CE807 Lab 1
Text preprocessing with Python

Most of the labs in this module will be Python-based. In this lab we will familiarize ourselves with some of the packages available in Python for training text classifiers / information extraction systems, and for doing text preprocessing. If you've never used Python before you should still be able to follow what we're doing but do consider taking a crash course, e.g online – it's time well-spent, Python is really a great language for text analytics.

1 A quick intro to Linux

1.1 Linux

There is a great deal of open source / freely available NLP software, and the great majority of this is designed to work on Unix machines – primarily Linux but also Macs. It is therefore a good idea to familiarize yourselves with this type of platform.

1.2 Using Ubuntu on the lab machines

The version of Linux installed on our machines is called Ubuntu.

1. Restart the computer in Ubuntu if it's not already there
2. Open a Terminal (by clicking right on the desktop)

If you haven't used Linux before, spend some time in this lab familiarizing yourself with it. It would be good in particular if you learned about

- how to use a shell (the command interpreter in the terminal), in particular the bash shell
- how to use environment variables.

There are lots of tutorials online to Linux in general and in particular to the bash shell: see, e.g., http://www.linux.org/forums/beginner-tutorials.53/ (more refs on the website)

2. Basic text processing with SciKit Learn

SciKit Learn is an open-source Python library for machine learning that comes with basic facilities for processing text to support clustering and classification – including tokenization, word counting, and stemming. In this part of the lab we will briefly review how to use SciKit Learn in Python 2.7 (many of you should have already become familiar with the package in the Machine Learning module), then look in more detail at the preprocessing facilities.

2.1 Using SciKit Learn

[This section may be skipped by those of you who are already familiar with the library]

The websites associated with scikit-learn are

http://scikit-learn.org/

(overall site, documentation and tutorials, and more) and
https://github.com/scikit-learn/scikit-learn

(download). The library should already be installed in the lab machines however.

In order to use scikit-learn, you need first of all to start Python (e.g., from idle). We're going to use Python 2.7 in this lab but if you're familiar with Python 3 go ahead.

To start an idle Python interface, type in your terminal:

```
Idle-python2.7
```

After you started idle or whatever interface you prefer (ipython, etc), do the following to load scikit-learn and numpy into Python, (>>> is the IDLE prompt):

```
>>> import sklearn
>>> import numpy as np
```

We will see a lot about the machine learning facilities included with SciKit-Learn in the following weeks. Another important aspect of the package is that it interfaces well with visualization packages such as pyplot, that you should load:

```
>>> import matplotlib.pyplot as plt
```

Next, you should create some data to visualize – e..g, a bidimensional array representing a function:

```
>>>data = np.array([[1,2], [2,3], [3,4], [4,5], [5,6]])
>>>x = data[:,0]
>>>y= data[:,1]
```

You can now visualize the function as follows:

```
>>> plt.scatter(x,y)
<matplotlib.collections.PathCollection object at 0x1073d7750>
>>> plt.grid(True)
>>> plt.show()
```

2.2 Text Preprocessing with SciKit Learn

Many of the text analytics applications we are going to consider require taking a text (e.g., a post), tokenizing it , and using the tokens as features, possibly after lemmatization / stop word removal. With SciKit Learn we do not need to write code to do that; we can use the CountVectorizer class instead. An instance of the class is created as follows:

```
>>> from sklearn.feature_extraction.text import CountVectorizer
>>> vectorizer = CountVectorizer(min_df=1)
```

Once we have created the instance, we can use it to extract a **bag of words** representation from a collection of documents using the SciKit-Learn method fit_transform. In this first toy example, we use a list of strings as documents, as follows:

```
>>> content = ["How to format my hard disk", " Hard disk format problems "]
>>> X = vectorizer.fit_transform(content)
```

fit_transform has extracted seven features from the two `documents' ; we can see that using the method get_feature_names().

```
>>> vectorizer.get_feature_names()
['disk', 'format', 'hard', 'how', 'my', 'problems', 'to']
```

You can see how many times each of these seven features occurs in the two `documents' by doing:

```
>>> X.toarray()
array([[1, 1, 1, 1, 1, 0, 1], [1, 1, 1, 0, 0, 1, 0]])
```

Notice that this call returns an array of two rows, one per `document'. Each row as seven elements. Each element specifies the number of items a given feature occurred in that document. So

```
>>> X.toarray()[0]
array([1, 1, 1, 1, 1, 0, 1])
```

gives us the vector for the first document ("How to format my hard disk"), which contains all the words chosen as features except for 'problems'. In turn,

```
>>> X.toarray()[1,2]
1
```

gives us the number of times word 'hard' occurs in the second document.

CountVectorizer has a number of very useful options, discussed at the page:
http://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html

you should spend some time now familiarizing yourself with them.

Let us now see how this works with a real collection of documents. We are going to use the ’20 Newsgroups’ dataset, which is a collection of around 20,000 documents from 20 different newsgroups, that is commonly used in experiments on text classification and text clustering. The dataset can be found at:

http://qwone.com/~jason/20Newsgroups/

But it is already included in scikit-learn and can be loaded into Python by doing:

```
>>> from sklearn.datasets import fetch_20newsgroups
```

To speed up things, in the rest of the lab we will only use a subset of the documents, those belonging to the following 4 categories:

```
>>> categories = ['alt.atheism', 'soc.religion.christian', 'comp.graphics', 'sci.med']
```

We can import the documents belonging to those categories as follows:

```
twenty_train = fetch_20newsgroups(subset='train', categories=categories,
shuffle=True, random_state=42)
```

(The first time you do this, it’s going to take a while, don’t worry. You may also may get a warning message about finding no handlers, ignore it)

The files have now been loaded in the data attribute of the twenty_train object.

Let us now create again a CountVectorizer object:

```
>>> from sklearn.feature_extraction.text import CountVectorizer
>>> vectorizer = CountVectorizer()
```

Again, the fit_transform function can be used to tokenize each document, identify the most relevant words, build a dictionary of such words, and create for each document a vector representation in which the words are features and the value of these features is the number of occurrences of each word in a document, just as in the previous toy example:

```
>>> train_counts = vectorizer.fit_transform(twenty_train.data)
```

For instance, we can now see how frequently the word ‘algorithm’ occurs in the subset of the 20Newgroups collection we are considering as follows:

```
>>> vectorizer.vocabulary_.get('algorithm')
```

Question what frequency do you get?

To see how many terms were extracted, we can use get_feature_names() that we saw earlier:

```
>>> len(vectorizer.get_feature_names())
```

35483

SciKit's CountVectorizer class can do more preprocessing of a collection of documents than simple tokenization. One important additional preprocessing step that the class can carry out is *stop word removal*. This can be done by specifying a value for the stop_words parameter of CountVectorizer, as follows:

```
>>> vectorizer = CountVectorizer(stop_words='english')
```

To see what words are considered stop words, do the following:

```
>>> sorted(vectorizer.get_stop_words())[:20]
['a', 'about', 'above', 'across', 'after', 'afterwards', 'again', 'against', 'all', 'almost',
'alone', 'along', 'already', 'also', 'although', 'always', 'am', 'among', 'amongst',
'amoungst']
```

For stemming and more advanced preprocessing, we need to supplement SciKit Learn with another Python library, NLTK. We discuss this next.

3. More advanced preprocessing with NLTK

NLTK is an open source Python library available from
http://www.nltk.org

that supports most of the types of preprocessing we discussed in yesterday's lectures, from POS tagging to chunking, *for English.*It also comes with several useful resources such as corpora and lexicon.

NLTK is described in detail in a book by Bird, Klein and Loper available online:

http://www.nltk.org/book_1ed/ for the version for Python 2.7
http://www.nltk.org/book/ for the version for Python 3

NLTK is a huge library, much bigger than SciKit Learn in fact, as it contains also its own implementation of many machine learning algorithms. We will only try here some of its functionalities.

Importing NLTK NLTK can be imported into Python by doing:

```
>>> import nltk
```

Stemming NLTK includes implementations of several well-known stemming algorithms, including the Porter Stemmer and the Lancaster Stemmer. (See

http://www.nltk.org/howto/stem.html

for a general intro, and

http://www.nltk.org/api/nltk.stem.html

for more details, including the languages covered.) To create an English stemmer you need to do the following:

```
>>> s = nltk.stem.SnowballStemmer('english')
```

After creating the stemmer, you can then use it to stem words as follows:

```
>>> s.stem("cats")
u'cat'
```

Other types of preprocessing NLTK includes implementations of many of the preprocessing modules and syntactic analyzers we discussed in the lectures:

- language identifiers
- tokenizers for a number of languages
- sentence splitters
- POS taggers
- Chunkers
- parsers

You may want to try out these tools, they will come out useful in the assignments. *In order to do this on the Lab machines you may have to download some NLTK datasets as not all have been downloaded. You do that by using nltk.download() and then choosing the package you need or directly, nltk.download(<package>).*

In addition, NLTK includes implementations of aspects of text analysis that we will discuss in this module, including

- NERs
- Sentiment analysis
- Extracting information from social media.

For example, the following instructions *(you may need to download the 'punkt' nltk package to do this)*

```
>>> from nltk.tokenize import word_tokenize
>>> nltk.download("punkt") #if not used nltk.download()
>>> text = word_tokenize("And now for something completely different")
```

produce a tokenized version of the sentence, that can then be fed into the in-built POS tagger *(you may need to download the 'maxent_...' package on a Maximum Entropy classifiert to do this)*

```
>>> nltk.download('averaged_perceptron_tagger')
>>> nltk.download('maxent_treebank_pos_tagger')
>>> nltk.pos_tag(text)
[('And', 'CC'), ('now', 'RB'), ('for', 'IN'), ('something', 'NN'), ('completely', 'RB'),
('different', 'JJ')]
```

For a complete list of NLTK's modules, see
http://www.nltk.org/py-modindex.html

4. Integrating the NLTK stemmer with SciKit's vectorizer

The NLTK stemmer can be used to stem documents before feeding into SciKit's vectorizer, thus obtaining a more compact index.

One way to do this is to define a new class StemmedCountVectorizer extending CountVectorizer by redefining the method build_analyzer() that handles preprocessing and tokenization:

http://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html

build_analyzer() takes a string as input and outputs a list of tokens:

```
>>> from sklearn.feature_extraction.text import CountVectorizer
>>> vectorizer = CountVectorizer(stop_words='english')
>>> analyze = vectorizer.build_analyzer()
>>> analyze("John bought carrots and potatoes")
['john', 'bought', 'carrots', 'potatoes']
```

if we modify build_analyzer() to apply the NLTK stemmer to the output of default build_analyzer(), we get a version that does stemming as well:

```
>>> import  nltk.stem
>>> english_stemmer = nltk.stem.SnowballStemmer("English")
>>> class StemmedCountVectorizer(CountVectorizer):
        def build_analyzer(self):
            analyzer = super(StemmedCountVectorizer, 
self).build_analyzer()
            return lambda doc: (english_stemmer.stem(w) for w in 
analyzer(doc))
```

we can now create an instance of this class:

```
>>> stem_vectorizer = StemmedCountVectorizer(min_df=1, stop_words="english")
>>> stem_analyze = stem_vectorizer.build_analyzer()
```

as you can see, this new vectorizer uses stemmed versions of tokens:

```
>>> Y = stem_analyze("John bought carrots and potatoes")
print(Y)
<generator object <genexpr> at 0x106adb2d0>
>>> for tok in Y:
        print(tok)

john
```

```
bought
carrot
potato
```

If we use this vectorizer to extract features for the subset of the 20_Newsgroups dataset we considered earlier, we’re going to get fewer features:

```
>>> from sklearn.datasets import fetch_20newsgroups
>>> categories = ['alt.atheism','soc.religion.christian',
                'comp.graphics', 'sci.med']
>>> twenty_train = fetch_20newsgroups(subset='train', categories=categories, shuffle=True,
random_state=42)
>>> train_counts = stem_vectorizer.fit_transform(twenty_train.data)
>>> len(stem_vectorizer.get_feature_names())
26889
>>>
```

(Compare this with the around 35,000 features we obtained using the unstemmed version.)