Text Classification Workshop using Python - Cheat Sheet

In this workshop, we will do the step-by-step process of text classification discussed in the lecture "Module 3 – Supervised Machine Learning".

> **Data:** Comments/Posts related to Universal Access to Quality Education (UAQTE) program. The dataset was scraped from Facebook, Youtube and X (formerly Twitter) and was manually annotated.
>
> **Goal**: To generate classification models that can predict or automatically label whether a given text or comment is positive(1), negative(0) or neutral(2) towards the UAQTE programs.

Before proceeding with the hands-on activity, make sure that Python, Anaconda and Jupyter notebook are installed in your computer.

Go to https://www.python.org/downloads/ to download the latest python version.

Visit https://www.anaconda.com/download to download Anaconda and for the installation guide.

Software and Tools Requirement:	1. python 3.10.8 or later version 2. Anaconda Navigator 3. Jupyter Notebook

Get Started

1. Open Jupyter Notebook by:
   a. Launching Anaconda Navigator -> start Jupyter Notebook; or
   b. Clicking Start menu -> search for Jupyter Notebook -> right click, "run as administrator".

INSTALLING PACKAGES

The code segment below, installs various python libraries needed for the text classification task using the "pip" command.

```
pip install pandas
pip install seaborn
pip install sklearn
pip install scikit-learn
pip install gensim
```

- **pandas** is a powerful data manipulation library in Python. It provides data structures like DataFrame, which is similar to a table in a database or an Excel spreadsheet. This allows you to easily read, manipulate, and analyze data.
- **seaborn** is a data visualization library based on **matplotlib**. It provides a high-level interface for drawing attractive and informative statistical graphics. It's especially useful for creating complex visualizations with minimal code.
- **scikit-learn** is a popular machine learning library in Python. It provides a wide range of machine learning algorithms, preprocessing, and evaluation tools. It's commonly used for tasks like classification, regression, clustering, and more.
- **gensim** is a library for topic modeling, document indexing, and similar tasks.

IMPORTING LIBRARIES

The first step is importing the necessary libraries for preparing the dataset, extracting features, generating models and measuring the performance of the generated models.

Cell #1:

```
import pandas as pd
import numpy as np

import seaborn as sns
import matplotlib.pyplot as plt

#for text pre-processing
import re, string
import nltk
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from nltk.stem import SnowballStemmer
from nltk.corpus import wordnet
from nltk.stem import WordNetLemmatizer

nltk.download('punkt') #divides a text into list of sentences
nltk.download('averaged_perceptron_tagger') #POS tagger
nltk.download('wordnet')

#for model-building
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.linear_model import SGDClassifier
from sklearn.naive_bayes import MultinomialNB

#performance metrics
from sklearn.metrics import classification_report, f1_score, accuracy_score,
confusion_matrix
#from sklearn.metrics import roc_curve, auc, roc_auc_score
from sklearn import model_selection, preprocessing, linear_model, naive_baye
s, metrics, svm

# bag of words
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.feature_extraction.text import CountVectorizer

#for word embedding
import gensim
from gensim.models import Word2Vec #Word2Vec is mostly used for huge dataset
s
```

LOADING AND EXPLORING THE DATASET

The data set that we will be using for this activity is the Universal Access to Quality Tertiary Education (UAQTE) sentiment dataset built under the e-Participation 2.1 project. The goal is to predict whether a given text or comment is positive (1), negative (0) or neutral (2) towards the UAQTE programs.

In this workshop, you will build a machine learning model that predicts which text or comment are positive, negative or neutral. The `'uaqte_balanced_dataset.csv'` file contains the columns, platform-group, text and label (see image below).

	A	B	C
1	platform-group	text	label
2	Youtube	agree na mas priority yung mga mahihirap at lalong	1
3	Youtube	ang problema ng mga stupedyante din kc nagmama	0
4	Youtube	Ano naman ngayon kung may panuntunang dapat s	0
5	Youtube	mag working student ka nalang	2
6	Youtube	ANTITRINITARIAN sobrang mura na nga 50% pa.	0
7	Youtube	aquino gusto namin ang no or less tuition pero stop	0
8	Youtube	BKIT DI NINYO ISINAMA ANG NGA GURO NA MAARI	0
9	Youtube	Bakit di nyo yan ginawa nun panahon ni PNOY??	0

Cell #2:

Loading Dataset to the DataFrame:

