Social Media Data Analysis - A.A. 2023-2024

Text Analysis (NLP) - Classification

Francesco Ragusa https://iplab.dmi.unict.it/ragusa/t (https://iplab.dmi.unict.it/ragusa/) - francesco.ragusa@unict.it (mailto:francesco.ragusa@unict.it)

In this laboratory, we will see the main tools for text analysis. In particular, we will see:

- The basic element of an NLP pipeline:
  - Word tokenization;
  - Identifying stop words;
  - Stemming;
  - Lemmatization;
  - POS tagging;
  - NER tagging;
  - Sentence segmentation.
- The Bag Of Words (BOW) representation;
- Text classificaiton.

1. Text Processing Basics

We will use the `spaCy` Python library, which can be installed with the following commands from command line/anaconda prompt:

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

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

To use spaCy, we first need to import it and load a corpus of text on which the algorithms have been trained. Assuming that we will work with the English language, we will load the `en` corpus:

In [1]:
```
# import the spacy library and load the English web corpus
import spacy
nlp = spacy.load('en_core_web_sm')
```

The nlp object is associated with a vocabulary (i.e., a set of known words), which depends on the chosen corpus. We can see the length of the corpus as follows:

```
In [2]: len(nlp.vocab)
```

```
Out[2]: 489
```

We can see the list of al the words in the vocabulary as follows:

```
In [3]: words = [t.text for t in nlp.vocab]
        print(words[:10])#print the first 10 words
```

```
['nuthin', 'there', 'ü.', '’nuff', 'havin', "'bout", '’Cause', 'Need', 'Somet
hin', 'gon']
```

To analyse text with spaCy, we first need to create a document object:

```
In [4]: #document object
        doc = nlp("\"Let's go to N.Y.!\"")
        doc
```

```
Out[4]: "Let's go to N.Y.!"
```

When we define a document, the vocabulary is updated by inserting any word which is present in the document but was not present in the corpus. For instance, the size of the vocabulary is now larger:

```
In [5]: len(nlp.vocab)
```

```
Out[5]: 493
```

We can see which are the new words as follows:

```
In [6]: words2 = [t.text for t in nlp.vocab]
        set(words2)-set(words)
```

```
Out[6]: {'!', '"', 'go', 'to'}
```

As we will see a spaCy document allows to easilly perform some basic text processing operations.

1.1 Tokenization

Given a spaCy document, it is possible to easilly iterate over tokens with a simple for loop:

In [7]:
```
for t in doc:
    print(t)
```

```
"
Let
's
go
to
N.Y.
!
"
```

We can alternatively obtain the list of all tokens simply by passing doc to `list` :

In [8]:
```
tokens = list(doc)
print(tokens)
```

```
[", Let, 's, go, to, N.Y., !, "]
```

The `doc` object can also be indexed directly. So, if we want to access to the 5^{th} token, we can simply write:

In [9]:
```
doc[4]
```

Out[9]:
```
to
```

We should note that each token is not a string, but actually a token object:

In [10]:
```
type(doc[4])
```

Out[10]:
```
spacy.tokens.token.Token
```

We can access to the text contained in the token as follows:

In [11]:
```
print(doc[4].text)
print(type(doc[4].text))
```

```
to
<class 'str'>
```

Similarly, it is possible to access multiple tokens by slicing. This will return a `span` object:

In [12]:
```
print(doc[2:4])
print(type(doc[2:4]))
```

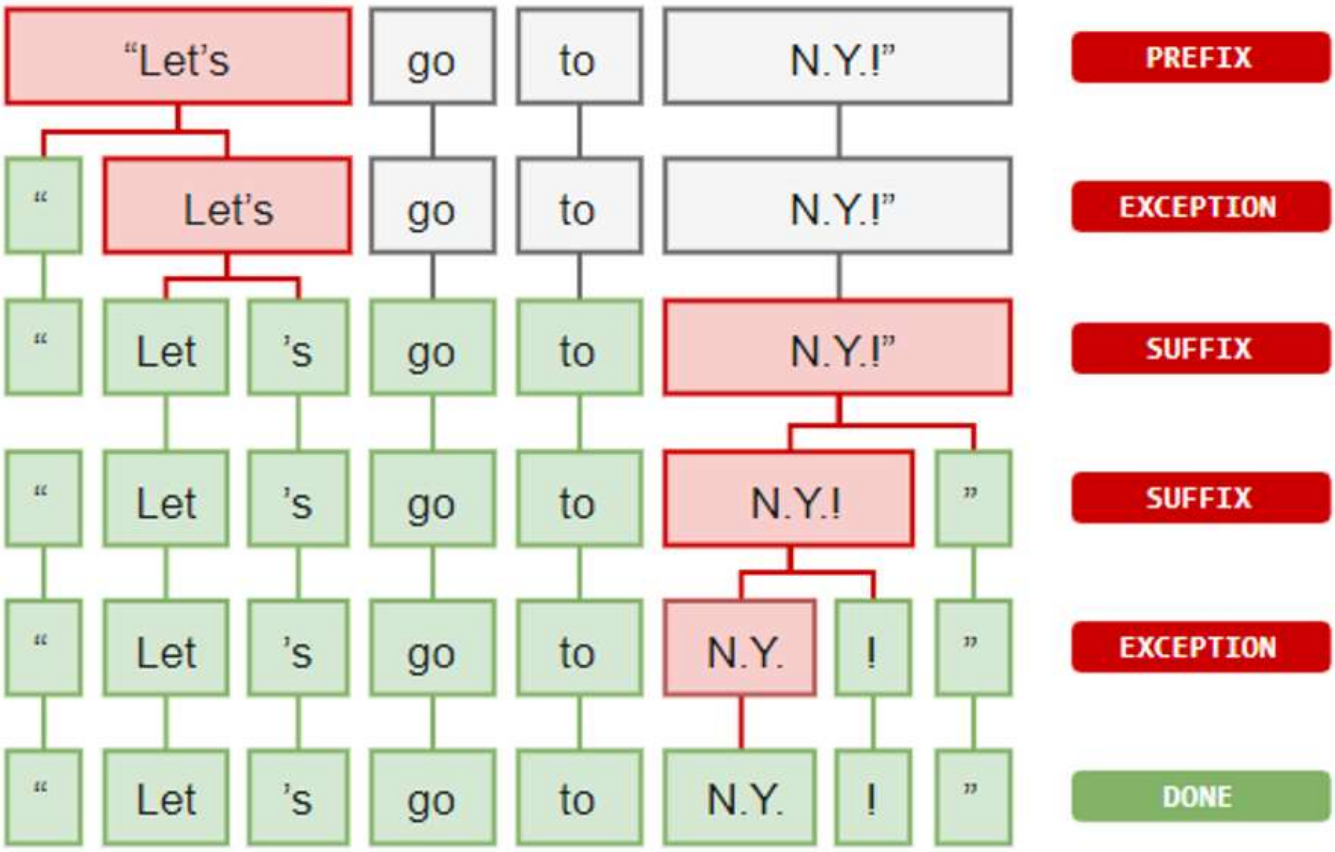
```
's go
<class 'spacy.tokens.span.Span'>
```

