

Groupes administratif : 2 et 4

Nom de groupe : Vinci

Membres du groupe : AMBEMOU Loba, TALSSI Mehdi, Pieprzyk Floriana,
JOUET Edouard, TRINH Donovan, LEVY Kevin, VERANI Damien

URL challenge : <https://codalab.lri.fr/competitions/401>

Repo GitHub du projet : <https://github.com/vinci232/vinci232.git>

N° de soumission : 7497

URL vidéo YouTube :

<https://youtu.be/o26p1jgoDDY>

URL des diapos :

<https://github.com/vinci232/vinci232/blob/master/PersodataVinci-Diapo.pdf>

Sommaire :

- I. Introduction (page 2)
- II. Preprocessing (page 3)
- III. Prédiction (page 4)
- IV. Visualisation (page 6)
- V. Discussion et Conclusion (page 8)
- VI. Bibliographie (page 9)

Introduction :

Dans le cadre de l'UE « Mini-Projet » de 2e année de Licence d'Informatique à l'Université Paris Sud, plusieurs challenges nous ont été proposés à relever, sous forme de projet complet à réaliser. Nous avons choisi le challenge PersoData with preprocessed image car il est plus simple de commencer par travailler sur des données prétraitées. L'objectif est de pouvoir détecter une fausse peinture par classification des données. Ce challenge est intéressant car de nos jours les ordinateurs sont très performants et d'autant plus capables de générer de fausses peintures qui pourraient induire en erreur des experts en art. Ces œuvres créées par des intelligences artificielles pourraient être mises en circulation comme étant des œuvres authentiques, il est donc primordial de pouvoir les authentifier. Nous nous retrouvons donc face au problème suivant : Est-il possible de créer un model pouvant authentifier une image de manière fiable? Nous travaillons donc avec deux classes, une correspondant aux vraies images et l'autre aux fausses. Ces classes contiennent plus de 65000 images représentées chacune par 200 features. Nous avons implémenté un code de preprocessing qui a pour objectif de faciliter la classification des données. Nous avons également à notre disposition un starting kit contenant un classifieur que nous avons modifié entraîné pour avoir de meilleurs résultats. Pour finir, nous avons eu à trouver une manière d'afficher les résultats de façon simple et efficace pour que nos résultats soient facilement interprétables.



Figure1:Exemple de fausses images(en haut) et des vraies images(en bas)

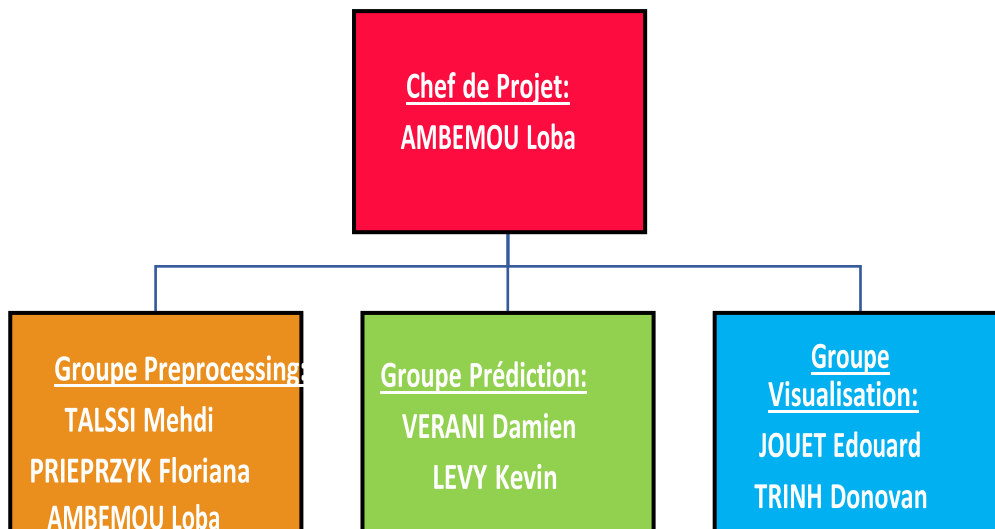


Figure 2: Organigramme de l'équipe VINCI

Preprocessing :

Cette partie consiste à transformer les données brutes pour pouvoir ensuite donner des données modifiées à l'algorithme du machine learning. On modifie les données brutes car cela réduit d'abord l'overfitting puisque que l'on réduit les données redondantes, ensuite, car cela améliore la précision car nous avons moins de données « trompeuse » et enfin car cela réduit le temps d'entraînement de la machine.

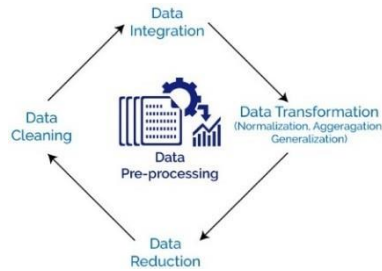


Figure 3: How data pre-processing works

Notre but étant de faciliter la classification des données, nous avons opté pour l'utilisation d'un PCA pour réduire le nombre de features représentant les images à 100. Pour cela nous avons initialisé un transformer avec un PCA en lui passant en paramètre le nombre de feature égale à 100. Ce même transformer va être utilisé dans les méthodes fit transform et fit_transform du preprocessing (cf. Annexe Code I).

