Identification d'oiseaux à partir d'images

2024

Un projet réalisé dans le cadre de la formation Datascientest :

Chef de projet en intelligence artificielle – data science
Développer une solution d'intelligence artificielle (machine et deep learning)

Armand BENOIT

Maxence REMI-HAROCHE

Gregory PECHIN

Yoni EDERY

DataScientest • com

Table des matières

I. Introduction

Le projet d'identification d'oiseaux à partir d'images vise à automatiser la reconnaissance des espèces d'oiseaux en utilisant des techniques d'intelligence artificielle. De nombreuses bases de données de photos d'oiseaux existent déjà et sont disponibles gratuitement sur internet.

Après avoir récupéré un premier set de données, nous allons l'explorer afin de vérifier l'état de ses données et déterminer ce qu'on peut en faire. Nous allons également produire plusieurs graphiques afin de faire apparaître les caractéristiques les plus pertinentes du dataset et des images.

Dans une seconde partie, nous nous pencherons sur plusieurs pistes visant à prédire l'espèce d'un oiseau à partir d'une image. Dans un premier temps, nous allons mettre en production les scripts permettant le formatage des données. Nous explorerons également plusieurs manières de changer la distribution des données.

Nous explorerons ensuite les différentes méthodes pour réaliser des modèles d'apprentissage automatique. Nous utiliserons des algorithmes de machine learning en nous basant sur la distribution des couleurs d'une image. Même si nous savons que ce modèle sera moins performant que la plupart des modèles de deep learning, nous trouvons pertinent de constater les possibilités et limites du machine learning dans ce cas d'étude. Nous nous intéresserons ensuite à la création de modèles *from scratch*. Enfin, plusieurs modèles de transfer learning seront implémentés (VGG16, EfficientNet…).

Une fois le meilleur modèle implémenté et entraîné, nous développerons un script permettant d'appliquer la méthode GradCam à nos prédictions afin de vérifier et d'expliquer leurs qualités et limites.

L'objectif principal du projet sera alors atteint. Nous avons cependant plusieurs pistes pour approfondir le sujet : augmentation du nombre d'espèces d'oiseaux, connexion à une ou plusieurs API de sites d'ornithologie permettant le téléchargement continu de nouvelles données, automatisation du nettoyage des nouvelles données ou réentraînement du modèle choisi.

Dans ce rapport, nous allons détailler les différentes étapes du projet, les défis rencontrés et les résultats obtenus.

II. Récupération et exploration des données

1. Collecte des données

La première étape consiste à collecter un ensemble de données d'images d'oiseaux. Nous avons choisi le dataset "BIRDS 525 SPECIES- IMAGE CLASSIFICATION" disponible sur le site kaggle.com. Il contient des images étiquetées pour 525 espèces d'oiseaux. Le dataset est subdivisé en 3 répertoires : *test*, *valid* et *train*. Les deux premiers contiennent 5 images par espèce. Pour le set d'entraînement, le nombre d'images par oiseau est variable.

2. Contrôle des données

Le dataset téléchargé est déjà divisé en trois sets d'images, eux-mêmes classés par espèces d'oiseaux.

Plusieurs opérations sont nécessaires :

1. Vérification de la présence de doublons et le cas échéant, suppression
2. Vérification visuelle des images
3. Analyse des metadatas

Doublons

La première étape est de vérifier la présence de doublons dans le dataset. Cela pourrait causer des problèmes de sur-apprentissage lors de l'entraînement du modèle.

Constatant que le nom de tous les fichiers est composé exclusivement de digits, nous avons commencé par les isoler et vérifier qu'aucun ne contenait de caractère alphabétique ou de symbole. Cela aurait pu mettre en évidence la présence d'un terme tel que "- copie" ou "(1)" démontrant le dédoublement d'un fichier. Cette méthode n'a retourné aucun fichier suspect.

Cette méthode n'est cependant pas satisfaisante car elle ne permet pas de discriminer un doublon portant un nom de fichier légitime. Pour pallier ce problème, la première idée a été de réaliser une analyse des pixels de chaque image. Cependant, avant de la mettre en place, nous avons pensé à une solution beaucoup plus simple : calculer le hash (signature) de chaque fichier et vérifier qu'il est unique au sein du dataset. Aucun doublon n'a été relevé.

