Faculty of Computer Science, Dalhousie University *11-Oct-2024*

CSCI 4152/6509 — Natural Language Processing

Lab 5: Python NLTK Tutorial 1

Lab Instructor: Sigma Jahan and Tymon Wranik-Lohrenz
Location: Mona Campbell 1108 (10am)/Goldberg CS 134 (4pm)
Time: Friday, 10:04–11:25 and 16:05–17:25
Notes author: Dijana Kosmajac, Vlado Keselj

Python NLTK Tutorial 1

Lab Overview

- Introduction to Natural Language Toolkit (NLTK)
- Python quick overview;
- Lexical analysis: Word and text tokenizer;
- n-gram and collocations;
- NLTK corpora;
- Naïve Bayes classifier with NLTK.

Files to be submitted:

1. `lab5-list_merge.py`
2. `lab5-stop_word_removal.py`
3. `lab5-explore_corpus.py`
4. `lab5-movie_rev_classifier.py`

This is the first of three Python tutorials in the course. Many students may have seen Python before, so to make it more interesting and novel we will also use Python in the context of some NLP tasks, and use some NLP libraries. From the start and this lab, we will use the NLTK Python library (Natural Language Toolkit).

What is NLTK?

Natural Language Toolkit (NLTK) is a popular platform for building Python programs to work with human language data; i.e., for Natural Language Processing. It is accompanied by a book that explains the underlying concepts behind the language processing tasks supported by the toolkit. NLTK is intended to support research and teaching in NLP or closely related areas, including empirical linguistics, cognitive science, artificial intelligence, information retrieval, and machine learning.

We will start with a quick Python introduction, but if you would like to learn more about Python, there are many resources on the Web and books. For a simple beginner Python tutorial take a look at:
`http://www.tutorialspoint.com/python/python_tutorial.pdf`

As in previous labs, we will login to the server `timberlea` for this lab, which has the NLTK installed. If you want to install NLTK to your local machine, you can refer to the following URLs:
`http://www.nltk.org/install.html`
`http://www.nltk.org/data.html`

In this lab we will explore:

- Python quick overview;
- Lexical analysis: Word and text tokenizer;

– n-gram and collocations;
– NLTK corpora;
– Naïve Bayes classifier with NLTK.

Python overview

Basic syntax

Identifiers: Python identifier is a name used to identify a variable, function, class, module, or other object. An identifier starts with a letter A to Z or a to z, or an underscore (_) followed by zero or more letters, underscores and digits (0 to 9). Other characters are not allowed in identifiers, so be careful not to start variables as in Perl with special characters @, $, or %. The identifiers are case-sensitive, so for example, `Variable` and `variable` are two different identifiers.

Lines and Indentation: Python provides no braces to indicate blocks of code for class and function definitions or flow control. Blocks of code are denoted by line indentation, which is rigidly enforced. The number of spaces in the indentation is variable, but all statements within the block must be indented the same amount.

Quotation: Python accepts single (`'`), double (`"`) and triple (`'''` or `"""`) quotes to denote string literals, as long as the same type of quote starts and ends the string. Example:

```
word = 'word'
sentence = "This is a sentence."
paragraph = """This is a paragraph. It is
               made up of multiple lines and sentences."""
```

Data types, assigning and deleting values: Python has five standard data types:

– numbers;
– strings;
– lists;
– tuples;
– dictionaries.

