CHAPTER

Commonsense Reasoning Using Unstructured Information 18

Structured information is the information that computers can easily understand and reason with, such as algebraic expressions, logical formulas, frames, and database tables. Unstructured information is the information that is not represented in such a form and is difficult for computers to understand. Examples of unstructured information include natural language text, audio, images, and video. Because unstructured information is so easy for humans to create, it is abundant—much of the world's data is unstructured. Given that there is so much unstructured information available, the question arises of how we might be able to use it for automated commonsense reasoning.

In this chapter, we discuss the use of unstructured information, specifically natural language text, for commonsense reasoning. First, we describe attempts to use natural language as a programming language. We then review several restricted versions of English for reasoning, including reasoning in the event calculus. Then, we discuss automated reasoning directly using natural language. Next, we describe the Watson system for natural language question answering and the WatsonPaths system built on top of Watson, which use unstructured information as a source of knowledge for reasoning. Finally, we compare the various natural language reasoning systems and methods.

18.1 NATURAL LANGUAGE AS A PROGRAMMING LANGUAGE

In the 1960s, Jean Sammet, Mark Halpern, and others proposed to use natural language as a programming language. In the 1970s, several researchers started to build automatic programming systems that take a natural language specification of a problem as input and produce a computer program as output. We discuss two representative systems, SAFE and OWL.

18.1.1 SAFE

In the mid-1970s, Robert Balzer, Neil Goldman, and David Wile developed SAFE, a system that handles three texts taken from specification manuals. SAFE was a research system developed to demonstrate the feasibility of automatic programming.

The goal was to take a specification and automatically turn it into an executable computer program, eliminating the need for manual computer programming.

SAFE operates as follows:

Input: bracketed natural language specification ⟶
resolve imprecisions in input ⟶
Output: executable computer program

An example of an imprecision is provided by the following bracketed input sentence:

```
((THE MESSAGE) (IS DISTRIBUTED) TO (EACH ((ASSIGNED)) OFFICE))
```

Because offices can be assigned to both messages and keys, `(EACH ((ASSIGNED)) OFFICE)` is ambiguous. But earlier in the specification, messages were assigned for distribution, and distribution requires a message and an office. SAFE infers that a message is distributed to every office assigned to the message, and it produces the following code as output:

```
(FOR ALL (offices assigned TO message FOR ANYTHING)
  (distribute-process#2 message office))
```

18.1.2 OWL

Also in the mid-1970s, Peter Szolovits, Lowell B. Hawkinson, and William A. Martin developed a knowledge representation language based on English called OWL (which is different from the later web ontology language OWL). In OWL, "check sensitivity due to thyroid function" is represented as

```
(CHECK (SENSITIVITY (DUE (TO THYROID-FUNCTION))))
```

The SAFE program previously described is used to convert a specification into a computer program, which can then be run. In contrast, OWL directly executes specifications stored in the knowledge base. The OWL interpreter operates as follows:

Input: bracketed English description of desired procedure ⟶
match input to method in knowledge base ⟶
recursively invoke interpreter for unsatisfied preconditions ⟶
execute method and create execution trace

OWL also generates English explanations of methods in the knowledge base and execution traces.

18.2 REASONING WITH RESTRICTED NATURAL LANGUAGE

The attempts to use natural language for programming ran into difficulty because natural language is very complex. One response to the complexity of natural language is to simplify or restrict it. Several researchers have introduced restricted versions of English and procedures for reasoning using them. We review some representative examples.

18.2.1 MONTAGOVIAN SYNTAX

In the late 1980s, David A. McAllester and Robert Givan introduced Montagovian syntax, inspired by Richard Montague's logical approach to natural language syntax and semantics. Montagovian syntax is both a restricted form of English and an alternative syntax for first-order logic.

Here are some sample expressions using this syntax:

```
(every police-officer (owns (some car)))
(every (child-of (some bird)) (friend-of (every bird-watcher)))
```

The meaning of these expressions is similar to their meaning in English. The symbols `police-officer`, `car`, `bird`, and `bird-watcher` are *class symbols*, which denote sets. The symbols `owns`, `child-of`, and `friend-of` are *binary relation symbols*, which denote binary relations.

Montagovian syntax is defined as follows:

Definition 18.1. A ***class expression*** is defined inductively as follows:

- A constant is a class expression.
- A class symbol is a class expression.
- A variable is a class expression.
- If ρ is a binary relation symbol and σ is a class expression, then `(`ρ `(some` σ`))` and `(`ρ `(every` σ`))` are class expressions.
- If ν is a variable and α is a formula, then `(`$\lambda\ \nu\ \alpha$`)` is a class expression.
- Nothing else is a class expression.

Definition 18.2. An ***atomic formula*** is `(some` $\sigma_1\ \sigma_2$`)` or `(every` $\sigma_1\ \sigma_2$`)`, where σ_1 and σ_2 are class expressions.

Definition 18.3. A ***formula*** is defined inductively as follows:

- An atomic formula is a formula.
- If α is a formula, then $\neg\alpha$ is a formula.
- If α and β are formulas, then $\alpha \wedge \beta$ and $\alpha \vee \beta$ are formulas.
- Nothing else is a formula.

