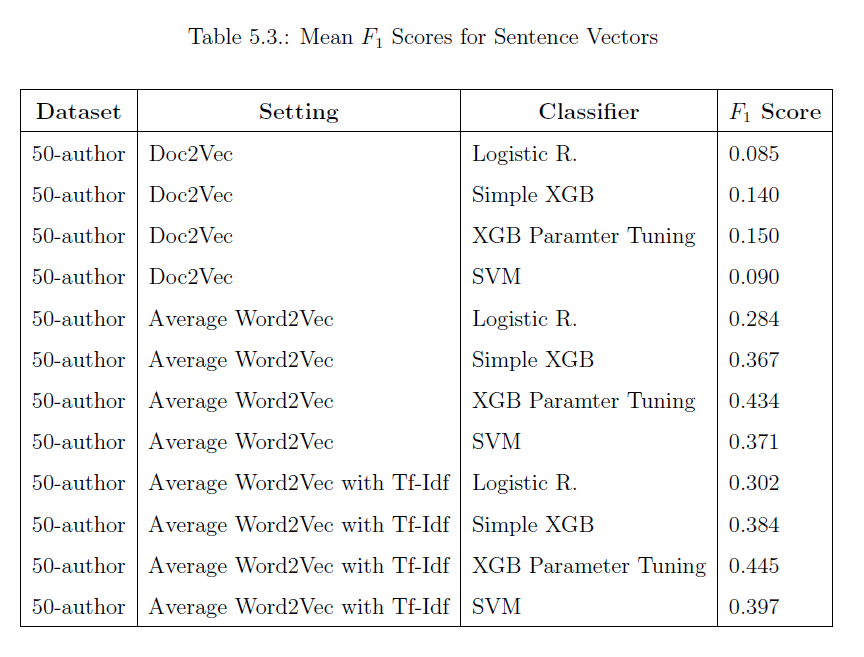
DS7004 Work-based Project Review  
Interim progress report  
(file name: u123456\_DS7004\_draft2.docx)  
27 August 2020

Student name: Chung Hoi CHIU  
Student number: u1720146

Note: I regret that I still cannot write a long draft of the coursework. During the past three weeks, I needed to spend almost all of my available time learning and internalising Python and the algorithm of word2vec. Approximately 20% of the thesis written by Gungor A. (see below) is related to word2vec, a ground-breaking open source tool invented in 2014 for natural language processing (nlp). I can only briefly list below what I have done or found during the past three weeks below.

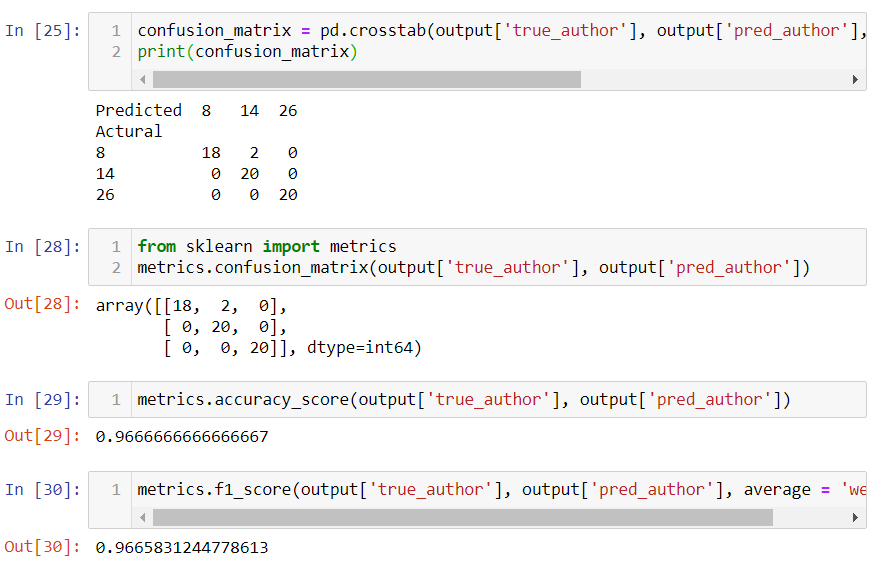
With regard to:  
Gungor A. (2018) Benchmarking authorship attribution techniques using over a thousand books by fifty Victorian Era novelists. Unpublished MSc Thesis, Purdue University, Available at: <https://scholarworks.iupui.edu/handle/1805/15938> (Accessed: 12 July 2020)

1. Approximately 20% of the thesis concern word2vec. However, the thesis indicates that the results obtained from using word2vec are very poor (Please see the below picture), because the author of the thesis used all words of the text data to perform analysis. However, in the field of authorship attribution, those content words contained in the text data should be regarded as ‘noise’ and should be filtered out before feeding the text data to the machine.



