#####################################

# Week 5.1: Content-Based Filtering #

#####################################

# importing libraries

import pandas as pd

from math import sqrt

import numpy as np

import matplotlib.pyplot as plt

########################

# Importing Movie Data #

########################

import wget

# import zipfile

# downloading data

# url = 'https://s3-api.us-geo.objectstorage.softlayer.net/cf-courses-data/CognitiveClass/ML0101ENv3/labs/moviedataset.zip'

# wget.download(url, 'moviedataset.zip')

# Note: Zipfile module seems to be acting up

# may just have to unzip by hand for now.

# reading and storing movie dataset

movies\_df = pd.read\_csv('ml-latest\movies.csv')

# reading and storing users dataset

ratings\_df = pd.read\_csv('ml-latest\\ratings.csv')

# printing movies dataset

movies\_df.head()

####################

# Parsing out Year #

####################

# extracting date from movies listing

movies\_df['year'] = movies\_df.title.str.extract('(\(\d\d\d\d\))', expand=False)

# removing parentheses

movies\_df['year'] = movies\_df.year.str.extract('(\d\d\d\d)', expand=False)

# removing year from title column

movies\_df['title'] = movies\_df.title.str.replace('(\(\d\d\d\d\))', '')

# applying strip function to trim whitespaces

movies\_df['title'] = movies\_df['title'].apply(lambda x: x.strip())

movies\_df.head()

####################

# Splitting Genres #

####################

# splitting genres on pipe characters

movies\_df['genres'] = movies\_df.genres.str.split('|')

movies\_df.head()

#########################################

# One-Hot Encoding for Categorical Data #

#########################################

# making a copy of the original dataframe, using copy method for dataframes.

moviesWithGenres\_df = movies\_df.copy()

# generating columns for each genre - basically converting long to wide

# assigning genre columns binary value of 1

for index, row in movies\_df.iterrows():

for genre in row['genres']:

moviesWithGenres\_df.at[index, genre] = 1

# populating empty cells w/ NaNs

moviesWithGenres\_df = moviesWithGenres\_df.fillna(0)

moviesWithGenres\_df.head()

################################

# Ranking Dataframe: Dataset 2 #

################################

# new, dataframe 2: supplementary

ratings\_df.head()

# dropping timestamp column from dataframe.

ratings\_df = ratings\_df.drop('timestamp', 1)

ratings\_df.head()

########################################

# Content-Based Recommendation Systems #

########################################

# super interesting

# starting with a list of movies the user liked and rated

# can extend this list using the same format

userInput = [

{'title':'Breakfast Club, The', 'rating':5},

{'title':'Toy Story', 'rating':3.5},

{'title':'Jumanji', 'rating':2},

{'title':"Pulp Fiction", 'rating':5},

{'title':'Akira', 'rating':4.5}

]

# casting user movie ratings to a dataframe

inputMovies = pd.DataFrame(userInput)

inputMovies

##########################################

# Filtering overall data using User data #

##########################################

# searching for users movies in the original dataframe

inputId = movies\_df[movies\_df['title'].isin(inputMovies['title'].tolist())]

# combining original dataframe with overall ratings one

inputMovies = pd.merge(inputId, inputMovies)

# dropping year and genre from combined dataframe

inputMovies = inputMovies.drop('genres', 1).drop('year', 1)

# printing dataframe with id, title, and ratings

inputMovies

# additionally, subsetting wide-genres dataframe to rows w/ user's films

userMovies = moviesWithGenres\_df[moviesWithGenres\_df['movieId'].isin(inputMovies['movieId'].tolist())]

userMovies

###################

# Resetting index #

###################

# resetting index

userMovies = userMovies.reset\_index(drop=True)

# only keeping binary genre columns as features

userGenreTable = userMovies.drop('movieId', 1).drop('title', 1).drop('genres', 1).drop('year', 1)

#############################

# User Preferences Analysis #

#############################

# input: user's movies + binary genre columns + ratings

print(inputMovies['rating'])

# using ratings as weights: running dot product

userProfile = userGenreTable.transpose().dot(inputMovies['rating'])

# and printing weighted table

print(userProfile)

######################

# Binary Genre Table #

######################

# Grabbing binary genre columns

genreTable = moviesWithGenres\_df.set\_index(moviesWithGenres\_df['movieId'])

