Natural Language Processing in Python

April 15, 2020

1 Part 1 - Introdcution and Assumptions

Welcome to the tutorial "Natural Language Processing in Python" which is part of the course "Computational Argumentation" at the Paderborn University in the summer term 2020.

The goal of this quick tutorial is to give you a short introduction into general text mining practices and how to apply them in the programming language Python. While you probably know most of the techniques used here from other courses, it might still be beneficial to read this quick recap to get a refresh.

In summary, this tutorial will cover the loading of a corpus in Python and a first exploration of the available data; the transformation of textual data into numerical features that can be used to train a machine learning classifier; training such a classifier and using it to make predictions for textual data.

The main task in this tutorial will be to predict the political ideology of users in the debate.org corpus (more details below) based on their presented arguments. For this, we will first import and explore the data and extract the needed information into a data structure that is easier to use, only containing the textual data and the necessary labels. Afterwards, we will apply some NLP tools in order to preprocess the data and extract numerical features from the text on which we will then utilize to train a machine learning classifier. Lastly, we will generate some predictions and evaluate the classifier using the F1 metric and a cross-fold validation.

I'm going to assume that you have installed **Python version 3.6.10**, which is the version we will be testing all assignments in this course with. Also, you should already have some knowledge about the Python programming language itself and how to write and run programs/scripts with it (see this link for a list of introductions). Additionally, you should be familiar with using the Python package manager `pip` and (to a certain extent) the following libraries: `pandas`, `scikit-learn` and `spacy` (please also install one of the English language models). The exact versions of all packages will be uploaded to the PANDA system as `requirements.txt`. Lastly, this tutorial is done in a Jupyter notebook. If you also want to use it, you can find instruction on how to set it up here. The code of this tutorial will be published as a notebook file, so it might be good to install it. In addition, I expect you to have downloaded the debate.org corpus we are going to use. You can find it here.

In the next part we will have a look at how to load the data and get some first insights into it. If you have any questions, please feel free to send us an E-Mail or attend one of the live-chat session in PANDA. Please refer to the organizational slides of the first lecture by Professor Wachsmuth for the exact time frames.

2 Part 2 - Data loading and exploration

Welcome to the second part of the tutorial "Natural Language Processing in Python". In this part we will see how to load the dataset in Python and do a first exploration of the available data. We will then save it in a nicer data structure for usage in later parts.

After downloading dataset, you should have at least two files: the `debate.json` file and the `users.json` file. For me, the two files are in the same folder as this Jupyter notebook. Since the data in both files is saved in the json-format, we will need the `json` Python module to load them.

[1]:
```
import json

with open("debates.json", "r") as f:
    debates = json.load(f)

with open("users.json", "r") as f:
    users = json.load(f)
```

The data is now saved in the two variables `debates` and `users`. Both are of the type `dict`. Let's have a quick look at the structure of the data. If we print the keys of the debates dict, we can see that the details of each debate on the portal can be accessed by its title.

[2]:
```
# Since there are so many keys, we better transform them to a list
# first to display only a fraction of them.
debate_keys = list(debates.keys())
print(debate_keys[:10])
```

```
['.-.-.-Ha-YOURE-GOING-DOWN-BEEMOR/1/', '.-Audis-are-junkers-except-to-rich-
kids-with-limitless-assets-and-time-to-maintain-them./1/', '…-Former-
Secretary-of-State-Madeleine-Albright-will-stand-with-Muslims-Are-we-
suprised/1/', '…Words-can-t-hurt-me-any./1/', '.9-repeated-is-equal-to-1./1/',
'.99-is-equal-to-one./1/', '.999-repeating-equals1/1/', '.999-repeating-is-
equal-to-1-in-reality/1/', '.999-Repeating-is-Equal-to-1/1/', '.999…-is-equal-
to-1/1/']
```

Each debate, in turn, is also saved in a dictionary, which we can print out as well to find out, which properties are present. Let's do that for a debate in the data and have a closer look at the text property (each of the debates are separated into rounds, which in turn has one argument/text from each side; so we need to use the `rounds` property to inspect the text). For convenience, I save the chosen debate into a new variable.