Even if we can index a document to obtain a token, tokens cannot be re-assigned:

In [13]:
```
try:
    doc[2]='a'
except:
    print('Error!')
```

```
Error!
```

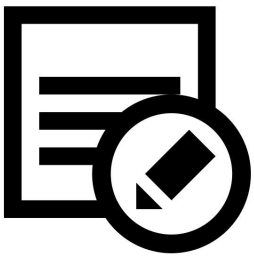
As we can see, spaCy takes care of all steps required to obtain a proper tokenization, including recognizing prefixes, suffixes, infixes and exceptions, as shown in the image below (image from [https://spacy.io/usage/spacy-101#annotations-token (https://spacy.io/usage/spacy-101#annotations-token)](https://spacy.io/usage/spacy-101#annotations-token)).



Question 1

Is the tokenization mechanism offered by spaCy useful at all? Compare the obtained tokenization with the result of splitting the string on spaces with `s.split(' ')`, where `s` is the variable containing the string.

Answer 1



1.2 Stemming

As we have discussed in the previous lessons, stemming is a crude way to map different words to a common root. Given the limitedness of stemming, and the fact that a stemmer can be replaced by a lemmatizer, SpaCy does not include a stemmer. To see some examples with stemming, we will use the `nltk` library. This can be installed with:

```
conda install -c anaconda nltk
```

Let's see how to use the Porter Stemmer:

In [15]:
```
#import the toolkit
import nltk
#import all functions from the porter stemmer library
from nltk.stem.porter import *
#instantiate the stemmer
p_stemmer = PorterStemmer()

words = ['go', 'goes', 'went', 'wish', 'wishes', 'wished', 'runner', 'ran', 'r
unning']

for w in words:
    print("{} -> {}".format(w,p_stemmer.stem(w)))
```

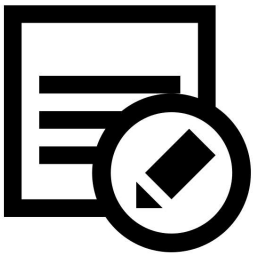
```
go -> go
goes -> goe
went -> went
wish -> wish
wishes -> wish
wished -> wish
runner -> runner
ran -> ran
running -> run
```

As we can see, some of the results make sense (e.g., wishes -> wish), but others don't (e.g., goes -> goe).

Question 2

Why does the Porter stemmer not always work?

Answer 2



1.3 Lemmatization

Lemmatization is a much more complex way to group words according to their meaning. This is done by looking both at a vocabulary (this is necessary to understand that, for instance 'knives' is the plural of 'knife') and at the context (for instance to understand if 'meeting' is used as a noun or as a verb). SpaCy performs lemmatization automatically and associates the correct lemma to each token.

In particular, apart from `text`, each token is assigned two properties:

- `lemma` : a numerical id which univocally identifies the lemma (this is for machines);
- `lemma_` : a string explaining the lemma (this is for humans);

For instance:

In [16]:
```
print(tokens[1].text)
print(tokens[1].lemma)
print(tokens[1].lemma_)
```

```
Let
278066919066513387
let
```

Let's see an example with a sentence:

In [17]:
```
doc = nlp("I will meet you in the meeting after meeting the runner when runnin
g.")
for token in doc:
    print("{} -> {}".format(token.text,token.lemma_))
```

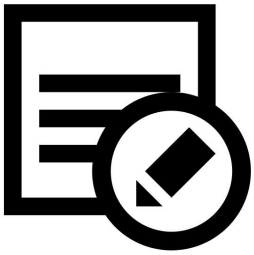
```
I -> -PRON-
will -> will
meet -> meet
you -> -PRON-
in -> in
the -> the
meeting -> meeting
after -> after
meeting -> meet
the -> the
runner -> runner
when -> when
running -> run
. -> .
```

The lemmatizer correctly associated the first occurrence of "meeting" as a verb (its lemma is "meet") and the second one as a noun (its lemma is "meet").

Question 3

Is a lemmatizer better than a stemmer? Compare the lemmas obtained in the previous example with the results obtained with the Porter stemmer.

Answer 3



1.4 Stop Words

Not all words are equally important. Some words such as "a" and "the" appear very frequently in the text and tell us little about the nature of the text (e.g., its category). These words are usually referred to as "stop words". SpaCy has a built in list of stop words for the English language. We can access them as follows:

In [18]:
```
print(len(nlp.Defaults.stop_words))
#let's print the first 10 stop words
print(list(nlp.Defaults.stop_words)[:10])
```

```
326
['least', 'more', 'none', 'into', 'that', 'less', 'than', 'under', 'yoursel
f', 'regarding']
```

We can check if a word is a stop word as follows:

In [19]:
```
"the" in nlp.Defaults.stop_words
```

Out[19]:
```
True
```

To make things easier, spaCy allows to assess if a given token is a stop word using the attribute `is_stop` :

In [20]:
```
for t in tokens:
    print("{} -> {}".format(t.text,t.is_stop))
```

```
" -> False
Let -> False
's -> True
go -> True
to -> True
N.Y. -> False
! -> False
" -> False
```

As we can see, some commong words such as "'s", "go" and "to" are stop words.