2. The above picture shows that the author used many learning machines, such as SVM, SGBoost and logistic regression, to perform the analysis. It is not necessary and is not difficult to do so. Interfaces between a learning machine and the input and output data are usually the same, and both Python and R provide ready-made learning machines.

3. I obtained a Python programme from Kaggle, which 1) uses word2vec to form a 300-dimension vector space with word-vectors embedded; and then 2) uses random forest to perform sentiment classification. I modified the programme and used it to perform an authorship attribution exercise on texts of Charles Dickens (author number allocated: 8), George Eliot (14) and Jane Austen (26). The training set contains 2500 x 1000 words of each of the authors and the test set contains 20 x 1000 words of each of the authors (the programme is appended). The result of running the programme is shown below:



The result is quite good (f1 = 0.97). However, it took approximately 45 minutes for my laptop to run the programme. word2vec is just a specially arranged one hidden layer artificial neural network, and ANN is notoriously very time consuming.

4. word2vec is a clever and useful invention. However, it is not difficult to understand. I even think that it may not be too difficult to use basic Python and common Python packages such as pandas, sklearn, numpy and nltk to write a programme to perform word2vec analysis, although people usually use the ready-made word2vec functions contained in genism. I need to have in-depth knowledge in word2vec if I want to apply it to authorship attribution, an unconventional branch of nlp. Developing a word2vec programme from scratch may be a good method to obtain in-depth knowledge in word2vec.

5. word2vec can be used to perform preliminary social bias studies on text data:

Below are the 20 words within a vector space model produced by using a hotel evaluation corpus and word2vec, cosine distances of them are closest to ‘man’ and ‘woman’ respectively. Do the texts contained in the corpus contain sex-biased elements?

model.most\_similar("man")

[('woman', 0.628918468952179),

 ('lady', 0.5967980623245239),

 ('lad', 0.5614994168281555),

 ('monk', 0.5355309247970581),

 ('soldier', 0.5319280624389648),

 ('millionaire', 0.531794548034668),

 ('chap', 0.5119810104370117),

 ('farmer', 0.5109984278678894),

 ('guy', 0.5098308324813843),

 ('men', 0.5085940361022949)]

Closest to ‘woman’:  
model.most\_similar("woman")

[('lady', 0.6906163692474365),

 ('girl', 0.6630470156669617),

 ('prostitute', 0.6561852693557739),

 ('man', 0.6289184093475342),

 ('widow', 0.6273212432861328),

 ('nun', 0.6217451691627502),

 ('housewife', 0.6163227558135986),

 ('waitress', 0.5760902166366577),

 ('heiress', 0.5679841041564941),

 ('maid', 0.5663273334503174)]

Below are results obtained from the famous 3.39GB GoogleNews-vectors-negative300 (Google300) pretrained word2vec model:

Closest to ‘man’:

model.most\_similar("man")

[('woman', 0.7664012312889099),

 ('boy', 0.6824870109558105),

 ('teenager', 0.6586930751800537),

 ('teenage\_girl', 0.6147903203964233),

 ('girl', 0.5921714305877686),

 ('suspected\_purse\_snatcher', 0.571636438369751),

 ('robber', 0.5585119128227234),

 ('Robbery\_suspect', 0.5584409236907959),

 ('teen\_ager', 0.5549196004867554),

 ('men', 0.5489763021469116)]

Closest to ‘woman’

model.most\_similar("woman")

[('man', 0.7664012312889099),

 ('girl', 0.7494640946388245),

 ('teenage\_girl', 0.7336829900741577),

 ('teenager', 0.631708562374115),

 ('lady', 0.6288785934448242),

 ('teenaged\_girl', 0.6141784191131592),

 ('mother', 0.607630729675293),

 ('policewoman', 0.6069462299346924),

 ('boy', 0.5975908041000366),

 ('Woman', 0.5770983099937439)]

