SRI VASAVI INSTITUTE OF ENGINEERING & TECHNOLOGY

2021-22 2nd Semester

LABORATORY MASTER MANUAL

of

NATURAL LANGUAGE PROCESSING WITH PYTHON

for

II B.Tech CSE(AI&ML)

Prepared by

A.Pavan Kumar

Associate Professor

DEPARTMENT OF

COMPUTER SCIENCE & ENGINEERING

II Year - II Semester	Skill Oriented Course- II	L	T	P	C
		0	0	4	2
NATURAL LANGUAGE PROCESSING WITH PYTHON					

List of Experiments

1. Demonstrate Noise Removal for any textual data and remove regular expression pattern such as hashtag from textual data.
2. Perform lemmatization and stemming using python library nltk.
3. Demonstrate object standardization such as replace social media slangs from a text.
4. Perform part of speech tagging on any textual data.
5. Implement topic modeling using Latent Dirichlet Allocation (LDA) in python.
6. Demonstrate Term Frequency – Inverse Document Frequency (TF – IDF) using python
7. Demonstrate word embeddings using word2vec.
8. Implement Text classification using naïve bayes classifier and text blob library.
9. Apply support vector machine for text classification.
10. Convert text to vectors (using term frequency) and apply cosine similarity to provide closenessamong two text.
11. Case study 1: Identify the sentiment of tweets

    In this problem, you are provided with tweet data to predict sentiment on electronicproducts of netizens.
12. Case study 2: Detect hate speech in tweets.

    The objective of this task is to detect hate speech in tweets. For the sake of simplicity, we say a tweet contains hate speech if it has a racist or sexist sentiment associated with it.So, the task is to classify racist or sexist tweets from other tweets.

Exercise-1:

Aim:

Demonstrate Noise Removal for any textual data and remove regular expression pattern such as hash tag from textual data.

Theory:

Noise Removal:

Any piece of text which is not relevant to the context of the data and the end-output can be specified as the noise.

For example – language stopwords (commonly used words of a language – is, am, the, of, in etc), URLs or links, social media entities (mentions, hashtags), punctuations and industry specific words. This step deals with removal of all types of noisy entities present in the text.

A general approach for noise removal is to prepare a dictionary of noisy entities, and iterate the text object by tokens (or by words), eliminating those tokens which are present in the noise dictionary.

Another approach is to use the regular expressions while dealing with special patterns of noise.

Program1a:

```
# Sample code to remove noisy words from a text

noise_list = ["is", "a", "this", "..."]

def remove_noise(input_text):

    words = input_text.split()

    noise_free_words = [word for word in words if word not in noise_list]

    noise_free_text = " ".join(noise_free_words)

    return noise_free_text

print(remove_noise("this is a sample text"))
```

Output:

sample text

program1b:

```
# Sample code to remove a regex pattern such as hash tag from textual data.
import re
def remove_regex(input_text, regex_pattern):
    urls = re.finditer(regex_pattern, input_text)
    for i in urls:
        input_text = re.sub(i.group().strip(), '', input_text)
    return input_text

regex_pattern = "#[\w]*"
inp=input("enter the sentence with hashtag:")
output=remove_regex(inp, regex_pattern)
print("the input text is",'\n',inp)
print("after removing hashtag from input text, the text is",'\n',output)
```

Output:

enter the sentence with hashtag:this is #college sviet college and nlp is handled by #faculty pavan kumar

the input text is

this is #college sviet college and nlp is handled by #faculty pavan kumar

after removing hashtag from input text, the text is

this is sviet college and nlp is handled by pavan kumar

Exercise-2:

Aim:

Perform lemmatization and stemming using python library nltk.

Theory:

Stemming:

Stemming may be defined as the process of obtaining a stem from a word by eliminating the affixes from a word.

For example, in the case of the word raining, stemmer would return the root word or stem word rain by removing the affix from raining.

The process of stemming takes place in a series of steps and transforms the word into a shorter word or a word that has a similar meaning to the root word.

Lemmatization:

Lemmatization is a process that maps the various forms of a word (such as appeared, appears) to the canonical or citation form of the word, also known as the lexeme or lemma (e.g., appear).

The WordNetLemmatizer library may be defined as a wrapper around the WordNet corpus, and it makes use of the morphy()function present in WordNetCorpusReader to extract a lemma. If no lemma is extracted, then the word is only returned in its original form. For example, for works, the lemma returned is the singular form, work.

The difference between stemming and lemmatization is the root word. Stem always returns a nonactual root word on the other hand lemmatization return the actual root word.

Program:

```
from nltk.stem import LancasterStemmer

from nltk.stem import WordNetLemmatizer

word = input("enter a word:")

print('Stemming')

stem = LancasterStemmer()

print(stem.stem(word))
```

```
print('Lemmatization')
lem = WordNetLemmatizer()
print(lem.lemmatize(word, "v"))
print(lem.lemmatize(word))
```

Output:

```
enter a word:drilling
Stemming
dril
Lemmatization
drill
drilling
```

Exercise-3:

Aim:

Demonstrate object standardization such as replace social media slangs from a text.

Theory:

Object Standardization:

Text data often contains words or phrases which are not present in any standard lexical dictionaries. These pieces are not recognized by search engines and models.

Some of the examples are – acronyms, hashtags with attached words, and colloquial slangs. With the help of regular expressions and manually prepared data dictionaries, this type of noise can be fixed, the code below uses a dictionary lookup method to replace social media slangs from a text.

Program:

```
lookup_dict = {'rt':'Retweet', 'dm':'direct message', "awsm" : "awesome", "luv" :"love"}

def lookup_words(input_text):
    words = input_text.split()
    new_words = []
    for word in words:
        if word.lower() in lookup_dict:
            word = lookup_dict[word.lower()]
        new_words.append(word)
        new_text = " ".join(new_words)
    return new_text

print(lookup_words("RT this is a retweeted awsm tweet by Pavan Kumar"))
```

Output:

Retweet this is a retweeted awesome tweet by Pavan Kumar

Exercise-4:

Aim:

Perform part of speech tagging on any textual data.