Vérification visuelle

Une étape inévitable lors du nettoyage d'un dataset d'images est la vérification visuelle. Le dataset étant relativement petit, nous avons décidé d'afficher l'intégralité des images. Le fichier check_visual.ipynb affiche l'intégralité des images de l'un des trois sets (*valid*, *train*, *test*). Il n'aura alors fallu qu'une dizaine de minutes pour passer grossièrement en revue la totalité des images. Nous avons pu confirmer qu'aucune n'était corrompue (noire, brûlée…).

Pour un dataset de très grande taille, nous aurions affiché seulement quelques images de chaque classe afin de savoir sur quel genre de fichier nous travaillions et vérifier que le dataset contienne bien des images d'oiseaux. Par la suite, nous aurions développé un algorithme afin de vérifier automatiquement l'intégrité de toutes les images.

Analyse des metadatas

Les metadatas d'une image donnent plusieurs informations pertinentes permettant d'analyser sa composition. Nous nous intéresserons ici au mode de chaque image ainsi qu'à ses dimensions.

La bibliothèque Pillow de python permet d'extraire très facilement les metadatas d'une image. En quelques lignes, nous créons un fichier CSV contenant le dataframe des metadatas de toutes les images.

Dimensions des images

La colonne size, qui contient le doublon hauteur-largeur, nous permet de voir qu'il y a 212 tailles d'images différentes sur la totalité du dataframe. Sur une base de données de près de 90.000 images, cela peut sembler assez peu, mais tout dépend de la répartition de ces tailles.

En séparant ce comptage par set, nous constatons que les sets *valid* et *test* ont chacun six tailles différentes.

En regardant la colonne size du dataframe, on constate que la taille 224*224 pixels est majoritaire. On cherche donc le nombre d'images par set dont la taille est différente de 224*224. On constate qu'il y en a cinq sur les sets de test et de validation et 201 sur le set de *train*. Nous savons que pour chaque classe, les sets de test et de validation proposent cinq images. On peut alors supposer que ce sont les cinq images d'une seule classe qui posent problème.

Il ne reste plus qu'à extraire dans un nouveau dataframe les images avec des tailles différentes de 224*224. Un groupby birdname et set permet de voir que l'hypothèse n'est pas tout à fait juste : la classe Plush Crested Jay pose majoritairement problème. Cependant, une image de la classe Loggerhead Shrike est également présente dans le nouveau dataframe.

La grande majorité des images du dataset sont des carrés. Si les images ayant des dimensions différentes de 224*224 sont des carrés, il sera facile de les conserver en les redimensionnant. Dans le cas contraire, nous décidons qu'avec un ratio hauteur/largeur supérieur à 1.2 ou inférieur à 0.8, l'image n'est pas récupérable.

La photo de la classe Loggerhead Shrike a un ratio de 1.15, nous pouvons la redimensionner facilement avec Pillow.

Concernant la classe Plush Crested Jay, nous constatons que 25% des images ne sont pas redimensionnables. C'est une perte importante. Notre dataset ne compte que 525 espèces d'oiseaux, ce qui est loin d'être exhaustif (5% des 9700 espèces connues dans le monde). Nous décidons donc de que supprimer la classe Plush Crested Jay a un impact négligeable.

Mode

Il pourrait être embêtant pour notre entraînement que les images n'aient pas toutes le même mode, en particulier si la distribution est déséquilibrée. La fonction value_counts appliquée à la colonne mode nous indique que toutes les images sont bien en RVB.