Definition 18.4. A ***literal*** is an atomic formula or the negation of an atomic formula.

Constants and variables denote singleton sets. We use several abbreviations:

- If τ is a constant or variable and σ is a class expression, then the atomic formulas `(some` $\tau\ \sigma$`)` and `(every` $\tau\ \sigma$`)` are equivalent, and we use `(`$\tau\ \sigma$`)` as an abbreviation for either atomic formula.
- If τ is a constant or variable and ρ is a binary relation, then the class expressions `(`ρ `(some` τ`))` and `(`ρ `(every` τ`))` are equivalent, and we use `(`$\rho\ \tau$`)` as an abbreviation for either class expression.

- If σ is a class expression, then we use $\exists\sigma$ as an abbreviation for the atomic formula `(some` σ σ `)`.

An example of the use of λ is `(λ x (Yhazi (bought x)))`, which denotes the set of all things bought by Yhazi. This is an abbreviation of `(λ x (some (Yhazi (bought (some x)))))`.

We reason with Montagovian syntax using inference rules such as the following:

$$\frac{(\texttt{every}\ \sigma_1\ \sigma_2) \quad (\texttt{every}\ \sigma_2\ \sigma_3)}{(\texttt{every}\ \sigma_1\ \sigma_3)} \tag{18.1}$$

$$\frac{(\texttt{every}\ \sigma_1\ \sigma_3) \quad (\texttt{every}\ \sigma_1\ \sigma_4) \quad \exists\sigma_1}{(\texttt{some}\ \sigma_3\ \sigma_4)} \tag{18.2}$$

Thus, from

```
(every vermouth wine)
(every wine alcoholic-beverage)
(every vermouth before-dinner-drink)
∃vermouth
```

we can infer

```
(some alcoholic-beverage before-dinner-drink)
```

Namely, from `(every vermouth wine)`, `(every wine alcoholic-beverage)`, and (18.1), we infer `(every vermouth alcoholic-beverage)`. From this, `(every vermouth before-dinner-drink)`, `∃vermouth`, and (18.2), we infer `(some alcoholic-beverage before-dinner-drink)`.

McAllester and Givan define a polynomial time decision procedure for the satisfiability of a set Σ of literals, provided that Σ satisfies the following properties:

- the literals in Σ do not contain λ
- for every class expression σ that occurs in Σ, we can determine from Σ whether σ denotes the empty set

The decision procedure makes use of inference rules (18.1), (18.2), and others.

The use of Montagovian syntax for reasoning can be summarized as follows:

Input: Montagovian syntax literals satisfying properties $\longrightarrow$
polynomial time decision procedure for satisfiability $\longrightarrow$
Output: **yes** or **no**

18.2.2 ATTEMPTO CONTROLLED ENGLISH

In the late 1990s, Norbert E. Fuchs, Uta Schwertel, and Rolf Schwitter introduced Attempto Controlled English (ACE), a restricted form of English designed to support the creation of precise technical specifications. The vocabulary of ACE consists of predefined function words and user-defined words. An ACE sentence is of the form

> *subject* + *verb* + *complements* + *adjuncts*

Complex sentences can be created using modifiers, coordination of sentences and phrases, subordinate clauses, prepositional phrases, quantification, and negation.

Examples of ACE sentences are the following:

```
The new user enters an initial PIN.
The user enters a digit or a letter.
A customer of the bank gives the card to the teller.
John does not enter a valid code in the morning.
```

Using the Prolog language, ACE parses an input text and converts it into a representation called a discourse representation structure. ACE uses a set of interpretation principles to resolve ambiguity. For example, a pronoun always refers to the most recent noun phrase that agrees in number and gender, and a prepositional phrase always modifies the verb. If the user is not satisfied with the interpretation of a sentence, the user must rephrase it.

The Attempto reasoner can be used to prove that one ACE text logically follows from another ACE text. The reasoner operates by converting discourse representation structures into first-order logic and then invoking a first-order logic theorem prover. The operation of ACE can be summarized as follows:

> *Input*: ACE texts $\longrightarrow$
> parse and translate into discourse representation structures $\longrightarrow$
> convert to first-order logic $\longrightarrow$
> invoke first-order theorem prover $\longrightarrow$
> *Output*: summary of results

18.2.3 PENG LIGHT AND THE EVENT CALCULUS

Rolf Schwitter describes how the PENG Light controlled natural language, which is similar to ACE, can be used to specify and reason with event calculus domain descriptions. The PENG Light language is designed so that it can be translated unambiguously into formal representation structures.