Theory:

Words can be grouped into classes, such as nouns, verbs, adjectives, and adverbs. These classes are known as lexical categories or parts-of-speech. Parts-of-speech are assigned short labels, or tags, such as NN and VB.

The process of automatically assigning parts-of-speech to words in text is called part-of-speech tagging, POS tagging, or just tagging.

The collection of tags used for a particular task is known as a tagset

Program:

```
import nltk

txt_data=input("enter the textual data:")

text1=nltk.word_tokenize(txt_data)

print("part of speech tagging on textual data is")

print(nltk.pos_tag(text1))
```

Output:

```
enter the textual data:I am learning NLP in sviet by pavan kumar

part of speech tagging on textual data is

[('I', 'PRP'), ('am', 'VBP'), ('learning', 'VBG'), ('NLP', 'NNP'), ('in', 'IN'), ('sviet', 'JJ'), ('by', 'IN'), ('pavan', 'NN'), ('kumar', 'NN')]
```

Exercise-5:

Aim:

Implement topic modeling using Latent Dirichlet Allocation (LDA) in python.

Theory:

Topic modeling is a process of automatically identifying the topics present in a text corpus, it derives the hidden patterns among the words in the corpus in an unsupervised manner. Topics are defined as "a repeating pattern of co-occurring terms in a corpus".

When we have a collection of documents for which we do not clearly know the categories, topic models help us to roughly find the categorization. The model treats each document as a mixture of topics, probably with one dominating topic.

For example, let's suppose we have the following sentences:

- Eating fruits as snacks is a healthy habit
- Exercising regularly is an important part of a healthy lifestyle
- Grapefruit and oranges are citrus fruits

A topic model of these sentences may output the following:

Topic A: 40% healthy, 20% fruits, 10% snacks

Topic B: 20% Grapefruit, 20% oranges, 10% citrus

Sentence 1 and 2: 80% Topic A, 20% Topic B

Sentence 3: 100% Topic B

From the output of the model, we can guess that Topic A is about health and Topic B is about fruits. Though these topics are not known apriori, the model outputs corresponding probabilities for words associated with health, exercising, and fruits in the documents.

Topic modeling is an unsupervised learning method. It helps in discovering structures or patterns in documents when we have little or no labels for doing text classification

The most popular algorithm for topic modeling is Latent Dirichlet Allocation (LDA). We will be using the gensim library for the LDA model to find topics in sample texts

Program:

```
doc1 = "Sugar is bad to consume. My sister likes to have sugar, but not my father."

doc2 = "My father spends a lot of time driving my sister around to dance practice."

doc3 = "Doctors suggest that driving may cause increased stress and blood pressure."

doc4 = "Sometimes I feel pressure to perform well at school, but my father never seems to drive my sister to do better."

doc5 = "Health experts say that Sugar is not good for your lifestyle."

doc_complete = [doc1,doc2,doc3,doc4,doc5]


# set of stopwords

from nltk.corpus import stopwords

from nltk.stem import WordNetLemmatizer

import string

stop = set(stopwords.words('english'))

exclude = set(string.punctuation)

lemma = WordNetLemmatizer()


def clean(doc):

    stop_free = " ".join([i for i in doc.lower().split() if i not in stop])

    punc_free = ''.join(ch for ch in stop_free if ch not in exclude)

    normalized = " ".join(lemma.lemmatize(word) for word in punc_free.split())

    return normalized


doc_clean = [clean(doc).split() for doc in doc_complete]
```

```
print('\n\nCleaned Data\n\n')

print(doc_clean)

print('--------------------------')


import gensim

from gensim import corpora


# Creating the term dictionary of our corpus, where every unique term is assigned an index.

dictionary = corpora.Dictionary(doc_clean)


# Converting list of documents (corpus) into Document Term Matrix using dictionary prepared above.

doc_term_matrix = [dictionary.doc2bow(doc) for doc in doc_clean]


# Creating the object for LDA model using gensim library

Lda = gensim.models.ldamodel.LdaModel


# Running and Training LDA model on the document term matrix

ldamodel = Lda(doc_term_matrix, num_topics=3, id2word = dictionary, passes=50)


# Results

print(ldamodel.print_topics())
```

Output:

Cleaned Data

[['sugar', 'bad', 'consume', 'sister', 'like', 'sugar', 'father'], ['father', 'spends', 'lot', 'time', 'driving', 'sister', 'around', 'dance', 'practice'], ['doctor', 'suggest', 'driving', 'may', 'cause', 'increased', 'stress', 'blood', 'pressure'], ['sometimes', 'feel', 'pressure', 'perform', 'well', 'school', 'father', 'never', 'seems', 'drive', 'sister', 'better'], ['health', 'expert', 'say', 'sugar', 'good', 'lifestyle']]

[(0, '0.091*"sugar" + 0.064*"sister" + 0.064*"father" + 0.036*"drive" + 0.036*"perform" + 0.036*"better" + 0.036*"feel" + 0.036*"school" + 0.036*"sometimes" + 0.036*"seems"'), (1, '0.079*"driving" + 0.045*"pressure" + 0.045*"stress" + 0.045*"cause" + 0.045*"doctor" + 0.045*"blood" + 0.045*"may" + 0.045*"increased" + 0.045*"suggest" + 0.045*"around"'), (2, '0.029*"sister" + 0.029*"father" + 0.029*"pressure" + 0.029*"say" + 0.029*"lifestyle" + 0.029*"health" + 0.029*"good" + 0.029*"expert" + 0.029*"consume" + 0.029*"bad"')]

Exercise-6:

Aim:

Demonstrate Term Frequency – Inverse Document Frequency (TF – IDF) using python

Theory:

TF-IDF is an information retrieval and information extraction subtask which aims to express the importance of a word to a document which is part of a colection of documents which we usually name a corpus.

TF-IDF stands for Term Frequency — Inverse Document Frequency and is a statistic that aims to better define how important a word is for a document, while also taking into account the relation to other documents from the same corpus.

This is performed by looking at how many times a word appears into a document while also paying attention to how many times the same word appears in other documents in the corpus.