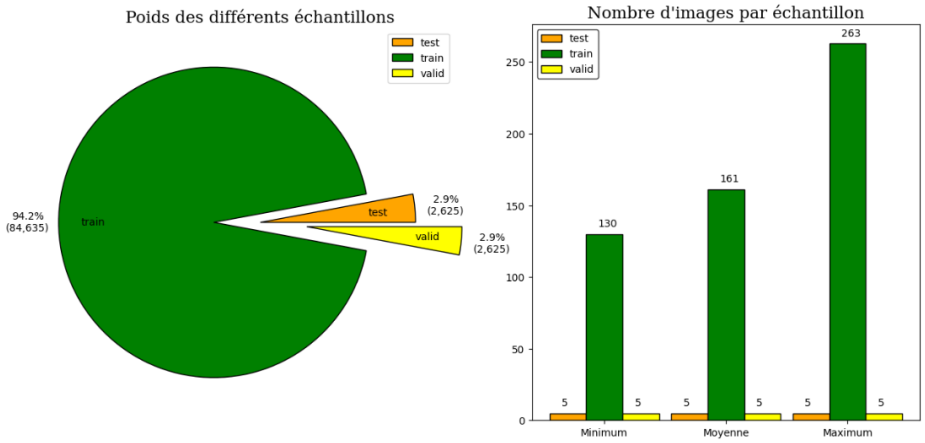
3. Premières analyses exploratoires

Nous avons créé plusieurs graphiques à partir des données disponibles afin d'évaluer les premières actions à mettre en place lors du pré-processing. Cette exploration a été réalisée en parallèle du nettoyage des données décrit plus tôt, c'est pourquoi on compte 525 classes et non 524 suite à la suppression de l'une d'entre elles.

Distribution des images parmi les sets

Le graphique suivant met en évidence un problème majeur de la structure initiale du dataset. Les sets de test et de validation sont beaucoup trop petits par rapport au set d'entraînement. Ils représentent chacun 2.9% du total des données. Une bonne répartition se rapproche généralement de 15% chacun. Il faudra donc fusionner les sets afin de les redistribuer.

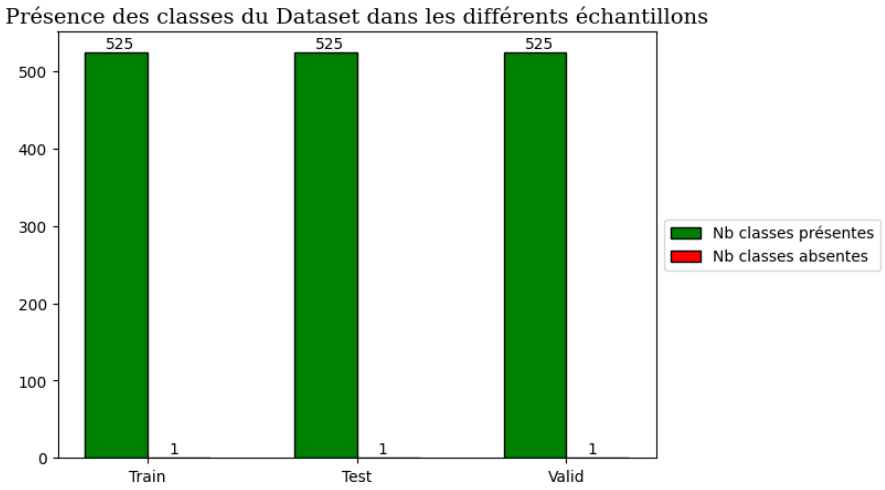
On constate également que le nombre d'images par classe pour les échantillons de test et de validation est toujours le même (5), alors que celui pour l'échantillon d'entraînement varie selon les classes. Il faudra se demander si on extrait aléatoirement 15% de l'ensemble du dataset, ou si on prend 15% de chaque classe.



Cohérence des classes entre les différents jeux de données

Après avoir supposé que toutes les classes étaient correctement représentées dans les trois sets, nous avons décidé de vérifier cette hypothèse. Le graphe suivant montre qu'une erreur s'est glissée quelque part : en se basant sur les noms des dossiers, il y a 526 classes dans le dataset au lieu de 525.

Nombre de classes uniques du Dataset (train + test + valid) : 526



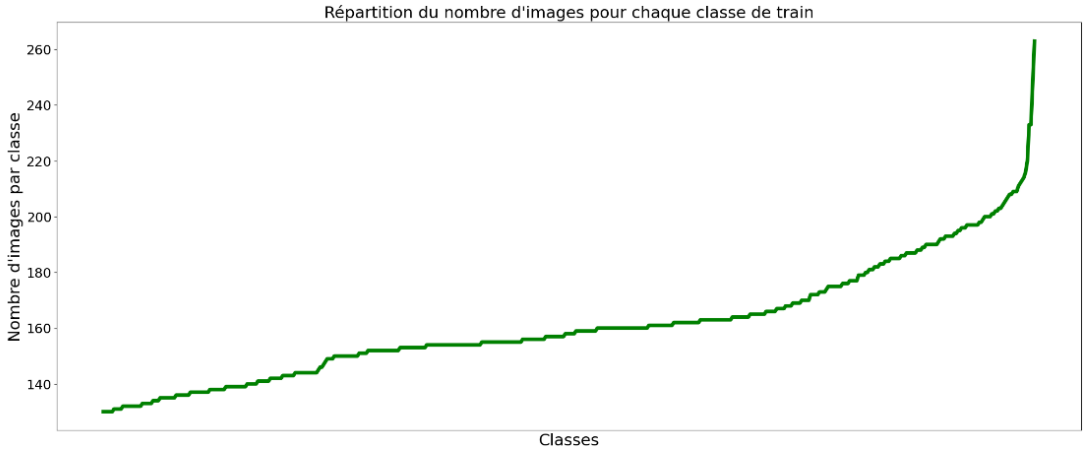
Une analyse des valeurs manquantes nous permet de détecter les incohérences de classes entre les différents échantillons. Nous constatons qu'une seule classe pose souci. Le dossier "PARAKETT AUKLET" de l'échantillon de test et d'apprentissage sera donc à renommer lors du pré-processing.

