorted on this attribute, matching pairs will all appear within a narrow window of one another. This algorithm is $O(M\ log\ M)$ where M is the number of distinct strings. However, there is no attribute or set of attributes that comes close to satisfying this assumption in the context of domain-independent information extraction.

There are several techniques that often provide speed-ups in practice, but in the worst case they make $O(M^2)$ comparisons at each merge iteration, where M is the number of distinct strings. This can cause problems on very large data sets. Notably, McCallum et al. [77] use a cheap comparison metric to place objects into overlapping "canopies," and

[3] If the Max parameter is allowed to vary with $log|D|$, rather than remaining constant, the same analysis leads to a slightly looser bound that is still better than $O(|D|^2)$.

then use a more expensive metric to cluster objects appearing in the same canopy. The RESOLVER clustering algorithm is in fact an adaptation of the canopy method; it adds the restriction that strings are not compared when they share only high-frequency properties. The canopy method works well on high-dimensional data with many clusters, which is the case with our problem, but its time complexity is worse than ours.

For information extraction data, a complexity of $O(M^2)$ in the number of distinct strings turns out to be considerably worse than our algorithm's complexity of $O(N\ log\ N)$ in the number of extracted assertions. This is because the data obeys a Zipf law relationship between the frequency of a string and its rank, so the number of distinct strings grows linearly or almost linearly with the number of assertions. The exact relationship depends on the shape parameter z of the Zipf curve. If $z < 1$, as it is for our data set, the number of total extractions grows linearly with the number of distinct strings extracted. For more details and a proof that the new bound is in fact an improvement, see Appendix C.

4.5.3 Status of the RESOLVER *Implementation*

RESOLVER currently exists as a Java package containing 23,338 lines of code. It has separate modules for calculating the Extracted Shared Property Model and the String Similarity Model, as well as for clustering extractions. The basic version of the system accepts a file containing TEXTRUNNER extractions as input, one extraction per line. Optionally, it accepts manually labeled clusters as input as well, and will use those to output precision and recall scores. The output of the system is two files containing all object clusters and relation clusters of size two or more, respectively. Optionally, the system also outputs precision and recall scores. Several other options allow the user to run extensions to the basic RESOLVER system, which are discussed below in Sections 4.7 and 4.8.

RESOLVER is currently a part of the TEXTRUNNER demonstration system.[4] This demonstration system contains extractions from 117 million Web documents. The extractions were fed into RESOLVER and the resulting clusters were added to the TEXTRUNNER index so

[4]The demonstration system is available for keyword searches over the Web at http://www.cs.washington.edu/research/textrunner/.

Figure 4.2: **The TEXTRUNNER search page, with the query `invented` entered into the search field.**

that keyword searches return results for any member of the cluster containing the keyword being searched for, and the displayed results are condensed such that members of the same cluster are not repeated. Figure 4.3 shows the results of searching TEXTRUNNER for the predicate `invented`. Note that in the results, a pop-up box displays all the synonyms for `Thomas Edison` that share the property `invented, light bulb`. Such pop-up boxes will display whenever a user clicks on a string that has a nontrivial cluster found by RESOLVER.

4.6 Experiments

Several experiments below test RESOLVER and ESP, and demonstrate their improvement over related techniques, Discovery of Inference Rules from Text (DIRT) [68] and the Cosine Similarity Metric (CSM) [101]. The first experiment compares the performance of the

Figure 4.3: **Results from searching for the predicate `invented` using TextRunner's search functionality. The synonyms that Resolver found for `Thomas Edison` are displayed in a pop-up box.**

various similarity metrics on measuring the similarity between pairs of strings, and it shows that ESP outperforms both DIRT and CSM, and that RESOLVER outperforms ESP and SSM on measuring similarity between strings. The second experiment measures the performance of the clustering method, and shows that RESOLVER's output clusters are significantly better than ESP's or SSM's, and ESP's clusters are in turn significantly better than DIRT's or CSM's.

4.6.1 Data

The models are tested on a data set of 2.1 million assertions extracted from a Web crawl. All models run over all assertions, but compare only those objects or relations that appear at least 25 times in the data, to give the distributional similarity models sufficient data for estimating similarity. In addition, only proper nouns[5] are compared, and only those relation strings that contain no punctuation or capital letters are compared. This helps to restrict the experiment to strings that are less prone to extraction errors. However, the models do use the other strings as features. In all, the data contains 9,797 distinct proper object strings and 10,151 distinct proper relation strings that appear at least 25 times. A gold standard data set was created by manually clustering a subset of 5,000 object and 2,000 relation strings.

4.6.2 Comparing Similarity Metrics

The first experiment compares similarity metrics using precision and recall. It compares the metrics discussed throughout this paper: the Cosine Similarity Metric (CSM) and the Discovery of Inference Rules From Text (DIRT) method, as two comparison points; the Extracted Shared Property (ESP) model; the String Similarity Metric (SSM); and the combination of ESP and SSM, or RESOLVER.

[5]Since the data set did not necessarily contain sufficient context around object strings in order to run standard named-entity recognizers, the following heuristic was used to detect proper nouns: if the string consisted of only alphabetic characters, whitespace, and periods, and if the first character of every word is capitalized, it is considered a proper noun. Otherwise, it is not.

Each model assigns a similarity score to every pair of strings that shares at least one property that appears fewer than Max times (see section 4.5). Each pair is labeled as correct if both strings belong to the same gold cluster, incorrect if they belong to different gold clusters, and irrelevant if one or both of the strings is not contained in the gold standard. Let the set of string pairs with similarity score above threshold t be S_t. Let the set of correct string pairs with score above t be C_t, and let the total number of correct string pairs in the gold standard be G. Then the precision for a model at threshold t is given by $\frac{|C_t|}{|S_t|}$, and the recall for a model at threshold t is given by $\frac{|C_t|}{|G|}$. Figures 4.4 and 4.5 show the precision-recall curves for ESP, DIRT, and CSM on objects and relations, respectively, as the threshold t is varied. Figures 4.6 and 4.7 show the performance of SSM and RESOLVER as well.

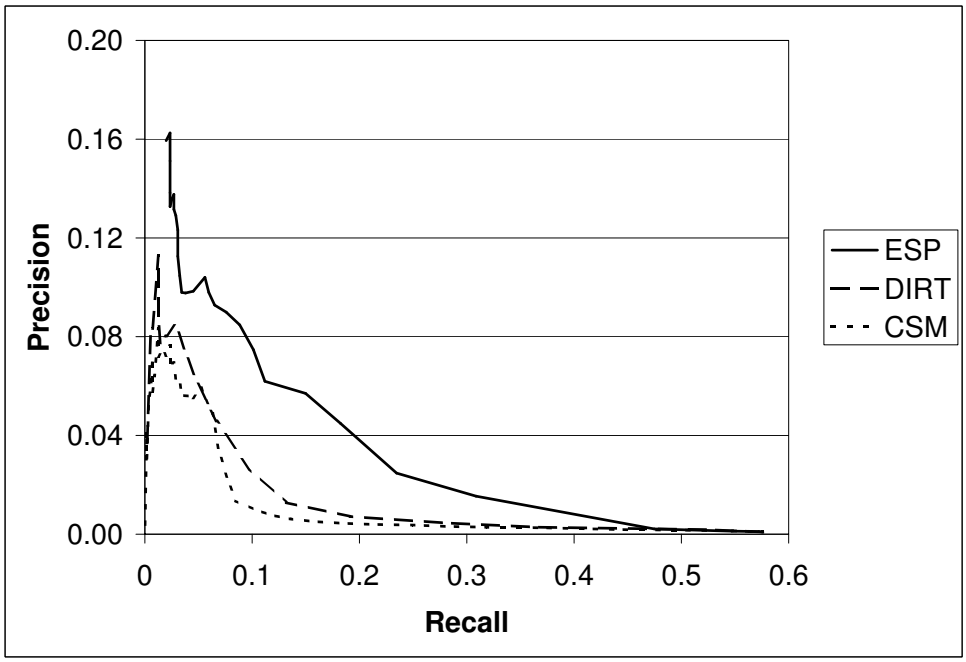


Figure 4.4: **The performance of the Extracted Shared Property (ESP) model, Discovery of Inference Rules from Text (DIRT), and Cosine Similarity Metric (CSM) on object pairs.** ESP's area under the curve is 193% higher than DIRT's and 273% higher than CSM's.

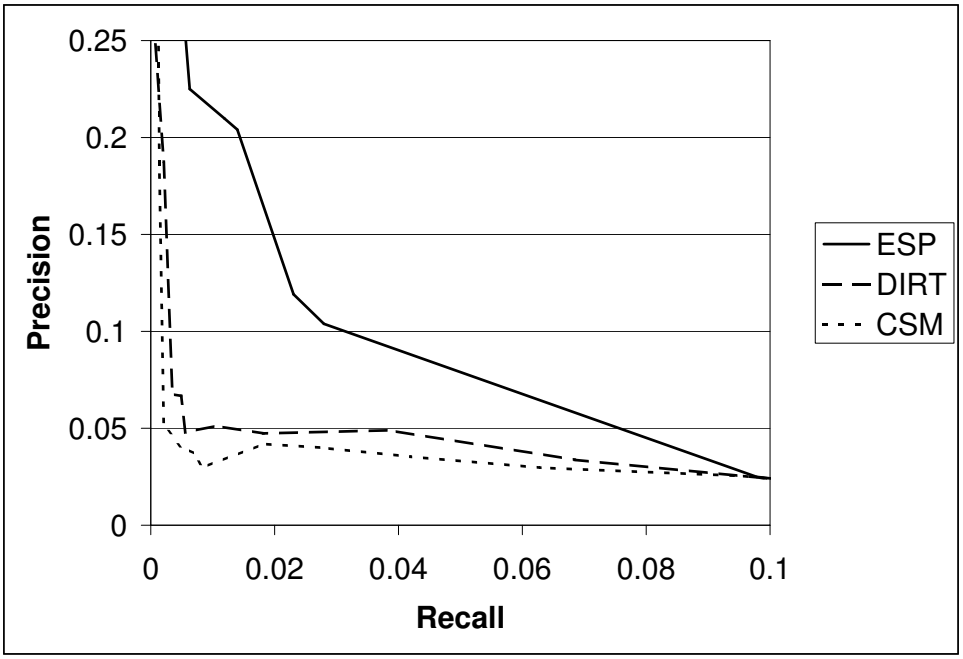


Figure 4.5: **The performance of the Extracted Shared Property (ESP) model, Discovery of Inference Rules from Text (DIRT), and Cosine Similarity Metric (CSM) on relation pairs.** ESP's area under the curve is 121% higher than DIRT's and 174% higher than CSM's.

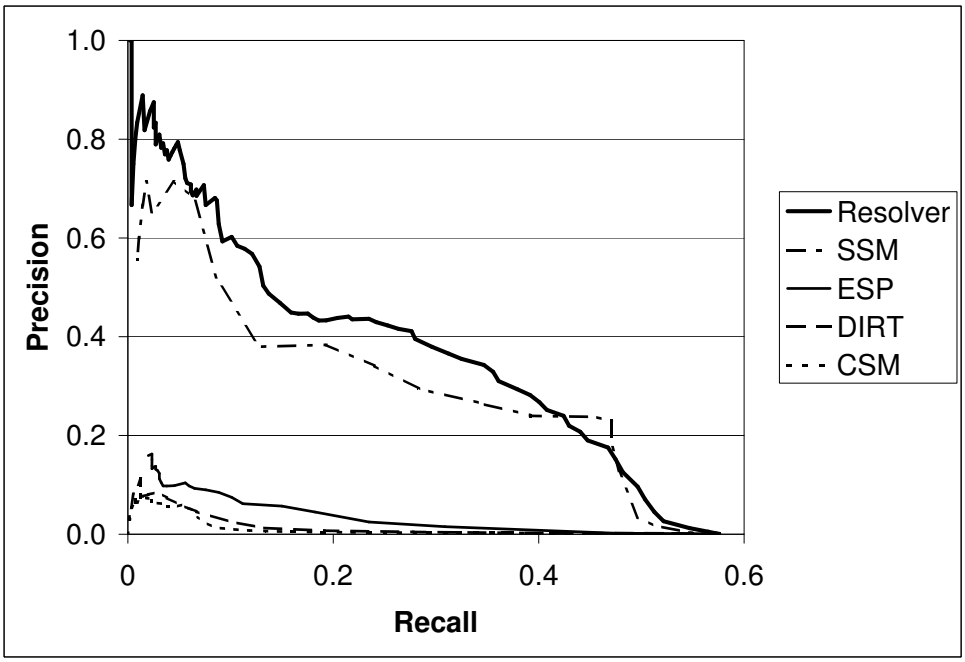


Figure 4.6: **Improvement in precision and recall of** RESOLVER **over the String Similarity Metric (SSM) for objects.** RESOLVER's area under the curve is 23% higher than SSM's.