So then TF-IDF is a score which is applied to every word in every document in our dataset. And for every word, the TF-IDF value increases with every appearance of the word in a document, but is gradually decreased with every appearance in other documents.

The formula that is used to compute the tf-idf for a term t of a document d in a document set is

$$\text{tf-idf}(t, d) = \text{tf}(t, d) * \text{idf}(t),$$

Here the *tf(t,d)* returns is how many times is the term *t* present in document *d* and

the idf(t) is computed as

$$\text{idf}(t) = \log [\, n / \text{df}(t) \,] + 1 \text{ (if smooth\_idf=False)},$$

where n is the total number of documents in the document set and df(t) is the document frequency of t; the document frequency is the number of documents in the document set that contain the term t.

Program:

```
import pandas as pd

from sklearn.feature_extraction.text import TfidfVectorizer
```

```
dataset = [
    "I enjoy reading about Machine Learning and Machine Learning is my PhD subject",
    "I would enjoy a walk in the park",
    "I was reading in the library"
]
tfIdfVectorizer=TfidfVectorizer(use_idf=True, smooth_idf=False)
tfIdf = tfIdfVectorizer.fit_transform(dataset)
df =pd.DataFrame(tfIdf[0].T.todense(), index=tfIdfVectorizer.get_feature_names_out(),
            columns=["TF-IDF"])
df = df.sort_values('TF-IDF', ascending=False)
print (df.head(25))
```

Output:

```
             TF-IDF
machine    0.518179
learning   0.518179
about      0.259090
subject    0.259090
phd        0.259090
and        0.259090
my         0.259090
is         0.259090
reading    0.173515
enjoy      0.173515
library    0.000000
park       0.000000
in         0.000000
the        0.000000
walk       0.000000
was        0.000000
would      0.000000
```

Exercise-7:

Aim:

Demonstrate word embeddings using word2vec.

Theory:

A word embedding is a semantic representation of a word expressed with a vector. It's also common to represent phrases or sentences in the same manner.

We often use it in natural language processing as a machine learning task for vector space modelling. We can use these vectors to measure the similarities between different words as a distance in the vector space, or feed them directly into the machine learning model.

Word2Vec and GloVe are the two popular models to create word embedding of a text.

Word2vec is a popular technique for modelling word similarity by creating word vectors. It's a method that uses neural networks to model word-to-word relationships. Basically, the algorithm takes a large corpus of text as input and produces a vector, known as a context vector, as output.

Word2vec starts with a simple observation: words which occur close together in a document are likely to share semantic similarities. For example, "king" and "queen" are likely to have similar meanings, be near each other in the document, and have related words such as "man" or "woman."

Word2vec is a shallow, three-layer neural network, where the first and last layers form the input and output; the intermediate layer builds latent representations, for the input words to be transformed into the output vector representation.

Program:

```
import gensim

from gensim.models import Word2Vec

sentences = [['data', 'science'], ['vidhya', 'science', 'data', 'analytics'],['machine', 'learning'], ['deep',
'learning']]


# train the model on your corpus

model = Word2Vec(sentences, min_count = 1)
```

```
#print(model)


print(model.wv.similarity('data', 'science'))

print(model.wv.most_similar('learning'))
```

Output:

```
-0.023671674

[('vidhya', 0.016134710982441902), ('science', -0.01083917822688818), ('machine', -
0.02775036171078682), ('data', -0.0523467510938644 4), ('analytics', -0.05987629294395447),
('deep', -0.1116705760359764 1)]
```

Exercise-8:

Aim:

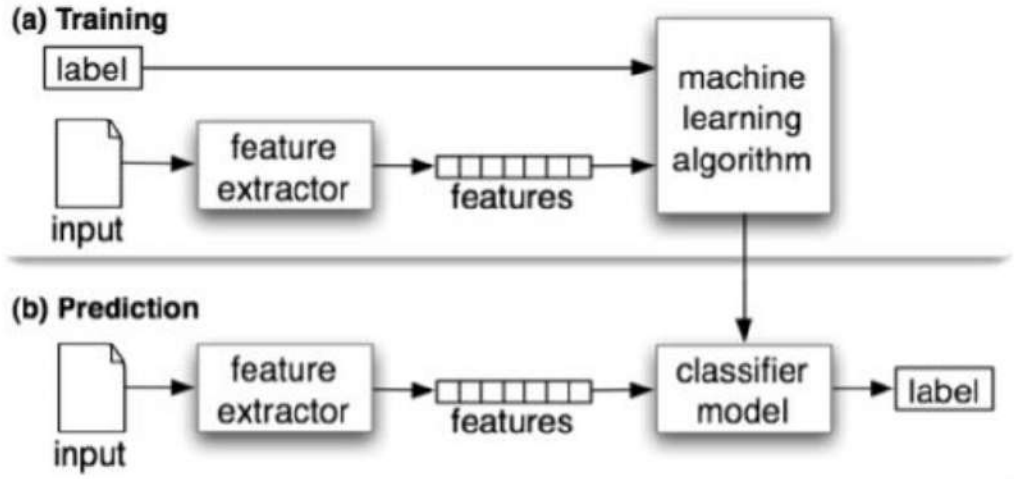
Implement Text classification using naïve bayes classifier and text blob library.

Theory:

Text Classification is an automated process of classification of text into predefined categories. We can classify Emails into spam or non-spam, news articles into different categories like Politics, Stock Market, Sports, sentiment classification and organization of web pages by search engines.

It is really helpful when the amount of data is too large, especially for organizing, information filtering, and storage purposes.

A typical natural language classifier consists of two parts: (a) Training (b) Prediction as shown in image below. Firstly the text input is processes and features are created. The machine learning models then learn these features and is used for predicting against the new text.



The code that uses naive bayes classifier using textblob library is given below.

