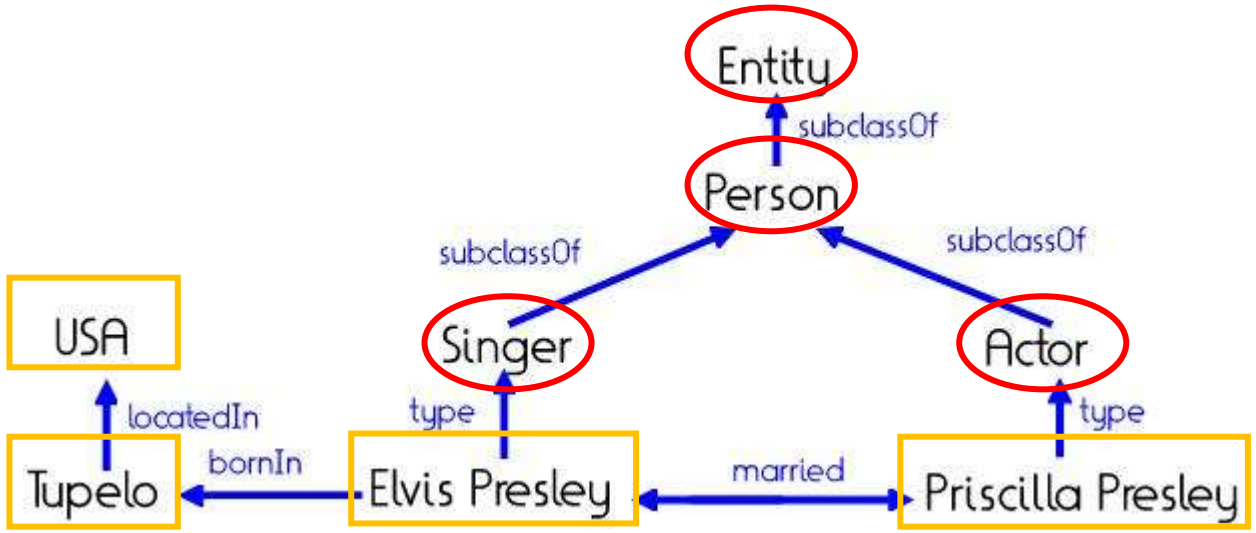
Knowledge Extraction from Unstructured Text Using Python

Outline

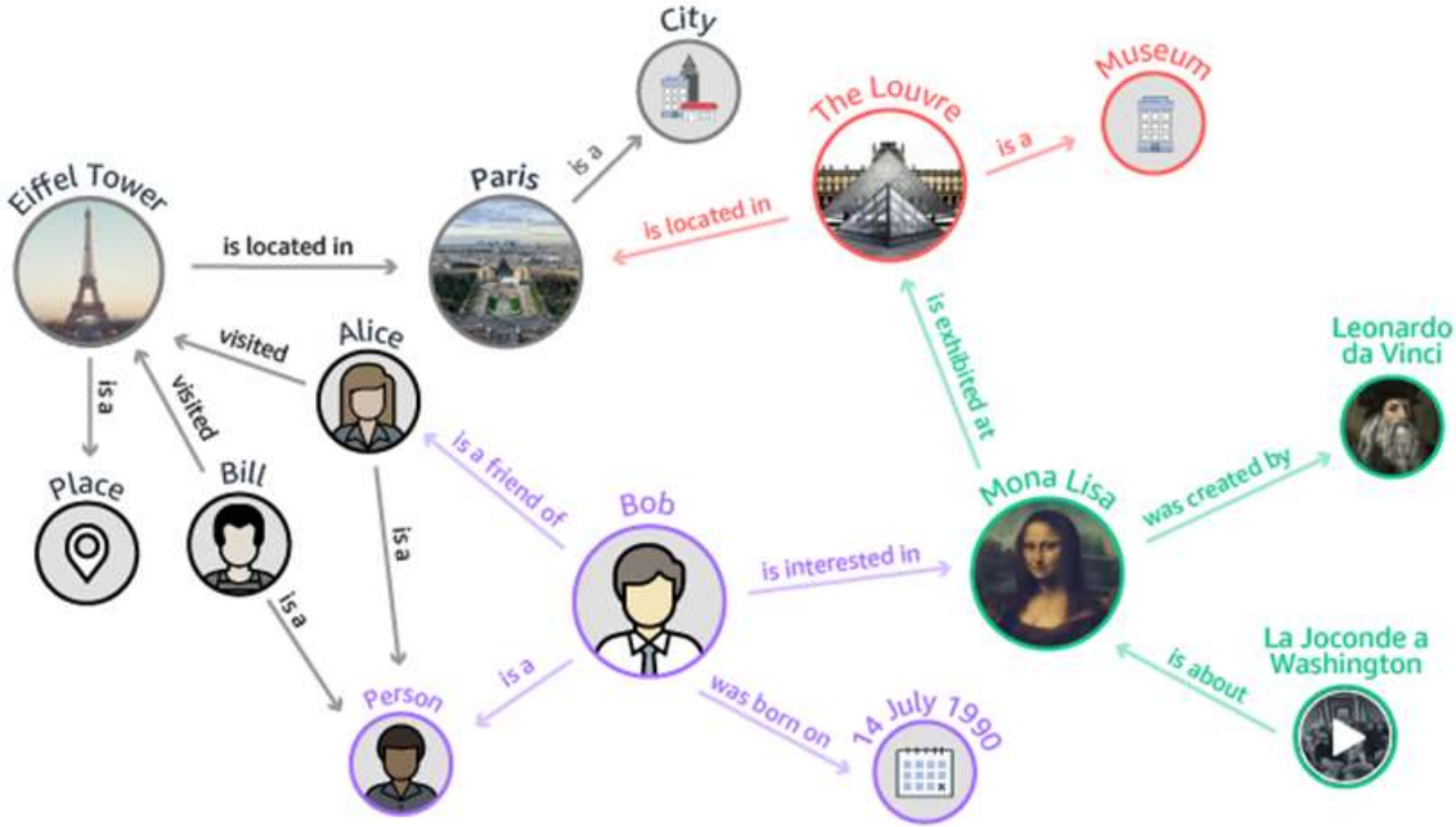
- Knowledge Graph Introduction
- Named Entity Extraction
- Relation Extraction
  - Hypernym relationship
  - Supervised learning
  - Distant supervision
- Information Extraction

Knowledge Graph

- A Knowledge Graph (KG) is a structured, computer-processable description of the world.
- A KG can be though of as a graph, in which the nodes are entities (instance and type) and the edges are relations.