Depending on the problem we are trying to solve, we may want to remove some stop words or add our own stop words. For instance, let's say we think "go" is valuable, and it should not be considered a stop word. We can remove it as follows:

In [21]:
```
#we need to perform two steps
#also, remember to use only lowercase letters
nlp.Defaults.stop_words.remove('go')
nlp.vocab['go'].is_stop = False
```

Now, "go" is not considered as a stop word anymore:

In [22]:
```
for t in nlp("\"Let's go to N.Y.!\""):
    print("{} -> {}".format(t.text,t.is_stop))
```

```
" -> False
Let -> False
's -> True
go -> False
to -> True
N.Y. -> False
! -> False
" -> False
```

Similary, we can add a stop word as follows:

In [23]:
```
nlp.Defaults.stop_words.add('!')
nlp.vocab['!'].is_stop = True
```

Let's check if "!" is now a stop word:

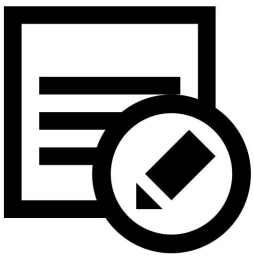
In [24]:
```
for t in nlp("\"Let's go to N.Y.!\""):
    print("{} -> {}".format(t.text,t.is_stop))
```

```
" -> False
Let -> False
's -> True
go -> False
to -> True
N.Y. -> False
! -> True
" -> False
```

Question 4

Why would we want to get rid of words which are very frequent? In which way is this related to the concept of information?

Answer 4



1.5 Part of Speech (POS) Tagging

SpaCy also allows to easily perform Part of Speech tagging. It is possible to assess which role a given token has in the text by using two properties of the tokens:

- `pos` : a numerical id which identifies the type of POS (for machines);
- `pos_` : a textual representation for the POS (for humans).

Let's see an example:

In [25]:
```
print(tokens[2].text)
print(tokens[2].pos_)
print(tokens[2].pos)
```

```
's
PRON
95
```

Let's see a more thorough example:

In [26]:
```
for t in nlp("\"Let's go to N.Y.!\""):
    print("{} -> {}".format(t.text,t.pos_))
```

```
" -> PUNCT
Let -> VERB
's -> PRON
go -> VERB
to -> ADP
N.Y. -> PROPN
! -> PUNCT
" -> PUNCT
```

The text contained in the `pos_` tags is a bit short. We can obtain a more datailed explanation with `spacy.explain` :

In [27]:
```
for t in nlp("\"Let's go to N.Y.!\""):
    print("{} -> {}".format(t.text,spacy.explain(t.pos_)))
```

```
" -> punctuation
Let -> verb
's -> pronoun
go -> verb
to -> adposition
N.Y. -> proper noun
! -> punctuation
" -> punctuation
```

We can access fine grained POS tags using `tag` and `tag_` :

In [28]:
```
for t in nlp("\"Let's go to N.Y.!\""):
    print("{} -> {}".format(t.text,t.tag_))
```

```
" -> ``
Let -> VB
's -> PRP
go -> VB
to -> IN
N.Y. -> NNP
! -> .
" -> ''
```

Similarly, we can obtain an explanation for each of the coarse grained tags:

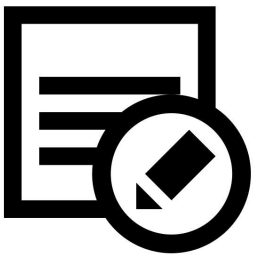
In [29]:
```
for t in nlp("\"Let's go to N.Y.!\""):
    print("{} -> {}".format(t.text,spacy.explain(t.tag_)))
```

```
" -> opening quotation mark
Let -> verb, base form
's -> pronoun, personal
go -> verb, base form
to -> conjunction, subordinating or preposition
N.Y. -> noun, proper singular
! -> punctuation mark, sentence closer
" -> closing quotation mark
```

Question 5

What is the difference between coarse and fine grained tags? Are there applications in which coarse grained tags can still be useful?

Answer 5



1.6 Named Entity Recognition (NER)

Named entity recognition allows to identify which tokens refer to specific entities such as companies, organizations, cities, money, etc. Named entities can be accessed with the `ents` property of a spaCy document:

In [30]:
```
doc = nlp('Boris Johnson is to offer EU leaders a historic grand bargain on Br
exit – help deliver his new deal this week or agree a “no-deal” departure by O
ctober 31.')
doc.ents
```

Out[30]:
```
(Boris Johnson, EU, Brexit, this week, October 31)
```

Each entity has the following properties:

- `text` : contains the text of the entity;
- `label_` : a string denoting the type of entity;
- `label` : an id for the entity; As usual, we can use `spacy.explain` to get more information on an entity:

In [31]:
```
for e in doc.ents:
    print("{} - {} - {} - {}".format(e.text, e.label, e.label_, spacy.explain
(e.label_)))
```

```
Boris Johnson - 380 - PERSON - People, including fictional
EU - 383 - ORG - Companies, agencies, institutions, etc.
Brexit - 384 - GPE - Countries, cities, states
this week - 391 - DATE - Absolute or relative dates or periods
October 31 - 391 - DATE - Absolute or relative dates or periods
```

SpaCy has a built-in visualizer for named entity:

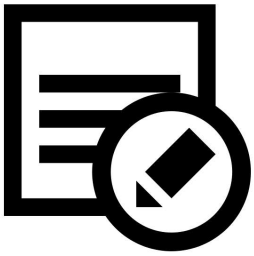
In [32]:
```
from spacy import displacy
displacy.render(doc, style='ent', jupyter=True)
```

Boris Johnson **PERSON** is to offer EU **ORG** leaders a historic grand bargain on Brexit **GPE** — help deliver his new deal this week **DATE** or agree a “no-deal” departure by October 31 **DATE** .