This code reads the `'uaqte_balanced_dataset.csv'` file, loads it into a DataFrame named `df_uaqte`, prints the dimensions of the DataFrame, and then displays the first 10 rows of the data to give the user a quick overview of its contents.

```
# Import social media dataset and load to a dataframe

df_uaqte=pd.read_csv('uaqte_balanced_dataset.csv')
print(df_uaqte.shape)
df_uaqte.head(10)
```

Cell #3:

Class Distribution

This code segment calculates and displays the frequency distribution of unique labels in the **`'label'`** column of the DataFrame **‘`df_uaqte`’**, and then creates a bar plot to visualize this distribution using the ‘**`seaborn`**’ library.

```
# CLASS DISTRIBUTION - check if dataset is balanced or not

# Labels:
# 0 - negative
# 1 - positive
# 2 - neutral

x=df_uaqte['label'].value_counts()
print(x)
sns.barplot(x=x.index, y=x)
```

Cell #4:

Word Count, Character Count and Unique Word Count

This code calculates and prints statistics related to text data in the **`df_uaqte`** DataFrame, grouped by different labels (positive, negative, and neutral). It includes word count, character count and unique word count for each category.

```
#1. WORD-COUNT
print('Word Count:')
df_uaqte['word_count'] = df_uaqte['text'].apply(lambda x: len(str(x).split()
))
print('\tPositive Comment/Text: ', df_uaqte[df_uaqte['label']==1]['word_coun
```

```
t'].mean()) #Positive
print('\tNegative Comment/Text: ', df_uaqte[df_uaqte['label']==0]['word_coun
t'].mean()) #Negative
print('\tNeutral Comment/Text: ', df_uaqte[df_uaqte['label']==2]['word_count
'].mean()) #Neutral

#2. CHARACTER-COUNT
print('\nCharacter Count:')
df_uaqte['char_count'] = df_uaqte['text'].apply(lambda x: len(str(x)))
print('\tPositive Comment/Text: ', df_uaqte[df_uaqte['label']==1]['char_coun
t'].mean()) #Positive
print('\tNegative Comment/Text: ', df_uaqte[df_uaqte['label']==0]['char_coun
t'].mean()) #Negative
print('\tNeutral Comment/Text: ', df_uaqte[df_uaqte['label']==2]['char_count
'].mean()) #Neutral


#3. UNIQUE WORD-COUNT
print('\nUnique Word Count:')
df_uaqte['unique_word_count'] = df_uaqte['text'].apply(lambda x: len(set(str
(x).split())))
print('\tPositive Comment/Text: ', df_uaqte[df_uaqte['label']==1]['unique_wo
rd_count'].mean()) #Positive
print('\tNegative Comment/Text: ', df_uaqte[df_uaqte['label']==0]['unique_wo
rd_count'].mean()) #Negative
print('\tNeutral Comment/Text: ', df_uaqte[df_uaqte['label']==2]['unique_wor
d_count'].mean()) #Neutral
```

Cell #5:

Plotting word count per label

```
#Plotting word-count per label/category

#plot for positive sentiments
fig,(ax1,ax2)=plt.subplots(1,2,figsize=(10,4))
train_words=df_uaqte[df_uaqte['label']==1]['word_count']
ax1.hist(train_words,color='red')
ax1.set_title('Negative')

#plot for negative sentiments
fig,(ax1,ax2)=plt.subplots(1,2,figsize=(10,4))
train_words=df_uaqte[df_uaqte['label']==0]['word_count']
ax1.hist(train_words,color='blue')
ax1.set_title('Negative')

#plot for neutral sentiments
train_words=df_uaqte[df_uaqte['label']==2]['word_count']
ax2.hist(train_words,color='green')
ax2.set_title('Neutral')
fig.suptitle('Words per text')
plt.show()
```

PREPROCESSING

Next cells demonstrate how to preprocess the dataset by removing punctuations & special characters, cleaning texts, removing stop words, and applying lemmatization

1. **Simple Text Cleaning**

- Lowercasing, stripping whitespace, removing HTML tags, replacing punctuation with space, removing extra spaces and tabs, removing digits, removing non-word characters and consolidating whitespace.

