COMP 474 UU,COMP 6741 UU 2214

Lab Session #10

Introduction

Welcome to Lab #10. This lab, we'll cover the foundations for developing *Text Mining* systems, in particular how to build NLP pipelines using the spaCy library for Python. Then, we'll introduce the Rasa framework for building chatbots, which we'll also use in part 2 of the course project.

Follow-up Lab #9

Solution Task #2 (Search based chatbots)

Here's a sample program for this task from last week that finds similar questions on the Amazon QnA dataset.

Task #1: Natural Language Processing with spaCy

Preprocessing text is a very crucial step for any NLP related tasks. There are many frameworks (GATE, UIMA) and libraries (NLTK, CoreNLP, spaCy) available out there to facilitate with this regard. Today, we will be looking into spaCy to get you started with the basic steps.

Installing Spacy:

pip: Before installing spaCy lets first make sure your pip and setuptools are upto date and install them:

```
pip install -U pip setuptools wheel
pip install -U spacy
```

Conda:

```
conda install -c conda-forge spacy
```

Downloading language Models:

Not all languages are preprocessed in the same way. Depending on the language, steps like tokenization, sentence splitting and POS tagging varies. To facilitate different languages and different genres, spaCy provides different pre-trained language pipelines to work with. For starters, we will be working with the basic English model. We can download the model with the following command:

```
python -m spacy download en_core_web_sm
```

Use the following code to import the spaCy library and load the language model:

```
import spacy
nlp = spacy.load("en_core_web_sm")
```

Now let's try to preprocess the following text and check the POS tags assigned by spaCy:

```
doc = nlp("I prefer a direct flight to Chicago.")
```

Depending on the language model and sentence provided as input, spaCy outputs an object "doc", in our case with a variety of annotations encoded within. Linguistic annotations are available as token attributes. Let's try to print some basic attributes:

```
for token in doc:
    print(token.text, token.lemma_, token.tag_, token.dep_)
```

Now, try to process the following paragraph to see how spaCy performs sentence splitting.

> "Superman was born on the planet Krypton and was given the name Kal-El at birth. As a baby, his parents sent him to Earth in a small spaceship moments before Krypton was destroyed in a natural cataclysm. His ship landed in the American countryside, near the fictional town of Smallville. He was found and adopted by farmers Jonathan and Martha Kent, who named him Clark Kent. Clark developed various superhuman abilities, such as incredible strength and impervious skin. His adoptive parents advised him to use his abilities for the benefit of humanity, and he decided to fight crime as a vigilante. To protect his privacy, he changes into a colorful costume and uses the alias "Superman" when fighting crime. Clark Kent resides in the fictional American city of Metropolis, where he works as a journalist for the Daily Planet. Superman's supporting characters include his love interest and fellow journalist Lois Lane, Daily Planet photographer Jimmy Olsen and editor-in-chief Perry White. His classic foe is Lex Luthor, who is either a mad scientist or a ruthless businessman, depending on the story."

Dependency Parse Trees

In addition to the constituent parse trees shown in the lecture, dependency parsing is another formalism in the family of English grammar. In dependency parse trees, two tokens are connected by a single arc, where the label of the arc is the dependency relation, the starting node of the arc is known as the **Governor** (**head** in spacy) and the token at which the arc points at is knows as the **Dependant** (**child** in spacy). A **Dependant/Child** can only have one **Governor/Head** but the vice versa is not true. Now try to use spacy's "displaCy visualizer" to visualize the dependency parse tree for the sentence given above ("I prefer a direct flight to Chicago.").

Named Entity Recognition

First, let's try to use spaCy for Named Entity Recognition (NER). NER is the task of identifying key information/entities in text, such as persons, locations, or organizations. Here, an entity can be a single token or a series of tokens. Every detected entity is then classified into predetermined categories (e.g., `Person, Organization, Date`). NER is an important step for many NLP tasks, when you want to extract the main subjects from excerpts of text or to get an overall idea of what a text is talking about.