Afin de tester le fonctionnement de notre PCA nous avons un teste parcourant le data_set et visant à savoir si des données ont été censurées(cf. Annexe Code II)

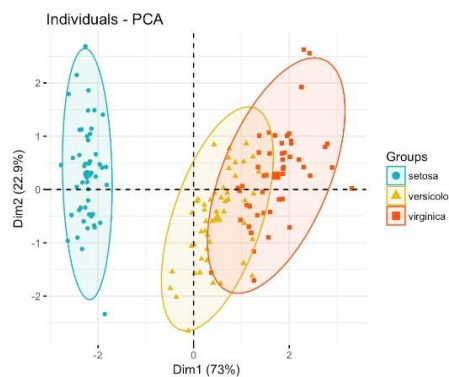


Figure 4: Exemples de l'action d'un PCA sur plusieurs groupes représenter par certaines features

Nous avons ensuite implémenté la méthode standard afin de standardiser et normaliser les données ce qui permet de transformer les données en se basant sur la loi normal pour les rendre plus similaires et les restreindre a un intervalle allant de zéro à un.

Nous avons défini la méthode fit_transform qui va rentrer appeler la méthode fit_transform de PCA sur les données passées en paramètre. Ces données seront ensuite standardisées grâce à un appel de standard sur les données réduite.(cf Annexe Code III).

Nous avons enfin créé un pipeline pour appliquer le preprocessing aux données et les classier en même temps. Pour ce faire, nous avons utilisé la méthode `make_pipeline` de `imbalanced-learn` car nous rencontrions des problèmes avec la méthode `pipeline` de `sklearn`. Nous l'avons donc téléchargé et mis dans le `strating_kit` sous le nom de `Pipeline`, puis nous l'avons utilisé dans `model.py` en lui passant en paramètre la méthode `fit_transform` de preprocessing et le classifieur choisis par le groupe de classification (à savoir le perceptron multicouches ,cf. Annexe Code V).

Pour finir nous voulions savoir quel était le meilleur nombre de feature pour le classifieur. Nous nous sommes donc servis d'une boucle `for` allant de 0 à 210 features avec un pas de 10, qui incrémente le nombre de features et affiche le score du classifieur en fonction du nombre de features (cf. annexe code VI).

Classification :

Le but de cette partie est de créer une intelligence artificielle capable de reconnaître une vraie d'une fausse image. Plusieurs contraintes et choix ont été fait par notre équipe afin d'optimiser au mieux notre modèle. Mais d'abord, une explication rapide du principe de l'entraînement d'une intelligence artificielle :

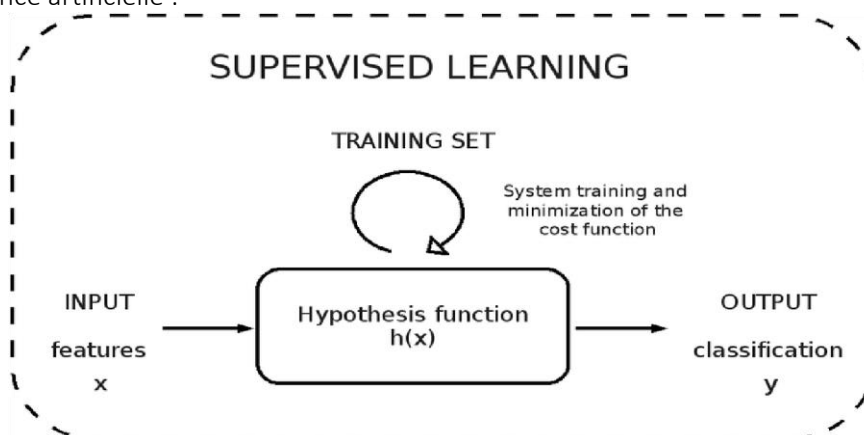


Figure 5: Principe de l'apprentissage supervisé

Le principe est de disposer d'un training set, un ensemble d'images dont on sait si elles sont vraies ou fausses. Nous laissons le modèle décider si l'image est fausse ou non, puis nous lui donnons la réponse. En lui donnant la réponse, le modèle « apprend » à détecter une image.

Le type de décision prises par l'IA dépend de sa nature. Ne sachant initialement pas quel type d'IA choisir, nous avons réalisé une boucle qui calcule les scores respectifs de plusieurs modèles, et nous avons sélectionné le meilleur.

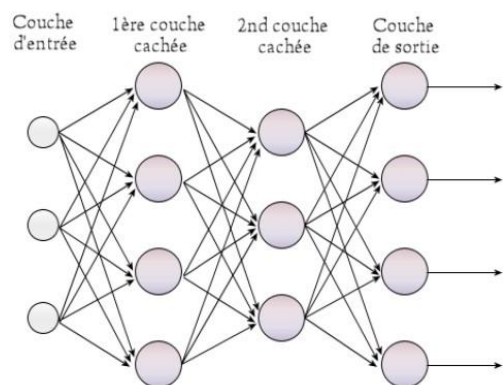
Les scores obtenus avec les différents modèles ont été répertorié dans le tableau ci-dessous.

	A	B	C	D
1	Classifieur	Score	CV	Rank
2	Neural Network	0.9344	0.83	1
3	Random Forest	0.9887	0.61	2
4	Decision Tree	0.9891	0.57	3
5	Linear Model	0.7251	0.72	4
6	Naive Bayes	0.5877	0.59	5
7	Gaussian Process	Err Memory	Err Memory	
8	SVM	X	X	
9	Radius Neighbours	X	X	

Figure 6: Tableau comparatif des différents modèles

Le meilleur modèle testé fut le MLP (Perceptron Multi Couches) (cf. Annexe Code VI).

