Facultatea de Informatica,
Universitatea din Bucuresti

Black-box Classification

Realizat de :

Paraschiv Andrei-Dumitru

Grupa 243

Prof. coord:

Cristina Madalina Noaica

Cuprins

Despre mine

Nume : Paraschiv Andrei-Dumitru

LinkedIn : https://www.linkedin.com/in/andrei-paraschiv-a3561b178/

GitHub : https://github.com/pandreyy99/

Scurta Descriere :

> “ Student in anul 2 la Facultatea de Matematica si Informatica, specializarea Informatica, grupa 43, din cadrul Universitatii din Bucuresti.
>
> Am studiat in cadrul facultatii Arhitectura sitemelor de calcul, Programare Procedurala, Concepte POO, Algoritmica si structuri de date, Analiza matematica, Statistica si Probabilistica, precum si diferite limbaje de programare(C/C++, Java, R, Python, etc.).
>
> In prezent particip la o competitie de pe platforma Kaggle. “

Cerinta Proiectului

Competitia este o provocare de tipul black-box in care trebuie sa antrenam un clasificator pentru a putea clasifica un set de 5.000 de date fara a sti ceea ce reprezinta, datele nefiind unele "human readable", avand la dispozitie si un set de 15.000 de astfel de date pentru antrenare, precum si etichetele corespunzatoare acestora.

Clasamentul in urma carora suntem clasificati se realizeaza in functie de scorul(acuratetea) clasificatorului antrenat in a recunoaste acel set de 5.000 de date, competitorii putand vizualiza in timpul competitiei doar un clasament provizoriu, bazat pe acuratetea obtinuta doar pe 20% din datele de testare, urmand ca la finalul competitiei sa fie disponibil si cel bazat pe restul datelor de testare.

Durata competitiei : 10.03.2019 19:04 UTC – 07.04.2019 23:59 UTC

Regulile competitiei :

1. O echipa este formata dintr-un singur student.
2. Competitorii au voie doar cu un singur cont, conceptul de "multy-account" fiind interzis.
3. Este interzisa transmiterea datelor si a codului intre participanti, precum si fuziunea intre echipe.
4. Fiecare echipa poate incarca maxim 2 output-uri pe zi.

Clasificatori antrenati

In cadrul competitiei m-am folosit de biblioteca scikit-learn din Python pentru a antrena diferiti clasificatori.

Pe parcursul competitiei am incercat diferiti clasificatori, precum :

- Diferite Clasificatori cu Vectori Suport(Support Vector Classifier aka SVC) din cadrul bibliotecii sklearn.svm
- Un clasificator bazat pe algoritmul de aflare a celor mai apropiati vecini(K-Nearest Neighbors aka KNN) din cadrul bibliotecii sklearn.neighbors
- Un clasificator de Regresie cu Cresterea bazata pe o componenta Nucleica(Kernel Ridge Regression aka KRR) din cadrul bibliotecii sklearn.kernel_ridge
- Mai multe retele neurale de Perceptroni (Multi Layer Perceptrons aka MLP) din cadrul bibliotecii sklearn.neural_network
- Un clasificator bazat pe algoritmul de Random Forest (Random Forest Clasifier aka RFC) din cadrul bibliotecii sklearn.ensemble
- Un clasificator de tip XGBoost (XGBClassifier aka XGBC) din cadrul bibliotecii xgboost

In urma rezultatelor obtinute si a unui proces de cercetare, am ales sa merg mai departe cu un clasificator ce se bazeaza pe algoritmul “Random Forest”(RFC) deoarece, chiar daca in urma impartirii datelor de antrenare in date de antrenare partiale si date de testare a avut o acuratete mai mica decat un clasificator cu vectori support, am inteles ca se descurca mai bine pe datele de testare care nu seamana atat de mult cu cele de antrenare, algoritmul din spate fiind unul probabilistic, si cum competitia este una de tip “black-box” si nu am putut sti cum arata datele fizic, am considerat ca merita incercat.

In prealabil am aplicat si un algoritm de determinare a Compenentei Principale(PCA – Principal Component Analysis) si am reusit sa reduc dimensiunea datelor de la 4096 de componente la doar 128, precum si un algoritm de cautare a unor parametric optimi(Grid Search) pentru clasificatorul antrenat(Random Forest Classifier) , am normalizat datele si am creat un Data Loader ce foloseste biblioteca Pandas pentru a citi datele si pentru a le stoca astfel incat sa fie mai usor accesibile si sa nu trebuieasca sa le citesc de fiecare data.

Random Forest Classifier

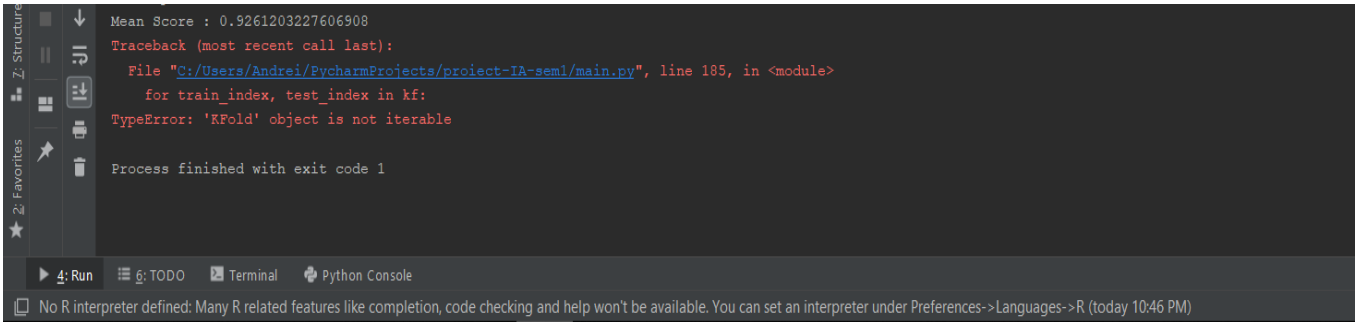
Parametri :

- n_estimators :
  - integer(implicit = 10 in versiunea 0.20 si 100 in versiunea 0.22)
  - numarul de "arbori" folositi
- criterion :
  - string(implicit = 'gini')
  - functia de masurare a calitatii impartirii datelor
  - specific "arborilor"
  - 2 variante : 'gini' pentru Gini Impurity si 'entropy' pentru functia bazata pe cresterea informatiei
- max_features :
  - int, float, string or None, optional (default="auto")
  - daca e int, atunci la fiecare split folosteste max_features de features
  - daca e float, atunci foloseste [max_features * n_features] features pentru fiecare split
  - daca e auto sau sqrt, atunci max_features=sqrt(n_features)
  - daca e log2, atunci max_features=log2(n_features)
  - daca e None, atunci max_features = n_features, *unde n_features = #features cand e rulat fit()*
- max_depth :
  - integer or None, optional (default=None)
  - Adâncimea maximă a "arborelui". Dacă e None, atunci nodurile sunt extinse până când toate frunzele sunt pure sau până când conțin mai puțin de min_samples_split mostre.
- random_state :
  - int, RandomState instance or None, optional (default=None)
  - daca e int, e folosit ca "samanta"(seed) pentru generatorul random
  - daca e instanta, atunci random_state e generatorul de numere random
  - daca e None, atunci generatorul de numere random este intanta folosita pentru np.random

```
best_randomForestClasifier  = RandomForestClassifier(criterion='entropy',
                                                     max_depth=8,
                                                     max_features='auto',
                                                     n_estimators=100,
                                                     random_state=42)
```

Procedura de tipul 3-fold cross-validation

Acuratetea medie :



Matricea de confuzie :

- Regasiti rezultatele in fisierul “confusion_matrix.txt” din cadrul arhivei

Codul Sursa

Data Loader

```
import os
import re
import pandas as pd
import numpy as np
from sklearn import preprocessing
from sklearn.utils import shuffle
import random
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA

class Data_Loader:
    random_distribution = 0
    def   init  (self):
        # self.dataset_path = "ml-unibuc-2019-24/"
        self.dataset_path = os.getcwd() + '/ml-unibuc-2019-24/'
        self.data = []
        self.labels = []
        self.eval = []

    def getStationMetadata(self):
        '''
        Retrieve metadata on preprocessed_data

        @return pandas dataframe with preprocessed_data information
        '''

        data file = os.path.exists('processed data.h5')
        # print(data file)
        if data_file is False:
            print('Dataset not available')
            return (False,None,0)

        store = pd.HDFStore('processed_data.h5', 'r')
        meta_data = store['preprocessed_data']
        random_file = open('random.txt', 'r')
        read = random_file.read()
        random = float(read)
        store.close()
        random file.close()

        return (True,meta_data,random)

        # functie care primeste path-ul catre un folder si
    # citeste toate fisierele din el, salvand recenziile intr-o lista

    def build dataset(self):
        # random_distribution = random.random()
        ok, data, random_distribution = self.getStationMetadata()
        if ok == False:
            random distribution = random.uniform(0.75,1)
            if (random_distribution < 0.5):
                random_distribution = 1 - random_distribution
            data = pd.read csv(self.dataset path + 'train samples.csv', header=None)

            # Create storage object with filename `processed_data`
            data_store = pd.HDFStore('processed_data.h5')

            # Put DataFrame into the object setting the key as 'preprocessed data'
            data_store['preprocessed_data'] = data
            random_file = open('random.txt', 'w')
            random file.write(str(random distribution) + '\n')
            data store.close()
            random_file.close()
        eval = pd.read csv(self.dataset path + 'test samples.csv', header=None)
        for rows in data:
            self.data = data.to_numpy(dtype='float')
        for rows in eval:
            self.eval = eval.to_numpy(dtype='float')
        labels = pd.read_csv(self.dataset_path + 'train_labels.csv', header=None, dtype='int')
        for rows in labels:
            self.labels = labels.to_numpy(dtype = 'int')
        self.labels = self.labels.ravel()
```

```
        # impartim setul de date in random% pentru antrenare si 1-random% pentru test
        num_training_samples_per_class = int(random_distribution * len(data))
        num_test_samples_per_class = len(data) - num_training_samples_per_class

        # in setul de antrenare bagam datele cu indecsii 0:random*15000
        train_data = self.data[0:num_training_samples_per_class]

        # la fel si in etichetele  datelor de intrare
        train_labels = self.labels[0:num_training_samples_per_class]

        # in setul de test salvam restul datelelor din cel de antrenare
        test_data = self.data[num_training_samples_per_class:len(data)]
        test_labels = self.labels[num_training_samples_per_class:len(labels)]

        # self.train_labels, self.train_data = train_labels, train_data
        # self.test_labels, self.test_data = test_labels, test_data

        # amestecam datele
        self.train_data, self.train_labels = shuffle(train_data, train_labels)
        self.test_data, self.test_labels = shuffle(test_data, test_labels)

        self.num_classes = max(np.max(self.train_labels), np.max(self.test_labels))
```

Antrenarea Clasificatorilor

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm
from sklearn.ensemble import RandomForestClassifier
from xgboost import  XGBClassifier
from sklearn.linear_model import Perceptron
from sklearn.neighbors import KNeighborsClassifier
from sklearn.neural network import MLPClassifier
from sklearn.metrics import confusion matrix
# from sklearn import preprocessing
from sklearn.kernel_ridge import KernelRidge
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
import pdb
from sklearn.model_selection import GridSearchCV

from data_loader import *

data loader = Data Loader()
data_loader.build_dataset()
train_data, test_data = data_loader.train_data, data_loader.test_data
print("Si-ncepe...")

####################################
#       Scaling before applying PCA    #
####################################

def normalize_data(train_data, test_data, type='none'):
    # ({'none','standard' , 'min_max' , 'l1' , 'l2' })

    scaler = None

    if type == 'standard':
        scaler = preprocessing.StandardScaler()

    elif type == 'min_max':
        # min_max_scaler = preprocessing.MinMaxScaler(feature_range=(0, 1))  # (0, 1) default
        scaler = preprocessing.MinMaxScaler()
    elif type == 'l1':
        scaler = preprocessing.Normalizer('l1')
    elif type == 'l2':
        # scaler = preprocessing.Normalizer() implicit l2
        scaler = preprocessing.Normalizer('l2')

    if scaler is not None:
        scaler.fit(train data)
        scaled_x_train = scaler.transform(train_data)
        scaled_x_test = scaler.transform(test_data)
        return scaled x train, scaled x test
        print("scaled")
    else:
        return train_data, test_data

# Here I apply Standard scaler on data

train datas , test datas = normalize data(train data, test data, 'standard')
scaler = preprocessing.StandardScaler()
scaler.fit(train_data)
eval_data = scaler.transform(data_loader.eval)

# Principal Component Analysis
pca = PCA(.85)
pca.fit(train_data)
train datas = pca.transform(train data)
test_datas = pca.transform(test_data)
eval_data = pca.transform(data_loader.eval)

# print('Train data shape : ', np.shape(train datas))
# print('Test data shape : ', np.shape(test_datas))
# print('eval data shape : ', np.shape(eval_data))

# print("test")
# print(len(data_loader.test_data))
# print("train")
# print(len(data loader.train data))
# print(len(data_loader.train_labels))
```

```
# print(data_loader.num_classes)

def get_accuracy(eticheta1, eticheta2):
    return len(eticheta1[eticheta1 == eticheta2]) / len(eticheta1)

def get accuracy statistics(train data, test data, Cs, normalization type='none', svm type='linear'):
    # 1 Aceasta normalizează datele
    train_data, test_data = normalize_data(train_data, test_data, normalization_type)

    # 2 Antrenează câte un SVM pentru fiecare valoare din C

    clase_train = np.zeros(len(Cs))
    clase test = np.zeros(len(Cs))
    for index in range(len(Cs)):
        svm_classifier = svm.SVC(C=Cs[index], kernel=svm_type)
        svm_classifier.fit(train_data, data_loader.train_labels)
        clase_train[index] = get_accuracy(svm_classifier.predict(train_data), data_loader.train_labels)
        clase_test[index] = get_accuracy(svm_classifier.predict(test_data), data_loader.test_labels)

    # 3 Returnează 2 vectori conţinând acurateţea fiecărui model pe datele de antrenare, respectiv de test

    return clase_train, clase_test

'''
def build confusion(etichete reale, etichete prezize):
    clase = max(etichete_reale) + 1
    confuzie = np.zeros((clase,clase))
    for i in range(clase):
        for j in range(clase):
            perechi = [(et1,et2) for (et1, et2) in zip(etichete_reale, etichete_prezize) if et1 == i and et2 ==
j]
            confuzie[i,j] = len(perechi)
    return confuzie

def precision(confusion_matrix, length):
    precisionArray = np.zeros(length)
    for i in range(length):
        sum = 0
        for j in range(length):
            sum += confusion_matrix[i,j]
        precisionArray[i] = confusion_matrix[i,i] / sum
    return  precisionArray
'''
# # Cs = [1e-8, 1e-7, 1e-6, 1]
# Cs = [1e-7, 1e-6, 1e-4, 1e-2, 1e-1, 0.5, 1, 10, 100]
# Cs = [1e-4, 1e-2, 1e-1, 0.2e-1, 0.5, 1, 10, 100 ]
# Cs = [100]

# train_data = np.reshape(train_data, (len(train_data),4096))
# test_data = np.reshape(test_data, (len(test_data),4096))

################################################
#                      SVM
################################################

#file_to_write_scores = open('scores.txt', 'a')

# std_accuracies_train, std_accuracies_test = get_accuracy_statistics(train_datas, test_datas, Cs, 'standard')
# file_to_write_scores.write('std accuracies train : ' +  str(std_accuracies_train) + '\n')
# file_to_write_scores.write('std acc test : ' + str(std_accuracies_test) + '\n')
# std_accuracies_train, std_accuracies_test = get_accuracy_statistics(train_datas, test_datas, Cs, 'standard',
'rbf')
# print(std_accuracies_train, end='\n')
# print(std_accuracies_test + '\n', end='\n')

# classifier = svm.SVC()
# print("It's fitting")
# classifier.fit(train_datas, data_loader.train_labels)
# score = classifier.score(test datas, data loader.test labels)
#
# print("score : " , end='\n')
# print(score,end='\n')


normalize_data(train_datas, test_datas, 'standard')
normalize_data(train_datas, eval_data, 'standard')
'''
rf_classifier = RandomForestClassifier(random_state=42)

param grid = {
    'n_estimators': [100, 200, 500],
```

```
    'max_features': ['auto', 'sqrt', 'log2'],
    'max_depth' : [4,5,6,7,8],
    'criterion' :['gini', 'entropy']
}

CV_rfc = GridSearchCV(estimator=rf_classifier, param_grid=param_grid, cv= 5)
CV_rfc.fit(train_data, data_loader.train_labels)
print(CV_rfc.best_params_)
best_p1 = CV_rfc.best_params_
'''
rfc = RandomForestClassifier(criterion='entropy', max_depth=8, max_features='auto', n_estimators=100,
random_state=42)
rfc.fit(train_data, data_loader.train_labels)
predicted_labels = rfc.predict(data_loader.eval)
score = rfc.score(test_data, data_loader.test_labels)
# print("score : " + str(score), end='\n')

from sklearn.model_selection import cross_val_score

scores = cross_val_score(rfc, train_data, data_loader.train_labels, cv=3)
scores = np.array(scores)
mean_score = scores.mean()
print("Mean Score : " + str(mean_score))

from sklearn.model_selection import KFold

kf = KFold(n_splits=3)

file_to_write_confusion_matrix = open('confusion_matrix.txt', 'w')

for train_index, test_index in kf:
    X_train, X_test = train_data[train_index], train_data[test_index]
    y_train, y_test = data_loader.train_labels[train_index], data_loader.train_labels[test_index]
    rfc.fit(X_train, y_train)
    file_to_write_confusion_matrix.write(confusion_matrix(y_test, rfc.predict(X_test)))

'''
CV_rfc = GridSearchCV(estimator=rf_classifier, param_grid=param_grid, cv= 5)
CV_rfc.fit(train_datas, data_loader.train_labels)
print(CV_rfc.best_params_)
best_p2 = CV_rfc.best_params_
'''


# file_to_write_scores.write('std_rbf_accuracies_train : ' +  str(std_accuracies_train) + '\n')
# file_to_write_scores.write('std_rbf_acc_test : ' + str(std_accuracies_test) + '\n')
# std_accuracies_train, std_accuracies_test = get_accuracy_statistics(train_datas, test_datas, Cs, 'standard',
'sigmoid')
# file_to_write_scores.write('std_sigmoid_accuracies_train : ' +  str(std_accuracies_train) + '\n')
# file_to_write_scores.write('std_sigmoid_acc_test : ' + str(std_accuracies_test) + '\n')

# print("de aici printeaza",end='\n')

'''
perceptron = Perceptron( penalty='l2', tol=1e-5, shuffle=True, eta0= 0.5, n_jobs= -1, early_stopping=True,
n_iter_no_change= 10)
perceptron.fit(train_datas, data_loader.train_labels)
score = perceptron.score(test_datas, data_loader.test_labels)
print(score,end='\n')
# file_to_write_scores.write('Perceptron : ' + str(score) + '\n')

perceptron = Perceptron( penalty='l1', tol=1e-7, shuffle=True, eta0= 0.2, n_jobs= -1, early_stopping=True,
n_iter_no_change= 10)
perceptron.fit(train_datas, data_loader.train_labels)
score = perceptron.score(test_datas, data_loader.test_labels)
print(score,end='\n')

perceptron = Perceptron( penalty='elasticnet', tol=1e-5, shuffle=True, eta0= 0.5, n_jobs= -1,
early_stopping=True, n_iter_no_change= 10)
perceptron.fit(train_datas, data_loader.train_labels)
score = perceptron.score(test_datas, data_loader.test_labels)
print(score,end='\n')
'''
'''
file_to_write_scores.write('Here starts mlps : ' + '\n')
mlp = MLPClassifier(activation='tanh', solver='lbfgs', learning_rate='adaptive', shuffle=True, verbose=True,
early_stopping=True)
mlp.fit(train_datas, data_loader.train_labels)
score = mlp.score(test_datas, data_loader.test_labels)
print(score,end='\n')
file_to_write_scores.write(str(score) + '\n')
```

```
mlp = MLPClassifier(activation='logistic', solver='lbfgs', learning_rate='adaptive', shuffle=True,
verbose=True, early_stopping=True)
mlp.fit(train_datas, data_loader.train_labels)
score = mlp.score(test_datas, data_loader.test_labels)
print(score,end='\n')
file_to_write_scores.write(str(score) + '\n')

mlp = MLPClassifier(activation='relu', solver='lbfgs', learning_rate='adaptive', shuffle=True, verbose=True,
early stopping=True)
mlp.fit(train datas, data loader.train labels)
score = mlp.score(test_datas, data_loader.test_labels)
print(score,end='\n')
file to write scores.write(str(score) + '\n')

mlp = MLPClassifier(activation='tanh', solver='sgd', learning_rate='adaptive', shuffle=True, verbose=True,
early_stopping=True)
mlp.fit(train_datas, data_loader.train_labels)
score = mlp.score(test_datas, data_loader.test_labels)
print(score,end='\n')
file_to_write_scores.write(str(score) + '\n')
'''
#########################################################
#    BEST       BEST        BEST        BEST        BEST
#########################################################

'''
mlp = MLPClassifier(activation='logistic', solver='sgd', learning rate='adaptive', shuffle=True,
early stopping=True)
mlp.fit(train_datas, data loader.train_labels)
score = mlp.score(test_datas, data_loader.test_labels)
print(score,end='\n')
file_to_write_scores.write(str(score) + '\n')
'''

'''
mlp = MLPClassifier(activation='relu', solver='sgd', learning_rate='adaptive', shuffle=True, verbose=True,
early_stopping=True)
mlp.fit(train datas, data loader.train labels)
score = mlp.score(test_datas, data_loader.test_labels)
print(score,end='\n')
file_to_write_scores.write(str(score) + '\n')

mlp = MLPClassifier(activation='tanh', solver='adam', learning_rate='adaptive', shuffle=True, verbose=True,
early_stopping=True)
mlp.fit(train_datas, data_loader.train_labels)
score = mlp.score(test datas, data loader.test labels)
print(score,end='\n')
file_to_write_scores.write(str(score) + '\n')

mlp = MLPClassifier(activation='logistic', solver='adam', learning rate='adaptive', shuffle=True, verbose=True,
early_stopping=True)
mlp.fit(train_datas, data_loader.train_labels)
score = mlp.score(test_datas, data_loader.test_labels)
print(score,end='\n')
file_to_write_scores.write(str(score) + '\n')

mlp = MLPClassifier(activation='relu', solver='adam', learning_rate='adaptive', shuffle=True, verbose=True,
early_stopping=True)
mlp.fit(train_datas, data_loader.train_labels)
score = mlp.score(test_datas, data_loader.test_labels)
print(score,end='\n')
file_to_write_scores.write(str(score) + '\n')

# train data = np.reshape(train data, (len(train data),4096))
# test data = np.reshape(test data, (len(test data),4096))

#################################################
#            Kernel Ridge Regresion(KRR)
#################################################

krr_classifier = KernelRidge(kernel='rbf')
krr classifier.fit(train datas, data loader.train labels)
score = krr_classifier.score(test_datas, data_loader.test_labels)
print(score,end='\n')
file_to_write_scores.write('KRR : ' + str(score) + '\n')

#########################
#       KNN
#########################
```

```
knn_clasifier = KNeighborsClassifier(weights='distance', algorithm='auto')
knn_clasifier.fit(train_datas, data_loader.train_labels)
score = knn_clasifier.score(test_datas, data_loader.test_labels)
print(score,end='\n')
file_to_write_scores.write('KNN : ' + str(score) + '\n')

# etichete = krr_classifier.predict(test_data)
#print(etichete)
'''
##############################################
#           Submission
##############################################
# print("Predict : ")
#
# predicted_labels = classifier.predict(data_loader.eval)
#
# import csv
#
# with open('sample_submission.csv', 'w', newline='') as csv_file:
#     writer = csv.writer(csv_file)
#     # header = ["Id" , "Prediction"]
#     writer.writerow(["Id", "Prediction"])
#     for i in range(len(predicted_labels)):
#         list = []
#         list.append(i + 1)
#         list.append(predicted_labels[i])
#         writer.writerow(list)

'''
predicted_labels = mlp.predict(eval_data)
predicted_labels = mlp.predict(data_loader.eval)
'''


for label in predicted_labels:
    print(label , end=" ")
print('\n\n\n')

import csv

with open('sample_submission-random_forest_classifier2.csv', 'w', newline='') as csv_file:
    writer = csv.writer(csv_file)
    # header = ["Id" , "Prediction"]
    writer.writerow(["Id", "Prediction"])
    for i in range(len(predicted_labels)):
        list = []
        list.append(i + 1)
        list.append(predicted labels[i])
        writer.writerow(list)

'''
xgb1 = XGBClassifier(learning_rate =0.1, n_estimators=1000, max_depth=5, min_child_weight=1, gamma=0,
subsample=0.8,colsample_bytree=0.8,
                     objective= 'binary:logistic', nthread=4, scale_pos_weight=1, seed=27)

param_grid = {
    'max_depth':range(3,10,2),
    'min_child_weight':range(1,6,2),
    'gamma':[i/10.0 for i in range(0,5)],
    'subsample':[i/10.0 for i in range(6,10)],
    'colsample_bytree':[i/10.0 for i in range(6,10)],
    'reg alpha':[1e-5, 1e-2, 0.1, 1, 100]
}

xgb grid = GridSearchCV(xgb1, param grid=param grid, scoring='roc auc', n jobs=4, iid=False, cv=5)
xgb grid.fit(train data, data loader.train labels)
print("xgb_grid score : " + str(xgb_grid.best_score_))
print("best param : ",end='\n')
print(xgb grid.best params )
params = open('params.txt', 'w')
params.write("Best Score : ")
params.write(xgb_grid.best_score_)
params.write("Best Params : ")
params.write(xgb_grid.best_params_)

xgbc = XGBClassifier()
xgbc.fit(train_datas, data_loader.train_labels)
predicted_labels_xgbc = xgbc.predict(eval_data)
score = xgbc.score(test_datas, data_loader.test_labels)
print("XGB score : " + str(score), end='\n')
```

```
for label in predicted_labels_xgbc:
    print(label , end=" ")
print('\n\n\n')

import csv

with open('sample_submission-XGBC.csv', 'w', newline='') as csv_file:
    writer = csv.writer(csv_file)
    # header = ["Id" , "Prediction"]
    writer.writerow(["Id", "Prediction"])
    for i in range(len(predicted labels xgbc)):
        list = []
        list.append(i + 1)
        list.append(predicted labels xgbc[i])
        writer.writerow(list)
'''
```