classe	pres_test	pres_train	pres_valid
PARAKETT AUKLET	1	1	0
PARAKETT AUKLET	0	0	1

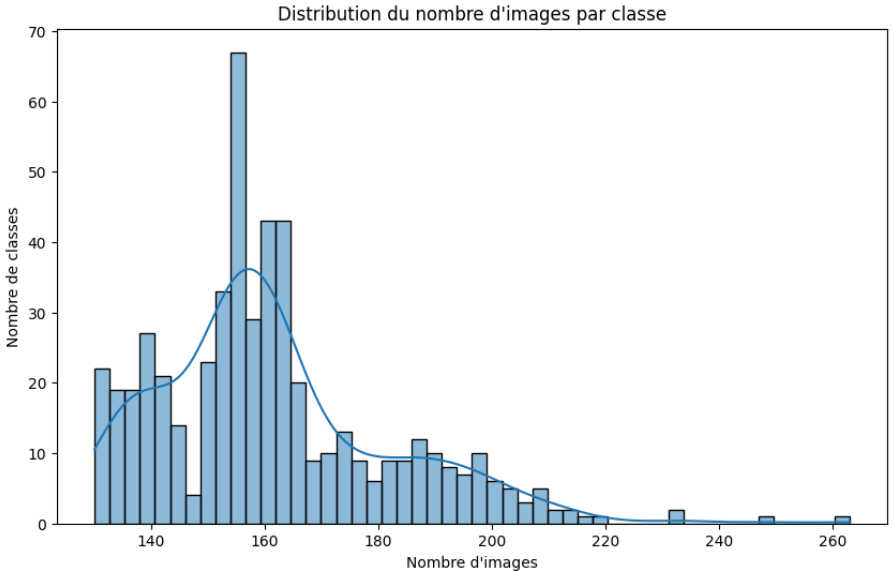
Nombre d'images par classe

Les graphes suivants montrent un déséquilibre significatif entre les classes dans le set *train*. Nous décidons d'ignorer ici les sets *test* et *valid* car ils contiennent chacun cinq images par classe. La plupart des classes sont représentées par un nombre d'images allant de 150 à 170, mais certaines sont sous-

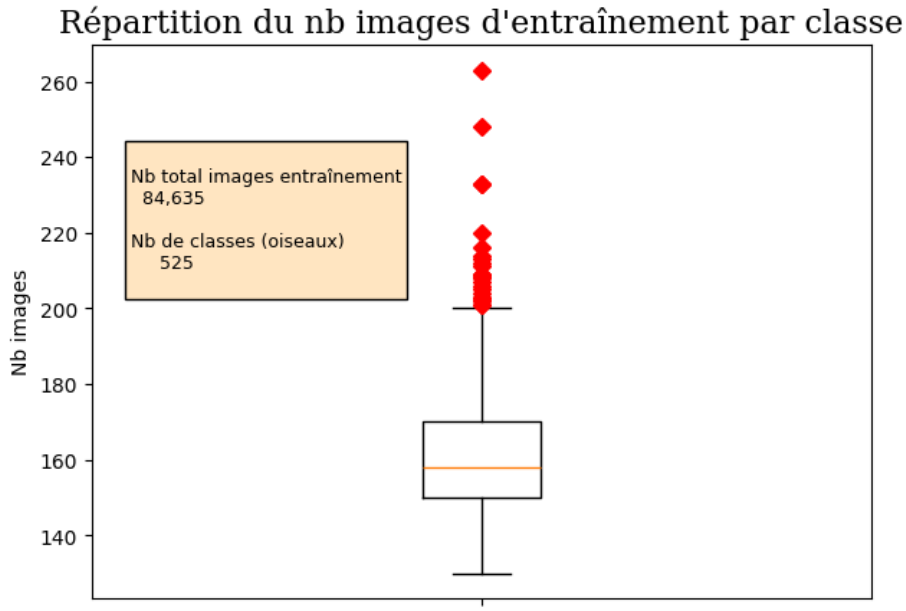
représentées (minimum à 130), et quelques-unes sur-représentées (maximum à 263). Il faudra ré-échantillonner le dataset. On devra alors décider si l'on préfère l'oversampling, l'undersampling, ou une combinaison des deux.