Program:

```
training = [
('Tom Holland is a terrible spiderman.','pos'),
('a terrible Javert (Russell Crowe) ruined Les Miserables for me...','pos'),
('The Dark Knight Rises is the greatest superhero movie ever!','neg'),
('Fantastic Four should have never been made.','pos'),
('Wes Anderson is my favorite director!','neg'),
('Captain America 2 is pretty awesome.','neg'),
('Let\s pretend "Batman and Robin" never happened..','pos'),
]
testing = [
('Superman was never an interesting character.','pos'),
('Fantastic Mr Fox is an awesome film!','neg'),
('Dragonball Evolution is simply terrible!!','pos')
]

from textblob import classifiers,TextBlob
classifier = classifiers.NaiveBayesClassifier(training)
#print (classifier.accuracy(testing))
blob = TextBlob('the weather is terrible!', classifier=classifier)
print ("class label=",blob.classify())
```

Output:

class label= neg

Exercise-9:

Aim:

Apply support vector machine for text classification

Theory:

Support Vector Machines (SVM) is considered to be a classification approach, It can easily handle multiple continuous and categorical variables.

SVM constructs a hyperplane in multidimensional space to separate different classes. SVM generates optimal hyperplane in an iterative manner, which is used to minimize an error. The core idea of SVM is to find a maximum marginal hyperplane(MMH) that best divides the dataset into classes.

Support Vectors

Support vectors are the data points, which are closest to the hyperplane. These points will define the separating line better by calculating margins. These points are more relevant to the construction of the classifier.

Hyperplane

A hyperplane is a decision plane which separates between a set of objects having different class memberships.

Margin

A margin is a gap between the two lines on the closest class points. This is calculated as the perpendicular distance from the line to support vectors or closest points. If the margin is larger in between the classes, then it is considered a good margin, a smaller margin is a bad margin.

Program:

```
from sklearn.feature_extraction.text import TfidfVectorizer

from sklearn import model_selection,svm

from sklearn.metrics import classification_report
```

```
training_corpus = [
('Tom Holland is a terrible spiderman.','pos'),
('a terrible Javert (Russell Crowe) ruined Les Miserables for me...','pos'),
('The Dark Knight Rises is the greatest superhero movie ever!','neg'),
('Fantastic Four should have never been made.','pos'),
('Wes Anderson is my favorite director!','neg'),
('Captain America 2 is pretty awesome.','neg'),
('Let\s pretend "Batman and Robin" never happened..','pos'),
]
test_corpus = [
('Superman was never an interesting character.','pos'),
('Fantastic Mr Fox is an awesome film!','neg'),
('Dragonball Evolution is simply terrible!!','pos')
]
# preparing data for SVM model
train_data = []
train_labels = []
for row in training_corpus:
    train_data.append(row[0])
    train_labels.append(row[1])

test_data = []
test_labels = []
for row in test_corpus:
    test_data.append(row[0])
```

```
    test_labels.append(row[1])


# Create feature vectors
vectorizer = TfidfVectorizer(min_df=4, max_df=0.9)
# Train the feature vectors
train_vectors = vectorizer.fit_transform(train_data)
# Apply model on test data
test_vectors = vectorizer.transform(test_data)


# Perform classification with SVM, kernel=linear
model = svm.SVC(kernel='poly')
model.fit(train_vectors, train_labels)
prediction = model.predict(test_vectors)


print (classification_report(test_labels, prediction))


print("Accuracy:",metrics.accuracy_score(test_labels, prediction))
```

Output:

```
              precision    recall  f1-score   support

         neg       0.50      1.00      0.67         1
         pos       1.00      0.50      0.67         2

    accuracy                           0.67         3
   macro avg       0.75      0.75      0.67         3
weighted avg       0.83      0.67      0.67         3

Accuracy: 0.6666666666666666
```

Exercise-10:

Aim:

Convert text to vectors (using term frequency) and apply cosine similarity to provide closeness among two text

Theory:

Text Similarity has to determine how the two text documents close to each other. This can be done by checking the words in both documents. If the majority of the words in both documents are similar then that means the documents are similar. And if the words in both the documents are dissimilar then the documents are dissimilar as well.

There are various text similarity metrics exist such as Cosine similarity, Euclidean distance and Jaccard Similarity.

Cosine similarity is one of the metric to measure the text-similarity between two documents irrespective of their size in Natural language Processing. A word is represented into a vector form. The text documents are represented in n-dimensional vector space.

Cosine similarity helps in measuring the cosine of the angles between two vectors. The value of cosine similarity always lies between the range -1 to +1. The value of +1 indicates that the vectors into consideration are perfectly similar. Whereas the value of -1 indicates that the vectors into consideration are perfectly dissimilar or opposite to each other.

The formula for calculating Cosine similarity is given by

$$cos(\theta) = \frac{A \cdot B}{||A||||B||}$$

In the above formula, A and B are two vectors. The numerator denotes the dot product or the scalar product of these vectors and the denominator denotes the magnitude of these vectors. When we divide the dot product by the magnitude, we get the Cosine of the angle between them.

Program:

```
import numpy as np
import math
from collections import Counter


def get_cosine(vec1, vec2):
    common = set(vec1.keys()) & set(vec2.keys())
    numerator = sum([vec1[x] * vec2[x] for x in common])


    sum1 = sum([vec1[x]**2 for x in vec1.keys()])
    sum2 = sum([vec2[x]**2 for x in vec2.keys()])
    denominator = math.sqrt(sum1) * math.sqrt(sum2)


    if not denominator:
        return 0.0
    else:
        return float(numerator) / denominator


def text_to_vector(text):
    words = text.split()
    return Counter(words)


text1 = 'This is an article on analytics vidhya'
text2 = 'article on analytics vidhya is about natural language processing'
```

```
vector1 = text_to_vector(text1)

vector2 = text_to_vector(text2)

print(vector1)

print(vector2)

cosine = get_cosine(vector1, vector2)


print(cosine)
```

Output:

```
Counter({'This': 1, 'is': 1, 'an': 1, 'article': 1, 'on': 1, 'analytics': 1, 'vidhya': 1})

Counter({'article': 1, 'on': 1, 'analytics': 1, 'vidhya': 1, 'is': 1, 'about': 1, 'natural': 1, 'language': 1, 'processing': 1})

0.629940788348712
```

Exercise-11:

Aim:

Identify the sentiment of tweets In this problem, you are provided with tweet data to predict sentiment on electronic products of netizens.