Here are examples of effect axioms expressed in PENG Light:

```
If an agent walks into a room then the agent will be in the room.
If a device is running and an agent turns off the device then the
device will no longer be running.
```

Here is an example of a narrative and observations expressed in PENG Light:

```
The dishwasher is running.
Mason walks into the kitchen.
Mason turns off the dishwasher.
```

PENG Light is used for event calculus reasoning as follows:

Input: PENG Light event calculus domain description ⟶
parse and translate into discourse representation structures ⟶
translate into Prolog rules and facts ⟶
reason using Prolog

The following Prolog rule is used to infer an event calculus event occurrence from a discourse representation structure event:

```
happens(E, T) :- event(E, Type), theta(E, time, S), timex(S, T).
```

18.3 REASONING DIRECTLY WITH NATURAL LANGUAGE

Most attempts at performing automated reasoning using natural language involve first converting natural language into a formal representation (such as through parsing). Another possibility is to reason directly using natural language without performing this conversion. In 2002, Push Singh introduced a prototype of such a system, called REFORMULATOR. This system applies *reformulation rules* to sets of English sentences to infer new English sentences. An example of a reformulation rule is as follows:

```
If the agent ?V ?NP, then ?NP will be ?AdjP
The agent ?V ?NP →
?NP is ?AdjP
```

NP is a noun phrase, V is a verb, and AdjP is an adjective phrase. Then, given the sentences

```
If the agent pushes the door, then the door will be open.
The agent pushes the door.
```

The system applies the reformulation rule to infer the new sentence

```
The door is open.
```

REFORMULATOR operates as follows:

Input: English sentences, reformulation rules ⟶
repeatedly apply reformulation rules ⟶
Output: inferences

So far, we have discussed several systems that work with tightly controlled forms of natural language. SAFE, OWL, ACE, and PENG Light focus on the creation and execution of specifications. Montagovian syntax, ACE, PENG Light, and REFORMULATOR focus on reasoning. Watson differs from all these systems because it works with arbitrary, unrestricted natural language text.

18.4 WATSON

The Watson question answering system, developed by a team at IBM Research led by David Ferrucci, uses a large corpus of natural language text to provide precise answers to input questions. Unlike a search engine, which requires the user to locate the answer to a given question in a document, Watson picks out and returns the exact answer. We discuss Watson and WatsonPaths, which enables reasoning using Watson.

18.4.1 WATSON ARCHITECTURE

Watson is built using the Unstructured Information Management Architecture (UIMA), which allows us to define *pipelines*, each of which consists of a sequence of *components* and possibly other pipelines. Watson consists of many components that work together to process an input question.

Watson's high performance results from the use of multiple components operating over unstructured information and the merging and ranking of the results of these components using machine learning. If one component is unable to handle a particular input correctly, there is often another component that is able to. Components are also able to check the work of other components, which leads to better performance than could be achieved if all the components worked independently. Using multiple components also allows many people to develop the system together. To extend Watson to perform better on a certain class of problem, we can add a component designed to improve the system's performance on that class of problem.

A UIMA component has an `initialize` method that is invoked once when an instance of the component class is first created and a `process` method that is invoked when needed to process a new item.

UIMA components take their input from and write their output to a shared data structure called a *common analysis structure* (CAS). A CAS consists of text along with *annotations* of text spans. An annotation consists of a begin index, end index, entity type, covered text, and arbitrary properties and values.

The Watson architecture is a pipeline architecture in which processing proceeds in a single direction from one component to the next. There is also, however, a high degree of fan-out, so that one component may feed its output to a number of components each of which may in turn feed its output to a number of components. Watson is CPU-intensive and data-intensive. UIMA allows Watson to be scaled out across as many machines as necessary to achieve the desired throughput and response time.

A high-level view of the operation of Watson is as follows:

Input: question $\longrightarrow$
analyze question $\longrightarrow$
search for documents/passages and look up answers $\longrightarrow$
generate candidates $\longrightarrow$
search for supporting passages $\longrightarrow$

score passages and answers ⟶
merge equivalent answers ⟶
calculate final scores for answers ⟶
Output: answers with scores

18.4.2 QUESTION ANALYSIS

The first step in processing an input question is *question analysis*. This consists of parsing the question, identifying entities and relation instances in the question, resolving coreferences in the question, and identifying key elements of the question. Given the question

```
What disease causes right upper quadrant pain?
```

question analysis identifies several key elements.

The question's *lexical answer type* (LAT) is `disease`. The LAT specifies the desired type of an answer to the question. It is later used by the type coercion components. LATs with modifiers, such as `heart condition` and `skeletal finding`, are also identified.

The question's *focus* is `What disease`. The focus is the part of the question that is to be replaced by the answer. If we replace the focus with the correct answer, then the result should be a true statement.

```
Cholecystitis causes right upper quadrant pain.
```

The focus is later used by context-dependent answer scoring components. The focus is identified by looking for patterns such as a noun phrase starting with an interrogative determiner (`what` or `which`). The basic algorithm for detecting the LAT is to use the headword of the focus.

The *question class* is `FACTOID:DiseaseGivenFinding`. The question class can modify which components are used to process the question and which machine learning models are used.

Watson uses the English Slot Grammar (ESG) parser and other recognizers and transformations to generate a dependency parse tree and predicate argument structure. ESG produces the following output for the example question:

```
-----------------------------------------------------------
  .----- ndet    what1(1)                           det
.------- subj(n) disease1(2)                        noun
o------- top     cause1(3,2,203,u)                  verb
‘------- obj(n)  right upper quadrant pain(203)     noun
  |    .- adjpre  right3(4)                          qual
  | .--- nadj    upper1(5,7)                        adj
  | .--- nnoun   quadrant1(6)                       noun
  ‘----- chsl(n) pain1(7,u)                         noun
-----------------------------------------------------------
```

The top-level node of the dependency parse tree is `cause`. The subject of `cause` is `what disease`, and the object of `cause` is `right upper quadrant pain`.