[3]:
```
any_debate = list(debates.values())[42]
print(f"Debate properties: {any_debate.keys()}")
```

```
Debate properties: dict_keys(['url', 'category', 'title', 'comments', 'votes',
'rounds', 'forfeit_label', 'forfeit_side', 'start_date', 'update_date',
```

```
'voting_style', 'debate_status', 'number_of_comments', 'number_of_views',
'number_of_rounds', 'number_of_votes', 'participant_1_link',
'participant_1_name', 'participant_1_points', 'participant_1_position',
'participant_1_status', 'participant_2_link', 'participant_2_name',
'participant_2_points', 'participant_2_position', 'participant_2_status'])
```

[4]:
```
# Print the first argument of the second debate round
print(f"Debate text: {any_debate['rounds'][1][1]['text']}")
```

```
Debate text:

Well 0.999999…doesnt stop but then again theres a missing .1 somewhere. If we
round 0.999999…to 3 decimal places it would be 0.999. For 0.999 to become
1,the number needs to be added with a 0.001. So although 0.999…doesnt stop,you
just need to round it off to a certain number of decimal places for lets say 3
decimal places and add a 0.001 from there on.

To correct you:3/3 or 3 divided by 3=1 not 0.99999 as 3/3 is one whole. You can
check with your calculator. I checked with mine,its 1 not 0.999999… If it was
0.999999… it would have shown 0.99999… like how 2/3 is shown as 0.66666667.
Now wait yes they rounded the number off but if you mentally divide it,why do
you think the answer is 1 instead of 0.999999…? And when you divide the
numbers,why do you not have any decimals to round off your answer to 1?

Question for my opponent:If 0.99999…=1, why do we call it 0.999999…
instead of 1?
```

Now that we know a thing or two about the debates structure, let's do the same for the user data.

[5]:
```
# Since there are so many keys, we better transform them to a list
# first to display only a fraction of them.
user_keys = list(users.keys())
print(user_keys[:10])
```

```
['000001', '00003', '000ike', '00110001', '00110022', '001Seraphina', '002682',
'007566', '007', '007Bond']
```

[6]:
```
any_user = user_keys[42]
print(f"Username: {any_user}")
print(f"User properties: {users[any_user]}")
```

```
Username: 100-Year-Old-Man
User properties: {'all_debates': ['Fallout-3-VS.-New-Vegas/1/'],
'big_issues_dict': {'Abortion': 'N/S', 'Affirmative Action': 'N/S', 'Animal
Rights': 'N/S', 'Barack Obama': 'N/S', 'Border Fence': 'N/S', 'Capitalism':
'N/S', 'Civil Unions': 'N/S', 'Death Penalty': 'N/S', 'Drug Legalization':
'N/S', 'Electoral College': 'N/S', 'Environmental Protection': 'N/S', 'Estate
```