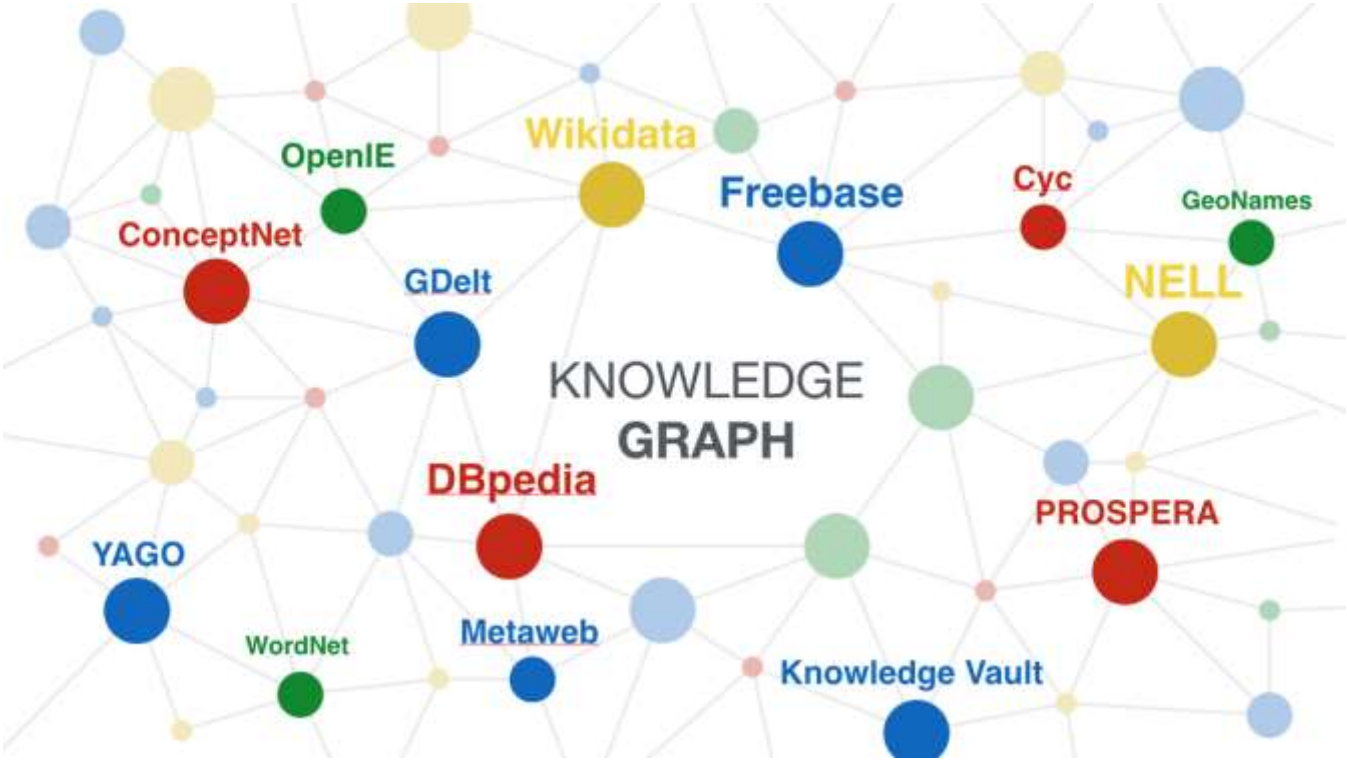


Knowledge Graph



Existing Knowledge Graph Products

- Google Knowledge Graph
  - Knowledge Vault
- Amazon Product Graph
- Facebook Graph API
- IBM Watson
- Microsoft Satori
  - Probase
- LinkedIn Knowledge Graph
- ……



Knowledge Graph – Semantic Search

- KBs serve to understand real-world entities and the relationships.

Things, not Strings!

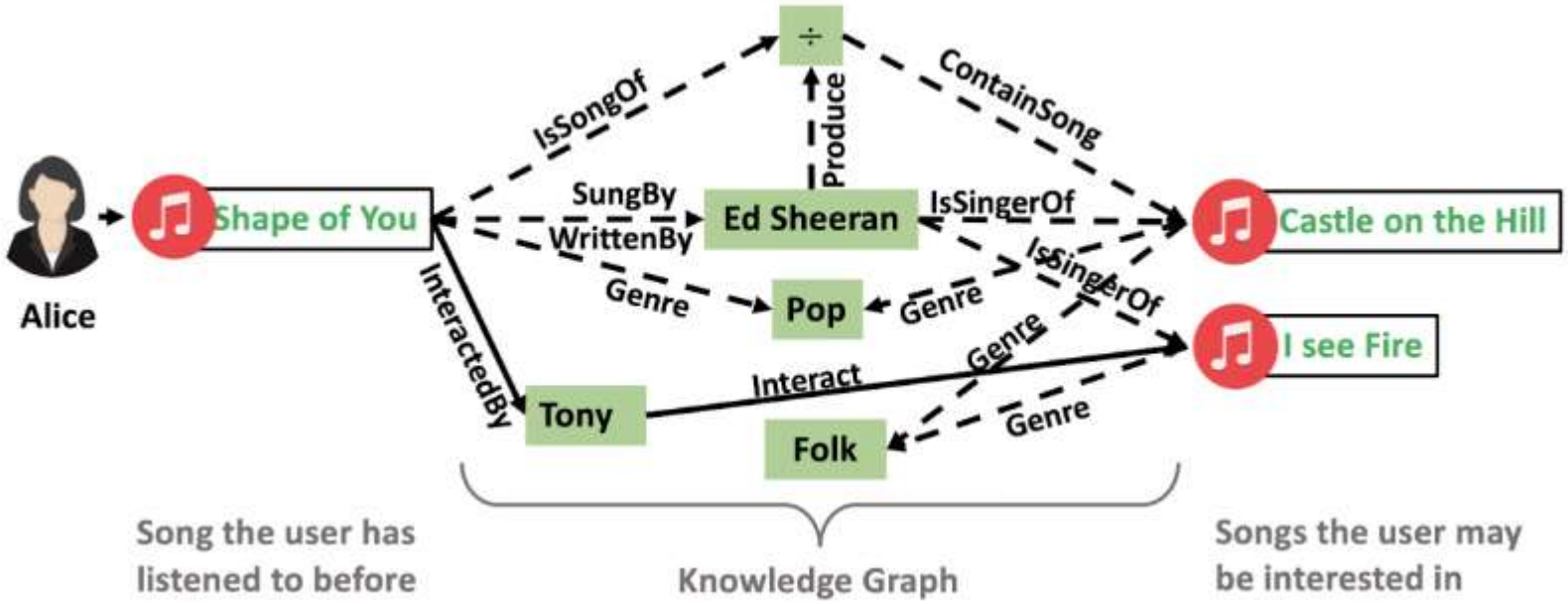
KG Applications – Question Answering

- "All movies by Spielberg this year"