Cell #6:

The code below defines **`preprocess()`** function for the abovementioned preprocessing techniques. Then, applies this function on the string named **"`text`".**

```
#1. Common text preprocessing
text = "   This is a message to be cleaned. It may involve some things like:
, ?, :, ''  adjacent spaces and tabs     .  "

#convert to lowercase and remove punctuations and characters and then strip
def preprocess(text):
    text = text.lower() #lowercase text
    text=text.strip()  #get rid of leading/trailing whitespace
    text=re.compile('<.*?>').sub('', text) #Remove HTML tags/markups
    text = re.compile('[%s]' % re.escape(string.punctuation)).sub(' ', text)
#Replace punctuation with space. Careful since punctuation can sometime be u
seful
    text = re.sub('\s+', ' ', text)  #Remove extra space and tabs
    text = re.sub(r'\[[0-9]*\]',' ',text) #[0-9] matches any digit (0 to 100
00...)
    text=re.sub(r'[^\w\s]', '', str(text).lower().strip())
    text = re.sub(r'\d',' ',text) #matches any digit from 0 to 100000..., \D
matches non-digits
    text = re.sub(r'\s+',' ',text) #\s matches any whitespace, \s+ matches m
ultiple whitespace, \S matches non-whitespace

    return text

text=preprocess(text)
print(text)  #text is a string
```

Cell #7:

This block of code is used to download specific resources from the Natural Language Toolkit (NLTK), a popular Python library for natural language processing (NLP). It also downloads the list of stop words in english.

Note: You only need to run these download commands once on your machine. After the resources are downloaded, you can use them in multiple projects without needing to redownload them.

```
import nltk
nltk.download('stopwords')
nltk.download('omw-1.4')
# Get the list of stopwords for a specific language (e.g., English)
stopwords_list = stopwords.words('english')

# Print the list of stopwords
print(stopwords_list)
```

Cell #8:

This defines the list of Tagalog stopwords. You can add or define your own list.

```
# Define a list of common Tagalog stopwords
tagalog_stopwords = [
    'ako', 'alin', 'am', 'amin', 'aming', 'ang', 'ano', 'anumang', 'apat', '
at',
```

```
    'atin', 'ating', 'ay', 'bababa', 'bago', 'bakit', 'bawat', 'bilang', 'da
hil',
    'dalawa', 'dapat', 'din', 'dito', 'doon', 'gagawin', 'gayunman', 'ginaga
wa',
    'ginawa', 'ginawang', 'gumawa', 'gusto', 'habang', 'hanggang', 'hindi',
'huwag',
    'iba', 'ibaba', 'ibabaw', 'ibig', 'ikaw', 'ilagay', 'ilalim', 'ilan', 'i
nyong',
    'isa', 'isang', 'itaas', 'ito', 'iyo', 'iyon', 'iyong', 'ka', 'kahit', '
kailangan',
    'kailanman', 'kami', 'kanila', 'kanilang', 'kanino', 'kanya', 'kanyang',
'kapag',
    'kapwa', 'karamihan', 'katiyakan', 'katulad', 'kaya', 'kaysa', 'ko', 'ko
ng', 'kulang',
    'kumuha', 'kung', 'laban', 'lahat', 'lamang', 'likod', 'lima', 'maaari',
'maaaring',
    'maging', 'mahusay', 'makita', 'marami', 'marapat', 'mga', 'minsan', 'mi
smo', 'mula',
    'muli', 'na', 'nabanggit', 'naging', 'nagkaroon', 'nais', 'nakita', 'nam
in', 'napaka',
    'narito', 'nasaan', 'ng', 'nga', 'ngayon', 'ni', 'nila', 'nilang', 'nito
', 'niya',
    'niyang', 'noon', 'o', 'pa', 'paano', 'pababa', 'paggawa', 'pagitan', 'p
agkakaroon',
    'pagkatapos', 'palabas', 'pamamagitan', 'panahon', 'pangalawa', 'para',
'paraan',
    'pareho', 'pataas', 'pero', 'pumunta', 'pumupunta', 'sa', 'saan', 'sabi'
, 'sabihin',
    'sarili', 'sila', 'sino', 'siya', 'tatlo', 'tayo', 'tulad', 'tungkol', '
una', 'walang',
    'ito', 'iyan'
]

# Print the list of Tagalog stopwords
print(tagalog_stopwords)
```

2. Lexicon-based Text Preprocessing

a. **Stopword removal** - removing insignificant words from English vocabulary using nltk. A few such words are 'i','you','a','the','he','which' etc.

b. Stemming - process of slicing the end or the beginning of words with the intention of removing affixes(prefix/suffix)

c. Lemmatization - process of reducing the word to its base form

Cell #9:

The code below defines **`stopword(), stemming() and lemmatizer()`** functions for the abovementioned lexicon-based preprocessing tasks. Then, applies this function on the string named **"text".**

```
# LEXICON-BASED TEXT PROCESSING EXAMPLES

#1. STOP WORDS REMOVAL
def stopword(string):
    english_stopwords = stopwords.words('english')
    combined_stopwords = english_stopwords + tagalog_stopwords

    words = [word for word in string.split() if word.lower() not in combined
_stopwords]
    return ' '.join(words)

text=stopword(text)
print(text)
```

```
#2. STEMMING

# Initialize the stemmer
snow = SnowballStemmer('english')
def stemming(string):
    a=[snow.stem(i) for i in word_tokenize(string) ]
    return " ".join(a)
text=stemming(text)
print(text)

#3. LEMMATIZATION
# Initialize the lemmatizer
wl = WordNetLemmatizer()

# This is a helper function to map NTLK position tags
# Full list is available here: https://www.ling.upenn.edu/courses/Fall_2003/
ling001/penn_treebank_pos.html
def get_wordnet_pos(tag):
    if tag.startswith('J'):
        return wordnet.ADJ
    elif tag.startswith('V'):
        return wordnet.VERB
    elif tag.startswith('N'):
        return wordnet.NOUN
    elif tag.startswith('R'):
        return wordnet.ADV
    else:
        return wordnet.NOUN

# Tokenize the sentence
def lemmatizer(string):
    word_pos_tags = nltk.pos_tag(word_tokenize(string)) # Get position tags
    a=[wl.lemmatize(tag[0], get_wordnet_pos(tag[1])) for idx, tag in enumera
te(word_pos_tags)] # Map the position tag and lemmatize the word/token
    return " ".join(a)

text = lemmatizer(text)
print(text)
```

FINAL PRE-PROCESSING

Applying all the preprocessing functions defined above to the data frame (**df_uaqte** / **uaqte_balanced_dataset.csv**)

Cell #10:

```
#FINAL PREPROCESSING

def finalpreprocess(string):
    return lemmatizer(stopword(preprocess(string)))
df_uaqte['clean_text'] = df_uaqte['text'].apply(lambda x: finalpreprocess(x)
)

df_uaqte.head(10)
```

FEATURE EXTRACTION

It's difficult to work with text data while building Machine learning models since these models need well-defined numerical data. The process to convert text data into numerical data/vector, is called **vectorization** or in the NLP world, word embedding**.**

For this workshop, we will use Bag-of-Words with TF-IDF and Word2Vec to convert our text data to numerical form which will be used to build the classification model.

Cell #11:

Splitting the dataset using 80:20 ratio. 80% as training set and 20% as test set and tokenize the text data for generating word2vec features

Before generating vectors, first partition the dataset into training set (80%) and test set (20%) using the below-mentioned code.

By splitting the dataset, you have a portion (the *training set*) to train your machine learning model, and a separate portion (the *testing set*) to evaluate its performance. This helps ensure that the model generalizes well to **new, unseen** data.

```
#SPLITTING THE TRAINING DATASET INTO TRAIN AND TEST

X_train, X_val, y_train, y_val = train_test_split(df_uaqte["clean_text"],
                                                  df_uaqte["label"],
                                                  test_size=0.2,
                                                  shuffle=True)

# Word2Vec runs on tokenized sentences
X_train_tok= [nltk.word_tokenize(i) for i in X_train]  #for word2vec
X_val_tok= [nltk.word_tokenize(i) for i in X_val]      #for word2vec

print("DONE SPLITTING AND WORK TOKENIZING.")
```

Extracting features/ vectors using Bag-of-words(with Tf- Idf) and Word2Vec

1. **Term Frequency-Inverse Document Frequencies (tf-Idf)**: An advanced variant of the Bag-of-Words that uses the **term frequency–inverse document frequency** (or Tf-Idf)**.** Basically, the value of a word increases proportionally to count in the document, but it is inversely proportional to the frequency of the word in the corpus

2. ***Word2Vec***: One of the major drawbacks of using **Bag-of-words** techniques is that it can't capture the meaning or relation of the words from vectors. Word2Vec is one of the most popular technique to learn word embeddings using shallow neural network which is capable of capturing context of a word in a document, semantic and syntactic similarity, relation with other words, etc.