Tax': 'N/S', 'European Union': 'N/S', 'Euthanasia': 'N/S', 'Federal Reserve': 'N/S', 'Flat Tax': 'N/S', 'Free Trade': 'N/S', 'Gay Marriage': 'N/S', 'Global Warming Exists': 'N/S', 'Globalization': 'N/S', 'Gold Standard': 'N/S', 'Gun Rights': 'N/S', 'Homeschooling': 'N/S', 'Internet Censorship': 'N/S', 'Iran-Iraq War': 'N/S', 'Labor Union': 'N/S', 'Legalized Prostitution': 'N/S', 'Medicaid & Medicare': 'N/S', 'Medical Marijuana': 'N/S', 'Military Intervention': 'N/S', 'Minimum Wage': 'N/S', 'National Health Care': 'N/S', 'National Retail Sales Tax': 'N/S', 'Occupy Movement': 'N/S', 'Progressive Tax': 'N/S', 'Racial Profiling': 'N/S', 'Redistribution': 'N/S', 'Smoking Ban': 'N/S', 'Social Programs': 'N/S', 'Social Security': 'N/S', 'Socialism': 'N/S', 'Stimulus Spending': 'N/S', 'Term Limits': 'N/S', 'Torture': 'N/S', 'United Nations': 'N/S', 'War in Afghanistan': 'N/S', 'War on Terror': 'N/S', 'Welfare': 'N/S'}, 'birthday': '- Private -', 'description': '113-year old', 'education': 'Not Saying', 'elo_ranking': '2,000', 'email': '- Private -', 'ethnicity': 'Not Saying', 'gender': 'Prefer not to say', 'friends': [], 'income': 'Not Saying', 'interested': 'No Answer', 'joined': '3 Years Ago', 'last_online': '3 Years Ago', 'last_updated': '3 Years Ago', 'looking': 'No Answer', 'lost_debates': [], 'number_of_all_debates': '1', 'number_of_lost_debates': '0', 'number_of_tied_debates': '1', 'number_of_won_debates': '0', 'number_of_friends': '0', 'number_of_opinion_arguments': '0', 'number_of_opinion_questions': '0', 'number_of_poll_topics': '0', 'number_of_poll_votes': '1', 'number_of_voted_debates': '0', 'opinion_arguments': [], 'opinion_questions': [], 'party': 'Not Saying', 'percentile': '71.38%', 'political_ideology': 'Not Saying', 'poll_topics': [], 'poll_votes': [{'vote link': '/polls/fallout-new-vegas-or-elder-scrolls-v-skyrim', 'vote text': 'Fallout: New Vegas', 'vote title': 'Fallout: New Vegas or Elder Scrolls V: Skyrim?', 'vote explanation': 'Skyrim can suck all it want. NEW VEGAS FTW!!'}], 'president': 'Not Saying', 'relationship': 'Not Saying', 'religious_ideology': 'Not Saying', 'url': 'http://www.debate.org/100-Year-Old-Man/', 'voted_debates': [], 'win_ratio': '0.00%', 'won_debates': [], 'tied_debates': ['Fallout-3-VS.-New-Vegas/1/']}

In order to make later tasks with the data a bit easier, lets compile the necessary information into a `pandas` `DataFrame` (`pandas` as in the Python library, not the PANDA course management). We will try to find each debate texts and the political ideology of the posting user. Since this is an introductory tutorial, we will focus on two main ideologies: Conservative and Liberal (mostly since those are the biggest groups in the data). Let's iterate over all debates, all rounds in those debates and all arguments in those rounds and save the respective texts, together with the political ideology of the posting user.

[7]:
```
debate_data = []

for key, debate in debates.items():
    # Sometimes, the users of the debate didn't exist anymore at the time
    # the data was collected. If this is the case, simply skip this debate.
    try:
```

```
        user1 = users[debate["participant_1_name"]]
        user2 = users[debate["participant_2_name"]]
    except KeyError:
        continue

    # For each round in this debate...
    for debate_round in debate["rounds"]:
        # For each argument in this round...
        for argument in debate_round:
            # Save the text and find the political ideology of the user.
            debate_data.append({
                "debate_text": argument["text"],
                "political_ideology": user1["political_ideology"]
                                      if argument["side"] ==␣
↪debate["participant_1_position"]
                                      else user2["political_ideology"]
            })
```

Now, let's filter the debates for the political ideology of the posting user to only include the two mentioned groups above...

[8]:
```
political_ideologies = ["Conservative", "Liberal"]
filtered_debates = list(filter(
    lambda x: x["political_ideology"] in political_ideologies,
    debate_data))
```

...and save the result into a `pandas DataFrame`. This makes following steps a bit easier and faster.