This example was parsed using a *chunk lexicon* for the medical domain based on the Unified Medical Language System (UMLS) (United States National Library of Medicine, 2014). ESG provides an analysis of the structure of the chunk `right upper quadrant pain`.

Relation extraction identifies the following relation instances in the example question:

```
causes(disease : focus, right upper quadrant pain)
findingOf(right upper quadrant pain, disease : focus)
```

A relation instance consists of a *relation name* such as `causes` and `findingOf` and one or more *arguments* such as `disease` and `right upper quadrant pain`.

18.4.3 PRIMARY SEARCH

After question analysis, the next step in the Watson pipeline is *primary search*. The goal of this step is to locate content relevant to answering the question. This involves converting a question into search engine queries and issuing these queries to search engines. Searches are performed both to retrieve *documents* and to retrieve *passages* consisting of one or more sentences. Search engines produce ranked lists of documents or passages and associated scores in response to a query.

To increase diversity, Watson uses two search engines: Indri (Lemur Project, 2014) and Lucene (Apache, 2014). Using both Indri and Lucene improves performance over using either search engine alone. Indri's existing passage retrieval capability is used. A two-phase mechanism for passage retrieval is implemented using Lucene: First, documents are retrieved using Lucene. Second, sentences are extracted from the documents and ranked using query-independent features such as sentence length and query-dependent features such as the degree of match between the sentence and the query.

Search queries are constructed from content words and phrases in the question. Words that are arguments of relation instances involving the focus are weighted higher. The following Indri queries are generated for the example question:

```
#combine(disease causes right upper quadrant pain)
#combine[passage20:6](disease causes right upper quadrant pain)
```

These queries retrieve documents and passages containing one or more of the search words; the more words that are found in a retrieved document or passage, the higher its score. The notation `passage20:6` specifies that Indri should evaluate dynamically generated passages of length 20 words created every six words. Watson extends the beginning and the end of each retrieved passage to sentence boundaries.

Searches are issued against a corpus of documents for the given application of Watson. For competing on the Jeopardy! television game show, the corpus includes Wikipedia (Wikimedia Foundation, 2014a), dictionaries, literary works, and news stories. For the Watson Medical Engine, the corpus includes medical textbooks, dictionaries, guidelines, and journal articles.

18.4.4 ANSWER LOOKUP

In addition to performing text searches, Watson performs *answer lookup*, which consists of looking up potential answers in structured resources. The structured resources for Jeopardy! include DBpedia (2014) and IMDb (2014). The structured resources for the Watson Medical Engine include UMLS and IBM's Structured Knowledge Base for Medical (SKB).

18.4.5 SKB

SKB provides informativeness scores for signs, symptoms, and laboratory tests relative to diseases. For example, given

```
findingOf(right upper quadrant pain, disease : focus)
```

the SKB answer lookup component retrieves all diseases associated with right upper quadrant pain and the corresponding informativeness scores.

SKB's informativeness scores are based on the *pointwise mutual information* between two medical concepts, which is calculated using the equation

$$pmi(c_1, c_2) = \log_2 \frac{P(c_1, c_2)}{P(c_1)P(c_2)}$$

where $P(c_1, c_2)$ is the probability that c_1 and c_2 co-occur, $P(c_1)$ is the probability of c_1, and $P(c_2)$ is the probability of c_2. These probabilities are estimated from a corpus of medical texts.

18.4.6 CANDIDATE GENERATION

Once search and answer lookup have been performed, *candidate generation* is performed, which consists of generating candidate answers along with their scores to be used by downstream components. Several methods are used to identify likely answer candidates in a document. For documents and passages, entities identified by entity recognizers and hyperlink anchor text are generated as candidates. For documents, the titles of the documents are generated as candidates. Answers produced by answer lookup are also generated as candidates along with scores. For example, candidate generation produces candidates such as the following:

`cholecystitis 0.812` (from document)
`gallstones 0.724` (from passage)
`cholelithiasis 0.714` (from passage)
`acute pancreatitis 0.697` (from SKB answer lookup)

18.4.7 SUPPORTING EVIDENCE RETRIEVAL

Once candidate answers have been generated, *supporting evidence retrieval* is performed. For each candidate answer, a new query is constructed from the candidate answer and the question. Watson uses the Indri search engine for supporting evidence retrieval. For the candidate `cholecystitis`, an Indri query such as the following is generated:

```
#combine[passage20:6](cholecystitis disease causes right upper
 quadrant pain)
```

For each candidate answer and passage retrieved for the answer, a number of *context-dependent answer scorers* are run. Each such scorer produces a score that measures the degree to which the passage provides evidence that the candidate is the correct answer to the question. Each passage is analyzed similarly to how questions are analyzed. The passage is parsed, entities and relation instances are identified, and coreferences are resolved. The search is run against a corpus, which can be the same as the corpus used in primary search or a different corpus.