Son principe est relativement simple : C'est un réseau de neurones artificiels (des noyaux décisionnels) disposés en plusieurs couches. Il dispose de plusieurs nœuds d'entrées, où nous mettons les features à classifier, de plusieurs nœuds de sortie, donnant la classification, et d'une série de couches cachées, constituées de neurones, sur laquelle seule l'IA a le contrôle.



Mais le Perceptron Multi Couches, comme n'importe quel modèle d'IA, dispose de ce qu'on appelle des hyperparamètres, des paramètres caractérisant le modèle que nous pouvons modifier pour rendre notre système de reconnaissance d'images le plus efficace possible. Pour bien choisir nos hyperparamètres, nous avons choisi une méthode dichotomique qui itère sur une sélection d'hyperparamètres (cf. Annexe Code VIII et IX).

	A	B	C	D
1			score	CV
2	Activation	Identity	0.6909	0,67 +- 0,03
3		Logistic	0.7466	0,72 +- 0
4		Tanh	0.7453	0,71 +- 0
5		Relu	0.928	0,84 +- 0,02
6	Solver	lbfgs	0.7151	0,69 +- 0,01
7		sgd	0.4995	0,50 +- 0,01
8		adam	0.9358	0,83 +- 0,03
9	learning_rate	constant	0.9358	0,83 +- 0,03
10		invscaling	0.9079	0,83 +- 0,02
11		Adaptive	0.9351	0,83 +- 0,01
12	Learning_rate_init	0.01	0.8087	0,75 +- 0,05
13		0.001	0.8949	0,83 +- 0,03
14		0.0001	0.9657	0,79 +- 0,01
16	Beta1	0.8	0.9598	0,80 +- 0,01
17		0.9	0.9657	0,79 +- 0,01

Figure 8: Tableau comparatif des hyperparamètres testés sur MLP

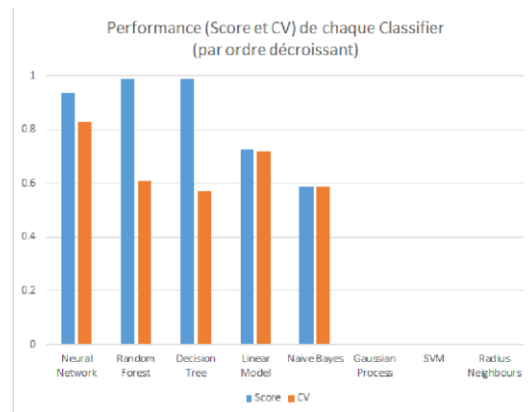


Figure 7: Diagramme de comparaison des modèles

Impact de différents hyperparamètres sur le Score de notre modèle

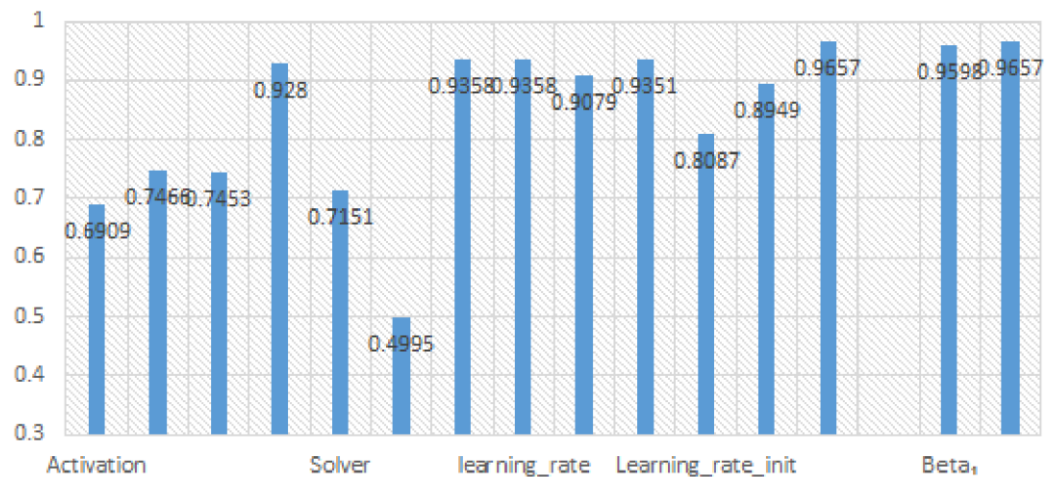


Figure 9: Diagramme représentant les scores obtenus avec chaque hyperparamètre

Visualisation :

La visualisation est une partie essentielle afin d'avoir un aperçu des données et une meilleure compréhension des résultats obtenus. Si les résultats sont présentés de façon concise et colorées, il sera plus facile pour toute l'équipe de comprendre les résultats et pouvoir les améliorer efficacement.

Nous avons notamment développé une Heat Map. Le principe de la heatmap est de pouvoir voir la différence de valeur (représentées par des couleurs) pour les valeurs de chaque features. Celle-ci permet de voir que pour les premières features, de nombreuses valeurs sont extrêmes (blanches, ou noires). Plus on s'approche des dernières features, et plus les résultats sont homogènes, aux alentours de 0 (les résultats se rapproche du rouge).

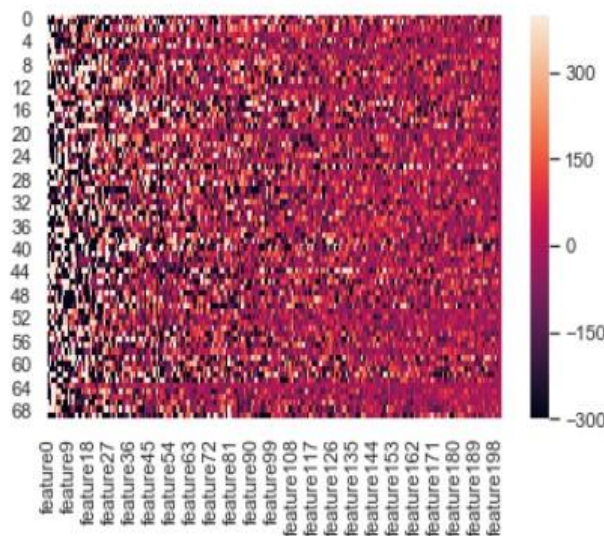


Figure10 :HeatMap

Pour la générer, nous utilisons la librairie seaborn, qui fera appel à la librairie Matplotlib. Pour avoir une HeatMap basique, il suffit de lui passer en paramètre les données, appelées "data" dans notre feuille jupyter, avec la commande suivante : `sn.heatmap(data)` ("sn" étant le raccourci de seaborn)

Seulement, cela seul ne suffit pas, car une erreur se produit : *seaborn TypeError: ufunc 'isnan' not supported for the input types, and the inputs could not be safely coerced to any supported types according to the casting rule "safe"*.

Pour corriger cela, on doit convertir chaque donnée en float (même si ce sont déjà des floats...). Pour cela, nous utilisons le fait que nos données soient en format Panda Data Frames, ce qui permet d'utiliser la commande : `data = data[data.columns].astype(float)`

Maintenant, nous avons une HeatMap qui s'affiche, seulement les axes ne sont pas du tout à la bonne échelle. En effet une échelle allant de -8000 à 8000 par défaut est bien trop grande, quasiment toutes nos valeurs semblent être aux alentours du 0. On va donc changer les paramètres vmin et vmax , où X et Y seront les limites de l'échelle. Pour obtenir une échelle correcte, il faut que la quantité de "case" (valeurs) noires ou blancs (les extrêmes) ne soient pas trop élevées. Il faut également que les couleurs de la heatmap ne soient pas trop homogènes..

Après avoir cherché manuellement, nous choisissons arbitrairement des valeurs de X=300 et Y=400. De cette façon, nous obtenons une HeatMap dont l'échelle va de -300 à 400, et offre un éventail de couleurs correcte par rapport au nombre de cases "aux valeurs extrêmes", noires ou blanches.

Les 2 lignes de codes finales sont donc :

```
#HeatMap avec Seaborn data =  
data[data.columns].astype(float)  
sn.heatmap(data, vmin = -300, vmax = 400)
```

L'utilisation d'une Confusion Matrix sur les train data montre que les résultats sont assez précis : 3.3E4 pour les faux négatifs et les bons positifs, contre seulement 4.5E2 et 1.3E2 pour les faux positifs et bons négatifs. Ceci signifie que les erreurs sur le train data sont de l'ordre de quelques pourcents.

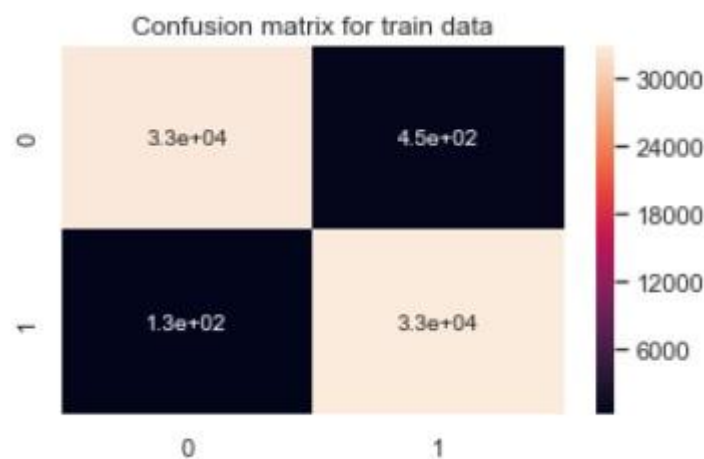


Figure 11 : Confusion Matrix 1

Nous utilisons aussi une ROC Curve qui a une très bonne allure, mais qui n'est pas encore parfaite.

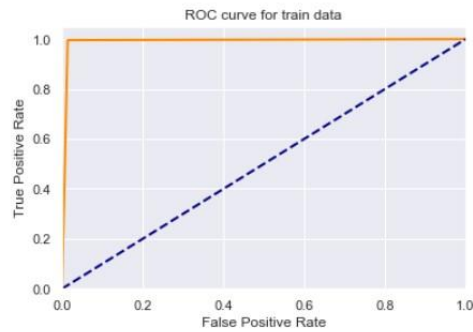


Figure 12 : ROC Curve for train data

Pour générer la ROC Curve et la Confusion Matrix, nous utilisons les fonctions initiales du README, à savoir "plot_ROC" et "plot_cm_matrix".

La fonction plot_ROC prend en paramètre le titre, false positive rate et true positive rate, puis fait appel à la librairie Matplotlib. Les paramètres sont pré-enregistrés par la fonction plot_ROC, puis cette dernière affiche la ROC Curve avec la commande plt.show().

La fonction plot_cm_matrix fonctionne de la même manière. Elle prend en paramètre un titre, les solutions, les prédictions, et affiche alors CM Matrix, en utilisant la librairie Matplotlib.

Résultats :

Nous avons remarqué que le preprocessing faisait baisser les performances de notre modèle. En effet, il s'avère que nous obtenons un score de 0.62 en prétraitant les données (cf. Annexe figure 21) alors que notre score est de 0.81 sans utiliser le preprocessing (cf. Annexe figure 22). Nous avons par conséquent pris la décision de ne pas faire appel au preprocessing pour garder un score correcte.

Nous ne sommes pas parvenus à générer avec Jupyter de ROC Curve et de Confusion Matrix sur les résultats. Nos courbes correspondent aux train data.

Discussion et conclusion :

Notre participation à ce mini projet nous a appris beaucoup de choses. Certains ont appris à coder en python, d'autres ont appris les bases du machine learning. Il fut très important de savoir être autonome notamment dans la recherche de compléments d'informations pour mieux avancer dans le projet. Le challenge Persodata nous a permis de mieux comprendre comment fonctionne la gestion et le déroulement d'un projet informatique. Tout au long de ce projet nous avons pu nous familiariser avec plusieurs aspects du développement informatique : le développement, le partage des tâches, le respect des deadlines, mais aussi et surtout, la gestion de projet et le travail en équipe.

Nos résultats ne nous permettent pas d'affirmer qu'un modèle peut authentifier une image avec une fiabilité de 100%, mais nous avons obtenu par classification un score de 0.81 ce qui est tout de même proche de 1. De plus avec un preprocessing plus efficace nous aurions pu avoir un score plus élevé ce qui aurait augmenté la fiabilité de notre modèle. Nous aurions pu avec un peu plus de temps nous pencher sur l'utilisation d'une LDA ou encore d'un réseau neuronal convolutif afin

d'augmenter l'efficacité de notre model. Tous cela montre bien qu'un model peut s'avérer fiable dans la détection de fausses peintures.

Bibliographie :

Figure 3 : Electronicsmedia, *What do you mean by Data preprocessing and why it is needed?* [en ligne].

Disponible sur : <https://www.electronicmedia.info/2017/12/20/what-is-data-preprocessing/>
[consulté le 01 avril 2019]

Figure 4 : Sthda, *Articles - Méthodes des Composantes Principales dans R: Guide Pratique* [en ligne].

Disponible sur : <http://www.sthda.com/french/articles/38-methodes-des-composantes-principales-dans-r-guidepratique/73-acp-analyse-en-composantes-principales-avec-r-l-essentiel/>
[consulté le 29 mars 2019]

Figure 5 : Nadège Langet, *Principe de l'apprentissage supervisé : schéma d'une unité logistique* [en ligne].

Disponible sur : https://www.researchgate.net/figure/2-Principe-de-lapprentissage-supervise-schema-dune-unite-logistique_fig7_280735747 [consulté le 08 avril 2019]

Annexe :

```
15
16
17 class Preprocessing(BaseEstimator):
18
19     def __init__(self, nbr_components=100):
20
21         self.transformer = PCA(n_components=nbr_components) #on veut reduire le nombre de composants a 100
22
23
24     def standard(self, X, Y=None):
25         X_transf=preprocessing.StandardScaler().fit(X)
26         return X_transf
27
28     def fit(self, X, Y=None):
29         transf=self.transformer.fit(X, Y)
30         return transf
31
32     def fit_transform(self, X, Y=None):
33
34         result=self.transformer.fit_transform(X)
35         return self.standard(result)
36
37     def transform(self, X, Y=None):
38         return self.transformer.transform(X)
39
```

Figure 13 Annexe Code1

```
if __name__=="__main__":
    ...
    fichier_data = "public_data/perso_train.data"#on va chercher les données d'entraînement
    datas = np.loadtxt(fichier_data, sepateur=" ")#on va chercher les données d'entraînement, chaque donnée est delimité par " "
    prepro = preprocessing()
    Tab = prepro.fit(datas)#on preprocess des données
    transf_Tab = prepro.transform(Tab)
    #on verifie que le preprocessing a agit sur les données, c'est a dire s'il y a des données censuré
    censured_data=0
    for i in range(0,transf_Tab.lenth):
        if i == 1 :
            print(i, " est Censuré")
            censured_data+=1

    if censured_data!=0:
        return True
    ...
```

Figure 14 : Annexe Code II

```

15
16
17 class Preprocessing(BaseEstimator):
18
19     def __init__(self, nbr_components=100):
20
21         self.transformer = PCA(n_components=nbr_components) #on veut reduire le nombre de composants a 100
22
23
24     def standard(self, X, Y=None):
25         X_transf=preprocessing.StandardScaler().fit(X)
26         return X_transf
27
28     def fit(self, X, Y=None):
29         transf=self.transformer.fit(X, Y)
30         return transf
31
32     def fit_transform(self, X, Y=None):
33
34         result=self.transformer.fit_transform(X)
35         return self.standard(result)
36
37     def transform(self, X, Y=None):
38         return self.transformer.transform(X)
39

```

On applique la méthode fit_transform du PCA

Puis on standardise les données sortis en résultat

Figure 15: Annexe code III

```

38 def __init__(self, nb_components=100):
39     """
40     This constructor is supposed to initialize data members.
41     Use triple quotes for function documentation.
42     """
43     self.num_train_samples = 65856
44     self.num_feat = 100
45     self.num_labels = 2
46     self.is_trained = False
47     self.preproc = Preprocessing.Preprocessing()
48     self.model = neural_network.MLPClassifier(activation = 'relu', solver = 'adam', learning_rate = 'adaptive', learning_rate_init = 0.00005, beta_1 = 0.9, beta_2 = 0.999, max_iter = 500, epsilon =
49
50 def fit(self, X, y):
51     """
52     This function should train the model parameters.
53     Here we do nothing in this example...
54     Args:
55     X: Training data matrix of dim num_train_samples * num_feat.
56     y: Training label matrix of dim num_train_samples * num_labels.
57     Both inputs are numpy arrays.
58     For classification, labels could be either numbers 0, 1, ... c-1 for c classe
59     or one-hot encoded vector of zeros, with a 1 at the kth position for class k.
60     The AutoML format support on-hot encoding, which also works for multi-labels problems.
61     Use data_converter.convert_to_num() to convert to the category number format.
62     For regression, labels are continuous values.
63     """
64     """
65     """
66     #This is the code given in example :
67
68     self.num_train_samples = X.shape[0]
69     if X.ndim>1: self.num_feat = X.shape[1]
70     print("FIT: dim(X)= [{:d}, {:d}]".format(self.num_train_samples, self.num_feat))
71     num_train_samples = y.shape[0]
72     if y.ndim>1: self.num_labels = y.shape[1]
73     print("FIT: dim(y)= [{:d}, {:d}]".format(num_train_samples, self.num_labels))
74     if (self.num_train_samples != num_train_samples):
75         print("ARRGH: number of samples in X and y do not match!")
76     self.is_trained=True
77     self.model = self.model.fit(X, y)
78     """
79
80     self.Pipe=imbPipeline.make_pipeline(['Preprocessing', self.preproc.fit_transform], ['clf', self.model.fit])
81     self.Pipe.fit(X, y)
82     print(self.Pipe.score(X, y))
83

```

Création du pipeline

Figure 16 : Annexe Code V

```

M = model()
P = Preprocessing()
D = DataManager('perso', 'public_data', replace_missing = True)

# Code for testing different features number values for the PCA

features_number = []
pca_scores = []
for i in range(0, 210, 10):
    X_train = D.data['X_train']
    Y_train = D.data['Y_train']

    P.transformer = PCA(n_components = i)
    X_train = P.fit_transform(X_train, Y_train)
    M.model.fit(X_train, Y_train)

    temp_score = M.model.score(X_train, Y_train)
    features_number.append(i)

    pca_scores.append(temp_score)
    print("features = ", i, "; score = ", temp_score)

plt.plot(features_number, pca_scores, 'r+')
plt.show()

# Code for testing different classifiers.
# We will keep the classifier with the highest score, with their default hyperparameters

X_train = D.data['X_train']
Y_train = D.data['Y_train']

scores = {}

classifier_names = [
    "decision_tree",
    "logistic_regression",
    "random_forest",

```

Figure 17 : annexe Code VI

```

99
100 # Code for testing different classifiers.
101 # We will keep the classifier with the highest score, with their default hyperparameters
102
103
104 scores = {}
105
106 classifier_names = [
107     "decision_tree",
108     "logistic_regression",
109     "random_forest",
110     "naive_bayes",
111     "gaussian_process",
112     "neural_network",
113     "svm",
114     "radius_neighbors"]
115
116 classifiers = [
117     tree.DecisionTreeClassifier(),
118     linear_model.LogisticRegression(),
119     ensemble.RandomForestClassifier(),
120     naive_bayes.GaussianNB(),
121     gaussian_process.GaussianProcessClassifier(),
122     neural_network.MLPClassifier(),
123     svm.SVC(),
124     neighbors.RadiusNeighborsClassifier()]
125
126 # This block is commented after its first execution.
127 # Indeed, we only need it once in order to give us the best classifier to use.
128 # But since it has to build a whole model and fit it at each iteration, it is really long to use.
129 # We first use in order to get the best classifier, then we comment it in order to gain in efficiency.
130
131 for i in range(len(classifiers)):
132     self.model = classifiers[i]
133     self.model.fit(X, y)
134     scores[classifier_names[i]] = self.model.score(X, y)
135 print(scores)
136

```

Figure 18: annexe code VII

```

144 # Now, let's fine tune the hyperparameters of this classifier
145 # We refer you to the classifier's documentation:
146 # https://scikit-learn.org/stable/modules/generated/sklearn.neural\_network.MLPClassifier.html
147 # We are going to use a dichotomy algorithm on specific hyperparameters that we selected
148
149 hyperparameters = {
150     'activation' : ['identity', 'logistic', 'tanh', 'relu'],
151     'solver' : ['lbfgs', 'sgd', 'adam'],
152     'learning_rate' : ['constant', 'invscaling', 'adaptative'],
153     'learning_rate_init' : 'float',
154     'beta_1' : 'float',
155     'beta_2' : 'float',
156     'max_iter' : 'int'}
157
158 # Same as before, after the first run, we took the best hyperparameters and commented the whole thing.
159 # We keep this code block here for reference, and for you to understand how we fine-tuned our HPs.
160
161 for i in range(len(hyperparameters['activation'])):
162     self.model = neural_network.MLPClassifier(activation = hyperparameters['activation'][i])
163     self.model.fit(X, y)
164     print("activation " + hyperparameters['activation'][i] + " : " + self.model.score(X, y))
165
166 for i in range(len(hyperparameters['solver'])):
167     self.model = neural_network.MLPClassifier(solver = hyperparameters['solver'][i])
168     self.model.fit(X, y)
169     print("solver " + hyperparameters['solver'][i] + " : " + self.model.score(X, y))
170
171 for i in range(len(hyperparameters['learning_rate'])):
172     self.model = neural_network.MLPClassifier(learning_rate = hyperparameters['learning_rate'][i])
173     self.model.fit(X, y)
174     print("learning_rate " + hyperparameters['learning_rate'][i] + " : " + self.model.score(X, y))
175
176 # Since these algorithms are really long to run, we only have time to make 5 iterations.
177 for i in range(5):
178     m = 0.0001
179     M = 0.001
180     s = 0.9657
181     S = 0.8949
182     # These HP values were ran by hand to initialize the dichotomy
183     self.model = neural_network.MLPClassifier(learning_rate_init = (m+M)/2)
184     s1 = self.model.score(X, y)
185     if (s1 > s) :
186         m = (m+M)/2
187         s = s1
188         best_value = m
189     else :
190         M = (m+M)/2
191         S = s1
192         best_value = M
193 print("learning_rate_init : " + best_value)
194

```

Figure 19: annexe code VIII

```


175
176 # Since these algorithms are really long to run, we only have time to make 5 iterations.
177 for i in range(5):
178     m = 0.0001
179     M = 0.001
180     s = 0.9657
181     S = 0.8949
182     # These HP values were ran by hand to initialize the dichotomy
183     self.model = neural_network.MLPClassifier(learning_rate_init = (m+M)/2)
184     s1 = self.model.score(X, y)
185     if (s1 > s) :
186         m = (m+M)/2
187         s = s1
188         best_value = m
189     else :
190         M = (m+M)/2
191         S = s1
192         best_value = M
193     print("learning_rate_init : " + best_value)
194
195 for i in range(5):
196     m = 0.8
197     M = 0.9
198     s = 0.9598
199     S = 0.9657
200     # These HP values were ran by hand to initialize the dichotomy
201     self.model = neural_network.MLPClassifier(beta_1 = (m+M)/2)
202     s1 = self.model.score(X, y)
203     if (s1 > s) :
204         m = (m+M)/2
205         s = s1
206         best_value = m
207     else :
208         M = (m+M)/2
209         S = s1
210         best_value = M
211     print("beta_1 : " + best_value)
212
213 # beta_2 fine-tuning took too long to run. We had to kill the process.
214 for i in range(5):
215     m = 0.0001
216     M = 0.001
217     s = 0.9657
218     S = 0.8949
219     # These HP values were ran by hand to initialize the dichotomy
220     self.model = neural_network.MLPClassifier(beta_2 = (m+M)/2)
221     s1 = self.model.score(X, y)
222     if (s1 > s) :
223         m = (m+M)/2
224         s = s1
225     else :
226         M = (m+M)/2
227         S = s1
228

```

Figure 20: annexe code IX

RESULTS						
#	User	Entries	Date of Last Entry	Prediction score ▲	Duration ▲	Detailed Results
1	reference2	1	01/24/19	0.9467 (1)	0.00 (1)	View
2	mokakill	10	03/03/19	0.8674 (2)	0.00 (1)	View
3	picasso	15	04/04/19	0.8674 (3)	0.00 (1)	View
4	Kahlo	3	03/03/19	0.8593 (4)	0.00 (1)	View
5	isabelleshao	28	04/05/19	0.8447 (5)	0.00 (1)	View
6	Zhen	1	02/18/19	0.7891 (6)	0.00 (1)	View
7	reference1	1	01/24/19	0.7807 (7)	0.00 (1)	View
8	reference0	1	01/24/19	0.7172 (8)	0.00 (1)	View
9	areal	2	01/24/19	0.7172 (8)	0.00 (1)	View
10	Monet	27	04/02/19	0.7070 (9)	0.00 (1)	View
11	vinci	19	04/03/19	0.6230 (10)	0.00 (1)	View
12	laetitiabenali	2	02/26/19	0.6054 (11)	0.00 (1)	View
13	NehzUx	5	02/17/19	0.5831 (12)	0.00 (1)	View
14	Yanis47S	6	04/05/19	-1.0000 (13)	0.00 (1)	View

Figure 21: Annexe X : Leaderboard (soumission de code avec preprocessing)



#	Submission Id	User	Phase	Date	Description	Prediction score	Duration	Likes	Downloads	Detailed results?
1	7497	vinci	Development Phase	Mar 20 2019		0.8131	0.0000	👍 2	📄 (2 dis)	• View detailed results
2	7508	picasso	Development Phase	Mar 20 2019		0.8581	0.0000	👍 1	📄 (2 dis)	• View detailed results
3	7751	Monet	Development Phase	Mar 26 2019		0.6768	0.0000	👍 1	📄 (3 dis)	• View detailed results
4	8068	picasso	Development Phase	Apr 04 2019		0.8674	0.0000	👍 0	📄 (2 dis)	• View detailed results
5	8031	Monet	Development Phase	Apr 02 2019		0.7070	0.0000	👍 0	📄 (0 dis)	• View detailed results

Figure 22:Annexe XI : Leaderboard (soumission de code sans preprocessing)

Annexe XII : Bonus 2 : Qu'est-ce que le sur-apprentissage ?

Un sur-apprentissage ou sur surinterprétation arrive lorsqu'un modèle est trop entrainer avec certaines données, au point qu'il s'adapte trop aux données d'apprentissage. La conséquence est que le modèle prédictif pourra donner de très bonnes prédictions sur les données d'entraînement, mais il prédira mal sur des données qu'il n'a pas encore vues lors de sa phase d'apprentissage. On dit que le model se généralise mal.

Le sur-apprentissage peut être identifié grâce à la cross validation.

Annexe XIII: Bonus 1 : Qu'est-ce que la Cross validation ?

C'est une méthode d'estimation de la fiabilité d'une méthode. Elle consiste à diviser le training set en n parties à peu près égales, à entrainer l'algorithme sur une des n partie puis effectuer les tests avec les n-1 autres parties. Chacune des parties du training set doit servir une fois à l'entraînement. Si la méthode est fiable, les résultats des n tests doivent être sensiblement similaires.

Annexe XIV : Bonus 3 : Métrique du challenge

Pour ce challenge nous utilisons la métrique AUC (aire sous la courbe ROC). C'est une valeur qui correspond l'intégralité de l'aire à deux dimensions située sous l'ensemble de la courbe ROC (par calculs d'intégrales). AUC fournit une mesure des performances de classification possibles. Les valeurs d'AUC sont comprises dans une plage de 0 à 1. Un modèle dont 100 % des prédictions sont erronées a un AUC de 0,0. Si toutes ses prédictions sont correctes, son AUC est de 1,0.

Annexe XV : Bonus Code de AUC :

```
def fpr_tpr(solution,
prediction):
    for i in
range(n_classes):
        fpr, tpr, _ = metrics.roc_curve(solution,
prediction)
        roc_auc = metrics.auc
c(fpr, tpr)
return (fpr,tpr)
```


Annexe XVI: tableaux bonus et diagrammes
explicatifs

Dataset	nombre d'exemples	Nombre de features	« sparsity »	Variables catégorielles	Données manquantes	Nombre d'exemples par classes(2 classes fake et real)
Trainnig	65856	200	0	0	0	Real :&32886 Fake :32970
Test	18817	200	0	0	0	Real :9485 Fake :9332
Validation	9408	200	0	0	0	Real :4670 Fake :4738

Tableau 1: Statistiques sur les données

	A	B	C	D
1	Classifier	Score	CV	Rank
2	Neural Network	0.9344	0.83	1
3	Random Forest	0.9887	0.61	2
4	Decision Tree	0.9891	0.57	3
5	Linear Model	0.7251	0.72	4
6	Naive Bayes	0.5877	0.59	5
7	Gaussian Process	Err Memory	Err Memory	
8	SVM	X	X	
9	Radius Neighbours	X	X	

Tableau 2 : comparatif des différents classifieurs

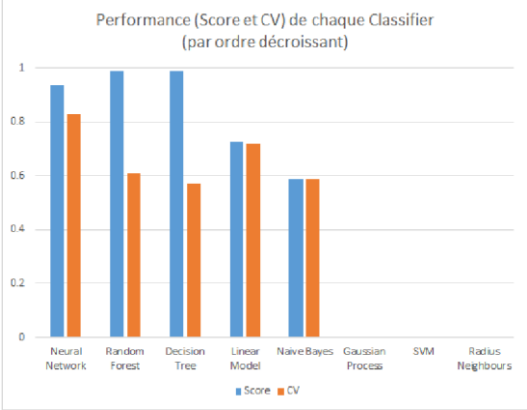


Diagramme de comparaison des hyper paramètres

	A	B	C	D
1			score	CV
2	Activation	Identity	0.6909	0,67 +- 0,03
3		Logistic	0.7466	0,72 +- 0
4		Tanh	0.7453	0,71 +- 0
5		Relu	0.928	0,84 +- 0,02
6	Solver	lbfgs	0.7151	0,69 +- 0,01
7		sgd	0.4995	0,50 +- 0,01
8		adam	0.9358	0,83 +- 0,03
9	learning_rate	constant	0.9358	0,83 +- 0,03
10		invscaling	0.9079	0,83 +- 0,02
11		Adaptive	0.9351	0,83 +- 0,01
12	Learning_rate_init	0.01	0.8087	0,75 +- 0,05
13		0.001	0.8949	0,83 +- 0,03
14		0.0001	0.9657	0,79 +- 0,01
16	Beta ₁	0.8	0.9598	0,80 +- 0,01
17		0.9	0.9657	0,79 +- 0,01

Tableau 3 : comparatif des hyperparamètres testés sur MLP

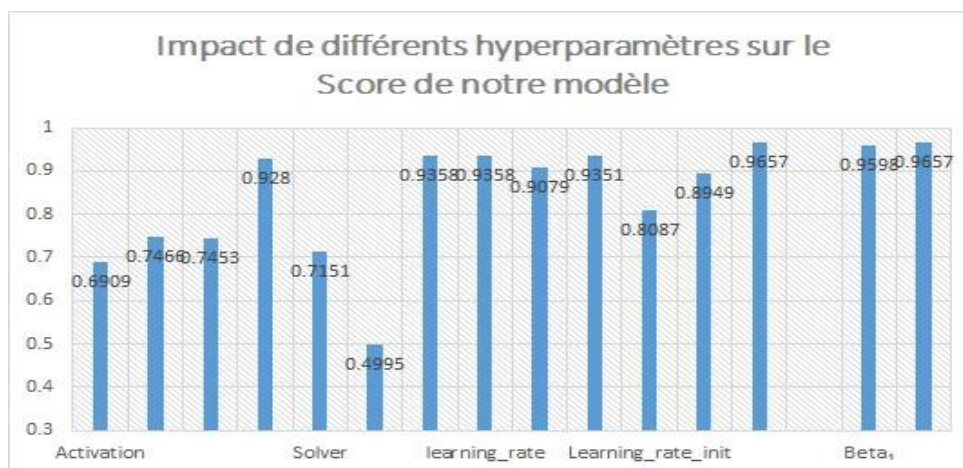


Diagramme représentant les scores obtenus avec chaque hyperparamètre