KG Applications – Understanding of NLP

- A real example happened to one of my friend

KG Applications – Recommender System



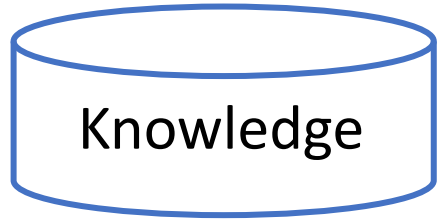
Perform explainable reasoning over knowledge graphs for recommendations

Knowledge Extraction

- **To build a knowledge graph from the text, it is important to make our machine understand natural language**.
- This can be done by using NLP techniques such as
  - Sentence segmentation
  - Dependency parsing
  - Parts of speech tagging
  - Entity recognition
  - Relation extraction
  - Coreference recognition
  - …

Knowledge Extraction

Documents



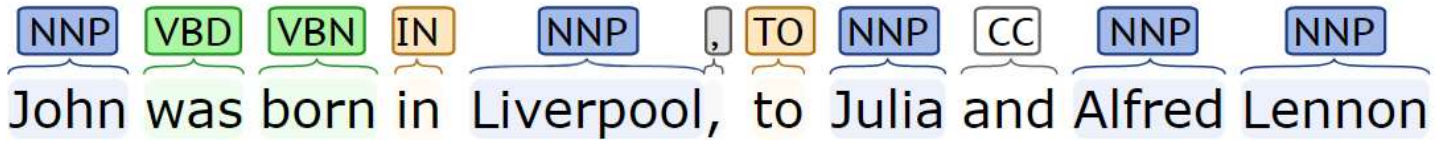
Unstructured
Ambiguous
Only human can understand

Structured
Precise
Machine can understand

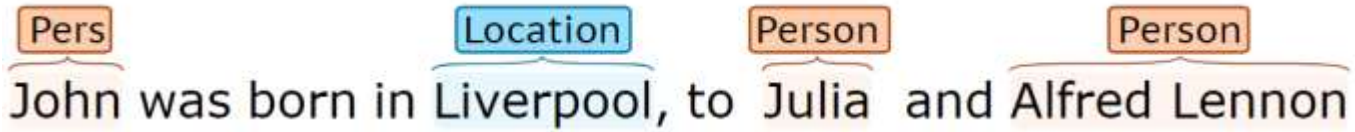
Can be used for downstream applications, such as creating Knowledge Graphs!

Knowledge Extraction – NLP Toolkit

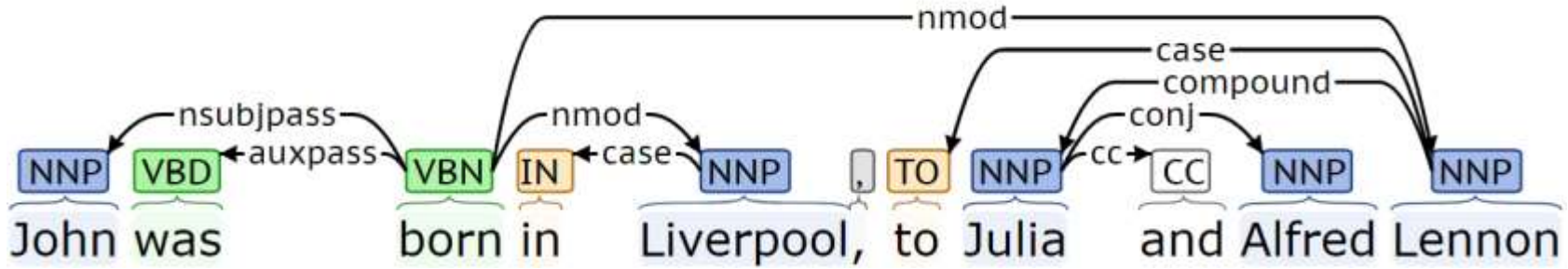
John was born in Liverpool, to Julia and Alfred Lennon [Text]



[Part-of-Speech]



[Named Entity]

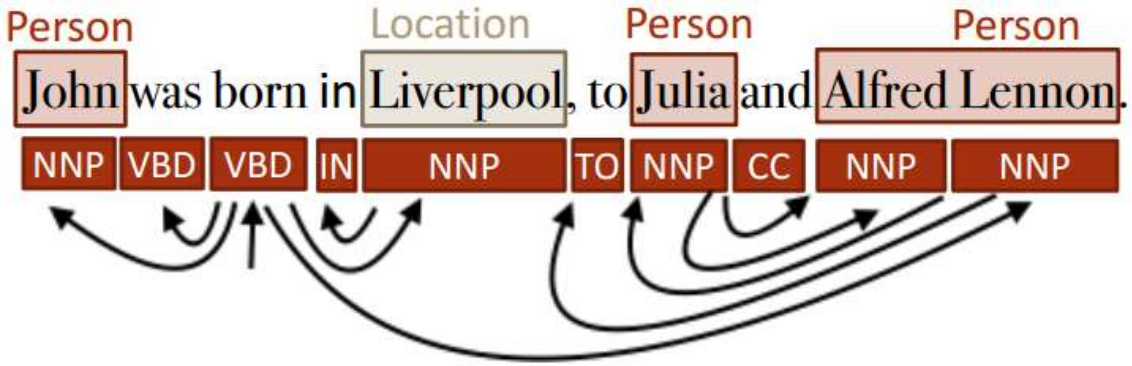


[Dependency]

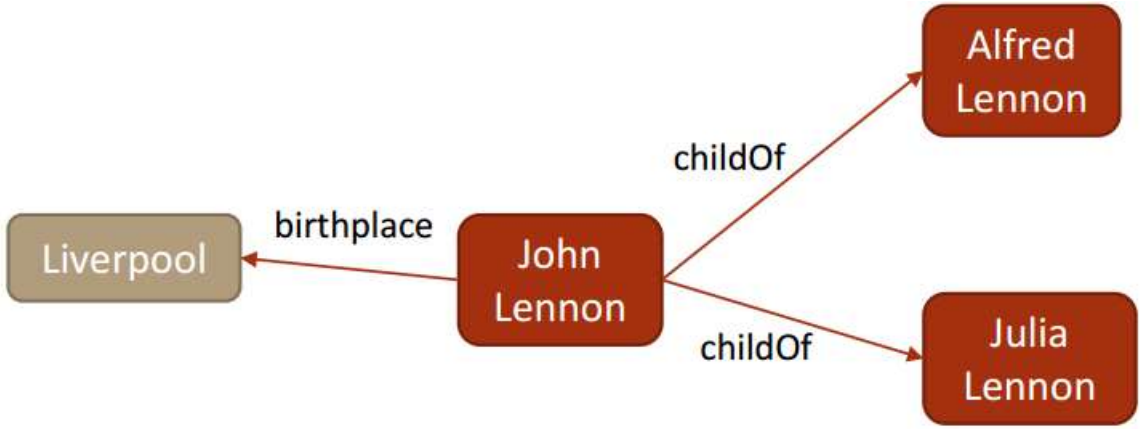
Online Demo: http://nlp.stanford.edu:8080/corenlp/process