Theory:

Sentiment analysis remains one of the key problems that has seen extensive application of natural language processing. This time around, given the tweets from customers about various tech firms who manufacture and sell mobiles, computers, laptops, etc, the task is to identify if the tweets have a negative sentiment towards such companies or products. Passengers and crew.

Program:

#https://github.com/ayushoriginal/Sentiment-Analysis-Twitter

 # Section 1: Import Library

```
import numpy as np

import pandas as pd

import seaborn as sns

import matplotlib.pyplot as plt


from nltk.corpus import stopwords

from nltk.stem import WordNetLemmatizer

from nltk import word_tokenize

import re

from sklearn.naive_bayes  import GaussianNB

from sklearn.ensemble import RandomForestClassifier

from sklearn.metrics import accuracy_score
```

```
from nltk.sentiment.vader import SentimentIntensityAnalyzer
from sklearn.svm import LinearSVC
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer
from sklearn.linear_model import SGDClassifier
from sklearn.model_selection import cross_val_score, train_test_split
from sklearn.metrics import f1_score, accuracy_score

import warnings
warnings.filterwarnings('ignore')
pd.set_option('display.notebook_repr_html', True)

"""# Section 2: Read Data"""

# reading dataset
train = pd.read_csv('/content/train.csv')
test = pd.read_csv('/content/test.csv')
# concatenating both datasets
df = pd.concat([train, test])
df.head()

train.head()

test.head()

# checking for shape
```

```
print(train.shape, test.shape, df.shape)


df.describe()


# checking dtypes
df.dtypes


df.isnull().sum()


"""# Section 3: EDA and Feature Generation"""


# Checking unique labels
df['label'].value_counts()


# Ploting unique labels
sns.countplot(x='label', data=df)


# Cleaning Raw tweets
def clean_text(text):


    #remove emails
    text = ' '.join([i for i in text.split() if '@' not in i])


    #remove web address
    text = re.sub('http[s]?://\S+', '', text)
```

```
    #Filter to allow only alphabets
    text = re.sub(r'[^a-zA-Z\']', ' ', text)

    #Remove Unicode characters
    text = re.sub(r'[^\x00-\x7F]+', '', text)

    #Convert to lowercase to maintain consistency
    text = text.lower()

    #remove double spaces
    text = re.sub('\s+', ' ',text)

    return text

df["clean_tweet"] = df.tweet.apply(lambda x: clean_text(x))

#defining stop words
STOP_WORDS = ['a', 'about', 'above', 'after', 'again', 'against', 'all', 'also', 'am', 'an', 'and',
            'any', 'are', "aren't", 'as', 'at', 'be', 'because', 'been', 'before', 'being', 'below',
            'between', 'both', 'but', 'by', 'can', "can't", 'cannot', 'com', 'could', "couldn't", 'did',
            "didn't", 'do', 'does', "doesn't", 'doing', "don't", 'down', 'during', 'each', 'else', 'ever',
            'few', 'for', 'from', 'further', 'get', 'had', "hadn't", 'has', "hasn't", 'have', "haven't", 'having',
            'he', "he'd", "he'll", "he's", 'her', 'here', "here's", 'hers', 'herself', 'him', 'himself', 'his',
            'how', "how's", 'however', 'http', 'i', "i'd", "i'll", "i'm", "i've", 'if', 'in', 'into', 'is', "isn't", 'it',
```

"it's", 'its', 'itself', 'just', 'k', "let's", 'like', 'me', 'more', 'most', "mustn't", 'my', 'myself', 'no', 'nor', 'not', 'of', 'off', 'on', 'once', 'only', 'or', 'other', 'otherwise', 'ought', 'our', 'ours', 'ourselves', 'out', 'over', 'own', 'r', 'same', 'shall', "shan't", 'she', "she'd", "she'll", "she's", 'should', "shouldn't", 'since', 'so', 'some', 'such', 'than', 'that', "that's", 'the', 'their', 'theirs', 'them', 'themselves', 'then', 'there', "there's", 'these', 'they', "they'd", "they'll", "they're", "they've", 'this', 'those', 'through', 'to', 'too', 'under', 'until', 'up', 'very', 'was', "wasn't", 'we', "we'd", "we'll", "we're", "we've", 'were', "weren't", 'what', "what's", 'when', "when's", 'where',"where's", 'which', 'while', 'who', "who's", 'whom', 'why', "why's", 'with', "won't", 'would', "wouldn't", 'www', 'you', "you'd", "you'll", "you're", "you've", 'your', 'yours', 'yourself', 'yourselves']

```
# Remove stopwords from all the tweets
df['cleaned_tweet'] = df['clean_tweet'].apply(lambda x: ' '.join([word for word in x.split() if word not in (STOP_WORDS)]))

#Adding New feature length of Tweet
df['word_count']=df.cleaned_tweet.str.split().apply(lambda x: len(x))

import nltk
nltk.download('vader_lexicon')

#Adding New Feature Polarity Score
sid= SentimentIntensityAnalyzer()
sid.polarity_scores(df.iloc[0]['cleaned_tweet'])
df['scores'] =df['tweet'].apply(lambda tweet: sid.polarity_scores(tweet))
df['compound'] =df['scores'].apply(lambda d:d['compound'])
```

```
df['comp_score'] = df['compound'].apply(lambda score: '0' if score>=0 else '1')


# Remove unnecessary

ndf=df.copy()

ndf = ndf.drop(['tweet','clean_tweet','scores','compound','word_count','comp_score'], axis = 1)


ndf.head()


"""# Section 4: Model Building"""


# Seperating Train and Test Set

train_set = ndf[~ndf.label.isnull()]

test_set = ndf[ndf.label.isnull()]


# Shape

print(train_set.shape,test_set.shape)


# Defining X and Y

X = train_set.drop(['label'], axis=1)

y = train_set.label

# Droping target columns

test_set = test_set.drop(['label'], axis=1)


X=X['cleaned_tweet'].astype(str)
```