[9]:
```
import pandas as pd

debates_df = pd.DataFrame(
    columns=["debate_text", "political_ideology"],
    data=filtered_debates)
```

We can now also easily sample the data for each political ideology and extract the same number of samples for each, so that we have a more or less stratified sample. For the sake of timeliness, we are going to limit the number of samples included for this tutorial to the first 5000. This will make the preprocessing and feature extraction a lot faster. If you have more time, you can of course also try to do it with more samples. Let's double check that by printing the number of entries in each `DataFrame`. Also, we should combine the two into a single `DataFrame` again.

[10]:
```
conservative_arguments = debates_df[
    debates_df.political_ideology == "Conservative"].sample(n=2500)
```

```
liberal_arguments = debates_df[
    debates_df.political_ideology == "Liberal"].sample(n=2500)

# Combining both samples into one dataframe again
debates_df = conservative_arguments.append(liberal_arguments)

# Have a look of the length of each group
print(f"Conservative arguments: {liberal_arguments.shape}")
print(f"Liberal arguments: {liberal_arguments.shape}")
```

```
Conservative arguments: (2500, 2)
Liberal arguments: (2500, 2)
```

And that's already the end of this part. In the next, we will take the just extracted texts and try to clean them a bit.

3 Part 3 - Preprocessing using spaCy & Extraction of textual features

Welcome to the third part of the tutorial "Natural Language Processing in Python". In this part we will process the previously extracted data using the `spaCy` library to tokenize and clean the texts. Afterwards, we will try to extract meaningful features from it that we can use to train a machine learning algorithm to predict the political ideology of the author.

First, lets import spaCy and load the language model that you downloaded already (as mentioned in the first video).

[11]:
```
import spacy

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

The `nlp` variable now allows us to parse any text with the language model and tokenize it. Let's define a function where a given text is taken as a parameter and the cleaned version is returned as a list of tokens.

[12]:
```
def clean_text(text: str) -> list:
    # Parse the text using the English language model
    # The returned object is an iterator over all tokens
    parsed_text = nlp(text)
    # Initialize a list which will later hold the tokens of the text
    tokenized_clean_text = []

    # For each token in the text...
    for token in parsed_text:
        # If the token is _not_ one of the following, append it to
```

```
        # the final list of tokens; continue otherwise
        if (not token.is_punct and  # Punctuation
                not token.is_space and  # Whitespace of any kind
                not token.like_url and # Anything that looks like an url
                not token.is_stop):  # Stopwords
            tokenized_clean_text.append(token.text.lower())

    # Return the list of clean tokens for this text
    return tokenized_clean_text
```

Now let's apply this cleaning function to all texts in the `DataFrame`. We can use the `.apply()`-function to do so and save the result in a new column. Depending on your processor and included samples, this might take a some time to complete.

[13]:
```
debates_df["cleaned_text"] = debates_df["debate_text"].apply(clean_text)
```

We can now start with extracting features from the texts. As you may know, the Bag-of-Words methods is a common approach to do that. While not the most advanced one, it is simple to understand and is often used as a simple first baseline for more complex features. So let's transform each of the debates into Bag-of-Words feature vectors using the `scikit-learn` library. You can find the documentation for the `CountVectorizer` here.

[14]:
```
from sklearn.feature_extraction.text import CountVectorizer

# The CountVectorizer expects the documents as complete strings
# so we need to join our tokens back together
documents = [" ".join(document) for document in debates_df.cleaned_text.values]

# Initialize the vectorizer
vectorizer = CountVectorizer()

# Fit the vectorizer's vocabulary on the data
# and transform the input to vectors/features; save the result to `X`
X = vectorizer.fit_transform(documents)
```

Now that we have the feature vectors for our classification, we also need to encode our labels, since they are still only present as strings. To do that, we use the `LabelBinarizer` class.

[15]:
```
from sklearn.preprocessing import LabelBinarizer

# Initialize the encoder
lb = LabelBinarizer()

# The the encoder to the dictionary and transform the labels
```

```
# into numbers
y = lb.fit_transform(debates_df.political_ideology.values)