Cell #12:

This code sets up a Word2Vec model, tokenizes sentences, and defines a custom transformer class (**`MeanEmbeddingVectorizer`**) to convert sentences into numerical vectors using the word vectors obtained from Word2Vec. This is a crucial step in preparing text data for machine learning models that require numerical input.

```
# create Word2vec model

df_uaqte['clean_text_tok']=[nltk.word_tokenize(i) for i in df_uaqte['clean_t
ext']] #convert preprocessed sentence to tokenized sentence
model = Word2Vec(df_uaqte['clean_text_tok'],min_count=1)  #min_count=1 means
word should be present at least across all documents,
#if min_count=2 means if the word is present less than 2 times across all th
e documents then we shouldn't consider it

w2v = dict(zip(model.wv.index_to_key, model.wv.vectors))  #combination of wo
rd and its vector

#for converting sentence to vectors/numbers from word vectors result by Word
```

```
2Vec
class MeanEmbeddingVectorizer(object):
    def __init__(self, word2vec):
        self.word2vec = word2vec
        # if a text is empty we should return a vector of zeros
        # with the same dimensionality as all the other vectors
        self.dim = len(next(iter(word2vec.values())))

    def fit(self, X, y):
        return self

    def transform(self, X):
        return np.array([
            np.mean([self.word2vec[w] for w in words if w in self.word2vec]
                    or [np.zeros(self.dim)], axis=0)
            for words in X
        ])

print("DONE RUNNING.")
```

Cell #13:

TF-IDF and Word2Vec

This cell prepares the text data for machine learning by converting it into numerical representations. It uses TF-IDF to capture the importance of words in documents and Word2Vec to convert sentences into dense vectors based on word embeddings. The validation data is transformed in a manner consistent with how the training data was processed.

```
#TF-IDF
# Convert x_train to vector since model can only run on numbers and not word
s- Fit and transform
tfidf_vectorizer = TfidfVectorizer(use_idf=True)
X_train_vectors_tfidf = tfidf_vectorizer.fit_transform(X_train) #tfidf runs
on non-tokenized sentences unlike word2vec

# Only transform x_test (not fit and transform)
X_val_vectors_tfidf = tfidf_vectorizer.transform(X_val) #Don't fit() your Tf
idfVectorizer to your test data: it will

#change the word-indexes & weights to match test data. Rather, fit on the tr
aining data, then use the same train-data-
#fit model on the test data, to reflect the fact you're analyzing the test d
ata only based on what was learned without
#it, and the have compatible

#Word2vec
# Fit and transform
modelw = MeanEmbeddingVectorizer(w2v)
X_train_vectors_w2v = modelw.transform(X_train_tok)
X_val_vectors_w2v = modelw.transform(X_val_tok)

print("DONE CREATING VECTORS.")
```

TRAINING MODELS USING ML ALGORITHMS

It's time to train a machine learning model on the vectorized dataset and test it. Now that we have converted the text data to numerical data, we can run ML models on **`X_train_vector_tfidf`** and **`y_train`**. We'll test this model on **`X_test_vectors_tfidf`** to get **`y_predict`** and further evaluate the performance of the model

Cells 13-16 contain codes to: a. generate classification models using different algorithms; b. evaluate its performance on a validation set, and; c. provide a detailed analysis of the model's classification results. The confusion matrix and a heatmap visualization are also generated to aid in performance assessment.

1. Multinomial Logistic Regression with TF-IDF

Multinomial Logistic Regression: a generalized form of logistic regression that extends the binary classification model to handle multiple classes directly. It is a widely used algorithm for multi-class classification tasks.

Cell #14:

Model name: **lr_tfidf**

```
#FITTING THE CLASSIFICATION MODEL using Logistic Regression(tf-idf)
lr_tfidf=LogisticRegression(solver = 'lbfgs', multi_class='multinomial', max
_iter=1000)
lr_tfidf.fit(X_train_vectors_tfidf, y_train)  #model

#Predict y value for test dataset
y_predict = lr_tfidf.predict(X_val_vectors_tfidf)

#Generate confusion matrix
conf_matrix = confusion_matrix (y_val, y_predict)

#Print accuracy score and classification report
print('Accuracy: %s\n' % metrics.accuracy_score(y_predict, y_val))
print(classification_report(y_val,y_predict))
print('Confusion Matrix: \n',conf_matrix)

# Plot confusion matrix as a heatmap
plt.figure(figsize=(8, 6))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues',
            xticklabels=lr_tfidf.classes_, yticklabels=lr_tfidf.classes_)
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix')
plt.show()
```

2. Naïve Bayes with TF-IDF

Naive Bayes: It's a probabilistic classifier that makes use of Bayes' Theorem, a rule that uses probability to make predictions based on prior knowledge of conditions that might be related.