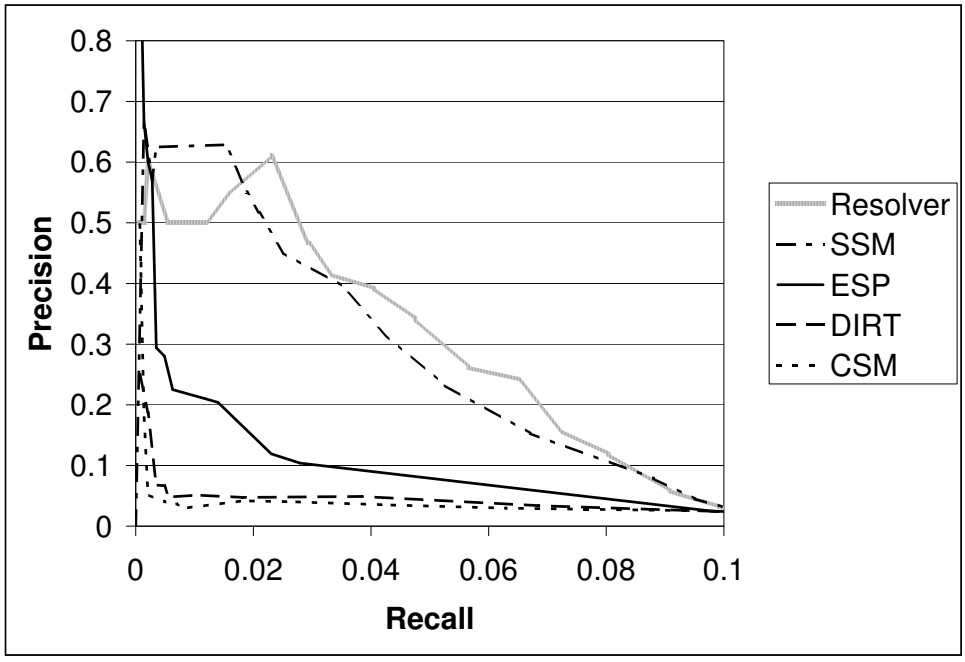


Figure 4.7: **Improvement in precision and recall of** RESOLVER **over the String Similarity Metric (SSM) for relations.** RESOLVER's area under the curve is 31% higher than SSM's.

4.6.3 Sensitivity Analysis

The ESP model requires a parameter for the number of potential properties of a string, but the performance of ESP is not strongly sensitive to the exact value of this parameter. As described in section 4.4.2, we assume that the number of potential properties is a multiple N of the number of extractions for a string. In the above experiments, we chose values of $N = 30$ for objects and $N = 500$ for relations, since they worked well on held-out data. However, as figures 4.8 and 4.9 show, the actual values of these parameters may vary in a large range, while still enabling ESP to outperform DIRT and CSM. Appendix D contains tables listing the areas under each of these precision-recall curves.

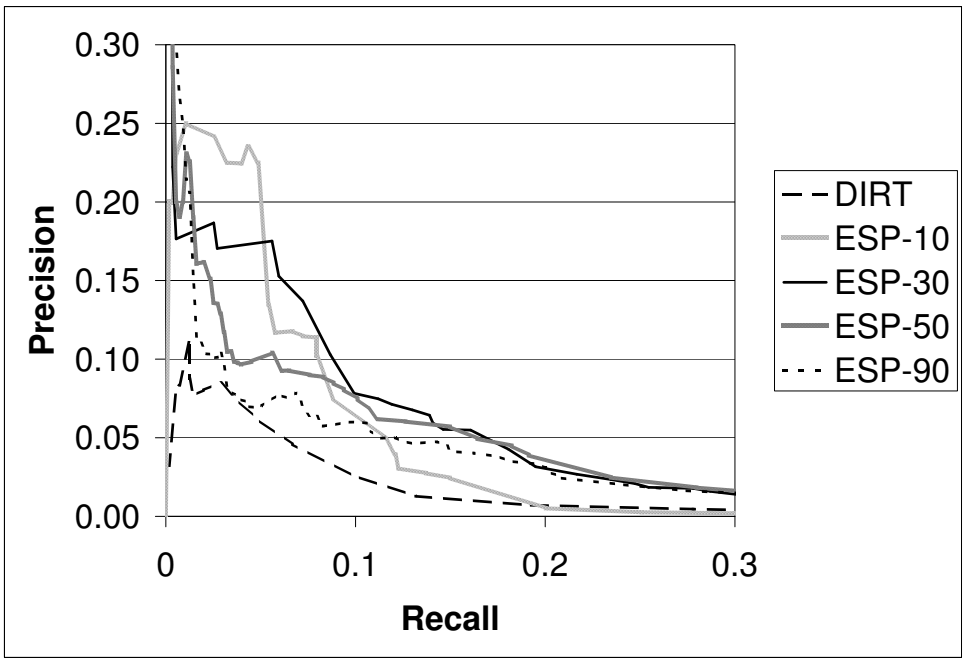


Figure 4.8: **A sensitivity analysis for the Extracted Shared Property (ESP) metric on objects.** ESP's area under the curve ranges between 121% (for ESP-90) and 193% (for ESP-30) higher than DIRT's.

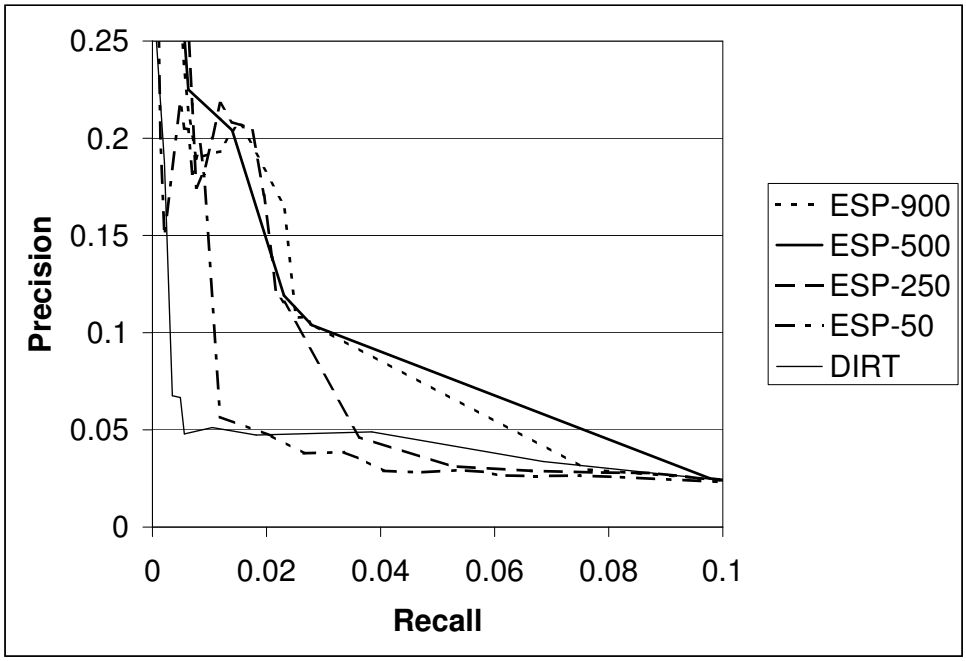


Figure 4.9: **A sensitivity analysis for the Extracted Shared Property (ESP) metric on relations.** ESP's area under the curve ranges between 98% (for ESP-250) and 133% (for ESP-900) higher than DIRT's, except for ESP-50, which is 9.5% higher than DIRT's.

4.6.4 Clustering Analysis

While the similarity metric is critical to the performance of RESOLVER, in and of itself it does not guarantee the accuracy of the clustering. The next experiment compares the precision and recall of clusterings output by the different similarity metrics, using the clustering algorithm described above.

The precision and recall of a clustering is measured as follows: hypothesis clusters are matched with gold clusters such that each hypothesis cluster matches no more than one gold cluster, and *vice versa*. This mapping is computed so that the number of elements in hypothesis clusters that intersect with elements in the matching gold clusters is maximized. All such intersecting elements are marked correct. Any elements in a hypothesis cluster that do not intersect with the corresponding gold cluster are marked incorrect, or irrelevant if they do not appear in the gold clustering at all. Likewise, gold cluster elements are marked as *found* if the matching hypothesis cluster contains the same element, or *not found* otherwise. The precision is defined as the number of correct hypothesis elements in clusters containing at least two relevant (correct or incorrect) elements, divided by the total number of relevant hypothesis elements in clusters containing at least two relevant items. The recall is defined as the number of found gold elements in gold clusters of size at least two, divided by the total number of gold elements in clusters of size at least two. We consider only clusters of size two or more in order to focus on the interesting cases.

Each model requires a threshold parameter to determine which scores are suitable for merging. For these experiments we arbitrarily chose a threshold of 3 for the ESP model (that is, the data needs to be 3 times more likely given the merged cluster than the unmerged clusters in order to perform the merge) and chose thresholds for the other models by hand so that the difference between them and ESP would be roughly even between precision and recall, although for relations it was harder to improve the recall. Table 4.1 shows the precision and recall of our models.

Table 4.1: **Comparison of the cosine similarity metric (CSM), Discovery of Inference Rules From Text (DIRT),** RESOLVER **components (SSM and ESP), and the** RESOLVER **system.** Bold indicates the score is significantly different from the score in the row above at $p < 0.05$ using the chi-squared test with one degree of freedom. Using the same test, RESOLVER is also significantly different from ESP, DIRT, and CSM in recall on objects, and from DIRT, CSM and SSM in recall on relations. RESOLVER's F1 on objects is a 19% increase over SSM's F1. RESOLVER's F1 on relations is a 28% increase over SSM's F1.

	Objects			Relations		
Model	Prec.	Rec.	F1	Prec.	Rec.	F1
CSM	0.51	0.36	0.42	0.62	0.29	0.40
DIRT	0.52	0.38	0.44	0.61	0.28	0.38
ESP	**0.56**	0.41	0.47	**0.79**	0.33	0.47
SSM	**0.62**	0.53	0.57	**0.85**	0.25	0.39
RESOLVER	**0.71**	0.66	0.68	**0.90**	**0.35**	0.50

4.6.5 Discussion

In all experiments, ESP outperforms both CSM and DIRT. The sensitivity analysis shows that this remains true for a wide range of hidden parameters for ESP, for both objects and relations. Moreover, ESP's improvement over the comparison metrics holds true when the metrics are used in clustering the data. DIRT's performance is largely the same as CSM in every experiment. Somewhat surprisingly, DIRT performs worse on relation clustering than on object clustering, even though it is designed for relation similarity.

ESP, DIRT, and CSM make predictions based on a very noisy signal. `Canada`, for example, shares more properties with `United States` in our data than `U.S.` does, even though `Canada` appears less often than `U.S.` The results show that the three distributional similarity models perform below the SSM model on its own for both objects and relations, both in the string similarity experiments and the clustering experiments. The one exception is in the clustering experiment for relations, where SSM had a poor recall, and thus had lower F1 score than ESP and CSM.

There is a significant improvement in both precision and recall when using a combined

model over using SSM alone. RESOLVER's F1 is 19% higher than SSM's on objects, and 28% higher on relations.

There is clearly room for improvement on the SR task. Error analysis shows that most of RESOLVER's mistakes are because of three kinds of errors:

1. *Extraction errors.* For example, `US News` gets extracted separately from `World Report`, and then RESOLVER clusters them together because they share almost all of the same properties.

2. *Similarity vs. Identity.* For example, `Larry Page` and `Sergei Brin` get merged, as do `Angelina Jolie` and `Brad Pitt`, and `Asia` and `Africa`.

3. *Multiple word senses.* For example, there are two President Bushes; also, there are many terms like `President` and `Army` that can refer to many different entities.

4.7 Distinguishing Between Similar and Identical Pairs

As the error analysis above suggests, similar objects that are not exact synonyms make up a large fraction of RESOLVER's errors. This section describes two techniques for dealing with such errors.

