# Vraie ou Fausse Peinture ?

**L2 MI Mini Projet**
**Nom du groupe :** Monet
**Nom du groupe séance (M, V1, V2)** : V1
**Nom du Challenge :** Persodata
**Membre du groupe et Groupe administratif :**
Mbaraka Nasrine [Gp 1], Benali Laetitia [Gp 1], Rami Amel , ZHOU Shuyang [Gp 1b], Ferreira Alves Patricia [Gp 1b], Shao Isabelle.
**Url du challenge** : https://codalab.lri.fr/competitions/401#learn_the_details
**Numéro de la dernière soumission de CODE sur Codalab : 8810**
**Url du repository GitHub de l'équipe** : https://github.com/TeamMonet/starting_kit
**Url de la vidéo YouTube** : https://www.youtube.com/watch?v=wdAVT9AenJI
**Url des diapos de la présentation :**
**https://drive.google.com/file/d/1hi7YtaOMABnAJ6g_MEGbR1kGod6OipRF/view?usp=sharing**

**Motivation et Présentation :**

Une oeuvre d'art inédite totalement créé par un ordinateur respectant les couleurs, formes et courbes d'une vraie peinture et vendue à plusieurs millions de dollars, ça semble incroyablement irréel mais cela est vrai, l'intelligence artificielle et l'art forment à présent un duo indissociable qui a même donné naissance à WikiArt ou les amoureux de l'art peuvent admirer une oeuvre d'art loin des musées. Toutefois, il est impératif d'arriver à différencier un tableau fait par un programme informatique et une vraie peinture. Le but est d'arriver à les séparer avec le Challenge Persodata qui consiste à mettre en place un système de détection de fausses peintures, en effet il est de plus en plus difficile pour l'œil nu de faire la différence entre des peintures artificielles et des peintures réelles. Cela devient un réel enjeu puisque les potentiels acheteurs se disent : à quoi bon acheter un authentique tableau à des milliers d'euros quand on peut en avoir un identique pour un prix trois fois moins cher. A l'aide d'une classification binaire, qui est un apprentissage supervisé, nous allons pouvoir trancher sur la nature des peintures et **ne plus se faire avoir !**

# I. Qu'est ce le challenge Persodata ?

L'algorithme du challenge Persodata doit être capable de détecter si une peinture provenant de la collection WikiArt est l'œuvre d'un peintre ou bien générée par un programme informatique, dans notre cas par le GAN (Generative Adversarial Network). Nous faisons la version « preprocessed » : les images ont été prétraitées. En effet l'image brute de dimension 64 * 64 * 3 est prétraitée en tant que vecteur de 1 * 200 où nous extrayons 200 PCA *principal component analysis* une méthode de la famille de l'analyse des données, elle consiste à transformer des variables liées entre elles en nouvelles variables décorrélées les unes des autres. Ces nouvelles variables sont nommées " Composantes principales " ou axes principaux dans le but de réduire le nombre de variables et de rendre l'information moins redondante dans notre cas obtenir des images moins lourdes .

On compte 65856 d'images ou données sur les quels les classifieurs vont s'entrainer et ils vont être testés sur un échantillon de 18817 images différentes. Pour détecter un **sur apprentissage**(cf Annexe 2), on sépare les données en deux sous-ensembles, **l'ensemble de Cross-Validation**(cf Annexe 3), qui sert à vérifier la pertinence du réseau avec des échantillons qu'il ne connaît pas et compte 9408 images et l'ensemble de test .

En équipe, nous nous sommes réparties les 3 tâches par binômes, premièrement la partie Preprocessing ( *Ferreira Alves Patricia , Shao Isabelle*) consistant à transformer les différentes données en données pertinentes et facilement utilisable par le Classificateur ( *Mbaraka Nasrine, Benali Laetitia*) puis ce dernier va analyser les données d'entraînement pour pouvoir mener à bien sa tâche c'est-à-dire différencier les fausses et réelles enfin l'Interface graphique ( *Rami Amel , ZHOU Shuyang*) nous permet d'afficher des résultats de manière claire et pertinente voir comparer les données de départ et les résultats obtenus après traitement, pour tirer des conclusions sur la qualité de l'algorithme choisi.

# II. Description des algorithmes étudiés et pseudo-code

Au cours de notre projet on a du tester un bon nombre d'algorithme que ce soit pour la classification, pour le prepossessing ou la visualisation, nécessitant des comparaisons et des améliorations. Voici la manière dont nous en sommes venus à bout.

## i. Classification

Pour séparer les fausses peintures des vrais, nous avions principalement travaillé sur les 6 classifieurs [1][5] qui sont : *NaiveBayes or Gaussian classifier, SGDC Classifier, Random Forest, Decision Tree, Quadratic Discriminant Analysis (QDA) et MLPClassifier* (cf Annexe 2). Chaque classifieur à sa propre manière de trier les données en voici leurs fonctionnalité et singularité. Nous nous intéressons seulement à ce dernier ci-dessous car il donne les résultats les plus probants ( cf Annexe **Explication 2 :** Détails sur les classifieurs) :

- MLPClassifier (MLP) se base sur le Perceptron, une méthode d'apprentissage supervisée reprenant le modèle biologique de réseau de neurone, qui en plus ici est multicouche et utilisant la *rétropropagation du gradient* (technique consistant à corriger les erreurs selon l'importance des éléments qui ont justement participé à la réalisation de ces erreurs, en ajustant les poids synaptique).

Pour appliquer ces classifieurs à nos données il suffit de modifier le document Model.py comme ceci, on importe notre classifieur de sklearn :

| | |
|---|---|
| `from sklearn.neural_network import MLPClassifier` | _ Import le classifieur MLP de sklearn |
| `self.model = MLPClassifier(hidden_layer_sizes=(200,100,50,20), max_iter=1500,solver='adam', learning_rate='invscaling', activation='relu')` | Application du classifieur à notre modéle |

**Figure 6 Pseudo-code et code [4]**

Certes ces classifieurs effectuent bien leurs tâches et certains mieux que d'autres mais sachez qu'en ajustant judicieusement les hyperparamètres, c'est-à-dire les paramètres d'un classifieur qui augmentent son taux de réussite. Pour trouver les meilleurs paramètres nous avions testé deux méthodes qui sont la Random et Grid Search .

- Définition de Grid et Random Search: Ces méthodes consistent à effectuer des recherches sur un algorithme donné en lui fournissant un tableau d'hyper paramètres afin de trouver les meilleures combinaisons entre celle ci et ainsi obtenir des résultats plus satisfaisants.*(cf Annexe pour plus d'explication ces méthodes)*
- Différence entre les deux: La recherche aléatoire et la recherche par grille explorent exactement le même espace de paramètres. Le résultat dans les réglages de paramètres est assez similaire, tandis que le temps d'exécution de la recherche aléatoire est considérablement réduit. [3]

Dans notre cas nous avons appliqué ces deux outils d'optimisations sur le classifieur

MLPClassifier, tous les paramètres qui influencent l'apprentissage sont recherchés simultanément.[3]

- Les hyper paramètres, on a choisi de modifier que les paramètres ayant un impact sur la classification comme [6]:
  - activation = les fonctions d'activation des couches cachés
  - max_iter = nombre d'itération maximum à effectuer pour entrainer le perceptron
  - solver = optimisation du poids des synapses
  - learning rate = planning de rectification du taux d'apprentissage
  - hidden_layer_sizes = nombre de couches cachés (donc sans la première et la dernière couche)

Pour cela nous avions essayé deux méthodes principals qui sont la **Random Search et la Grid Search**. Ces dernières nous donne les meilleurs paramètres de façon plus ou moins rapide, qu'on a ensuite remplacé dans le model classifieur du model.py. A noter qu'on ne recherche pas simultanément sur autant de paramètres différents en utilisant la recherche sur grille, mais uniquement ceux qui sont jugés les plus importants.

Après avoir effectué une petite batterie de combinaisons en passant à chaque fois différents paramètres à notre algorithme qui d'ailleurs prenaient beaucoup de temps à s'exécuter à cause de la recherche de combinaisons (45min à 3h00), nous avons fait le choix de commenter ces codes une fois avoir obtenu les résultats qu'il nous fallait que nous avons par la suite intégré dans notre model.py [4].

Au final, nous sommes passés par différents chemins pour espérer avoir les meilleurs résultats, mais ce qui nous a essentiellement compliqué la tâche était le temps que nos algorithmes prenaient pour s'exécuter, pour des fois obtenir des erreurs ou bien des résultats non concluants, les alternatives ont donc été décrites plus haut (passer en mode manuel en choisissant les hyperparamètres pour plus de rapidité). cf Annexe 4 et 5 pour le **code et pseudo code** de **Random Search et la Grid Search** sur Random et MLP Classifier.

## ii. Preprocessing

Le but du preprocessing[9] est de choisir les données les plus pertinentes pour que la classification soit la plus efficace possible. Sur les données préprocessées ont été retenues les 200 features les plus intéressantes. Nous allons réduire à notre tour encore ces données.

- L'avantage du preprocessing est de pouvoir accélérer l'entraînement de l'intelligence artificielle, et d'éliminer les composants redondants ou les bruits.
- Le désavantage est de supprimer des données qui auraient pu être utiles, et donc de diminuer les performances de l'algorithme d'apprentissage.

**Explication PCA :** [8] Méthode permettant de réduire le nombre de composantes d'un jeu de données, de par une approche géométrique et statistique. On projette nos données contre les composantes principales et on étudie leur covariance. Plus la variance est élevé, plus la variable donne d'informations uniques, donc plus elle est importante. On garde donc les variables ayant les variances les plus élevées.

**Explication SelectKBest :** [16] Algorithme de sélection de features. On lui donne en paramètre une fonction et un nombre k et il nous renvoie les k features avec le score le plus élevé, selon la fonction. Comme notre problème est un problème de classification, on l'utilise avec la fonction f_classif de scikit learn, qui nous renvoie un score représentant l'efficacité des différentes features.

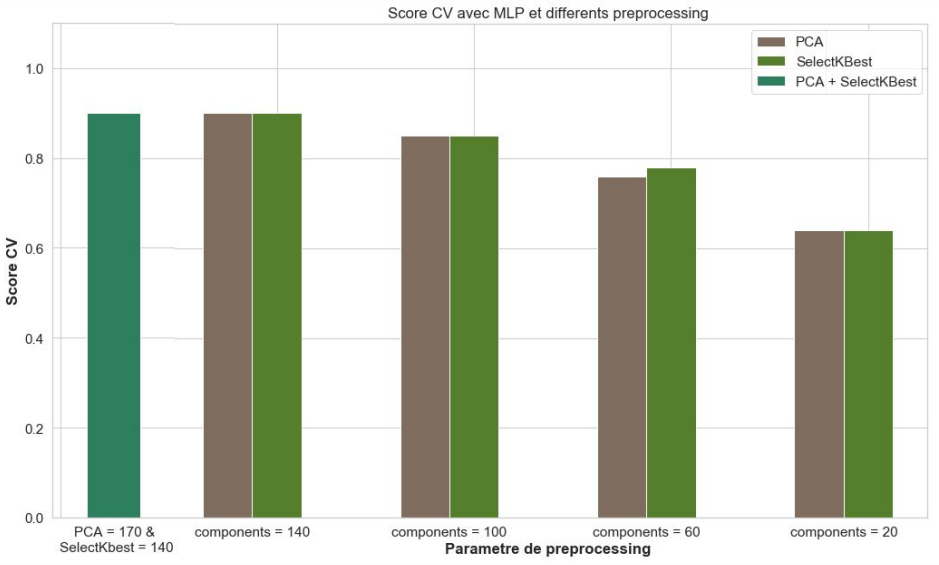



**Figure 10 :** Score CV avec MLP selon différentes méthodes de preprocessing

**Interprétation:** Pour preprocesser les données, les étudiants de master ont appliqué la méthode PCA (indiqué sur leur powerpoint de présentation du projet). Nous supposons donc que les features ont été déjà classé par ordre de variance plus élevée, au moins élevé. Du coup, toutes les données sont déjà significatifs. Ensuite, nous pensons que dû au faible nombre de features au départ dans ce challenge (200), faire une comparaison entre ces deux méthodes de préprocessing (ainsi que la combinaison des deux) n'était pas suffisamment significatif pour trouver une méthode qui se démarquerait de l'autre.

**Notre choix:** Nous étions parti dans un premier temps pour le PCA, puisque c'était la méthode la plus rapide, et qui donnait un résultat satisfaisant. Nous sommes parties d'une base de 200 components mais comment faire pour préserver autant que possible l'intégrité des données en faisant un deuxième preprocessing sans pour autant trop sacrifier la variance. Pour cela la visualisation a été notre amie.Comme vous pouvez le voir sur la Figure 7, nous voyons que la variance stagne entre 130 et 140 components. Nous avons donc décidé d'approcher au plus près cet arrêt, grâce à la visualisation. Nous avons donc déduit grâce à la Figure 8, que 133 était le nombre parfait de components avant de baisser trop notre variance, ce qui correspond à environ 0.98%.

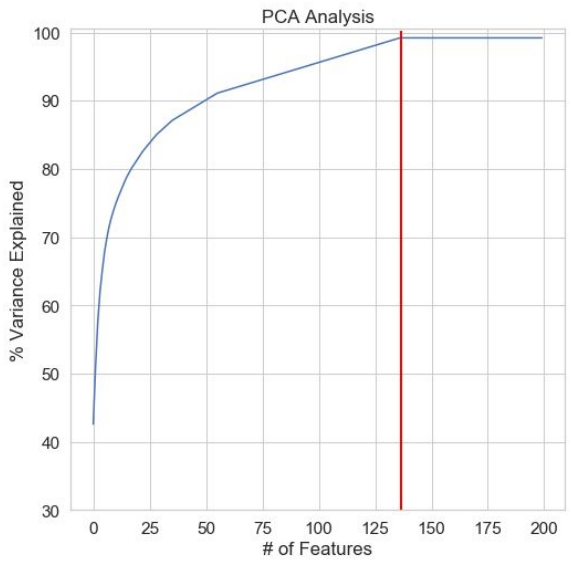



**Figure 7 :** Visualisation de la variance sur les 200 components au départ [2]

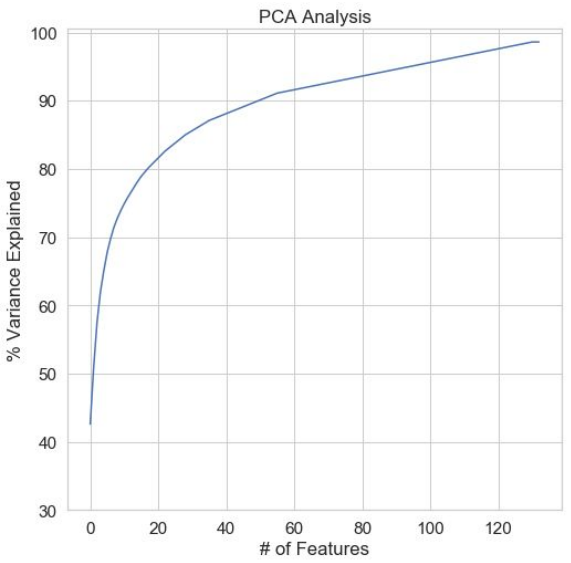



**Figure 8** : Visualisation de la variance sur 133 components à l'arrivée[2]

Cependant par la suite, nous avons finalement procédé, grâce au Pipeline [11], à un vote[10] du meilleur jeu de donnée avec différents preprocessing avec la méthode de classification, comme qui suit: self.model = clf = Pipeline([('preprocessing', Preprocessor()),('classification', méthode de classification du binôme Prediction)])

Le pipeline sert à assembler ces différentes étapes, et est bien pratique pour faire des tests tels quel *GridSearchCV*, *RandomizedSearch*, ou bien pour utiliser le *VotingClassifier* [10].

**Difficultés :** Le preprocessing (cf annexe 1) peut être utile pour se débarrasser de components inutiles. Or sur ces données préprocéssées par les étudiants en master, les données étaient toutes significatives. Cela se traduisait pour notre part par le fait que, quelque soit la méthode ou le pourcentage, nous obtenions un résultat inférieur aux données initiales du starting kit, sans notre preprocessing.

## iii. Visualisation

Il est essentiel dans tout travail traitant beaucoup de données de vouloir les visualiser puisqu'une image, transmet vite et résume bien une information, elle permet d'exploiter et d'expliciter de manière concise les données que le classifieur a pu traiter.

Nous avons pour notre travail des peintures réalisées artificiellement par une machine et de vraies peintures, réaliser la métrique d'évaluation autrement dit la courbe ROC (cf Annexe 6 et 7 ) nous permettra de distinguer les faux des vrais positifs, ainsi ce sera notre référence par rapport au code utilisé. Comme l'intérêt du challenge porte aussi sur l'utilisation des librairies classiques de Python comme scikitlearn, Pandas et SeaBorn , Panda Dataframe (cf Annexe 10 )Nous avons pu créer des diagrammes et courbes représentatives des 200 Features à notre disposition (Annexe 10).

Nous rechargeons les données avec la classe AutoML DataManager, puis nous allons dériver une classe de Sample_Data pour l'aperçu, cela consiste à convertir les données en trames de données et vous permet ensuite d'appeler toutes sortes de méthodes pour visualiser les données, en incluant la matrice de corrélation et de confusion (Figure 6). Nous avons aussi essayé de visualiser les résultats avec chacun des modèle de  préprocessing et classification utilisés, un bon choix de classification  était primordial pour atteindre l'un des trois scores de référence, sans que le classifieur n'apprenne des données et nous rendent un résultat peu crédible.

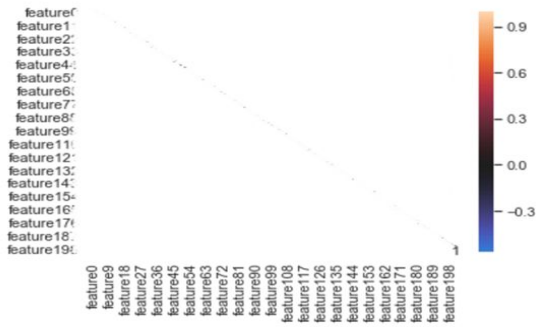


**Figure 3 :** la matrice de corrélation

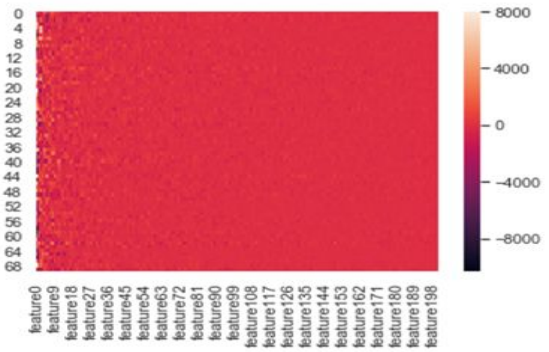


**Figure 6** : Corrélation avec caractéristique [2]

Cross validation Les nouvelles données n'ont pas accès aux labels Y_valid et Y_test pour auto-évaluer leurs validations et test performances. Mais la performance de la formation n'est pas une bonne prévision de la validation ou de la test performance. Dans ce cas, on évalue à l'aide de la cross validation. (cf. annexe 3)

Le training data sont divisées en plusieurs training / test folds, ce qui permet aux participants d'auto-évaluer leur modèle au cours du développement. Le résultat moyen du CV et 'intervalle

de confiance de 95% sont affichés. Dans notre cas, le score des résultats est présenté dans le tableau ci-dessous. On le réalise par appeler la fonction cross_val_score (annexe 9)

## Les méthodes pour visualiser les données :

Nous chargeons les données sous forme panda data frame. Nous pouvons donc utiliser les fonctions intégrées "pandas" et " seaborn " pour explorer les données. (Annexe 10)

Pandas DataFrame a une méthode corr qui calcule le coefficient de corrélation de Pearson (ou un autre) entre tous les couples de colonnes numériques de DataFrame. (figure 6)

# III. Résultats

Suite à la recherche de paramètre nous avons obtenu avec le MLP les trois combinaisons les plus intéressantes qui sont dans **l'Annexe 8.** Nous avons mené une baterie de recherche comme ci-dessus et voici notre résultat final :

| | |
|---|---|
| Parameters: {'solver': 'adam', 'max_iter': 2500, 'learning_rate': 'adaptive', 'hidden_layer_sizes': (100, 50, 100, 50, 20), 'activation': 'relu'} | Train score : 0.9969<br>CV score : 0.94 (+/- 0.01)<br>Test : |

et nous obtenons donc la matrice de confusion et la courbe ROC suivante :

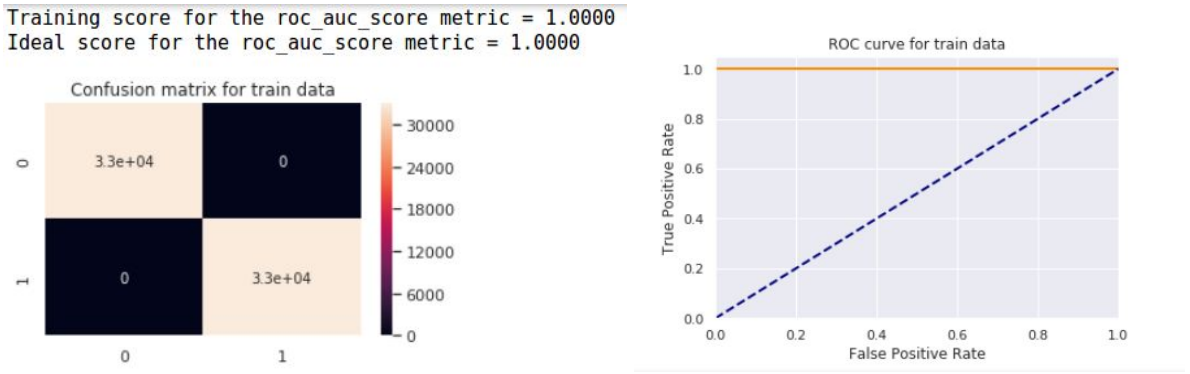



**Figure 1** : Matrice de confusion de train data et Courbe de ROC obtenus avec MLP Classifier [2]

Comme vous pouvez le voir notre algorithme classe parfaitement les données du Train en effet on a aucun faux négatif ni faux positifs ! Par conséquent la courbe ROC est parfaite. De plus la différence entre le score de la CV(cf Annexe 3) et du Test, étant si petite nous indique donc qu'il y'a pas eu de sur apprentissage(cf Annexe 6) .

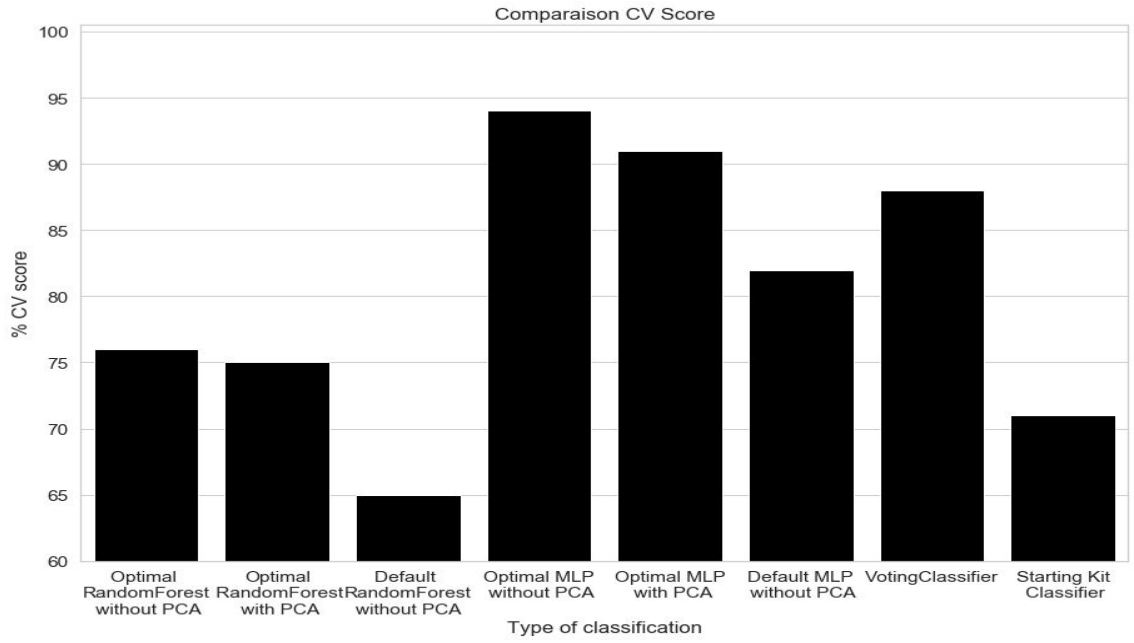



**Figure 9** : **Figure comparative du score de Cross validation**(Annexe 6) sur les différentes méthodes de classification les plus pertinentes testées, avec ou sans preprocessing[2]

# IV. Discussion

Il est incontestable que le challenge Persodata est l'un des meilleurs proposés cette année puisqu'il traite un thème d'actualités qui relie le développement technologique et surtout l'intelligence des machines, qui se rapprochent de plus en plus de l'humain, de plus derrière son aspect abordable (consistant à utiliser des classifieurs), nous avons découvert quelques recoins plus techniques, demandant de la réflexion, il nous a également permis d'en savoir un peu plus sur l'apprentissage supervisé, de pratiquer nos acquis en IA mais aussi sur Python, Github, ScikitLearn ..Nous avons aussi appris grâce à ce challenge que le monde du Machine Learning était si vaste , il y a tant de connaissances à acquérir, un travail personnel et l'envie d'en apprendre plus pourrait être nécessaire à tout amateur de Machine Learning. Tout ce travail et le score obtenu n'aurait pas pu être fait sans **la coordination de tous le groupe et la bonne volonté de tous ces membres**.

Ce challenge nous a permis de mieux gérer le travail en équipe en particulier nous partagions le même intérêt à ce travail, un preprocessing en utilisant la PCA a beaucoup simplifié le travail s'en est suivi un algorithme pertinent pour le classifieur (ici MLP) nous a valu un étroit rapprochement au score de référence ainsi qu'un amélioration du classifieur passant par plusieurs méthodes. Nous conseillons vivement ce challenge aux étudiants des années à venir.

Cependant d'après notre expérience, il est primordial de faire de Google votre meilleur ami, ne pas **hésiter à essayer et chercher des méthodes** sur internet par exemple des Tutorats sur Youtube .Un fidèle allié n'est autre que la doc de ScikitLearn pour éclairer les lanternes. Ne pas hésiter à faire des recherches en Anglais afin de diversifier les résultats

Une autre clé pour réussir le projet est de prendre en considération le facteur temps , en effet l'une des difficultés du challenge est l'attente des résultats pouvant excéder 2 à 3 heures. De plus le deuxième code preprocessing ( Dans ce challenge les données de sont déjà prétraitées) peut entraîner une baisse conséquente du score selon le classifieur ! Comme alternative serait d'occulter ce deuxième preprocessing ou changer de classifieur ce que nous avions faite remplaçant RandomClassifier par MLPClassifier.

# SOURCES:


[1] Tableau 1 : Résultats Préliminaires, de la proposition de projet

[2] Readmy.py du starting kit

[3] Depuis ScikitLearn:
https://scikit-learn.org/stable/auto_examples/model_selection/plot_randomized_search.html

[4]model.py : dans le Starting kit

[5] Classifier comparison :
https://scikit-learn.org/stable/auto_examples/classification/plot_classifier_comparison.html

[6] sklearn.neural_network.MLPClassifier :
https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html#sklearn.neural_network.MLPClassifier

[7]k-fold validation : https://scikit-learn.org/stable/modules/cross_validation.html

[8] A One-Stop Shop for Principal Component Analysis :
https://towardsdatascience.com/a-one-stop-shop-for-principal-component-analysis-5582fb7e0a9c

[9] Data pre-processing : https://en.wikipedia.org/wiki/Data_pre-processing

[10] Selecting dimensionality reduction with Pipeline and GridSearchCV :
https://scikit-learn.org/stable/auto_examples/compose/plot_compare_reduction.html#sphx-glr-auto-examples-compose-plot-compare-reduction-py

[11] sklearn.pipeline.Pipeline :
https://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html

[12] scikit learn: https://scikit-learn.org/stable/modules/naive_bayes.html

[13] scikit learn
https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDClassifier.html

[14] scikit learn:
https://scikit-learn.org/stable/modules/generated/sklearn.discriminant_analysis.QuadraticDiscriminantAnalysis.html

[15] scikit learn:
https://scikit-learn.org/stable/modules/generated/sklearn.gaussian_process.GaussianProcessClassifier.html#examples-using-sklearn-gaussian-process-gaussianprocessclassifier

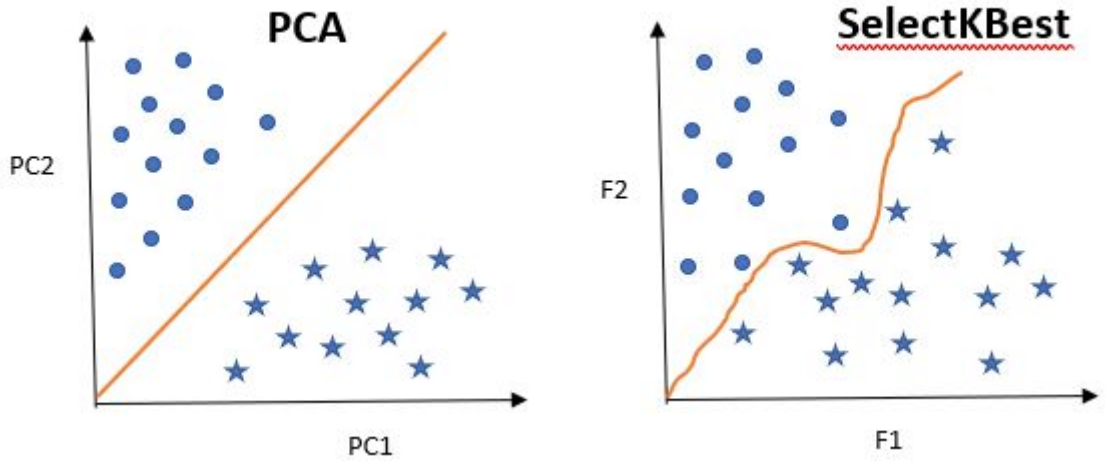
[16] sklearn.feature_selection.SelectKBest
https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.SelectKBest.html

# ANNEXES

## Annexe 1 : Un peu plus d'explications pour les méthodes PCA et SelectKBest

Supposons que dans notre cas nous avons affaire à des peintures qui représentent des données à classer linéairement; c'est à dire qu'il existe une classe Real et une classe Fake grâce auxquelles nou allons différencier nos peintures.

Il existe plusieurs méthodes afin de classer nos données mais avant d'avoir affaire aux algorithmes de classifications, il existe des méthodes tel que PCA et SelectKBest qui pourront corréler nos données, c'est à dire qu'elle va les regrouper selon des caractéristiques précises et nous pouvons obtenir les deux interprétations graphiques suivantes :




Dans le graphique de PCA précédent on voit deux masses de données corrélées suivant deux axes PC1 et PC2 séparées par une droite (données linéairement séparables)

Dans le graphique de KBest nous observons toujours les mêmes masses de données suivant 2 axes f1 f2 sauf que celles ci ont été séparées par une fonction f.

## **Annexe 2 :** Détails sur les classifieurs

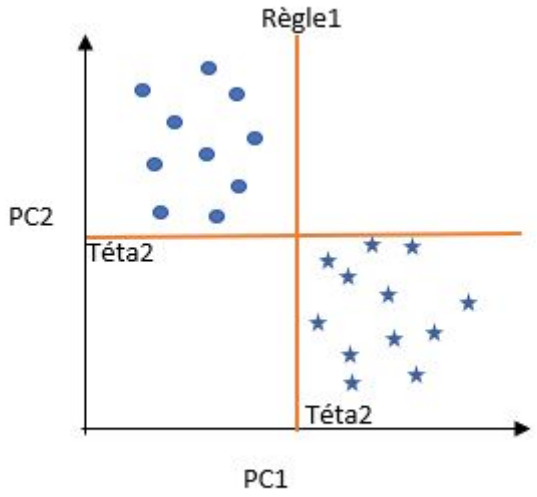
Nous avions testé plus de 7 classifieurs voici, selon notre compréhension la manière dont ils fonctionnent :

- Quadratic Discriminant Analysis (QDA) est une généralisation du modèle linéaire son objectif est d'étendre la capacité du classifieur à représenter des surfaces de séparation plus complexes en utilisant …
- Naive Bayes: Les méthodes naïves de Bayes sont un ensemble d'algorithmes d'apprentissage supervisé reposant sur l'application du théorème de Bayes avec l'hypothèse «naïve» d'indépendance conditionnelle entre chaque paire d'entités, compte tenu de la valeur de la variable de classe.
- SGDC Classifier: C'est un classificateur linéair, il met en œuvre des modèles linéaires régularisés avec apprentissage par descente de gradient stochastique (SGD): le gradient de la perte est estimé chaque échantillon à la fois et le modèle est mis à jour en cours de route avec un plan de force décroissant (vitesse d'apprentissage). SGD permet l'apprentissage par minibatch (en ligne / hors du cœur), voir la méthode partial_fit. [13]

- Quadratic Discriminant Analysis: C'est un classifieur avec une limite de décision quadratique, généré en ajustant les densités conditionnelles de classe aux données et en utilisant la règle de Bayes.[14]

- Gaussian Process Classifier: Classification des processus gaussiens (CPG) basée sur l'approximation de Laplace.[15]

- Random Forest : Afin de bien comprendre cette méthode de classification, nous devons passer par le Decision Tree, ce classifieur se base sur les données préprocessées de PCA comme suit:




Suivant nos données le **Decision Tree** va séparer nos donnés grâce à des séparatrices qui créeront un arbre, pour suivre l'exemple du graphique donné, nous allons construire notre arbre comme suit: nous allons classer suivant la séparatrice verticale si le PC1 est supérieur ou non à Téta1, si c'est le cas nous mettons le label sur nos données comme étant Fake, sinon on crée une nouvelle séparatrice horizontale et comparons le PC2 au Téta2, s'il est supérieur le label sera posé en tant que réel sinon nous continuons jusqu'à terminer notre classification. c'est ainsi que nous venons de créer notre arbre de décision et ces arbres peuvent arriver à de grandes profondeurs et gérer ces profondeurs là est primordiale pour obtenir une bonne classification car nous pouvons vite tomber dans le surapprentissage.

La question qui se pose est donc: Pourquoi nous pouvons nous retrouver dans un surapprentissage avec un arbre de décision?

La réponse est tel que le **sur-apprentissage** arrive quand notre modèle colle exagérément aux données et dans cet exemple, nous voyons effectivement qu'à mesure que la taille de l'arbre augmente, le taux d'erreur calculé sur les données d'apprentissage diminue constamment. En revanche, le taux d'erreur calculé sur l'échantillon test montre d'abord une décroissance rapide, jusqu'à un arbre avec une quinzaine de feuilles, puis nous observons que le taux d'erreur reste sur un plateau avant de se dégrader lorsque l'arbre est manifestement surdimensionné, nous comprenons maintenant l'importance de la profondeur des arbres.

**le sur-apprentissage** s'interprète comme un apprentissage « par cœur » des données, un genre de "mémorisation". Il résulte souvent d'une trop grande liberté dans le choix du modèle.

Pour revenir au **Random Forest**, ce modèle de classification créé une forêt d'arbres et la rend aléatoire. La « forêt » qu'elle construit est un ensemble d'arbres de décision, entraînés la plupart du temps avec la méthode de « bagging ». Autrement dit Random forest construit plusieurs arbres de décision et les fusionne grâce à un vote majoritaire avec les poids des prédictions des arbres pour obtenir une prédiction plus précise et plus stable.

## **Annexe 3 :** Définition d'une Cross-Validation:

Lorsque nous voulons évaluer notre méthode de classification, il nous est possible de le faire sur l'ensemble d'entraînement, sauf que c'est inutile parce qu'il s'est justement entraîné sur ces données et c'est fort possible de se retrouver dans un overfitting, c'est pourquoi nous pouvons avoir recours à la Validation Croisée qui elle utilise un ensemble de test, le mécanisme de la Cross Validation consiste en partitionnant l'ensemble d'apprentissage en k ensembles plus petits, dans notre cas où notre modele.py contenait 2 fonctions principales "fit" et "predict" nous allons donc utiliser k-1 de nos données partitionnées que nous allons évaluer avec notre "fit" et en garder à chaque fois un pour le "predict", il suffira après ça de recoller nos folds qui sont les résultats de l'évaluation du k-ème élément dans le predict, le résultat de notre CV ne sera d'autre que la moyenne des valeurs calculées par ce processus multiplié par k.

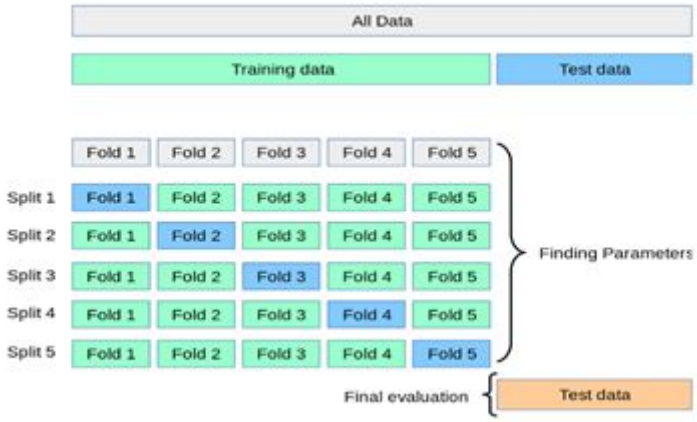



Figure 4: k-fold validation [7]

**Annexe 4 :** Capture d'écran du script Grid et Random Search, appliqué au RandomForestClassifier [2]

| | |
|---|---|
| """ # Recherche des meilleurs paramétre avec la methode Random Search<br>from sklearn.ensemble import RandomForestClassifier<br>from sklearn.model_selection import RandomizedSearchCV<br><br># Create a based model<br>clf = RandomForestClassifier()<br><br># Create the parameter grid based on the results of random search<br>param_grid = {<br>'n_estimators': [50,60,180],<br>'max_depth': [10,15,20],<br>'max_features': ['auto', 'sqrt', 'log2']<br>}<br><br># Instantiate the grid search model<br>grid_search = RandomizedSearchCV(clf, param_grid)""" | _ Import le classifieur RandomForestClassifier de sklearn<br>_ Import la méthode de recherche RandomizedSearchCV de sklearn<br>_ Applique le modèle à nos données<br>_ Création de param_grid , chaque paramétré est définie selon un tableau<br><br>Instanciation de grid_search |
| """ # Recherche des meilleurs paramétre avec la methode Grid Search<br>from sklearn.ensemble import RandomForestClassifier<br>from sklearn.model_selection import GridSearchCV<br><br># Create a based model<br>clf = RandomForestClassifier()<br><br># Create the parameter grid based on the results of Grid Search<br>param_grid = {<br>'n_estimators': [50,60,180],<br>'max_depth': [10,15,20],<br>'max_features': ['auto', 'sqrt', 'log2']<br>}<br><br># Instantiate the grid search model<br>grid_search = GridSearchCV(clf, param_grid) """<br><br>if not(M.is_trained):<br>X_train = D.data['X_train']<br>Y_train = D.data['Y_train']<br># Fit the grid search to the data<br>#grid_search.fit(X_train, Y_train)<br><br># affiche les meilleurs parametres<br>#print(grid_search.best_params_)<br>M.fit(X_train, Y_train) | _ Import le classifieur RandomForestClassifier de sklearn<br>_ Import la méthode de recherche GridSearchCV de sklearn<br>_ Applique le modèle à nos données<br>_ Création de param_grid, chaque paramètre est définie selon un tableau<br><br>Instanciation de grid_search<br><br>Affichage des meilleurs paramétrés avec une méthode |

## Annexe 5 :

## Annexe 6 : La courbe de ROC

La courbe de ROC est une mesure de la performance d'un classificateur binaire , c'est-à-dire catégoriser des éléments en deux groupes distincts sur la base d'une ou plusieurs des caractéristiques de chacun de ces éléments. Graphiquement, on représente souvent la mesure ROC sous la forme d'une courbe qui donne le taux de vrais positifs(fraction des positifs qui sont effectivement détectés) en fonction du taux de faux positifs (fraction des négatifs qui sont incorrectement détectés) . Nous pouvons en dessiner une et comparer notre score avec le score de référence

## Annexe 7 : Code et pseudo code de la Courbe de ROC, métrique de notre challenge

| | |
|---|---|
| **from sklearn import metrics** | |
| n_classes=2 | // Compute Receiver operating characteristic |
| **def** fpr_tpr(solution, prediction): | //calcule air sous la courbe (AUC) en utilisant la loi trapézoidale |
| for i in range(n_classes): | //Déclaration de la fonction p2c qui crée la superficie |
| fpr, tpr, _ = metrics.roc_curve(solution, prediction) | |
| roc_auc = metrics.auc(fpr, tpr) | |
| return (fpr,tpr) | |
| **def** p2c(prediction,threshold=0.5) : | |
| c = [] | |
| for ele in prediction : | |
| if(ele>=0.5) : | // condition qur la variable ele lors de la prédiction , insere le contenur spécifié par le paramètre , pour la fin de chaque élément |
| c.append(1) | |
| else : | |
| c.append(0) | |
| return np.array(c) | retourne la superficie demandée |
| **def** plot_cm_matrix(solution,prediction,title) : | // fonction qui évalue la qualité les output du classifieur et permet donc la création de la matrice |
| prediction = p2c(prediction) | |
| cm = confusion_matrix(solution, prediction) | |
| df_cm = pd.DataFrame(cm, index = [i for i in "01"],columns = [i for i in "01"]) | |
| plt.figure(figsize = (5,3)) | |
| sn.heatmap(df_cm, annot=True) | //variable de la matrice de confusion |
| plt.title(title) | |
| **def** plot_ROC(fpr,tpr,title) : | |
| plt.figure() | |
| lw = 2 | // utilisé pour la matrice de corrélation |
| plt.plot(fpr, tpr, color='darkorange',lw=lw) | // met les titres pour les axes de la courbe des ROC préalablement préparée |
| plt.plot([0, 1], [0, 1], color='navy', lw=lw, linestyle='--') | // fonction qui nous permettra de dessiner la courbe de ROC pour notre classificateur |
| plt.xlim([0.0, 1.0]) | |
| plt.ylim([0.0, 1.05]) | |
| plt.xlabel('False Positive Rate') | |
| plt.ylabel('True Positive Rate') | |
| plt.title(title) | |
| plt.show() | |
| **if**(os.path.exists("sample_data/perso_test.solution") and os.path.exists("sample_data/perso_valid.solution")) : | // Variables utilisées pour la création de la courbes |
| fpr_train,tpr_train = fpr_tpr(Y_train, Y_hat_train) | |
| fpr_test,tpr_test = fpr_tpr(Y_test, Y_hat_test) | |
| fpr_valid,tpr_valid = fpr_tpr(Y_valid, Y_hat_valid) | |
| print('Training score for the', metric_name, 'metric = %5.4f' % scoring_function(Y_train, Y_hat_train)) | |
| print('Ideal score for the', metric_name, 'metric = %5.4f' % scoring_function(Y_train, Y_train)) | //Condition sur l'importation du path importé de sample_data |
| print('Test score for the', metric_name, 'metric = %5.4f' % scoring_function(Y_test, Y_hat_test)) | |
| print('Valid score for the', metric_name, 'metric = %5.4f' % scoring_function(Y_valid, Y_hat_valid)) | |
| **plot_cm_matrix**(Y_train,Y_hat_train,"Confusion matrix for train data") | /fonction utilisée pour visualiser la matrice de confusion |
| **plot_ROC**(fpr_train,tpr_train,"ROC curve for train data") | //Dessiner la matrice |
| **plot_cm_matrix**(Y_test,Y_hat_test,"Confusion matrix for test data") | */dessin de la courbe de ROC pour les données de /est |
| **plot_ROC**(fpr_test,tpr_test,"ROC curve for test data") | // dessin de la courbe de ROC pour les valeurs |

## **Annexe 8 :** Résultats des recherches de paramètres

RandomizedSearchCV took 9330.64 seconds for 20 candidates parameter settings :
Model with rank: 1 Mean validation score: 0.873 (std: 0.002)

Parameters: {'solver': 'adam', 'max_iter': 1500, 'learning_rate': 'invscaling', 'hidden_layer_sizes': (200, 100, 50, 20), 'activation': 'relu'}

Model with rank: 2 Mean validation score: 0.843 (std: 0.005)
Parameters: {'solver': 'adam', 'max_iter': 1500, 'learning_rate': 'adaptive', 'hidden_layer_sizes': (100, 50, 100, 50, 20), 'activation': 'relu'}

Model with rank: 3 Mean validation score: 0.841 (std: 0.003)
Parameters: {'solver': 'adam', 'max_iter': 500, 'learning_rate': 'adaptive', 'hidden_layer_sizes': (100, 50), 'activation': 'relu'}

## Annexe 9:

code pour realiser le CV:

```
1. from sklearn.metrics import make_scorer
2. from sklearn.model_selection import cross_val_score   #appeler la fonction d'assistance cross_val_score sur l'estimateur
3. scores = cross_val_score(M, X_train, Y_train, cv=5, scoring=make_scorer(scoring_function))   #calculer le score 5 fois consecutive
4. cv_score = scores.mean()
5. cv_ebar = scores.std() * 2
6. print('\nCV score (95 perc. CI): %0.2f (+/- %0.2f)' % (cv_score, cv_ebar))
```

code pour l'histograms de Panda DataFrame (figure 5)

```
1. data_dir = 'public_data'
2. data_name = 'perso'
3. !dir $data_dir
4. from data_io import read_as_df
5. data = read_as_df(data_dir  + '/' + data_name)        # The perso_data is loaded as a Pandas Data Frame
6. data.head()
7. data.boxplot()
```

code pour corrélation avec caractéristique (figure 6)

```
1. print(data.head())
2. data_num = data.copy()
3. data_num['target']= data_num['target'].astype('category')
4. data_num['target'] = data_num['target'].cat.codes
5. print(data_num.head())
6. sns.heatmap(data_num)
```

**Annexe 10** : Histogramme de Panda DataFrame

une représentation de la distribution des données. Cette fonction appelle matplotlib.pyplot.hist () sur chaque série du DataFrame, ce qui donne un histogramme par colonne.