COMP 474 UU,COMP 6741 UU 2204

Lab Session #9

Introduction

Welcome to Lab #9. This week, we'll start developing two types of chatbots: pattern-based (using AIML) and search-based (using cosine similarity on tf.idf vectors).

Follow-up Lab #8

Solution Task #1 (KNN Regression)

Here's a sample program for kNN Regression exercise from lecture Worksheet #6.

Solution Task #2 (KNN Classification)

Here's a solution for kNN Classification on the *sklearn* wine dataset

Task #1: Building your first Chatbot

The oldest but perhaps still most widely used technique for building a chatbot is using question-answer patterns. Basically, regular expressions are matched against the user's input and rewrite/output patterns generate a response (see the lecture slides and material for more details). As discussed, this technique has its limitations, but this type of bot can be easily combined with the other methods discussed in class.

We'll start with the technology mentioned in the lecture, using the Artificial Intelligence Markup Language (AIML). This has the advantage of being an open standard, but with the downside that active development has largely moved to other platforms (see below for notes on other bot frameworks).

To get started, you need one of the AIML-compatible libraries, e.g., AIML-Bot. Install the library and create your first bot:

```
import aiml_bot
bot = aiml_bot.Bot(learn="mybot.aiml")
while True:
    print(bot.respond(input("> ")))
```

You'll need an AIML file (which is in XML format) to define the question/answer patterns. Here is a first one to test:

```
<?xml version="1.0" encoding="UTF-8"?>
<aiml version="1.0.1">
  <category>
    <pattern>HELLO *</pattern>
    <template>Hi Human!</template>
</category>
<category>
  <pattern>HELLO TROLL</pattern>
  <template>Good one, human.</template>
</category>
</aiml>
```

Experiment with generating `<random>` responses as shown in the lecture.

To avoid duplicating patterns for the same kind of interaction, you can use the `<srai>` tag:

```
<?xml version = "1.0" encoding = "UTF-8"?>
<aiml version="1.0.1">
   <category>
      <pattern>WHAT IS COMP 474</pattern>
      <template>COMP 474 is the Intelligent Systems course.</template>
   </category>

   <category>
      <pattern>WHAT IS COMP 6721</pattern>
      <template>COMP 6721 is the Applied Artificial Intelligence course.</template>
   </category>

   <category>
      <pattern>DO YOU KNOW WHAT * IS</pattern>
      <template>
         <srai>WHAT IS <star/></srai>
      </template>
   </category>
</aiml>
```

As you can see, questions matching the pattern `DO YOU KNOW WHAT * IS` are now redirected to the patterns for `WHAT IS *`.

You can find various AIML files online; for example, there is a std-65% AIML file that covers "65% of all standard questions" that a bot could be asked. Try some of them out, but remember that the Python AIML implementations only support AIML v1, so AIML v2 files will not work.

Where to go from here

As mentioned in the lecture, bot development languages & frameworks have become very fragmented, with numerous vendors, all using their own non-standard extension or proprietary bot language. They all offer convenient cloud development and deployment frameworks, but using them locks your bot to their platform (e.g., you cannot develop a bot with Amazon Lex and move it to Google's Dialogflow):

- Pandorabots are based on AIML2.0, but using their proprietary cloud framework
- Amazon's Lex started out with a modified, JSON-based version of AIML that has evolved into its own language
- Google's Dialogflow has its own bot language, originally developed by api.ai
- Microsoft has its own Bot Framework and specialized offers like QnA Maker
- IBM has of course Watson Assistant

Most of these make it easy to connect to other existing services running on the same platform and mix multiple bot techniques (patterns, retrieval, grounding, generation). They also offer convenient cloud deployment to multiple platforms (e.g., Slack, Skype, Twitter, Facebook Messenger).

Some open source frameworks that are under more active development than the AIML-based ones are Hubot, Rasa, Chatterbot, Errbot, and Will.

Task #2: Search-based chatbots

In this task, we'll use another technique discussed in the lecture: Search-based bots, which look for an answer that is similar to an existing question in a corpus.

To compute the similarity (user question vs. corpus question or user question vs. corpus answer), we'll use the same techniques that you already developed: Vectorization using tf.idf and similarity computation using cosine. So, your task is to rite a program that:

- takes a user's question as input and converts it into a tf.idf vector
- finds the closed matching question in the dataset by using cosine similarity
- and prints out the existing answer for that question from the dataset.

For our experiments, we'll use the Amazon Q&A dataset that you can download at https://jmcauley.ucsd.edu/data/amazon/qa/

To keep things simple, start with a single category from the "Per-category files" list. Download one of the zip file from the link given above and load it into your program using the function below:

```
import gzip
def read_file(path):
    g = gzip.open(path, 'r')
    for l in g:
        yield eval(l)
dataset = read_file('qa_Appliances.json.gz')
```

From the dataset we need to read only the questions i.e., we need to extract the data that have "question" as the key value.

```
question_list = []
for i in dataset:
    question_list.append(i['question'])
    answer_list.append(i['answer'])
```

Now convert the list to array using Numpy's `asarray()` function:

```
import numpy as np
question_dataset = np.asarray(question_list)
```

Then, just like in previous exercises, use a TfidfVectorizer() to convert the natural language data to vector.