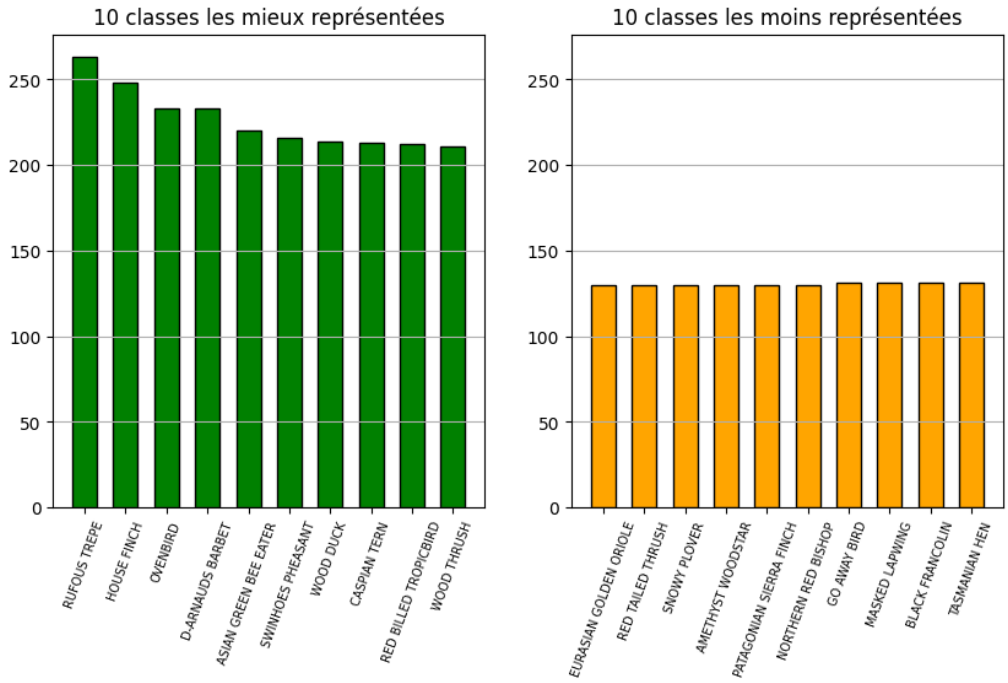
Ce graphique montre que la répartition du nombre d'images entre les classes est très hétérogène, avec quelques valeurs très élevées.



Ce graphique indique que la majorité des classes ont un nombre d'images compris entre 150 et 170, tandis que le reste est inférieur ou supérieur à cette plage. Il met en évidence la rareté des classes dépassant les 200 images.



Ce graphique permet de mettre plus en avant les valeurs extrêmes. On voit aussi que la médiane est de 158 images par classe et que la moitié des classes contiennent entre 150 et 170 images.