# Reshape the data into a one-dimensional list
y = y.reshape(len(y),)
```

In the last preparation step we now need to split our data into training and test sets. Again, we can use the `scikit-learn` library for that.

[16]:
```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y)
```

Now that the preparation of the data is done, we can move on to the actual machine learning. We will do this in the next and last part of this tutorial.

4 Part 4 - Training and Prediction of a classifier

Welcome to the fourth and last part of the tutorial "Natural Language Processing in Python". In this part we will use the previously prepared data and train a classifier on it and fine-tune it by finding better hyperparameter values. We will also look at how to evaluate the results and how to do a cross-fold validation.

First, let's import the classifier. We will again use the `scikit-learn` library for this. I choose a simple SVM classifier for this task, as it is fast and might just work for our use case. Since the library has a lot of different classifiers, all with the same API, you can also try others on your own. You can find all classifiers in the `scikit-learn` documentation.

[17]:
```
from sklearn.svm import SVC

# Initialize the SVM and upping the maximum iterations to
# allow for a longer training
clf = SVC(gamma="scale")

# Start the training on the training data
clf.fit(X_train, y_train)
```

[17]:
```
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=3, gamma='scale', kernel='rbf',
    max_iter=-1, probability=False, random_state=None, shrinking=True,
    tol=0.001, verbose=False)
```

Now we can simply use the trained classifier to predict labels for the test dataset and save it to a variable to later evaluate the results.

[18]:
```
y_pred = clf.predict(X_test)
```

To evaluate the predictions, we will use the F1 score implementation of the `scikit-learn` library.

[19]:
```
from sklearn.metrics import f1_score

# Evaluate the predictions and print the result
print(f1_score(y_test, y_pred))
```

```
0.3897216274089936
```

If we want to have some more reliable results, we can also utilize the cross-validation method of the `scikit-learn` library.

[20]:
```
from sklearn.model_selection import cross_val_score

print(cross_val_score(clf, X, y, scoring="f1", cv=5))
```

```
[0.36870027 0.36565097 0.36995828 0.36237898 0.39782016]
```

Additionally, it makes sense to do a hyperparameter optimization for our classifier. Again, the `scitkit-learn` library provides some easy-to-use function to start a grid or randomized search. Explaining these concepts is out of the scope of this tutorial, but you can find additional information here. Also, you can try to use a more advanced method, namely a *Bayesian optimization*. You can find additional information on the method here and a Python library implementing it here.

[21]:
```
from sklearn.model_selection import GridSearchCV

# Defining the grids that should be searched
param_grids = [
    {
        "C": [0.001, 0.01, 0.1, 1.0, 10.0, 100.0],
        "gamma": ["scale"],
        "kernel": ["linear", "poly"]
    },
    {
        "C": [0.001, 0.01, 0.1, 1.0, 10.0, 100.0],
        "gamma": [0.001, 0.0001],
        "kernel": ["rbf"]
    }
]

# Initializing a new classifier
grid_clf = SVC()
```

[22]:
```
# Initializing the grid search class
grid_search = GridSearchCV(grid_clf, param_grid=param_grids, cv=5, n_jobs=8,␣
↪scoring="f1")

# Starting the grid search
grid_search.fit(X, y)

# Print the best parameter combination
print(grid_search.best_params_)
```

```
{'C': 100.0, 'gamma': 0.001, 'kernel': 'rbf'}
```

[23]:
```
print(
    cross_val_score(
        SVC(**grid_search.best_params_),
        X,
        y,
        scoring="f1",
        cv=5))
```

```
[0.55044074 0.54404647 0.57475728 0.56374502 0.55414634]
```

And that's basically it. Of course, we could try to improve the results further by using a different classifier like a multi-layer perceptron or more complex neural networks. Additionally, we could choose different input features, such as Tfidf vectors or word embeddings. But the general steps described here will stay the same.

Apart from that, this is the end of this introductory tutorial. If you have any questions, feel free to ask us in the chat during the tutorial time or simply send us an E-Mail.