For example, RESOLVER is likely to make a mistake with the pair `Virginia` and `West Virginia`. They share many properties because they have the same type (U.S. states), and they have high string similarity. Perhaps the easiest approach for determining that these two are not synonymous is simply to collect more data about them. While they are highly similar, they will certainly not share all of their properties; they have different governors, for example. However, for highly similar pairs such as these two, the amount of data required to decide that they are not identical may be huge, and simply unavailable.

Fortunately, there are more sophisticated techniques for making decisions with the available data. One approach is to consider the distribution of words that occur between candidate synonyms. Similar words are likely to be separated by conjunctions (*e.g.*, "Virginia and West Virginia") and domain-specific relations that hold between two objects of the same type (*e.g.*, "Virginia is larger than West Virginia"). On the other hand, synonyms

are more likely to be separated by highly specialized phrases such as "a.k.a." Section 4.7.1 describes a method for using this information to distinguish between similar and identical pairs.

A second approach is to consider how candidate synonyms behave in the context of relations with special distributions, like functions or inverse functions. For example, the "x is capital of y" relation is an inverse function: every y argument has at most one x argument[6]. If capitals are extracted for both West Virginia and Virginia, then they may be ruled out as a synonymous pair when the capitals are seen to be different. On the other hand, if `Virginia` and `VA` share the same capital, that is much stronger evidence that the two are the same than if they shared some other random property, such as that a town called `Springfield` is located there. Section 4.7.2 describes a method for eliminating similar pairs because they have different values for the same function or inverse function, and section 4.7.3 illustrates a technique for assigning different weights to different evidence based on how close to functional it is. Section 4.7.4 gives results for each of these techniques, and section 4.7.5 investigates the function weighting technique further on artificial data.

4.7.1 Web Hitcounts for Synonym Discovery

While names for two similar objects may often appear together in the same sentence, it is relatively rare for two different names of the same object to appear in the same sentence. Moreover, synonymous pairs tend to appear in idiosyncratic contexts that are quite different from the contexts seen between similar pairs. RESOLVER exploits this fact by querying the Web to determine how often a pair of strings appears together in certain contexts in a large corpus. When the hitcount is high, RESOLVER can prevent the merge.

Specifically, given a candidate synonym pair s_1 and s_2, the Coordination-Phrase Filter uses a KnowItAll-style discriminator phrase of the form "s_1 and s_2". It then computes a variant of pointwise mutual information, given by

$$\text{coordination score}(s_1, s_2) = \frac{\text{hits}(s_1 \text{ and } s_2)^2}{\text{hits}(s_1) \times \text{hits}(s_2)}$$

[6]It is also a function.

The filter removes from consideration any candidate pair for which the coordination score is above a threshold, which is determined on a small development set. The results of coordination-phrase filtering are presented below.

The Coordination-Phrase Filter uses just one possible context between candidate synonym pairs. A simple extension is to use multiple discriminator phrases that include common context phrases like "or" and "unlike." A more complex approach could measure the distribution of words found between a candidate pair, and compare that distribution with the distributions found between known similar or known identical pairs. These are important avenues for further investigation.

One drawback of this approach is that it requires text containing a pair of objects in close proximity. For a pair of rare strings, such data will be extremely unlikely to occur — this type of test exacerbates the data sparsity problem. The following two sections describe two techniques that do not suffer from this particular problem.

4.7.2 Function Filtering

Functions and inverse functions can help to distinguish between similar and identical pairs. For example, `Virginia` and `West Virginia` have different capitals: `Richmond` and `Charleston`, respectively. If both of these facts are extracted, and if RESOLVER knows that the `capital of` relation is an inverse function, it should prevent `Virginia` and `West Virginia` from merging.

Given a candidate synonym pair x_1 and x_2, the Function Filter prevents merges between strings that have different values for the same function. More precisely, it decides that two strings y_1 and y_2 *match* if their string similarity is above a high threshold. It prevents a merge between x_1 and x_2 if there exists a function f and extractions $f(x_1, y_1)$ and $f(x_2, y_2)$, and there are no such extractions such that y_1 and y_2 match (and *vice versa* for inverse functions). Experiments described in section 4.7.4 show that the Function Filter can improve the precision of RESOLVER without significantly affecting its recall.

The Function Filter requires knowledge about which relations are actually functions or inverse functions. Others have investigated techniques for determining such properties of relations automatically [92]; in the experiments, a pre-defined list of functions is used. Table

Table 4.2: **The set of functions used by the Function Filter.**

is capital of	is capital city of
named after	was named after
headquartered in	is headquartered in
was born in	was born on

4.2 lists the set of functions used in the experiments for the Function Filter.

4.7.3 Function Weighting

While the Function Filter uses functions and inverse functions as negative evidence, it is also possible to use them as positive evidence. For example, the relation `married` is not strictly one-to-one, but for most people the set of spouses is very small. If a pair of strings are extracted with the same spouse—*e.g.*, `FDR` and `President Roosevelt` share the property `(married, Eleanor Roosevelt)`—this is far stronger evidence that the two strings are identical than if they shared some random property, such as `(spoke to, reporters)`.

There are several possibilities for incorporating this insight into RESOLVER, and those possibilities vary in two dimensions. First, any such technique will need some method for estimating the "function-ness" of a property, or how much weight to attach to the property if it is shared by a candidate synonym pair. Let this weight be called the *degree of functionhood*, or simply *degree*, of the property. Let a *high-degree* property be a property with a large degree of functionhood, and similarly for a *low-degree* property.

The degree may be estimated from the relation involved in the property and the set of extractions for that relation; or it may be based on the entire property and how many objects it applies to; or it may be some combination of both. For example, if there are 100 unique extractions for the `married` relation, and there are 80 unique x argument strings in those 100 extractions, then on average each x string participates in $100/80 = 1.25$ `married` relations. One method might assign every property containing the `married` relation this statistic as the degree. On the other hand, suppose there are two extractions for the property `(married,`

`John Smith)`. A second method is to assign a degree of 2 to this property.

The second dimension for incorporating function weighting into RESOLVER is how to use the degree of each property. The ESP model may be altered so that it directly models the degrees of the properties during the process of selecting balls from urns, but this vastly complicates the model and may make it much more computationally expensive. A second option is to reweight the number of shared properties between strings based on a TF-IDF style weighting of the properties, and calculate the ESP model using this parameter instead. This requires modifying the ESP model so that it can handle non-integer values for the number of shared properties.

In experiments so far, one set of these options is explored, while others remain for future investigation. The Weighted Extracted Shared Property Model (W-ESP) sets the degree of a property to be the number of extractions for that property. Second, if strings s_i and s_j share all properties $p \in P$, it sets the value for the number of shared properties between s_i and s_j to be

$$\sum_{p \in P} \frac{1}{\text{degree}(p)}$$

The ESP model is changed to handle continuous values for the number of shared properties by changing all factorials to gamma functions, and using Stirling's approximation whenever possible.

Unlike the Function Filter, the W-ESP model does not require additional knowledge about which relations are functional. And unlike the Coordination-Phrase Filter, it does not require Web hitcounts or a training phase. It works on extracted data, as is.

4.7.4 Experiments

The extensions to RESOLVER attempt to address the confusion between similar and identical pairs. Experiments with the extensions, using the same datasets and metrics as in section 4.6 demonstrate that the Function Filter (FF) and the Coordination-Phrase Filter (CPF) boost RESOLVER's precision. Unfortunately, the W-ESP model yielded essentially no improvement of RESOLVER.

Table 4.3 contains the results of our experiments. With coordination-phrase filtering,

Table 4.3: **Comparison of object merging results for the** Resolver **system,** Resolver **plus Function Filtering,** Resolver **plus Coordination-Phrase Filtering,** Resolver **using the Weighted Extracted Shared Property Model, and** Resolver **plus both types of filtering. Bold indicates the score is significantly different from** Resolver**'s score at $p < 0.05$ using the chi-squared test with one degree of freedom.** Resolver**+ Coordination Phrase Filtering's F1 on objects is a 28% increase over SSM's F1, and a 7% increase over** Resolver**'s F1.**

Model	Prec.	Rec.	F1
Resolver	0.71	0.66	0.68
Resolver + Function Filtering	0.74	0.66	0.70
Resolver + Coordination Phrase Filtering	**0.78**	0.68	0.73
Resolver + Weighted ESP	0.71	0.65	0.68
Resolver + Function Filtering + Coordination Phrase Filtering	**0.78**	0.68	0.73

Resolver's F1 is 28% higher than SSM's on objects, and 6% higher than Resolver's F1 without filtering. While function filtering is a promising idea, FF provides a smaller benefit than CPF on this dataset, and the merges that it prevents are, with a few exceptions, a subset of the merges prevented by CPF. This is in part due to the limited number of functions available in the data.

Both the Function Filter and the Coordination-Phrase Filter consistently blocked merges between highly similar countries, continents, planets, and people in our data, as well as some other smaller classes. The biggest difference is that CPF more consistently has hitcounts for the similar pairs that tend to be confused with identical pairs. Perhaps as the amount of extracted data grows, more functions and extractions with functions will be extracted, allowing the Function Filter to improve.

Part of the appeal of the W-ESP model is that it requires none of the additional inputs that the other two models require, and it applies to each property, rather than to a subset of the relations like the Function Filter. The next section investigates why it yields no benefit in our test.

4.7.5 Experiments with Function Weighting on Artificial Data

Function weighting proves not to have a great effect in experiments on the extracted TextRunner data. Since this is a surprising result, and because the TextRunner data contains messy errors from the extraction process, further experiments are carried out on artificial data to investigate why the function weighting has so little effect.

There are a number of possible causes to investigate. Recall that a *high-degree* property is one which is extracted with many different strings, while a *low-degree* property applies to only a few strings. Function weighting is affected by the following factors, among others:

1. Whether similar pairs tend to share high-degree or low-degree properties.

2. Whether identical pairs tend to share high-degree or low-degree properties.

3. The relative number of high-degree and low-degree properties.

4. The number of tuples per object and tuples per relation, which in turn affect the distribution of high-degree and low-degree properties.

5. The W-ESP model, which may be a poor model for function-weighting.

The TextRunner data in particular has a large number of distinct objects for the number of extractions: there are around 1.2 million distinct objects in 2 million extractions, or less than 2 extractions per object. In the experiment below, this factor (factor 4) is tested as a possible cause for the poor performance of function weighting; the others remain as future work.

The experiment proceeds by generating a number of different data sets, each with the same number of tuples and distinct relations, but varying the number of distinct objects. Then the ESP and W-ESP models are run on each data set to produce clusterings, which are measured for precision and recall in the same way as in Section 4.6.

The tuples and gold standard clusters are generated in a three step process. The process begins with a Zipf distribution over a fixed set of object clusters, and a separate Zipf distribution over a fixed set of relation clusters. In the first step, it creates a set of potential

tuples for each relation cluster by randomly selecting arguments to the relation from the Zipf distribution for objects, and throwing out duplicate tuples. Enough tuples are created for the relation so that there are N times the expected number of extractions for the relation (as governed by the number of output extractions and the Zipf distribution for relations), where N is a parameter called the *property multiple* that controls how many more potential instances of the relation there are over the number of actual extractions for the relation.

After creating tuples of potential extractions, the data generation process converts the tuples from tuples of clusters to tuples of strings as follows. To simplify the experiment, no relation synonyms are created, so relation strings are identified with their clusters. To more accurately reflect natural data, only one of every five object clusters is allowed to have synonyms. For every tuple that contains an object cluster which has synonyms, the tuple is repeated for every different synonym of the object cluster, from the most common synonym to the jth most common, where j is randomly generated between one and ten.

The first two steps create a set of potential extractions with synonyms. Finally, the data generator selects (without replacement) $\frac{1}{N}$ of the potential tuples as actual extractions.

Each generated data set contains one million tuples, and is generated with twenty thousand relations clusters and a property multiple of ten. The number of object clusters ranges from two thousand to two million across the different data sets. For both object and relation cluster Zipf distributions, the Zipf parameter is one. ESP and W-ESP both use a hidden parameter of thirty during these experiments; ESP uses a threshold of 2 while W-ESP uses a threshold of 1. Figure 4.10 shows the improvement by W-ESP over ESP in precision and recall for each data set.

The W-ESP model significantly outperforms the ESP model in all cases. As expected, there is a significant drop-off in the precision improvement as the number of tuples per object decreases. This confirms that the performance of function weighting is closely tied to the number of extractions per distinct object, and it suggests that if TextRunner extracted more data for the current set of objects, function weighting might have a significant positive effect.