Now you can ask the user to input a question:

```
input_question = input("What is your question: ")
```

Read the user's input and find similar questions that are present in the dataset. For finding the similarity use cosine similarity. Make sure you also vectorise the user's input.

For example, the highest cosine similarity question for the question "Is the blender powerful?" (in the appliances data set) should be:

```
Similar question to user's question: Is the Genuine OEM FSP Whirlpool Kitchen Aid Blender Rubber Seal part number 9704204
suitable for Kitchenaid blender KSB560CU1?
```

```
Answer given to the similar question: Not sure, but it did work fine for me. FYI - It is inexpensive enough to order it and
try.
If it does not fit, send it back. As an alternative you should be able to contact the company to get a better answer to your
question.
```

You can also try printing out the top 5 highest similar questions for the user's question.

Task #3: Classification with k-Nearest-Neighbor (kNN)

For more sophisticated question-answering systems, a first step is to classify the question by type to make sure the generated answer makes sense (there are a lot of possible questions, but most fall into a few similar patterns). The approach shown in the lecture classifies question by the expected answer type, like *Person*, *Location*, *Date*, or *Definition*. Giving enough labeled examples, we can train a machine learning algorithm to classify a new (unseen) question in one of the predefined types.

For our first experiments, we will use the kNN algorithm as discussed in the lecture. Luckily, it's already implemented for us in Scitkit-learn.

```
import numpy as np
from sklearn.neighbors import KNeighborsClassifier
```

To see how this works, let's start with some simple data:

```
corpus = np.array([
        'Who is Bill Gates',
        'Where is Concordia located',
        'What is AI',
        'What city is McGill located in',
        'Who is McGill'
])
```

As always, we need to convert our text input into a feature vector we can use for machine learning. To keep it simple for now, we'll use the representation in form of tf-idf vectors as before:

```
from sklearn.feature_extraction.text import TfidfVectorizer
vectorizer = TfidfVectorizer()
X = vectorizer.fit_transform(corpus)
```

However, for supervised learning we also need the labels for (ground truth) for each question. We'll encode this in a vector as follows:

```
# Encode labels: 0="Person", 1="Location", 2="Definition"
y = np.array([0,1,2,1,0])
```

Now you can "train" a classifier (for kNN, this simply stores the vectors with their labels):

```
clf = KNeighborsClassifier(3)
clf.fit(X, y)
```

Here, "3" is *k*, the number of neighbors voting when classifying unseen data (see the documentation). Note that this is a standard pattern when creating a ML model with Scikit-learn, you can use other algorithms (e.g., Naive Bayes, SVM) in the same way (here is a nice cheat sheet for working with Scitkit-learnfrom Datacamp).

Once you have a trained classifier (again, other ML algorithms involve building a model, whereas kNN falls into the "lazy learner" category), you can use it to classify unseen data:

```
q = 'What city is Concordia located in'
q_vec = vectorizer.transform([q])
predict = clf.predict(q_vec)
print('Predicted class = ', predict)
```

This question is close enough to find the right answer (1 = "Location"), but you'll see for most questions you'll get a wrong results: We simply do not have enough training data and additionally use a representation (tf-idf vectors) that have a high dimensionality.

From here, you have two ways to improve:

- Use different features for the machine learning algorithm, like we did on the worksheet. A more realistic solution could NLP techniques like, POS-tagging, to pre-process extract the features (we'll cover NLP libraries later).
- Get more training data; for example, here is a dataset you could use with questions and their types, which looks like this:

  ```
  NUM:count How many people in the world speak French ?
  LOC:other Where is the Orinoco ?
  DESC:def What is a Canada two-penny black ?
  ```

Task #4: Natural Language processing with spaCy

Preprocessing text is a very crucial step for any NLP related tasks. There are many tools (GATE, CoreNLP) and libraries (NLTK4, spaCy5) 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 language are preprocessed the same way. Depending on the language, tokenization, sentence splitting and POS tagging etc... varies. To facilitate different languages and different genres spaCy facilitates in providing different 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 to import 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 tag assigned by spaCy:

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

Depending on the language model and sentence provided as an 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 constituent parse trees learnt in the lecture, dependency parsing is another formalism in the family of English grammar. In dependency parse trees 2 tokens are connected by a single arc, where the label of the are is the dependency relation, and starting 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 wise 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 a task of identifying key information/entities in text. Where entities can either be a token or series of tokens. Every detected entity is then classified into predetermined category (*e.g. Person, Organization, Date etc...*). NER is an important step for some NLP tasks when you want extract main subjects from excerpts of text or to get an overall idea of what the text is talking about.

In spaCy Named entities are available as *ents* property of a *doc* object. Try the following code to extract Named entities and some of it's 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_)
```

Try to use displaCy visualizer to visualize the entities identified.

Try restricting your visualizer to only display specific categories of named entities through visualizer for the following sentence. (First try to visualize all the named entities to make sure you are only visualising the ones you wanted to see)

```
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"
```

That's all for this lab!

Last modified: Thursday, 8 April 2021, 4:27 PM

◄ Worksheet #09

Jump to...

Lecture Slides #11 ►