Knowledge Extraction

John was born in Liverpool, to Julia and Alfred Lennon [Text]



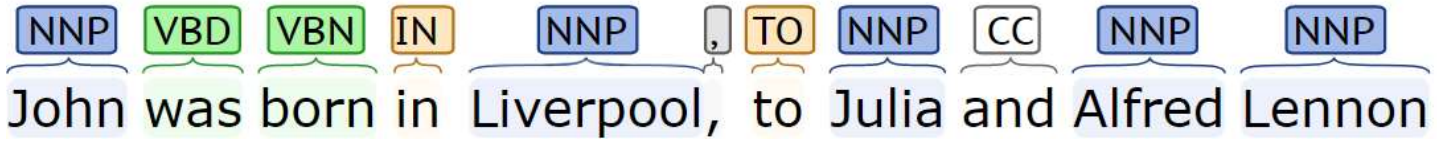
[Annotated Text]



[Knowledge Graph]

Named Entity Extraction

- Name of an entity belonging to a specified class, e.g., person, location, organization, data and so on.
- The extraction of a single word entity from a sentence is not a tough task. We can easily do this with the help of parts of speech (POS) tags. The nouns and the proper nouns would be our entities.



[Part-of-Speech]

Nouns are entities: Jonh, Liverpool, Julia, Alfred, Lennon

POS Tag list

- CC: coordinating conjunction 连接词 ;
- CD : cardinal digit 基数词
- DT : determiner 限定词 (this , that , these , those , such, little, some, any, every...)
- EX: existential there 存在句 (like: “there is” ... think of it like “there exists”);
- FW : foreign word 外来词
- IN: preposition/subordinating conjunction 介词或从属连词;
- JJ: adjective 形容词 (‘big’) ; JJR: adjective comparative 形容词比较级 (‘bigger‘) ; JJS adjective, superlative 形容词最高级 (‘biggest’);
- LS : list marker 列表标示 (‘1)’) ;
- MD: modal 情态助动词 (could, will)
- NN: noun, singular 常用名词单数 (‘desk’); NNS: noun plural 常用名词复数 (‘desks‘) ; NNP: proper noun, singular 专有名词单数 (‘Harrison’); NNPS: proper noun, plural 专有名词复数 (‘Americans’)
- PDT: predeterminer 前位限定词 (‘all the kids’)
- POS: possessive ending 所有格结束词 (parent's)

POS Tag list

- PRP: personal pronoun 人称代词 (I, he, she)
- PRP$: possessive pronoun 所有格代词 (my, his, hers)
- RB: adverb 副词 (very, silently); RBR: adverb, comparative 副词比较级 (better) ; RBS adverb, superlative 副词最高级 (best)
- RP: particle (give up) ;
- TO : to (go 'to' the store).
- UH interjection 感叹词 (errrrrrrrm)
- VB : verb, base form 动词基本形式 (take); VBD : verb, past tense 动词过去式 (took) ; VBG verb, gerund/present participle: 动名词和现在分词 (taking)
- VBN verb, past participle 过去分词 (taken) ; VBP verb, sing. present, non-3d take动词非第三人称单数; VBZ verb, 3rd person sing. present 动词第三人称单数 (takes)
- WDT: wh-determiner 限定词（如关系限定词： whose,which. 疑问限定词： what,which,whose.）
- WP: wh-pronoun 代词 (who, what);
- WP$: possessive wh-pronoun 所有格代词 (whose)
- WRB: wh-abverb 疑问代词 (where, when)

Named Entity Extraction

- Single word entity extraction based on POS tag is not enough, we need to be able to grouping the words into what are known as "noun phrases"
- These are phrases of one or more words that contain a noun, maybe some descriptive words, maybe a verb, and maybe something like an adverb. The idea is to group nouns with the words that are in relation to them.
- There are many there are a lot of open source tools that can be applied to discover noun phrases in the source text. In this project, we choose NLTK for noun phrase detection (http://www.nltk.org/howto/relextract.html, http://www.nltk.org/book/).

Named Entity Extraction

- Chunking: group words into meaningful chunks
  - Combine the part of speech tags with regular expressions
  - Chunk with NLTK (http://www.nltk.org/book/ch07.html)
- NLTK provides a classifier that has already been trained to recognize named entities, accessed with the function nltk.ne_chunk()
  - If we set the parameter binary=True , then named entities are just tagged as NE;
  - Otherwise, the classifier adds category labels such as
    - PERSON, ORGANIZATION, LOCATION, DATA, TIME, MONEY, PERCENT, FACILITY, and GPE (*Geo-Political Entity*).

NLTK

- First install python environment
- Install NLTK: pip install nltk
- Data distribution for NLTK
  - Install using NLTK downloader: nltk.download()
  - Or download from github (https://github.com/nltk/nltk_data/tree/gh-pages), unzip and then copy the six sub-directory in the packages into your nltk_data directory (check the directory in your own directory: chunkers, corpora, help, stemmers, taggers, tokenizers, unzip each subdirectory)
    - Use nltk.data.find(“.”) to check the directory

Named Entity Extraction - Example

- Tokenize sentence:

```
1  import nltk
2  from nltk import word_tokenize
3  raw = """Jonh was born in Liverpool, to Julia and Alfred Lennon"""
4  tokens = word_tokenize(raw)
5  tokens
```

```
['Jonh',
 'was',
 'born',
 'in',
 'Liverpool',
 ',',
 'to',
 'Julia',
 'and',
 'Alfred',
 'Lennon']
```

Named Entity Extraction - Example

- Pos-tag:

```
1 tagged = nltk.pos_tag(tokens)
2 print(tagged)
```

```
[('Jonh', 'NNP'), ('was', 'VBD'), ('born', 'VBN'), ('in', 'IN'), ('Liverpool', 'NNP'), (',', ','), ('to', 'TO'), ('Julia', 'NNP'), ('and', 'CC'), ('Alfred', 'NNP'), ('Lennon', 'NNP')]
```