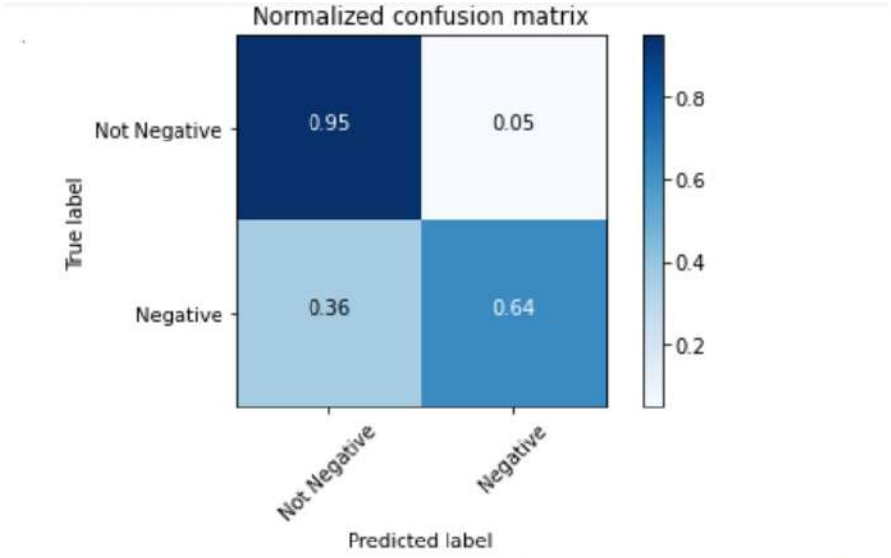
```
#Train test Split

X_train, X_test, y_train, y_test = train_test_split(X,y, test_size = 0.3, random_state = 3)


"""To perform further analysis we need to transform our data into a format that can be processed by our machine learning models.


- **CountVectorizer** does text preprocessing, tokenizing and filtering of stopwords and it builds a dictionary of features and transform documents to feature vectors.

- **TfidfTransformer** transforms the above vector by dividing the number of occurrences of each word in a document by the total number of words in the document. These new features are called tf for Term Frequencies.

"""


print(X_train.shape, X_test.shape, y_train.shape, y_test.shape)


from sklearn.feature_extraction.text import TfidfTransformer


vect = CountVectorizer()

vect.fit(X_train)

X_train_dtm = vect.transform(X_train)

X_test_dtm = vect.transform(X_test)


"""### Random Forest"""


model = RandomForestClassifier(n_estimators=200)

model.fit(X_train_dtm,y_train)

rf = model.predict(X_test_dtm)
```

```
print("Accuracy:",accuracy_score(y_test,rf)*100,"%")


import itertools
from sklearn.metrics import confusion_matrix


def plot_confusion_matrix(cm, classes,
                          normalize=False,
                          title='Confusion matrix',
                          cmap=plt.cm.Blues):
    """
    This function prints and plots the confusion matrix.
    Normalization can be applied by setting `normalize=True`.
    """
    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
        print("Normalized confusion matrix")
    else:
        print('Confusion matrix, without normalization')
    print(cm)

    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
```

```
    plt.yticks(tick_marks, classes)

    fmt = '.2f' if normalize else 'd'
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, format(cm[i, j], fmt),
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
# Compute confusion matrix
cnf_matrix = confusion_matrix(y_test, rf)

np.set_printoptions(precision=2)
class_names = ['Not Negative','Negative']
# Plot normalized confusion matrix
plt.figure()
plot_confusion_matrix(cnf_matrix, classes=class_names, normalize=True,
                      title='Normalized confusion matrix')

plt.show()

from sklearn.metrics import classification_report
```

```
eval_metrics = classification_report(y_test, rf, target_names=class_names)
print(eval_metrics)
```

Final Output:



	precision	recall	f1-score	support
Not Negative	0.88	0.95	0.92	1765
Negative	0.82	0.64	0.71	611
accuracy			0.87	2376
macro avg	0.85	0.79	0.81	2376
weighted avg	0.87	0.87	0.86	2376

Exercise-12:

Aim:

Case study 2: Detect hate speech in tweets. The objective of this task is to detect hate speech in tweets. For the sake of simplicity, we say a tweet contains hate speech if it has a racist or sexist sentiment associated with it. So, the task is to classify racist or sexist tweets from other tweets.

Theory:

Follow the below steps

1. Load Libraries and Data
2. Text PreProcessing and Cleaning
3. Extracting Features from Cleaned Tweets
4. Model Building

Program:

Loading Libraries and Data

```
import re
import pandas as pd
pd.set_option("display.max_colwidth", 200)
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import string
import nltk # for text manipulation
import warnings
warnings.filterwarnings("ignore", category=DeprecationWarning)

%matplotlib inline
```

```
"""Let's read train and test datasets."""

train  = pd.read_csv('/content/train_E6oV3lV.csv')
test = pd.read_csv('/content/test_tweets_anuFYb8.csv')

"""## Text PreProcessing and Cleaning

### Data Inspection

Let's check out a few non racist/sexist tweets.
"""

train[train['label'] == 0].head(10)

"""
Now check out a few racist/sexist tweets.
"""

train[train['label'] == 1].head(10)

"""Let's check dimensions of the train and test dataset."""

train.shape, test.shape

"""Train set has 31,962 tweets and test set has 17,197 tweets.
```

Let's have a glimpse at label-distribution in the train dataset.

"""

train["label"].value_counts()

"""In the train dataset, we have 2,242 (~7%) tweets labeled as racist or sexist, and 29,720 (~93%) tweets labeled as non racist/sexist. So, it is an imbalanced classification challenge.

Now we will check the distribution of length of the tweets, in terms of words, in both train and test data.

"""

length_train = train['tweet'].str.len()

length_test = test['tweet'].str.len()

plt.hist(length_train, bins=20, label="train_tweets")

plt.hist(length_test, bins=20, label="test_tweets")

plt.legend()

plt.show()

"""The tweet-length distribution is more or less the same in both train and test data.