Supporting evidence retrieval improves system performance. By adding the candidate answer to the search query, it retrieves relevant passages that might have been missed by primary search. For example, primary search might retrieve the passage

```
Abdominal pain, nausea, and vomiting are symptoms of cholecystitis.
```

and generate the candidate `cholecystitis`. The trouble is that this passage does not match the question well—only the words `pain` and `cholecystitis` are contained in the question and candidate—so the passage does not receive a very high score from the answer scorers. But supporting evidence retrieval might return the following passage:

```
Cholecystitis causes RUQ tenderness.
```

This passage does match the question well, taking into account that `RUQ` is an abbreviation for `right upper quadrant`. This passage receives a high score from the answer scorers with respect to the correct answer `cholecystitis`.

18.4.8 TERM MATCHING

A *term* is a word or *phrase*, which is a sequence of words. Terms are identified by the ESG parser and other recognizers. Answer scorers depend on *term matchers*, which compute a measure of the degree of match between two terms called the *term match score*. Watson may be configured with different term matchers depending on the application. Different answer scorers may also use different term matchers. Watson contains a variety of term matchers, including the following:

- The *text equality term matcher* returns 1.0 if the term strings are equal ignoring case and 0.0 otherwise.
- The *chunk headword term matcher* returns 0.64 if the headwords of the chunk terms are equal ignoring case and 0.0 otherwise. Chunks are identified by the ESG parser using a chunk lexicon. The headword of the chunk `right upper quadrant pain` is `pain`.

- The *whitespace term matcher* returns 0.1 if any word in one term equals any word in the other term ignoring case and 0.0 otherwise.
- The *WordNet synonym term matcher* returns 1.0 if the terms are synonyms according to WordNet and 0.0 otherwise.

The scores from multiple term matchers may be combined by taking the maximum or using functions learned with machine learning techniques.

18.4.9 TERM WEIGHTING

Less frequent terms like `appendicitis` are more informative than frequent terms like `the`. Therefore, some answer scorers multiply term match scores by the *inverse document frequency* (idf), which is computed from a large corpus using the equation

$$idf(t) = \log_2 \frac{N}{c(t) + 1}$$

where $c(t)$ is the number of documents that contain term t and N is the total number of documents in the large corpus.

The *idf-weighted term match score* of a question term and a passage term is the product of (1) the idf of the question term and (2) the term match score of the question term and the passage term.

18.4.10 ANSWER SCORING

Scorers are broken down into *context-dependent answer scorers* and *context-independent answer scorers*. As previously described, context-dependent answer scorers produce a score for a candidate in the context of a given passage. Context-independent answer scorers do not involve passages. They simply score the degree to which a candidate is the correct answer to the question using other resources such as structured resources.

Context-dependent answer scoring

A context-dependent answer scorer might be fed the following:

```
Candidate: cholecystitis
Question: What disease causes right upper quadrant pain?
Focus: What disease
Passage: Cholecystitis may cause right upper quadrant pain.
```

The scorer's job is to produce a score of how well the passage supports the answer to the question. Watson contains a number of context-dependent answer scorers. We discuss four of them: the passage term match scorer, the textual alignment candidate scorer, the skip-bigram scorer, and the logical form answer candidate scorer.

Passage term match scorer

The *passage term match scorer* computes how well the terms in the question match the terms in the passage without regard to the order of terms. It uses the following algorithm:

1. For each question term in the question, compute a *question term score* equal to the maximum of the idf-weighted term match scores between the question term and all the passage terms in the passage.
2. Return the sum of the question term scores.

Textual alignment candidate scorer

The *textual alignment candidate scorer* (TACS) computes how well the question matches the passage, taking into account the order of terms. TACS uses the algorithm of Smith and Waterman (1981), which has been used for nucleotide and protein sequence matching, to find local alignments between the question and the passage. Idf weighting is used, and high credit is given for matching the focus in the question with the candidate answer in the passage.

Syntactic-semantic graph

The next two answer scorers make use of a *syntactic-semantic graph* of the question and the passage. The syntactic portion of the graph for a text consists of a predicate-argument structure representation abstracted from the ESG parse of the text. The semantic portion of the graph for a text consists of the relation instances extracted from the text by relation extraction.

Skip-bigram scorer

A *skip-bigram* is defined to be a pair of terms that are either (1) directly connected to each other in the syntactic-semantic graph or (2) both directly connected to a third common node in the graph. The *skip-bigram scorer* computes how well the skip-bigrams in the question graph match the skip-bigrams in the passage graph. The focus in the question graph is assumed to be an exact match for the candidate answer in the passage graph. Idf weighting is not used in the skip-bigram scorer because it was found to hurt performance.

Logical form answer candidate scorer

The *logical form answer candidate scorer* (LFACS) computes how well the question matches the passage based on how well their syntactic-semantic graphs match, subject to the constraint that the focus in the question graph must correspond to the candidate answer in the passage graph. The LFACS algorithm, which is similar to the structure-mapping algorithm described in Section 17.2, is as follows:

1. Align the question graph to the passage graph, to the degree possible.
2. For each question term in the question graph, compute a *question term score* equal to the product of (a) the *structural match score* of the question term and (b) the idf-weighted term match score of the question term and the corresponding passage term in the passage graph.
3. Return the sum of the question term scores.