- Meaning of each tag:

```
1 nltk.help.upenn_tagset('NNP')
```

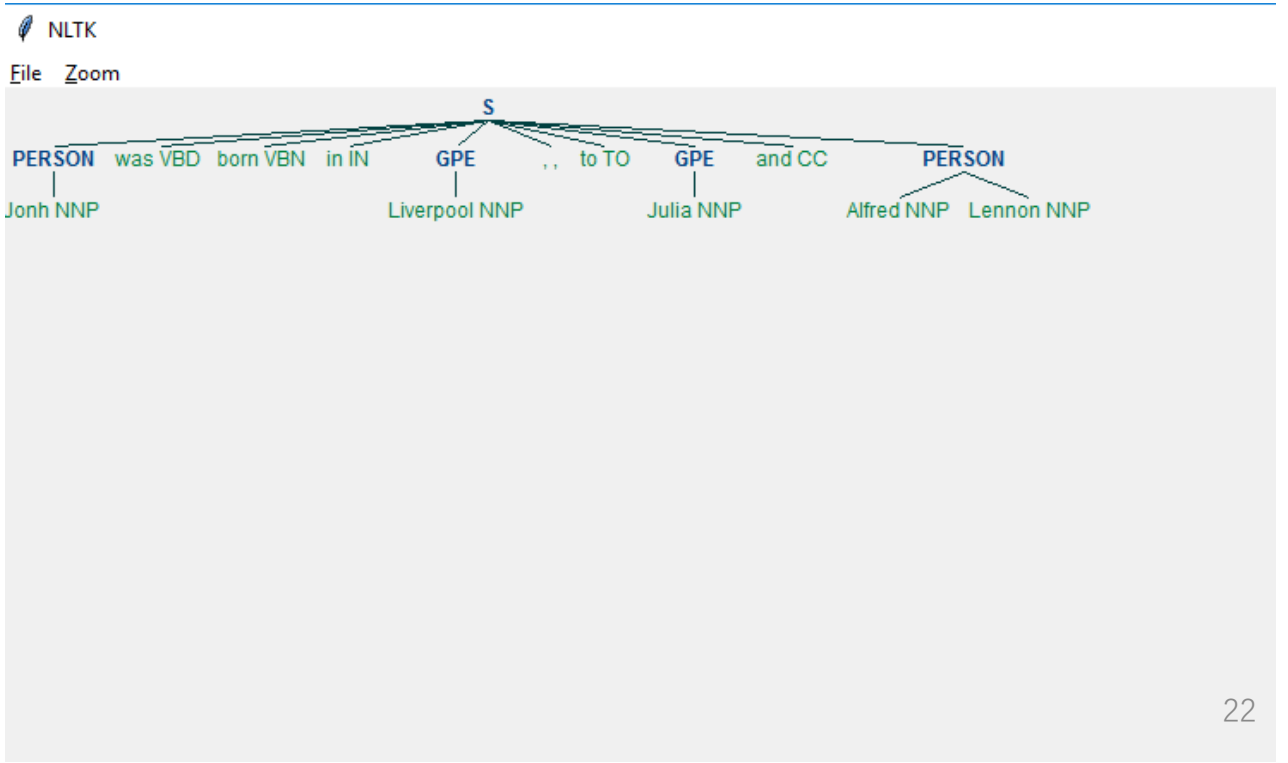
```
NNP: noun, proper, singular
    Motown Venneboerger Czestochwa Ranzer Conchita Trumplane Christos
    Oceanside Escobar Kreisler Sawyer Cougar Yvette Ervin ODI Darryl CTCA
    Shannon A.K.C. Meltex Liverpool ...
```

Named Entity Extraction - Example

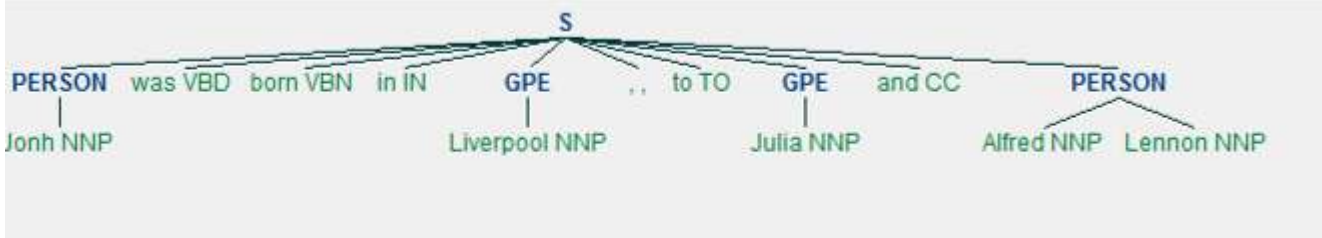
- Chunking:
  - **`ne_chunk`** needs part-of-speech annotations to add **`NE`** labels to the sentence. The output of the **`ne_chunk`** is a **`nltk.Tree`** object
  - **`ne_chunk`** produces 2-level trees:
    - Nodes on Level-1: outsides any chunk
    - Nodes on Level-2: inside a chunk (the label of the chunk is denoted by the label of the subtree)

```
1  from nltk import word_tokenize, pos_tag, ne_chunk
2  chunks = ne_chunk(pos_tag(word_tokenize(raw)))
3  print(chunks)
4  chunks.draw()
```

```
(S
  (PERSON Jonh/NNP)
  was/VBD
  born/VBN
  in/IN
  (GPE Liverpool/NNP)
  ,/,
  to/TO
  (GPE Julia/NNP)
  and/CC
  (PERSON Alfred/NNP Lennon/NNP))
```



Named Entity Extraction - Example



- Parse the tree to extract named entities:
  - Traverse the level-1 nodes in the tree:

```
for i in chunks:
    print(i, type(i))
```

```
(PERSON Jonh/NNP) <class 'nltk.tree.Tree'>
('was', 'VBD') <class 'tuple'>
('born', 'VBN') <class 'tuple'>
('in', 'IN') <class 'tuple'>
(GPE Liverpool/NNP) <class 'nltk.tree.Tree'>
(',', ',') <class 'tuple'>
('to', 'TO') <class 'tuple'>
(GPE Julia/NNP) <class 'nltk.tree.Tree'>
('and', 'CC') <class 'tuple'>
(PERSON Alfred/NNP Lennon/NNP) <class 'nltk.tree.Tree'>
```

- Parse the sub-tree to extract named entities:
  - Traverse the level-2 nodes in the sub-tree:

```
for i in chunks:
    if type(i) == Tree:
        print('Chunk detect!')
        chunk_phrase = []
        for token, pos in i.leaves():
            print(token, pos)
```

```
Chunk detect!
Jonh NNP
Chunk detect!
Liverpool NNP
Chunk detect!
Julia NNP
Chunk detect!
Alfred NNP
Lennon NNP
```

Named Entity Extraction - Noun Phrase Chunking

- Instead of using the built-in chunking library, you can also define own rule for noun phrase/entity chunking
  - In order to create an NP-chunker, we will first define a chunk grammar, consisting of rules that indicate how sentences should be chunked. (**`Hint: grammar= "NP:{rule1} \n {rule2} \n {rule3} ...`**", rule1 is a regular expression over pos-tags, for example, rule1 can be "**`<NN.*>+`**", means one or more nouns can be a single chunk)
  - Using this grammar, we create a chunk parser (**`Hint: nltk.RegexpParser(grammar))`**
  - Using the **`parse`** method to parse taged input
  - The result is a tree, which we can either print , or display graphically

