Lab5 - Assignment 5 about extraction of properties

This notebook describes the LAB-5 assignment of the Text Mining course. It is about Property Extraction.

Due: 17 Mar at 23:59

How to submit: Please submit your assignment using Canvas (see *Assignments* -> *Lab Session Property Extraction*). Convert your notebook to PDF (in JupyterLab, this can be done by clicking on *File* in the menu bar, select *Export Notebook As*, then select *Export Notebook to PDF*)

Points: each exercise is suffixed with the number of points you can obtain for the exercise.

Assignment goals:

- Get insight into the challenges of entity property extraction.
- Learn how to build a transparent property extraction method based on patterns.
- Get insight into the pros and cons of two pattern-based property extraction methods.
- Be able to run your extractors on unseen documents from Wikipedia.
- Be able to evaluate property extractors.

In this assignment, the main focus lies on creating your own pattern-based property extractors. You are then going to run them on Wikipedia texts, evaluate them against gold values, and reflect on their relative performance.

We recommend that you go through the notebooks in the following order:

- *Read the assignment (see below)*
- *Lab5-Property-extraction.ipynb*
- *Answer the questions of the assignment (see below) using the provided notebooks and submit*

Hint: in the explanation notebook, we had an example about extraction of properties with substring matching and with dependencies. You can use much of that code here, but make sure you make the right adjustments.

Good luck & have fun!

In [1]:
```
import spacy
import lab5_utils as utils

model="en_core_web_sm"

nlp = spacy.load(model)
# print("Info: Loaded model '%s'" % model)
```

1. Extracting properties with substring matching (12 points)

Exercise 1a Write code that extracts the birth year of a person by using substring matching. (4 points)

In [2]:
```
def extract_birth_year_regex(doc, patterns):
    # Extract the birth year of a person with regular expressions
    property_value_type='DATE'
    target_entity_type='PERSON'

    # the following 3 lines merge entities and noun chunks into one toke
n
    # this is useful in our cases, so we will always do it.
    spans = list(doc.ents) + list(doc.noun_chunks)
    for span in spans:
        span.merge()

    relations = {}

    # step Ia - generate possible property values
    dates=utils.get_entities_of_type(property_value_type, doc)
    for date in dates:
        # step Ib - is one of our patterns found before the date
        if utils.pattern_found_on_the_left(doc, date.i, patterns):
            # step II - find the closest entity of some target type
            pers=utils.find_closest_entity(doc.ents, date.idx, target_en
tity_type)
            # step III - normalize the year
            year=utils.extract_year_from_date(date.text)
            if year and pers:
                relations[pers]=year

    return relations
```

Exercise 1b Test your *birth year substring matching extractor* in the following way.

- Write a sentence on which you expect that the extractor *WILL* work.
- Write a sentence on which you expect that the extractor *WILL NOT* work.

Run your extractor on both sentences and print the results. Make sure that the results are as expected. (2 points)

In [3]:
```
born_patterns=['born in', 'birthdate', 'born on']

text='Peter was born in 1975.'
text2 = "Peter Pan's dog was born in 1990"

print( "This sentence \033[1m WILL\033[0m work:",text)
birth_year_relations=extract_birth_year_regex(nlp(text), born_patterns)
print(birth_year_relations)
print()
print( "This sentence \033[1m WILL NOT\033[0m work:",text2)
birth_year_relations2=extract_birth_year_regex(nlp(text2), born_pattern
s)
print(birth_year_relations2)
```

```
This sentence  WILL work: Peter was born in 1975.
{'Peter': 1975}

This sentence  WILL NOT work: Peter Pan's dog was born in 1990
{}
```

Exercise 1c Write code that extracts the manufacturer of a device by using substring matching. (4 points)

