Rapport : Projet Khalo

Description du groupe :

- Membres : COHEN Raphaël, LANSARI Mohammed, BOUZID Yanis, MILLS Dylan LESPINE Nathan, RAIMBAUD REMAZEILLES Raphaël
- URL GitHub du Repo : https://github.com/Kahloo/kahlo
- URL du projet : https://codalab.lri.fr/competitions/404

Introduction et motivation

Avec la venue de l'intelligence artificielle au sein de plusieurs domaines qu'ils soient culturels, scientifique ou encore le commercial, beaucoup de question d'éthique se sont posé. Parmi ces questions, beaucoup d'interrogation se posent dans le domaine de l'art et plus particulièrement dans la peinture. Ayant pour but premier de comparer et classifier des images, les réseaux de neurones appliqués à la différenciation de photo ont ensuite était décliné pour reproduire des images à partir des données apprises. Les ressemblances entre vrais peinturent et fausses peintures étant très semblables, une problématique s'est créée autour de l'authenticité des peintures. Ainsi le but de notre projet sera justement de crée un algorithme capable de classifier et donc de différencier les fausses (crées artificiellement) des vraies peintures. Pour cela nous auront à notre disposition 65856 images d'entrainement, 9408 images de validation et 18817 images de test respectivement. Ces images seront compréssé de part le fait que ces dernières ont une grande taille : ceci facilitera le temps de calcul des algorithmes de classification. Nous utiliserons plusieurs algorithmes de classifications puis afin d'évaluer les performances de ces derniers, nous pourrons tracer la courbe ROC correspondante qui est vraiment très significatif. On étudiera plus particulièrement l'aire sous la courbe (AUC) ROC qui d'après le site de google sur la machine Learning "fournit une mesure agrégée des performances pour tous les seuils de classification possibles"[5] . Ce dernier sera déterminé lors des soumissions Codalab que nous effectuerons.
Pour réaliser ce projet, nous nous sommes réparti le travail selon trois étapes essentielles. Dans un premier temps un groupe s'occupera de la partie preprocessing. Ensuite un autre binôme réalisera la comparaison des différents algorithmes (classifieur). Et pour finir, le dernier groupe s'occupera de représenter graphiquement les résultats obtenus afin de faciliter la lecture et la comparaison des données.

I) Preprocessing : préparation des données (Raphaël et Raphaël)

Durant le processus de *preprocessing*, nous allons transcrire les images fournies sous une forme simple et facilement utilisable par le modèle. Les bibliothèques *scikit-learn* [1] et *numpy* [2] seront utilisées afin de standardiser et redimensionner les données.

D'abord, nous allons transformer les images en vecteurs à 1 dimension. Les images utilisées sont de dimensions 64*64*3, ce qui correspond donc au format initial des vecteurs représentant chaque image. La fonction *flatten* de la bibliothèque *numpy* sera utilisée pour « aplatir » les données en vecteurs de 12 288 éléments qui décrivent les 12 288 pixels de chaque image.

Ensuite, nous allons standardiser les données afin de les rendre plus facilement traitables par le modèle de classification. Pour cela on utilise la classe *StandardScaler* de *scikit-learn*afin de standardiser les valeurs des éléments de chaque vecteur de données. Cette opération transforme les éléments de manière à ce que la moyenne des éléments d'un vecteur soit égale à 0 et que leur écart-type soit égal à 1. Cette transformation est ainsi appliquée sur tous les vecteurs.

Enfin, nous allons réduire les dimensions des données. En effet, chaque image est représentée par un vecteur à 12 288 éléments, ce qui est beaucoup trop grand pour le modèle. Nous allons donc nous servir de la classe *PCA* de la bibliothèque *scikit-learn* afin de transformer chaque vecteur en vecteur à 200 éléments.

En résumé, nous allons :

1. Transformer les images en vecteurs à 1 dimension.
2. Standardiser les données (avec une distribution Gaussienne) pour avoir une moyenne des éléments du vecteur égale à 0 et un écart-type égal à 1.
3. Réduire la dimension des vecteurs.

Afin de tester cette classe, nous allons vérifier que les données après le *preprocessing* ont bien les propriétés attendues. La fonction *shape* sera utilisée pour connaître le nombre d'images dans chaque set ainsi que le nombre d'éléments décrivant chacune de ces images. Cette fonction sera utile pour savoir si la réduction de dimensionalité s'est déroulée correctement et si le nombre d'images n'a pas été altéré au cours du *preprocessing*. Pour savoir si les éléments ont bien une valeur standardisée, on pourra afficher la valeur des éléments de la première image après le *preprocessing*.

II) Classifieur : comparaison des algorithmes (Mohammed et Yanis)

Un fois toutes les données traitées, il faut dans le cadre de l'apprentissage supervisé utiliser un algorithme de classification binaire. En effet puisque nous voulons simplement différencier une fausse d'une vraie peinture, il suffit d'implémenter un algorithme du type perceptron qui va s'entrainer à « séparer linéairement » nos données afin de pouvoir affirmer si une peinture qu'il n'a jamais vue est bonne ou non [4]. Pour Cela nous avons à notre disposition plusieurs modules python : notre stratégie sera en effet de choisir les algorithmes les plus pertinents à notre cas d'utilisation afin d'avoir un pourcentage de de réussite le plus élevé possible (dans la mesure du possible étant donnée la définition de nos images). Nous utiliserons principalement des modules de la bibliothèque *sklearn.linear_model* [1] .

Dans un premier temps nous pourront implémenter un simple perceptron [3] [4] afin d'avoir un premier point de départ pour pouvoir comparer nos modèles entres eux. Le binôme s'occupant de la visualisation pourra commencer à représenter les premiers résultats toujours à titre de comparaison avec les modèles que nous réaliserons par la suite.

Par la suite nous testeront différents algorithmes d'entrainement comme par exemple le module *GaussianNB* [1] qui semble très efficace.

Avant de les implémenter nous testerons leur bon fonctionnement grâce à des tests unitaires où l'algorithme devra déterminer le label des deux points violets visible en annexe dans un repère en deux dimensions.

Pour effectuer nos comparaisons nous pourrons en coopération avec le binôme de visualisation, déterminer les AUC des ROC [5] afin d'avoir des valeurs comparables entre elles. On pourrait ensuite utiliser un code permettant de comparer les classificateurs entre eux en affichant des courbes représentatifs (séparateur linéaire ou nuage de point) et ainsi savoir lequel est le plus performant pour séparer linéairement les fausses des vraies peintures. On utilisera pour cela le fichier python zClassifier du TP2 qui est très complet.
En effet nous avons déjà réussi à essayer plusieurs algorithmes de classification dont les premiers scores de validation sont en annexes.

III) Visualisation : exploitation des résultats (Dylan et Nathan)

Le but de notre travail est d'afficher les résultats et données obtenus après les parties « preprocessing » et « Classifieur » afin de les rendre lisibles et compréhensibles par tous.

On utilisera comme base de code les fonction et classes fournies dans le Tp2 sur les graphes, pour cela on s'inspirera du code du fichier « zDataManager.py » auquel on apportera des modifications ultérieurement.

Les données utilisées pour cette partie seront celles en sortie de « classifieur », elles seront donc sous format binaire, soit le tableau analysé est un faux soit un vrai.

Méthode de Visualisation : Afin de visualiser les résultats (sous forme binaire) obtenus à l'aide de notre modèle de prédiction, nous utiliserons les matrices de confusion. Celles-ci nous permettent de comparer les données obtenues après modèle de prédiction avec les données déjà existantes. Cela permet de savoir si le modèle de prédiction fonctionne correctement.

La classification binaire permet de définir des règles pour classer différentes catégories d'éléments, de données dans deux groupes distincts. Cela se fait à partir de propriétés qualitatives ou de caractéristiques spéciales trouvé sur les données fournis. Ici nous l'utiliserons dans le cas de « pass or fail » pour attester que les spécifications données sont validées, c'est à dire vérifier que les peintures sont bien identiques.

On va donc avoir pour commencer la peinture originale, sur lequel les règles de classement seront définis, ensuite il nous faut tester la fiabilité de ses règles, sur un deuxième échantillon pour comparer les données obtenues.

On a différentes méthodes de visualisation binaire (un type de classification supervisé), classification binaire car on a 2 catégories prédéfinie exploitable avec un :

- Arbre de décision
- Forêt d'arbres décisionnels
- Réseaux de neurone
- Machine à vecteurs de support
- Régression logistique
- Modèle probit
- Réseau bayésien

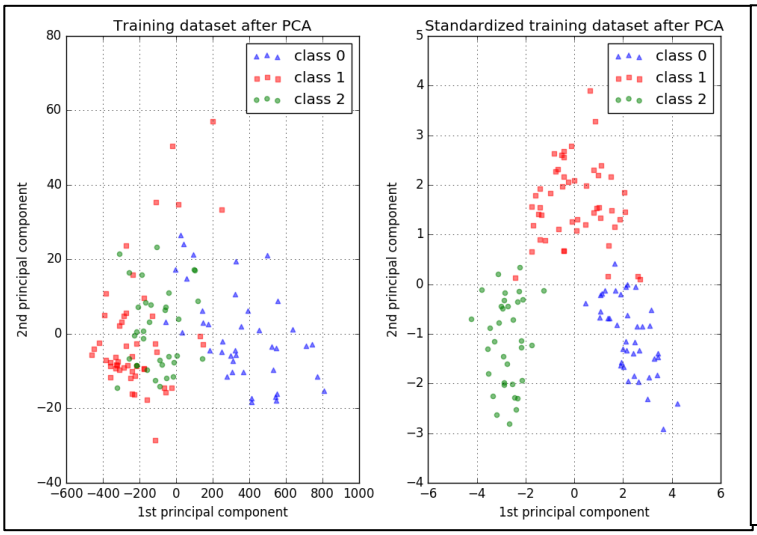
Chacun de ces classificateurs sera utilisé selon son domaine et les données fournis car certains fonctionnent mieux que d'autre selon les données fournis.

Figures

A) Standardisation des données - Figures explicatives Preprocessing

Nous aurons donc par exemple en entrée un jeu de donnée comme ceci :
Une array : [[34], [9], [27], ...]
Puis après application de la fonctions flatten :
Une nouvelle array : [34 9 27 ...] qui représente un vecteur avec 12 288 éléments.

Puis, on applique le PCA et standardise les données, ce qui devrait donner un résultat sous cette forme :



Exemple d'un modèle de données après avoir appliqué la standardisation et la classe PCA (tiré de la bibliothèque Scikit-Learn)[1]

Exemple de standardisation avec nos données :

Feature	**Valeur**
pixel_0_0_R	67
pixel_0_0_G	76
pixel_0_0_B	76
pixel_0_1_R	59
…	…
pixel_63_63_G	173
pixel_63_63_B	200

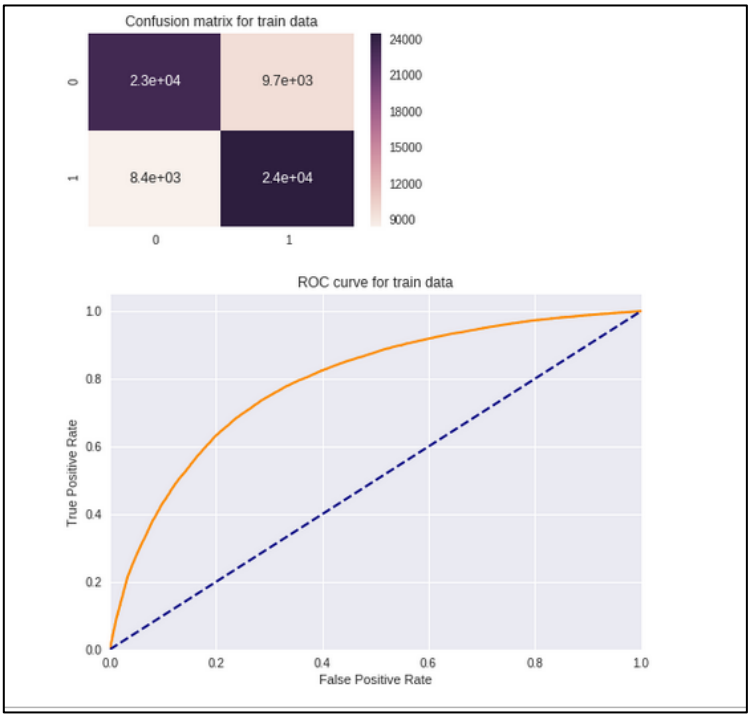
↓ Standardisation

Feature	**Valeur**
pixel_0_0_R	-0.93893845
pixel_0_0_G	-0.78090807
pixel_0_0_B	-0.78090807
pixel_0_1_R	-1.07940991
…	…
pixel_63_63_G	0.92230834
pixel_63_63_B	1.3963995

Ici, on prend la première image du set de validation comme exemple. On voit qu'une image est caractérisée par 12 288 valeurs correspondant à l'intensité de chaque couleur primaire pour chaque pixel.

Au départ, les valeurs sont situées entre 0 à 255, puis, lors de la standardisation, elles sont transformées de manière à ce que leur moyenne soit de 0 et leur écart-type de 1.

B) Représentation des résultats - Figures explicatives Visualisation



La représentation des résultats sera de la forme pourcentages d'erreurs, de vrai positifs et faux positifs (comme ci-contre). Les graphiques ci-contre sont tirés tu README du starting_kit_c1_final en prenant en compte les données « train ».

La première figure (celle du haut) est une matrice de confusion obtenue avec les résultats de « train ». Une matrice de confusion est un outil permettant de mesurer la qualité d'un système de classification.
Il nous sera donc très utile d'utiliser ce genre d'outils pour tester notre propre système.
Les deux cases foncées (1,1) et (0,0) correspondent aux prédictions qui se sont révélées justes. Les deux cases claires (1,0) et (0,1) correspondent aux mauvaises prédictions.

Le second graphique est une courbe représentant les vrais positifs (tableaux correctement détectés par le test) en fonction des faux positifs (tableaux déclarés positifs alors qu'ils ne l'étaient pas). De même, cet outil sera très pratique pour tester nos résultats dans la suite du projet.

Pour afficher les résultats, nous utiliserons les bibliothèques de programmation python comme « Numpy » et « MatPlotLib » (précédemment utilisées dans le cours de Python pour le calcul scientifique au semestre 3).

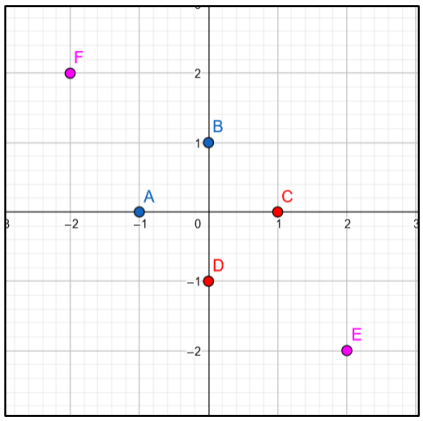
La bibliothèque « sklearn.metrics » nous permettra de créer des matrices de confusion grâce à la commande « confusion_matrix ».

```
>>> from sklearn.metrics import confusion_matrix
>>> y_true = [2, 0, 2, 2, 0, 1]
>>> y_pred = [0, 0, 2, 2, 0, 2]
>>> confusion_matrix(y_true, y_pred)
array([[2, 0, 0],
       [0, 0, 1],
       [1, 0, 2]])
```

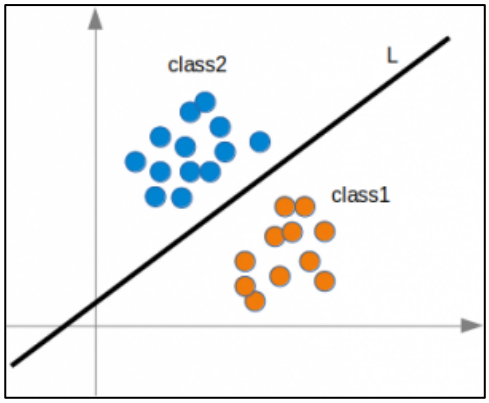
```
>>> y_true = ["cat", "ant", "cat", "cat", "ant", "bird"]
>>> y_pred = ["ant", "ant", "cat", "cat", "ant", "cat"]
>>> confusion_matrix(y_true, y_pred, labels=["ant", "bird", "cat"])
array([[2, 0, 0],
       [0, 0, 1],
       [1, 0, 2]])
```

Exemple d'utilisation des commandes de « sklearn.metrics » tiré de : https://scikit-learn.org/stable/modules/generated/sklearn.metrics.confusion_matrix.html

C) Représentation des tests Unitaires et la séparatrice - Figures explicatives du Classifieur



Représentation des points utilisés dans le test unitaire pour tester nos classifier. En bleu les points de Label 1 et en rouge les points de label 0. Les labels des points violets doivent être déterminé par le classifier. Si ces deux derniers est correct alors la méthode passée en paramètre fonctionne correctement.



La séparatrice comme son nom l'indique, est une droite qui permet de séparer nos différents échantillons. Les différents algorithmes de classification linéaire ont pour objectif de déterminer cette séparatrice de façon a ce qu'elle sépare au mieux les échantillons en fonction de la classe a laquelle ils appartiennent.

Pseudo-Code et algorithme

A) Pseudo-Code du Perceptron binaire et choix du model de Classification (Classifieur)

Voici le pseudo-code du perceptron binaire [3] que l'on pourrait essayer d'implémenter à la fonction main afin d'avoir un premier appuie de notre projet (ici w est le poids de chaque neurone, b le batch de donnée et y le label de chaque image) :

Algorithm 1: The perceptron algorithm.

Input: $X \in \mathbb{R}^{n \times d}$, $\mathbf{y} \in \{-1, 1\}^n$, $\mathbf{w} = \mathbf{0}_d$, $b = 0$, max_pass $\in \mathbb{N}$
Output: $\mathbf{w}, b, mistake$

```
for t = 1, 2, ..., max_pass do
    mistake(t) ← 0
    for i = 1, 2, ..., n do
        if y_i(⟨x_i, w⟩ + b) ≤ 0 then
            w ← w + y_i x_i                    // x_i is the i-th row of X
            b ← b + y_i
            mistake(t) ← mistake(t) + 1
```

Et voici le pseudo code de la méthode qui permet de choisir le modèle de classification (inspiré du fichier Model.py):

choixModel = 0	// Choix du model (0 pour la régression logistique, 1 pour le classifieur gaussien, 2 pour l'arbre de décision, 3 pour forets d'arbre décisionnel, Plus proche voisin sinon)

Fonction init :
Classifier.is_trained = Faux
Si choixModel == 0 :
  Classifier.model = LogisticRegression(parametres)
Sinon si choixModel == 1 :
  Classifier.model = GaussianNB(parametres)
Sinon si choixModel == 2 :
  Classifier.model = DecisionTreeClassifier(parametres)
Sinon si choixModel == 3 :
  Classifier.model = RandomForestClassifier(parametres)
Sinon choixModel :
  Classifier.model = KNeighborsClassifier(parametres)
	// Fonction init : qui a pour but d'initialiser les paramètres de notre classe Classifier (en particulier d'initialiser notre attribut model en fonction de la valeur de choixModel. L'attribut model contiendra une classe de la bibliothèque sklearn qui encapsule le classifieur choisit)

Fonction fit(X,Y) :
Classifier.is_trained = Vrai
Classifier.model = Classifier.model.fit(X,Y)
 	
Fonction predict(X) :
Y = Classifier.model.predict(X)
Retourner Y
	// Fonction fit : qui a pour but d'entrainer le classifieur. Elle prend en paramètres des données d'entrainement et leur label, et ajuste les paramètres du classifieur pour avoir les meilleurs résultats possibles pour ces données. On utilise pour cela la méthode fit de la classe de notre model (implémenté dans la bibliothèque sklean).
	
	// Fonction prédict : Qui a pour but de prédire la classe de la donnée X a l'aide de la méthode predict de la classe de notre model.
	
	(toutes les classes qui encapsule les classifieurs plus haut possède une méthode fit et une méthode predict)

B) Pseudo-Code de la partie préprocessing

```
import numpy as np
from sys import argv
from zDataManager import DataManager
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
from sklearn.base import BaseEstimator

class Preprocessor(BaseEstimator):

def  __init__(self):                          //Une instance de PCA est initialisée avec
   self.transformer=PCA(n_components=200)     //n_components égal à 200, signifiant
                                              //qu'elle peut réduire un vecteur à 200
                                              //éléments.

def standardizing(self, X)
nbImg = len(X)                                //Retourne les données standardisées
new_X = X.flatten().reshape(nbImg, 12288)     //après les avoir « aplatit » en vecteurs de
scaler = StandardScaler()                     //12288 éléments.
   return scaler.fit_transform(new_X)
```

```
def fit(self, X, y=none):
new_X = self.standardizing(X)
fit_X = np.zeros(len(X))
   for I in range len(X):
fit_X[i] = self.transformer.fit(new_X[i])
   return fit_X
```

//Appelle la fonction *fit* sur les données
//après les avoir standardisées, et retourne
//le résultat.

```
def fit_transform(self, X, y=none):
new_X = self.standardizing(X)
ft_X = np.zeros(len(X))
   for I in range len(X):
ft_X[i] = self.transformer.fit_transform(new_X[i])
   return ft_X
```

//Appelle la fonction *fit_transform* sur les
//données après les avoir standardisées, et
//retourne le résultat.

```
def transform(self, X, y=none):
new_X = self.standardizing(X)
transform_X = np.zeros(len(X))
   for I in range len(X):
transform_X[i] =self.transformer.transform(new_X[i])
   return transform_X
```

//Appelle la fonction *transform* sur les
//données après les avoir standardisées, et
//retourne le résultat.

```
if __name__==”__main__”:

   if len(argv) == 1:
input_dir = “../public_data”
   else:
input_dir = argv[1];

   Print(***Original Data***)
   Print(D)

Prepro = Preprocessor()
```

//Teste la classe sur des données prédéfinies.

//Les données seront prises dans le
//répertoire dont le chemin est passé en
//argument ou « ../public_data » si aucun
//argument n'est donné.

//Affiche les données originales pour pouvoir
//les comparer avec les données après le
//*preprocessing*.

basename = “Paintings” input_dir = “../public_data” D = DataManager(“Paintings”, input_dir)	//La classe DataManager est utilisée pour //obtenir les images sous forme de vecteurs
size_train = D.data[‘X_train’].shape[0] size_valid = D.data[‘X_valid].shape[0] size_test = D.data[‘X_test].shape[0]	
D.data['X_train'] = Prepro.fit_transform(D.data['X_train'], D.data['Y_train']) D.data['X_valid'] = Prepro.fit_transform(D.data['X_valid']) D.data['X_test'] = Prepro.fit_transform(D.data['X_test']) D.feat_name = np.array(['PC1', 'PC2']) D.feat_type = np.array(['Numeric', 'Numeric'])	//Les données sont transformées par le //*preprocessing*

failed = false	//Déclare *failed*, un variable de type booléen
	//qui vérifie si la classe a passé les tests ou
	//non

if (D.data[‘X_train’].shape[0] != size_train or D.data[‘X_valid].shape[0] != size_valid or D.data[‘X_test’].shape[0] != size_test):
failed = true
Print(“Taille altérée par le preprocessing”)
	//Vérifie que le nombre d’images est resté
	//inchangé aprés le *preprocessing* et donné
	//la valeur *true* à *failed* sinon, et affiche un
	//message pour le signaler

if (D.data[‘X_train’].shape[1] != 200 or D.data[‘X_valid’].shape[1] != 200 or D.data[‘X_test’].shape[1] != 200):
failed = true
Print(“Problème de reduction de la dimensionnalité”)
	//Vérifie que les données sont bien des
	//vecteurs à 200 éléments et donne la valeur
	//*true* à *failed* sinon, et affiche un message
	//pour le signaler

if(failed == false):
For i in range(len(D.data[‘X_train’][0])):
Print(D.data[‘X_train’][0][i])
Print(“Test réussi”)
	//Si *failed* a pour valeur *false* alors :
	//On affiche les éléments de la première
	//image pour observer si elles ont bien été
	//standardisées et on affiche « Test réussi »

C) Pseudo-Code méthode test unitaire du Classifieur

def tesUunitaireClassifier(typeDeClassifier):
  X = [[0, -1], [1, 0], [0, 1], [-1, 0]]
  Y = [0,0,1,1]
  clf = typeDeClassifier
  clf = clf.fit(X, Y)
  assert clf.predict([[-2.,2.]]) == np.array([1])
  assert clf.predict([[2.,-2.]]) == np.array([0])
	// On déclare la méthode
	// On déclare un tableau de points
	// On déclare les labels correspondants
	// On appel la méthode de classification
	// On entraine le modèle
	// On vérifie que le modèle est bon

D) Pseudo-Code matrice de confusion + Test Unitaire

Fpr = false positive rate
Tpr = true positive rate

from sklearn import metrics
import matplotlib.pyplot as plt
import numpy
import pandas as pd
 	
plot_conf_matrix (solution, prediction, title) :
 	
cm = confusion_matrix(solution,prediction)
df_cm =pd.DataFrame(cm, index, columns)
 	
sn.heatmap(df_cm, annot = True)
plt.title(title)
plt.show()
	
	// on définit une fonction prenant en argument « solution »(les étiquettes de données sous forme de tableau), « prediction »(les étiquettes renvoyées par le modèle de prédiction), « title »(un titre, chaine de caractère)
	// La fonction affiche une matrice de confusion
	//definition de la matrice de confusion
	//methode permettant de créer un tableau bidimensionnel
	//coloration de la matrice
	//Attribution d'un titre à la matrice
	//Affichage de la figure

Pour le test unitaire (que l'on a exécuté avec « train » pour l'instant) on se propose d'utiliser une fonction fournie dans le squelette du projet).

import numpy as np
 	
def conversion (prediction):
  tab = []
  for i in prediction:
  if(i >= 0.5):
  tab.append(1)
	
	// cette fonction permet de convertir les données à valeurs réelles de prediction en valeurs binaires. Prend en argument un tableau prediction issu du modele de prédiction.

else:
tab.append(0)
return tab
	// pour ceci on definit un seuil (0.5), pour une donnée au-dessus de ce seuil on ajoute 1 au tableau tab sinon on ajoute 0.
	
	//on renvoie le tableau tab

E) Pseudo-Code Visualisation (en plus de la matrice de confusion)

import matplotlib.pyplot as plt
import numpy
 	
def plot_fpr_tpr (fpr, tpr, title):
 	
plt.figure()
plt.plot(fpr,tpr,color)
plt.label()
plt.title(title)
plt.show()
	// fpr = false positive rate (taux de faux positifs)
	// tpr = true positive rate (taux de faux positifs)
	
	// definition d'une fonction permettant de tracer un graphique représentant tpr en fonction de fpr
	
	//on créé la courbe
	//on définit les labels en abscisse et ordonnée
	//Attribution d'un titre à la courbe
	//Affichage de la figure

Table 1: Statistics of the data

Dataset	Num. Examples	Num. Variables/Features	Sparsity	Has category variables ?	Has missing data ?
Training	65856	200	50.03%	Non	Non
Valid(ation)	9408	200	50.23%	Non	Non
Test	18817	200	50.00%	Non	Non

Table 2: Preliminary results

Method	NaiveBayes	Linear Regression	Decision Tree	Random Forest
Training (roc_auc_score)	0.587	0.725	1.00	0.989
CV	0.59 (± 0.01)	0.79 (± 0.00)	0.57 (± 0.00)	0.61 (± 0.01)
Validation	0.57	0.71	0.56	0.60

Tableau obtenu montrant les performances des différents algorithmes de classification selon trois scores : le AUC du training, le score du Cross validation et le score de validation obtenus sur Codalab.

Annexe

#	SCORE	FILENAME	SUBMISSION DATE	STATUS	✔	
1	0.5671419649	Khalo-Decision Tree.zip	02/23/2019 16:34:54	Finished	✔	+
2	0.7172266147	Khalo-Linear Regression.zip	02/23/2019 16:37:10	Finished		+
3	0.6050979235	Khalo-Random Forest.zip	02/23/2019 16:40:39	Finished		+
4	0.575694892	Khalo-GaussianNB.zip	02/23/2019 16:42:16	Finished		+

Score de validation obtenu sur Codalab grâce à nos différents classifieur.

Références

- [1] : Modules et fonctions de Scikit : https://scikit-learn.org/stable/index.html
- [2] : Modules et fonctions de Numpy : http://www.numpy.org/
- [3] : Image prise du site Stack OverFlow : https://stackoverflow.com/questions/46376010/doubts-regarding-this-pseudocode-for-the-perceptron-algorithm
- [4] : Cours d'introduction à la vie artificielle d'Aurélien Decelle : https://www.lri.fr/~adecelle/dokuwiki/doku.php?id=enseignement
- [5] : Cours sur le ROC publié par google : https://developers.google.com/machine-learning/crash-course/classification/roc-and-auc