Text Processing with nltk

Python's standard library for text processing is called the natural language toolkit (`nltk`). In this tutorial, we will learn how to pre-process text data using nltk and other built-in Python functions, and then how to build a document-word matrix for analysis. In HW9, you will continue from this point to build tf-idf scores.

In [15]:
```
import string
import nltk
import numpy as np
```

In this tutorial, we will work with the *Universal Declaration of Human Rights* as our corpus. The text file is available with this tutorial on the course website. We will consider each line in the file to be a "document".

In [16]:
```
with open("universal_decl_of_human_rights.txt", "r") as myfile:
    corpus = myfile.read()  # corpus is all the text in the file
    docs = corpus.splitlines()  # docs is a list of all the documents,
                                # with each document being one line of t
he corpus
```

In [17]:
```
print(type(corpus))
print(type(docs))
print(len(docs))
print(docs[0:5])
```

```
<class 'str'>
<class 'list'>
69
['Whereas recognition of the inherent dignity and of the equal and inal
ienable rights of all members of the human family is the foundation of
freedom justice and peace in the world', 'Whereas disregard and contemp
t for human rights have resulted in barbarous acts which have outraged
the conscience of mankind and the advent of a world in which human bein
gs shall enjoy freedom of speech and belief and freedom from fear and w
ant has been proclaimed as the highest aspiration of the common peopl
e', 'Whereas it is essential if man is not to be compelled to have reco
urseas a last resort to rebellion against tyranny and oppression that h
uman rights should be protected by the rule of law', 'Whereas it is ess
ential to promote the development of friendly relations between nation
s', 'Whereas the peoples of the United Nations have in the Charter reaf
firmed their faith in fundamental human rights in the dignity and worth
of the human person and in the equal rights of men and women and have d
etermined to promote social progress and better standards of life in la
rger freedom']
```

Step 1: Tokenization

First, we will break the text into the tokens (n-grams) that we want to consider. In this case, the tokens are words. We can tokenize manually, or using `nltk` , with only subtle differences between the two approaches:

In [18]:
```
# a) tokenize it manually
doc_tokens_0 = [x.split() for x in docs]

# b) use nltk, for more info refer to https://www.nltk.org/index.html
nltk.download('punkt')
doc_tokens = [nltk.word_tokenize(x) for x in docs]

# a) and b) have suttle differences
# specifically, if docs is "x, y"
# a) ['x,', 'y']  b) ['x', ',', 'y']

print(doc_tokens[0])
```

```
['Whereas', 'recognition', 'of', 'the', 'inherent', 'dignity', 'and',
'of', 'the', 'equal', 'and', 'inalienable', 'rights', 'of', 'all', 'mem
bers', 'of', 'the', 'human', 'family', 'is', 'the', 'foundation', 'of',
'freedom', 'justice', 'and', 'peace', 'in', 'the', 'world']

[nltk_data] Downloading package punkt to /Users/cgb/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
```

In this example, we are interested in analyzing the words in the document. Thus, as part of the tokenization process, we will want to remove punctuation. We can use a list comprehension to do this:

In [19]:
```
doc_tokens_no_punc = [[x for x in a_doc if x not in string.punctuation]
for a_doc in doc_tokens]
```

Step 2: Lowercase and Stopword Removal

Next, we need to make all words lowercase, as well as remove the stopwords from analysis. After downloading a standard stopword list, we can use the `.lower()` method in a list comprehension and do it all in one line:

In [20]:
```
nltk.download('stopwords')
from nltk.corpus import stopwords
stop = stopwords.words('english')
```

```
[nltk_data] Downloading package stopwords to /Users/cgb/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
```

In [21]:
```
doc_tokens_clean = [[x.lower() for x in words if x.lower() not in stop]
for words in doc_tokens_no_punc]
print(doc_tokens_clean[0])
```

```
['whereas', 'recognition', 'inherent', 'dignity', 'equal', 'inalienabl
e', 'rights', 'members', 'human', 'family', 'foundation', 'freedom', 'j
ustice', 'peace', 'world']
```

Step 3: Lemmatizing/Stemming

Next, we will want to reduce words down to simpler forms so that different forms of the same word are counted in a single token. There are two ways to do this:

- Stemming reduces inflected words to their word stem (*e.g.,studies, studying* -> *studi*).
- Lemmatization maps words to their dictionary form, representing them as words (*e.g., studies, studying* -> *study*).

Lemmatization is more complex, because we need to tag a word's Part of Speech (POS) to get the right result. Because of this, stemming is often used. But when POS tagging is reasonable, lemmatization is preferred.