# dropping extraneous columns

genreTable = genreTable.drop('movieId', 1).drop('title', 1).drop('genres', 1).drop('year', 1)

print(genreTable.head())

# assessing dimensions

print(genreTable.shape)

############################################

# Weighted Genre Table: Using User Profile #

############################################

# multiplying all movie genres by user weights / ratings, and taking weighted average for each column

# bc there's multiple tags, it's possible to develop a more layered understanding

# and subsequently better recommendations

recommendationTable\_df = ((genreTable\*userProfile).sum(axis=1))/(userProfile.sum())

print(recommendationTable\_df.head())

# sorting recs by movie id

recommendationTable\_df = recommendationTable\_df.sort\_values(ascending=False)

print(recommendationTable\_df.head())

# final recommendation table

print(movies\_df.loc[movies\_df['movieId'].isin(recommendationTable\_df.head(20).keys())])

# in order to display plot within window

# plt.show()

#####################################

# Week 5.2: Collaborative Filtering #

#####################################

# importing libraries

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from math import sqrt

########################

# Importing Movie Data #

########################

# reading and storing movie title, and genres dataset

movies\_df = pd.read\_csv('ml-latest\movies.csv')

# reading and storing user ratings dataset

ratings\_df = pd.read\_csv('ml-latest\\ratings.csv')

# printing movies dataset

movies\_df.head()

####################

# Parsing out Year #

####################

# extracting date from movies listing

movies\_df['year'] = movies\_df.title.str.extract('(\(\d\d\d\d\))', expand=False)

# removing parentheses

movies\_df['year'] = movies\_df.year.str.extract('(\d\d\d\d)', expand=False)

# removing year from title column

movies\_df['title'] = movies\_df.title.str.replace('(\(\d\d\d\d\))', '')

# applying strip function to trim whitespaces

movies\_df['title'] = movies\_df['title'].apply(lambda x: x.strip())

# printing data with year column

movies\_df.head()

###################

# Dropping Genres #

###################

# dropping genres column from movie dataframe.

movies\_df = movies\_df.drop('genres', 1)

movies\_df.head()

################################

# Ranking Dataframe: Dataset 2 #

################################

# dataframe 2: user ratings

ratings\_df.head()

# dropping timestamp column from dataframe.

ratings\_df = ratings\_df.drop('timestamp', 1)

ratings\_df.head()

###########################

# Collaborative Filtering #

###########################

# starting with a list of movies the user liked and rated

userInput = [

{'title':'Breakfast Club, The', 'rating':5},

{'title':'Toy Story', 'rating':3.5},

{'title':'Jumanji', 'rating':2},

{'title':"Pulp Fiction", 'rating':5},

{'title':'Akira', 'rating':4.5}

]

# casting user movie ratings to a dataframe

inputMovies = pd.DataFrame(userInput)

inputMovies

##########################################

# Filtering overall data using User data #

##########################################

# filtering out movies on title: original data vs. user's list

inputId = movies\_df[movies\_df['title'].isin(inputMovies['title'].tolist())]

# merging original data id w/ user movies, using title by default

inputMovies = pd.merge(inputId, inputMovies)

# dropping year from combined dataframe

inputMovies = inputMovies.drop('year', 1)

# printing dataframe with id, title, and ratings

inputMovies

###############################

# Grouping Ratings by User ID #

###############################

# input movies = user ratings + movie id -> using that to subset

# overall ratings

userSubset = ratings\_df[ratings\_df['movieId'].isin(inputMovies['movieId'].tolist())]

userSubset.head()

# grouping rows by user id (contained in ratings dataframe)

userSubsetGroup = userSubset.groupby(['userId'])

# checking one of the users

userSubsetGroup.get\_group(1130)

########################################

# Sorting Users' Ratings by Similarity #

########################################

# grouped ratings - not aggregated yet.

userSubsetGroup = sorted(userSubsetGroup, key=lambda x: len(x[1]), reverse=True)

userSubsetGroup[0:3]

# selecting in most closely correlated users

userSubsetGroup = userSubsetGroup[0:100]

##############################################################

# Saving the Pearson Coefficient for All users to Input User #

##############################################################

# creating empty shell

pearsonCorrelationDict = {}

# creating and filling correlations

for name, group in userSubsetGroup:

# sorting input grouped ratings

group = group.sort\_values(by='movieId')

inputMovies = inputMovies.sort\_values(by='movieId')

# grabbing number of users

nRatings = len(group)

# grabbing ratings for in-common movies

temp\_df = inputMovies[inputMovies['movieId'].isin(group['movieId'].tolist())]

# and storing in a list

tempRatingList = temp\_df['rating'].tolist()

# setting user group to a list

tempGroupList = group['rating'].tolist()

# finally, calculating correlation coefficient b/w users

Sxx = sum([i\*\*2 for i in tempRatingList]) - pow(sum(

tempRatingList),2)/float(nRatings)

Syy = sum([i\*\*2 for i in tempGroupList]) - pow(sum(

tempGroupList),2)/float(nRatings)

Sxy = sum( i\*j for i, j in zip(tempRatingList,

tempGroupList)) - sum(tempRatingList)\*sum(

tempGroupList)/float(nRatings)

# setting condition, if denom=0, corr=0

if Sxx != 0 and Syy != 0:

pearsonCorrelationDict[name] = Sxy/sqrt(Sxx\*Syy)

else:

pearsonCorrelationDict[name] = 0

# printing correlation dictionary

pearsonCorrelationDict.items()

# casting dictionary to a dataframe

pearsonDF = pd.DataFrame.from\_dict(pearsonCorrelationDict,

orient='index')

pearsonDF.columns = ['similarityIndex']

pearsonDF['userId'] = pearsonDF.index

pearsonDF.index = range(len(pearsonDF))

print(pearsonDF.head())

#######################

# Capturing Top Users #

#######################

# top 50 similar users

topUsers=pearsonDF.sort\_values(by='similarityIndex',

ascending=False)[0:50]

print(topUsers.head())

# merging similar users (by input movies) to their rated movies

topUsersRating=topUsers.merge(ratings\_df, left\_on='userId',

right\_on='userId', how='inner')

print(topUsersRating.head())

# multiplying similarity index by users overall ratings

topUsersRating['weightedRating'] = topUsersRating['similarityIndex'

]\*topUsersRating['rating']

print(topUsersRating.head())

# grouping users again, by id, this time aggregating to sum

tempTopUsersRating = topUsersRating.groupby('movieId').sum()[[

'similarityIndex', 'weightedRating']]

# assigning to columns

tempTopUsersRating.columns = ['sum\_similarityIndex',

'sum\_weightedRating']

print(tempTopUsersRating.head())

# declaring data frame shell

recommendation\_df = pd.DataFrame()

# casting weighted average to the dataframe

recommendation\_df['weighted average recommendation score'

] = tempTopUsersRating['sum\_weightedRating'

]/tempTopUsersRating['sum\_similarityIndex']

recommendation\_df['movieId'] = tempTopUsersRating.index

print(recommendation\_df.head())

# ready to sort and display results

recommendation\_df = recommendation\_df.sort\_values(

by='weighted average recommendation score', ascending=False)

print(recommendation\_df.head(10))

# casting top 10 recommended movies to list based on user ratings & printing

print(movies\_df.loc[movies\_df['movieId'].isin(

recommendation\_df.head(10)['movieId'].tolist())])

# in order to display plot within window

# plt.show()

#############################

# Week 6.1: Loan Classifier #

#############################

# importing libraries

# import itertools

import numpy as np

import matplotlib.pyplot as plt

from matplotlib.ticker import NullFormatter

import pandas as pd

import matplotlib.ticker as ticker

from sklearn import preprocessing

##############################

# 1: Importing Loans Dataset #

##############################

# importing library

import wget

# downloading, and saving

# url = 'https://s3-api.us-geo.objectstorage.softlayer.net/cf-courses-data/CognitiveClass/ML0101ENv3/labs/loan\_train.csv'

# wget.download(url, 'loan\_train.csv')

# reading in, printing

df = pd.read\_csv("loan\_train.csv")

df.head()

# printing out dimensions

df.shape

###################################

# Converting to date time objects #

###################################

df['due\_date'] = pd.to\_datetime(df['due\_date'])

df['effective\_date'] = pd.to\_datetime(df['effective\_date'])

df.head()

############################################

# 2: Data visualization and pre-processing #

############################################

# sizing up data set - paid / gone to collector

df['loan\_status'].value\_counts()

# visualization packge

import seaborn as sns

#############################################

# diagram 1: repayment vs amount, by gender #

#############################################

# plotting ratio of paid to unpaid as a stacked bar

# split by gender horizontally

# and with amount on the x-axis, debtors on the y-axis

# cutting up x axis, by amount owed, in 10 unit increments

'''

bins = np.linspace(df.Principal.min(), df.Principal.max(), 10)

g = sns.FacetGrid(df, col="Gender", hue="loan\_status",

palette= "Set1", col\_wrap=2)

g.map(plt.hist, 'Principal', bins=bins, ec="k")

g.axes[-1].legend()

plt.show()

'''

# clearly women are a lot more diligent about paying off loans

# % / ratio wise.

##########################################

# diagram 2: repayment vs age, by gender #

##########################################

# cutting up x axis, by age, in 10 unit increments

'''

bins = np.linspace(df.age.min(), df.age.max(), 10)

g = sns.FacetGrid(df, col="Gender", hue="loan\_status",

palette= "Set1", col\_wrap=2)

g.map(plt.hist, 'age', bins=bins, ec="k")

g.axes[-1].legend()

plt.show()

'''

# women at all ages are good about paying off loans

# men in mid 20s who are debtors reprent the largest proportion

# of the whole number of men at that age who've borrowed credit.

# whereas at other ages, unpaid borrowers, represent a smaller proportion,

# though still large relative to women.

##############################################

# 3: Selecting Features to use as Predictors #

##############################################

# new dimension along which to conduct analysis

df['dayofweek'] = df['effective\_date'].dt.dayofweek

# cutting up x axis, by age, in 10 unit increments

'''

bins = np.linspace(df.dayofweek.min(), df.dayofweek.max(), 10)

g = sns.FacetGrid(df, col="Gender", hue="loan\_status",

palette= "Set1", col\_wrap=2)

g.map(plt.hist, 'dayofweek', bins=bins, ec="k")

g.axes[-1].legend()

plt.show()

'''

# people who receive loans in the first half of the week

# are more likely to pay it off!

# creating a binary variable, for week vs. weekend

df['weekend'] = df['dayofweek'].apply(lambda x: 1 if (x>3) else 0)

df.head()

###################################################

# 4: Converting Var types: Categorical to Numeric #

###################################################

# printing out normalized loan payment status by gender

df.groupby(['Gender'])['loan\_status'].value\_counts(

normalize=True)

# converting gender to indicator var

df['Gender'].replace(to\_replace=['male', 'female'], value=[0,1],

inplace=True)

df.head()

#######################

# 5: One Hot Encoding #

#######################

df.groupby(['education'])['loan\_status'].value\_counts(

normalize=True)

# printing select features

df[['Principal', 'terms', 'age', 'Gender', 'education'

]].head()

############################################

# Converting Var Categories to Binary Vars #

############################################

# subsetting

Feature = df[['Principal', 'terms', 'age', 'Gender', 'weekend']]

# binarizing categories

Feature = pd.concat([Feature, pd.get\_dummies(df['education'])],

axis=1)

# dropping outliers

Feature.drop(['Master or Above'], axis=1, inplace=True)

Feature.head()

################################

# 6: Feature Selection, contd. #

################################

# predictor variables

X = Feature

X[0:5]

# target variables

y =df['loan\_status'].values

y[0:5]

####################

# Normalizing Data #

####################

# standardizing data, fitting to replace original dataframe

X = preprocessing.StandardScaler().fit(X).transform(X)

X[0:5]

#####################

# 7: Classification #

#####################

####################################

# model 1: K Nearest Neighbor(KNN) #

####################################

# creating a test train split suited to our problem

from sklearn.model\_selection import train\_test\_split

# test size is 20 percent

X\_train1, X\_test1, y\_train1, y\_test1 = train\_test\_split( X, y, test\_size=0.2, random\_state=4)

print ('Train set:', X\_train1.shape, y\_train1.shape)

print ('Test set:', X\_test1.shape, y\_test1.shape)

# K nearest neighbor (KNN):

from sklearn.neighbors import KNeighborsClassifier

# Training the algorithm, with k = 4

k = 7 # can be varied to improve accuracy

# training model and predicting

neigh = KNeighborsClassifier(n\_neighbors = k).fit(X\_train1, y\_train1)

neigh

# Predicting estimated(y-values) using testset(x-values) as input

yhat1 = neigh.predict(X\_test1)

yhat1[0:5]

# Evaluating model accuracy using inbuilt sklearn functions

from sklearn import metrics

print("Train set Accuracy: ", metrics.accuracy\_score(y\_train1, neigh.predict(X\_train1)))

print("Test set Accuracy: ", metrics.accuracy\_score(y\_test1, yhat1))

# running a loop through to check for most optimal / accurate k

Ks = 10

mean\_acc = np.zeros((Ks-1))

std\_acc = np.zeros((Ks-1))

ConfustionMx = [];

for n in range(1, Ks):

# Training our Model and Predicting

neigh = KNeighborsClassifier(n\_neighbors = n).fit(X\_train1, y\_train1)

yhat1=neigh.predict(X\_test1)

mean\_acc[n-1] = metrics.accuracy\_score(y\_test1, yhat1)

std\_acc[n-1]=np.std(yhat1==y\_test1)/np.sqrt(yhat1.shape[0])

mean\_acc

# plotting model-accuracy for different numbers of neighbors

'''

plt.plot(range(1,Ks), mean\_acc,'g')

plt.fill\_between(range(1,Ks),mean\_acc - 1 \* std\_acc,

mean\_acc + 1 \* std\_acc,

alpha=0.10)

plt.legend(('Accuracy ', '+/- 3xstd'))

plt.ylabel('Accuracy ')

plt.xlabel('Number of Nabors (K)')

plt.tight\_layout()

plt.show()

'''

print( "The best accuracy was with", mean\_acc.max(), "with k=", mean\_acc.argmax()+1)

# k = 7 seems to yield the highest accuracy for our algorithm

##########################

# model 2: Decision Tree #

##########################

from sklearn.tree import DecisionTreeClassifier

# creating a test train split using python package

X\_trainset2, X\_testset2, y\_trainset2, y\_testset2 = train\_test\_split(

X, y, test\_size=0.3, random\_state=3)

# displaying shapes and size of trainsets:

# size means num of cells, wheras shape shows dimension

X\_trainset2.size

X\_trainset2.shape

y\_trainset2.size

y\_trainset2.shape

# 2: display shapes and size of testsets

X\_testset2.size

X\_testset2.shape

y\_testset2.size

y\_testset2.shape

# Modeling our Data:

# creating decision tree object

loan\_classification = DecisionTreeClassifier(criterion="entropy", max\_depth = 4)

loan\_classification

# fitting decision tree classifications to our training data

loan\_classification.fit(X\_trainset2, y\_trainset2)

# Prediction on our test data:

# using decision tree object to predict test data classification

predTree2 = loan\_classification.predict(X\_testset2)

# printing and comparing outcomes

predTree2 [0:5]

y\_testset2 [0:5]

# model performs fairly well, predicts 3/5 values correctly

# Evaluation of the Decision Tree

# 65% accuracy, might want to improve in some ways

print("DecisionTree's Accuracy: ", metrics.accuracy\_score(y\_testset2, predTree2))

# skipping past the visualization portion, and considering

# how we might improve model accuracy

###################################

# model 3: Support Vector Machine #

###################################

# creating a fresh test and training split

X\_train3, X\_test3, y\_train3, y\_test3 = train\_test\_split( X, y, test\_size=0.2, random\_state=4)

# using default rbf for modeling in svm: radial basis function

from sklearn import svm

# declaring and fitting object

# varying kernel type will vary fit

# for instance, setting kernel = 'linear'

clf = svm.SVC(kernel='rbf')

clf.fit(X\_train3, y\_train3)

# predicting outcome values

yhat3 = clf.predict(X\_test3)

yhat3 [0:5]

# choosing different models, and comparing results

# then choosing the best performing model

# Evaluation metrics:

from sklearn.metrics import classification\_report, confusion\_matrix

import itertools

def plot\_confusion\_matrix(cm, classes,

normalize=False,

title='Confusion matrix',

cmap=plt.cm.Blues):

"""

the current function print and plots confusion matrix

normalization can be applied by setting parameter 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')

# computing confusion matrix for current data

cnf\_matrix = confusion\_matrix(y\_test3, yhat3, labels=['PAIDOFF','COLLECTION'])

np.set\_printoptions(precision=2)

print(classification\_report(y\_test3, yhat3))

# plotting our non normalized matrix

plt.figure()

plot\_confusion\_matrix(cnf\_matrix, classes=['PAIDOFF (PAIDOFF)',

'COLLECTION (COLLECTION)'], normalize= False,

title='Confusion matrix')

plt.show()

# using the f1\_score for scoring performance

from sklearn.metrics import f1\_score

print(f1\_score(y\_test3, yhat3, average='weighted'))

# using the jaccard index for scoring performance

from sklearn.metrics import jaccard\_similarity\_score

print(jaccard\_similarity\_score(y\_test3, yhat3))

################################

# model 4: Logistic Regression #

################################

# fresh 20 - 80 split, 4 folds

X\_train4, X\_test4, y\_train4, y\_test4 = train\_test\_split(X, y, test\_size=0.2, random\_state=4)

# Modeling: Logit w/ Scikit Learn, importing libraries

from sklearn.linear\_model import LogisticRegression

from sklearn.metrics import confusion\_matrix

# iteration 1: inverse regularization = .01, solver = liblinear

# varying inverse regularization parameter, and solver type will vary fit.

# fitting regression model to our training dataset

LR = LogisticRegression(C=0.01, solver='liblinear').fit(X\_train4,y\_train4)

LR

# predicting outcome variable of interest

yhat4 = LR.predict(X\_test4)

yhat4

# this returns probabilities of all binary outcomes yhat

yhat\_prob4 = LR.predict\_proba(X\_test4)

yhat\_prob4

# Evaluating our Logistic Regression model

# jaccard index

print(jaccard\_similarity\_score(y\_test4, yhat4))

# so the model performs not so well - scoring a .577,

# constructing a confusion matrix

def plot\_confusion\_matrix(cm, classes,

normalize=False,

title='Confusion matrix',

cmap=plt.cm.Blues):

'''printing and plotting the confusion matrix

can normalize using option, 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)

# creating plot features

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)

# formatting it to our liking

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')

confusion\_matrix(y\_test4, yhat4, labels=['PAIDOFF','COLLECTION'])

# computing confusion matrix, to predict false positives, and false negatives

cnf\_matrix = confusion\_matrix(y\_test4, yhat4, labels=['PAIDOFF','COLLECTION'])

np.set\_printoptions(precision=2)

# plotting non-normalized confusion matrix

plt.figure()

plot\_confusion\_matrix(cnf\_matrix, classes=['PAIDOFF (PAIDOFF)',

'COLLECTION (COLLECTION)'], normalize= False, title='Confusion matrix')

plt.show()

# printing out our vals for comparison

classification\_report(y\_test4, yhat4)

# log loss calculations

from sklearn.metrics import log\_loss

print(log\_loss(y\_test4, yhat\_prob4))

######################################

# 8: Model Evaluation using Test Set #

######################################

# model1

print('model 1')

# using the f1\_score for scoring performance

print(f1\_score(y\_test1, yhat1, average='weighted'))

# using the jaccard index for scoring performance

print(jaccard\_similarity\_score(y\_test1, yhat1))

# model2

print('model 2')

# using the f1\_score for scoring performance

print(f1\_score(y\_testset2, predTree2, average='weighted'))

# using the jaccard index for scoring performance

print(jaccard\_similarity\_score(y\_testset2, predTree2))

# model3

print('model 3')

# using the f1\_score for scoring performance

print(f1\_score(y\_test3, yhat3, average='weighted'))

# using the jaccard index for scoring performance

print(jaccard\_similarity\_score(y\_test3, yhat3))

# model4

print('model 4')

# using the f1\_score for scoring performance

print(f1\_score(y\_test4, yhat4, average='weighted'))

# using the jaccard index for scoring performance

print(jaccard\_similarity\_score(y\_test4, yhat4))

# log loss calculations

print(log\_loss(y\_test4, yhat\_prob4))

# turns out model 1 and model 3 perform best.

# and predict with roughly 75% accuracy.

# in order to display plot within window

# plt.show()