In spaCy, named entities are available as the *ents* property of a *doc* object. Try the following code to extract NEs and some of their properties:

```
import spacy
nlp = spacy.load("en_core_web_sm")
doc = nlp("Apple is looking at buying U.K. startup for $1 billion")
for ent in doc.ents:
    print(ent.text, ent.start_char, ent.end_char, ent.label_)
```

Again, use displaCy to visualize the entities identified. Try restricting your visualizer to only display specific categories of named entities for the following sentence. First try to visualize all the named entities to make sure the ones you want to see are actually present, then restrict the displayed entities:

```
text = "European authorities fined Google a record $5.1 billion on Wednesday for abusing its power in the mobile phone
market and ordered the company to alter its practices"
```

Task #2: Question Classification

We've so far looked into classification tasks using tf-idf and cosine similarity to identify the classes the query vector falls into. Since you've now learned about using spaCy to extract linguistic features, let's try to utilize them to create more sophisticated *feature vectors* for our data and then perform classification using a decision tree classifier from the scikit-learn library.

Let's first import the necessary libraries and initialize some variables required for this task:

```
import numpy as np
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn.preprocessing import LabelEncoder
from collections import Counter
import spacy

nlp = spacy.load("en_core_web_sm")
```

Now, let's create a function that accepts a text as an input and return a list resembling a feature vector. A feature vector can contain any combination of features of your desire. For this task following are the features we are interested to experiment with:

1. Lemma of the ROOT text
2. Total number of occurance of the POS tag "WP -
3. Total number of occurance of the POS tag "WRB"
4. Total number of PERSON entity tags.
5. Total number of GPE entity tags.
6. Total number of ORG entity tags.
7. Length of the text.

Use the following snippet and fill in the code to extract the features listed above:

```
def create_features(text):
    doc = nlp(text)
    pos_list = [] # Create a list with all the POS tags in the text
    pos_count_dict = Counter(pos_list) # Count the number of POS tags withing the POS list
    entity_list = [] # Create a list with all the entities withing the text
    ent_count_dict = Counter(entity_list) #Count the number of entity types withing the entity list
    root_lemma =  # Lemma of the root token in the text
    sentence_length = # Length of the text (number of characters within the text)

    return [root_lemma, pos_count_dict['WP'], pos_count_dict["WRB"], ent_count_dict["PERSON"], ent_count_dict["GPE"],
ent_count_dict['ORG'], sentence_length]
```

Let's define our corpus:

```
corpus = [
    'Who is Bill Gates',
    'Where is Concordia located',
    'What is AI',
    'What city is McGill located in',
    'Who is McGill'
]
```

Iterate through each document within the corpus and create a feature matrix:

```
features = []
for text in corpus:
    features.append(create_features(text))

feature_mtrx = np.array(features) # Convert the feature matrix to a numpy array.
```

The first column in our data contains a string value, which is not a suitable datastructure for the DecisionTreeClassifier we are intending to use. Let's assign integer labels to these string values using sklearn's LabelEncoder:

```
le = LabelEncoder()
feature_mtrx[:, 0] = le.fit_transform(feature_mtrx[:, 0])
```

Create a label array containing the class of each document within the corpus:

```
# 0=Who, 1= Where, 2=What
y = np.array([0, 1, 2, 2, 0])
```

Train a DecisionTreeClassifier using our feature matrix created above:

```
clf = DecisionTreeClassifier(random_state=0)
clf.fit(feature_mtrx, y)
```

Now test your system by trying to predict the class for the following query text:

```
q = "Where is Canada"
```

You can also visualize the features your decision tree considered for the prediction task (note you'll need to have Matplotlib installed):

```
import matplotlib.pyplot as plt

plt.figure()
plot_tree(clf, filled=True, feature_names=["root", "wp_count", "wrb_count", "person", "gre", "org", "length"])
plt.show()
```

Task #3: Building your own chatbot with Rasa

So far, you've learned the building blocks required for the backend of a chatbot: Knowledge Graphs, SPARQL queries, classification algorithms, and a natural language procession library.

