Perceptron votato

Imports - Prime variabili

In [ ]:

```
###########
# Imports #
###########

import numpy as np
import pandas as pd
import json
import sys
import os
import matplotlib.pyplot as plt
import time
from sklearn import datasets as ddd
import sklearn

###################
# Prime variabili #
###################

file_path = "Dataset/firewall/log2.csv"                 # Dataset
df_complete = pd.read_csv(file_path)                # Seleziona CSV - Tutto il Dataset in fil
df_complete.dropna()
```

In [ ]:

```
####################
# Dataframe intero #
####################
columns_to_drop = ["Source Port", "Destination Port", "NAT Source Port", "Action", "NAT Desti
df_complete = df_complete.drop(columns=columns_to_drop)
print(df_complete)
```

Funzioni - Wrappers

In [ ]:

```
###########################
# Funzioni per il dataset #
###########################

# Crea la mappa degli attributi, che serve per capire se un attributo è positivo o negativo
# Se si passa map_json, vuol dire che si è già ricomposto un json precedentemente salvato, e
# Una versione incapsulata è quella dopo, ovvero get_attribute_map_from_json

def generate_attributes_map(map_json=[], dataframe=None, save_json=False, name="DELETABLE", n
    map_of_attributes = map_json
    if dataframe is None and len(map_of_attributes)==0:
        return "error - bad call"
    if len(map_of_attributes) == 0 :
        for col in list(dataframe.columns.values):
            if numeric_dataset:
                column_to_array = dataframe[col].to_numpy()
                max = float(dataframe[col].max())
                min = float(dataframe[col].min())
                mid_unstandardized = np.mean(column_to_array)#(max + min) / 2
                if standardize:
                    #mean = float(np.mean(column_to_array))
                    #std = float(np.std(column_to_array))
                    for i in range(len(dataframe[col])):
                        dataframe.at[i, col] = 2 * (dataframe.at[i, col]-min)/(max - min) - 1
                    new_max = float(dataframe[col].max())
                    new_min = float(dataframe[col].min())
                    mid = (new_max + new_min) / 2
                else:
                    mid = mid_unstandardized
                map_of_attributes.append((str(col), "middle", [mid, mid_unstandardized]))
            else:
                attributes = df_complete[col].unique()
                split = np.array_split(attributes, 2)
                map_of_attributes.append((str(col), "positive", list(split[0])))
                map_of_attributes.append((str(col), "negative", list(split[1])))
    attributes_map = pd.DataFrame(map_of_attributes, columns=["Column", "Sign", "Attributes"]

    if save_json:
        with open("Dataset/MappedAttributes/"+name+".json", "w") as outfile:
            outfile.write(json.dumps(map_of_attributes))

    return attributes_map

# Crea un dataframe che contiene tutte le tuple di entry (ovvero tutti i vettori x) che servo

def map_values(dataframe, attributes_map, numeric_values=False):
    df_mapped = pd.DataFrame(columns=dataframe.columns)
    if not numeric_values:
        for col in dataframe.columns:
            positives = list(attributes_map.query("Column == '%s' & Sign == 'positive'" %col
            negatives = list(attributes_map.query("Column == '%s' & Sign == 'negative'" %col
            for i in range(len(dataframe)):
                if dataframe.loc[i][col] in positives:
                    df_mapped.at[i, col] = 1
                elif dataframe.loc[i][col] in negatives:
                    df_mapped.at[i, col] = -1
                else:
                    df_mapped.at[i, col] = 0
                    print("ERROR: This attribute isn't positive or negative!!" + str(datafram
                    return
    else:
        return pd.DataFrame(dataframe)    # Se sono valori numerici, sono stati già normalizz
    return df_mapped

def generate_weights(n_attributes, contain_target=True, init_zero=False):
    attributes = n_attributes
    if contain_target:
        attributes -= 1
    # attributes -= len(columns_to_drop)              # Bisogna rimuovere anche il numero di
    weights = np.zeros( attributes )
    if init_zero:
        return weights
    len_weights = len(weights)
    for i in range(len_weights):
        weight = 1/len_weights
        weights[i] = weight
    return weights

def calculate_R(df_mapped, full):
    max = 0
    len_df_mapped = len(df_mapped)
    count = len_df_mapped // 10
    for i in range(len_df_mapped):
        valid_avg = len(df_mapped) - 1
        sys.stdout.flush()
        sys.stdout.write("\r["+str(i+1)+" on "+str(len_df_mapped)+"] - R = "+str(max)+". Rema
        for j in range(i, len_df_mapped):
            distance = np.linalg.norm(np.array(df_mapped.loc[i][:]) - np.array(df_mapped.loc[
            if distance > max:
                if not (distance - 2. <= max and max < distance + 2.):
                    valid_avg = len(df_mapped) - 1
                    count = len_df_mapped // 10
                max = distance
            else:
                if distance - 2. <= max and max < distance + 2.:
                    valid_avg -= 1
        if valid_avg > 0:
            count -= 1
            if count == 0 and not full:
                return max
    return max


def split_dataset(dataset, train_percentage=80):
    total_entries = len(dataset)
    x = total_entries // 100 * train_percentage
    remaining = total_entries - x
    dataset.head(x).to_csv("Dataset/Productions/Train/" + new_file + ".csv", index=False)
    dataset.tail(remaining).to_csv("Dataset/Productions/Test/" + new_file + ".csv", index=Fal

# Funzioni per aggiornare il valore dentro "Last.txt" che serve per differenziare i vari test

def get_last_ID(increase=False):
    with open("Last.txt") as opened:
        a = str(opened.read())
    if increase:
        increase_ID()
    return a

def increase_ID():
    actual = int(get_last_ID())
    actual += 1
    with open("Last.txt", "w") as outfile:
        outfile.write(str(actual))

def reset_ID():
    with open("Last.txt", "w") as outfile:
        outfile.write(str(1))

# Da usare per caricare in un DataFrame pandas gli attributi da un json salvato in Dataset/Ma
def get_attributes_map_from_json(name, numeric=False, dataframe=df_complete):
    with open("Dataset/MappedAttributes/" + name + ".json") as json_file:
        return generate_attributes_map(map_json=json.load(json_file), dataframe=dataframe, nu

# Conta il numero totale di attributi unici nella colonna "Column" della mappa degli attribut
def get_count_attributes(attributes_map):
    return len(attributes_map["Column"].unique())
```

In [ ]:

```
############
# Wrappers #
############

# Standardizza l'intero dataset - Inoltre, creo la attribute map, che contiene il valore di m
# Ritorna una attribute_map che contiene i threshold di ogni singolo attributo

def standardize_dataset(dataset):
    generate_attributes_map(dataframe=dataset, numeric_dataset=True, standardize=True)


# Calcola i punti di mid per ogni attributo del dataframe
def get_mid_thresholds(dataset):
    return generate_attributes_map(dataframe=dataset, numeric_dataset=True, standardize=False


# Carica la mappa degli attributi - Nel caso di dataset numerici, genera la mappa dei thresho
def get_attributes_map(name, numeric=False, dataframe=df_complete):
    if not os.path.exists("Dataset/MappedAttributes/" + name + ".json"):
        attributes_map = generate_attributes_map(dataframe=dataframe, save_json=True, name=na
    else:
        attributes_map = get_attributes_map_from_json(name, numeric=numeric)
    return attributes_map

# Crea un vettore dei pesi - Inizializzato a 0 || Inizializzato a 1/n per ogni i
def get_weights(n_attributes, contain_target=True, init_zero=False):
    return generate_weights(n_attributes, contain_target, init_zero)

# Calcola la distanza massima tra ogni vettore del dataframe (COSTOSO)
def get_R(dataframe, full=True):
    return calculate_R(dataframe, full)
```

Preparazione del Dataset

In [ ]:

```
######################################################################
# Dividi dataframe - Inizializza scegliendo un nome per il dataframe #
######################################################################

# Seleziona dataframe - BISOGNA AVERE UN DATASET che contenga anche l'attributo target
new_file = "firewall"                                      # Nome del file in cui salvare il
numeric_dataset = True
standardize = False

if standardize:
    standardize_dataset(df_complete)

split_dataset(df_complete)                                  # Dividi il dataset in due .csv -

attributes_map = get_attributes_map(name=new_file, numeric=numeric_dataset, dataframe=df_comp
attributes_number = get_count_attributes(attributes_map=attributes_map)

root = "Dataset/Productions/"
df_train = pd.read_csv(root+"/Train/"+new_file+".csv")      # Dataframe di TRAIN
df_test = pd.read_csv(root+"/Test/"+new_file+".csv")        # Dataframe di TEST
```

Training

Funzioni

In [ ]:

```
def sign(val):
    if val >= 0:
        return 1
    return -1

def count_targets(df_train, name_target):
    medium = get_medium(standardize, name_target)
    positives = 0
    negatives = 0
    for k in range(len(df_train[name_target])):
        if (df_train.at[k, name_target]) >= medium:
            positives += 1
        else:
            negatives += 1
    print("Positivi:", positives, "- Negativi:", negatives)

# Separa il dataframe dal target
def get_dataframes_train(df_train=[], name_target="", attributes_map=[]):
    if len(attributes_map)==0 or len(df_train)==0 or name_target=="":
        return "error - bad call"
    df_mapped = map_values(df_train, attributes_map, numeric_values=numeric_dataset)    # Cre
    df_target = df_mapped.pop(name_target)                                              # Est
    return df_mapped, df_target

# Calcola segno del target - Se non ho un dataset numerico, ho già il target mappato
# Estraggo il segno del target, mettendo "-" se appartiene all'intervallo (-inf, medium)
def get_target_sign(target, medium):
    if not numeric_dataset:
        return target
    if target < medium:
        return -1
    else:
        return 1

# Data la attribute map, controlla un input per vedere se appartiene alla classe positiva o n
def is_wrong(w_sum, target, medium):
    if not numeric_dataset:
        return w_sum * target < 0
    else:
        y = get_target_sign(target, medium)
        yhat = sign(w_sum)
        if yhat * y < 0:                       # i valori da -inf a medium, sono da considera
            return True
        return False

# Utilizza name_target per estrarre i range positivi e negativi di un target
def get_medium(standardize, name_target):
    if numeric_dataset:
        if standardize:
            medium = list(attributes_map.query("Column=='"+name_target+"' & Sign=='middle'")[
        else:
            medium = list(attributes_map.query("Column=='"+name_target+"' & Sign=='middle'")[
    else:
        return 0
    return medium

# Crea un dizionario con [k] = {[Weight_k], [bias], [c (num errori)]}
# e una lista di coppie [epoca, num_errori]
def train_model(df_mapped, df_target, weights, bias, medium):
    perceptrons = {}
    epochs_errors = []
    k = 0
    c = 0
    for e in range(epochs):
        num_errors = 0
        for i in range(len(df_target)):
            w_sum = np.dot(df_mapped.loc[i][:], weights) + bias
            if is_wrong(w_sum, df_target[i], medium):
                num_errors += 1
                perceptrons[k] = [list(weights), bias, c]
                c = 1
                #norm = np.linalg.norm(weights)
                for j in range(len(weights)):
                    weights[j] = weights[j] + (get_target_sign(df_target[i], medium) * df_map
                bias = get_target_sign(df_target[i], medium) * (R**2)
                k += 1
            else:
                c += 1
        epochs_errors.append([e, num_errors])
        sys.stdout.flush()
        sys.stdout.write( "\rEpoch: "+ str(e) + " - Errors:" + str(num_errors))
    if len(perceptrons) == 0:
        perceptrons[k] = [list(weights), bias, c]
    return perceptrons, epochs_errors

# Si passano i due dataframe di train e target, fa il train su quel dataset ed eventualmente
# medium serve per i problemi di classificazione binaria su valori reali. Usare get_medium(st
# E' possibile inserire l'indice
def train_and_save_res(df_mapped, df_target, weights, bias, save=True, add_index=True):
    perceptrons, epoch_errors = train_model(df_mapped, df_target, weights, bias, medium)
    ind = ""
    if save:
        if add_index:
            ind = "_"+get_last_ID(True)
        json_perceptrons = "Perceptrons/"+new_file+ind+".json"        # Devo salvare la lista
        epo_erro = "Evidences/Train/"+new_file+ind+".json"

        with open(json_perceptrons, "w") as outfile:
            outfile.write(json.dumps(perceptrons))

        with open(epo_erro, "w") as outfile:
            outfile.write(json.dumps(epoch_errors))
    return perceptrons, epoch_errors
```

Seleziona i parametri per il Train

Inserisci in *name_target* scegliendo uno di quelli sopra

In [ ]:

```
# Stampa la lista degli attributi
print(df_train)
```

In [ ]:

```
name_target = "Rings"                                   # Imposta il nome dell'attributo
```

In [ ]:

```
# Dati da calcolare sul dataset (operazioni costose)
count_targets(df_train, name_target)

df_mapped, df_target = get_dataframes_train(df_train=df_train, name_target=name_target, attri

R = get_R(df_mapped, full=False)   # Alternativamente, imposta il valore se già conosciuto
```

In [ ]:

```
# Costanti per il perceptrons
weights = get_weights(attributes_number, contain_target=True, init_zero=True)
bias = 1
epochs = 30
medium = get_medium(standardize, name_target)
```

In [ ]:

```
perc, epc = train_and_save_res(df_mapped, df_target, weights, bias, save=True, add_index=True
```

In [ ]:

```
print(medium)
print(weights)
```

Test

In [ ]:

```
perceptron_name = "abalone"
index = "_3"

with open("Perceptrons/" + perceptron_name + index +".json") as json_file:
    test_perceptrons = json.load(json_file)

df_test_mapped = map_values(dataframe=df_test, attributes_map=attributes_map, numeric_values=
df_test_target = df_test_mapped.pop(name_target)          # name_target è il nome della col
```

In [ ]:

```
# Perceptrons from json : [0] lista pesi , [1] bias, [2] c (peso, ovvero numero di previsioni

def predict(perceptrons_from_json, input_values):
    average = 0.
    voted = 0.
    for i in perceptrons_from_json:
        w_sum = np.dot(perceptrons_from_json[i][0], input_values)# + perceptrons_from_json[i]
        average += perceptrons_from_json[i][2] * w_sum
        voted += perceptrons_from_json[i][2] * sign(w_sum)
    return sign(average), sign(voted)
```

In [ ]:

```
# couple_avg_voted{k} [0] è il risultato di perceptron avg, [1] risultato di voted e [2] targ
couple_avg_voted = {}

for k in range(len(df_test_mapped)):
    couple_avg_voted[k] = list(predict(test_perceptrons, df_test_mapped.loc[k][:]))
    couple_avg_voted[k].append(df_test_target[k])

total_test_values = len(df_test_target)
n_correct_avg = 0
n_correct_vote = 0

for k in couple_avg_voted:
    if couple_avg_voted[k][0] == get_target_sign(couple_avg_voted[k][2], medium):
        n_correct_avg += 1
    if couple_avg_voted[k][1] == get_target_sign(couple_avg_voted[k][2], medium):
        n_correct_vote += 1

mistakes_avg = total_test_values - n_correct_avg
mistakes_vote = total_test_values - n_correct_vote

avg = {"mistakes" : mistakes_avg, "correct" : n_correct_avg, "total":total_test_values}
vote = {"mistakes": mistakes_vote, "correct" : n_correct_vote, "total":total_test_values}

str_mistakes_avg = "Mistakes in avg: " + str(mistakes_avg) + " - Total correct: " + str(n_cor
str_mistakes_vote = "Mistakes in voted: " + str(mistakes_vote) + " - Total correct: " + str(n

with open("Evidences/Test/AVG/"+perceptron_name+".json", "w") as opened:
    opened.write(json.dumps(avg))

with open("Evidences/Test/Vote/"+perceptron_name+".json", "w") as opened:
    opened.write(json.dumps(vote))

print(str_mistakes_vote)
print(str_mistakes_avg)
```

In [ ]:

```
print(couple_avg_voted)
```

Creazione di grafici

In [ ]:

```
def get_x_y_train(name, time=""):
    x = []
    y = []
    if time != "":
        id = "_"+ time
    with open("Evidences/Train/"+name+id+".json") as opened:
        data = json.load(opened)
    for k in data:
        x.append(k[0])
        y.append(k[1])
    return x, y

x, y = get_x_y_train("abalone", str(3))

plt.plot(x,y)
plt.xlabel("Epochs")
plt.ylabel("N. Errors")
plt.title(perceptron_name + " - Training")
```

In [ ]:

```
##
# Create Dataframe contenente PERC - Indovinate - Sbagliate
printer = []

printer.append(avg)
printer.append(vote)

pd.DataFrame(printer, index=["AVG", "Vote"])
```