In [3]:
```
def extract_manufacturer_regex(doc, patterns, main_entity):
    # Extract the manufacturer of a device by using regular expressions
    property_value_type='ORG'
    target_entity_type='PRODUCT'

    # the following 3 lines merge entities and noun chunks into one toke
n
    # this is useful in our cases, so we will always do it.
    spans = list(doc.ents) + list(doc.noun_chunks)
    for span in spans:
        span.merge()

    relations = {}

    manus =utils.get_entities_of_type(property_value_type, doc)

    for manu in manus:
        if utils.pattern_found_on_the_left(doc, manu.i, patterns):
            prod=utils.find_closest_entity(doc.ents, manu.idx, target_en
tity_type)
            if not prod:
                prod = main_entity
            if manu and prod:
                relations[prod]=manu.text
    return relations
```

Exercise 1d Test your *manufacturer substring matching extractor* in the following way.

- Write a sentence on which you expect that the extractor *WILL* work.
- Write a sentence on which you expect that the extractor *WILL NOT* work.

Run your extractor on both sentences and print the results. Make sure that the results are as expected. (2 points)

In [5]:
```
manu_predicates=['manufactured', 'produced', 'developed by', 'developed
']
main_entity = 'iPhone'
sentence='the iPhone was developed by Apple in 2000.'
sentence2 = 'Apple made the iPad in 2005 .'

print( "This sentence \033[1m WILL\033[0m work:",sentence)
manu_relations=extract_manufacturer_regex(nlp(sentence), manu_predicate
s, main_entity)
print(manu_relations)
print()
print( "This sentence \033[1m WILL NOT\033[0m work:",sentence2)
manu_relations2=extract_manufacturer_regex(nlp(sentence2), manu_predicat
es, main_entity)
print(manu_relations2)
```

```
This sentence  WILL work: the iPhone was developed by Apple in 2000.
{'iPhone': 'Apple'}

This sentence  WILL NOT work: Apple made the iPad in 2005 .
{}
```

2. Extracting properties by using dependency information (12 points)

In [4]:
```
def fitting_dependency_year(token, predicates):
    """
    Check whether the we find the right keyword in the correct part of t
he dependency tree.
    """
    # Find prepositional objects that have a head with dependency label
'agent'
    # and its head has a dependency label 'acl'
    # Also, we make sure that the head of the head of our object is one
of our keywords.
#     if token.dep_ == 'nsubjpass' and token.head.dep_ == 'ROOT':
    if token.dep_ == 'pobj' and token.head.dep_ == 'prep' and token.hea
d.head.dep_ =='ROOT':
        pred=token.head.head
        if pred.text in predicates:
            return True
        else:
            return False
    else:
        return False
```

Exercise 2a Write code that extracts the birth year of a person by using dependency information. (4 points)

In [5]:
```
def extract_birth_year_dep(doc, predicates):

    property_value_type='DATE'
    target_entity_type='PERSON'

    # the following 3 lines merge entities and noun chunks into one toke
n
    # this is useful in our cases, so we will always do it.
    spans = list(doc.ents) + list(doc.noun_chunks)
    for span in spans:
        span.merge()

    relations={}

    # step Ia - generate possible property values
    dates=utils.get_entities_of_type(property_value_type, doc)

    for date in dates:
        # step Ib - do we find the right keyword in the correct part of
the dependency tree?
        if fitting_dependency_year(date, predicates):
            # step II - find the closest entity of some target type
            person=utils.find_closest_entity(doc.ents, date.idx, target_
entity_type)
            year =utils.extract_year_from_date(date.text)
            if person and year:
                relations[person]=year
    return relations
```

In [6]:
```
def extract_birth_year_dep_entity(doc, predicates, entity):

    property_value_type='DATE'
    target_entity_type='PERSON'

    # the following 3 lines merge entities and noun chunks into one toke
n
    # this is useful in our cases, so we will always do it.
    spans = list(doc.ents) + list(doc.noun_chunks)
    for span in spans:
        span.merge()

    relations={}

    # step Ia - generate possible property values
    dates=utils.get_entities_of_type(property_value_type, doc)

    for date in dates:
        # step Ib - do we find the right keyword in the correct part of
the dependency tree?
        if fitting_dependency_year(date, predicates):
            # step II - find the closest entity of some target type
            person=utils.find_closest_entity(doc.ents, date.idx, target_
entity_type)
            year =utils.extract_year_from_date(date.text)
            # Devices are often not recognized properly by SpaCy -
            # if we find no device, we assume that the relation is about
the main entity of the document
            if not person:
                person=main_entity
            if year and person:
                relations[person]=year
    return relations
```

Exercise 2b Test your *birth year dependency extractor* in the following way.