Python variables do not need explicit declaration to reserve memory space. The declaration happens automatically when you assign a value to a variable. The equal sign (=) is used to assign values to variables. The operand to the left of the = operator is the name of the variable and the operand to the right of the = operator is the value stored in the variable. For example:

```
counter = 100    # An integer assignment
miles = 1000.0   # A floating point
name = "John"    # A string
```

Lists   Lists can contain a list or sequence of objects in Python. They are similar to lists (or arrays) in Perl. Python uses brackets ('[' and ']') to denote lists.

```
print(len([1, 2, 3]))        # 3 - length
print([1, 2, 3] + [4, 5, 6]) # [1, 2, 3, 4, 5, 6] - concatenation
print(['Hi!'] * 4)           # ['Hi!', 'Hi!', 'Hi!', 'Hi!']
                             #  - repetition
print(3 in [1, 2, 3])        # True - checks membership
for x in [1, 2, 3]: print(x) # 1 2 3 - iteration
```

Some of the built-in functions useful in work with lists are `max`, `min`, `cmp`, `len`, `list` (converts tuple to list), etc.

Some of the list-specific functions are `list.append`, `list.extend`, `list.count`, etc.

Tuples   Tuples are similar to lists, in the sense that they also contain sequences of objects. One difference is that tuples are immutable; i.e., cannot be changed once created, and because of that they are more efficient. Tuples are denoted by parentheses ('(' and ')') instead of brackets.

```
tup1 = ('physics', 'chemistry', 1997, 2000);
tup2 = (1, 2, 3, 4, 5, 6, 7);
print(tup1[0])      # prints: physics
print(tup2[1:5])    # prints: [2, 3, 4, 5]
```

Basic tuple operations are same as with lists: length, concatenation, repetition, membership and iteration.

Dictionaries.   Dictionaries are structures that map elements called keys to other elements called values. Hence they are similar to associative arrays in Perl, and they are also called hashes or maps in some languages.

```
dict = {'Name':'Zara', 'Age':7, 'Class':'First'}
dict['Age'] = 8                 # update existing entry
dict['School'] = "DPS School" # Add new entry
del dict['School]               # Delete existing entry
```

List comprehension.   Comprehensions are constructs that allow sequences to be built from other sequences. Python 2.0 introduced list comprehensions and Python 3.0 comes with dictionary and set comprehensions. The following is the example:

```
a_list = [1, 2, 9, 3, 0, 4]
squared_ints = [e**2 for e in a_list]

print(squared_ints)         # [ 1, 4, 81, 9, 0, 16 ]
```

This is same as:

```
a_list = [1, 2, 9, 3, 0, 4]
squared_ints = []
for e in a_list:
    squared_ints.append(e**2)

print(squared_ints)         # [ 1, 4, 81, 9, 0, 16 ]
```

Now, let us see an example with the 'if' statement. The example shows how to filter out non integer types from mixed list and apply operations.

```
a_list = [1, '4', 9, 'a', 0, 4]
squared_ints = [ e**2 for e in a_list if type(e) is int ]

print(squared_ints)       # [ 1, 81, 0, 16 ]
```

However, if you want to include an 'if-else' statement, the arrangement looks a bit different.

```
a_list = [1, '4', 9, 'a', 0, 4]
squared_ints = [ e**2 if type(e) is int else 'x' for e in a_list]

print(squared_ints)         # [1, 'x', 81, 'x', 0, 16]
```

You can also generate dictionary using list comprehension:

```
a_list = ["I", "am", "a", "data", "scientist"]
science_list = { e:i for i, e in enumerate(a_list) }

print(science_list)   # {'I': 0, 'am': 1, 'a': 2, 'data': 3,
                      #  'scientist': 4}
```

... or list of tuples:

```
a_list = ["I", "am", "a", "data", "scientist"]
science_list = [ (e,i) for i, e in enumerate(a_list) ]

print(science_list)   # [('I', 0), ('am', 1), ('a', 2),
                      # ('data', 3), ('scientist', 4)]
```

String handling

Examples with string operations:

```
str = 'Hello World!'
print(str)            # Prints complete string
print(str[0])         # Prints first character of the string
print(str[2:5])       # Prints characters starting from 3rd to 5th
print(str[2:])        # Prints string starting from 3rd character
print(str*2)          # Prints string two times
print(str + "TEST")   # Prints concatenated string
```

Other useful functions include `join`, `split`, `count`, `capitalize`, `strip`, `upper`, `lower`, etc.