The difference in performance on the artificial data is still significant even in the case most closely matching the natural data set, where there are two million potential objects

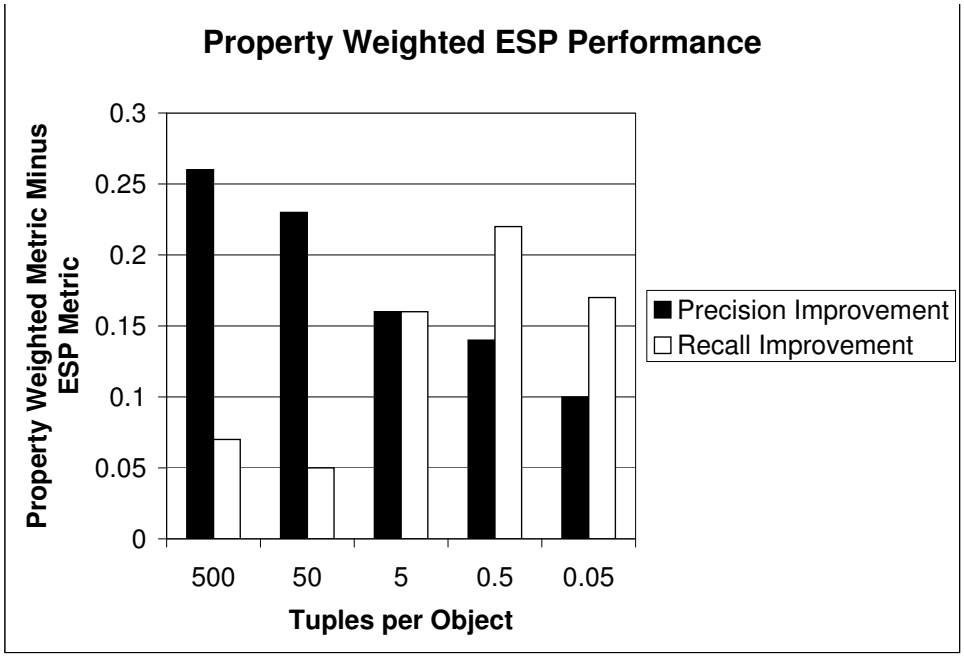


Figure 4.10: **The improvement in precision and recall by Weighted ESP (W-ESP) over ESP in clustering objects. The measurements are taken for five automatically generated data sets, where each data set differs in the average number of tuples per distinct object string (displayed on the x axis).**

and on average 0.5 tuples per object. This perhaps reflects the fact that the data was generated with a property multiple of ten, whereas a setting of thirty was most appropriate for the natural data, so the natural data still has less information about each object. Further experiments are necessary to test the performance of ESP and W-ESP at different settings of the property multiple.

An unexpected result from this experiment is that the recall improvement actually grows with the number of objects. This is misleading, however, because the difference in absolute numbers of discovered synonym pairs is similar across the board; there are simply fewer objects with a count of 25 or more as the number of objects grows, and so there are fewer synonym pairs to find.

4.8 Mutual Recursion

The RESOLVER clustering algorithm from Figure 4.1 clusters objects and relations independently. That is, if two object strings are determined to be synonymous, that has no effect on how relations are clustered, and *vice versa*.

Figure 4.11 shows a modified version of the clustering algorithm, which remedies this shortcoming using Mutual Recursion (MR). In MR, as the algorithm clusters relation strings together into sets of synonyms, it also collapses properties together for object strings. Thus it potentially finds more shared properties between coreferential object strings. Likewise, as it clusters objects together into sets of coreferential names, it collapses instances of relations together and potentially finds more shared instances between coreferential relations. Thus the clustering decisions for relations and objects mutually depend on one another.

For example, the strings `Kennedy` and `President Kennedy` appear in 430 and 97 assertions in the TEXTRUNNER data, respectively, but none of their extracted properties match exactly. Some properties, however, *almost* match. The assertions `(challenged, Kennedy, Premier Krushchev)` and `(stood up to, President Kennedy, Kruschev)` both appear in our data. Because `challenged` and `stood up to` are similar, and `Krushchev` and `Premier Krushchev` are similar, our algorithm is able to merge these pairs into two clusters, thereby creating a new shared property between `Kennedy` and `President Kennedy`. Eventually it can merge these two strings as well.

S := set of all strings

For each property or instance p,

    $S_p := \{s \in S | s \text{ has property } p\}$

1. $Scores := \{\}$
2. Build index mapping properties (and instances)

    to strings with those properties (instances)

3. For each property or instance p:

    If $|S_p| <$ Max:

        For each pair $\{s1, s2\} \subset S_p$:

            Add mergeScore$(s1, s2)$ to $Scores$

4. Repeat until no merges can be performed:

    Sort $Scores$

    $UsedClusters := \{\}$

    While score of top clusters c_1, c_2 is above Threshold:

        Skip if either is in $UsedClusters$

        Merge c_1 and c_2

        Add c_1, c_2 to $UsedClusters$

        Merge properties containing c_1, c_2

    Recalculate merge scores as in Steps 1-3

Figure 4.11: **Clustering Algorithm with Mutual Recursion. Differences from the original clustering algorithm are highlighted in bold.**

It turns out that MR has very little effect on clustering performance for the TextRunner data set: in a clustering experiment on objects, Resolver has precision 0.70 and recall 0.67 using MR, as compared with 0.71 and 0.66 without it; on relations, Resolver has precision 0.89 and recall 0.35 using MR, compared with 0.9 and 0.35 without it. MR helps to increase recall at the expense of precision, but the differences are not statistically significant.

As with the function weighting above, we turn to artificial data to investigate why MR has so little effect. Again, there are many possible factors to explore, including:

1. If synonyms share only a few properties in the data, the mutual recursion process will be more likely to create shared properties between non-synonymous strings than between synonymous ones.

2. MR requires pairs of tuples in which the same fact is repeated twice, but with two or three of the strings replaced with synonyms. If such pairs are rare, MR will have little effect.

3. If such pairs of tuples exist, but they mostly occur only where the same fact is also repeated with only one string replaced by a synonym, MR will not create new shared properties.

4. MR may propagate errors in clustering decisions so that those errors affect other clustering decisions.

In the experiment below, the first two factors are explored.

In this experiment, a number of artificial data sets are created such that all parameters are held constant except for two: the probability that a fact is repeated, but with some strings replaced by synonyms; and the probability that when a fact is repeated, exactly two strings are replaced by synonyms rather than one. For each data set, clusterings are computed using ESP and ESP with MR, and the precision and recall of each clustering is calculated using the same method as in Section 4.6.

Each data set is generated in three steps. Using a Zipf distribution for object clusters and a separate Zipf distribution for relation clusters, a set of unique tuples are created by independently generating two arguments and a relation for each tuple, and throwing out any duplicate tuple. Each tuple is then transformed from a tuple of clusters to a tuple of strings by generating a synonym string for each argument and relation. With probability p_{syn}, the cluster is replaced with a string representing the most common synonym for the cluster. If the first synonym is not chosen, then with probability p_{syn} it is replaced with the next most common synonym. Otherwise, with probability p_{syn}, it is replaced with the third most common synonym, and so on, until a synonym is finally chosen. The maximum number of synonyms allowed is ten.

In the last step, some repeated facts are added to the set of extractions. These repeated facts are controlled by two parameters: the *repetition probability* (p_{rep}), the probability that a fact is repeated; and the *double synonym probability* (p_{double}), the probability that a repeated tuple differs from the original in two positions rather than just one. For each tuple, the data generation process first determines if the tuple is to be repeated, as governed by p_{rep}. If not, it moves on to the next tuple. If so, it next determines if one or two of the three arguments are to be replaced by synonyms, as governed by p_{double}. It then picks one or two arguments with equal probability, and replaces each with another string belonging to the same cluster, using the above method and retrying if the same string is selected as in the original.

In this experiment, every data set starts with one million tuples of clusters, but the exact number of tuples depends on p_{rep}. The distribution for object clusters is a Zipf distribution over 100,000 distinct clusters, with a Zipf parameter of one. Likewise, the distribution for relation clusters is a Zipf distribution over 20,000 distinct clusters, with a Zipf parameter of one. In all data sets, $p_{syn} = 0.6$. The parameter p_{rep} is set to 0.05, 0.1, and 0.25, and the parameter p_{double} is set to 0.1, 0.25, and 0.5. The ESP model is run with a hidden parameter of thirty for both objects and relations; a threshold of 2.5 is used for objects and 2 for relations. Figures 4.12 and 4.13 show the difference in precision and recall between ESP and ESP using MR at these different parameter settings, on objects and relations respectively. Appendix D contains tables that separately list the precision and recall results

for both ESP and ESP using MR.

In contrast to the settings used for the artificial data sets, there are just 2,270 duplicate facts in the TextRunner data, out of 2,080,060 total extractions, so only about one in every 1,000 facts is repeated. And of the 2,270 duplicate facts, only 156 (6.9%) are double synonyms, in which more than one argument is replaced with a synonym.

The experiments on artificial data show that MR can have a significant effect (both negative and positive), depending on the data set. But, at least on artificial data, much higher rates of repetition probability and double synonym probability are necessary to achieve significant performance advantages using MR. For both objects and relations, significant gains do not occur until p_{rep} reaches 0.1 and p_{double} reaches 0.25, as opposed to 0.001 and 0.07 for the natural data set. When both parameters reach their maximum, MR boosts recall by a dramatic 16% for objects, and 13% for relations, with minor changes to the precision.

On the artificial data, at $p_{rep} = 0.05$ MR tends either to significantly hurt precision (when p_{double} is low), or to have no effect on performance (when $p_{double} = 0.5$). At these low levels of p_{rep}, a high value for p_{double} means that there are very few repeated facts with a single synonym change, and ESP therefore has very little evidence to get started. A low value for p_{double} means that merging properties is unlikely to result in a new shared property between synonyms; far more often, it will result in a new shared property between non-synonyms. This causes the drop in precision.

At higher levels of p_{rep}, a low value for p_{double} still causes problems, since a merged property is still more likely to connect non-synonyms than synonyms. The higher values for p_{rep} allow ESP to recover to some extent, however, since synonyms are already very well-connected.

Somewhat surprisingly, MR sometimes hurts recall in these situations without hurting precision. This is surprising because MR is intended to increase or leave the same the number of shared properties between any two strings, and therefore to increase the likelihood score. Since the threshold for merging is the same for ESP with and without MR, one would expect a greater number of merges using MR. Thus if the precision is the same, one would expect a higher recall using MR.

The trouble with this logic is that MR can, in fact, reduce the number of shared prop-