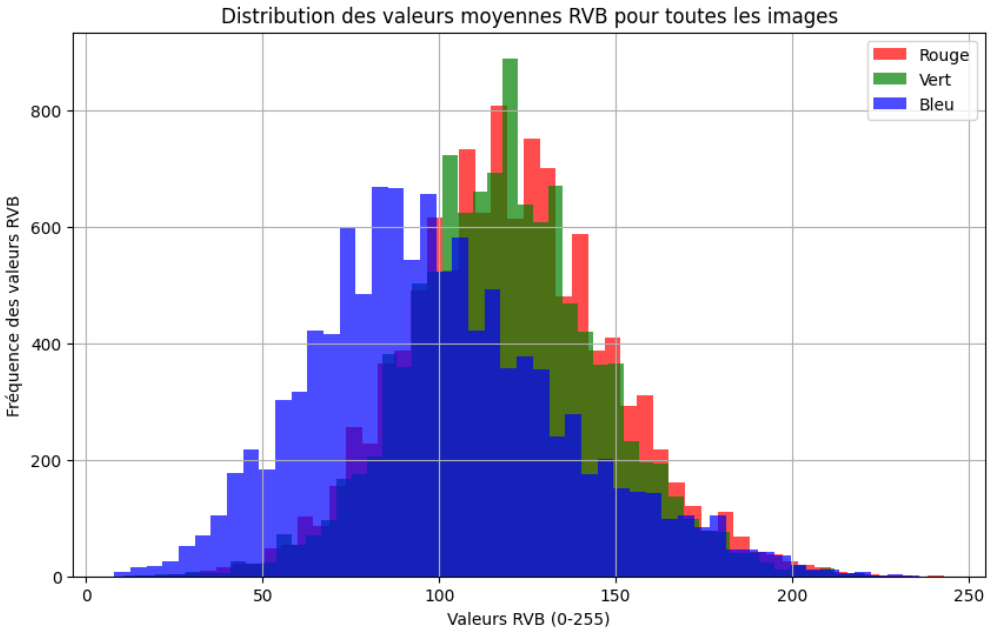


Enfin, on voit ici que de nombreuses classes sont représentée par le minimum d'observations, alors que pour les plus représentées, le nombre d'observations par classe décroît rapidement.

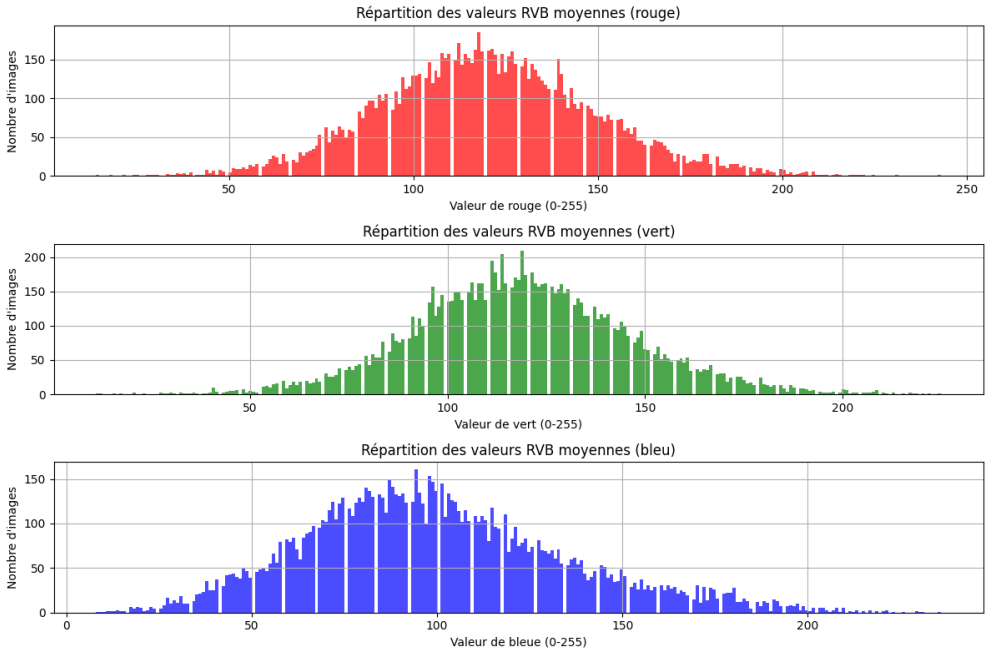
4. Analyse des composantes RVB

Dans cette étape, nous cherchons à analyser les répartitions des couleurs RVB des images. Par commodité, l'analyse est d'abord réalisée sur le dataset de test (images originales, sans détourage de l'oiseau).

