Assignment No. 5

1.1 Title: Natural language processing (NLP).

1.2 Problem Definition: Natural language processing (NLP) is the ability of a computer program to understand human language as it is spoken. NLP is a component of artificial intelligence (AI).

1.3 Prerequisite: Install Anaconda Python, Jupyter Notebook, Spyder on Ubuntu 18.04. Add bashrc path.

1.4 Software Requirement: Jupyter Notebook, Spyder on Ubuntu/Windows.

1.5 Hardware Requirement: 2GB RAM, 500 GB HDD

1.6 Objectives: Understand the implementation of the Natural Language Processing model

1.7 Outcomes: After completion of this assignment students can develop and analyze the Natural Language Processing model and will understand the working.

1.8 Theory Concepts:

- **What is Natural Language Processing?**

  Natural language processing (NLP) is the ability of a computer program to understand human language as it is spoken. NLP is a component of artificial intelligence (AI).

- **How natural language processing works:**

  Syntax and semantic analysis are two main techniques used with natural language processing. Syntax is the arrangement of words in a sentence to make grammatical sense. NLP uses syntax to assess meaning from a language based on grammatical rules. Syntax techniques used include parsing (grammatical analysis for a sentence), word segmentation (which divides a large piece of text to units), sentence breaking (which places sentence boundaries in large texts), morphological segmentation (which divides words into groups) and stemming (which divides words with inflection in them to root forms).

Semantics involves the use and meaning behind words. NLP applies algorithms to understand the meaning and structure of sentences. Techniques that NLP uses with semantics include word sense disambiguation (which derives meaning of a word based on context), named entity recognition (which determines words that can be categorized into groups), and natural language generation (which will use a database to determine semantics behind words).

- **Text Analysis Operations using NLTK**

NLTK is a powerful Python package that provides a set of diverse natural languages algorithms. It is free, opensource, easy to use, large community, and well documented. NLTK consists of the most common algorithms such as tokenizing, part-of-speech tagging, stemming, sentiment analysis, topic segmentation, and named entity recognition. NLTK helps the computer to analysis, pre-process, and understand the written text.

- Tokenization

Tokenization is the first step in text analytics. The process of breaking down a text paragraph into smaller chunks such as words or sentence is called Tokenization. Token is a single entity that is building blocks for sentence or paragraph.

- Sentence Tokenization
- Word Tokenization

- Stopwords

Stopwords considered as noise in the text. Text may contain stop words such as is, am, are, this, a, an, the, etc. In NLTK for removing stopwords, you need to create a list of stopwords and filter out your list of tokens from these words.

- Lexicon Normalization

Lexicon normalization considers another type of noise in the text. For example, connection, connected, connecting word reduce to a common

word "connect". It reduces derivationally related forms of a word to a common root word.

- Stemming
- Lemmatization

- POS Tagging

The primary target of Part-of-Speech(POS) tagging is to identify the grammatical group of a given word. Whether it is a NOUN, PRONOUN, ADJECTIVE, VERB, ADVERBS, etc. based on the context. POS Tagging looks for relationships within the sentence and assigns a corresponding tag to the word.

- **Benefits of NLP**

NLP hosts benefits such as:

- Improved accuracy and efficiency of documentation.
- The ability to automatically make a readable summary text.
- Useful for personal assistants such as Alexa.
- Allows an organization to use chatbots for customer support.
- Easier to perform sentiment analysis.

- **Challenges associated with NLP**

NLP has not yet been wholly perfected. For example, semantic analysis can still be a challenge for NLP. Other difficulties include the fact that abstract use of language is typically tricky for programs to understand. For instance, NLP does not pick up sarcasm easily. These topics usually require the understanding of the words being used and the context in which the way they are being used. As another example, a sentence can change meaning depending on which word the speaker puts stress on.

NLP is also challenged by the fact that language, and the way people use it, is continually changing.

1.9 Code with output :

1.10 Code Explanation

```
#!/usr/bin/env python
# coding: utf-8

# # NLP (Natural Language Processing)
#

# In[5]:


# !conda install nltk #This installs nltk
import nltk # Imports the library
# nltk.download() #Download the necessary datasets


# ## Get the Data

# We'll be using a dataset from the [UCI datasets]
(https://archive.ics.uci.edu/ml/datasets/SMS+Spam+Collection)

# In[7]:


messages = [line.rstrip() for line in
open('/home/bvcoew/Desktop/SMSSpamCollection')]
print(len(messages))


# In[8]:


import pandas as pd


# We'll use **read_csv** and make note of the **sep** argument,
we can also specify the desired column names by passing in a list
of *names*.

# In[10]:


messages = pd.read_csv('/home/bvcoew/Desktop/SMSSpamCollection',
```

```
sep='\t', names=["label", "message"])
messages.head()



# ## Exploratory Data Analysis

# In[11]:


messages.describe()


# Let's use **groupby** to use describe by label, this way we can
begin to think about the features that separate ham and spam!

# In[12]:


messages.groupby('label').describe()


# In[13]:


messages['length'] = messages['message'].apply(len)
messages.head()


# ### Data Visualization

# In[14]:


import matplotlib.pyplot as plt
import seaborn as sns

get_ipython().run_line_magic('matplotlib', 'inline')


# In[15]:


messages['length'].plot(bins=50, kind='hist')


# In[16]:


messages.length.describe()
```

```
# Woah! 910 characters, let's use masking to find this message:

# In[17]:


#will display iloc[x] 'x' location msg (with length == y ) of
length y
messages[messages['length'] == 40]['message'].iloc[0]


# In[18]:


messages.hist(column='length', by='label', bins=5,figsize=(15,8))


# ## Text Pre-processing

# data is that it is all in text format (strings). The
classification algorithms will need some sort of numerical
feature vector in order to perform the classification task. There
are actually many methods to convert a corpus to a vector format.
The simplest is the the [bag-of-words]
(http://en.wikipedia.org/wiki/Bag-of-words_model) approach, where
each unique word in a text will be represented by one number.
# We will convert the raw messages (sequence of characters) into
vectors (sequences of numbers).
#
# As a first step, let's write a function that will split a
message into its individual words and return a list. We'll also
remove very common words, ('the', 'a', etc..). To do this we will
take advantage of the NLTK library, standard library in Python
for processing text and has a lot of useful features.
#
# Let's create a function that will process the string in the
message column, then we can just use **apply()** in pandas do
process all the text in the DataFrame.
#
# First removing punctuation. We can just take advantage of
Python's built-in **string** library to get a quick list of all
the possible punctuation:

# In[19]:


import string

mess = 'Sample message! Notice: it has punctuation.'

# Check characters to see if they are in punctuation
nopunc = [char for char in mess if char not in
```

```
string.punctuation]

# Join the characters again to form the string.
nopunc = ''.join(nopunc)


# In[20]:


from nltk.corpus import stopwords
stopwords.words('english')[0:10] # Show some stop words


# In[21]:


nopunc.split()


# In[22]:


# Now just remove any stopwords
clean_mess = [word for word in nopunc.split() if word.lower() not
in stopwords.words('english')]


# In[23]:


clean_mess


# Now let's put both of these together in a function to apply it
to our DataFrame later on:

# In[24]:


def text_process(mess):
    """
    Takes in a string of text, then performs the following:
    1. Remove all punctuation
    2. Remove all stopwords
    3. Returns a list of the cleaned text
    """
    # Check characters to see if they are in punctuation
    nopunc = [char for char in mess if char not in
string.punctuation]

    # Join the characters again to form the string.
    nopunc = ''.join(nopunc)
```

```
    # Now just remove any stopwords
    return [word for word in nopunc.split() if word.lower() not
in stopwords.words('english')]


# Here is the original DataFrame again:

# In[25]:


messages.head()


# Now let's "tokenize" these messages. Tokenization is just the
term used to describe the process of converting the normal text
strings in to a list of tokens (words that we actually want).

# In[26]:


# Check to make sure its working
messages['message'].head(5).apply(text_process)


# In[27]:


# Show original dataframe
messages.head()


# ## Vectorization

# Currently, we have the messages as lists of tokens (also known
as [lemmas](http://nlp.stanford.edu/IR-
book/html/htmledition/stemming-and-lemmatization-1.html)) and now
we need to convert each of those messages into a vector the
SciKit Learn's algorithm models can work with.
#
# Now we'll convert each message, represented as a list of tokens
(lemmas) above, into a vector that machine learning models can
understand.
#
# We'll do that in three steps using the bag-of-words model:
#
# 1. Count how many times does a word occur in each message
(Known as term frequency)
#
# 2. Weigh the counts, so that frequent tokens get lower weight
(inverse document frequency)
#
```

```
# 3. Normalize the vectors to unit length, to abstract from the
original text length (L2 norm)
#
# Let's begin the first step:

# Each vector will have as many dimensions as there are unique
words in the SMS corpus.  We will first use SciKit Learn's
**CountVectorizer**. This model will convert a collection of text
documents to a matrix of token counts.