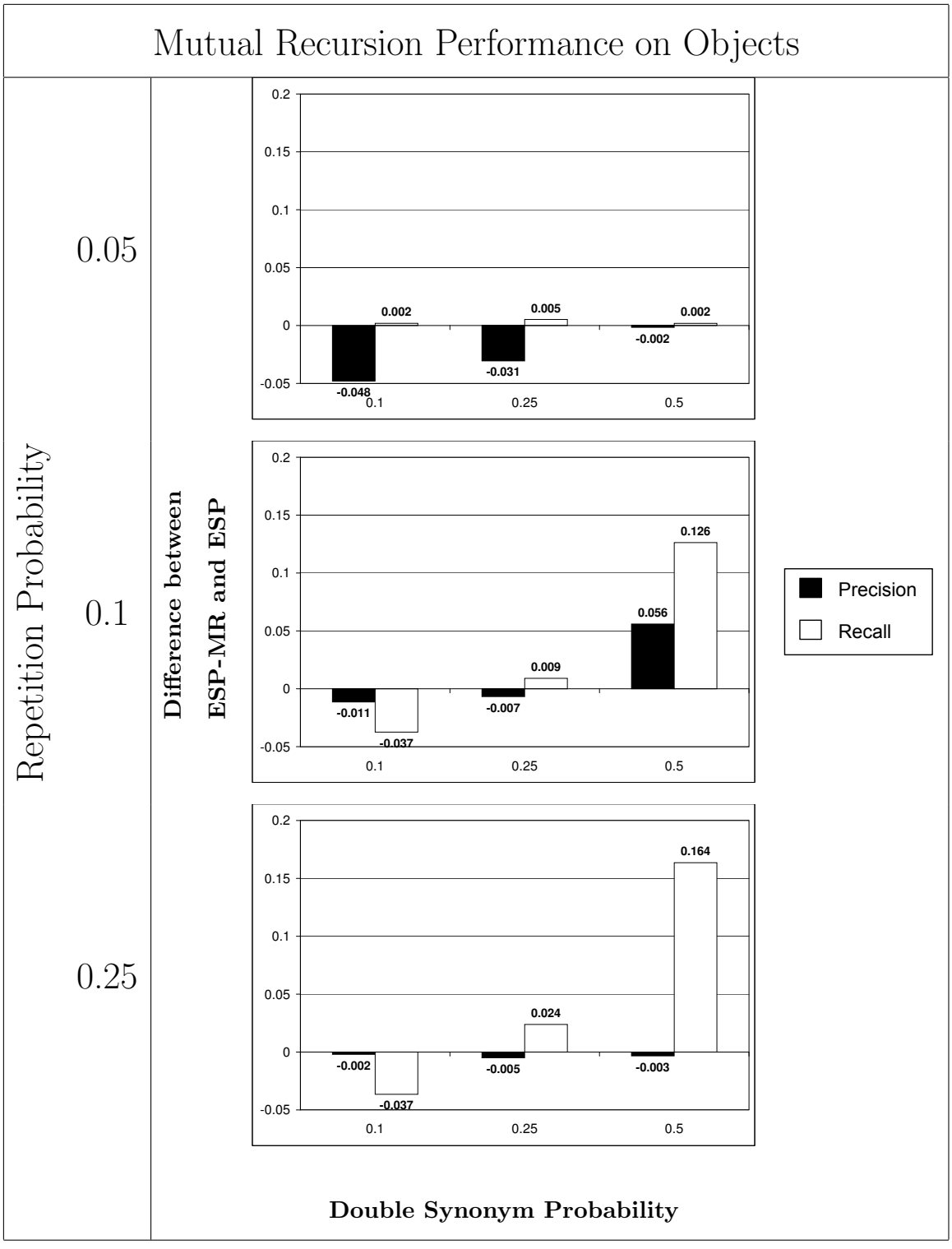


Figure 4.12: **Difference in precision and recall between ESP clusterings and ESP using MR (ESP-MR) clusterings, on artificial data.**

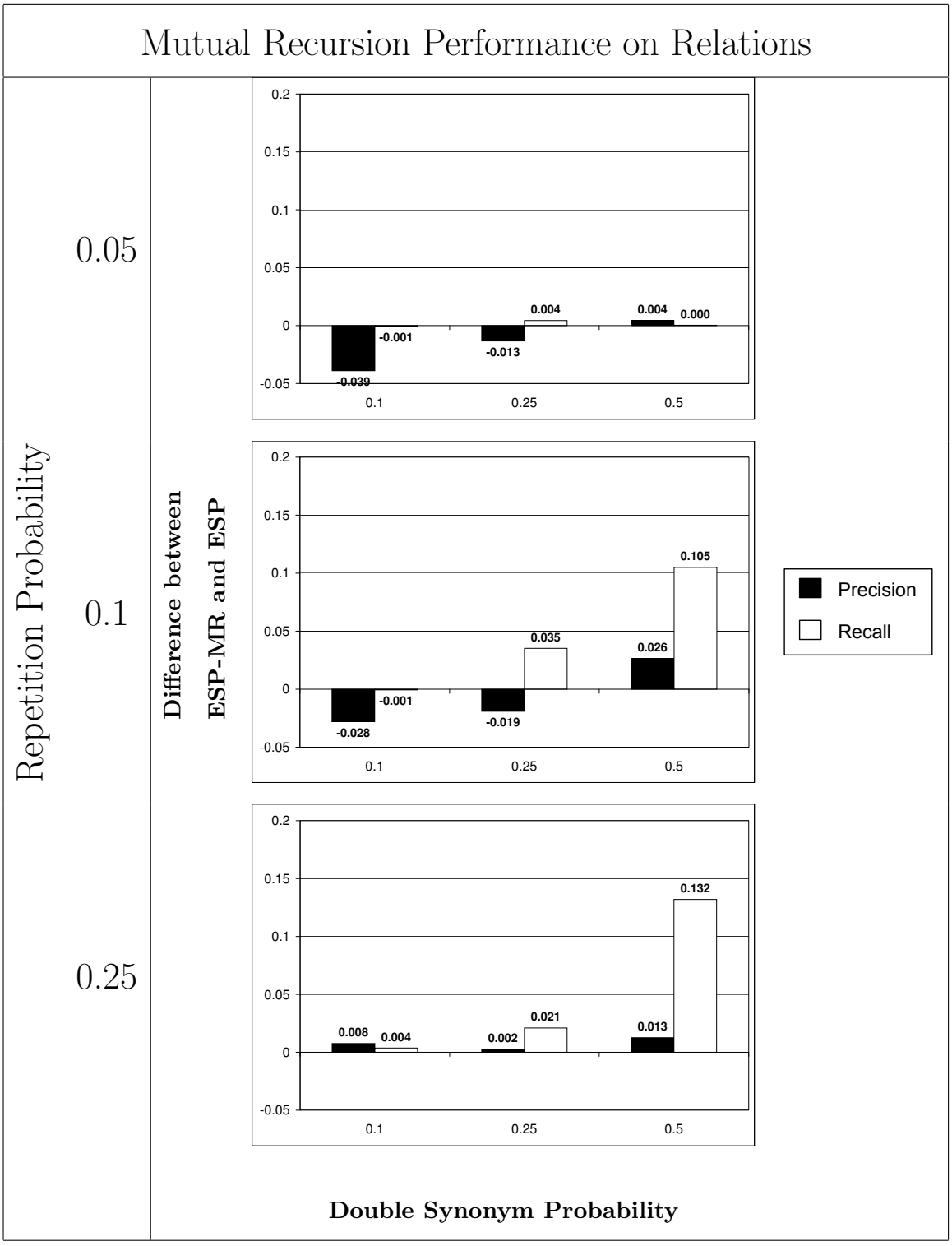


Figure 4.13: **Difference in precision and recall between ESP clusterings and ESP using MR (ESP-MR) clusterings, on artificial data.**

erties between two synonyms. Consider the following four extractions: `(is capital of, D.C., U.S.)`; `(is capital of, Washington, U.S.)`; `(is capital city of, D.C., U.S.)`; `(is capital city of, Washington, U.S.)`. Without MR, `D.C.` and `Washington` would share two properties. But with MR, if `is capital of` and `is capital city of` are merged, then those two properties would be collapsed into one.

In several cases, MR actually improves the precision of the ESP model. When MR creates enough new shared properties between synonyms (that is, at high levels of both parameters), the benefit of the increase in shared properties outweighs any propagation of errors due to MR and the increase in the number of shared properties between random non-synonyms. Also, when a gold standard cluster is split into two clusters by the ESP model, that hurts ESP's precision. If MR allows the two smaller clusters to be merged together, it can consequently boost the precision. Both of these effects tend to be small relative to the increased recall, however.

4.9 Conclusion and Future Work

We have shown that the unsupervised and scalable RESOLVER system is able to find clusters of co-referential object names in extracted relations with a precision of 78% and a recall of 68% with the aid of coordination-phrase filtering, and can find clusters of co-referential relation names with precision of 90% and recall of 35%. We have demonstrated significant improvements over using simple similarity metrics for this task by employing a novel probabilistic model of coreference.

Several extensions to RESOLVER have dealt with ruling out highly similar non-synonyms, with varying degrees of success. Yet these methods only scratch the surface of the possible methods. Just as RESOLVER uses distributional similarity to merge synonyms, it should prove interesting to use distributional similarity to detect non-synonyms, as suggested above. This remains an important item for future work.

Synonymy is critical to information extraction, but it is likewise extremely important to discover automatically when a word has multiple meanings. Polysemy, in fact, causes a substantial fraction of RESOLVER's errors in finding synonyms. Extending the ESP model and RESOLVER's clustering algorithm to be able to detect and handle polysemy could sub-

stantially improve its results.

Chapter 5

CONCLUSION

This thesis has demonstrated novel techniques and applications for Web Information Extraction. The RESOLVER system addresses Synonym Resolution, one major hurdle for unsupervised information extraction, and it improves the state of the art in identifying and efficiently clustering synonyms. Techniques like RESOLVER promise to make extracted information broadly useful to many different applications. WOODWARD shows how to make the extracted knowledge useful for one particular language processing application, the statistical parser.

More specifically, the thesis has described the following contributions to Web Information Extraction. Within the area of synonym resolution, it has demonstrated a provably scalable algorithm for clustering synonyms in Zipf-distributed data and a novel, unsupervised model for determining the probability that two strings are synonyms, called the Extracted Shared Property Model (ESP). Experiments have shown that ESP outperforms state-of-the-art distributional similarity techniques on information extracted from the Web, and that when combined with an edit-distance notion of similarity, ESP can find object synonyms with 72% precision and 68% recall, and relation synonyms with 90% precision and 35% recall.

Three extensions to RESOLVER have exhibited new techniques for improving the ability to determine the difference between similar and identical pairs: a Function Filter based on the use of functions and inverse functions; a Coordination-Phrase Filter based on Web hitcounts; and a Property Weighted ESP model based on weighting properties according to how informative they are for resolving synonyms. Experiments show that the Function Filter and Coordination-Phrase Filter improve RESOLVER's precision by 3% and 6% respectively on TEXTRUNNER data. The Property Weighted ESP model yields no benefit on natural data, but experiments on artificial data show that the model tends to have a greater benefit when there are more extracted properties per object; this suggests that the model may be

important on other, more suitable data sets.

A fourth extension to RESOLVER established a technique for making object and relation synonym resolution dependent on one another, via mutual recursion. While mutual recursion had little effect on natural data, experiments showed that on artificial data, there were suitable conditions such that mutual recursion would have a large impact on recall, with increases of as much as 16%.

Within the area of parsing, the thesis has described a new set of techniques for identifying incorrectly-parsed sentences using information extracted from the Web. The techniques are based on Web search engine hitcounts, collections of extracted information in TEXTRUNNER, patterns for discovering from the Web the number of arguments appropriate for a verb, and a question-answering tool. Experiments show that the four semantic filters that use these techniques are able to reduce the parser errors on TREC 2004 data by 67%, and on harder sets of Wall Street Journal sentences the WOODWARD system still reduces error by 20%.

In short, the thesis has demonstrated the value of extracted information for a particular natural language processing application — statistical parsers — and it has advanced the state of the art in one task in information extraction — synonym resolution.

BIBLIOGRAPHY

[1] E. Agichtein and L. Gravano. Snowball: Extracting relations from large plain-text collections. In *Procs. of the Fifth ACM International Conference on Digital Libraries*, 2000.

[2] E. Agichtein and L. Gravano. Querying Text Databases for Efficient Information Extraction. In *Procs. 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 *Procs. of the 2001 ACM SIGMOD International Conference on Management of Data*, Santa Barbara, California, 2001.

[4] E. Agichtein, S. Lawrence, and L. Gravano. Learning Search Engine Specific Query Transformations for Question Answering. In *Proceedings of the 10th International World Wide Web Conference*, pages 169–178, Hong Kong, China, 2001.

[5] J. Allen. *Natural Language Understanding*. Benjamin/Cummings Publishing, Redwood City, CA, 2nd edition, 1995.

[6] G. Attardi, A. Cisternino, F. Formica, M. Simi, and A. Tommasi. PiQASso: Pisa Question Answering System. In *TREC*, 2001.

[7] A. Bagga and B. Baldwin. Entity-based cross-document coreferencing using the vector space model. In *COLING-ACL*, 1998.

[8] M. Banko, M. J. Cafarella, S. Soderland, M. Broadhead, and O. Etzioni. Open information extraction from the web. In *IJCAI*, 2007.

[9] I. Bhattacharya and L. Getoor. Relational Clustering for Multi-type Entity Resolution. In *11th ACM SIGKDD Workshop on Multi Relational Data Mining*, 2005.

[10] I. Bhattacharya and L. Getoor. Query-time entity resolution. In *KDD*, 2006.

[11] 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.

[12] J. Bos, S. Clark, M. Steedman, J. R. Curran, and J. Hockenmaier. Wide-coverage semantic representations from a CCG parser. In *COLING*, 2004.

[13] E. Brill. Some Advances in Rule-Based Part of Speech Tagging. In *AAAI*, pages 722–727, Seattle, Washington, 1994.

[14] 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.

[15] R. Bunescu and R. Mooney. A shortest path dependency kernel for relation extraction. In *Proc. of the HLT/EMLNP*, 2005.

[16] Razvan Bunescu and Raymond Mooney. Learning to extract relations from the web using minimal supervision. In *Proceedings of the ACL*, 2007.