Named Entity Extraction- Exercise

- First part finish now you need to be able to extract all the named entity given any text inputs.
- Exercise: Download and finish the exercise in *EntityExtraction_Exercise.ipynb*
  1. Extract all entities as well as its type/label.
     - Input: *"John was born in Liverpool, to Julia and Alfred Lennon"*
     - Output: '*John* ': '*PERSON*', '*Liverpool*': '*GPE*', '*Julia*': '*GPE*', '*Alfred Lennon*': '*PERSON*'
  2. Extract only entities of specific type (For example, PERSON).
     - Input: *"John was born in Liverpool, to Julia and Alfred Lennon"*
     - Output: '*John*', '*Alfred Lennon*'
  3. Define your own chunk grammars for noun phrase chunking, instead of using **`ne_chunk`**
     - Input1: *"John was born in Liverpool, to Julia and Alfred Lennon"*
     - Output1: '*John*', '*Liverpool*', '*Julia*', '*Alfred Lennon*'
     - Input2*: "the little dog barked at the cat"*
     - Output2*: 'little dog', 'cat'*

Relation Extraction

- Relation Extraction standardly consists of identifying specified relations between Named Entities.
- Typically, we will focus on extracting binary relations
  - For example, assuming that we can recognize ORGANIZATIONs and LOCATIONs in text, we might want to also recognize pairs *(o, l)* of these kinds of entities such that *located-in(CMU , Pittsburgh)*
- Relation extractors ([3] https://web.stanford.edu/~jurafsky/slp3/18.pdf)
  - Hand-written patterns (our focus in this project)
  - Supervised machine learning
  - Semi-supervised and unsupervised machine learning
    - Boostrap
    - Distant supervision
    - Unsupervised from web

Handwritten Patterns – Hyponym Relationship

- Patterns for learning hyponyms (KnowItAll [1],Microsoft Probase[2] https://www.microsoft.com/en-us/research/project/probase/).
  - Hearst pattern (NP stands for noun phrase/named entity):

ID	Pattern
1	*NP* such as {*NP*,}*{(or | and)} *NP*
2	such *NP* as {*NP*,}*{(or | and)} *NP*
3	*NP*{,} including {*NP*,}*{(or | and)} *NP*
4	*NP*{,*NP*}*{,} and other *NP*
5	*NP*{,*NP*}*{,} or other *NP*
6	*NP*{,} especially {*NP*,}*{(or | and)} *NP*

  - High quality pattern that can be used to extract *isA* (hypernym-hyponym) relationship
    - "... *domestic animals such as cats* ...", we obtain the relationship: "cat *isA* animal"
    - "... we provide tours to cities such as Paris, Nice, and Monte Carlo...", we obtain the relationships "Paris isA city", "Nice isA city" and "Monte Carlo isA city"

Hearst Pattern - Exercise

- Extract the hyponym relation using the Hearst Pattern
- Implement Hearst Patterns for hyponym relation extraction
  - Requires noun-phrase chunking and then regex pattern matching where these patterns are relevant Hearst patterns.
- Download and finish the exercise in *HypernymExtraction-Exercise.ipynb*

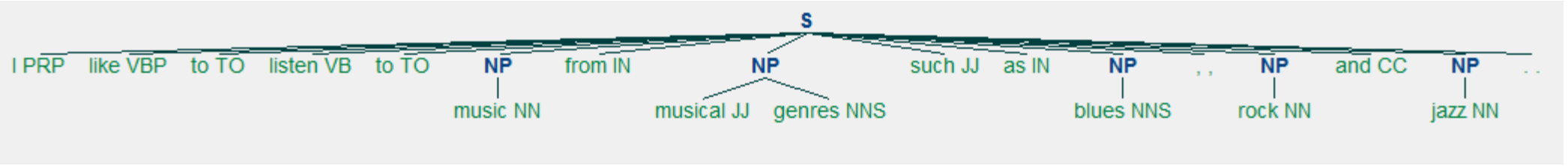
Hearst Pattern

- **Step0: Hearst Pattern matching practice**
  - In the following example, we only show one regex pattern example for Hearst pattern

```
import re
regex = r"(NP_\w+ (, )?such as (NP_\w+ ?(, )?(and |or )?)+)"
test_str = "NP_1 such as NP_2, NP_3 and NP_4 "
matches = re.search(regex, test_str)
if matches:
    print(matches.group(0))
```

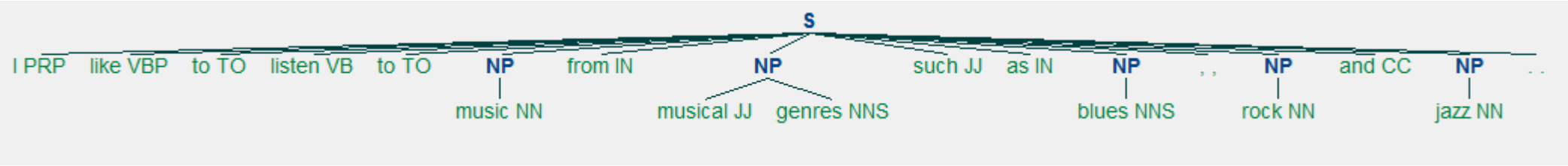
```
NP_1 such as NP_2, NP_3 and NP_4
```

- **Step1: Noun-phrase chunking**
  - According to the Hearst pattern, we need to perform noun-phrase chunking first (different with previous exercise, we do not need the entities/np, instead, we need the chunk tree object as the result)
  - Input: *"I like to listen to music from musical genres,such as blues,rock and jazz."*
  - *Output:*



Hearst Pattern

- **Step2: Prepare the chunked result for subsequent Hearst pattern matching**
  - Traverse the chunked result, if the label is “NP”, then merge all the words in this chunk and add a prefix “NP_”
  - Input:



  - Output: “*I like to listen to NP_music from NP_musical_genre , such as NP_blues , NP_rock and NP_jazz .*”
- Hint:
  - After preparation, all the NP are detected and added a prefix “NP_”; Moreover, every token has been separated with a space (“ ”), including the commas/periods.
  - Use WordNetLemmatizer to lemmatize words (`from nltk.stem import WordNetLemmatizer`) (NP_musical_genre not NP_musical_genres)

Hearst Pattern

- **Step3: Refinement chunking (Optional)**
  - If two or more NPs next to each other should be merged into a single NP.
  - Input: "*NP_foo NP_bar blah blah*"
  - Output: "*NP_foo_bar blah blah*“