#
# We can imagine this as a 2-Dimensional matrix. Where the 1-
dimension is the entire vocabulary (1 row per word) and the other
dimension are the actual documents, in this case a column per
text message.
#
# For example:
#
# <table border = â€œ1â€œ>
# <tr>
# <th></th> <th>Message 1</th> <th>Message 2</th> <th>...</th>
<th>Message N</th>
# </tr>
# <tr>
# <td><b>Word 1 Count</b></td><td>0</td><td>1</td><td>...</td>
<td>0</td>
# </tr>
# <tr>
# <td><b>Word 2 Count</b></td><td>0</td><td>0</td><td>...</td>
<td>0</td>
# </tr>
# <tr>
# <td><b>...</b></td> <td>1</td><td>2</td><td>...</td><td>0</td>
# </tr>
# <tr>
# <td><b>Word N Count</b></td> <td>0</td><td>1</td><td>...</td>
<td>1</td>
# </tr>
# </table>
#
#
# Since there are so many messages, we can expect a lot of zero
counts for the presence of that word in that document. Because of
this, SciKit Learn will output a [Sparse Matrix]
(https://en.wikipedia.org/wiki/Sparse_matrix).

# In[40]:


from sklearn.feature_extraction.text import CountVectorizer


# There are a lot of arguments and parameters that can be passed
```

```
to the CountVectorizer. In this case we will just specify the
**analyzer** to be our own previously defined function:

# In[4]:


# Might take a while...
bow_transformer =
CountVectorizer(analyzer=text_process).fit(messages['message'])

# Print total number of vocab words
print(len(bow_transformer.vocabulary_))


# Let's take one text message and get its bag-of-words counts as
a vector, putting to use our new `bow_transformer`:

# In[49]:


message4 = messages['message'][4]
print(message4)


# Now let's see its vector representation:

# In[50]:


bow4 = bow_transformer.transform([message4])
print(bow4)
print(bow4.shape)


# This means that there are seven unique words in message number
4 (after removing common stop words). Two of them appear twice,
the rest only once. Let's go ahead and check and confirm which
ones appear twice:

# In[51]:


print(bow_transformer.get_feature_names()[4073])
print(bow_transformer.get_feature_names()[9570])


# Now we can use **.transform** on our Bag-of-Words (bow)
transformed object and transform the entire DataFrame of
messages. Let's go ahead and check out how the bag-of-words
counts for the entire SMS corpus is a large, sparse matrix:

# In[52]:
```

```
messages_bow = bow_transformer.transform(messages['message'])


# In[53]:


print('Shape of Sparse Matrix: ', messages_bow.shape)
print('Amount of Non-Zero occurences: ', messages_bow.nnz)


# In[54]:


sparsity = (100.0 * messages_bow.nnz / (messages_bow.shape[0] *
messages_bow.shape[1]))
print('sparsity: {}'.format(sparsity))


# In[55]:


from sklearn.feature_extraction.text import TfidfTransformer

tfidf_transformer = TfidfTransformer().fit(messages_bow)
tfidf4 = tfidf_transformer.transform(bow4)
print(tfidf4)


# In[56]:


print(tfidf_transformer.idf_[bow_transformer.vocabulary_['u']])
print(tfidf_transformer.idf_[bow_transformer.vocabulary_['univers
ity']])


# To transform the entire bag-of-words corpus into TF-IDF corpus
at once:

# In[57]:


messages_tfidf = tfidf_transformer.transform(messages_bow)
print(messages_tfidf.shape)


# ## Training a model

# In[58]:
```

```
from sklearn.naive_bayes import MultinomialNB
spam_detect_model = MultinomialNB().fit(messages_tfidf,
messages['label'])


# Let's try classifying our single random message and checking
how we do:

# In[59]:


print('predicted:', spam_detect_model.predict(tfidf4)[0])
print('expected:', messages.label[3])


# ## Part 6: Model Evaluation
# Now we want to determine how well our model will do overall on
the entire dataset. Let's begin by getting all the predictions:

# In[60]:


all_predictions = spam_detect_model.predict(messages_tfidf)
print(all_predictions)


# In[61]:


from sklearn.metrics import classification_report
print (classification_report(messages['label'], all_predictions))


# In[62]:


from sklearn.model_selection import train_test_split

msg_train, msg_test, label_train, label_test =
train_test_split(messages['message'], messages['label'],
test_size=0.2)

print(len(msg_train), len(msg_test), len(msg_train) +
len(msg_test))


# The test size is 20% of the entire dataset (1115 messages out
of total 5572), and the training is the rest (4457 out of 5572).
Note the default split would have been 30/70.
#
# ## Creating a Data Pipeline
```

```
#
# Let's run our model again and then predict off the test set. We
will use SciKit Learn's [pipeline](http://scikit-
learn.org/stable/modules/pipeline.html) capabilities to store a
pipeline of workflow. This will allow us to set up all the
transformations that we will do to the data for future use. Let's
see an example of how it works:

# In[63]:


from sklearn.pipeline import Pipeline

pipeline = Pipeline([
    ('bow', CountVectorizer(analyzer=text_process)),  # strings
to token integer counts
    ('tfidf', TfidfTransformer()),  # integer counts to weighted
TF-IDF scores
    ('classifier', MultinomialNB()),  # train on TF-IDF vectors
w/ Naive Bayes classifier
])


# Now we can directly pass message text data and the pipeline
will do our pre-processing for us! We can treat it as a
model/estimator API:

# In[64]:


pipeline.fit(msg_train,label_train)


# In[65]:


predictions = pipeline.predict(msg_test)


# In[66]:


print(classification_report(predictions,label_test))
```

1.11 Conclusion : Thus, after successfully completing this assignment, we were able to understand & implement Natural Language Processing.