[17] M. Cafarella, D. Downey, S. Soderland, and O. Etzioni. KnowItNow: Fast, Scalable Information Extraction from the Web. In *Procs. of EMNLP*, 2005.

[18] 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.

[19] Joyce Yue Chai and Alan W. Biermann. The use of word sense disambiguation in an information extraction system. In *AAAI/IAAI*, pages 850–855, 1999.

[20] Hai Leong Chieu, Hwee Tou Ng, and Yoong Keok Lee. Closing the gap: learning-based information extraction rivaling knowledge-engineering methods. In *Proceedings of the ACL*, 2003.

[21] T. Chklovski and P. Pantel. VerbOcean: Mining the web for fine-grained semantic verb relations. In *Procs. of the Conference on Empirical Methods in Natural Language Processing (EMNLP-04)*, pages 33–40, 2004.

[22] M. Ciaramita, A. Gangemi, E. Ratsch, J. Saric, and I. Rojas. Unsupervised learning of semantic relations between concepts of a molecular biology ontology. In *Proceedings of IJCAI*, 2005.

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

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

[25] W.W. Cohen, P. Ravikumar, and S.E. Fienberg. A comparison of string distance metrics for name-matching tasks. In *IIWeb*, 2003.

[26] M. Collins. Discriminative reranking for natural language parsing. In *ICML*, pages 175–182, 2000.

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

[28] Michael Collins. Three generative, lexicalised models for statistical parsing. In *Proceedings of the 35th Annual Meeting of the ACL*, 1997.

[29] Michael Collins. *Head-driven statistical models for natural language parsing.* PhD thesis, University of Pennsylvania, 1999.

[30] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* The MIT Press, 1990.

[31] 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.

[32] A. Culotta, A. McCallum, and J. Betz. Integrating probabilistic extraction models and relational data mining to discover relations and patterns in text. In *Proceedings of HLT-NAACL, New York, NY*, 2006.

[33] I. Dagan, O. Glickman, and B. Magnini. The PASCAL Recognising Textual Entailment Challenge. *Lecture Notes in Computer Science*, 3944:177–190, 2006.

[34] S. Dill, N. Eiron, D. Gibson, D. Gruhl, R. Guha, A. Jhingran, T. Kanungo, S. Rajagopalan, A. Tomkins, J. Tomlin, and Z. 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.

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

[36] X. Dong, A.Y. Halevy, and J. Madhavan. Reference reconciliation in complex information spaces. In *SIGMOD*, 2005.

[37] D. Downey, M. Broadhead, and O. Etzioni. Locating complex named entities in web text. In *Procs. of the 20th International Joint Conference on Artificial Intelligence (IJCAI 2007)*, 2007.

[38] D. Downey, O. Etzioni, and S. Soderland. A Probabilistic Model of Redundancy in Information Extraction. In *IJCAI*, 2005.

[39] D. Downey, S. Schoenmackers, and O. Etzioni. Sparse information extraction: Unsupervised language models to the rescue. In *ACL*, 2007.

[40] O. Etzioni. Moving Up the Information Food Chain: Softbots as Information Carnivores. In *AAAI*, 1996. Revised version reprinted in AI Magazine special issue, Summer '97.

[41] 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 *WWW*, pages 100–110, New York City, New York, 2004.

[42] O. Etzioni, M. Cafarella, D. Downey, S. Kok, A. Popescu, T. Shaked, S. Soderland, D. Weld, and A. Yates. Unsupervised named-entity extraction from the web: An experimental study. *Artificial Intelligence*, 165(1):91–134, 2005.

[43] O. Etzioni, M. Cafarella, D. Downey, A. Popescu, T. Shaked, S. Soderland, D. Weld, and A. Yates. Methods for domain-independent information extraction from the Web: An experimental comparison. In *Procs. of the 19th National Conference on Artificial Intelligence (AAAI-04)*, pages 391–398, San Jose, California, 2004.

[44] 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.

[45] Daniel Gildea. Corpus Variation and Parser Performance. In *Conference on Empirical Methods in Natural Language Processing*, 2001.

[46] R.V. Guha and D.B. Lenat. Cyc: a mid-term report. *AI Magazine*, 11(3), 1990.

[47] R. Gupta and M. J. Kochenderfer. Common sense data acquisition for indoor mobile robots. In *Nineteenth National Conference on Artificial Intelligence*, 2004.

[48] V. Hatzivassiloglou and K. McKeown. Towards the automatic identification of adjectival scales: clustering adjectives according to meaning. In *Proceedings of ACL*, pages 182–192, 1993.

[49] V. Hatzivassiloglou and K. McKeown. Predicting the semantic orientation of adjectives. In *Proceedings of ACL/EACL*, pages 174–181, 1997.

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

[51] M. A. Hernandez and S. J. Stolfo. The merge/purge problem for large databases. In *SIGMOD*, 1995.

[52] D. Hindle. Noun classification from predicage-argument structures. In *ACL*, 1990.

[53] M. Hu and B. Liu. Mining and Summarizing Customer Reviews. In *Proceedings of KDD*, pages 168–177, Seattle, WA, 2004.

[54] N. Ide and J. Veronis. Word Sense Disambiguation: The State of the Art. *Computational Linguistics*, 24(1):1–40, 1998.

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

[56] N. Kambhatla. Combining lexical, syntactic and semantic features with maximum entropy models. In *Proceedings of ACL*, 2004.

[57] A. Kehler. Probabilistic coreference in information extraction. In *EMNLP*, 1997.

[58] S. Kim and E. Hovy. Determining the sentiment of opinions. In *Procs. of COLING*, 2004.

[59] Dan Klein and Christopher D. Manning. Accurate unlexicalized parsing. In *Proceedings of the ACL*, 2003.

[60] Stanley Kok and Pedro Domingos. Statistical predicate invention. In *Proceedings of the Twenty-Fourth International Conference on Machine Learning*, 2007.

[61] C. C. T. Kwok, O. Etzioni, and D. S. Weld. Scaling question answering to the web. In *WWW*, 2001.

[62] 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.

[63] M. Lapata and F. Keller. Web-based models for natural language processing. *ACM Transactions on Speech and Language Processing*, 2:1–31, 2005.

[64] S. Lappin and H. J. Leass. An algorithm for pronominal anaphora resolution. *Computational Linguistics*, 20(4):535–561, 1994.

[65] C. Leacock, M. Chodorow, and G. A. Miller. Using corpus statistics and WordNet relations for sense identification. *Computational Linguistics*, 24(1):147–165, 1998.

[66] Lillian Lee. Measures of distributional similarity. In *Proceedings of the 37th ACL*, 1999.

[67] Hang Li and Naoki Abe. Word clustering and disambiguation based on co-occurence data. In *COLING-ACL*, pages 749–755, 1998.

[68] D. Lin and P. Pantel. DIRT – Discovery of Inference Rules from Text. In *KDD*, 2001.

[69] H. Liu, H. Lieberman, and T. Selker. Exploiting agreement and disagreement of human annotators for word sense disambiguation. In *Proceedings of the Seventh International Conference on Intelligent User Interfaces*, 2003.

[70] H. Liu and P. Singh. OMCSNet: A commonsense inference toolkit. Technical Report SOM02-01, MIT Media Lab Society Of Mind Group, 2003.

[71] 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.

[72] G.S. Mann and D. Yarowsky. Unsupervised personal name disambiguation. In *CoNLL*, 2003.

[73] K. Markert, N. Modjeska, and M. Nissim. Using the web for nominal anaphora resolution. In *EACL Workshop on the Computational Treatment of Anaphora*, 2003.

[74] C. Matuszek, M. Witbrock, R.C. Kahlert, J. Cabral, D. Schneider, P. Shah, and D. Lenat. Searching for common sense: Populating Cyc from the web. In *Proceedings of AAAI*, 2005.

[75] 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.

[76] 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.

[77] A. McCallum, K. Nigam, and L. Ungar. Efficient clustering of high-dimensional data sets with application to reference matching. In *KDD*, 2000.

[78] A. McCallum and B. Wellner. Conditional models of identity uncertainty with application to noun coreference. In *NIPS*, 2004.

[79] George A. Miller, Richard Beckwith, Christiane Fellbaum, Derek Gross, and Katherine J. Miller. Introduction to WordNet: An on-line lexical database. *International Journal of Lexicography*, 3(4):235–312, 1990.

[80] S. Miller, H. Fox, L. Ramshaw, and R. Weischedel. A novel use of statistical parsing to extract information from text. In *6th Applied Natural Language Processing Conference*, 2000.

[81] D. Moldovan, C. Clark, S. Harabagiu, and S. Maiorano. Cogex: A logic prover for question answering. In *HLT*, 2003.

[82] D. Moldovan, S. Harabagiu, R. Girju, P. Morarescu, F. Lacatusu, A. Novischii, A. Badulescu, and O. Bolohan. LCC tools for question answering. In *Proceedings of the Eleventh Text REtrieval Conference (TREC 2002)*, 2002.

[83] Alvaro E. Monge and Charles Elkan. The Field Matching Problem: Algorithms and Applications. In *Knowledge Discovery and Data Mining*, pages 267–270, 1996.

[84] V. Ng and C. Cardie. Improving machine learning approaches to coreference resolution. In *ACL*, 2002.

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

[86] K. Nigam, A.K. McCallum, S. Thrun, and T.M. Mitchell. Learning to classify text from labeled and unlabeled documents. In *Procs. of the 15th National Conference on Artificial Intelligence (AAAI-98)*, pages 792–799, 1998.

[87] P. Pantel and D. Lin. Discovering Word Senses from Text. In *Proceedings of ACM Conference on Knowledge Discovery and Data Mining*, 2002.

[88] M. Pasca, D. Lin, J. Bigham, A. Lifchits, and A. Jain. Names and similarities on the web: Fact extraction in the fast lane. In *(To appear) Proc. of ACL/COLING 2006*, 2006.

[89] M. Pennacchiotti and P. Pantel. A Bootstrapping Algorithm for Automatically Harvesting Semantic Relations. In *Proceedings of Inference in Computational Semantics*, pages 87–96, 2006.

[90] B. Pentney, A. Popescu, S. Wang, M. Philipose, and H. Kautz. Sensor-based understanding of daily life via large-scale use of commonsense. In *Proceedings of AAAI*, 2006.

[91] Fernando Pereira, Naftali Tishby, and Lillian Lee. Distributional clustering of English words. In *Proceedings of the 31st ACL*, 1993.

[92] Ana-Maria Popescu. *Information Extraction from Unstructured Web Text*. PhD thesis, University of Washington, 2007.

[93] Ana-Maria Popescu, Alex Armanasu, Oren Etzioni, David Ko, and Alexander Yates. Modern natural language interfaces to databases: Composing statistical parsing with semantic tractability. In *Proceedings of the 20th International Conference on Computational Linguistics*, 2004.

[94] A. Ratnaparkhi. *Maximum Entropy Models for Natural Language Ambiguity Resolution*. PhD thesis, University of Pennsylvania, 1998.

[95] Adwait Ratnaparkhi. A Maximum Entropy Part-Of-Speech Tagger. In *Proceedings of the Empirical Methods in Natural Language Processing*, 1996.

[96] 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.

[97] P. Ravikumar and W. W. Cohen. A hierarchical graphical model for record linkage. In *UAI*, 2004.

[98] E. Riloff. Automatically constructing extraction patterns from untagged text. In *Procs. of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, pages 1044–1049, 1996.

[99] 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.

[100] B. Rosario and M. Hearst. Classifying semantic relations in bioscience text. In *Proc. of ACL*, 2004.

[101] G. Salton and M.J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, 1983.

[102] L. Schubert. Can we derive general world knowledge from texts. In *Procs. of Human Language Technology Conference*, 2002.

[103] S. Sekine. On-demand information extraction. In *Procs. of COLING*, 2006.

[104] Y. Shinyama and S. Sekine. Preemptive information extraction using unrestricted relation discovery. In *Proceedings of the HLT-NAACL*, 2006.

[105] P. Singla and P. Domingos. Entity Resolution with Markov Logic. In *ICDM*, 2006.

[106] Ravi Sinha and Rada Mihalcea. Unsupervised graph-based word sense disambiguation using measures of word semantic similarity. In *Proceedings of the IEEE International Conference on Semantic Computing (ICSC 2007)*, 2007.

[107] R. Snow, D. Jurafsky, and A. Y. Ng. Semantic taxonomy induction from heterogenous evidence. In *COLING/ACL*, 2006.