Two other interesting usages of word2vec:

1) Vector equation:

The result below represents a vector equation obtained from the Google300 pretrained word2vec model (Oscar\_Wilde – man + woman = Jane\_Austen):

>>>model.most\_similar(positive=['Oscar\_Wilde', 'woman'], negative=['man'])

>>>[('Jane\_Austen', 0.6260595321655273),

('Noël\_Coward', 0.600019097328186),

('Madame\_Bovary', 0.5772191286087036),

('Charlotte\_Bronte', 0.5637412667274475),

('Somerset\_Maugham', 0.5583583116531372),

('Noel\_Coward', 0.5572813749313354),

('Bernard\_Shaw', 0.5557636022567749),

('DH\_Lawrence', 0.5536727905273438),

('Antonia\_Fraser', 0.5471805334091187),

('An\_Ideal\_Husband', 0.541670024394989)]

Obtained from the word2vec model produced by my programme which is appended here, and the data of 2500 x 1000 words each from Charles Dickens, George Eliot and Jane Austen respectively (king – man + woman = queen):

>>>model.most\_similar(positive=['king', 'woman'], negative=['man']

>>> [('queen', 0.3741656541824341),

('saxon', 0.31361255049705505),

('throne', 0.3131310045719147),

('conqueror', 0.31052806973457336),

('earl', 0.29378655552864075),

('girl', 0.28226879239082336),

('child', 0.2698107063770294),

('dying', 0.26731109619140625),

('reign', 0.26275989413261414),

('pillow', 0.2603800594806671)]

2) Pick up the non-matching word:

>>>model.doesnt\_match("man woman child kitchen".split())

>>> 'kitchen'

6. After learning word2vec, I understand why Gungor A. did not provide labels to the test set of the dataset (38,810 lines x 1000 words) that he contributed to the UCI Machine Learning Repository. The test set is supposed to be used to form a vector space, and labelling each line of the texts (i.e. assigning each line a number that represents the author of the text of the line) is not required to form the vector space. However, if the 38,810 lines are for forming a vector space, calling them a test set is misleading. Furthermore, it is still better to provide labels to the test set since people may use the 38,810 lines of texts for other purposes.

With regard to:  
Segarra S. (2014) Word adjacency networks for authorship attribution: solving Shakespearean controversies. University of Pennsylvania, Available at: <https://cpb-us-e1.wpmucdn.com/blogs.rice.edu/dist/a/9284/files/2018/07/MSc_Thesis_Segarra-zwwuyf.pdf> (Accessed: 12 July 2020).  
(Note: although this thesis has not been published, the content of the first part of it was published as a peer reviewed conference paper (IEEE) in 2013, before completion of the thesis. The content of the thesis was also published in 2016 and 2017 respectively (However, wording, arrangement and style are different), by two peer reviewed journals (‘Shakespeare Quarterly’ and ‘Digital Scholarship in the Humanities’). Segarra S. is a co-author of the said three papers. The other authors are Egan G., a renowned Shakespearean, Riberiro A., the supervisor of the thesis, and Eisen M., a scientist specialised in machine learning and statistics)

1. At the moment, for the reasons I mentioned at my first report, I would not delve into this method. I would rather wait and see.

2. I found that, although the author did not disclose the code and the language used to write the code, he did provide details on how he treated the text of his first corpus (nineteenth century novels). For example, he said that {. ? ! ;} would be treated as stoppers of sentences. But for the second corpus (early modern dramas), he did not provide sufficient details on how he treated the texts. For example, he did not provide details on how he treated early modern punctuation. In early modern era, a comma indicated a pause of one unit, a semi-colon two units, a colon three units and a period indicated a final stop. In addition, an exclamation indicated a shout and a question mark indicated a question. In my opinion, an early modern semi-colon should be treated as a comma and an early modern colon should be treated as a period. Furthermore, the author did not mention how he treated enjambments (break lines) and stichomythias (split lines), two important features of early modern dramas.

Appendix:

##This is a .py script/ Python version 3.8

##Python packages needed to be installed: BeautifulSoup, gensim, logging,

##nltk, numpy, pandas, re, sklearn and time   
##for finding who wrote each of the 60 lines of texts contained in the test set. Charles Dickens, George Eliot or Jane Austen

##!!!The data file Gungor\_2018\_VictorianAuthorAttribution\_data-train.csv

##can be obtained from https://archive.ics.uci.edu/ml/machine-learning-databases/00454/

##input data

import pandas as pd

#!!!use your own path

path\_to\_datafile = '..//..//DS7004//u1720146\_DS7004\_courseworkCodeAndData//preparationWorks//fromDS7003\_Gungor2018VictorianAuthorAttribution\_NGram//Gungor\_2018\_VictorianAuthorAttribution\_data-train.csv'

pathToGungorVict = path\_to\_datafile

gungorVictRow = pd.read\_csv(pathToGungorVict, encoding = 'ISO-8859-1')

##form training data (2500 lines x 3) and test data (20 x 3)

##each line about 1000 words

#Use three authors' data:

#author:8 Charles Dickens total lines: 6914/ 14 George Eliot 2696/ 26 Jane Austen 4441

#each first 2500 lines for training, last 20 lines for testing. Each line has 1000 words

for i in [14, 26, 8]:

allLines = gungorVictRow.loc[gungorVictRow['author'] == i]

lines2500 = allLines.iloc[0:2500]

linesLast20 = allLines.iloc[-20:]

try:

train = train.append(lines2500)

test = test.append(linesLast20)

except:

train = lines2500

test = linesLast20

train = train.sample(frac=1, random\_state=42).reset\_index(drop = True) #7500 lines suffled

test = test.sample(frac=1, random\_state=42).reset\_index(drop = True) #60 lines suffled

## Import various modules for forming a string cleaning function

from bs4 import BeautifulSoup

import re

from nltk.corpus import stopwords

def text\_to\_wordlist( text, remove\_stopwords=False ):

# Function to convert a document to a sequence of words,

# optionally removing stop words. Returns a list of words.

#

# 1. Remove HTML

text = BeautifulSoup(text).get\_text()

#

# 2. Remove non-letters

text = re.sub("[^a-zA-Z]"," ", text)

#

# 3. Convert words to lower case and split them

words = text.lower().split()

#

# 4. Optionally remove stop words (false by default)

if remove\_stopwords: #These three lines will not be used. Pleasesee the second parameter of this function

stops = set(stopwords.words("english"))

words = [w for w in words if not w in stops]

#

# 5. Return a list of words

return(words)

## Download the punkt tokenizer and form a sentence splitting function

import nltk.data

#nltk.download() #no need to use this line again after it has been used once

# Load the punkt tokenizer

tokenizer = nltk.data.load('tokenizers/punkt/english.pickle')

# Define a function to split a text into parsed sentences

def text\_to\_sentences( text, tokenizer, remove\_stopwords=False ):

# Function to split a text into parsed sentences. Returns a

# list of sentences, where each sentence is a list of words

#

# 1. Use the NLTK tokenizer to split the paragraph into sentences

raw\_sentences = tokenizer.tokenize(text.strip())

#

# 2. Loop over each sentence

sentences = []

for raw\_sentence in raw\_sentences:

# If a sentence is empty, skip it

if len(raw\_sentence) > 0:

# Otherwise, call text\_to\_wordlist to get a list of words

sentences.append( text\_to\_wordlist( raw\_sentence, remove\_stopwords )) #defined as false in text\_to\_wordlist

#

# Return the list of sentences (each sentence is a list of words,

# so this returns a list of lists

return sentences

#function for parsing the training set

def parsing\_sentence\_set(text\_df):

sentences = [] # Initialize an empty list of sentences

print("Parsing sentences from training set")

for text in text\_df["text"]:

sentences += text\_to\_sentences(text, tokenizer)

return sentences

##use the functions to form a cleaned unlabelled training set

##for performming unsupervised learning

sentences = parsing\_sentence\_set(train)

## Import the built-in logging module and configure it so that Word2Vec

# creates nice output messages

import logging

logging.basicConfig(format='%(asctime)s : %(levelname)s : %(message)s', level=logging.INFO)

# Set values for the single neural network layer's various parameters

#num\_features = 300 # Word vector dimensionality

#min\_word\_count = 40 # Minimum word count

#num\_workers = 4 # Number of threads to run in parallel

#context = 10 # Context window size

#downsampling = 1e-3 # Downsample setting for frequent words

num\_features = 300 # Word vector dimensionality

min\_word\_count = 5 # Minimum word count

num\_workers = 4 # Number of threads to run in parallel

context = 6 # Context window size

downsampling = 1e-3 # Downsample setting for frequent words

epochs= 20 #number of epochs

## Initialize and train the model (this will take some time)

# need to install gensim's word2vec

from gensim.models import word2vec

def form\_model\_from\_sentences(sentences):

print("Training model...")

model = word2vec.Word2Vec(sentences, workers=num\_workers, \

size=num\_features, min\_count = min\_word\_count,\

window = context, sample = downsampling, iter = epochs)

# If you don't plan to train the model any further, calling

# init\_sims will make the model much more memory-efficient.

model.init\_sims(replace=True)

return model

##form the word2vec model with the training set which will be

##used in the following two methods:

##vector averaging and vector clustering of stop words

##it will take a few minutes

model = form\_model\_from\_sentences(sentences)

##check the model

# king - man + woman = queen?

print(model.most\_similar(positive=['king', 'woman'], negative=['man']))

# \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

##first method: vector averaging of stop words:

import gensim

all\_stopwords = set(gensim.parsing.preprocessing.STOPWORDS)

#be careful: nword and counter must be integers --Chiu

import numpy as np # Make sure that numpy is imported

def makeFeatureVec(words, model, num\_features):

# Function to average all of the word vectors in a given

# paragraph which are stop words

#

# Pre-initialize an empty numpy array (for speed)

featureVec = np.zeros((num\_features,),dtype="float32")

#

nwords = 0

#

# Index2word is a list that contains the names of the words in

# the model's vocabulary. Convert it to a set, for speed

index2word\_set = set(model.wv.index2word)

index2word\_set2 = all\_stopwords

#

# Loop over each word in the text and, if it is in the model's

# vocaublary and is a stop word add its feature vector to the total

for word in words:

if word in index2word\_set and word in index2word\_set2:

nwords = nwords + 1

featureVec = np.add(featureVec, model[word])

#

# Divide the result by the number of words to get the average

if nwords == 0:

nwords = 1 #avoid devided by zero (i.e. no stop word)

featureVec = np.divide(featureVec,nwords)

return featureVec

def getAvgFeatureVecs(texts, model, num\_features):

# Given a set of texts (each one a list of words), calculate

# the average feature vector for each one and return a 2D numpy array

#

# Initialize a counter

counter = 0

#

# Preallocate a 2D numpy array, for speed

textFeatureVecs = np.zeros((len(texts),num\_features),dtype="float32")

#

# Loop through the texts

for text in texts:

#

# Print a status message every 100th text

if counter%100 == 0:

haha = counter; hihi = len(texts)

print(f"Text {haha} of {hihi}") #% (counter, len(texts))

#

# Call the function (defined above) that makes average feature vectors

#textFeatureVecs[counter] = makeFeatureVec(text, model, num\_features)

textFeatureVecs[counter] = makeFeatureVec(text, model, num\_features)

# Increment the counter

counter = counter + 1

return textFeatureVecs

# Calculate average feature vectors for training and testing sets,

# using the functions we defined above.

clean\_train\_texts = []

for text in train["text"]:

#clean\_train\_reviews.append( review\_to\_wordlist( review, \

#remove\_stopwords=True )) #do not remove stop words

clean\_train\_texts.append( text\_to\_wordlist( text ))

trainDataVecs = getAvgFeatureVecs( clean\_train\_texts, model, num\_features )

print("Creating average feature vecs for test texts")

clean\_test\_texts = []

for text in test["text"]:

#clean\_test\_texts.append( text\_to\_wordlist( review, remove\_stopwords=True ))

clean\_test\_texts.append( text\_to\_wordlist( text ))

testDataVecs = getAvgFeatureVecs( clean\_test\_texts, model, num\_features )

# Fit a random forest to the training data, using 100 trees

from sklearn.ensemble import RandomForestClassifier

forest = RandomForestClassifier( n\_estimators = 100 )

print("Fitting a random forest to labeled training data...")

forest = forest.fit( trainDataVecs, train["author"] )

# Test & extract results

result = forest.predict( testDataVecs )