- Write a sentence on which you expect that the extractor *WILL* work.
- Write a sentence on which you expect that the extractor *WILL NOT* work.

Run your extractor on both sentences and print the results. Make sure that the results are as expected. (2 points)

In [9]:
```
born_patterns=['born', 'birthdate']
sentence='Peter was born in 1975.'
sentence2 = "In 1975 Peter's dog was born."

print( "This sentence \033[1m WILL\033[0m work:",sentence)
birth_year_relations=extract_birth_year_dep(nlp(sentence), born_pattern
s)
print(birth_year_relations)
print()
print( "This sentence \033[1m WILL NOT\033[0m work:",sentence2)
birth_year_relations2=extract_birth_year_dep(nlp(sentence2), born_patter
ns)
print(birth_year_relations2)
```

```
This sentence  WILL work: Peter was born in 1975.
{'Peter': 1975}

This sentence  WILL NOT work: In 1975 Peter's dog was born.
{}
```

Exercise 2c Write code that extracts the manufacturer of a device by using dependency information. (4 points)

In [7]:
```
def fitting_dependency_manu(token, predicates):
    """
    Check whether the we find the right keyword in the correct part of t
he dependency tree.
    """
    # Find prepositional objects that have a head with dependency label
'agent'
    # and its head has a dependency label 'acl'
    # Also, we make sure that the head of the head of our object is one
of our keywords.
    if token.dep_ == 'pobj' and token.head.dep_ == 'agent' and token.hea
d.head.dep_ =='ROOT':
        pred=token.head.head
        if pred.text in predicates:
            return True
        else:
            return False
    else:
        return False
```

In [8]:
```
def extract_manufacturer(doc, predicates, main_entity):

    property_value_type='ORG'
    target_entity_type='PRODUCT'

    # the following 3 lines merge entities and noun chunks into one toke
n
    # this is useful in our cases, so we will always do it.
    spans = list(doc.ents) + list(doc.noun_chunks)
    for span in spans:
        span.merge()

    relations={}

    # step Ia - generate possible property values
    manus=utils.get_entities_of_type(property_value_type, doc)

    for manu in manus:
        # step Ib - do we find the right keyword in the correct part of
the dependency tree?
        if fitting_dependency_manu(manu, predicates):
            # step II - find the closest entity of some target type
            device=utils.find_closest_entity(doc.ents, manu.idx, target_
entity_type)
            # Devices are often not recognized properly by SpaCy -
            # if we find no device, we assume that the relation is about
the main entity of the document
            if not device:
                device=main_entity
            if device and manu:
                relations[device]=manu.text
    return relations
```

Exercise 2d Test your *manufacturer dependency extractor* in the following way.

- Write a sentence on which you expect that the extractor *WILL* work.
- Write a sentence on which you expect that the extractor *WILL NOT* work.

Run your extractor on both sentences and print the results. Make sure that the results are as expected. (2 points)

In [12]:
```
manu_predicates=['manufactured', 'produced', 'developed']
main_entity = 'iPhone'
main_entity2 ='Walkman'
sentence='the iPhone was developed by Apple in 2000.'
sentence2 = 'Apple made iPad in 2005.'
sentence3= 'Walkman is a brand of portable media players manufactured by
Sony. The original Walkman, released in 1979, was a portable cassette pl
ayer that changed listening habits by allowing people to listen to music
of their choice on the move. It was devised by Sony founders Masaru Ibuk
a and Akio Morita, who felt Sony existing portable player was too unwiel
dy and expensive'
print( "This sentence \033[1m WILL\033[0m work:",sentence)
manu_relations=extract_manufacturer(nlp(sentence), manu_predicates, main
_entity)
print(manu_relations)

print()
print( "This sentence \033[1m WILL NOT\033[0m work:",sentence2)
manu_relations2=extract_manufacturer(nlp(sentence2), manu_predicates, ma
in_entity)
print(manu_relations2)
print( "This sentence \033[1m WILL NOT\033[0m work:",sentence3)
manu_relations3=extract_manufacturer(nlp(sentence2), manu_predicates, ma
in_entity2)
print(manu_relations3)
```

```
This sentence  WILL work: the iPhone was developed by Apple in 2000.
{'iPhone': 'Apple'}

This sentence  WILL NOT work: Apple made iPad in 2005.
{}
This sentence  WILL NOT work: Walkman is a brand of portable media player
s manufactured by Sony. The original Walkman, released in 1979, was a por
table cassette player that changed listening habits by allowing people to
listen to music of their choice on the move. It was devised by Sony found
ers Masaru Ibuka and Akio Morita, who felt Sony existing portable player
was too unwieldy and expensive
{}
```

3. Running and evaluating extractors on Wikipedia (8 points)

We will run our extractors on 50 documents about people and 50 documents about devices. We provide code to load the lists of entities and the gold values.

In [13]:
```
import json
with open("birthyears.json", 'rb') as f:
    gold_birthyears=json.load(f)
    wiki_people=list(gold_birthyears.keys())

with open("manufacturers.json", 'rb') as f:
    gold_manufacturers=json.load(f)
    wiki_devices=list(gold_manufacturers.keys())
print(wiki_people)
```

```
['Al Pacino', 'Alan Rickman', 'Albert Finney', 'Alyson Hannigan', 'Andie
MacDowell', 'Andrew Lloyd Webber', 'Andrzej Wajda', 'Andrzej Żuławski', '
Angela Davis', 'Anthony Quinn', 'Antonio Banderas', 'Ashley Judd', 'Ava G
ardner', 'Barbara Stanwyck', 'Ben Elton', 'Bernardo Bertolucci', 'Betty M
arsden', 'Billy Wilder', 'Blake Edwards', 'Bob Black', 'Bob Keeshan', 'Br
ad Pitt', 'Cameron Diaz', 'Carmen Miranda', 'Carole Lombard', 'Catherine
Deneuve', 'Cesare Zavattini', 'Chandra Levy', 'Charlton Heston', 'Chaz Bo
no', 'Christine McVie', 'Christopher Lambert', 'Christopher Lee', 'Clark
Gable', 'Clint Eastwood', 'Clive Sinclair', 'Cybill Shepherd', 'Dan Aykro
yd', 'Dannii Minogue', 'Dave Cutler', 'David Blaine', 'David Boies', 'Dav
id Gauthier', 'David Jason', 'David Niven', 'Denise Richards', 'Desmond L
lewelyn', 'Don Siegel', 'Dudley Moore', 'Dustin Hoffman']
```

The lists `wiki_people` and `wiki_devices` contain the names of 50 people and 50 devices, respectively.

The dictionaries `gold_birthyears` and `gold_manufacturers` contain gold values for each of these entities.

We provide a function that evaluates your extracted property values against known ("gold") property values. The function returns three evaluation scores: precision, recall, f1-score. You can find call this function as follows:

```
utils.evaluate_property(system_json, gold_json)
```

(make sure to replace the system_json and the gold_json with the concrete dictionaries you are comparing, depending on the property and the method)

Now that we have stored the gold values for both properties in our dictionaries `gold_birthyears` and `gold_manufacturers`, and written the evaluation function, we need to obtain the system output as well and then perform evaluation.

For this purpose, we will run our extractors on texts about the same 50 people and 50 devices from Wikipedia. As in the explanation notebook, we will use the `Wikipedia` library for this purpose. Same as in the explanation notebook, we will only process the first three sentences.

In exercises 3a and 3b, we will run all our four processing functions and store the results in four different dictionaries. Then, in exercise 3c, we will run the evaluation function four times to compute precision, recall, and F1-score for all four functions.

Exercise 3a Run your two extractors about birth years of people (from exercise 1a and 1b) on all 50 documents about people. Save the extracted values in two different dictionaries: `birthyear_regex` and `birthyear_dep`. (3 points)

In [14]:
```
import wikipedia
```

In [15]:
```
# al = 'Al Pacino'
# print(al)
# al = al.replace(" ", "")
# print(al)
# wp = wikipedia.page(al.replace(" ", ""))
# print(wp)
```

In [16]:
```
texts_p={}
# print(wiki_people)

for entity in wiki_people:
        try:
            a= entity

            wp = wikipedia.page(a.replace(" ", ""))
            first_three_sentences=wp.content.split('.')[:3]
            entity_text=('.').join(first_three_sentences)
            texts_p[entity] = entity_text
#             print(entity_text)
#             print()
        except:
            pass
#             print(entity, "was not found on wikipedia")
#             print()
```

In [17]:
```
birthyear_regex={}
birthyear_dep={}

# texts_p
# print(wiki_people)
born_patterns=['born in', 'birthdate', 'born on','(born','born']




for entity in wiki_people:
    if entity in texts_p.keys():
#         print('entity is',entity)
#         print('manu_predicates',born_patterns)
#         print(nlp(texts_p[entity]))
        a=extract_birth_year_regex(nlp(texts_p[entity]), born_patterns)
#         print("relation  is",a)
        for p in a.values():
            birthyear_regex[entity] = str(p)



for entity in wiki_people:
    if entity in texts_p.keys():
#         print('entity is',entity)
#         print('born_predicates',born_patterns)
#         print(nlp(texts_p[entity]))
        a=extract_birth_year_dep(nlp(texts_p[entity]), born_patterns)
#         print("relation  is",a)
#         print()
        for p in a.values():
            birthyear_dep[entity] = str(p)


print(birthyear_regex)
print(birthyear_dep)
```

```
{'Al Pacino': '1940', 'Alyson Hannigan': '1974', 'Andie MacDowell': '1958
', 'Andrew Lloyd Webber': '1948', 'Angela Davis': '1944', 'Antonio Bander
as': '1960', 'Bob Black': '1951', 'Brad Pitt': '1963', 'Cameron Diaz': '1
972', 'Catherine Deneuve': '1943', 'Christine McVie': '1943', 'Clint East
wood': '1930', 'Cybill Shepherd': '1950', 'Dan Aykroyd': '1952', 'Dave Cu
tler': '1942', 'David Blaine': '1973', 'David Boies': '1941', 'David Gaut
hier': '1932', 'Denise Richards': '1971', 'Dustin Hoffman': '1937'}
{'David Gauthier': '1932'}
```

Exercise 3b Run your extractors about manufacturers of devices (from exercise 2a and 2b) on all 50 documents about devices. Make sure you only process the first three sentences from each document. Save the extracted values in two lists: `manufacturers_regex` and `manufacturers_dep` . (3 points)

In [20]:
```
def fitting_dependency_manu(token, predicates):
    """
    Check whether the we find the right keyword in the correct part of t
he dependency tree.
    """
    # Find prepositional objects that have a head with dependency label
'agent'
    # and its head has a dependency label 'acl'
    # Also, we make sure that the head of the head of our object is one
of our keywords.
    if token.dep_ == 'pobj' and token.head.dep_ == 'agent' and token.hea
d.head.dep_ =='acl':
        pred=token.head.head
        if pred.text in predicates:
            return True
        else:
            return False
    else:
        return False
```

In [18]:
```
textsdev={}
# wiki_devices
for entity in wiki_devices:
    try:
#         print(entity)
        a=entity
        wp = wikipedia.page(a.replace(" ", ""))
        # get the first 3 sentences of a wikipedia article
        first_three_sentences=wp.content.split('.')[:3]
        entity_text=('.').join(first_three_sentences)
        # create a dictionary (JSON) where the key is your entity, and t
he value is its 3-sentences wikipedia text.
        textsdev[entity]=entity_text
#         print(entity_text)
#         print()
    except:
        pass
#         print(entity, "was not found on wikipedia")
#         print()



# print(textsdev)
```

```
/home/pleun/anaconda3/lib/python3.7/site-packages/wikipedia/wikipedia.py:
389: UserWarning: No parser was explicitly specified, so I'm using the be
st available HTML parser for this system ("lxml"). This usually isn't a p
roblem, but if you run this code on another system, or in a different vir
tual environment, it may use a different parser and behave differently.

The code that caused this warning is on line 389 of the file /home/pleun/
anaconda3/lib/python3.7/site-packages/wikipedia/wikipedia.py. To get rid
of this warning, pass the additional argument 'features="lxml"' to the Be
autifulSoup constructor.

  lis = BeautifulSoup(html).find_all('li')
```

In [21]:
```
manufacturers_regex={}
manufacturers_dep={}
# textsdev
# print(wiki_devices)
manu_predicates=['manufactured', 'produced', 'developed','designed',' de
veloped by','produced by','released by','made by','marketed by', 'manufa
ctured by']
for entity in wiki_devices:
    if entity in textsdev.keys():
#         print('entity is',entity)
#         print('manu_predicates',manu_predicates)
#          print(nlp(textsdev[entity]))
         manu_relations=extract_manufacturer(nlp(textsdev[entity]), manu
_predicates, entity)
#         print("relation  is",manu_relations)
#         print()
         manufacturers_dep.update(manu_relations)
#         manufacturers_dep[entity]=manu_relation[1]



for entity in wiki_devices:
    if entity in textsdev.keys():
#         print('entity is',entity)
#         print('manu_predicates',manu_predicates)
        manu_relations=extract_manufacturer_regex(nlp(textsdev[entity]),
manu_predicates, entity)
#         print("relation  is",manu_relations)
#         print()
         manufacturers_regex.update(manu_relations)
       #manufacturers_regex[entity]=manu_relation[1]


print(manufacturers_regex)
print(manufacturers_dep)
```

```
{'PDP-7': 'Digital Equipment Corporation', 'The original Walkman': 'Sony
', 'Sega TeraDrive': 'IBM', 'PlayStation 2': 'Nintendo', 'Cray-1': 'Cray
Research', 'Vectrex': 'General Computer Electronics', "Sinclair's ZX80":
'Sinclair Research', 'Volkswagen': 'Volkswagen Group', 'IPod Mini': 'Appl
e Inc.', 'Mac Mini': 'Apple Inc. It', 'Coleco Gemini': 'Coleco Industrie
s, Inc.', 'Zune 30': 'Microsoft', 'Motoblur': 'Motorola', 'Motorola A3100
': 'Motorola', 'Motorola Photon Q': 'Motorola', 'Game Boy': 'the Game & W
atch', 'Nokia 6210 Navigator': 'Nokia', 'Nokia 6710 Navigator': 'Nokia'}
{'PDP-7': 'Digital Equipment Corporation', 'The original Walkman': 'Sony
', 'Vectrex': 'Smith Engineering', 'Volkswagen': 'Volkswagen Group', 'Col
eco Gemini': 'Coleco Industries, Inc.', 'Zune 80, 120': 'Microsoft', 'Mot
orola Photon Q': 'Motorola'}
```

Exercise 3c Run the evaluation function `evaluate_property` to compute the performance for each of your four functions. Print the precision, recall, and F1-scores. (2 points)