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

[109] D. G. Stork. The OpenMind Initiative. *IEEE Expert Systems and Their Applications*, 14(3):19–20, 1999.

[110] D. G. Stork. Open data collection for training intelligent software in the Open Mind initiative. In *Proceedings of Engineering Intelligent Systems*, 2000.

[111] Fabian M. Suchanek, Gjergji Kasneci, and Gerhard Weikum. Yago: A core of semantic knowledge. In *Proceedings of the 16th International Conference on the World Wide Web (WWW)*, 2007.

[112] K. Toutanova, C. D. Manning, D. Flickinger, and S. Oepen. Stochastic HPSG parse disambiguation using the Redwoods Corpus. *Journal of Logic and Computation*, 2005.

[113] P. Turney. Inference of Semantic Orientation from Association. In *CoRR cs. CL/0309034*, 2003.

[114] P. D. Turney. Mining the Web for Synonyms: PMI–IR versus LSA on TOEFL. *Lecture Notes in Computer Science*, 2167:491–502, 2001.

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

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

[117] 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.

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

[119] M. Volk. Exploiting the WWW as a corpus to resolve PP attachment ambiguities. In *Corpus Linguistics*, 2001.

[120] W.E. Winkler. The state of record linkage and current research problems. Technical report, U.S. Bureau of the Census, Washington, D.C., 1999.

[121] A. Yates, S. Schoenmackers, and Oren Etzioni. Detecting parser errors using web-based semantic filters. In *Proceedings of EMNLP*, 2006.

[122] Alexander Yates. Unsupervised resolution of objects and relations on the web. In *Proceedings of HLT-NAACL*, 2007.

[123] Alexander Yates, Michele Banko, Michael J Cafarella, Stephen Soderland, Matt Broadhead, and Oren Etzioni. TextRunner: Open Information Extraction on the Web. In *Proceedings of the NAACL HLT Demonstrations Program*, 2007.

[124] Alexander Yates, Oren Etzioni, and Daniel Weld. A reliable natural language interface to household appliances. In *Proceedings of the International Conference on Intelligent User Interfaces*, 2003.

[125] P. Ye and T. Baldwin. Verb Sense Disambiguation Using Selectional Preferences Extracted With A State-of-the-art Semantic Role Labeler. In *Proceedings of the 2006 Australasian Language Technology Workshop*, 2006.

Appendix A

DERIVATION OF THE EXTRACTED SHARED PROPERTY MODEL

The Extracted Shared Property (ESP) Model is introduced in Chapter 4. It is a method for calculating the probability that two strings are synonymous, given that they share a certain number of extractions in a data set. This appendix gives a derivation of the model.

Let s_i and s_j be two strings, each with a set of extracted properties E_i and E_j. Let U_i and U_j be the set of potential properties for each string, contained in their respective urns. Let $S_{i,j}$ be the number of properties shared between the two urns, or $|U_i \cap U_j|$. Let $R_{i,j}$ be the random variable for the synonymy relationship between s_i and s_j, with $R_{i,j} = R^t_{i,j}$ denoting the event that they are, and $R^f_{i,j}$ that they are not. The ESP model states the following:

Proposition 2 *If two strings s_i and s_j have $|U_i| = P_i$ and $|U_j| = P_j$ potential properties (or instances), with $min(P_i, P_j) = P_{min}$; and they appear in extracted assertions E_i and E_j such that $|E_i| = n_i$ and $|E_j| = n_j$; and they share k extracted properties (or instances), the probability that s_i and s_j co-refer is:*

$$P(R^t_{i,j}|E_i, E_j, P_i, P_j) = \frac{\binom{P_{min}}{k} \sum_{r,s\geq 0} \binom{S_{i,j}-k}{r+s} \binom{r+s}{r} \binom{P_i - P_{min}}{n_i-(k+r)} \binom{P_j - P_{min}}{n_j-(k+s)}}{\sum_{k\leq S_{i,j}\leq P_{\min}} \binom{S_{i,j}}{k} \sum_{r,s\geq 0} \binom{S_{i,j}-k}{r+s} \binom{r+s}{r} \binom{P_i - S_{i,j}}{n_i-(k+r)} \binom{P_j - S_{i,j}}{n_j-(k+s)}} \quad \text{(A.1)}$$

The ESP model makes several simplifying assumptions:

1. Balls are drawn from the urns without replacement.

2. Draws from one urn are independent of draws from any other urn.

3. Each ball for a string is equally likely to be selected from its urn: if $U = \{u_1, \ldots, u_m\}$ and X denotes a random draw from U, $P(X = u_i) = \frac{1}{|U|}$ for every u_i.

4. The prior probability for $S_{i,j}$, given the number of properties in U_i and U_j, is uniform: $\forall_{0 \leq s \leq \min(P_i,P_j)} P(S_{i,j} = s | P_i, P_j) = \frac{1}{\min(P_i,P_j)+1}$

5. Given extracted properties for two strings and the number of potential properties for each, the probability of synonymy depends only on the number of extracted properties for each, and the number of shared properties in the extractions: $P(R^t_{i,j} | E_i, E_j, P_i, P_j) = P(R^t_{i,j} | k, n_i, n_j, P_i, P_j)$.

6. Two strings are synonymous if and only if they share as many potential properties as possible: $R^t_{i,j} \equiv (|U_i \cap U_j| = \min(P_i, P_j))$.

.

Before proving Proposition 2, we prove a simple property of urns under the assumptions above.

Lemma 1 *Given n draws without replacement from an urn containing a set of properties U, the probability of selecting a particular set $S \subset U$ is $\frac{1}{\binom{|U|}{|S|}}$ if $|S| = n$, and zero otherwise.*

Proof of Lemma 1: Let $U = \{u_1, \ldots, u_m\}$ denote the elements of U, and let $X_1, \ldots, X_n$ denote the independent draws from the urn. If $n = 1$, then $P(S = \{u_i\}) = P(X_1 = u_i) = \frac{1}{|U|}$ by assumption 3 above. Now suppose that $n = n_0$, and that the lemma holds for every $n' < n_0$.

$$\begin{aligned}
P(S = \{x_1, \ldots, x_{n_0} | x_i \in U\}) &= \sum_i P(S^{n_0-1} = \{x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_{n_0}\}) P(X_n = x_i) \\
&= \sum_i \frac{1}{\binom{|U|}{n_0-1}} \frac{1}{|U| - n_0 + 1} \\
&= \sum_i \frac{(n_0-1)!(|U| - n_0 + 1)!}{|U|!} \frac{1}{|U| - n_0 + 1} \\
&= \frac{n_0(n_0-1)!(|U| - n_0 + 1)(|U| - n_0)!}{|U|!(|U| - n_0 + 1)} \\
&= \frac{n_0!(|U| - n_0)!}{|U|!} \\
&= \frac{1}{\binom{|U|}{n_0}}
\end{aligned}$$

□

Proof of Proposition 2: We begin by transforming the desired expression, $P(R^t_{i,j}|E_i, E_j, P_i, P_j)$, into something that can be derived from the urn model. By assumptions 5 and 6, we get

$$P(R^t_{i,j}|E_i, E_j, P_i, P_j) = P(S_{i,j} = P_{min}|k, n_i, n_j, P_i, P_j) \tag{A.2}$$

Then, by applying Bayes Rule, we get

$$P(S_{i,j} = P_{min}|k, n_i, n_j, P_i, P_j) = \frac{P(k|S_{i,j} = P_{min}, n_i, n_j, P_i, P_j)P(S_{i,j} = P_{min}|n_i, n_j, P_i, P_j)}{\sum_{k \le S_{i,j} \le P_{min}} P(k|n_i, n_j, P_i, P_j)P(S_{i,j}|n_i, n_j, P_i, P_j)} \tag{A.3}$$

Since we have assumed a uniform prior for $S_{i,j}$ (assumption 4), the prior terms vanish, leaving

$$P(R^t_{i,j}|E_i, E_j, P_i, P_j) = \frac{P(k|S_{i,j} = P_{min}, n_i, n_j, P_i, P_j)}{\sum_{k \le S_{i,j} \le P_{min}} P(k|n_i, n_j, P_i, P_j)} \tag{A.4}$$

The second step of the derivation is to find a suitable expression for $P(k|S_{i,j}, n_i, n_j, P_i, P_j)$. The probability can be written out fully as:

$$P(k|S_{i,j}, n_i, n_j, P_i, P_j) = \frac{\sum_{\substack{E_i \subset U_i : |E_i| = n_i \\ E_j \subset U_j : |E_j| = n_j \\ |E_i \cap E_j| = k}} P(E_i, E_j|S_{i,j}, n_i, n_j, P_i, P_j)}{\sum_{\substack{E_i \subset U_i : |E_i| = n_i \\ E_j \subset U_j : |E_j| = n_j}} P(E_i, E_j|S_{i,j}, n_i, n_j, P_i, P_j)} \tag{A.5}$$

By assumption 2, $P(E_i, E_j) = P(E_i)P(E_j)$. By Lemma 1, all $P(E_i)$ terms are equal, since they are all sets of size n_i, and likewise for $P(E_j)$ terms. Thus, to get the desired probability expression, we simply need to count the number of ways of taking subsets from the two urns such that they share k properties.

$$P(k|S_{i,j}, n_i, n_j, P_i, P_j) = \frac{\sum_{\substack{E_i \subset U_i : |E_i| = n_i \\ E_j \subset U_j : |E_j| = n_j \\ |E_i \cap E_j| = k}} 1}{\sum_{\substack{E_i \subset U_i : |E_i| = n_i \\ E_j \subset U_j : |E_j| = n_j}} 1} \tag{A.6}$$

$$= \frac{\text{Count}(k, n_i, n_j|S_{i,j}, P_i, P_j)}{\text{Count}(n_i, n_j|S_{i,j}, P_i, P_j)} \tag{A.7}$$

There are $\binom{P_i}{n_i}$ ways of picking each set E_i, so

$$\text{Count}(n_i, n_j | S_{i,j}, P_i, P_j) = \binom{P_i}{n_i}\binom{P_j}{n_j} \tag{A.8}$$

To complete the derivation, we need an expression for $\text{Count}(k, n_i, n_j | S_{i,j}, P_i, P_j)$. This involves splitting the relevant sets into several parts. First U_i and U_j each contain some shared and unshared properties. Let $T_{i,j} = U_i \cap U_j$, $V_i = U_i - T_{i,j}$, and $V_j = U_j - T_{i,j}$. Second, the selected sets from each urn, E_i and E_j, each have properties that come from the set of shared properties and the set of unshared properties. Let $K = E_i \cap E_j$, $F_i = (E_i \cap T_{i,j}) - K$, and $F_j = (E_j \cap T_{i,j}) - K$.

With these sets defined, each set E_i and E_j is composed of three distinct subsets: the shared subset (K); a subset also selected from the shared potential properties, $T_{i,j}$, but which is not shared (F_i and F_j); and the remaining elements, which are chosen from the complements of the shared properties (V_i and V_j). Since the subsets are distinct, we can count them separately and multiply the results to arrive at the final count.

The number of ways of selecting the shared subset is clearly $\binom{S_{i,j}}{k}$. The sizes of F_i and F_j are unknown, however, so we must sum over all possibilities. Let $r = |F_i|$, and $s = |F_j|$. There are $S_{i,j} - k$ remaining shared potential properties in $T_{i,j}$ from which to choose the $r + s$ elements of F_i and F_j, and then $\binom{r+s}{s}$ ways to split the two into distinct subsets. There are $n_i - (k + r)$ elements left to choose in E_i, and $n_j - (k + s)$ elements left to choose in E_j. These must be selected from the unshared potential properties in V_i and V_j, which have sizes $P_i - S_{i,j}$ and $P_j - S_{i,j}$ respectively. Putting these pieces together, we have

$$\text{Count}(k, n_i, n_j | S_{i,j}, P_i, P_j) = \binom{S_{i,j}}{k} \sum_{r,s} \binom{S_{i,j} - k}{r + s}\binom{r + s}{s}\binom{P_i - S_{i,j}}{n_i - (k + r)}\binom{P_j - S_{i,j}}{n_j - (k + s)} \tag{A.9}$$

The ranges for r and s are somewhat involved. They must obey the following constraints:

1. $r, s \geq 0$

2. $r \geq n_i - k - P_i + S_{i,j}$

3. $s \geq n_j - k - P_j + S_{i,j}$

4. $r \leq n_i - k$

5. $s \leq n_j - k$

6. $r + s \leq S_{i,j} - k$

Plugging Equation A.9 into Equation A.7, and that in turn into Equation A.4 yields the desired result.□

Appendix B

FAST CALCULATION OF THE EXTRACTED SHARED PROPERTY MODEL

The ESP model can be expensive to calculate if done the wrong way. We use two techniques to speed up the calculation immensely. For reference, the full formulation of the model is:

$$P(R^t_{i,j}|k,n_i,n_j,P_i,P_j) = \frac{\binom{P_{min}}{k}\sum_{r,s\geq 0}\binom{S_{i,j}-k}{r+s}\binom{r+s}{r}\binom{P_i-P_{min}}{n_i-(k+r)}\binom{P_j-P_{min}}{n_j-(k+s)}}{\sum_{k\leq S_{i,j}\leq P_{\min}}\binom{S_{i,j}}{k}\sum_{r,s\geq 0}\binom{S_{i,j}-k}{r+s}\binom{r+s}{r}\binom{P_i-S_{i,j}}{n_i-(k+r)}\binom{P_j-S_{i,j}}{n_j-(k+s)}} \tag{B.1}$$

Note that the equation involves three sums, ranging over $O(P_{min})$, $O(n_i)$, and $O(n_j)$ values respectively. In effect, this is $O(n^3)$ in the number of extractions for a string. Furthermore, each step requires the expensive operation of calculating binomial coefficients. Fortunately, there are several easy ways to drastically speed up this calculation.