Cell #15:

Model name: **nb_tfidf**

```
#FITTING THE CLASSIFICATION MODEL using Naive Bayes(tf-idf)

nb_tfidf = MultinomialNB()
nb_tfidf.fit(X_train_vectors_tfidf, y_train)  #model

#Predict y value for test dataset
y_predict = nb_tfidf.predict(X_val_vectors_tfidf)

#Generate confusion matrix
conf_matrix = confusion_matrix (y_val, y_predict)
```

```
#Print accuracy score and classification report
print('Accuracy: %s\n' % metrics.accuracy_score(y_predict, y_val))
print(classification_report(y_val,y_predict))
print('Confusion Matrix: \n',conf_matrix)

# Plot confusion matrix as a heatmap
plt.figure(figsize=(8, 6))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues',
            xticklabels=nb_tfidf.classes_, yticklabels=nb_tfidf.classes_)
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix')
plt.show()
```

3. Multinomial Logistic Regression with Word2Vec

Cell #16:

Model name: **lr_w2v**

```
#FITTING THE CLASSIFICATION MODEL using Logistic Regression (W2v)
lr_w2v=LogisticRegression(solver = 'lbfgs', multi_class='multinomial', max_it
er=1000)
lr_w2v.fit(X_train_vectors_w2v, y_train)  #model

#Predict y value for test dataset
y_predict = lr_w2v.predict(X_val_vectors_w2v)

#Generate confusion matrix
conf_matrix = confusion_matrix (y_val, y_predict)

#Print accuracy score and classification report
print('Accuracy: %s\n' % metrics.accuracy_score(y_predict, y_val))
print(classification_report(y_val,y_predict))
print('Confusion Matrix: \n',conf_matrix)

# Plot confusion matrix as a heatmap
plt.figure(figsize=(8, 6))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues',
            xticklabels=lr_w2v.classes_, yticklabels=lr_w2v.classes_)
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix')
plt.show()
```

4. Linear Support Vector Machine (SVM) with Word2Vec

Linear Support Vector Machine: a powerful machine learning algorithm commonly used for binary and multi-class classification tasks. It's especially effective in text classification due to its ability to handle high-dimensional data efficiently.

Cell #17:

Model name: **svm_w2v**

```
#FITTING THE CLASSIFICATION MODEL using Linear SVM (W2v)
svm_w2v=sgd = SGDClassifier(loss='hinge', penalty='l2',alpha=1e-3, random_sta
te=123, max_iter=5, tol=None)

svm_w2v.fit(X_train_vectors_w2v, y_train)#model

#Predict y value for test dataset
```

```
y_predict = svm_w2v.predict(X_val_vectors_w2v)

#Generate confusion matrix
conf_matrix = confusion_matrix (y_val, y_predict)

#Print accuracy score and classification report
print('Accuracy: %s\n' % metrics.accuracy_score(y_predict, y_val))
print(classification_report(y_val,y_predict))
print('Confusion Matrix: \n',conf_matrix)

# Plot confusion matrix as a heatmap
plt.figure(figsize=(8, 6))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues',
            xticklabels=svm_w2v.classes_, yticklabels=svm_w2v.classes_)
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix')
plt.show()
```

GENERATE PREDICTIONS USING THE BEST CLASSIFIER MODEL

This code performs predictions on a new dataset (`make_predictions.csv`) using the best generated classification model. It also saves the results, including predicted labels and probabilities, to a CSV file for further analysis or submission.

Cell #18:

In the cell below, **`lr_tfidf`** is the name of the model used.

Based from your training experiments above, *choose the model with the highest accuracy*.

Then, replace **`lr_tfidf`** with your best model. See sample code replacements below:

Code	Replacement
1. `lr_tfidf.predict`	`<best_model_name>.predict(X_vector)`
2. `lr_tfidf.predict_proba(X_vector)`	`<best_model_name>.predict_proba(X_vector)`

```
#Testing it on new dataset with the best model
df_test=pd.read_csv('make_predictions.csv')  #reading the data
df_test['clean_text'] = df_test['text'].apply(lambda x: finalpreprocess(x)) #
preprocess the data
X_test=df_test['clean_text']

X_vector=tfidf_vectorizer.transform(X_test) #converting X_test to vector
y_predict = lr_tfidf.predict(X_vector)      #use the trained model on X_vecto
r
y_prob = lr_tfidf.predict_proba(X_vector)[:,1]
df_test['predict_prob']= y_prob
df_test['label']= y_predict

print(df_test.head())
final=df_test[['text','label']].reset_index(drop=True)
final.to_csv('submission.csv')
```

Now, you have your own classifier model that can automatically label the sentiments or texts related to the implementation of UAQTE.