Question 6

How named entities are different from POS? Isn't this the same as knowing that a given token is a noun?

Answer 6



1.7 Sentence Segmentation

SpaCy also allows to perform sentence segmentation very easily by providing a `sents` property for each document:

In [33]:
```
doc = nlp("I gave you $3.5. Do you remember? Since I owed you $1.5, you should
now give me 2 dollars.")
list(doc.sents)
```

Out[33]:
```
[I gave you $3.5.,
 Do you remember?,
 Since I owed you $1.5, you should now give me 2 dollars.]
```

Also, we can check if a given token is the first token of a sentence using the property `is_sent_start` :

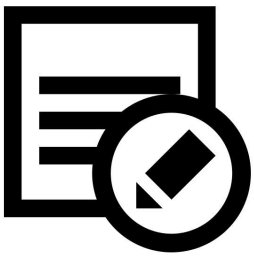
In [34]:
```
for t in doc:
    print("{} -> {}".format(t,t.is_sent_start))
```

```
I -> True
gave -> None
you -> None
$ -> None
3.5 -> None
. -> None
Do -> True
you -> None
remember -> None
? -> None
Since -> True
I -> None
owed -> None
you -> None
$ -> None
1.5 -> None
, -> None
you -> None
should -> None
now -> None
give -> None
me -> None
2 -> None
dollars -> None
. -> None
```

Question 7

Is the sentence segmentation algorithm necessary at all? Compare the obtained segmentation with the result of splitting the string on punctuation with `s.split('.')`.

Answer 7



2. Bag of Words Representation

We will now see how to represent text using a bag of words representaiton. To deal with a concrete example, we will consider the classificaiton task of distinguishing spam messages from legitimate messages. We will consider the dataset of SMS spam, available here: https://www.kaggle.com/uciml/sms-spam-collection-dataset/version/1# (https://www.kaggle.com/uciml/sms-spam-collection-dataset/version/1#).

The dataset can be downloaded after logging in. Download the file `spam.csv` and place it in the current working directory.

We will load the csv using Pandas:

In [35]:
```
import pandas as pd
#due to the way the CSV file has been saved,
#we need to specify the latin-1 encoding
spam = pd.read_csv('spam.csv', encoding='latin-1')
```

For this lab, we will use only the first two columns (the others contain mostly `None` elements). We will also rename them from 'v1' and 'v2' to something more meaningful:

In [36]:
```
spam=spam[['v1','v2']]
spam=spam.rename(columns={'v1':'class', 'v2':'text'})
spam.head()
```

Out[36]:

	class	text
0	ham	Go until jurong point, crazy.. Available only ...
1	ham	Ok lar... Joking wif u oni...
2	spam	Free entry in 2 a wkly comp to win FA Cup fina...
3	ham	U dun say so early hor... U c already then say...
4	ham	Nah I don't think he goes to usf, he lives aro...

Here `v1` represents the class, while `v2` contains the messages. Legitimate messages are called `ham`, as opposed to `spam` messages.

Let's inspect some messages:

In [37]:
```
print(spam.iloc[0]['class'], '---', spam.iloc[0]['text'])
print()
print(spam.iloc[15]['class'], '---', spam.iloc[15]['text'])
print()
print(spam.iloc[25]['class'], '---', spam.iloc[25]['text'])
```

```
ham --- Go until jurong point, crazy.. Available only in bugis n great world
la e buffet... Cine there got amore wat...

spam --- XXXMobileMovieClub: To use your credit, click the WAP link in the ne
xt txt message or click here>> http://wap. xxxmobilemovieclub.com?n=QJKGIGHJJ
GCBL

ham --- Just forced myself to eat a slice. I'm really not hungry tho. This su
cks. Mark is getting worried. He knows I'm sick when I turn down pizza. Lol
```

Since we will need to apply machine learning algorithms at some point, we should start by splitting the current dataset into training and testing sets. We will use the `train_test_split` function from `scikit-learn`. If the library is not installed, you can install it with the command:

```
conda install scikit-learn
```

Let's split the dataset:

In [38]:
```
from sklearn.model_selection import train_test_split
#we will set a seed to make sure
#that the split is always performed in the same way
#this is for instructional purposes only
#and it is not generally done when
#analyzing data to make sure that the
#split is truly random
import numpy as np
np.random.seed(1234)
#let's use 25% of the dataset as test set
train_set, test_set = train_test_split(spam, test_size=0.25)
```

Let's print some information about the two sets:

In [39]:
```
train_set.info(); print('\n'); test_set.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 4179 entries, 5062 to 2863
Data columns (total 2 columns):
class    4179 non-null object
text     4179 non-null object
dtypes: object(2)
memory usage: 97.9+ KB


<class 'pandas.core.frame.DataFrame'>
Int64Index: 1393 entries, 1537 to 4118
Data columns (total 2 columns):
class    1393 non-null object
text     1393 non-null object
dtypes: object(2)
memory usage: 32.6+ KB
```

Let's see the first elements of each set:

In [40]:
```
train_set.head()
```

Out[40]:

	class	text
5062	ham	Ok i also wan 2 watch e 9 pm show...
39	ham	Hello! How's you and how did saturday go? I wa...
4209	ham	No da:)he is stupid da..always sending like th...
4500	ham	So wat's da decision?
3578	ham	Multiply the numbers independently and count d...