Data Cleaning

"""

```
combi = train.append(test, ignore_index=True)
combi.shape

"""Given below is a user-defined function to remove unwanted text patterns from the tweets."""

def remove_pattern(input_txt, pattern):
    r = re.findall(pattern, input_txt)
    for i in r:
        input_txt = re.sub(i, '', input_txt)

    return input_txt

"""__1. Removing Twitter Handles (@user)__"""

combi['tidy_tweet'] = np.vectorize(remove_pattern)(combi['tweet'], "@[\w]*")
combi.head()

"""__2. Removing Punctuations, Numbers, and Special Characters__"""

combi['tidy_tweet'] = combi['tidy_tweet'].str.replace("[^a-zA-Z#]", " ")
combi.head(10)

"""__3. Removing Short Words__"""

combi['tidy_tweet'] = combi['tidy_tweet'].apply(lambda x: ' '.join([w for w in x.split() if len(w)>3]))
```

"""Let's take another look at the first few rows of the combined dataframe."""

combi.head()

"""__4. Text Normalization__

Here we will use nltk's PorterStemmer() function to normalize the tweets. But before that we will have to tokenize the tweets. Tokens are individual terms or words, and tokenization is the process of splitting a string of text into tokens.

"""

tokenized_tweet = combi['tidy_tweet'].apply(lambda x: x.split()) # tokenizing

tokenized_tweet.head()

"""Now we can normalize the tokenized tweets."""

from nltk.stem.porter import *

stemmer = PorterStemmer()

tokenized_tweet = tokenized_tweet.apply(lambda x: [stemmer.stem(i) for i in x]) # stemming

"""Now let's stitch these tokens back together."""

for i in range(len(tokenized_tweet)):

```
    tokenized_tweet[i] = ' '.join(tokenized_tweet[i])

combi['tidy_tweet'] = tokenized_tweet

"""
## Story Generation and Visualization from Tweets

__A) Understanding the common words used in the tweets: WordCloud__

Now I want to see how well the given sentiments are distributed across the train dataset. One way to accomplish this task is by understanding the common words by plotting wordclouds.

A wordcloud is a visualization wherein the most frequent words appear in large size and the less frequent words appear in smaller sizes.

Let's visualize all the words our data using the wordcloud plot.
"""

all_words = ' '.join([text for text in combi['tidy_tweet']])
from wordcloud import WordCloud
wordcloud = WordCloud(width=800, height=500, random_state=21, max_font_size=110).generate(all_words)

plt.figure(figsize=(10, 7))
plt.imshow(wordcloud, interpolation="bilinear")
plt.axis('off')
```

```
plt.show()
```

"""We can see most of the words are positive or neutral. Words like love, great, friend, life are the most frequent ones. It doesn't give us any idea about the words associated with the racist/sexist tweets. Hence, we will plot separate wordclouds for both the classes (racist/sexist or not) in our train data.

__B) Words in non racist/sexist tweets__

"""

```
normal_words =' '.join([text for text in combi['tidy_tweet'][combi['label'] == 0]])

wordcloud = WordCloud(width=800, height=500, random_state=21, max_font_size=110).generate(normal_words)

plt.figure(figsize=(10, 7))

plt.imshow(wordcloud, interpolation="bilinear")

plt.axis('off')

plt.show()
```

"""Most of the frequent words are compatible with the sentiment, i.e, non-racist/sexists tweets. Similarly, we will plot the word cloud for the other sentiment. Expect to see negative, racist, and sexist terms.

__C) Racist/Sexist Tweets__

"""

```
negative_words = ' '.join([text for text in combi['tidy_tweet'][combi['label'] == 1]])
```

```
wordcloud = WordCloud(width=800, height=500,
random_state=21, max_font_size=110).generate(negative_words)
plt.figure(figsize=(10, 7))
plt.imshow(wordcloud, interpolation="bilinear")
plt.axis('off')
plt.show()

"""As we can clearly see, most of the words have negative connotations. So, it seems we have a pretty good text data to work on. Next we will the hashtags/trends in our twitter data.

__D) Understanding the impact of Hashtags on tweets sentiment__
"""

# function to collect hashtags
def hashtag_extract(x):
    hashtags = []
    # Loop over the words in the tweet
    for i in x:
        ht = re.findall(r"#(\w+)", i)
        hashtags.append(ht)

    return hashtags

# extracting hashtags from non racist/sexist tweets

HT_regular = hashtag_extract(combi['tidy_tweet'][combi['label'] == 0])
```

```
# extracting hashtags from racist/sexist tweets

HT_negative = hashtag_extract(combi['tidy_tweet'][combi['label'] == 1])


# unnesting list

HT_regular = sum(HT_regular,[])

HT_negative = sum(HT_negative,[])


"""Now that we have prepared our lists of hashtags for both the sentiments, we can plot the top 'n' hashtags. So, first let's check the hashtags in the non-racist/sexist tweets.


__Non-Racist/Sexist Tweets__

"""


a = nltk.FreqDist(HT_regular)

d = pd.DataFrame({'Hashtag': list(a.keys()),

                  'Count': list(a.values())})


# selecting top 20 most frequent hashtags

d = d.nlargest(columns="Count", n = 20)

plt.figure(figsize=(16,5))

ax = sns.barplot(data=d, x= "Hashtag", y = "Count")

ax.set(ylabel = 'Count')

plt.show()
```

"""All these hashtags are positive and it makes sense. I am expecting negative terms in the plot of the second list. Let's check the most frequent hashtags appearing in the racist/sexist tweets.

__Racist/Sexist Tweets__

"""

b = nltk.FreqDist(HT_negative)

e = pd.DataFrame({'Hashtag': list(b.keys()), 'Count': list(b.values())})

selecting top 20 most frequent hashtags

e = e.nlargest(columns="Count", n = 20)

plt.figure(figsize=(16,5))

ax = sns.barplot(data=e, x= "Hashtag", y = "Count")

"""As expected, most of the terms are negative with a few neutral terms as well. So, it's not a bad idea to keep these hashtags in our data as they contain useful information. Next, we will try to extract features from the tokenized tweets.