The structural match score of a question term is computed as follows:

1. For each path from the question term to the focus in the question graph, compute a *path score* equal to the product of all (a) idf-weighted term match scores and (b) *edge match scores* between the terms and edges along the path in the question graph and the corresponding terms and edges in the passage graph.
2. Return the maximum of the path scores.

Edge match scores are similar to term match scores. If two edges have the same relation name, their edge match score is 1.0.

Context-independent answer scoring

A context-independent answer scorer takes a question and a candidate answer (along with all the associated annotations in the CAS) and returns a score of how well the candidate answer answers the question.

Type coercion

An important set of context-independent answer scorers are the *type coercion* components. These take the question's LAT and candidate answer as input and produce a measure of the degree to which the candidate answer is an instance of the LAT. Type coercion components use structured resources such as WordNet, YAGO (Max Planck Institute for Computer Science, 2014), and UMLS.

18.4.11 FINAL MERGING AND RANKING

After supporting evidence retrieval and answer scoring, *final merging and ranking* are performed. A final score is computed using machine learning techniques for each candidate answer. The candidate answers are then sorted by decreasing score and produced as output. The details of final merging and ranking are as follows.

Feature merging

Watson components such as candidate generation and answer scorers generate *features*, which consist of a name and a numeric value. At various points in processing, multiple values of a feature need to be combined into a single number. For example, context-dependent answer scorers such as LFACS produce a score for each candidate answer and passage retrieved for that answer. As part of final merging and ranking, these scores are combined into a single LFACS score for each candidate answer. This is done using a *feature merging* process that combines feature values using a particular function such as `maximum` or `sum` assigned to each feature name.

The function `maximum` was found to work best for LFACS feature merging. Therefore, if LFACS produces the following scores for passages retrieved for the candidate answer `cholecystitis`

```
Cholecystitis is associated with RUQ pain.         0.798
Cholecystitis may cause right upper quadrant pain. 0.821
Abdominal pain is a symptom of cholecystitis.      0.677
```

then the maximum score 0.821 is used as the single LFACS score for the candidate answer `cholecystitis`.

Standardized features

For every feature produced by the candidate generators and answer scorers, a *standardized* version of that feature is computed for use in machine learning. For each question and every candidate answer, the standardized version of each feature is computed using the equation

$$x_i' = \frac{x_i - \mu_i}{\sigma_i}$$

where μ_i and σ_i are the mean and standard deviation, respectively, of the values of x_i for all the candidate answers to the question.

Logistic regression models

Watson uses machine learning techniques to estimate binary logistic regression models for classifying whether candidate answers are incorrect (0) or correct (1). A score between 0.0 and 1.0 for a candidate answer with feature values $x_1, \ldots, x_n$ is computed using the logistic function

$$score = \frac{1}{1 + e^{-(\beta_0 + \sum_{i=1}^{n} \beta_i x_i)}} \tag{18.3}$$

where β_0 is the *intercept* and $\beta_1, \ldots, \beta_n$ are the *feature weights*. Together, β_0 and $\beta_1, \ldots, \beta_n$ are the *model*.

A model is *trained* by finding the values of $\beta_0, \ldots, \beta_n$ that provide the best fit between the scores computed using (18.3) and the correct scores as given by an *answer key*. A model is *applied* by using (18.3) to compute scores.

To train Watson's models, we must supply representative questions along with correct answer strings or regular expressions for each question. The more questions the better. *Training questions* are used for training, and *test questions* are used for testing trained models.

Machine learning phases

In Watson, models are trained and applied in several *phases*. When Watson is run on training questions, the models are trained and applied in each phase. Once the models are trained and Watson is run on test questions or on new, live questions, the models are only applied in each phase.

1. In the *hitlist normalization phase*, for each question, candidate answers whose strings are equal are merged using feature merging, standardized feature values are computed, and the *hitlist normalization model* is (trained and) applied.
2. In the *base phase*, the top 100 answers from the hitlist normalization phase are retained, the standardized features are recomputed, and the *base model* is (trained and) applied.

3. In the *answer merging phase*, equivalent answers like `gallstones` and `cholelithiasis` are merged using feature merging, the standardized features are recomputed, and the *answer merging model* is (trained and) applied to produce a final score for each candidate answer.

Synonym and inflection dictionaries and pattern-based methods are used to detect equivalent candidate answers. When several candidate answers are detected to be equivalent, the string of the one with the highest score is used as the string of the merged candidate answer.

Watson can also use different models for different question classes.