In [41]:
```
test_set.head()
```

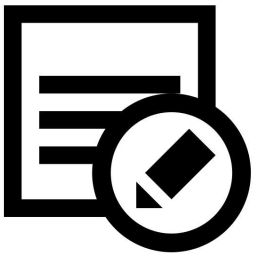
Out[41]:

	class	text
1537	ham	All sounds good. Fingers . Makes it difficult ...
963	ham	Yo chad which gymnastics class do you wanna ta...
4421	ham	MMM ... Fuck .... Merry Christmas to me
46	ham	Didn't you get hep b immunisation in nigeria.
581	ham	Ok anyway no need to change with what you said

Question 8

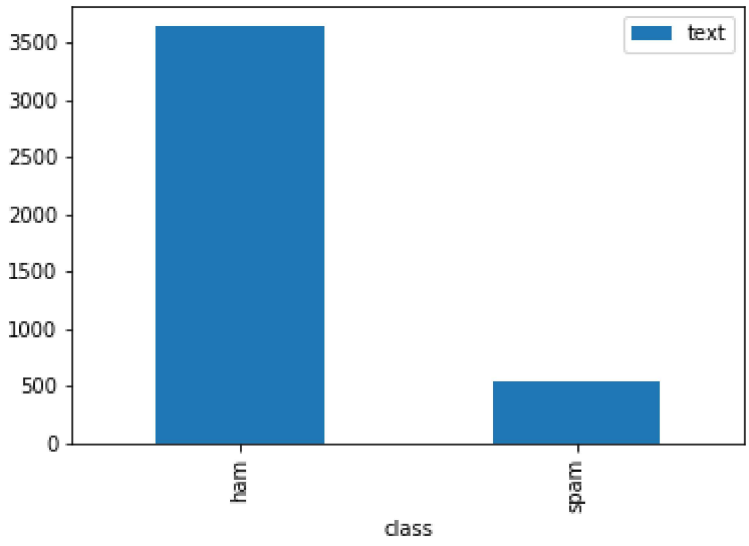
Compare the indexes of the two DataFrames with the indexes of the full `spam` DataFrame. In which order have the elements been sorted before splitting the dataset? Why?

Answer 8



We may want to check if how many elements belong to each category:

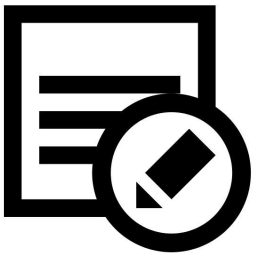
In [43]:
```
%matplotlib inline
from matplotlib import pyplot as plt
train_set.groupby('class').count().plot.bar()
plt.show()
```



Question 9

The dataset is very unbalanced. Why is this something we should keep in mind?

Answer 9



2.1 Tokenizating and counting words with CountVectorizer

In order to create our bag of words representation, we will need to tokenize each message, remove stop words and compute word counts. We could do this using spaCy. However, scikit-learn makes available some tools to perform feature extraction in an automated and efficient way.

To perform tokenization, we will use the `CountVectorizer` object. This object allows to process a series of documents (the training set), extract a vocabulary out of it (the set of all words appearning in the document) transform each document into a vector which reports the number of instances of each word. Let's import and create a `CountVectorizer` object:

In [44]:
```
from sklearn.feature_extraction.text import CountVectorizer
count_vect = CountVectorizer()
```

`CountVectorizer` uses a syntax which we will see is common to many `scikit-learn` objects:

- A method `fit` can be used to tune the internal parameters of the `CountVectorizer` object. In this case, this is mainly the vocabulary. The input to the method is a list of text messages;
- A method `transform` can be used to transform a list of documents into a sparse matrix in which each row is a vector containing the number of words included in each document. Since most of these numbers will be zero (documents don't usually contain all words), a sparse matrix is used instead of a conventional dense matrix to save memory;
- A method `fit_transform` which performs `fit` and `transform` at the same time.

Let's an example:

In [45]:
```
count_vect.fit(['this is', 'a list of', 'short messages'])
```

Out[45]:
```
CountVectorizer(analyzer='word', binary=False, decode_error='strict',
                dtype=<class 'numpy.int64'>, encoding='utf-8', input='conten
t',
                lowercase=True, max_df=1.0, max_features=None, min_df=1,
                ngram_range=(1, 1), preprocessor=None, stop_words=None,
                strip_accents=None, token_pattern='(?u)\\b\\w\\w+\\b',
                tokenizer=None, vocabulary=None)
```

We can access the vocabulary created by `CountVectorizer` as follows:

In [46]:
```
count_vect.vocabulary_
```

Out[46]:
```
{'this': 5, 'is': 0, 'list': 1, 'of': 3, 'short': 4, 'messages': 2}
```

The vocabulary is a dictionary which maps each word to a unique integer identifier. `CountVectorizer` also automatically removed stop words such as `a` . We can now transform text using the `transform` method:

In [47]:
```
features=count_vect.transform(['this is', 'a list of', 'short messages'])
features
```

Out[47]:
```
<3x6 sparse matrix of type '<class 'numpy.int64'>'
        with 6 stored elements in Compressed Sparse Row format>
```

As previously mentioned, the output will be a sparse matrix for memory efficiency. Since the matrix is small in our simple example, we can visualize its dense version with no trouble:

In [48]:
```
features=features.todense()
features
```

Out[48]:
```
matrix([[1, 0, 0, 0, 0, 1],
        [0, 1, 0, 1, 0, 0],
        [0, 0, 1, 0, 1, 0]], dtype=int64)
```

Each row fo the matrix corresponds to a document. Each column corresponds to a word, according to the ids included in the vocabulary dictionary. For instance, if we compare the first document with the corresponding vector:

In [49]:
```
print ('this is',features[0])
```

```
this is [[1 0 0 0 0 1]]
```

We note that it contains one instance of `this` (index 5 in the vocabulary) and one instance of `is` (index 0).

Interestingly, if a new document contains words which were not contained in the original corpus of documents, they are simply discarded:

In [50]:
```
features=count_vect.transform(['this is a new message'])
features.todense()
```

Out[50]:
```
matrix([[1, 0, 0, 0, 0, 1]], dtype=int64)
```

As we can see, `new` and `message` have been ignored as they were not in the original training set.