Now it's time to work on the front end part of the chatbot.

If you are to develop a fully integrated chatbot from scratch, you'd have to take care of the following basic steps:

1. Obtain user input, either through a console or some interface to an existing tool (e.g., WhatsApp, LINE, Slack, Discord, ...).
2. Process the user input with an NLP library and extract important components, e.g., *Subject* of the user input, *Entities* if there are any.
3. Create feature vectors of the user input and run them through a pre-trained classification model to obtain the category of the user input (e.g., question, action, confirmation).
4. Based on the category of question, construct appropriate SPARQL queries by parsing the necessary information extracted at Step 2.
5. Obtain results by querying the knowledge graph
6. Reconstruct a natural language answer based on the response obtained from the query and present it to the user.

But to make things easier, there are many tools available out there with different capabilities and functionalities to cater different requirements. SpaCy's ChatterBot, RASA, ChatOps are some examples of different libraries and frameworks available for making your life easy when creating your own chatbots.

In this lab session, we will be looking into how to use Rasa to create our chatbot. There are many interesting articles available online; to learn more about Rasa check out: Article 1, article 2.

Essentially, RASA is an open source machine learning framework for building chatbots. It has two main modules:

1. **Rasa NLU:** for understanding user messages, in the form of *Intent* and *Entity*
2. **Rasa Core:** Maintains the conversation flow by trying to predict dialogues as a reply based on the user message

Let's now get our hands dirty with some technical work.

Installing Rasa

You can install Rasa Open Source using pip (requires Python 3.6, 3.7 or 3.8). This will by default install Tensorflow as well. So it's better to first create a virtual environment specific for RASA:

Conda:

```
conda install python=3.6
conda create -n rasa python=3.6
source activate rasa
pip install rasa
```

Python VirtualEnv:

```
virtualenv -p /usr/bin/python3.6 rasa
source ./rasa/bin/activate
pip install rasa
```

After you finished installing, try running:

```
rasa init
```

If the installation succeeded without any error, then you would be prompted to enter a path to create a new project. If you have a specific path you would prefer to have your project copy the path and paste in the console. Else pressing enter would create a new project in the current path.

Follow the interactive session, continue pressing enter until you reach a point where it prompts to try out the chatbot in the console. Press enter and type "Hello" to see what the bot says back to you.

Rasa Overview

After you finished playing with the chatbot a little, let's get a little serious and look into the files Rasa created for us. Navigate in your file explorer to the folder containing all the files (path provided during the rasa init step.) Following are some descriptions of the important files we will be using today:

- data/nlu.yml: Contains your NLU training data. Here you define your Intent and provide all statements or question that is related to that intent.
- data/stories.yml: Contains all your stories. This is required by your Rasa Core. *Story* is a training data format for the dialogue model, consisting of a conversation between a user and a bot. The user's messages are represented as annotated intents and entities, and the bot’s responses are represented as a sequence of actions.

- domain.yml: This is the assistant's domain. Defines all the inputs and outputs of the assistant. Includes a list of all the intents, entities, actions, slots and actions.
- actions/actions.py: Code for your custom actions. This can be used to call exeternal server via REST API or API call.

Create custom action

We are now going to create a custom function to print a message about a person. Following are list of steps to follow to achieve this. But to get a quick idea of what you have to do you can check this.

Step 1: Create a new Intent "about_person" in *nlu.yml* file. Add training data to trigger this intent. (Make sure the syntax is similar to other intents that are already in the file.)

```
- intent: about_person
  examples: |
    - Who is he?
    - Who is [Joe](person) ?
    - Who is [Kate](person)?
    - Who is [Harry](person)
    - Tell me about him
    - Tell me about
    - Tell me about [Peter](person)
    - Tell me about [Joe](person)
    - Can you tell me about him?
    - Can you tell me about [Jane](person)?
    - Can you tell me about [Peter](person) ?
    - Can you tell me about [Jonathan](person) ?
    - Do you know [Alice](person) ?
    - Tell me about [Jack](person) ?
```

in the above given example Tokens within the square brackets ([*Joe][Peter]...*) are examples of what instance we are interested in extracting and variables within round brackets "*(person)*" are known as entities, which functions similar to a variable to contain the value the use inputs. (i.e., person = "Joe").

Step 2: Once we add a new Intent in nlu.yml we should let the domain.yml file know about it. Add the name of the intent "about_person" in the *domain.yml* file below the other already included intents.

Step 3: We'll have to register the entities we created as well. In the same *domain.yml* file add the following lines to register the "person" entity we just created.

```
entities:
  - person
```

We'll also have to define the type of the entity and provide default value as well. This goes inside "slots" keyword as follows in the same *domain.yml* file.

```
slots:
  person:
    type: text
    initial_value: "initial"
    mappings:
      - type: from_entity
        entity: person
```

Step 4: Now let's create a custom action to give a custom feed back about the person user inputs. Navigate to *actions.py* file, and uncomment the defualt class already given to you. Two functions are critical for a custom action.

1. name function: returns the name of the action. Replace the name returned by the function as:

```
"action_person_info"
```

2. run function: performs a custom task and prints the response using the "dispatcher.utter_message(text="")". Add the following line just above the *return* keyword.

```
dispatcher.utter_message(text=f"If you are asking about {tracker.slots['person']}, Best Human Ever!!! ;-) ")
```

All action classes with have the entity values within the tracker.slots variable in a dictionary format.

Step 5: We'll again have to register the action in the *domain.yml* file. If you don't see "actions" keyword in the file add a new one, same as "intent" and add the action name returned by your custom actions class.

```
actions:
  - action_person_info
```

Step 6: Now let's create a story with the intent and action we created. Navigate to *data/stories.yml* file. Add a new story named get person info and add the lines of your story script (just the way you would write a conversation in a drama script. :D)

```
- story: get person info
  steps:
    - intent: greet
    - action: utter_greet
    - intent: about_person
    - action: action_person_info
```

Step 7: In order to access our custom actions we'll have to enable action endpoint to run on another port. Navigate to *endpoints.yml* file and uncomment the following lines.

```
action_endpoint:
url: "http://localhost:5055/webhook"
```

It's time to "break a leg"

Open 2 consoles, and navigate to the main folder where you have all the rasa project files. In one console run the following command to train our model with the new data provided:

```
rasa train
```

Once it finishes training run the following command:

```
rasa shell
```

At the same start the Rasa action server with the following command. (Use the other console to run this.)

```
rasa run actions
```

Now try greeting your chat and ask the following question.

```
Who is Joe?
```

You should be getting a reply back saying...

```
"If you are asking about Joe, Best Human Ever !!!"
```

If you get this to work, then now it's time for you to play with rasa a little to explore more functionalities and features which you think that will be suitable for your requirement. Below are a few hints for you to explore a little more.

1. When trying to extract entities, if the use enters a word which is not in the trained model's vocabulary it would not identify the word inside the custom actions, instead it would use the default value entered in the slots, in domain.yml file. So to make rasa accept OutOfVocabulary (OOV) words you'll have to perform some additional actions. Read about it here: https://rasa.com/docs/rasa/components/#intent-featurizer-count-vectors.
2. You can use the custom actions functions to make a call to your Fuseki server with the requests library available in Python. Of course, your Fuseki server has to be up and running as well.

That's all for this lab!

Last modified: Tuesday, 5 April 2022, 3:36 PM

◄ Worksheet #09

Jump to...

Lecture Slides #11 ►