Example of string formatting:

```
print("My name is %s and age is %d!" % ('Zara',21))
```

IO handling

Python has two major versions which have some significant differences: Python 2 and Python 3. The default version that we will use is Python 3. One of the differences is the input function, which is called `raw_input` in Python 2 and is renamed to `input` in Python 3.

```
str = input("Enter your input: ")
print("Received input is : ", str)
```

File opening.   To handle files in Python, you can use function `open`. Syntax:

```
file object = open(file_name [, access_mode][, buffering])
```

One of the useful packages for handling tsv and csv files is `csv` library.

Functions

An example how to define a function in Python:

```
def functionname(parameters):
  "function_docstring"
  function_suite
  return [expression]
```

Running your code on `timberlea`

To run the Python code on `timberlea`, you can use the command `python`. The server `timberlea` has both Python versions installed, 2 and 3, which can be run using the commands `python2.7` or `python3`. The command `python` is the same as `python3` command, which can be checked using the command:
  `python -V`
which should produce the output:
  `Python 3.12.2`
or similar, but with clearly version 3 of Python

Python code can be run in two ways, similarly to Perl code. You can either explicitly call Python interpreter with the name of our script, or call the script directly if you included Python interpreter in the first line of the script:

```
python mypscript.py
```

or

```
./mypscript.py
```

where `mypscript.py` looks like:

```
#!/local/bin/python

print("Hello World!")
```

Step 1. Logging in to server `timberlea`

  – Login to the server `timberlea`
    As in previous lab, login to your account on the server `timberlea`.
  – Change directory to `csci4152` or `csci6509`
    Change your directory to `csci4152` or `csci6509`, whichever is your registered course. This directory
    should have been already created in your previous lab.
  – Create the directory `lab5` and change your current directory to `lab5`:
    ```
    mkdir lab5
    cd lab5
    ```

Step 2. Python list, tuple and dictionary example

Create a file called `lab5-list_merge.py`. Type the following code and fill in the missing parts (`<your_code>`).
Create a dictionary `result`, where the keys are the values from `some_list`, and values from `some_tuple`.
Use list comprehension or a standard loop.

```
some_list = ["first_name", "last_name", "age", "occupation"]
some_tuple = ("John", "Holloway", 35, "carpenter")

result = <your_code>

print(result)
# The result should be:
# {'first_name': 'John', 'last_name': 'Holloway', 'age': 35, 'occupation': 'carpenter'}
```

Submit: Submit the program `lab5-list_merge.py` using the `submit-nlp` command.

Step 3. Lexical Analysis: tokenization

Word tokenization.   A sentence or data can be split into words using the method `word_tokenize()`:

```
from nltk.tokenize import sent_tokenize, word_tokenize

data = "All work and no play makes jack a dull boy, all work and no play"
print(word_tokenize(data))
```

This will output:

```
['All', 'work', 'and', 'no', 'play', 'makes', 'jack', 'dull', 'boy', ',',
 'all', 'work', 'and', 'no', 'play']
```

All of them are words except the comma. Special characters are treated as separate tokens.

Sentence tokenization   The same principle can be applied to sentences. Simply change the method `word_tokenize` to `sent_tokenize` We have added two sentences to the variable data:

```
#!/local/bin/python

from nltk.tokenize import sent_tokenize, word_tokenize

data = ("All work and no play makes jack dull boy.\n"+
        "All work and no play makes jack a dull boy.")
print(sent_tokenize(data))
```

Outputs:

```
['All work and no play makes jack dull boy.',
 'All work and no play makes jack a dull boy.']
```

Storing words and sentences in lists.   If you wish to you can store the words and sentences in lists:

```
#!/local/bin/python

from nltk.tokenize import sent_tokenize, word_tokenize

data = ("All work and no play makes jack dull boy.\n"+
```

```
           "All work and no play makes jack a dull boy.")

phrases = sent_tokenize(data)
words = word_tokenize(data)

print(phrases)
print(words)
```

Step 4. Stop-word removal

English text may contain stop-words, such as 'the, 'is, or 'are', which are very frequent functional words that are in some NLP applications removed from the text. We will see now how to use NLTK to remove stop-words from a text. There is no universal list of stop-words for English in NLP research, but the NLTK library contains a list that may be useful for many applications. Now, we will learn how to remove stop-words using the NLTK.