# Write the test results

output = pd.DataFrame( data={"true\_author":test["author"], "pred\_author":result} )

output.to\_csv( "Word2Vec\_AverageVectors.csv", index=False, quoting=3 )

confusion\_matrix = pd.crosstab(output['true\_author'], output['pred\_author'], rownames=['Actural'], colnames=['Predicted'])

print('Confusion matrix:\n', confusion\_matrix)

from sklearn import metrics

print('Accuracy: ', metrics.accuracy\_score(output['true\_author'], output['pred\_author']))

print('f1 score: ', metrics.f1\_score(output['true\_author'], output['pred\_author'], average = 'weighted'))

# \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

##second method: vector clustering of stop words (use KMeans):

from sklearn.cluster import KMeans

import time

start = time.time() # Start time (several to tens of minutes)

# Set "k" (num\_clusters) to be 1/5th of the vocabulary size, or an

# average of 5 words per cluster

word\_vectors = model.wv.syn0

num\_clusters = word\_vectors.shape[0] / 5

# Initalize a k-means object and use it to extract centroids

kmeans\_clustering = KMeans( n\_clusters = int(num\_clusters) )

idx = kmeans\_clustering.fit\_predict( word\_vectors )

# Get the end time and print how long the process took

end = time.time()

elapsed = end - start

print("Time taken for K Means clustering: ", elapsed, "seconds.")

# Create a Word / Index dictionary, mapping each vocabulary word to

#a cluster number

word\_centroid\_map = dict(zip( model.wv.index2word, idx ))

# For the first 10 clusters

for cluster in range(0,10):

#

# Print the cluster number

#print "\nCluster %d" #% cluster

print(f"\nCluster {cluster}")

#

# Find all of the words for that cluster number, and print them out

a\_view = word\_centroid\_map.items()

tuples = list(a\_view)

words = []

for i in range(0,len(word\_centroid\_map.values())):

if( tuples[i][1] == cluster ):

words.append(tuples[i][0])

print(words)

def create\_bag\_of\_centroids( wordlist, word\_centroid\_map ):

#

# The number of clusters is equal to the highest cluster index

# in the word / centroid map

num\_centroids = max( word\_centroid\_map.values() ) + 1

#

# Pre-allocate the bag of centroids vector (for speed)

bag\_of\_centroids = np.zeros( num\_centroids, dtype="float32" )

#

# Loop over the words in the review. If the word is in the vocabulary,

# find which cluster it belongs to, and increment that cluster count

# by one

for word in wordlist:

if word in word\_centroid\_map and word in all\_stopwords:

index = word\_centroid\_map[word]

bag\_of\_centroids[index] += 1

#

# Return the "bag of centroids"

return bag\_of\_centroids

# Pre-allocate an array for the training set bags of centroids (for speed)

train\_centroids = np.zeros( (train["text"].size, int(num\_clusters)), dtype="float32" )

# Transform the training set reviews into bags of centroids

counter = 0

for text in clean\_train\_texts:

train\_centroids[counter] = create\_bag\_of\_centroids( text, word\_centroid\_map )

counter += 1

# Repeat for test reviews

test\_centroids = np.zeros((test["text"].size, int(num\_clusters)), dtype="float32" )

counter = 0

for text in clean\_test\_texts:

test\_centroids[counter] = create\_bag\_of\_centroids( text, word\_centroid\_map )

counter += 1

# This cell take some minutes

# Fit a random forest and extract predictions

forest = RandomForestClassifier(n\_estimators = 100)

# Fitting the forest may take a few minutes

print("Fitting a random forest to labeled training data...")

forest = forest.fit(train\_centroids,train["author"])

result = forest.predict(test\_centroids)

# Write the test results

output = pd.DataFrame(data={"true\_author":test["author"], "pred\_author":result})

output.to\_csv( "BagOfCentroidsAuthor.csv", index=False, quoting=3 )

confusion\_matrix = pd.crosstab(output['true\_author'], output['pred\_author'], rownames=['Actural'], colnames=['Predicted'])

print('Confusion matrix:\n', confusion\_matrix)

from sklearn import metrics

print('Accuracy: ', metrics.accuracy\_score(output['true\_author'], output['pred\_author']))

print('f1 score: ', metrics.f1\_score(output['true\_author'], output['pred\_author'], average = 'weighted'))