Hearst Pattern

- **Step4: Find the hypernym and hyponyms on processed chunked results**
  - Define Hearst patterns. Besides the regex, we also need to specify whether the hypernym is in the first part or the second part in the pattern.
  - For example, in the pattern "NP1 such as NP2 AND NP3", the hypernym is the first part of the pattern; in the pattern "NP1 , NP2 and other NP3", the hypernym is the last part of the pattern.
  - Hint: Two regular expression examples are given in the notebook (Hearst pattern 1 and Hearst pattern 4/5). Try to define the regular expression for remaining patterns.
  - After regex matching, find all the NPs and extract the hypernym and hyponym pairs based on the "first" or "last" attribute.
  - Input: "*I like to listen to NP_music from NP_musical_genre , such as NP_blues , NP_rock and NP_jazz .*"
  - Output: *[('NP_blue', 'NP_musical_genre'), ('NP_rock', 'NP_musical_genre'), ('NP_jazz', 'NP_musical_genre')]*

Hearst Pattern

- Finally: Merge every step to get the extractor
  - Your extractor need to read the input corpus and parse sentence by sentence (you can use **`nltk.sent_tokenize`**).
  - Remember to clean the output by removing the “NP_” and “_”
- Test case:
  - Input: *“I like to listen to music from musical genres such as blues, rock and jazz. He likes to play basketball , football and other sports.”*
  - Output: *[('blues', 'genres'), ('rock', 'genres'), ('jazz', 'genres'), ('basketball', 'sports'), ('football', 'sports')]*

Test Case

- Test cases for Hearst Pattern:

Pattern	Example	Extraction
1	Agar is a substance prepared from a mixture of red algae, such as Gelidium, for laboratory or industrial use.	('Gelidium', 'red algae')
2	... works by such authors as Herrick, Goldsmith, and Shakespeare.	('Herrick', ' author'), ('Goldsmith', ' author'), ('Shakespeare', ' author')
3	...all common law countries, including Canada and England.	('Canada', 'common law country'), ('England', 'common law country')
4/5	... bistros, coffee shops, and other cheap eating places.	('bistro', ' cheap eating place'), ('coffee shop', ' cheap eating place')
6	...most European countries, especially France, England, and Spain.	('France', 'European country'), ('England', 'European country'), ('Spain', 'European country')

Test Case

- Define other patterns listed in Hearst pattern table and test your program to see whether the hypernym relation are correctly extracted.
- Hint:
  - Handle the special cases that in pattern 2 and 4, such/other will be tagged as “JJ”, which will results in the “such NP” be chunked into a single noun phrase.

ID	Pattern
1	*NP* such as {*NP*,}*{(or | and)} *NP*
2	such *NP* as {*NP*,}*{(or | and)} *NP*
3	*NP*{,} including {*NP*,}*{(or | and)} *NP*
4	*NP*{,*NP*}*{,} and other *NP*
5	*NP*{,*NP*}*{,} or other *NP*
6	*NP*{,} especially {*NP*,}*{(or | and)} *NP*

Pattern for Richer Relations

- Similar with the hypernym-hyponym relationship, we can also define patterns for other relations
- Intuition: relations often hold between specific entities
  - *located-in (ORGANIZATION, LOCATION), founded (PERSON, ORGANIZATION), cures (DRUG, DISEASE)*
  - Make use of named entity recognition and syntax pattern to define the extraction rules for each relation
- Pros:
  - Human patterns tend to be high-precision
  - Can be tailored to specific domains
- Cons:
  - Human patterns are often low-recall (high-quality patterns are rere).
  - A lot of work to think of all possible patterns!
  - Don’t want to have to do this for every relation!

Supervised Learning for Relation Extraction

- Choose a set of relations we'd like to extract
- Choose a set of relevant named entities
- Find and label data
  - Choose a representative corpus
  - Label the named entities in the corpus
  - Hand-label the relations between these entities
  - Break into training, validation, and test
- Train a classifier on the training set
  - Feature can include bag of words, type of the entity, phrase chunk paths, dependency-tree paths..... [3]

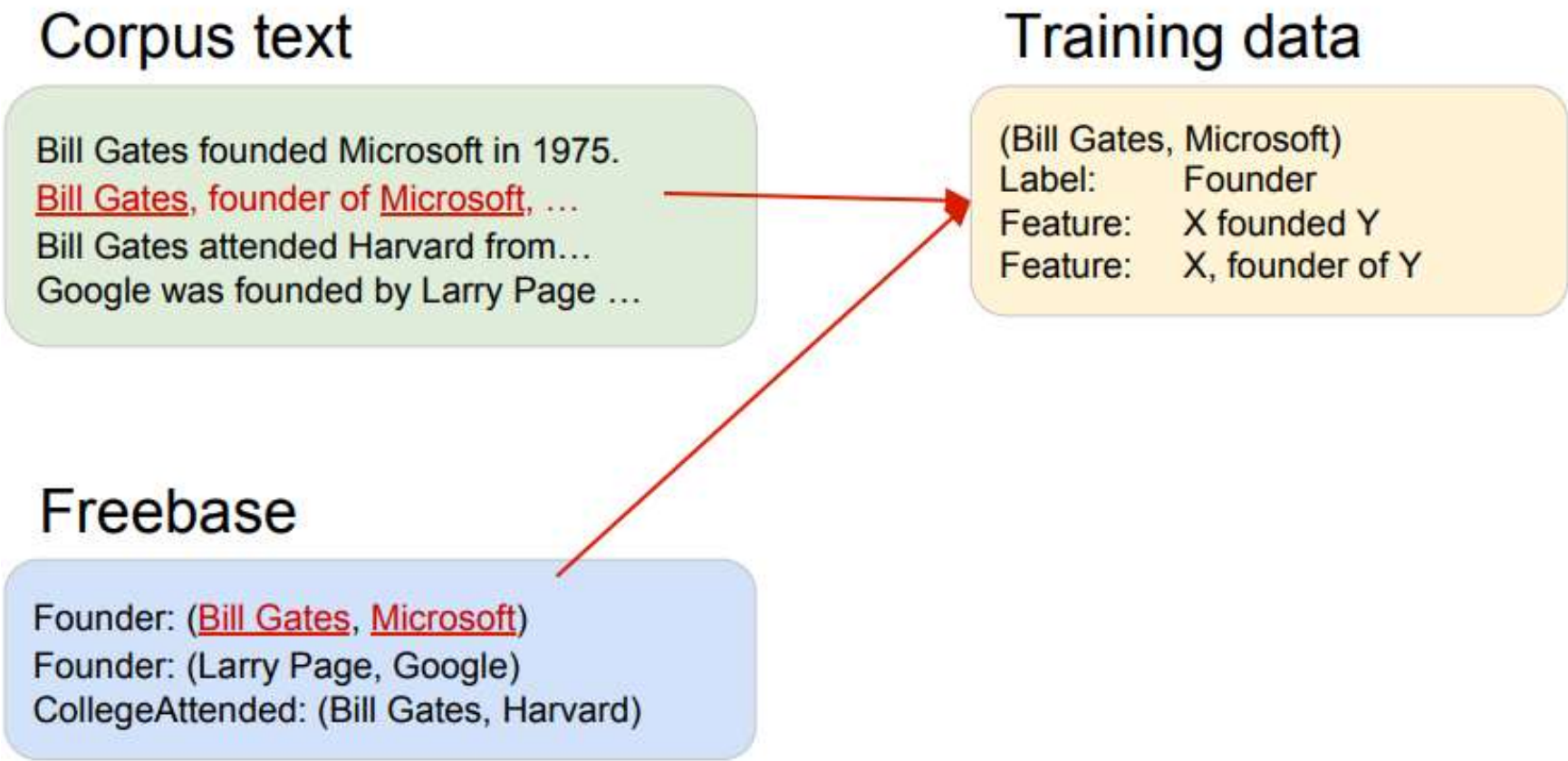
Supervised Learning for Relation Extraction