18.4.12 WATSONPATHS

WatsonPaths is a reasoning system built on top of Watson. Like Watson, it is implemented using UIMA. Consider the following question:

```
What is the treatment for jaundice and right upper quadrant
pain aggravated by fatty foods?
```

Answering this question requires two steps: First, we must determine the patient's disease. Second, we must determine the treatment for that disease. WatsonPaths answers this *top-level question* by asking Watson multiple *subquestions*. First, it asks the following subquestions:

```
What disease causes jaundice?
What disease causes right upper quadrant pain aggravated by
fatty foods?
```

Then, given the diseases returned in response to these subquestions, WatsonPaths asks further subquestions:

```
What is the treatment for cholecystitis?
What is the treatment for gallstones?
...
```

WatsonPaths uses the following algorithm:

1. Identify important *factors* in the top-level question and add them to the *assertion graph*, which consists of nodes and links. Each node and link has an associated *score*. In our example, WatsonPaths adds nodes with score 1.0 for the factors `jaundice` and `right upper quadrant pain aggravated by fatty foods`.
2. Use *question asking strategies* to ask Watson subquestions based on nodes in the graph whose score is above a threshold.
3. In response to answers to these subquestions, add nodes and links to the graph. Links in a WatsonPaths assertion graph include `indicates`, `causes`, `associatedWith`, `affects`, `contains`, and `matches`. In our example, WatsonPaths adds a `causes` link from `cholecystitis` to `jaundice` and a

   `causes` link from `cholecystitis` to `right upper quadrant pain aggravated by fatty foods`.

4. Use Bayesian inference to recalculate the scores associated with nodes and links in the graph as it is updated. This is done by the *belief engine*.
5. If a stopping condition is reached, exit with a set of conclusions along with scores for those conclusions.
6. Go to step 2.

We summarize the operation of WatsonPaths as follows:

Input: top-level question ⟶
identify important factors in the question ⟶
add the factors to the graph ⟶
repeatedly ask subquestions and update graph and scores ⟶
Output: conclusions and scores

The interface to WatsonPaths allows the user to explore the assertion graph and drill down into the evidence for each node and link in the graph, including the evidence for answers returned by Watson. WatsonPaths also allows the user to teach WatsonPaths and Watson to improve the performance of both systems on various subtasks of the top-level question answering task.

18.4.13 WATSONPATHS FOR COMMONSENSE REASONING

Watson serves as a dynamic medical knowledge base to support medical reasoning in WatsonPaths. It answers questions such as `What disease causes right upper quadrant pain?` Similarly, Watson can serve as a dynamic commonsense knowledge base to support commonsense reasoning. It can be trained to answer questions about commonsense knowledge such as object properties, event effects, preconditions, and goals:

```
What is the color of the sky?
What is a pen used for?
What is the effect of eating a large meal?
What is a precondition of eating dinner?
What is the goal of drinking alcohol?
What goal is initiated by being hungry?
```

We can also use it for story understanding. Roger Schank and Robert Abelson introduced the example

Willa was hungry.
She took out the Michelin Guide.

The MICRO PAM program generates an explanation of this story using structured information. This can be reimplemented in WatsonPaths using unstructured information. We supply WatsonPaths with the following subquestion generation strategies:

Source Type	Target Type	Question	MICRO PAM Link	Direction
State	Goal	What goal is initiated by *state*?	`initiate`	Forward
Goal	Plan	What is a plan for *goal*?	`planFor`	Forward
Plan	Goal	What is a subgoal of *plan*?	`subgoal`	Forward
Event	Goal	What is the goal of *event*?	`instantiation`	Backward

Given the above story, WatsonPaths asks forward and backward questions to an instance of Watson previously trained on questions of these types. The result is a WatsonPaths assertion graph containing the following nodes and links:

state: Willa was hungry. →
`initiate` *(What goal is initiated by Willa was hungry?)* →
goal: satisfy hunger →
`planFor` *(What is a plan for satisfy hunger?)* →
plan: eat at a restaurant →
`subgoal` *(What is a subgoal of eat at a restaurant?)* →
goal: find a good restaurant →
`instantiation` *(What is the goal of She took out the Michelin Guide?)* →
event: She took out the Michelin Guide.

18.4.14 THE ADAPTWATSON METHODOLOGY

The key to developing a successful Watson application is applying the AdaptWatson methodology, an empirical method for developing artificial intelligence applications. This methodology provides an efficient way of driving up the performance of an intelligent application to meet performance objectives. This methodology consists of the following steps:

1. Acquire a large amount of natural language content about the application domain.
2. Acquire training and test data using human annotators or crowdsourcing.
3. Generate ideas for components.
4. Implement the components or use state-of-the-art components where available.
5. Perform component-level tests and drive up the performance of the components.
6. Use machine learning to learn which components work and how to weight their contribution.
7. Evaluate the end-to-end impact of adding the components to the system.

8. Perform *accuracy analysis* to identify problem areas and generate ideas for improving the components or acquiring additional content.
9. Perform *headroom analysis* to estimate the maximum end-to-end impact of successfully implementing the improvements or adding the content.
10. If the headroom of the proposed improvement or content acquisition is significant, implement the improvement or acquire the content.
11. Iterate on the above until performance objectives are met.
12. Recurse on the above. Use the same methodology to develop subcomponents as needed.

18.5 COMPARISON

Systems and methods for reasoning using natural language are compared in Table 18.1. All the systems support chaining in some form. SAFE, OWL, and Montagovian syntax required bracketed input and do not accept free-form English. ACE and PENG Light add a parser so that input does not need to be bracketed, but they restrict input to expressions of a controlled language. REFORMULATOR does not require bracketed input, but can only reason with natural language expressions that are matched by reformulation rules. Watson and WatsonPaths work with arbitrary natural language text, and they are able to exploit the redundancy in large natural language corpora.

BIBLIOGRAPHIC NOTES

Automatic programming

Sammet (1966) proposed to use natural language as a programming language. M. Halpern (1966) makes a strong case for the use of natural language as a programming language and points out the advantages of exploiting the redundancy in natural language. Automatic programming is discussed by Balzer (1985). SAFE is discussed by Balzer, Goldman, and Wile (1978). OWL is discussed by Szolovits,

Table 18.1 Natural Language Reasoning Systems and Methods[a]

	SAFE	OWL	MS	AP	R	W
Can perform chaining	✓	✓	✓	✓	✓	✓
Accepts nonbracketed text				✓	✓	✓
Accepts unrestricted text						✓
Exploits redundancy						✓

[a]*MS, Montagovian syntax; R, REFORMULATOR; W, Watson/WatsonPaths; AP, Attempto Controlled English and PENG Light.*

Hawkinson, and Martin (1977). Swartout (1977) provides examples of OWL methods and describes OWL's explanation generator.

Reasoning with natural language

Montague (1970) wrote, "There is in my opinion no important theoretical difference between natural languages and the artificial languages of logicians" (p. 373). He introduced what is now called Montague grammar in a series of papers; they are reprinted in a book by Thomason (1974). A introductory book on Montague grammar is by Dowty, Wall, and Peters (1981). Montagovian syntax is described by McAllester and Givan (1989, 1992). There have been many proposals for logics that are closer to natural language than classical predicate logic (Iwańska & Shapiro, 2000). Some examples are the proposals of Lakoff (1970), Suppes (1979), Hobbs (1985), Hoeksema (1986), van Benthem (1987), Schubert and Hwang (1989), Purdy (1991), Sánchez Valencia (1991), Ali and Shapiro (1993), and MacCartney and Manning (2008). Books on formal semantics of natural language are by Portner (2005) and Blackburn and Bos (2005). Attempto Controlled English is described by Fuchs, Schwertel, and Schwitter (1999). The Attempto reasoner is discussed by Fuchs and Schwertel (2002). Discourse representation structures are described by Kamp and Reyle (1993). The use of PENG Light for specifying event calculus domain descriptions is described by Schwitter (2011). P. Clark, Harrison, Jenkins, Thompson, and Wojcik (2005) also use restricted English for specifying the effects of events. Kuhn (2014) provides a comprehensive survey and classification of 100 controlled natural languages based on English. Singh (2002) describes REFORMULATOR. See also the discussions of Liu and Singh (2004a) and Singh, Barry, and Liu (2004).

Watson

Early question answering systems include Baseball (Green, Jr., Wolf, Chomsky, & Laughery, 1961), LUNAR (Woods, 1973), and QUALM (Lehnert, 1978). Book-length treatments of question answering systems are by Maybury (2004) and Strzalkowski and Harabagiu (2008). An introduction to Watson is provided by Ferrucci (2012). UIMA is described by Ferrucci and Lally (2004). Scaling up Watson is discussed by Epstein et al. (2012). Watson's question analysis is described by Lally et al. (2012). The ESG parser is described by McCord (2006a, 2006b, 2010). Watson's parsing, predicate argument structure generation, and relation extraction are described by McCord, Murdock, and Boguraev (2012). Watson's primary search, answer lookup, and candidate generation are described by Chu-Carroll et al. (2012). SKB is discussed by Ferrucci, Levas, Bagchi, Gondek, and Mueller (2013). Supporting evidence retrieval and answer scoring are described by Murdock, Fan, Lally, Shima, and Boguraev (2012). LFACS is described by Murdock (2011) and Murdock, Fan, Lally, Shima, and Boguraev (2012). The structure-mapping algorithm is described by Falkenhainer, Forbus, and Gentner (1989). Pointwise mutual information and inverse document frequency are discussed by Manning and Schütze (1999). Watson's type coercion components are described by Murdock, Kalyanpur, et al. (2012). Watson's final merging and ranking is described by Gondek et al. (2012). The Watson Medical Engine is described by Ferrucci, Levas, Bagchi, Gondek, and Mueller (2013). WatsonPaths is discussed by Lally et al. (2014). The Willa example was introduced by Schank and Abelson (1977, p. 71). The code for MICRO PAM, a simplified version of PAM (Wilensky, 1978, 1983), and the Willa example are given by Schank and Riesbeck (1981). The AdaptWatson methodology is described by Ferrucci and Brown (2012). Empirical methods for artificial intelligence are discussed by Cohen (1995).

EXERCISES

18.1 Write a program that takes bracketed natural language specifications as input and produces programs as output.

18.2 Define a controlled language for event calculus domain descriptions. Write a program that converts a domain description written using the controlled language into an answer set program.

18.3 Write a program that takes English sentences and reformulation rules as input and produces inferred English sentences as output.