Let's compute word counts on all the training set:

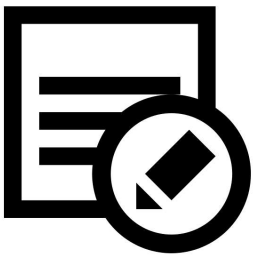
In [51]:
```
x_train = count_vect.fit_transform(train_set['text'])
x_train
```

Out[51]:
```
<4179x7398 sparse matrix of type '<class 'numpy.int64'>'
        with 55633 stored elements in Compressed Sparse Row format>
```

Question 10

Why `x_train` is represented as a sparse matrix? How much memory would we need to store it as a dense matrix?

Answer 10



The feature matrix is now a large sparse matrix with many rows (the number of samples) and many columns (the number of words). We can check the length of the vocabulary also as follows:

In [52]:
```
print(len(count_vect.vocabulary_))
```

```
7398
```

3 Nearest Neighbor

Word counts are a simple representation for text. Let's see how well we can classify samples by using this representation. We will consider the K-Nearest Neighbor classifier for this task. Using it is very straightforward with scikit-learn. Let's import the `KNeighborsClassifier` object and create a 1-NN:

In [53]:
```
from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier(n_neighbors=1)
```

This object has a similar interface:

- A `fit` method is used to tune the parameters of the classifier. In this case, it is used to provide (and memorize) the training set. We need to provide both the input features and the corresponding labels:
- A `predict` function is used to classify a sample;

In [54]:
```
knn.fit(x_train, train_set['class'])
```

Out[54]:
```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                     metric_params=None, n_jobs=None, n_neighbors=1, p=2,
                     weights='uniform')
```

We can now use the `knn` object to classify a new message, but first, we need to extract features from our message. Let's consider an example from the test set:

In [55]:
```
message = test_set.iloc[260]['text']
cl = test_set.iloc[260]['class']
message
```

Out[55]:
```
'FREE MSG:We billed your mobile number by mistake from shortcode 83332.Please
call 08081263000 to have charges refunded.This call will be free from a BT la
ndline'
```

To see if this is spam or ham, we need to first extract features form it:

In [56]:
```
feats=count_vect.transform([message])
feats
```

Out[56]:
```
<1x7398 sparse matrix of type '<class 'numpy.int64'>'
        with 20 stored elements in Compressed Sparse Row format>
```

Now we can classify it with `predict` :

In [57]:
```
knn.predict(feats)
```

Out[57]:
```
array(['ham'], dtype=object)
```

We can compare this with the actual class of the message:

In [58]:
```
cl
```

Out[58]:
```
'spam'
```

Apparently, the KNN classifier got it wrong. A good idea would be to assess the performance on a larger set of samples. Let's do this for the whole test set:

In [59]:
```
x_test = count_vect.transform(test_set['text'])
y_test_pred = knn.predict(x_test)
y_test_pred
```

Out[59]:
```
array(['ham', 'ham', 'ham', ..., 'ham', 'ham', 'ham'], dtype=object)
```

We now need to evaluate how good our classifier is at guessing the right class. We can compute accuracy using the `accuracy_score` function from scikit-learn:

In [60]:
```
from sklearn.metrics import accuracy_score
acc = accuracy_score(test_set['class'],y_test_pred)
acc
```

Out[60]:
```
0.9461593682699211
```

Alternatively, we can directly obtain the accuracy using the `score` method of the KNN object providing both the test features and labels:

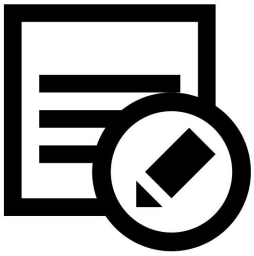
In [61]:
```
knn.score(x_test,test_set['class'])
```

Out[61]:
```
0.9461593682699211
```

Question 11

Our method achieved a good accuracy on the test set. Is this enough to evaluate its performance? What would be the performance of an approach systematically classifying messages as `ham` ?

Answer 11



Scikit-learn also offers a `confusion_matrix` function to compute confusion matrices:

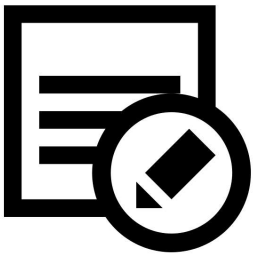
In [62]:
```
from sklearn.metrics import confusion_matrix
#labels=['ham','spam'] is needed to make sure that the first class is ham and
the second one is spam
cm = confusion_matrix(test_set['class'],y_test_pred, labels=['ham','spam'])
cm
```

Out[62]:
```
array([[1188,    1],
       [  74,  130]], dtype=int64)
```

Question 12

Compare the confusion matrix with the accuracy. Are we learning something different?

Answer 12



Similarly, we can compute per-class F_1 scores with the `f1_score` function:

In [63]:
```
from sklearn.metrics import f1_score
#average=None is needed to obtain per-class scores
#labels=['ham','spam'] is needed to make sure that the first score is for ham
and the second is for spam
f1_scores = f1_score(test_set['class'],y_test_pred, average=None, labels=['ha
m','spam'])
f1_scores
```

Out[63]:
```
array([0.96940024, 0.7761194 ])
```

We can obtain a single performance score (the mean F_1 score) by averaging the two f1 scores:

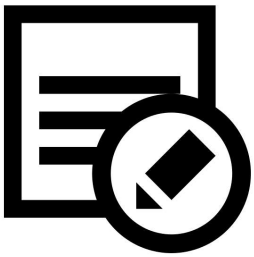
In [64]:
```
f1_scores.mean()
```

Out[64]:
```
0.8727598238915582
```

Question 13

What are the F1 scores telling us? Compare this with the accuracy and the confusion matrix.

