Rapport : Projet Kahlo

Description du groupe :

- Noms des membres avec leur groupe administratif *:* COHEN Raphaël (1 bis) et RAIMBAUD REMAZEILLES Raphaël (3) pour le binôme preprocessing, LANSARI Mohammed (1bis) et BOUZID Yanis (1 bis) pour le binôme classification, MILLS Dylan (2) et LESPINE Nathan (2) pour le binôme visualisation.
- Nom de l'équipe : Kahlo
- URL Challenge Codalab choisi (préprocessing) : https://codalab.lri.fr/competitions/401
- URL Challenge Codalab choisi (RAW) : https://codalab.lri.fr/competitions/404
- Numéro de la dernière soumission de Code Codalab : Codalab ne reconnait pas le module Keras donc merci de vous conferer aux lien GitHub pour le code preprocessing.
  (Score Preprocessing : 8267)
  (Score Raw : 8803 / Code Raw : 8800)
- Meilleur score en Preprocessing sur Codalab : 0.9478
- Meilleur score en RAW Data sur Codalab : 0.5838
- URL vidéo Youtube finale : https://www.youtube.com/watch?v=oPBhsrtIOKc&feature=youtu.be
- URL du repo Gihub : https://github.com/Kahloo/kahlo
- URL Github contenant le code Raw data: https://github.com/Kahloo/kahlo/tree/master/RAW_data_FINAL
- URL Github contenant le code Preprocessing : https://github.com/Kahloo/kahlo/tree/master/Prepross_data_FINAL
- URL des diapos de la présentation (format .pptx à download) : https://github.com/Kahloo/kahlo/blob/master/Pr%C3%A9sentation-kahlo.pptx

I. Introduction

Avec la venue de l'intelligence artificielle au sein de plusieurs domaines qu'ils soient culturels, scientifique ou encore commercial, beaucoup de questions d'éthique se sont posées. Parmi ces questions, beaucoup d'interrogations se posent dans le domaine de l'art et plus particulièrement dans la peinture. Les ressemblances entre vraies et fausses peintures étant très semblables. Une problématique s'est créée autour de l'authenticité des peintures.
Notre projet (persodata), consiste à créer un algorithme capable de **classifier** et donc de différencier les fausses (créées artificiellement) des vraies peintures.
Pour cela nous avons à notre disposition 65 856 images d'entrainement, 9 408 images de validation et 18 817 images de test respectivement.
Pour résoudre notre problème, contrairement au *starting* kit, nous avons décidé de coder notre propre réseau de neuronnes (qui sera expliqués par la suite) en nous aidant de la bibliothèque *keras*.
C'est un perceptron multicouche où nous définissons le nombre de couches souhaitées en variant les paramètres et hyper-paramètres afin que ce nombre de couches soit optimale pour notre problème et qu'il donne le meilleur résultat possible.
Pour réaliser ce projet, nous nous sommes réparti le travail selon trois étapes essentielles.
Dans un premier temps un groupe s'est occupé de la partie preprocessing. Ensuite un autre binôme a réalisé la comparaison des différents algorithmes (classifieur).
Et pour finir, le dernier groupe s'est occupé de représenter graphiquement les résultats obtenus afin de faciliter la lecture et la comparaison des données.

II. Descriptions des algorithmes étudiés

a) *Preprocessing*

Pour cette partie du projet, nous allons utiliser les bibliothèques numpy et scikit-learn. Le Preprocessing a pour objectif de transformer les données afin de rendre le processus de Classification plus efficace.

Tout d'abord, les données d'entraînement, de test et de validation sont chargées dans 3 vecteurs numpy, 'X_train', 'X-test' et 'X_valid' respectivement. Ces données sont standardisées durant ce chargement à l'aide du module StandardScaler qui transforme les données afin que leur moyenne soit égale à 0 et leur écart-type à 1.
Cependant cela ne suffit pas. En effet, les données sont non seulement nombreuses, mais elles ont aussi des dimensions bien trop grandes pour le modèle de Classification. Chaque image a pour dimensions 64*64 pixels et chacun de ces pixels a 3 composantes pour les couleurs rouge, vert et bleu (RGB). Cela nous donne un total de 12288 valeurs pour chaque image. Nous allons donc utiliser l'algorithme du PCA afin de réduire les dimensions des données.
Mais le problème est qu'utiliser l'algorithme du PCA pour de si grandes données s'avère être un processus lent et coûteux. Pour régler ce problème, nous allons utiliser le module IncrementalPCA qui permet d'appliquer l'algorithme du PCA sur de petits paquets de données.
Sachant que les données sont chargées par paquets de 1000 images dans leurs vecteurs respectifs, nous allons effectuer le fitting pour l'IncrementalPCA sur ces paquets de données avant qu'elles ne soient placées dans leur vecteur. Pour cela, la fonction partial_fit est utilisée. De cette manière, le fitting est moins coûteux puisqu'il est effectué progressivement sur une partie des données à la fois.
Une fois le fitting terminé, les données sont transformées afin de réduire leur dimension. On se retrouve donc avec des données où chaque image est représentée par un vecteur à 200 éléments.

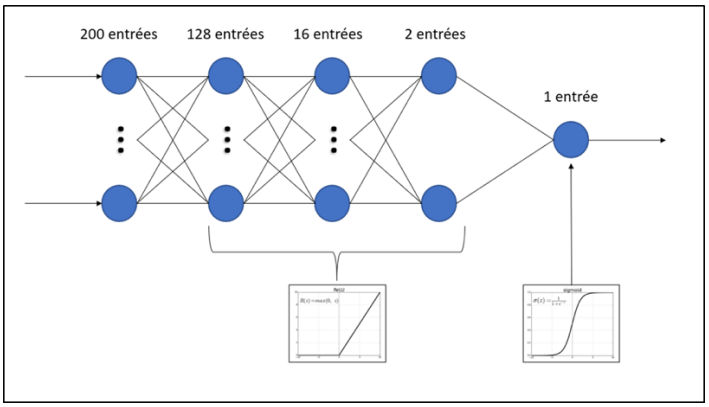
b) *Classification*

Une fois les données préparées, il faut désormais utiliser un algorithme qui va apprendre à différencier les données pour finalement permettre d'identifier si une image est fausse ou non avec une marge d'erreur très faible. Le point le plus crucial de cette partie est de choisir les meilleures méthodes à essayer pour notre problème spécifique. Ainsi pour optimiser notre de temps de travail nous menons une recherche intensive sur le web pour retenir les méthodes de classification les plus utilisées : nous consulterons surtout les articles où les auteurs compares différents algorithmes et y exposent leurs scores. Grâce à plusieurs sources nous retiendrons au final les méthodes RandomForestClassifier, LogisticRegression et MultiLayerPerceptron de Scikit Learn[1].
Tout les résultats obtenus par nos différents classificateurs seront comparés dans la partie visualisation.
Une fois nos classifieurs choisis nous avons dans un premier temps utilisé ses méthodes avec le StartingKit où les données sont déjà prêtes pour deux raisons : d'une part, pour le fait que le score va dépendre essentiellement du classifieur utilisé et d'autre part pour ensuite comparer ces scores avec ceux obtenus sur notre préprocessing pour évaluer si ce dernier est plus ou moins efficace que la version des Masters.
Les versions de code fournies nous permettent de seulement modifier la classe objet model présente dans le fichier *model.py.*

[1] https://stackabuse.com/classification-in-python-with-scikit-learn-and-pandas/ https://lukesingham.com/whos-going-to-leave-next/

La première méthode utilisée est la LogisticRegression qui est une méthode de classification linéaire qui va simplement ajuster les coefficients d'un polynôme de degrés n qui représente le nombre d'éléments caractérisants nos images après traitement par le préprocessing[2].
La deuxième méthode quant à elle est la RandomForestClassifier qui va utiliser un grand nombre d'arbre de décision entrainés sur des sous-ensembles[3].

Après avoir fait varier les hyperparamètres nous avons conclu sur les avantages comme les inconvénients de ces deux méthodes : LogisticRegression est rapide mais donne un score plutôt moyen comme nous le verrons dans la partie visualisation tandis que le RandomForestClassifier donne un score très bon mais avec un temps d'apprentissage très long (une vingtaine de minute aux pires des cas). Ce résultat vient rejoindre nos recherches théoriques : en effet la LogisticRegression est plus efficace sur des données linéairement séparables et le RandomForestClassifier a une complexité $O\big(v \times nlog(n)\big)$ où n est le nombre d'arbres créés et v le nombre de variables.
La méthode la plus convaincante et celle qui nous aura permis d'effectuer les meilleurs scores sera le MLPClassifier : Ce dernier est simplement un réseau de neurone composé de plusieurs couches dites cachés dont les paramètres sont très variables (fonction d'activation, nombre de de couche…).
Nous nous sommes donc penchés sur cette méthode de classification qui présentait d'énormes avantages : d'une part le rapport temps d'exécution / score est très bon et la personnalisation nous permet d'adapter notre réseau à notre problème.
Néanmoins cette personnalisation s'est très vite montrée moins flexible qu'au premier abord. Ainsi nous avons décider de créer nous même notre propre réseau multicouche tout en restant dans un niveau d'abstraction raisonnable.
Nous avons finalement décidé de créer un réseau grâce au module Keras inspiré du perceptron multicouche[4].
On peut schématiser notre modèle comme suit :



Avec ces trois couches Relu et sa sortie Sigmoid, ce réseau nous permet d'obtenir de très bon score (assez pour battre la référence) et tout cela en un temps record : ce dernier met en effet moins de deux minutes pour la phase d'apprentissage.
Ce modèle alliant efficacité et rapidité, nous déciderons de le conserver pour le restant du projet.

III. Résultats obtenus dans le challenge et Visualisation

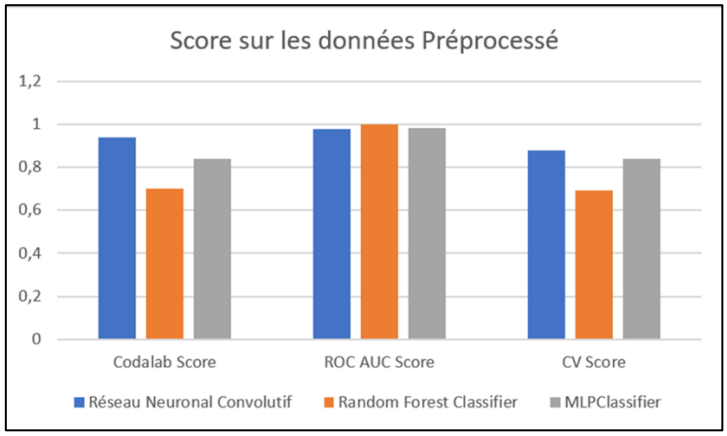
Une partie plus pratique mais néanmoins très utile de notre projet est la visualisation de nos données. En effet grâce à de nombreuses méthodes et modules, nous avons pus durant notre projet représenter visuellement nos données afin d'améliorer ou de complètement abandonner certaines voies abordées.

[2] https://mrmint.fr/logistic-regression-machine-learning-introduction-simple
[3] https://fr.wikipedia.org/wiki/For%C3%AAt_d%27arbres_d%C3%A9cisionnels
[4] http://penseeartificielle.fr/focus-reseau-neurones-artificiels-perceptron-multicouche/

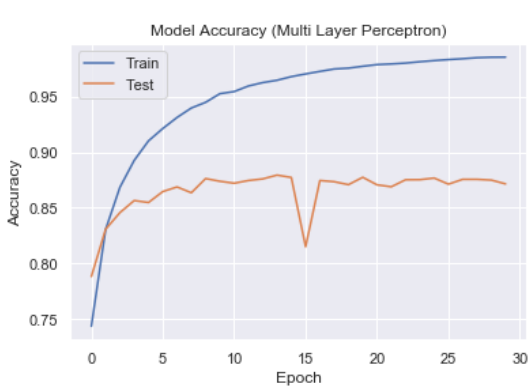
Dans un premier temps nous avons tracés sur un diagramme les différents scores obtenus de nos meilleurs classifieurs afin de choisir la méthode la plus efficace et la développer. Les méthodes de visualisation nous ont surtout permis d'interpréter nos différents résultats et de conclure. Nous avons donc comparé plusieurs types de scores entres eux :

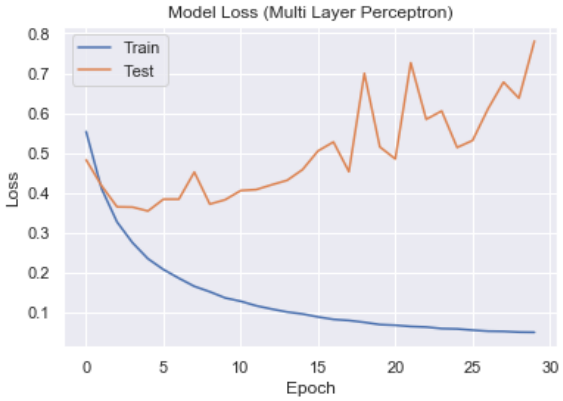
- Le score de CodaLab qui résulte des bonnes réponses prédites par notre classifieur sur des données où ce dernier ne s'est pas entrainé.
- Le score AUC qui est calculé à partir de l'air sous la courbe ROC qui est elle-même obtenue sur données d'apprentissage.
- Le score de Cross Validation ou Validation croisée qui est obtenu par échantillonnage. Cette dernière peut être calculée selon plusieurs méthodes que l'on explicitera dans le Bonus (Mettre un Saut sur ce chapitre).

En partant des données ayant déjà subi un préprocessing, puisque nous savons que le LinearClassifier ne nous donne pas un bon score par rapport aux autres classifieurs, nous l'avons délibérément pas pris en compte dans nos résultats.

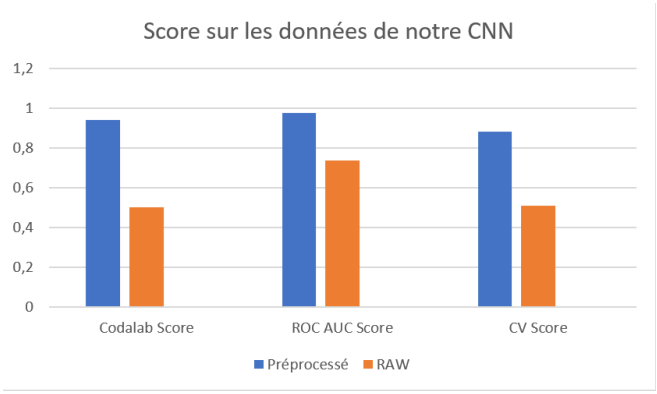


Comme nous pouvons le voir, globalement les score AUC sont très bon (proche de 1) et ceci est normal du fait que ce dernier est calculé sur l'ensemble d'entrainement contrairement aux deux autres : Ainsi on retiendra que le AUC score n'est pas très significatif pour comparer nos données si ce n'est le fait que tout les classifieurs utilisées doivent évidemment être très bon sur notre d'entrainent.
Ensuite comme nous pouvons le constater les deux réseaux de neurone (le MLPClassifier fournit par Scikit Learn et l'autre définit par nous-même grâce à Keras) donnent de bien meilleurs scores sur les ensembles que ces derniers n'ont pas « vu » par rapport au RandomForestClassifier. En effet c'est en comparant les scores de Codalab et les CV score que nous sommes arrivé à la conclusion qu'un réseau de neurone serait la méthode la plus adapté à notre problème. Pour finir, il est aisé de constater que notre réseau convolutif donne de bien meilleur score que le MLPClassifier.
Pendant cette phase de recherche du score le plus élevé possible, nous avons décidé de tracer la précision et la perte de notre réseau en fonctions des époques de notre réseau convolutif :





La notion de précision (accurancy) est la proportion des bonnes réponses parmi les données proposés[5] tandis que la perte (loss) est une notion complexe qui pour faire simple est un bon indicateur de la notion de surapprentissage (d'over-fitting). Ces deux courbes nous ont permis de trouver le nombre d'époque où un surapprentissage peut se faire ressentir. Dans notre cas on observe qu'a environ 15 époque, notre modèle garde le même score de précision pour les données de test tandis que la perte augmente à vue d'œil : de ces deux graphiques ont fixe le nombre d'époque de notre modèle à 15 époques.
Pour finir, nous sommes passés sur nos données brute (RAW) et nous avons obtenus l'histogramme suivant :



Comme nous pouvons l'apercevoir, pour un même classifieur le score de notre version préprocessing est beaucoup plus basse que celle des Master : on en conclut que notre préprocessing n'est pas optimisé pour notre jeu de donnée.

Résultats obtenus dans le challenge

Puisque nous avons pu faire les des challenges proposés (Raw data et Preprocess data), nous allons maintenant comparer les classifieurs entre eux et les codes des deux challenge.
Si vous avez pu voir dans l'introduction que nous avons un score moins élevé dans le challenge preprocessing, c'est tout simplement que la partie sur le code pour preprocesser les données est moins efficace que celle où les données sont déjà preprocessées.
En effet, le temps pour preprocessées les données est extrèmement long (cf la suite) et donc pour améliorer à chaque fois notre modèle cela prend beaucoup de temps.
De plus, dans notre cas, nous n'avons utilisé seulement deux classifieurs pour ces projets (classifieurs que nous n'avons pas prit au hasard évidemment mais que nous avons choisi par rapport à leur temps d'exécution et à leur efficacité. Donc nous allons comparer ces deux classifieurs entre eux).
Comparons maintenant les classifieurs sous forme d'un tableau, pour voir dans notre cas, lesquels sont les plus efficaces :

Score fournit par Codalab	RandomForestClassifier	Réseau personalisé
Données préprocessées	0.72	0.94
Données brutes	0.58	0.54

[5] https://fr.wikipedia.org/wiki/Pr%C3%A9cision_et_rappel

On voit donc que suivant le challenge choisit, la performance du classifieur est différente. En effet on peut expliquer cela de plusieurs manières : Notre réseau personalisé est très particulier, c'est-à dire qui faut faire varier les **hyper-paramètres** (qui sont ici le nombre de couche du perceptron) et de trouver manuellement le bon nombre de couche à mettre. Dans un cas où il n'y a pas énormément de données, celà reste convenable. Tandis que le RandomForestClassifier est un classifieur "général" et qui peut s'adapter facilement à un nombre gigantesque de donnée.

IV. Discussion et Conclusion

1) Leçons apprises

Dans cette partie, nous allons parler des quelques problèmes rencontrés dans ce projet.
Le problème "majeur" de ce projet est le temps de chargement des données. Je vais vous donner quelques raisons :

- La lecture et le stockage dans des vecteurs à 1 dimension de 5 Go de données prend tout d'abord énormément de temps (même avec de l'optimisation de données) à exécuter (car nous avons choisi de travailler sur les raw data).
- L'apprentissage sur le très grand nombre de données que nous avons est important.
  Par exemple, la complexité du classifieur que nous avons choisi d'utiliser (Random Forest) est $O\big(v \times nlog(n)\big)$ où n est le nombre d'arbres créés et v le nombre de variables.
  Or nous avons décidé de mettre le paramètre v à 1000 et avec notre très grand nombre de données, la complexité devient donc excessivement grande.

Les solutions qu'on peut apporter à ces deux problèmes sont :

- Pour le premier problème : Implémenter un *Gaussian blur* sur nos images compressées afin d'optimiser la classification. Toujours dans l'optique d'améliorer le Preprocessing, on aurait aussi pu implémenter une fonction qui segmente les images.
- Une solution radicale permettant de réduire considérablement la complexité de la classification serait d'utiliser des réseaux de neurones pré-entraînés qui nous suffirait d'adapter à nos classes pour les rendre fonctionnels et efficace.

2) Messages aux étudiants de l'an prochain / Conseils

Pour les étudiants qui liront ce message et qui auraient choisi un projet ressemblant au notre, nous vous avons rassemblé ici, une petite liste qui, pour nous, rassemblerait les erreurs à éviter pour bien entreprendre ce type de projet :

i. *Conseil n°1* : Ce conseil est plutôt général à n'importe quel type de projet et correspond à la cohésion de groupe. Vous serez sûrement dans une équipe de projet qui sera divisé en plusieurs sous-groupes correspondants à différentes parties du projet. Ce qui nous a permis de bien avancer est l'entraide entre les sous-groupes. On pourrait penser que chacun fait sa partie et si aujourd'hui je fais partie du binôme classification et que je n'ai pas grand chose à faire alors c'est cool pour moi ! C'est une erreur à éviter car il n'y a pas de partie qui a plus de travail qu'une autre et cela veut dire qu'à un autre moment vous aurez une tache lourde/un programme complexe à coder pendant que les autres binômes n'en auront pas, et on peut vous dire que coder un programme à 6 est beaucoup plus productif qu'à 2. Tout ça pour dire que pour réussir ce projet il faut absolument de l'entraide entre les groupes et se motiver mutuellement !

ii. *Conseil n°2* : Pour bien débuter dans ce type de projet et ne pas baisser les bras dès le début, il faut commencer par bien comprendre ce qui est demandé. C'est donc naturellement que débuter par des algorithmes et des données facilement utilisables/exécutables est une règle d'or. Donc pour commencer ce projet il faut éviter de vouloir classifier les données brutes (raw data). Il faut commencer par les données préprocessées (gentiment déjà préprocessées par les professeurs !).

iii. *Conseil n°3* : Par rapport à un conseil un peu plus technique sur ce projet (mais est sûrement valable avec d'autres projets !), le choix du classifieur est très important. Comme nous le disions plus haut, nous avons une quantité astronomique de données à analyser et trier (les fameux 5 Go). Un classifieur très précis et qui a un algorithme qui souhaite trier très minutieusement pourrait ne pas être optimale pour ce genre de projet. Il nous faut une combinaison d'un classifieur qui trie assez bien les données tout en ayant une complexité assez bonne pour ne pas durer une éternité à compiler à chaque fois (Il y a deux conséquences à ce problème qui sont :
Plus l'algorithme est long à compiler et plus on aura du temps à coder notre classification car si à chaque fois que l'on apporte une modification à notre code on attend 45 min que notre code compile pour au final avoir une autre erreur, la recorriger et ainsi de suite, cela va nous prendre énormément de temps.Et de plus, plus notre jeu de données est grand, plus il faut un classifieur qui soit d'une complexité moindre qu'un jeu de donnée petit. Car si maintenant ce n'est pas 5 Go de données mais 1 Terra ... le temps de classification peut varier énormément suivant le choix du classifieur).Pour résumer il faut choisir un classifieur adapté (dans notre cas) à la taille du jeu de données que l'on souhaite classifier.

Bonus

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	MLP	Réseau de Neuronnes Convolutif
Training (roc_auc_score)	0.587	0.725	1.00	0.989	0.98	0.97
CV	0.59 (± 0.01)	0.79 (± 0.00)	0.57 (± 0.00)	0.61 (± 0.01)	0.83 (± 0.00)	0.92(± 0.00)
Validation	0.57	0.71	0.56	0.60	0.81	0.90

nouvelles méthodes explorées

Les nouvelles méthodes que nous avons rajoutées ont été expliquées plus haut dans toute la partie classification.

Cross-Validation : Découpe notre jeu de données en n échantillons. Chaque échantillion va être utilisé comme jeu de test (chacun à des moments différents), pendant que les autres échantillons seront des jeux d'entrainement. On entraine donc nos données à chaque fois (donc sans utiliser le jeu de test), puis comme cela, nous obtenons une performance sur chacun de nos échantillons que nous pouvons par exemple moyenner pour avoir une approximation moyenne de nos performances.

Surapprentissage : S'effectue lorsqu'un classifieur a trop appris les particularités des données d'entrainement (avec un score proche de 100%) et commence donc à ne savoir classifier très bien que ces types de données au détriment des données étrangères que ce dernier n'a jamais rencontré.