S3
2022

Détection du type de cartes de séjour

TP PYTHON – PROJET DU SEMESTRE 3

MARC-ANTOINE AUVRAY – PAUL DELAMARRE – MATTHIEU LE GALL – NATHAN PAPET

Sommaire

Objectifs et problématique

Automatiser les processus de tâches répétitives telles que la reconnaissance automatique de documents est l'un des enjeux majeurs pour beaucoup d'entreprises ou autres structures. Cela peut notamment offrir aux équipes plus de confort accru pendant l'utilisation de ces processus ou aussi plus de facilité quant à la compréhension du fonctionnement et de l'objectif de ces processus. Ces processus automatisés étant plus efficients, cela apporte alors un gain de temps, et en conséquence, les équipes peuvent investir ce temps pour travailler sur d'autres tâches notamment certaines qui seraient plus avancées.

Parmi les documents qu'il peut être utile de classer de manière automatique dans un contexte d'entreprise ou plus souvent d'administration publique, on retrouve les titres de séjour. Ces cartes sont nécessaires pour les ressortissants étrangers (hors Union Européenne) souhaitant résider en France pour une période supérieure à trois mois. Ces titres de séjour ont changé d'aspect en 2020 pour passer d'une carte rose à une carte blanche. Certaines informations ont également changé.

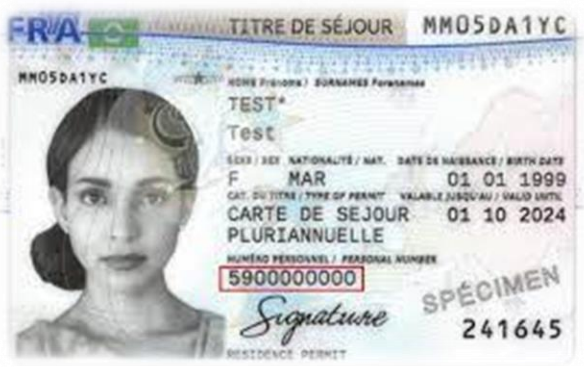


Figure 1 - Nouvelle carte de séjour

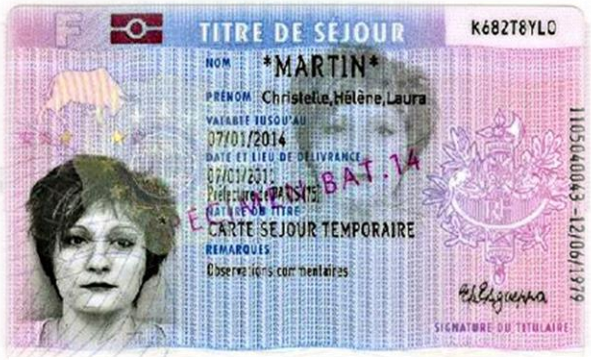


Figure 2 - Ancienne carte de séjour

Voici un exemple des deux types de carte de séjour. A gauche une nouvelle carte de séjour et à droite une ancienne carte de séjour.

Nous allons donc chercher à automatiser le processus de catégorisation de ces deux types de cartes. Le but de ce projet n'est pas de trouver une solution parfaite. La motivation est plus d'ordre exploratoire. En l'occurrence, nous avons pour objectif de travailler les notions suivantes sur *Python* :

-traitement et préparation de données;

-machine learning;

-deep learning;

-cross-validation;

-classification supervisée;

-réseaux de neurones;

-*data augmentation*.

Dans la première partie du corps de ce rapport, nous présenterons les données et les outils *Python* utilisés. Les deux parties suivantes seront consacrées à l'application, la description et l'explication de méthodes de *machine learning* et de *deep learning*. Enfin nous conclurons en analysant les résultats et en proposant des perspectives. Ce rapport a été rédigé en prenant en compte les modalités demandées par monsieur Ho, qui a validé notre sujet dérivé du sujet sur la classification des cartes d'identités. Nous nous sommes également basés sur les modalités du rapport du mémoire d'ingénieur du CNAM pour la forme.

1. Etat des lieux

1.1. Présentation des données

Pour travailler sur ce sujet, nous n'avons pas eu de jeu de données préconstitué fourni. En effet, étant donné les données personnelles présentes sur ces cartes, il est difficile de trouver un jeu de données public sur ce thème. Ainsi, nous avons dû constituer notre base de données grâce à des recherches internet via Google Images. Celle-ci est constituée d'images qui peuvent être de deux types différents de cartes de séjours : des nouvelles cartes de séjours et des anciennes cartes de séjours.

Lors de nos recherches, nous avons été confrontés à plusieurs problèmes. Certaines images étaient erronées ou encore floues laissant des informations partielles ou inexistantes. La qualité de beaucoup d'images étaient basses et donc non exploitables.

Pour finir, nous avons pu avoir un jeu de données de 28 images correctes : 16 anciennes cartes et 12 nouvelles cartes. Les nouvelles cartes étaient plus difficiles à trouver. Une fois ce premier travail effectué nous avons pu passer sous *Python*.

1.2. Outils *Python*

1.2.1. Package *NumPy*

Il est utile pour des opérations rapides sur les objets semblables à des listes de données ou des tableaux. Il sert principalement à créer, manipuler les objets de type *numpy.array* qui sont des listes de listes. On peut appliquer des opérations assez intuitives qu'on ne pourrait pas sur des objets *Python* de type *list* ou *matrix*. (1)

- Fonction *asarray* :

Elle copie des objets comme des listes, listes de tuples, tuples, tuples de tuples, tuples de listes et *ndarrays* vers un objet *numpy.array*.

- Fonction *shape* :
  Elle renvoie les dimensions d'un objet *numpy.array* sous forme de tuple.
- Fonction *array* :
  Elle crée un objet *numpy.ndarray*.
- Fonction *reshape* :
  Elle redimensionne un objet *numpy.array* dans des dimensions données. Il faut qu'elles soient cohérentes avec les dimensions d'origine. Par exemple, un *numpy.array* de dimensions (2,6) peut être redimensionné en des *numpy.array* de dimensions (6,2) ou (12,).
- Fonction *transpose* :
  Elle échange les dimensions d'un *numpy.array*. Plus précisément, elle renvoie une vue transposée de l'objet, ce qui peut permettre de le modifier. S'il a des dimensions de (6,2), la vue transposée aura des dimensions de (2,6). S'il a des dimensions de (3, 6, 2), ce sera (2, 6, 3).

1.2.2. Package *Pandas*

Il permet de créer, manipuler des objets nommés *dataframe* qui sont des tableaux de données avec des index. Chaque colonne est un objet *Series* (structure de données à une seule dimension). (2)

- Fonction *DataFrame* :
  Elle crée un objet de type *pandas.DataFrame*. Elle prend en argument obligatoire *data* où on met un objet rempli de données (conseillé à 2 dimensions max) qui peuvent être de types différents. Cet objet peut être un dictionnaire, un *numpy.array* ou une liste. Il faut bien regarder la documentation à propos de cet argument pour être sûr des bonnes dimensions qu'il prend.

1.2.3. Package *Sklearn*

Il permet de faire du *machine learning* supervisé ou non-supervisé. Il permet aussi de préparer les données et d'évaluer les modèles. Il a beaucoup de sous-packages comme *linear_model*, *model_selection*, *preprocessing* ou *pipeline*. (3)

- Fonction *linear_model.LogisticRegression* :
  Elle importe la classe *LogisticRegression* du sous-package *linear_model* afin de pouvoir créer des objets de cette classe qui auront pour objectif de faire de la régression logistique.
- Méthode *fit* d'un objet *LogisticRegression* :
  Elle ajuste les paramètres du modèle de régression logistique sur les données fournies, afin de prédire au mieux les données à expliquer.
- Méthode *predict* d'un objet *LogisticRegression* :

Elle prédit des données à expliquer grâce à un modèle *LogisticRegression* déjà entrainé.

- Fonction *preprocessing.LabelEncoder* :
  Elle initialise un objet *LabelEncoder* qui est un transformeur de données qui les labellise.
- Méthode *fit* d'un objet *LabelEncoder* :
  Elle entraine un modèle *LabelEncoder* initialisé grâce un *numpy.array* passé en argument et supposé être des données à expliquer.
- Méthode *transform* d'un objet *LabelEncoder* :
  Elle labellise des données passées en argument grâce à un modèle *LabelEncoder* entrainé.
- Fonction *pipeline.Pipeline* :
  Elle initialise un objet de type *Pipeline* avec une liste de couples (nom, modèle) passée en argument, où chaque modèle est une étape de traitement ou transformation de données. Le dernier doit être un modèle de type estimateur (qui peut implémenter les méhodes *fit* et *predict*).
- Méthode *fit* d'un objet *Pipeline* :
  Elle va transformer les données avec chaque étape de la *Pipeline* en utilisant la méthode *fit_transform()* de chaque modèle sur les données respectivement dans l'ordre, et va ajuster les paramètres du dernier modèle.
- Fonction *model_selection.StratifiedKFold* :
  Elle initialise un objet *StratifiedKFold* qui est une méthode de cross-validation de données. Elle prend notamment en argument un nombre de coupes et un booléen qui indique si on veut mélanger les données. La méthode *K-Folds* coupe les données en deux jeux (entrainement, test) plusieurs fois (selon un nombre donné) et va former le modèle en l'entrainant sur chaque coupe. La méthode s*tratified* rajoute en plus la condition que les jeux d'entrainement sont représentatifs de l'ensemble des données (par exemple s'il y 40% de 0 et 60% de 1 dans l'ensemble des données, alors chaque jeu d'entrainement respectera ces proportions).
- Fonction *cross_val_score* :
  Elle permet, en utilisant une méthode de validation croisée, d'évaluer un modèle (ou une *pipeline* de modèles) en prenant en argument des données d'entrainement, un modèle et une méthode cross-validation. Elle renvoie un tableau de scores. On peut ensuite calculer sa moyenne et son écart-type pour visualiser la qualité du modèle.
- Fonction *preprocessing.StandardScaler* :
- Elle initialise un objet *StandardScaler* qui est un transformeur de données qui les normalise entre 0 et 1.
- Méthode *fit* d'un objet *StandardScaler* :

Elle normalise des données passées en argument grâce à un modèle *StandardScaler* entrainé.

- Fonction *model_selection.LeavePOut* :
  Elle initialise un objet *LeavePOut* qui est une méthode de cross-validation des données. Elle permet de couper un jeu de données en un jeu d'entrainement et un jeu de test de taille *p*. Elle va faire autant de coupes qu'il y a de coupes différentes possibles.
- Méthode *get_n_splits* d'un objet *LeavePOut* :
  Elle retourne le nombre de coupes.
- Méthode *split* d'un objet *LeavePOut* :
  Elle retourne deux *ndarray*, un avec les indices des données d'entrainement d'une coupe et un avec les indices des données de test de cette coupe.

1.2.4. Package *Lazypredict*

Il permet d'entraîner rapidement un grand nombre de modèles de *machine learning* avec très peu de lignes de code. Cela permet alors de mettre en évidence les modèles adaptés aux données. (4)

- Fonction *Supervised.LazyClassifier* :
  Elle permet d'initialiser un objet *LazyClassifier* qui lui va appliquer 27 modèles de classification supervisée sur les données.
- Méthode *fit* d'un objet *LazyClassifier* :
  Elle entraine les modèles d'un *LazyClassifier* sur des données d'entrainement puis les teste sur des données de test. Elle retourne la liste des modèles avec leur *accuracy* et la liste des prédictions.

1.2.5. Package *Matplotlib*

Il permet de produire un grand nombre de graphiques simples. (5)

- Fonction *pyplot.pie* :
  Elle permet de créer un camembert avec en argument obligatoire *x* des données sous forme d'une séquence de sommes des parties du camembert. On peut aussi ajouter *colors* la liste des couleurs pour les parties du camembert, *labels* une liste de chaines de caractères sur les étiquettes à afficher pour les données sur le graphique et *labeldistance* la distance à laquelle les étiquettes les données doivent être placées (si *None* elle ne s'affichent pas mais restent stockées pour une utilisation dans la fonction *pyplot.legend*).
- Fonction *pyplot.title* :
  Elle rajoute un titre au graphique grâce à un *string* passé en argument.
- Fonction *pyplot.legend* :

Elle rajoute une légende au graphique (par défaut à gauche de la figure, si l'argument *loc* n'est pas précisé).

- Fonction *pyplot.show* :

  Elle affiche les graphiques créent jusqu'elle.

1.2.6. Package *Statistics*

Il permet d'utiliser un large panel d'indicateurs statistiques. (6)

- Fonction *mean* :

  Elle renvoie la moyenne arithmétique de données passées en argument, comme une liste.

1.2.7. Package *Random*

Il permet de créer des valeurs "aléatoires". (7)

- Fonction *sample* :

  Elle renvoie une liste d'éléments uniques dans une liste, dont on choisit le nombre.

1.2.8. Package *PIL*

Ca veut dire *Python Imaging Library* et il permet de traiter des images sur *Python*. (8)

- Fonction *Image* :

  Elle va convertir un ficher image en un objet *Image* traitable sur *Python.*
- Méthode *open* d'un objet *Image* :

  Elle ouvre une image, ce qui permet d'appliquer des opérations dessus et d'en récupérer des données.
- Méthode *convert* d'un objet *Image* :

  Elle va convertir un objet *Image* en une copie qui respecte les arguments passés. Par exemple avec "*RGB*" l'*Image* sera convertie en données de couleurs primaires.
- Méthode *getpixel* d'un objet *Image* :

  Elle permet d'accéder au pixel d'une *Image* avec les coordonnées de position passées en argument.
- Méthode *size* d'un objet *Image* :

  Elle permet de récupérer les deux dimensions (format pixel) d'une *Image*.

1.2.9. Package *Os*

Il permet d'interagir avec son système d'exploitation, ses fonctionnalités et l'explorateur de fichier. (9)

- Fonction *listdir* :

Avec un chemin de dossier passé en argument, elle retourne les noms de tous les fichiers de ce dossier dans une liste de chaines de caractères.

1.2.10. Package *Pickle*

Il permet de stocker des données de manière efficace car il utilise des protocoles de "sérialisation". Le format des données est fait pour *Python*. (10)

- Fonction *dump* :
  Une fois un fichier *pickle* ouvert, elle permet de convertir de stocker des variables dedans. Le fichier image et les variables sont passées en argument de cette fonction.
- Fonction *load* :
  A l'inverse, elle permet d'importer des données d'un fichier *pickle* pour pouvoir les récupérer dans des variables.

1.2.11. Package *Keras*

Il permet de faire du *deep learning* sur *Python*. Il a plusieurs sous-packages comme *preprocessing.image*, *utils*, *models* et *layers*. (11)

- Fonction *preprocessing.image.ImageDataGenerator* :
  Elle permet d'initialiser une fonction qui permettra de générer une image similaire à une image mais différentes selon plusieurs critères passés en argument sur la rotation, le zoom, la découpe, etc.
- Fonction *utils.load_img* :
  Elle charge un fichier image passé en argument et retourne un objet *PIL Image*.
- Fonction *utils.img_to_array* :
  Elle convertit un objet *PIL Image* en *numpy.array*.
- Méthode *flow* d'un objet *ImageDataGenerator* :
  Elle permet de générer une image selon les conditions d'une fonction *ImageDataGenerator*. Elle prend notamment en arguments un *numpy.array* des données, ensuite le chemin et le format d'image à enregistrer.
- Fonction *models.Sequential* :
  Elle permet d'initialiser un objet *Sequential* qui est un modèle séquentiel de *deep learning*.
- Méthode *add* d'un objet *Sequential* :
  Elle permet d'ajouter une couche (*Dense*) à un modèle *Sequential* initialisé.
- Méthode *compile* d'un objet *Sequential* :

Elle permet, avant l'entrainement de configurer le processus d'apprentissage d'un modèle *Sequential* (initialisé, et avec des couches) selon un optimiseur, une fonction de coût et une liste de métriques passés en argument.

- Fonction *layers.Dense* :
  Elle permet de créer une couche de réseau de neurones.

1.2.12. Package *Scikeras*

Il permet d'utiliser de manière compatible *Keras* avec *Sklearn* notamment à travers un de ses sous-packages *wrappers*. (12)

- Fonction *wrappers.KerasClassifier* :
  En prenant en argument un modèle de type *Keras*, il va pouvoir l'initialiser tel qu'un modèle de classification *Sklearn*.
- Méthode *fit* d'un objet *KerasClassifier* :
  Elle va entrainer un modèle *KerasClassifier* sur des données d'entrainement passées en argument.

1. *Machine Learning*

1.1. Introduction

1.1.1. *Data Preparation*

Pour prédire une variable binaire en utilisant des algorithmes de *machine learning* sous *Python*, il est nécessaire de disposer de variables explicatives numériques en entrée. Il faut donc trouver le moyen de résumer chaque image en quelques valeurs numériques la décrivant : l'étape de *data preparation*. Nous nous sommes donc concentrés sur les colorations RGB des images (Rouge – Vert – Bleu).

Chaque image a un certain nombre de pixels (longueur de l'image × largeur de l'image). Chaque pixel a une valeur de rouge, une valeur de vert et une valeur de bleu. Pour chaque image, on peut donc faire la moyenne de tous les rouges, la moyenne de tous les bleus et la moyenne de tous les verts. On arrive donc à un tableau à quatre colonnes, avec pour chaque image sa catégorie (0/1, soit ancien/nouveau type) et trois valeurs numériques décrivant sa coloration. Nous avons alors eu l'idée de prendre la moyenne pour résumer chaque couleur primaire. On pourrait aussi choisir plutôt la médiane, puis comparer les résultats des deux.

Nous avons alors créé sur *Python* une classe nommé *ColorsData* dont nous allons présenter les objectifs, les attributs, les fonctions et comment on peut créer et utiliser un tel objet.

- **Objectifs** :

  -pouvoir à partir de deux dossiers avec des images, créer une base de données catégorisée binairement;

  -traduire les informations de fichiers images en informations numériques;

  -proposer de manière optionnelle des noms aux catégories;

  -ajouter à sa base de données des nouvelles images;

  -exporter la base de données vers un fichier *pickle*;

  -importer une base de données depuis un fichier *pickle*.

- **Attributs** :

  -*files_path* : un *string*, le chemin du projet (nous avons mis une valeur par défaut);

-*cat_paths* : une liste de deux *strings*, chaque *string* correspond au chemin d'un dossier où sont les images d'une catégorie mais sans le chemin entier du projet (nous avons mis des valeurs par défaut);

-*data_file_name* : un *string*, étant le nom d'un fichier *pickle* qui contient ou contiendra les données, il doit alors se finir par "*.pickle*" (nous avons mis une valeur par défaut);

-*cat_names* : une liste de deux valeurs qui représentent des noms personnalisés décrivant chacune des deux catégories, il faut faire attention à ce que les chemins et les noms personnalisés correspondent bien (nous leur avons mis des valeurs par défauts);

-*data* et *target* : des listes initialisées vides qui sont les deux seuls attributs non-obligatoires lors de la création d'un objet du type de cette classe : la première contiendra les données de variables explicatives et la deuxième les données de la variable à expliquer.

- **Fonctions** :

-*get_one_data* : elle prend en arguments les deux *strings* que sont le nom d'un fichier image et celui d'un chemin, elle retourne une liste de trois valeurs numériques qui décrivent la coloration de l'image comme expliqué précédemment;

-*get_one_data_and_target* : elle prend quasiment les mêmes arguments que la fonction précédente (le chemin doit être ici un sous-chemin d'un dossier d'une catégorie dans le dossier du projet) et utilise aussi les attributs *files*_path, *cat_paths* et la fonction *get_one*_data de l'objet, elle retourne alors un tuple composé d'une liste retournée grâce à la fonction *get_one_data* et d'un entier (0 ou 1) décrivant la catégorie de l'image;

-*add_image* : elle prend les mêmes arguments que la fonction précédente et utilise directement les attributs *data*, *target* de l'objet et la fonction *get_one_data_and_target*, elle permet d'ajouter une image et sa catégorie à la base de données de l'objet, plus précisément elle va récupérer le tuple que cette dernière retourne et elle va alors ajouter le premier élément de celui-ci (la liste décrivant les couleurs de l'image) à la fin de l'attribut *data* de l'objet et ajouter le deuxième élément du tuple (la valeur décrivant la catégorie de l'image) à la fin de l'attribut *target*, elle retourne ces deux attributs;

-*get_data_and_target* : elle prend notamment en argument un *string* nommé *updated_file_name* (dont nous avons choisi de mettre en valeur défaut la même valeur de l'attribut *data_file_name* de l'objet) , peut utiliser directement les attributs *data*, *target*, *cat_paths*, *data_file_name* de l'objet et la fonction *add_image*, et prend également en argument

un booléen nommé *update* (par défaut à *False*), elle permet d'importer ou mettre à jour la base de données de l'objet, plus précisément en fonction de la valeur de celui-ci, elle peut faire deux choses différentes : si l'on souhaite réactualiser les données (*update*=*True*) alors cette fonction va prendre toutes les images issues des deux dossiers de catégories et créer un fichier *pickle* qui contiendra deux listes de même taille (une liste avec pour chaque image une liste de trois couleurs la décrivant, et une liste avec les numéros de catégorie de chacune des images), mais si l'on part du principe qu'il y a déjà un fichier *pickle* avec les données agencées comme décrit précédemment (*update*=*False*) alors cette fonction va récupérer le contenu de fichier censé contenir deux listes de même taille (une liste avec pour chaque image une liste de trois couleurs la décrivant, et une liste avec les numéros de catégorie de chacune des images), dans les deux cas elle va affecter les listes trouvées aux attributs *data* et *target* de l'objet et les retourner.

- **Utilisation** :

  -initialiser un objet de type *ColorsData* en laissant nos valeurs par défaut aux attributs (car nous avons déjà créé un fichier *pickle* avec les données dedans lors de nos travaux sur *Python*);

  -appliquer sur cet objet la fonction *get_data_and_target* sans nouvelles valeurs à la place des valeurs par défaut non plus;

  -extraire les attributs *data* et *target* de cet objet pour une utilisation dans un modèle de *machine learning*.

1.1.2. *Cross-Validation*

Pour essayer de tester les performances d'un modèle de *machine learning*, il va falloir garder des données de côté dans un "jeu de test" pour les tester sur un modèle entraîné sur les autres données, du "jeu d'entraînement" : l'étape de *cross-validation*.

Parmi les méthodes de *cross-validation* existantes, nous nous sommes intéressés au méthodes *LeaveOneOut* (13) et *LeavePOut* (14), dont la documentation est disponible sur le site de *sklearn*.

La première permet de mettre une seule donnée dans le jeu de test et le reste dans le jeu d'entraînement. On peut ensuite faire autant de coupes et de modèles correspondants qu'il y a d'individus/de cartes. Quand l'échantillon est très petit, cette méthode est idéale et on peut la rendre facilement visualisable. La deuxième permet de mettre un nombre *p* de données dans le jeu de test et le reste dans le jeu d'entraînement. On peut aussi boucler pour créer autant de modèles qu'il existe de couples entraînement-test. Il faut aussi que le nombre de données soit petit.

Nous avons donc choisi de garder deux données (images) pour chaque jeu de test. Nous voulions néanmoins avoir une donnée par catégorie différente (un couple nouvelle carte – ancienne carte). Il n'existe pas de méthode toute faite par *sklearn*. Nous avons décidé de baptiser cette méthode "*LeaveTwoDiffOut*" (*leave two different data out*). Nous nous sommes aidés d'une aide proposée sur le forum de *stackoverflow*. (15)

Nous avons alors créé la classe *LeaveTwoDiffOut* dont nous allons présenter notamment les attributs, les fonctions et un exemple d'initialisation et premièrement les objectifs.

- **Objectifs** :

  -à partir d'un jeu de données et d'un modèle de *machine learning*, proposer toutes les coupes possibles selon la méthode décrite précédemment;

  -proposer ces coupes à travers une base de données dont chaque "individu" représenterait une coupe possible et différente des autres avec à l'intérieur précisées : les données explicatives du jeu de d'entrainement de la coupe, les données à expliquer de son jeu d'entrainement, les données explicatives de son jeu de test, les données à expliquer de son jeu de test et les données du jeu de test prédites par le modèle donné;

  -proposer la comparaison (d'égalité) entre les données à expliquer du jeu de test et les prédictions;

  -proposer cette comparaison à travers un texte qui la décrit de manière résumée;

  -représenter cette comparaison à travers un camembert en 2 parties (les prédictions réussies et les prédictions ratées).

- **Attributs** :

  -*model* : un objet de type modèle de classification supervisée *sklearn* ou un objet semblable (contenant des fonctions *fit* et *predict* semblables aux fonctions *fit* et *predict* de ce précédent type d'objet);

  -*data* : (les variables explicatives) une liste (de la même taille que *target*) de listes de trois valeurs numériques décrivant la coloration d'une image comme expliqué précédemment;

  -*target* : (la variable à expliquer) une liste (de la même taille que *data*) d'entiers (parmi 0 et 1) chacun décrivant la catégorie d'une image;

-*cat_names* : une liste de deux valeurs qui représentent des noms personnalisés décrivant chacune des deux catégories, il faut faire attention à l'ordre (nous leur avons mis des valeurs par défaut);

-*splits* : un dictionnaire *Python* initialisé vide qui contiendra la base de données décrite dans les objectifs, c'est le seul qui n'est pas obligatoire lors de la création d'un objet du type de cette classe.

- **Fonctions** :

-*split* : elle utilise les attributs *data*, *target*, *model* et *splits* de l'objet, grâce à ces trois premiers attributs, *splits* (un dictionnaire *Python*) va être rempli de la manière suivante : chaque élément du dictionnaire contient pour une coupe : les données explicatives de son jeu de d'entrainement (de longueur 25), les données à expliquer de son jeu d'entrainement (de longueur 25), les données explicatives de son jeu de test (de longueur 2), les données à expliquer de son jeu de test (de longueur 2) et les données du jeu de test prédites par le modèle donné (de longueur 2) (visuellement :
{0 : {"*X_train*": [], "*X_test*": [], "*y_train*": [], "*y_test*": [], "*y_test_pred*": []},
1 : {"*X_train*": [], "*X_test*": [], "*y_train*": [], "*y_test*": [], "*y_test_pred*": []},
… }), elle retourne ce *splits*;

-*trues_falses* : elle utilise l'attribut *splits* de l'objet (s'il est vide, elle utilise la fonction *split* qui le remplira), elle retourne deux entiers : le nombre de fois ou une catégorie est prédite correctement et le nombre de fois où la prédiction est ratée;

-*description* : elle utilise la précédente fonction et va afficher ses deux valeurs retournées avec un texte explicatif;

-*pie* : elle utilise la précédente fonction et va représenter graphiquement ses deux valeurs retournées à travers un camembert.

- **Utilisation** :

-initialiser un objet *LeaveTwoDiffOut* avec un modèle initialisé (non *fitted*), des données *data* et *target* adaptées (exemple : celles d'un objet *ColorsData*);

-appliquer la fonction *split* (temps d'exécution relativement long);

-appliquer les fonctions *pie* et *description* pour visualiser les résultats.

1.2. Modèle introductif : régression logistique

Au cours de cette première partie, nous avons entrainé un modèle de régression logistique.

- **Apprentissage du modèle** :

  Ce modèle étant introductif à ce projet, celui choisi est un modèle de régression logistique. Il est idéal pour prédire des données binaires. L'un des modèles linéaires traduisant une régression logistique entraînée sur *Python* obtenu est le suivant :

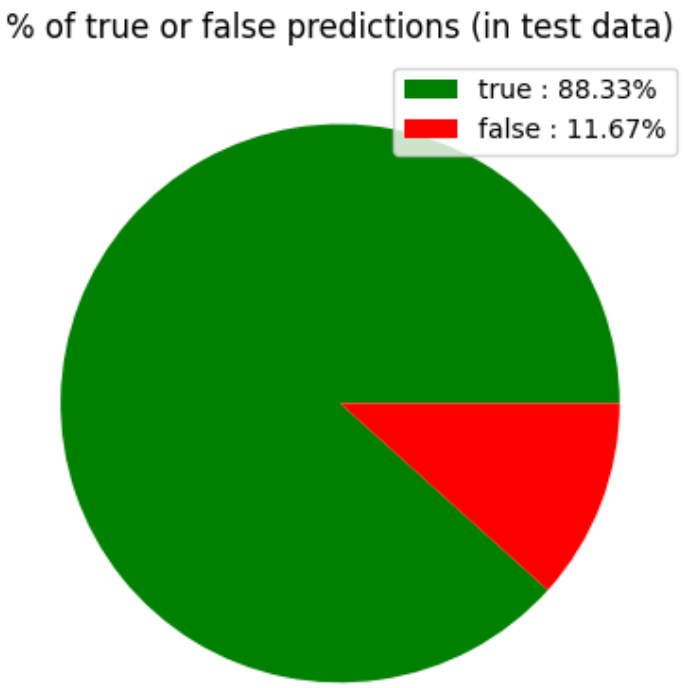
  -0.74174263 - 0.39915359*R + 1.16623781*G -0.75074005*B

  Exemples :

  Pour l'image de catégorie 0/ancienne ayant pour valeurs R, G, B = [205, 192, 214] et pour celle de catégorie 1/nouvelle ayant pour valeurs R, G, B = [160, 153, 141], on obtient respectivement -19.30893819 et 7.97372203. Une valeur négative implique la catégorie 0/ancienne et une valeur positive la catégorie 1/nouvelle. La prédiction de ces exemples est réussie.

- **Test du modèle** :

  Pour valider ce modèle, nous avons donc fait 360 coupes où à chaque fois le couple de jeux entraînement-test est différent. Nous avons représenté la comparaison entre prédiction et vraie donnée à travers ce camembert. Les « *true* » sont toutes les fois où le modèle a prédit la bonne catégorie. Les « *false* » sont les fois où il s'est trompé.



- **Résultat** :

  On a vu que le modèle pouvait se tromper. Nous pensons à deux explications opposées :

  -quand le modèle ne reçoit pas les images qui caractérisent fortement leur catégorie (couleurs très prononcées) dans le jeu d'entraînement, elles manquent alors au modèle qui sera moins strict dans la catégorisation;

  -quand les images caractérisant faiblement leur catégorie (couleurs peu prononcées) passent en jeu de test, le modèle se trompe tout simplement.

Pour cette étape, nous avons créé une classe *ColorsModel*. La motivation était de pouvoir boucler sur une liste de modèles de *sklearn* pour comparer les résultats, mais nous avons trouvé une solution beaucoup plus rapide. Finalement, cette classe censée définir un objet possédant un objet de type modèle de classification supervisée *sklearn* n'apporte pas beaucoup plus qu'un tel objet. Nous l'avons quand même gardée et allons expliciter ses objectifs, ses attributs, ses fonctions et son utilisation.

- **Objectifs** :

  -entraîner un modèle de *machine learning;*

  -afficher les prédictions des données entrainées ou des prédictions sur de nouvelles données;

  -afficher grâce à un *dataframe* ces résultats (avec notamment une comparaison entre données à prédire et prédictions).

- **Attributs** :

  -*model* : un objet de type modèle de classification supervisée *sklearn* (qui n'est pas encore "*fitted*", il est obligatoire et n'a pas de valeur pas défaut;

  -*cat_names* : une liste de deux valeurs qui représentent des noms personnalisés décrivant chacune des deux catégories (il faut faire attention à l'ordre), il est obligatoire (mais nous avons mis une valeur par défaut).

- **Fonctions** :

  -*add_new_cat_names* : elle prend en argument une liste qu'elle va affecter à l'attribut *cat_names* de l'objet qu'elle va retourner;

-*fit* : elle prend en argument une liste de données explicatives et une liste de données à expliquer, elle entraine le modèle attribut de l'objet sur toutes ces données et retourne le modèle (qui est désormais *fitted*);

-*predict* : elle prend en argument seulement des données explicatives, qu'il y ait un individu ou plusieurs, grâce au modèle (qu'il faut avoir entrainé), elle va effectuer une prédiction et la retourner;

-*fit_predict* : elle prend en argument des données explicatives, des données à expliquer et *new_cat_names* une liste de nouveaux noms de catégories (par défaut à *None*), elle va utiliser la fonction *fit* sur ces données passées en argument, puis récupérer la prédiction, ensuite si l'argument *new_cat_names* est vide, elle retourne ces prédictions, sinon elle va retourner et afficher un *dataframe* avec cinq colonnes : les données à expliquer observées, les prédictions (0, 1), les données à expliquer observées mais avec les nouveaux noms, les prédictions mais avec les nouveaux noms et la comparaison entre observations et prédictions.

- **Utilisation** :

  -initialiser un objet avec en modèle "*LogisticRegression()*" de *sklearn*;

  -entrainer le modèle sur des données d'entrainement en appliquant la fonction *fit* sur l'objet qui prend en argument des données *data* et *target* adaptées (exemple : celles d'un objet *ColorsData*).

1.3. Vingt-sept modèles

Nous n'allons pas nous intéresser à l'aspect scientifique du modèle utilisé alors nous allons essayer de boucler sur les données un maximum de types de modèles à la place de la régression logistique. Pour ça, une solution pas mal du tout s'offre à nous : le package *lazypredict* (16) qui propose notamment d'appliquer en quelques lignes, 27 modèles de *sklearn* sur des données. Nous n'avons pas jugé utile de créer une classe.

- **Première application** :

  Nous avons déjà appliqué le "modèle" (qui permet en fait de boucler sur 27 modèles) sur les données. Il propose ensuite un tableau avec en lignes les noms de classifieur et en colonnes des indicateurs de qualité ('Accuracy', 'Balanced Accuracy', 'ROC AUC', 'F1 Score'). Il utilise donc tous ces algorithmes *sklearn*, nous n'avons pas vérifié par algorithme s'il y avait des

mauvais traitements des données et avons fait confiance au package. Le tableau et ses indicateurs changeaient à chaque lancement et donc on ne pouvait pas déterminer un meilleur algorithme ni même une liste.

- **Deuxième application** :

  Nous avons appliqué une nouvelle fois la méthode *LeaveTwoDiffOut* pour chacun des 27 modèles. Mais les résultats des indicateurs étaient logiquement extrêmes : précisément 1, 0.5, et 0. Néanmoins, nous avons repéré que 25 modèles avaient eu au moins une fois des indicateurs parfaitement performant, et 2 autres jamais. On a fait la moyenne de l'*accuracy* selon le modèle et obtenu des résultats très hétérogènes. Lors de l'un de nos lancements, certains modèles (*RandomForestClassifier*, *ExtraTreeClassifier*, *BaggingClassifier*) ne dépassaient même pas les 50% de prédictions réussies, d'autres arrivaient à 85% (*CalibratedClassifierCV*, *LinearDiscriminantAnalysis*, *LinearSVC*, *RidgeClassifier*, *RidgeClassifierCV*).

- **Utilisation** :

  -initialiser un objet classifieur avec la fonction *lazypredict.Supervised.LazyClassifier* avec en arguments : *verbose=0*, *ignore_warnings=True*, *custom_metric=None*);

  -utiliser des objets *ColorsData*, ColorsModel et *LeaveTwoDiffOut* (voir parties "Utilisation" de ces objets);

  -boucler un nombre de fois égal au nombre de *splits* d'un objet *LeaveTwoDiffOut* (qui a été initialisé et a lancé sa fonction *split*);

  -initialiser une liste vide qui prendra après chaque modèle;

  -à chaque tour de la boucle : récupérer un élément du grand dictionnaire *Python splits*, c'est un dictionnaire de cinq éléments (*X_train*, *X_test*, *y_train*, *y_test*, *y_test_pred*) qu'il faudra passer en arguments d'une fonction nommée *split* de l'objet classifieur (déclaré au tout début), on récupèrera en sortie de cette fonction deux objets dont le premier qu'on ajoutera à la fin de la liste de modèles;

  -transformer cette liste en un *dataframe* en la passant argument d'une fonction *pandas.concat;*

  -manipuler ce *dataframe* pour afficher (voire représenter graphiquement) ses données, comme le nombre de types de modèles différents, le nombre de coupes ou le nombre modèles différents;

  -afficher quels types de modèles ont déjà eu une *accuracy* de 1 (100% de prédictions réussies), afficher la moyenne de l'*accuracy* selon le modèle.

2. *Deep Learning*

2.1. Introduction

2.1.1. *Data Preparation*

Pour cette partie, nous avons cherché à manipuler différentes notions autour du *deep learning*. L'une des différences avec le *machine learning* est qu'il peut prendre en données explicatives des objets plus complexes, notamment des listes de nombres, des listes de listes de nombres, etc. Nous allons alors pouvoir soutirer beaucoup plus d'informations de nos images de cartes.

Nous avons alors créé une classe *ArrayData* inspirée de notre *ColorsData*. Nous allons en présenter en premier lieu ses objectifs ensuite ses attributs et fonctions et enfin comment l'utiliser.

- **Objectifs** :

  -extraire un maximum de données pour chaque image;

  -créer une base de données avec données explicatives et données à expliquer;

  -adapter ces données à une utilisation pour un modèle de réseau de neurones convolutif sur *Python*.

- **Attributs** :

  -*files_path* : un *string*, le chemin du projet (nous avons mis une valeur par défaut);

  -*cat_paths* : une liste de deux *strings*, chaque *string* correspond au chemin d'un dossier où sont les images d'une catégorie mais sans le chemin entier du projet (nous avons mis des valeurs par défaut);

  -*new_sizes* : un tuple de deux entiers égaux plutôt petits qui sont la longueur et la largeur avec laquelle le modèle va redimensionner les images (nous avons mis 32 par défaut) ;

  -*data* et *target* : des listes initialisées vides qui sont les deux seuls attributs non-obligatoires lors de la création d'un objet du type de cette classe : la première contiendra les données de variables explicatives et la deuxième les données de la variable à expliquer.

- **Fonction** *get_data_and_target* :

  Nous nous sommes contentés d'une seule fonction pour ce type d'objet. Elle ne prend pas d'argument et retourne les attributs *data* et *target* de l'objet qu'elle a permis de remplir en s'aidant des autres attributs notamment. Elle permet d'ignorer si l'image finit par ".*png*" ou si le fichier s'appelle "*Keras_photos*" (voir plus tard pourquoi).

- **Utilisation** :

  -initialiser un objet *ArrayData* en laissant nos paramètres par défaut ;

  -lancer la fonction *get_data_and_target* sur l'objet ;

  -récupérer *data* et *target* de l'objet, et regarder ses dimensions.

2.1.2. *Data Augmentation*

Pour faire du *deep learning*, notamment grâce à un réseau de neurones convolutif, il peut être intéressant d'augmenter la base de données grâce à une base de données que l'on a déjà. C'est faisable grâce au package *keras*. Nous avons donc fait deux fonctions dans un fichier *Python* : *one_data_augmentation* et *loop_data_augmentation*.

- **Fonction** *one_data_augmentation* :

  Elle prend en arguments :
  -*image_file* : un *string* qui est un fichier image (pas de valeur par défaut);
  -*cat* : un entier qui est l'un des deux numéros de catégorie de l'image (pas de valeur par défaut) ;
  -*number_photos* : un entier qui est le nombre d'images que l'on veut tirer d'une image (10 par défaut) ;
  -*files_path* : un *string*, le chemin du projet (nous avons mis une valeur par défaut);

  -*cat_paths* : une liste de deux *strings*, chaque *string* correspond au chemin d'un dossier où sont les images originelles d'une catégorie mais sans le chemin entier du projet (nous avons mis des valeurs par défaut);
  -*keras_photos_path* : un *string* qui correspond au nom d'un dossier où devront être les nouvelles images d'une catégorie et sans le chemin entier du projet (nous avons mis une valeur par défaut);

  Elle ne retourne rien mais crée un nombre (*number_photos*) d'images un peu différentes d'une image, dans un nouveau dossier.

- **Fonction** *loop_data_augmentation* :
  Elle ne prend pas d'arguments mais a été créée seulement pour éviter de relancer ce qu'elle fait par mégarde. Elle va boucler la fonction précédente sur toutes les images des deux dossiers d'images originelles, ce qui donnera deux nouveaux dossiers avec des nouvelles images.

- **Utilisation** :

  -créer deux dossiers vides nommés "*Keras_photos*" dans chacun des deux dossiers d'images;

  -lancer une fois la fonction *loop_data_augmentation*;

  -récupérer, préparer des nouvelles données à partir de ces images, par exemple en créant un objet *ArrayData* en spécifiant bien des nouveaux attributs.

2.2. Réseau de neurones convolutif

Le modèle de prédiction de *deep learning* va alors être un réseau de neurones convolutif (Convolutional Neural Network). Il fait partie de la catégorie des réseaux neurones et il est réputé pour être performant dans la classification d'images. On ne passera pas de temps sur l'explication scientifique des réseaux de neurones. Nous avons créé une classe d'objet nommé *CNNModel* inspiré de *ColorsModel*. Nous allons en expliquer les objectifs puis décrire ses attributs et fonctions. Pour un exemple d'utilisation, nous avons dû utiliser cette classe et ses fonctions sur *Pycharm* à cause d'incompatibilité entre le package *tensorflow* et *Jupuyter Notebook*.

- **Objectifs** :
  -entraîner un modèle de *deep learning;*
  -afficher les prédictions sur de nouvelles données préparées;
  -afficher grâce à un *dataframe* les résultats (avec notamment une comparaison entre données à prédire et prédictions).
- **Attributs** :
  -*data* et *target* : les attributs (les seuls obligatoires) de type liste *data* et *target* d'un objet *ArrayData* décrivant chacun respectivement les données de variables explicatives et la deuxième les données de la variable à expliquer;
  -*dim_1*, *dim_2*, *dim_3* et *dim_4* : les quatre dimensions entières de *data*;
  -*X_y_prepared* : un booléen qui décrit si les données ont été bien préparées pour être entrainées;
  -*sets* : un dictionnaire initialisé vide qui aura quatre *numpy.*array sur les données explicatives du jeu de d'entrainement, les données à expliquer du jeu d'entrainement, les données explicatives du jeu de test et les données à expliquer du jeu de test;

-*sets_detailled* : un *pandas.DataFrame* initialisé vide qui aura en plus de *sets* des colonnes plus descriptives sur les résultats obtenus après entrainement sur de nouvelles données explicatives;

-*pipeline* : initialisé *None*, elle centralisera les différentes opérations de transformation;

-*cvs* : initialisé *None*, il aura le score de cross-validation du modèle.

- **Fonctions** :

-*prepare_X_y* : elle ne prend aucun argument, elle se sert des attributs *target*, *data*, et des quatre dimensions de *data*, elle renvoie *X* un *numpy.array* contenant les données de *data* mais redimensionné en deux dimensions (le nombre d'images, le nombre de pixels (32×32) × le nombre de couleurs (3)) et *y* un *numpy.array* sur les données de *target* mais qui ont été bien converties en entier à l'aide de la fonction *transform* du package *LabelEncoder*;

-*split* : elle prend en argument l'entier *n_test* initialisé à 2 et utilise la première dimension de *data* (le nombre d'individus), elle met dans deux *numpy.array* ceux retournés par la fonction *prepare_X_y*, elle prend un nombre (*n_test*) d'index aléatoires et met dans deux listes nommées *test_index* et *train_index* respectivement les quelques index choisis aléatoires et les autres, enfin elle va mettre dans l'attribut dictionnaire *sets* de l'objet quatre *numpy.array* : "*X_train*" les données de *X* aux index *train_index*, "*X_test*" les données de *X* aux index *test_index*, "*y_train*" les données de *y* aux index *train_index* et "*y_test*" les données de *y* aux index *test_index*;

-*fit* : elle prend quatre arguments, le booléen *do_split* par défaut à *False*, *n_test* par défaut à 2, *verbose* par défaut à *False* et *new_cat_names* par défaut à *None*, elle peut être découpée en trois étapes. Premièrement, si *do_split* est *True* la fonction *split* est lancée sur l'objet avec le *n_test* passé en argument ; sinon on prend dans *X* et *y* la sortie de la fonction *prepare_X_y* et on les met dans les éléments *X_train* et *y_train* de l'attribut dictionnaire *sets* de l'objet (ses éléments *X_test* et *y_test* sont initialisés vides). Deuxièmement on initialise un modèle *Sequential* du package *keras.models* auquel on ajoute une première couche d'unité 32×32×3 et *relu* en fonction d'activation, une deuxième d'unité 32×32×3/2 et une troisième d'unité 1 et de fonction d'activation *sigmoid* ; on compile le modèle en tant que classification binaire (*binary_crossentropy*) avec un optimiseur de type *adam* et l'*accuracy* en fonction métrique ; on met ce modèle dans un objet *KerasClassifier* du package *scikeras.wrappers* avec 10 en cycle de traitement des données et 5 en *batch size* ; ensuite on *fit* une *pipeline* sur les données d'entrainement grâce à un objet *Pipeline* du package *sklearn.pipeline* qui a un objet *StandardScaler* du package *sklearn.preprocessing* qui va normaliser les données et on la met dans l'attribut *pipeline* de l'objet ; enfin, un objet *StratifiedKfold* du package *sklearn.model_selection* avec un nombre de coupes de 10 et un mélange des données activé

est mis en argument dans une fonction *cross_val_score* du package *sklearn.model_selection* avec la *pipeline* et les données d'entrainement, elle permettra d'évaluer la qualité du modèle et sera mise dans l'attribut *cvs* de l'objet. Troisièmement, si *do_split* est vrai, on crée un *dataframe* dans l'attribut *sets_detailled* avec en colonnes les données à expliquer observées, celles prédites, les probabilités (arrondies) que le résultat soit 0, les probabilités que 1 et les comparaisons entre l'observation et la prédiction, si l'argument *cat_names* était rempli on rajoute aussi deux colonnes calqués sur les deux premières mais qui remplacement 0 et 1 respectivement par deux chaînes de caractères mises dans une liste (*cat_names*), enfin si *verbose* est vrai, la fonction affichera les moyenne et écart-type de *cvs* en pourcentage et si *do_split* est vrai, elle affichera aussi le nombre d'individus testés et les nombres de bonnes et mauvaises prédictions.

- **Utilisation** :

  -initialiser un objet *ArrayData* en laissant nos paramètres par défaut mais avec les dossiers où sont les images générées par *Keras*;

  -lancer la fonction *get_data_and_target* sur l'objet ;

  -intialiser un objet *CNNModel* avec en *data* et *target* les *data* et *target* de l'objet *ArrayData* ;

  -lancer sur cet objet la fonction *fit* avec un *do_split* vrai, un *n_test* de 30, un *verbose* vrai et des *new_cat_names* (*old* et *new) ;*

  -la fonction affichera la qualité du modèle, l'écart-type, la taille du jeu de test, le nombre de prédictions sur les données réussies et le nombre de ratées.

Voici un exemple de résultats :

```
Qualite : 96.61% (4.33%)
number of images to test: 3
good (True) or bad (False) predictions :
 True    3
```

3. Conclusion

En résumé, nous avons constaté que le deep learning était particulièrement adapté pour traiter des données issues d'images et en extraire des informations utiles. Cependant, avec notre jeu de données de taille limitée, nous avons observé que le machine learning parvenait à éviter le surapprentissage de manière plus efficace que le deep learning. Pour choisir les méthodes les plus appropriées pour augmenter notre base de données, il serait judicieux de disposer de plus d'exemples de cartes de séjour scannées ou photographiées de manière incorrecte. De plus, il pourrait être intéressant d'utiliser des techniques de deep learning en amont de l'étape de prédiction, afin de localiser les bords de la carte sur une image de celle-ci. Cela pourrait s'avérer utile pour améliorer la qualité de nos prédictions.

Bibliographie

1. NumPy Documentation. [En ligne] https://numpy.org/doc/.

2. Pandas. [En ligne] https://pandas.pydata.org/pandas-docs/stable/index.html.

3. Scikit-learn. [En ligne] https://scikit-learn.org/stable/index.html#.

4. Lazy Predict. [En ligne] https://lazypredict.readthedocs.io/en/stable/index.html.

5. Matplotlib. [En ligne] https://matplotlib.org/stable/index.html.

6. Statistics. [En ligne] https://docs.python.org/3/library/statistics.html.

7. Random. [En ligne] https://docs.python.org/3/library/random.html.

8. Pillow. [En ligne] https://pillow.readthedocs.io/en/stable/index.html.

9. Python os.listdir() method. [En ligne] https://www.javatpoint.com/python-os-listdir-method#:~:text=The%20listdir%20%28%29%20function%20is%20a%20function%20provided,current%20working%20directory%20%28where%20the%20program%20is%20present%29..

10. Pickle. [En ligne] https://docs.python.org/fr/3/library/pickle.html.

11. Tf.keras. [En ligne] https://www.tensorflow.org/api_docs/python/tf/keras.

12. Scikeras.wrappers.KerasClassifier. [En ligne] https://www.adriangb.com/scikeras/stable/generated/scikeras.wrappers.KerasClassifier.html.

13. Sklearn.model_selection.LeaveOneOut. [En ligne] https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.LeaveOneOut.html.

14. Sklearn.model_selection.LeavePOut. [En ligne] https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.LeavePOut.html.

15. How to split data with leave one pair out cross validation (LeavePOut) for binary classification? [En ligne] https://stackoverflow.com/questions/63705004/how-to-split-data-with-leave-one-pair-out-cross-validation-leavepout-for-binar.

16. Lazy Predict – Best Suitable Model for You. [En ligne] https://www.analyticsvidhya.com/blog/2021/05/lazy-predict-best-suitable-model-for-you/.