We start with the code from the previous section with tokenized words, and develop the following program named `lab5-stop_word_removal.py`:

```
#!/local/bin/python
from nltk.tokenize import sent_tokenize, word_tokenize
from nltk.corpus import stopwords   # We imported auxiliary corpus
                                    # provided with NLTK

data = ("All work and no play makes jack dull boy.\n"+
        "All work and no play makes jack a dull boy.")

stopWords = set(stopwords.words('english'))  # a set of English
words = word_tokenize(data.lower())          #         stopwords
wordsFiltered = []

for w in words:
    if w not in stopWords:
        wordsFiltered.append(w)

print(len(stopWords))   # Print the number of stopwords
print(stopWords)        # Print the stopwords
print(wordsFiltered)    # Print the filtered text
```

Note: When you run this code the first time, it is possible that you will get a Python error, including the following message at the end:

```
Resource stopwords not found.
Please use the NLTK Downloader to obtain the resource:

  >>> import nltk
  >>> nltk.download('stopwords')
```

You can run these suggested commands in the Python interpreter, or include them in the code, and after the resource is saved in your local account, the error message will disappear. By running these command, the `stopword` resource will be saved in your local account, in the directory `~/nltk_data`.

Submit: Create a file named `lab5-stop_word_removal.py` with the previous code snippet and submit it using the `submit-nlp` command.

Step 5. Stemming

We covered the concept of stemming in class. We can recall that stemming is a process of replacing a word with its stem, which is the main part of the word in a sense, and it is obtained by removing a word suffix. For example, the stem of the word *waiting* is *wait*. NLTK contains an implementation of the most popular stemming algorithm for English—the Porter stemmer.

To write an example of a program using stemming, we start by defining some words:

```
words = ["game","gaming","gamed","games"]
```

We import the Porter stemmer module:

```
from nltk.stem import PorterStemmer
from nltk.tokenize import sent_tokenize, word_tokenize
```

and stem the words in the list as follows, where we put all components together:

```
from nltk.stem import PorterStemmer
from nltk.tokenize import sent_tokenize, word_tokenize

words = ["game","gaming","gamed","games"]
ps = PorterStemmer()

for word in words:
    print(ps.stem(word))
```

You can do word stemming for sentences too; we just need to tokenize them first:

```
from nltk.stem import PorterStemmer
from nltk.tokenize import sent_tokenize, word_tokenize

ps = PorterStemmer()

sentence = "gaming, the gamers play games"
words = word_tokenize(sentence)

for word in words:
    print(word + ":" + ps.stem(word))
```

There are more stemming algorithms, but the Porter stemmer is the most popular.

Step 6. N-grams

In this step, we will see how to use the NTLK module `ngrams` to collect word and character n-grams.

Word n-grams

```
from nltk import ngrams
sentence = "This is my sentence and I want to ngramize it."
n = 6
w_6grams = ngrams(sentence.split(), n)
for grams in w_6grams:
  print(grams)
```

Character n-grams

```
from nltk import ngrams
sentence = "This is my sentence and I want to ngramize it."
n = 6
c_6grams = ngrams(sentence, n)
for grams in c_6grams:
  print(''.join(grams))
```

Step 7. Exploring corpora

Now, we will use the NLTK corpus module to read the corpus `austen-persuasion.txt`, included in the
Gutenberg corpus collection, and answer the following questions:

  – How many total words does this corpus have?
  – How many unique words does this corpus have?
  – What are the counts for the 10 most frequent words?