Answer 13



We can easily repeat the process for a KNN with a different K. Let's try a 5-NN:

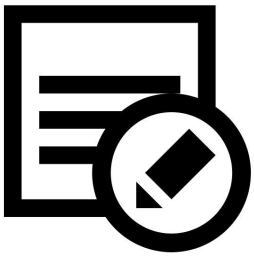
In [65]:
```
knn_5 = KNeighborsClassifier(n_neighbors=5)
knn_5.fit(x_train,train_set['class'])
y_test_pred_5 = knn_5.predict(x_test)
f1_scores_5=f1_score(test_set['class'],y_test_pred_5, average=None, labels=['h
am','spam'])
print(f1_scores_5, f1_scores_5.mean())
```

```
[0.94741036 0.52173913] 0.7345747445002598
```

Question 14

Which of the two classifiers worked best?

Answer 14



4. Advanced Tools

In this section, we will see some advanced tools which may be useful to obtain more sophisticated natural language processing pipelines.

4.1 TF-IDF

We have seen that, in practice, it can be useful to weight words by their frequency in the corpus and in the single document by using TF-IDF. This can be done very easily in `scikit-learn` using a `TfidfTransformer` transformer object. Let's see how this changes the results of a KNN classifier. For more clarity, we will report the entire training/test processing pipeline:

In [66]:
```
from sklearn.feature_extraction.text import TfidfTransformer
count_vect = CountVectorizer()
tf_transformer = TfidfTransformer()

x_train_counts = count_vect.fit_transform(train_set['text'])
x_train_tf_idf = tf_transformer.fit_transform(x_train_counts)

x_test_counts = count_vect.transform(test_set['text'])
x_test_tf_idf = tf_transformer.fit_transform(x_test_counts)

classifier = KNeighborsClassifier(n_neighbors=1)
classifier.fit(x_train_tf_idf, train_set['class'])

y_test_preds = classifier.predict(x_test_tf_idf)
f1_scores=f1_score(test_set['class'],y_test_preds, average=None, labels=['ha
m','spam'])
print(f1_scores, f1_scores.mean())
```

```
[0.97061224 0.78571429] 0.8781632653061224
```

When performing TF-IDF, we can skip the "idf" part. This way, the words will be only weighted by their frequency in the corpus. To do so, we need to specify a `use_idf=False` flag when creating the TF-IDF transformer:

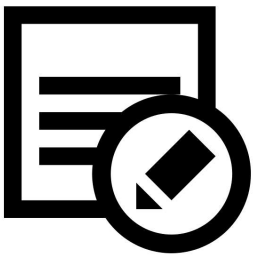
In [67]:
```
count_vect = CountVectorizer()
tf_transformer = TfidfTransformer(use_idf=False)

x_train_counts = count_vect.fit_transform(train_set['text'])
x_train_tf_idf = tf_transformer.fit_transform(x_train_counts)

x_test_counts = count_vect.transform(test_set['text'])
x_test_tf_idf = tf_transformer.fit_transform(x_test_counts)

classifier = KNeighborsClassifier(n_neighbors=1)
classifier.fit(x_train_tf_idf, train_set['class'])

y_test_preds = classifier.predict(x_test_tf_idf)
f1_scores=f1_score(test_set['class'],y_test_preds, average=None, labels=['ha
m','spam'])
print(f1_scores, f1_scores.mean())
```

```
[0.97454844 0.82285714] 0.8987027914614121
```

Question 16

Compare the results of the classifiers using TF-IDF with those obtained with the classifier using word counts. Which achieves better performance? Repeat the same comparison with a Naive Bayes classifier. Do we observe similar patterns? Why?

Answer 16



4.2 N-Grams

Sometimes, we may want to use n-grams rather than single words counts. `CountVectorizer` allows to handle this in a very simple way. In practice, it is sufficient to specify the range of ngrams to consider when building the `CountVectorizer` . For instance, if we want to consider from uni-grams to tri-grams, we can build our `CountVectorizer` by specifying `n_gram_range=(1,3)` . Let's see a simple example with just bi-grams:

In [68]:
```
count_vect = CountVectorizer(ngram_range=(2,2))
count_vect.fit(train_set['text'])
```

Out[68]:
```
CountVectorizer(analyzer='word', binary=False, decode_error='strict',
                dtype=<class 'numpy.int64'>, encoding='utf-8', input='conten
t',
                lowercase=True, max_df=1.0, max_features=None, min_df=1,
                ngram_range=(2, 2), preprocessor=None, stop_words=None,
                strip_accents=None, token_pattern='(?u)\\b\\w\\w+\\b',
                tokenizer=None, vocabulary=None)
```

If we inspect the vocabulary, we will see only bi-grams:

In [69]:
```
#Let's see 10 keys from the vocabulary
list(count_vect.vocabulary_.keys())[:10]
```

Out[69]:
```
['ok also',
 'also wan',
 'wan watch',
 'watch pm',
 'pm show',
 'hello how',
 'how you',
 'you and',
 'and how',
 'how did']
```

Inserting this in a classification pipeline is straightforward:

In [71]:
```
#from uni-gram to tri-gram
count_vect = CountVectorizer(ngram_range=(1,3))

x_train_counts = count_vect.fit_transform(train_set['text'])
x_test_counts = count_vect.transform(test_set['text'])

classifier = KNeighborsClassifier(n_neighbors=1)
classifier.fit(x_train_counts, train_set['class'])

y_test_preds = classifier.predict(x_test_counts)
f1_scores=f1_score(test_set['class'],y_test_preds, average=None, labels=['ha
m','spam'])
print(f1_scores, f1_scores.mean())
```

```
[0.96666667 0.74846626] 0.8575664621676891
```

However, beware that using n-grams slows down count vectorization and does not always improve the results. In general, this is task-dependent and trying it out to see if it helps might be a good idea.

4.3 Custom Tokenization

Scikit-learn does not include tools to perform stemming, lemmatization, part-of-speech tagging etc. In some cases, however, it can be useful to consider these as features (or as additional features). To achieve this, we can combine scikit-learn with an external library such as spaCy.

This is done by providing to the `CountVectorizer` object a custom tokenizer which splits a sentence into tokens. Let's build a tokenizer which considers parts of speech:

In [72]:
```
class POSTokenizer(object):
    def __init__(self):
        self.nlp = spacy.load('en_core_web_sm')
    def __call__(self, doc):
        return [t.pos_ for t in self.nlp(doc)]
```

We can use the tokenizer to split any sentence into tokens:

In [73]:
```
tokenizer = POSTokenizer()
tokenizer('Hello, How are you?')
```

Out[73]:
```
['INTJ', 'PUNCT', 'ADV', 'AUX', 'PRON', 'PUNCT']
```

We can hence build a `CountVectorizer` which uses the POSTokenizer as follows:

In [74]:
```
count_vect_pos = CountVectorizer(tokenizer=POSTokenizer())
```

This can be easily integrated into a classification pipeline as follows:

In [76]:
```
class POSTokenizer(object):
    def __init__(self):
        self.nlp = spacy.load('en_core_web_sm')
    def __call__(self, doc):
        return [t.pos_ for t in self.nlp(doc)]

count_vect_pos = CountVectorizer(tokenizer=POSTokenizer())

x_train_pos = count_vect_pos.fit_transform(train_set['text'])
x_test_pos = count_vect_pos.transform(test_set['text'])

classifier = KNeighborsClassifier(n_neighbors=1)
classifier.fit(x_train_pos, train_set['class'])

y_test_preds = classifier.predict(x_test_pos)
f1_scores=f1_score(test_set['class'],y_test_preds, average=None, labels=['ha
m','spam'])
print(f1_scores, f1_scores.mean())
```

```
[0.96003401 0.78341014] 0.8717220759271451
```

Note that by using coarse-grained POS, we only have 18 features:

In [77]:
```
x_train_counts.shape
```

Out[77]:
```
(4179, 83378)
```

We can combine these 18 features with the previous representation based on word counts by concatenating the features:

In [78]:
```
from scipy.sparse import hstack
count_vect = CountVectorizer()
x_train_counts = count_vect.fit_transform(train_set['text'])
x_train=hstack([x_train_counts,x_train_pos])
print(x_train_counts.shape, x_train_pos.shape, x_train.shape)
```

```
(4179, 7398) (4179, 18) (4179, 7416)
```

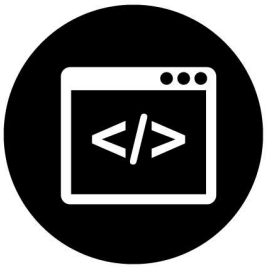
Let's integrate this into the classification pipeline:

In [81]:
```
class POSTokenizer(object):
    def __init__(self):
        self.nlp = spacy.load('en_core_web_sm')
    def __call__(self, doc):
        return [t.pos_ for t in self.nlp(doc)]

count_vect = CountVectorizer()
count_vect_pos = CountVectorizer(tokenizer=POSTokenizer())

x_train_count = count_vect.fit_transform(train_set['text'])
x_test_count = count_vect.transform(test_set['text'])

x_train_pos = count_vect_pos.fit_transform(train_set['text'])
x_test_pos = count_vect_pos.transform(test_set['text'])

x_train=hstack([x_train_count,x_train_pos])
x_test=hstack([x_test_count,x_test_pos])

classifier = KNeighborsClassifier(n_neighbors=1)
classifier.fit(x_train, train_set['class'])

y_test_preds = classifier.predict(x_test)
f1_scores=f1_score(test_set['class'],y_test_preds, average=None, labels=['ha
m','spam'])
print(f1_scores, f1_scores.mean())
```

```
[0.97614064 0.85642317] 0.9162819092123535
```

Similar kinds of processing can be perfomed using other tokes such as named entities.

Exercises



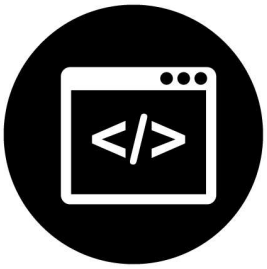
Exercise 1

Consider the following wikipedia page:

https://en.wikipedia.org/wiki/Harry_Potter (https://en.wikipedia.org/wiki/Harry_Potter)

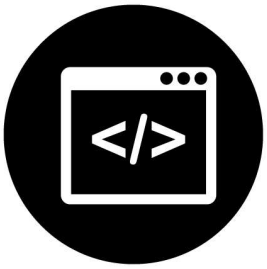
Download it and extract the text using Beautiful Soup. Hence, perform the following processing:

- Split the document into sentences;
- Split the document into tokens;
- Extract all named entities;
- Extract all coarse-grained POS tags and create and summarize their distribution in the text with a bar plot;



Exercise 2

Train ham-vs-spam classifiers based on a bag of words representation which considers only named entities. Compare the performance of a 1-NN with those of a Naive Bayes classifier.



Exercise 3

Use the `fetch_20newsgroups` of scikit-learn (https://scikit-learn.org/stable/modules/generated/sklearn.datasets.fetch_20newsgroups.html (https://scikit-learn.org/stable/modules/generated/sklearn.datasets.fetch_20newsgroups.html)) to download the 20 newsgroups dataset. This dataset contains documents belonging to 20 different classes, already divided into training and test set. Build a classifier to distinguish between the 20 classes. Evaluate the classifier using confusion matrix and F_1 scores. Try different classifiers, different representations and different parameters to achieve the highest performance on the test set.

Referenze

- spaCy documentation: https://spacy.io/ (https://spacy.io/)
- NLTK documentation: https://www.nltk.org/ (https://www.nltk.org/)
- Scikit-learn tutorial on text processing: https://scikit-learn.org/stable/tutorial/text_analytics/working_with_text_data.html (https://scikit-learn.org/stable/tutorial/text_analytics/working_with_text_data.html)