First, Stirling's approximation can be used to calculate factorials (and therefore the binomial function). Stirling's approximation is given by:

$$n! \approx \sqrt{\pi\left(2n+\frac{1}{3}\right)}\left(\frac{n^n}{e^n}\right)$$

To avoid underflow and overflow errors, log probabilities are used everywhere possible. This calculation can then be done using a few simple multiplications and logarithm calculations. Stirling's formula converges to $n!$ like $O(\frac{1}{n})$; in practice it proved to be accurate enough of an approximation of $n!$ for $n > 100$. In ESP's implementation, all other values of $n!$ are calculated once, and stored for future use.

Second, the calculation of $P(k|n_1,n_2,P_1,P_2)$ can be sped up by simplifying the expression to get rid of two of the sums. The result is the following equivalent expression, assuming without loss of generality that $P_2 \leq P_1$:

$$P(k|n_1,n_2,P_1,P_2) = \frac{\binom{P_2+1}{n_2+1}\sum_{r=k}^{n_2}\binom{r}{k}\binom{P_1-r}{n_1-k}}{\binom{P_2}{n_2}\binom{P_1}{n_1}} \tag{B.2}$$

This simplification removes two of the sums, and therefore changes the complexity of calculating ESP from $O(P_2 n_2 n_1)$ to $O(n_2)$. This was sufficient for our data set, but on larger data sets it might be necessary to introduce sampling techniques to improve the efficiency even further.

Appendix C

A BETTER BOUND ON THE NUMBER OF COMPARISONS MADE BY THE RESOLVER CLUSTERING ALGORITHM

This appendix provides a proof that the RESOLVER clustering algorithm has better worst-case performance on textual data than previous methods. Chapter 4 showed that the RESOLVER clustering algorithm initially makes $O(N \log N)$ comparisons between strings in the data, where N is the number of extractions. Heuristic methods like the Canopies method [77] require $O(M^2)$ comparisons, where M is the number of distinct strings in the data. The claim is that $O(N \log N)$ is asymptotically better than $O(M^2)$ for Zipf-distributed data.

Zipf-distributed data is controlled by a shape parameter, which we call z. The claim above holds true for any shape parameter $z < 2$, as shown below. Fortunately, in natural data the shape parameter is usually very close to $z = 1$, and in RESOLVER data it was observed to be $z < 1$.

Let S be the set of distinct strings in a set of extractions D. For each $s \in S$, let $\text{freq}(s)$ denote the number of times that s appears in the extractions. Thus $|D| = \sum_{s \in S} \text{freq}(s)$. Let $M = |S|$ and $N = |D|$.

Proposition 3 *If S has an observed Zipf distribution with shape parameter z, then*

1. *if $z < 1$, $N = \Theta(M)$*

2. *if $z = 1$, $N = \Theta(M \ log \ M)$*

3. *if $z > 1$, $N = \Theta(M^z)$*

Proof: Let $s_1, \ldots, s_M$ be the elements of S in rank order from highest frequency string (s_1) to lowest frequency string (s_M). Since S has an observed Zipf distribution with shape parameter z, $\text{freq}(s_i) = \frac{M^z}{i^z}$. Given the assumptions, z and M determine the number of

extractions made:

$$N_{M,z} = \sum_{s \in S} \text{freq}(s) \tag{C.1}$$

$$= \sum_{1 \le i \le M} \frac{M^z}{i^z} \tag{C.2}$$

We can build a recurrence relation for the value of N as M changes (holding z constant) by noting that

$$N_{2M,z} = \sum_{1 \le i \le 2M} \frac{(2M)^z}{i^z} \tag{C.3}$$

$$= (2M)^z \sum_{1 \le i \le M} \frac{1}{i^z} + (2M)^z \sum_{M+1 \le i \le 2M} \frac{1}{i^z} \tag{C.4}$$

$$= 2^z N_{M,z} + f_z(M) \tag{C.5}$$

where $f_z(M) = \sum_{M+1 \le i \le 2M} \frac{(2M)^z}{i^z}$.

There are three important properties of $f_z(M)$.

1. Note that every term in the sum for $f_z(M)$ is less than $\frac{(2M)^z}{M^z} = 2^z$. Thus $f_z(M)$ is bounded above by $2^z \cdot M$, so if z is held constant, $f_z(M) = O(M)$.

2. Every term in the sum is at least 1, so $f_z(M) \ge M$ and $f_z(M) = \Omega(M)$; combining these two facts yields $f_z(M) = \Theta(M)$.

3. Let $c = 2^{z-1}$.

$$\begin{aligned} 2^z f_z(M/2) &\le 2^z M/2 \\ &= 2^{z-1} M \\ &= cM \\ &\le c f_z(M) \end{aligned}$$

   Furthermore, $c < 1$ whenever $z < 1$.

These three properties of $f_z(M)$ will be used below.

We can now use the recurrence relation and the Master Recurrence Theorem [30] to prove the three claims of the proposition. For reference, the Master Recurrence Theorem states the following:

Theorem 1 *Let $a \geq 1$ and $b \geq 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the non-negative integers by the recurrence*

$$T(n) = aT(n/b) + f(n)$$

Then T(n) can be bounded asymptotically as follows.

1. *If $f(n) = O(n^{log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{log_b a})$*

2. *If $f(n) = \Theta(n^{log_b a})$, then $T(n) = \Theta(n^{log_b a} log\ n)$*

3. *If $f(n) = \Omega(n^{log_b a + \epsilon})$, for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.*

First consider the case where $z > 1$. The recurrence for $N_{M,z}$ can clearly be made to fit the form for Theorem 1 by setting $a = 2^z$, $b = 2$, and $f = f_z(M)$. Since $f_z(M)$ is bounded above by $2^z \cdot M = O(M)$, it is also clearly bounded above by $O(M^{\log_b a - \epsilon}) = O(M^{z-\epsilon})$, where we can take ϵ to be anything in $(0, z-1)$. Thus the case one of Theorem 1 applies, and $N_{M,z} = \Theta(M^{\log_b a}) = \Theta(M^z)$.

Next consider the case where $z = 1$. Since $f_{z=1}(M) = \Theta(M)$ and $\Theta(M^{\log_b a}) = \Theta(M)$, case two of Theorem 1 applies. Thus $N_{M,z=1} = \Theta(M^{\log_b a} \log\ M) = \Theta(M\ \log\ M)$.

Finally, consider the case where $z < 1$. $f_z(M)$ is bounded below by $\Omega(M^{\log_b a + \epsilon}) = \Omega(M^{z+\epsilon})$ for any ϵ in $(0, 1-z)$. This fact, together with property 3 of $f_z(M)$ allow us to apply the third case of the Theorem 1. Thus $N_{M,z} = \Theta(f_z(M)) = \Theta(M)$, and Proposition 3 is proved. □

The data used in RESOLVER experiments in Chapter 4 had a shape parameter $z < 1$, so the bound on the number of comparisons made was $O(N\ \log\ N) = O(M\ \log\ M)$. For $z = 1$, the bound would have been $O(N\ \log\ N) = O(M\ \log(M)\ \log(M\ \log\ M))$. Not until $z = 2$ would the asymptotic performance of $O(M^2\ \log\ M)$ have been worse than $O(M^2)$.

Appendix D

ADDITIONAL RESULTS FOR THE RESOLVER SYSTEM

Table D.1: **Area under the precision-recall curve for several similarity metrics on pairs of object strings.** The maximum possible area is less than one because many correct string pairs share no properties, and are therefore not compared by the clustering algorithm. The third column shows the score as a fraction of the maximum possible area under the curve, which for relations is 0.57. The improvement over baseline shows how much the ESP curves improve over DIRT, and how much RESOLVER improves over SSM.

Metric	AUC	Fraction of Max. AUC	Improvement over Baseline
Cosine Similarity Metric	0.0061	0.011	-21%
DIRT	0.0083	0.014	0%
ESP-10	0.019	0.033	136%
ESP-30	0.024	0.041	**193%**
ESP-50	0.022	0.037	164%
ESP-90	0.018	0.031	121%
String Similarity Metric	0.18	0.31	0%
RESOLVER	0.22	0.38	**23%**

Table D.2: **Area under the precision-recall curve for several similarity metrics on pairs of relation strings.** The maximum possible area is less than one because many correct string pairs share no properties, and are therefore not compared by the clustering algorithm. The third column shows the score as a fraction of the maximum possible area under the curve, which for relations is 0.094. The improvement over baseline shows how much the ESP curves improve over DIRT, and how much RESOLVER improves over SSM.

Metric	AUC	Fraction of Max. AUC	Improvement over Baseline
Cosine Similarity Metric	0.0035	0.034	-19%
DIRT	0.0044	0.042	0%
ESP-50	0.0048	0.046	9.5%
ESP-250	0.0087	0.083	98%
ESP-500	0.0096	0.093	121%
ESP-900	0.010	0.098	**133%**
String Similarity Metric	0.022	0.24	0%
RESOLVER	0.029	0.31	**31%**

Table D.3: **The precision and recall of ESP and ESP using Mutual Recursion for clustering objects on artificial data sets.**

		Double Repetition Probability		
	Repetition Probability	0.1	0.25	0.5
ESP Precision	0.05	0.83	0.76	0.62
ESP-MR Precision	0.05	0.79	0.73	0.62
ESP Recall	0.05	0.45	0.42	0.39
ESP-MR Recall	0.05	0.45	0.43	0.40
ESP Precision	0.1	0.99	0.98	0.86
ESP-MR Precision	0.1	0.98	0.97	0.92
ESP Recall	0.1	0.64	0.57	0.46
ESP-MR Recall	0.1	0.60	0.58	0.59
ESP Precision	0.25	1.0	1.0	1.0
ESP-MR Precision	0.25	1.0	1.0	0.99
ESP Recall	0.25	0.88	0.82	0.68
ESP-MR Recall	0.25	0.84	0.85	0.85

Table D.4: **The precision and recall of ESP and ESP using Mutual Recursion for clustering relations on artificial data sets.**

		Double Repetition Probability		
	Repetition Probability	0.1	0.25	0.5
ESP Precision	0.05	0.78	0.69	0.58
ESP-MR Precision	0.05	0.75	0.68	0.58
ESP Recall	0.05	0.52	0.47	0.41
ESP-MR Recall	0.05	0.52	0.47	0.41
ESP Precision	0.1	0.94	0.91	0.76
ESP-MR Precision	0.1	0.92	0.89	0.79
ESP Recall	0.1	0.72	0.66	0.52
ESP-MR Recall	0.1	0.72	0.69	0.62
ESP Precision	0.25	0.97	0.98	0.96
ESP-MR Precision	0.25	0.98	0.98	0.97
ESP Recall	0.25	0.94	0.90	0.77
ESP-MR Recall	0.25	0.95	0.92	0.91

VITA

Alexander Pieter Yates was born in Philadelphia, Pennsylvania, and currently lives in Seattle, Washington. In 2001 he received a B.A. in Applied Mathematics from Harvard University in Cambridge, Massachusetts. He received a Masters degree in 2003 from the Department of Computer Science and Engineering at the University of Washington in Seattle. In August, 2007 he earned a Doctor of Philosophy in Computer Science and Engineering from the University of Washington.