- Supervised approach can achieve high accuracy for some relations if we have lots of hand-labeled training data
- But has significant limitations!
  - Labeling 5,000 relations (+ named entities) is expensive
  - Doesn’t generalize to different relations

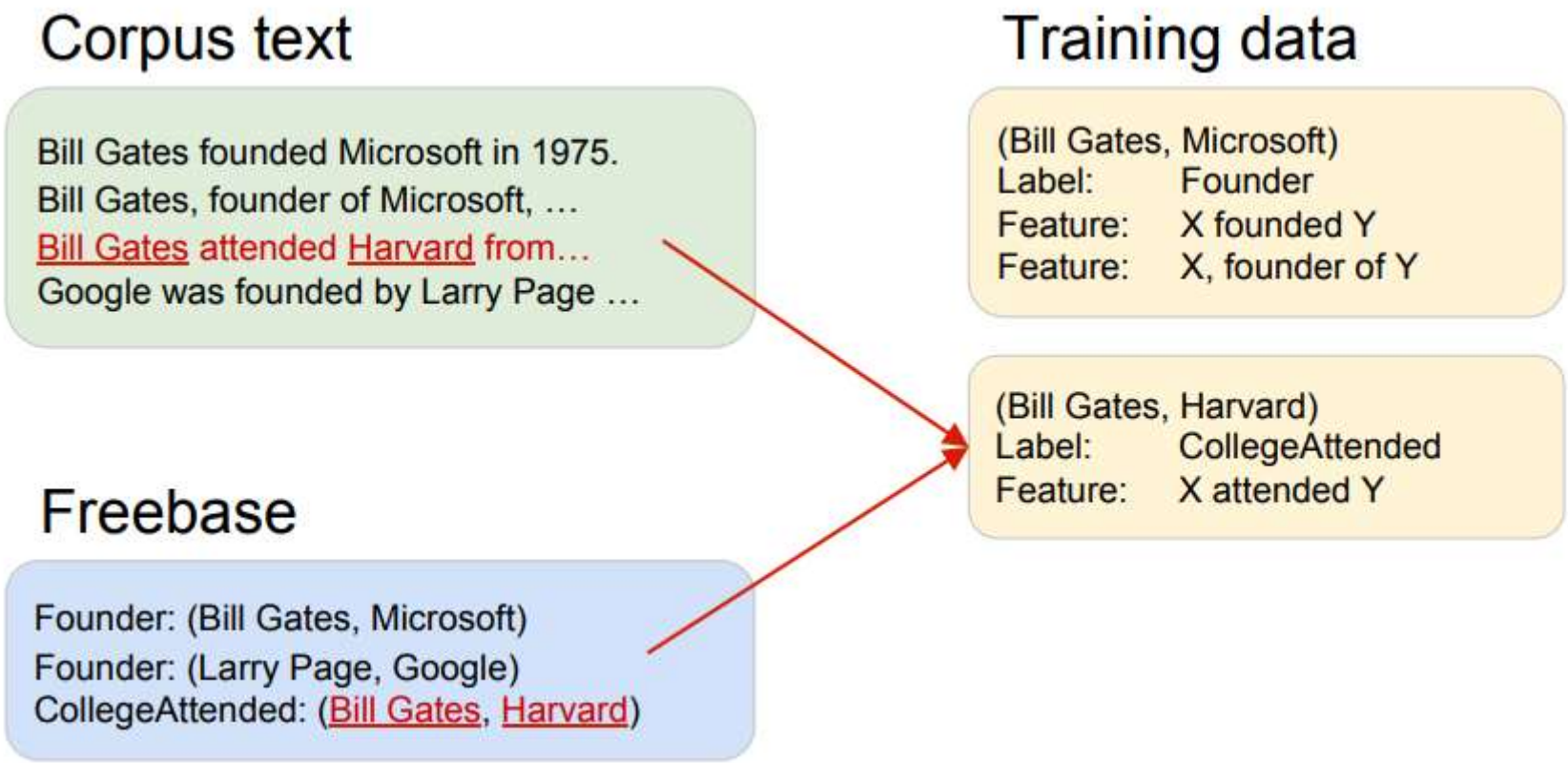
Distant Supervision for Relation Extraction

- Hypothesis: If two entities belong to a certain relation, any sentence containing those two entities is likely to express that relation
- Key idea: Use a database of relations to get lots of noisy training examples instead of using hand-labeled corpus
- For example
  - Use existing KG relations, e.g., Freebase
  - Collecting training data from corpus by preparing positive and negative samples
  - Train classifier for relation extraction

Distant Supervision - Example



Distant Supervision - Example



Open Information Extraction

- Open information extraction (open IE) refers to the extraction of relation tuples, typically binary relations, from plain text.
- The central difference from other relation/information extraction is that schema for these relations does not need to be specified in advance.
- Stanford OpenIE System [4]
  - First learn a classifier for splitting a sentence into shorter utterances
  - Then appeal to natural logic to maximally shorten these utterances while maintaining necessary context

Information Extraction - Exercise

- Download and finish the exercise in *OpenIE-Exercise.ipynb*
  - Practice relation/knowledge extraction using open information extraction tookit, Stanford OpenIE(https://nlp.stanford.edu/software/openie.html)
  - Construct a simple KG from extracted knowledge triples and visualize it.
- Input:
  - "*Barack Obama was the 44th president of the United States, and the first African American to serve in the office. On October 3, 1992, Barack Obama married Michelle Robinson at Trinity United Church in Chicago.*"
- Output: