COMP 474 UU,COMP 6741 UU 2214

Lab Session #11

Introduction

Welcome to Lab #11. This lab, we'll continue with NLP and learn how to build a text mining system with spaCy's matcher for rule-based named entity (NE) extraction. Then, we'll cover the basics of *neural networks* and start working with *word embeddings*, which are an important foundation for *deep learning* based NLP.

Follow-up Lab #10

Following are some sample scripts for last lab's questions:

Task #1: Here is a sample script for the spaCy exercise.

Task #2: Here's some example code for question classification using linguistic features extracted with spaCy NLP.

Task #1: Text Mining

The goal for this week is to build a simple *Text Mining (TM)* system. Generally, text mining means extracting interesting, structured information from unstructured (natural language) documents. The extracted information can then be exported in various formats for further analysis, e.g., using CSV, database export, or automated knowledge graph construction. One important text mining task is *Information Extraction (IE)*, used for applications such as business intelligence, scientific research, news/media monitoring, social media mining, and healthcare/medical record management.

NLP Pipelines

As mentioned in the lecture, spaCy's models come with a default pipeline that you can then modify and extend. The first step is to go through the spaCy Pipelines documentation to understand the components run in a default pipeline. You can view the default pipeline with the following code snippet:

```
print(nlp.pipe_names)
```

Not let's try adding a simple custom component to the pipeline to count the number of title cased tokens within a text and print them out in the console:

```
import spacy
from spacy.language import Language


@Language.component("custom_title_counter")
def custom_title_counter(doc):
    title_count = sum([1 for token in doc if token.is_title])
    print(f"This document has {title_count} title cased tokens!!!")
    return doc
```

Once we have our pipeline module to perform the desired function, add it to your `nlp` pipeline as the last module.

Print the pipeline modules in your console to make sure your custom module is added to the nlp pipeline. Process the following paragraph with your new pipeline and see the output of your custom module.

```
text = """Concordia University (French: Université Concordia; commonly referred to as Concordia) is a public comprehensive research university located in Montreal, Quebec, Canada. Founded in 1974 following the merger of Loyola College and Sir George Williams University, Concordia is one of the three universities in Quebec where English is the primary language of instruction (the others being McGill University and Bishops University). As of the 2018–19 academic year, there were 46,829 students enrolled in credit courses at Concordia, making the university among the largest in Canada by enrollment. The university has two campuses, set approximately 7 kilometres (4 miles) apart: Sir George Williams Campus is the main campus, located in Downtown Montreal in an area known as Quartier Concordia; and Loyola Campus in the residential district of Notre-Dame-de-Grâce. """
```

Information Extraction

The next step is to add an information extraction component to the pipeline. In particular, we'll start by extracting named entities (NEs), just like the entities you've extracted using the default model above. As always, the two general options for NE extraction are using either (1) machine learning or (2) a rule-based approach. While machine learning can generally provide more accurate results, rule-based systems can still be very useful in first experiments, developing demos/prototyping, and bootstrapping the manual annotations (gold standard) required for machine learning. Some entities can also be easily and accurately identified with rules (e.g., URLs, IP addresses, phone numbers) so it wouldn't make sense to apply the ML overhead.

For our first experiments here, we'll develop a rule-based IE system. As discussed in the lecture, this does not mean just using regular expressions for matching the literal text (so-called surface form). Rather, you want to combine features as extracted by your components above to obtain more general rules, e.g., by incorporating lemmas and POS tags.

To start, go through the spaCy documentation for rule-based matching. Your goal today is to develop rules for identifying organizational units in a university, like faculties or departments, as you would find them on websites or scientific research publications. Rather than hard-coding string matching rules for all possible combinations like *"Department of Mathematics and Computer Science"* or *"Faculty for Extraterrestrial Space Travel"*, you should develop rules that combine unit names ("Faculty", "Department", "School", ...) with POS-tags, in particular nouns (check the POS tags documentation to find the right tag names).

Use the text given above: try to come up with some rules and test them using the interactive Rule-based Matcher Explorer shown in the lecture.

Now let's try doing it with Python code:

```
import spacy
from  spacy.matcher import Matcher
nlp = spacy.load("en_core_web_sm")
matcher = Matcher(nlp.vocab)
```

Copy the pattern you used from the Rule-based Matcher Explorer as your pattern within the code.

```
pattern = #Pattern from Rule-based Matcher Explorer
```

Now add the pattern with a label to the "matcher" object created above.

```
matcher.add("uni_identifier", [pattern])
```

You can then process the text given above with your `nlp` pipeline. Now let's try to extract the matches matched by your pattern and print them to the console:

```
matches = matcher(doc)


for match_id, start, end in matches:
    string_id = nlp.vocab.strings[match_id]  # Get string representation
    span = doc[start:end]  # The matched span
    print(match_id, string_id, start, end, span.text)
```

Compare the results you obtained from the Python code with the Explorer. Try experimenting with multiple patterns to extract entities.

Task #2: Perceptron

You can find a Perceptron implementation in *scikit-learn*, which is used like any other classifier:

```
import numpy as np
from sklearn.linear_model import Perceptron
```

Create the dataset for the AND function as mentioned in the worksheet:

```
dataset = np.array([[1,1,1],
                    [1,0,0],
                    [0,1,0],
                    [0,0,0],])
```

For our feature vectors, we need the first two columns:

```
X = dataset[:, 0:2]
```

and for the training labels, we use the last column from the dataset:

```
y = dataset[:, 2]
```

(a) Now, create a Perceptron classifier and train it with the AND dataset.

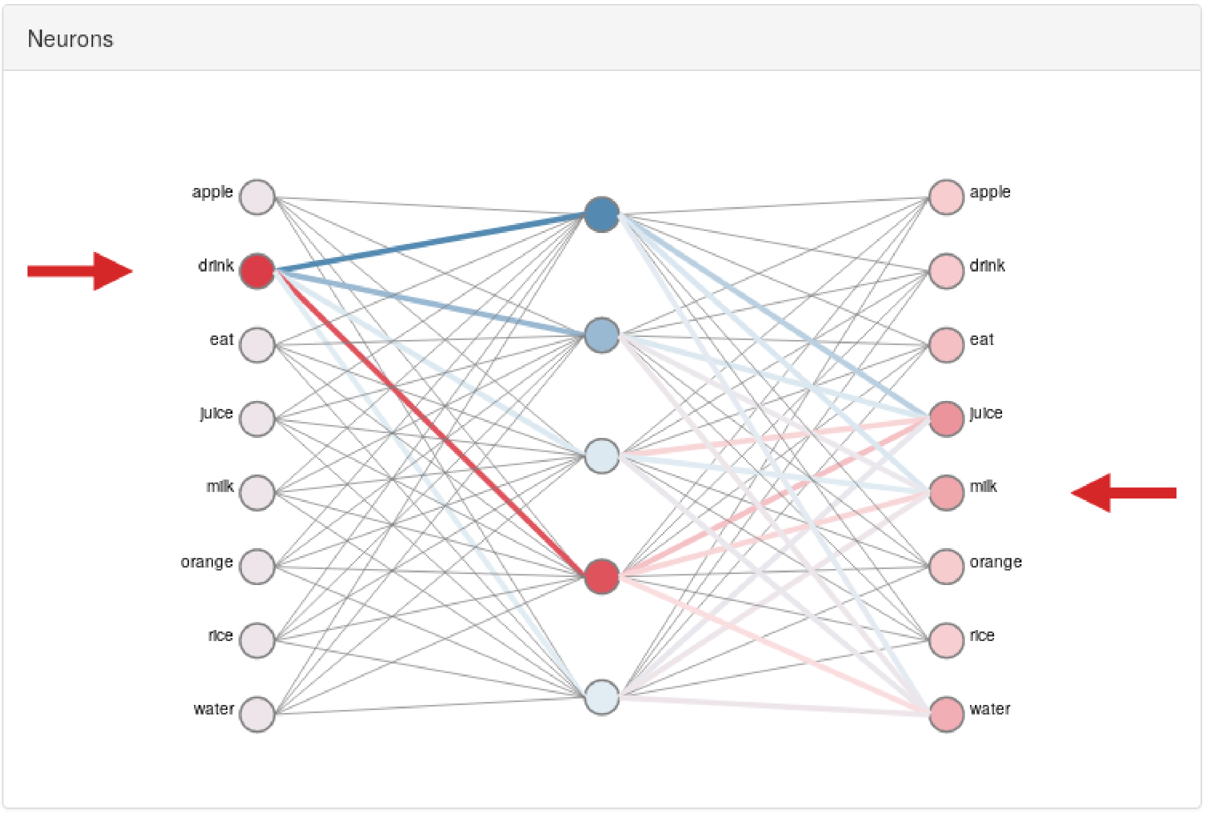
(b) Apply the trained model to all training samples and print out the prediction. Did it learn all values correctly?

Observe how the weights changed before & after the learning (find the weights associated with the feature and print them out). See what happens when you try to learn the XOR function.

Task #3: Word Embeddings

Note: to be able to work with the existing spaCy models, you need a computer with sufficient RAM (at least 8GB for the "medium" and 12GB for the "large" models).

Make sure you understand the idea of word embeddings as discussed in the lecture. Here is a neat online tool (works best in Chrome) to visualize how Word2vec-type vectors are created using a neural network:



The spaCy library has built-in support for word vectors. Assuming you installed spaCy for the previous lab, you will still need to download a larger language model, since the "small" English model we used in the previous lab does not include word vectors. You can download and install the medium-sized English model with:

```
python -m spacy download en_core_web_md
```

Afterwards, you should be able to see the similarity of words based on their word embeddings, using spaCy's built-in cosine similarity:

```
import spacy

nlp = spacy.load("en_core_web_md")  # make sure to use larger model!
tokens = nlp("dog cat banana")

for token1 in tokens:
    for token2 in tokens:
        print(token1.text, token2.text, token1.similarity(token2))
```

For details on word vectors in spaCy, make sure you read the documentation.

A simple way to obtain a vector for a whole sentence (or even a document) is to simply *average* all the individual word vectors it contains. This is the result you get when you call, e.g.:

```
nlp('why are we still wearing masks').vector
```

(you can compute the average yourself using a few words to verify). Note that, at this point, the vector is not an accurate semantic representation of the sentence anymore (for example, if it contains words like "good" and "bad", they will mostly cancel each other out). Nevertheless, you can use these vectors just like we previously used count or tf-idf vectors, for example, in a QA chatbot to find answer sentences by similarity to a question sentence, to make recommendations, or cluster documents (e.g., using k-means). A more sophisticated method to build document vectors is the Doc2vec method briefly mentioned in the lecture.

More on embeddings

- If you need to build a custom model with your own word embeddings and use them in spaCy, one option is to use the Gensim library
- The spaCy folks also came up with a fun extension to Word2vec called sense2vec, which you can also try in an online demo
- If you want to do word vector math for computing analogies like shown in the lecture, this is easier to do with Gensim's `most_similar` vector function (see an example here)
- And for training and using neural networks using Python, look at Keras and TensorFlow (which is also used inside our Rasa chatbot library)

That's all for this lab!

Last modified: Tuesday, 5 April 2022, 4:36 PM

◄ Worksheet #10

Jump to...