In [22]:
```
# print(utils.evaluate_property(manufacturers_regex, gold_manufacturer
s))
print('Manufacturers_regex:')
precision, recall, f1=utils.evaluate_property(manufacturers_regex, gold_
manufacturers)
print("precision: %f, \nrecall: %f, \nF1-score: %f\n" % (precision, reca
ll, f1))

# print(utils.evaluate_property(manufacturers_dep, gold_manufacturers))
print('Manufacturers_dependency:')
precision, recall, f1=utils.evaluate_property(manufacturers_dep, gold_ma
nufacturers)
print("precision: %f, \nrecall: %f, \nF1-score: %f\n" % (precision, reca
ll, f1))

# print(utils.evaluate_property(birthyear_regex, gold_birthyears))
print('birthyear_regex:')
precision, recall, f1=utils.evaluate_property(birthyear_regex, gold_birt
hyears)
print("precision: %f, \nrecall: %f, \nF1-score: %f\n" % (precision, reca
ll, f1))

# print(utils.evaluate_property(birthyear_dep, gold_birthyears))
print('birthyear_dependency:')
precision, recall, f1=utils.evaluate_property(birthyear_dep, gold_birthy
ears)
print("precision: %f, \nrecall: %f, \nF1-score: %f\n" % (precision, reca
ll, f1))
```

```
Manufacturers_regex:
precision: 0.692308,
recall: 0.187500,
F1-score: 0.295082

Manufacturers_dependency:
precision: 0.600000,
recall: 0.062500,
F1-score: 0.113208

birthyear_regex:
precision: 1.000000,
recall: 0.400000,
F1-score: 0.571429

birthyear_dependency:
precision: 1.000000,
recall: 0.020000,
F1-score: 0.039216
```

4. Reflection (8 points)

For each entity, we will now compare the two methods to extract properties in terms of precision and recall.

Question 4a Comparing the precision between the methods based on regular expressions and on syntax dependencies:

- Which method yields lower precision?
- Why do you think this is the case?
- Give an example to support your argument.

(4 points)

Both the birthyear extractor functions have a precision of 1, the precision score is the amount of true positives in all the predicted positives, in this case it means that all the found properties are correct. Opposite are the manufacturer extraction functions, they both have a precision score of approximately 0.6, this means only 60% of the positives are correct. A possible explanation is that a device has multiple manufacturers or a manufacturers company was taken over by another company. For example the Vectrex: *'The Vectrex is a vector display-based home video game console developed by Smith Engineering. It was first released for North America in November 1982 and for both Europe and Japan in 1983. Originally manufactured by General Computer Electronics, it was licensed to Milton Bradley after they acquired the company.'* The gold label is: *{Vectrex: "Milton Bradley Company"}* and the functions output is: *{'Vectrex': 'General Computer Electronics'}*.

Question 4b Let's compare the recall for both properties.

- Which method yields lower recall?
- Why do you think this is the case?
- Give an example to support your argument.

(4 points)

The recall of the extractor using regular expressions is noticeably higher than the one using dependency information for both the birthyear and manufacturer relation. The recall shows how many of the actual positives are predicted by the system. That means that for the birthday extractor using regular expressions, 40% of all possible positives have been predicted by the system. There is no noticeable pattern between the recall for the birthday and manufacturer relations apart from the extraction methods.

The difference in recall between using regular expressions or dependency information for your extractor could be caused by the fact that the dependency extractor relies on the structure of the sentence to be able to identify a relation. The extractor used in this model looks for the entity to be a 'pobj', it's head should be an 'agent' and the head of that an 'acl', if this is the case the head of the head of the entity is returned as predicate. When the sentence has a different structure this will not be recognized, like for the device the Cray-1 which was found with regular expressions and not with dependency information. The sentence containing the relation is: "The Cray-1 was a supercomputer designed, manufactured and marketed by Cray Research". Here the entity, the Cray-1 is a 'nsubj' and its head is the root. Hence the relation could not be recognized here. This does then not count toward the number of positives predicted out of the possible positives and lowers the recall.

In [ ]: