COMP 474 UU,COMP 6741 UU 2204

Lab Session #11

Introduction

Welcome to Lab #11. This lab, we'll cover the foundations for neural networks and working with word embeddings, as well as the *Spacy* chatbot library.

Follow-up Lab #10

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

Task #1: part 1

Task #2: part 1, part 2

Task #1: 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.

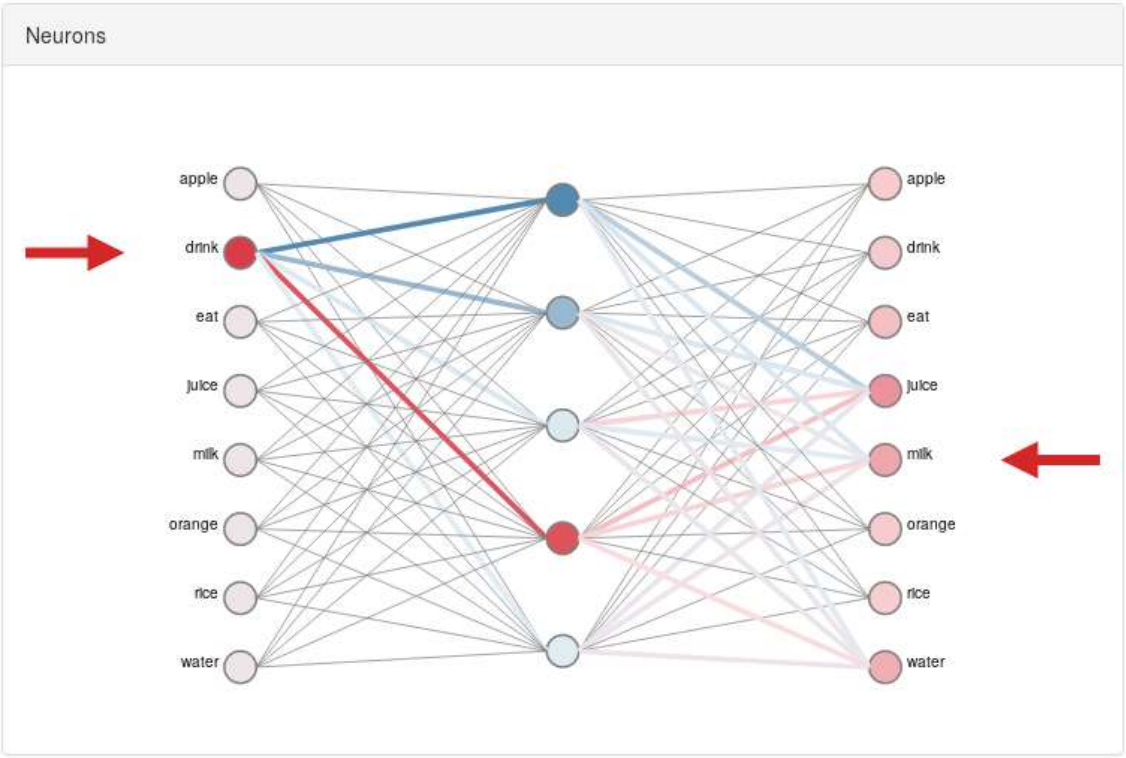
(b) Apply the trained model to all training samples and print out the prediction. Did it learn all values correctly?

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 #2: 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 stuck inside').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

Task #3: Building your own chatbot with Rasa

So far you've learnt 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, following are the basic steps you would probably follow:

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 a language processing 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 pretrained classification model to obtain the category of the user intput.
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 KG
6. Reconstruct an answer based on the response obtianed from the KG 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 to create 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 if you are interested refer: 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.

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.

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.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 knows 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: any
    initial_value: "initial"
```

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, 6 April 2021, 11:30 AM

◄ Lecture Slides #12

Jump to...

Lab Session #12 ►

You are logged in as Yaohua Zhang (Log out)
COMP-474-2204-UU

Academic Integrity
Academic Assessment Tool
English (en)
Deutsch (de)
English (en)
Español - Internacional (es)
Français (Canada) (fr_ca)
Français (fr)
Italiano (it)
العربية (ar)