Before we proceed with answering these questions, we will describe an NLTK built-in class which can help us to
get the answers in a simple way.

FreqDist   When dealing with a classification task, one may ask how can we automatically identify the words of a
text that are most informative about the topic and genre of the text? One method would be to keep a tally for each
vocabulary item. This is known as a *frequency distribution*, and it tells us the frequency of each vocabulary item
in the text. It is a "distribution" because it tells us how the total number of word tokens in the text are distributed
across the vocabulary items. NLTK automates this through `FreqDist`. Example:

```
#!/local/bin/python

from nltk import FreqDist
from nltk.tokenize import word_tokenize

data = ("All work and no play makes jack dull boy.\n"+
        "All work and no play makes jack a dull boy.")
words = word_tokenize(data)

fdist1 = FreqDist(words)

print(fdist1.most_common(2)) # Prints two most common tokens
print(fdist1.hapaxes())      # Prints tokens with frequency 1
```

Type the following code snippet in a file named `lab5-explore_corpus.py` and fill in the comments with the
answers where indicated. In those comments, you will need to answer questions of home many tokens are in the
novel, how many unique tokens, and which is the third most frequent token.

```
# lab5-explore_corpus.py
from nltk.corpus import gutenberg
from nltk import FreqDist

# Count each token in austen-persuasion.txt of the Gutenberg collection
list_of_words = gutenberg.words("austen-persuasion.txt")
fd = FreqDist(list_of_words) # Frequency distribution object
```

```
print("Total number of tokens: " + str(fd.N()))  # <insert_comment_how_many>
print("Number of unique tokens: " + str(fd.B())) # <insert_comment_how_many>
print("Top 10 tokens:")                          # <insert_comment_which_is_3rd>
for token, freq in fd.most_common(10):
  print(token + "\t" + str(freq))
```

To find out more about `FreqDist` refer to `http://www.nltk.org/book/ch01.html` section 3.1.

Submit: Create a file named `lab5-explore_corpus.py` with the previous code snippet and submit it using the `submit-nlp` command.

Step 8. Document Classification

In the previous example we have explored corpus, which, you may have noticed, was imported from `nltk.corpus`. NLTK offers a package of pre-trained, labeled corpora for different purposes. In this section we will do a simple classification task of movie reviews. The corpus is taken from `nltk.corpus.movie_reviews`. The classifier will be `NaiveBayesClassifier`. Type the following code in the file named `movie_rev_classifier.py` with the following code. Run the code 5 times and report the accuracy for the each run. Explain why each time we got different accuracy. Write the comments below the code snippet as a Python comment.

```
#!/local/bin/python

from nltk import FreqDist, NaiveBayesClassifier
from nltk.corpus import movie_reviews
from nltk.classify import accuracy
import random

documents = [(list(movie_reviews.words(fileid)), category)
             for category in movie_reviews.categories()
             for fileid in movie_reviews.fileids(category)]
random.shuffle(documents)      # This line shuffles the order of the documents

all_words = FreqDist(w.lower() for w in movie_reviews.words())
word_features = list(all_words)[:2000]

def document_features(document):
    document_words = set(document)
    features = {}
    for word in word_features:
        features['contains({})'.format(word)] = (word in document_words)
    return features

featuresets = [(document_features(d), c) for (d,c) in documents]
train_set, test_set = featuresets[100:], featuresets[:100] # Split
                                        # data to train and test set
classifier = NaiveBayesClassifier.train(train_set)

print(accuracy(classifier, test_set))

# <answer_area>
# <answer_area>
```

```
# <answer_area>
```

Submit: Create a file named `lab5-movie_rev_classifier.py` with the previous code snippet and submit it using the `submit-nlp` command.

This is the end of Lab 5.