Information Extraction (IE)

IE is a process which takes unseen texts as input and produces fixed-format, unambiguous data as output. This data may be used directly for display to users, or may be stored in a database or spreadsheet for later analysis, or may be used for indexing purposes in Information Retrieval (IR) applications such as Internet search engines. [Cowie and Lehnert 96, Appelt 99]

IE is not IR

▶ IR pulls documents from large text collections (usually the Web) in response to specific keywords or queries. You analyse the documents.
▶ IE pulls facts and structured information from the content of large text collections. You analyse the facts.
IE is an enabling technology for many other applications

- Text mining
- Semantic annotation
- Question answering
- Opinion mining
- and so on ...

Typical subtasks of IE

- **Named Entity recognition (NE)**
  Finds and classifies names, places, etc.
- **Coreference resolution (CO)**
  Identifies identity relations between entities in texts.
- **Template Element construction (TE)**
  Adds descriptive information to NE results (using CO).
- **Template Relation construction (TR)**
  Finds relations between TE entities.
- **Scenario Template production (ST)**
  Fits TE and TR results into specified event scenarios.

Example of IE

The **Atlas rocket** was fired on **Tuesday**. It is the brainchild of Dr. Big Head. **Dr. Head** is a staff scientist at **We Build Rockets Inc**.

- **NE**: entities → (Atlas rocket, Tuesday, Dr. Head, We Build Rockets Inc)
- **CO**: it → Atlas rocket
- **TE**: shiny red rocket → Head’s brainchild
- **TR**: Dr. Head → We Build Rockets Inc
- **SR**: → rocket launching event with various entities involved in
Two kinds of approaches

Knowledge Engineering
- rule based
- developed by experienced language engineers
- make use of human intuition
- require only small amount of training data
- development can be very time consuming
- some changes may be hard to accommodate

Learning Systems
- use statistics or other machine learning methods
- developers do not need LE expertise
- require large amounts of annotated training data
- some changes may require re-annotation of the entire training corpus

The cornerstone of IE:

Named Entity Recognition
- Identification of proper names in texts, and their classification into a set of predefined categories of interest
- Persons
- Organisations (companies, government organisations, committees, etc)
- Locations (cities, countries, rivers, etc)
- Date and time expressions

Why is Named Entity Recognition important?
- NE provides a foundation from which to build more complex IE systems
- Relations between NEs can provide tracking, ontological information and scenario building
- Tracking (co-reference): Dr. Head, Joe Head, Joe, he
Typical NE pipeline

- **Pre-processing**: tokenisation, sentence splitting, morphological analysis, POS tagging
- **Entity finding**: gazetteer lookup, NE grammars
- **Coreference**: alias finding, orthographic coreference etc.
- **Export** to database, XML, ontology

Example of IE

*Joe Lives in Cologne. He works there for IBM.*

- NE Recognition
- Coreference
- Relations

General Architecture for Text Engineering

- Is a framework for language processing ([http://gate.ac.uk](http://gate.ac.uk))
- Open Source (LGPL licence)
- A framework for programmers, GATE is an object-oriented class library that implements the architecture
- A development environment: For language engineers, computational linguists et al, a graphical development environment
- GATE includes support for reading in various formats and converting to the internal annotation representation: HTML, XML, PDF, SGML, RTF, email, plain text
- Over ten years old with 1000s of users at 100s of sites
- Current version 6

GATE includes

- **Components for language processing** e.g. parsers, machine learning tools, stemmers, IR tools, IE components for various languages...
- Tools for visualising and manipulating text, annotations, ontologies, parse trees, etc.
- **Various information extraction** tools
- **Evaluation and benchmarking** tools
GATE Components

- GDM: the GATE Document Manager
- GGI: the GATE Graphical Interface
- CREOLE: a Collection of REusable Objects for Language Engineering: a set of LE components integrated with the system

GATE Components are one of three types:

- **Language Resources** (LRs): lexicons, corpora, ontologies
- **Processing Resources** (PRs): represent entities that are primarily algorithmic
- **Visual Resources** (VRs): represent visualisation and editing components that participate in GUIs

GATE APIs

- Everything is a replaceable bean
- All communication via fixed APIs

A Nearly-New Information Extraction System

- **ANNIE** is a ready-made collections of algorithms that performs IE on unstructured text.
- The ANNIE application contains a set of core PRs:
  1. Tokeniser
  2. Sentence Splitter
  3. POS tagger
  4. Gazetteers
  5. Named entity tagger (JAPE transducer)
  6. Orthomatcher (orthographic coreference)
Each PR in the ANNIE pipeline creates some new annotations or modifies existing ones:

- **Document Reset** → removes annotations
- **Tokeniser** → Token annotations
- **Sentence Splitter** → Sentence, Split annotations
- **POS tagger** → adds category features to Token annotations
- **Gazetteers** → Lookup annotations
- **Named entity tagger (JAPE transducer)** → Date, Person, Location, Organisation annotations
- **Orthomatcher (orthographic coreference)** → adds match features to NE annotations

### Tokeniser

- Splits the text into very simple tokens such as numbers, punctuation and words of different types
- **Tokeniser Rules:**
  1. **left hand side (LHS)** → regular expression which has to be matched on the input and is separated from the RHS by `>`
     - **Operators:**
       - `|` (OR)
       - `*` (0 or more occurrences)
       - `?` (0 or 1 occurrences)
       - `+` (1 or more occurrences)
  2. **right hand side (RHS)** → describes the annotations to be added to the AnnotationSet, uses `;` as a separator:

### RHS

```
{LHS} > {Annotation type};{attribute1}={value1};...;{attribute n}={value n}
```

### Example

```
'UPPERCASE_LETTER' 'LOWERCASE_LETTER'* > Token;orth=upperInitial;kind=word;
```
Gazetteers

- Gazetteers are plain text files containing lists of names
- The lists are compiled into Finite State Machines
- Each gazetteer has an index file listing all the lists, plus features of each list
- Lists can be modified either internally using the Gazetteer Editor, or externally in your favourite editor

Editing gazetteer lists

- The ANNIE gazetteer has about 60,000 entries arranged in 80 lists
- Each list reflects a certain category
- List entries might be entities or parts of entities, or they may contain contextual information

Sentence Splitter

- Finds sentences based on Tokens
- Creates Sentence annotations and Split annotations on the sentence delimiters
- Uses a gazetteer of abbreviations etc. and a set of JAPE grammars which find sentence delimiters and then annotate sentences and splits
POS tagger

- ANNIE POS tagger is a modified version of the Brill tagger
- Uses a default lexicon and ruleset, trained on the Wall Street Journal corpus
- Default ruleset and lexicon can be modified manually
- Requires Tokeniser and Sentence Splitter to be run first

NE transducers

- Gazetteers can be used to find terms that suggest entities
- Entities can often be ambiguous
  - "May Smith" vs "May 2010" vs "May I help you?"
  - "General Motors" vs "General Smith"
- Handcrafted grammars are used to define patterns over the lookup and other annotations
- These patterns can help disambiguate, and they can combine different annotations: Date = day + number + month
- Each NE transducer consists of one or more grammars written in the JAPE language

Java Annotation Pattern Engine

- Jolly And Pleasant Experience :-)
- Specially developed pattern-matching language for GATE
- Each JAPE rule consists of:
  - LHS which consists patterns to match
  - RHS which details the annotations to be created
JAPE example

- Match all university names in Germany, e.g. “University of Bonn”

- The gazetteers might contain the word “Bonn” in the list of cities

- The rule looks for specific words such as “University of” followed by the name of a city

- This wouldn’t be enough to match all university names, but it’s a start :-)

Rule name

Rule: University1

LHS

(  
  {Token.string == "University"}
  {Token.string == "of"}
  {Lookup.minorType == city}
)
:orgName

RHS

:orgName.Organisation =
  {kind = "university", rule = "University1"}

Matching a text string

- Everything to be matched must be specified in terms of annotations

- Each annotations is enclosed in a curly brace

- To match a string of text, use the “Token” annotation and the “string” feature: { Token.string == "by" }

- You can combine sequences of annotations as a pattern
Labels on the LHS

- For every combination of patterns that you want to create an annotation for, you need a label
- The pattern combination is enclosed in round brackets, followed by a colon and the label

One or more cities or countries in any order and combination

( {Lookup.minorType == city} | {Lookup.minorType == country} )+

is not the same as

One city OR one or more countries

( {Lookup.minorType == city} | ( {Lookup.minorType == country}+ ) )

Coreference

- Different expressions may refer to the same entity
- Orthographic coreference matches proper names and their variants in a document
  - Mary Smith and Mrs. Smith
  - International Business Machines Ltd. will match IBM
- Classification of unknown entities very useful for surnames which match a full name, or abbreviations – Smith (unknown) will match Sir John Smith (person)

A Walk-Through Example

- A 3-stage procedure
- Recognise the phrase “800,000 US dollars” as an entity of type “Number”, with the feature “money”
- Give an example of a grammar rule for money
- Step 1: Tokenisation
- Step 2: List Lookup
- Step 3: Grammar Rules
Grammar rule for money

Macro: MILLION_BILLION
{{Token.string == "m"}}
{{Token.string == "million"}}
{{Token.string == "b"}}
{{Token.string == "billion"}}

Macro: AMOUNT_NUMBER
{{Token.kind == number}}
{{Token.string == ","}}
{{Token.string == "."}}
{{Token.kind == number}}
{{SpaceToken.kind == space}}
{{MILLION_BILLION}}

Rule: Money1
// e.g. 30 pounds
{{AMOUNT_NUMBER}}
{{SpaceToken.kind == space}}
{{Lookup.majorType == currency_unit}}

:money --> :money.Number = {kind = "money", rule = "Money1"}

System development cycle

1. Collect corpus of texts
2. Define what is to be extracted
3. Manually annotate gold standard
4. Split annotated data into training and evaluation sample
5. Create system from training data
6. Evaluate performance on evaluation sample
7. Return to step 4, until desired performance is reached

Before you start annotating...

- You need to think about annotation guidelines
- You need to consider what you want to annotate and then to define it appropriately
- With multiple annotators it’s essential to have clear set of guidelines for them to follow
- Consistency of annotation is really important for a proper evaluation

Tokenisation

Token, string = '800', kind = number, length = 3
Token, string = ',', kind = punctuation, length = 1
Token, string = '000', kind = number, length = 3
SpaceToken, string = ' ', kind = space, length = 1
Token, string = 'US', kind = symbol, length = 2, orth = allCaps
SpaceToken, string = ' ', kind = space, length = 1
Token, string = 'dollars', kind = word, length = 7, orth = lowercase

List Lookup

Lookup, minorType = post_amount, majorType = currency_unit

Grammar Rules

Number, kind = money, rule = Money1

Before you start annotating...

- You need to think about annotation guidelines
- You need to consider what you want to annotate and then to define it appropriately
- With multiple annotators it’s essential to have clear set of guidelines for them to follow
- Consistency of annotation is really important for a proper evaluation
Annotation guidelines

- People need clear definition of what to annotate in the documents, with examples
- Typically written as a guidelines document
- Piloted first with few annotators, improved, then “real” annotation starts, when all annotators are trained
- Annotation tools require the definition of a formal DTD (e.g. XML schema)

Performance Evaluation

- **Evaluation metric**: mathematically defines how to measure the system’s performance against human-annotated gold standard
- **Scoring program**: implements the metric and provides performance measures
  - For each document and over the entire corpus
  - For each type of annotation

Annotation in GATE GUI

- Adding annotation sets
- Adding annotations
- Resizing them (changing boundaries)
- Deleting
- Changing highlighting colour
- Setting features and their values

Terminology Comparison

**Gold Standard IE**
- Correct
- Missing
- Spurious
- Partially Correct

**Gold Standard IR**
- True Positive
- False Negative
- False Positive
- True Negative
Terminology Comparison

- **Correct**: things annotated correctly
  - annotating “Norbert Fuhr” as a Person

- **Missing**: things not annotated that should have been
  - not annotating “Duisburg” as a Location

- **Spurious**: things annotated wrongly
  - annotating “Norbert Fuhr” as a Location

- **Partially Correct**: the annotation type is correct, but the span is wrong
  - annotating just “Fuhr” as a Person is too short or
  - annotating “luckily Norbert Fuhr” as a Person is too long

Precision and Recall

- How many of the entities your application found were correct?

\[
\frac{\text{Correct}}{\text{Correct} + \text{Spurious}} = \text{Precision}
\]

- How many of the entities that exist did your application found?

\[
\frac{\text{Correct}}{\text{Correct} + \text{Missing}} = \text{Recall}
\]

F-Measure

- Precision and Recall tend to trade off against one another:
  - specifying rules precisely to improve precision, may cause a lower recall
  - very general rules, may deliver good recall, but low precision

- This make it difficult to compare applications, or to check whether a change has improved or worsened the results overall

- F-measure combines precision and recall into one measure

\[
F = \frac{2}{\frac{1}{P} + \frac{1}{R}}
\]