Extracting Features from Cleaned Tweets

"""

from sklearn.feature_extraction.text import TfidfVectorizer, CountVectorizer

import gensim

"""### Bag-of-Words Features"""

```
bow_vectorizer = CountVectorizer(max_df=0.90, min_df=2, max_features=1000, stop_words='english')

bow = bow_vectorizer.fit_transform(combi['tidy_tweet'])

bow.shape


"""### TF-IDF Features"""


tfidf_vectorizer = TfidfVectorizer(max_df=0.90, min_df=2, max_features=1000, stop_words='english')

tfidf = tfidf_vectorizer.fit_transform(combi['tidy_tweet'])

tfidf.shape


"""### Word Embeddings


#### 1. Word2Vec Embeddings

"""


tokenized_tweet = combi['tidy_tweet'].apply(lambda x: x.split()) # tokenizing


model_w2v = gensim.models.Word2Vec(

        tokenized_tweet,

        size=200, # desired no. of features/independent variables

        window=5, # context window size

        min_count=2,

        sg = 1, # 1 for skip-gram model

        hs = 0,
```

```
        negative = 10, # for negative sampling
        workers= 2, # no.of cores
        seed = 34)

model_w2v.train(tokenized_tweet, total_examples= len(combi['tidy_tweet']), epochs=20)
```

"""Let's play a bit with our Word2Vec model and see how does it perform. We will specify a word and the model will pull out the most similar words from the corpus."""

```
model_w2v.wv.most_similar(positive="dinner")

model_w2v.wv.most_similar(positive="trump")

model_w2v['food']

len(model_w2v['food']) #The length of the vector is 200
```

"""__Preparing Vectors for Tweets__

We will use the below function to create a vector for each tweet by taking the average of the vectors of the words present in the tweet.

"""

```
def word_vector(tokens, size):
    vec = np.zeros(size).reshape((1, size))
    count = 0.
```

```
    for word in tokens:
        try:
            vec += model_w2v[word].reshape((1, size))
            count += 1.
        except KeyError: # handling the case where the token is not in vocabulary


            continue
    if count != 0:
        vec /= count
    return vec

"""Preparing word2vec feature set..."""

wordvec_arrays = np.zeros((len(tokenized_tweet), 200))

for i in range(len(tokenized_tweet)):
    wordvec_arrays[i,:] = word_vector(tokenized_tweet[i], 200)

wordvec_df = pd.DataFrame(wordvec_arrays)
wordvec_df.shape

"""Now we have 200 new features, whereas in Bag of Words and TF-IDF we had 1000 features.

#### 2. Doc2Vec Embedding
```

```
Let's load the required libraries.
"""

from tqdm import tqdm
tqdm.pandas(desc="progress-bar")
from gensim.models.doc2vec import LabeledSentence

"""To implement doc2vec, we have to __labelise__ or __tag__ each tokenised tweet with unique IDs. We can do so by using Gensim's *LabeledSentence()* function."""

def add_label(twt):
    output = []
    for i, s in zip(twt.index, twt):
        output.append(LabeledSentence(s, ["tweet_" + str(i)]))
    return output

labeled_tweets = add_label(tokenized_tweet) # label all the tweets

"""Let's have a look at the result."""

labeled_tweets[:6]

"""
Now let's train a __doc2vec__ model.
"""
```

```
model_d2v = gensim.models.Doc2Vec(dm=1, # dm = 1 for 'distributed memory' model
                                  dm_mean=1, # dm = 1 for using mean of the context word vectors
                                  size=200, # no. of desired features
                                  window=5, # width of the context window
                                  negative=7, # if > 0 then negative sampling will be used
                                  min_count=5, # Ignores all words with total frequency lower than 2.
                                  workers=3, # no. of cores
                                  alpha=0.1, # learning rate
                                  seed = 23)

model_d2v.build_vocab([i for i in tqdm(labeled_tweets)])

model_d2v.train(labeled_tweets, total_examples= len(combi['tidy_tweet']), epochs=15)

"""__Preparing doc2vec Feature Set__"""

docvec_arrays = np.zeros((len(tokenized_tweet), 200))

for i in range(len(combi)):
    docvec_arrays[i,:] = model_d2v.docvecs[i].reshape((1,200))

docvec_df = pd.DataFrame(docvec_arrays)
docvec_df.shape
```

"""---

Model Building

We are now done with all the pre-modeling stages required to get the data in the proper form and shape. We will be building models on the datasets with different feature sets prepared in the earlier sections — Bag-of-Words, TF-IDF, word2vec vectors, and doc2vec vectors. We will use the following algorithms to build models:

1. Logistic Regression

1. Logistic Regression

"""

```
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import f1_score

"""#### Bag-of-Words Features"""

train_bow = bow[:31962,:]
test_bow = bow[31962:,:]

# splitting data into training and validation set
xtrain_bow, xvalid_bow, ytrain, yvalid = train_test_split(train_bow, train['label'],
                                                          random_state=42,
```

```
                                    test_size=0.3)

lreg = LogisticRegression()
lreg.fit(xtrain_bow, ytrain) # training the model

prediction = lreg.predict_proba(xvalid_bow) # predicting on the validation set
prediction_int = prediction[:,1] >= 0.3 # if prediction is greater than or equal to 0.3 than 1 else 0
prediction_int = prediction_int.astype(np.int)

f1_score(yvalid, prediction_int) # calculating f1 score

"""Now let's make predictions for the test dataset and create a submission file."""

test_pred = lreg.predict_proba(test_bow)
test_pred_int = test_pred[:,1] >= 0.3
test_pred_int = test_pred_int.astype(np.int)
test['label'] = test_pred_int
submission = test[['id','label']]
submission.to_csv('sub_lreg_bow.csv', index=False) # writing data to a CSV file

"""Public Leaderboard F1 Score: 0.567
```

Sample Output:

The file named sub_lreg_bow.csv will be created containing the classification data. Some of data of the file is given below.

id	label
31963	0
31964	0
31965	0
31966	0
31967	0
31968	0
31969	0
31970	0
31971	0
31972	0
31973	0
31974	0
31975	0
31976	0
31977	0
31978	0
31979	0
31980	0
31981	0
31982	1
31983	0
31984	0
31985	0
31986	0
31987	0
31988	0
31989	1
31990	0
31991	0