In nltk, we have the `WordNetLemmatizer` for lemmatizing and the `PorterStemmer` for stemming:

In [22]:
```
nltk.download('wordnet')
from nltk.stem import WordNetLemmatizer
from nltk.stem import PorterStemmer
stemmer = PorterStemmer()
lemmatizer = WordNetLemmatizer()
doc_tokens_clean_lem = [[lemmatizer.lemmatize(x) for x in words] for wor
ds in doc_tokens_clean]
print(doc_tokens_clean[1])
```

```
['whereas', 'disregard', 'contempt', 'human', 'rights', 'resulted', 'ba
rbarous', 'acts', 'outraged', 'conscience', 'mankind', 'advent', 'worl
d', 'human', 'beings', 'shall', 'enjoy', 'freedom', 'speech', 'belief',
'freedom', 'fear', 'want', 'proclaimed', 'highest', 'aspiration', 'comm
on', 'people']
```

```
[nltk_data] Downloading package wordnet to /Users/cgb/nltk_data...
[nltk_data]   Package wordnet is already up-to-date!
```

In the rest of this tutorial, we will proceed with lemmatizing. But before we do that, here are a few examples which will illustrate the differences between stemming and lemmatizing:

In [23]:
```
stemmer = PorterStemmer()
lemmatizer = WordNetLemmatizer()
#The lemmatizer will assume we want the word lemmatized to a noun unless
we specify the part of speech (POS)
#Changing the POS tag will then change the result we get
def show_words(words):
    for w, pos in words:
        print(f'Word: {w:10}, Stem: {stemmer.stem(w):10}, Lemma: {lemmat
izer.lemmatize(w, pos):10}')
show_words([('stones', 'n'), ('jokes', 'n')])
```

```
Word: stones    , Stem: stone     , Lemma: stone
Word: jokes     , Stem: joke      , Lemma: joke
```

In [24]:
```
show_words([('speak', 'v'), ('speaking', 'v'), ('spoken', 'v')])
```

```
Word: speak     , Stem: speak     , Lemma: speak
Word: speaking  , Stem: speak     , Lemma: speak
Word: spoken    , Stem: spoken    , Lemma: speak
```

In [25]:
```
show_words([('spoke', 'v'), ('spoke', 'n')])
```

```
Word: spoke     , Stem: spoke     , Lemma: speak
Word: spoke     , Stem: spoke     , Lemma: spoke
```

In [26]:
```
show_words([('foot', 'n'), ('feet', 'n'), ('goose', 'n'), ('geese', 'n'
)])
```

```
Word: foot      , Stem: foot      , Lemma: foot
Word: feet      , Stem: feet      , Lemma: foot
Word: goose     , Stem: goos      , Lemma: goose
Word: geese     , Stem: gees      , Lemma: goose
```

In [27]:
```
show_words([('is', 'v'), ('are', 'v'), ('be', 'v')])
```

```
Word: is        , Stem: is        , Lemma: be
Word: are       , Stem: are       , Lemma: be
Word: be        , Stem: be        , Lemma: be
```

Step 4: Building the doc-word matrix

Now that we have the cleaned up text stored in `doc_tokens_clean_lem` , we can proceed to build the document-word matrix. We will investigate two ways of doing this: one which is a more straightforward implementation, and another which leverages `numpy` to get some efficiency improvements. These efficiency gains won't make much of a difference in this reasonably small example, but when we are dealing with a corpus of millions of documents, it certainly will!

a) An intuitive way of building the document-word matrix

In [28]:
```
#First, gather all of the unique words in the corpus into a list
word_list = []
for doc in doc_tokens_clean_lem:
    for word in doc:
        if(not(word in word_list)):
            word_list.append(word)

#Then, construct the bag-of-words representation of each document
doc_word_simple = []
for doc in doc_tokens_clean_lem:
    doc_vec = [0]*len(word_list) #Each document is represented as a vect
or of word occurrences
    for word in doc:
        ind = word_list.index(word)
        doc_vec[ind] += 1 #Increment the corresponding word index
    doc_word_simple.append(doc_vec)
```

In [29]:
```
doc_word_simple[0][:10]
```

Out[29]:
```
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

In [30]:
```
doc_word_simple[2][:10]
```

Out[30]:
```
[1, 0, 0, 0, 0, 0, 1, 0, 1, 0]
```

In [31]:
```
doc_word_simple = np.array(doc_word_simple) #Now we can use numpy operat
ions on the matrix
```

In [32]:
```
doc_word_simple
```

Out[32]:
```
array([[1, 1, 1, ..., 0, 0, 0],
       [1, 0, 0, ..., 0, 0, 0],
       [1, 0, 0, ..., 0, 0, 0],
       ...,
       [0, 1, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 1, 1, 1]])
```

b) A more efficient way using numpy

In [33]:
```
# A few optimizations:
# 1. Create a dictionary of words:indexes which has faster lookup time t
han the list.
# 2. Allocate memory ahead of time via numpy
word_to_ind = {word:ind for ind, word in enumerate(word_list)}
doc_word = np.zeros((len(doc_tokens_clean_lem), len(word_list)))
for doc, doc_vec in zip(doc_tokens_clean_lem, doc_word):
    for word in doc:
        ind = word_to_ind[word]
        doc_vec[ind] += 1

# Check that this produces the same result
np.all(np.isclose(doc_word, doc_word_simple))
```

Out[33]:
```
True
```