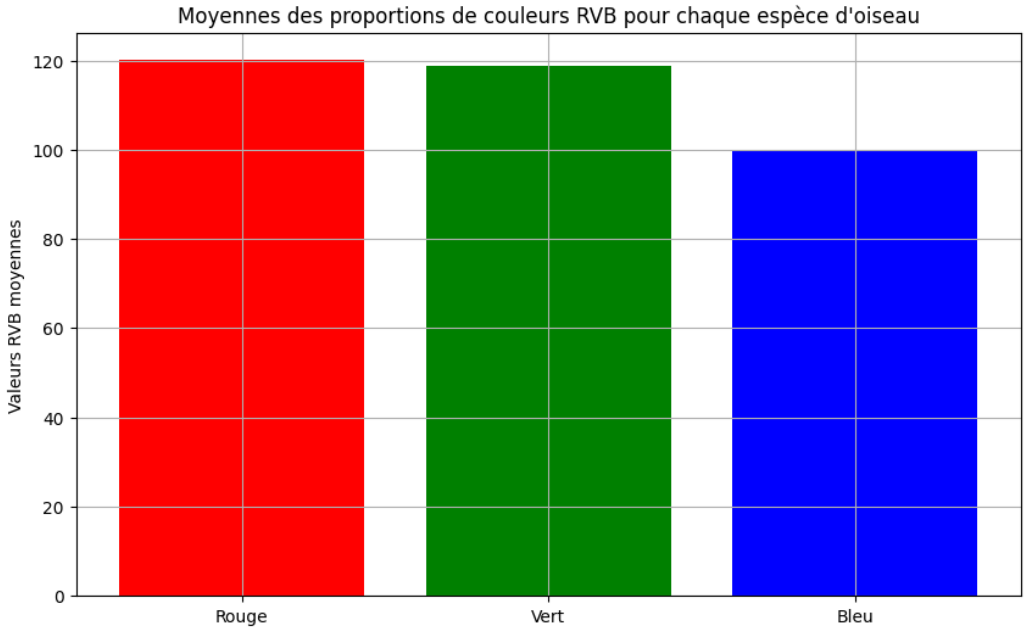
Tout d'abord, nous cherchons à connaître la distribution des valeurs RVB moyennes pour l'ensemble de nos images. Nous obtenons l'histogramme suivant :



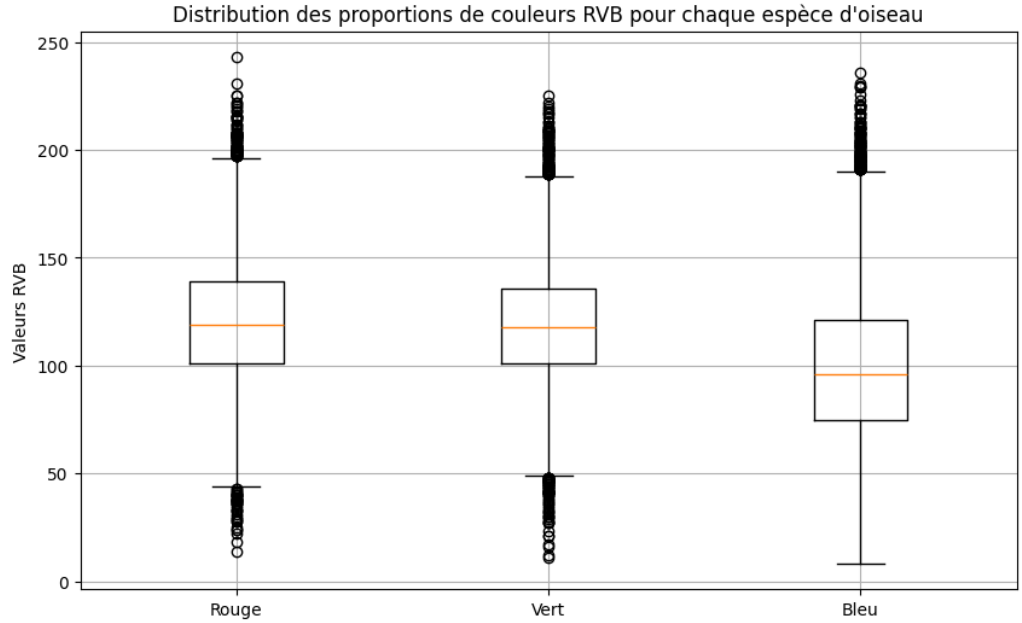
Pour une visualisation plus claire, nous pouvons séparer les composantes sur trois graphiques différents :



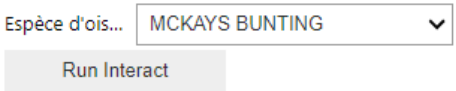
Dans une autre approche, nous calculons la moyenne de toutes les valeurs RVB pour toutes les espèces d'oiseaux et créons un diagramme à barres pour visualiser ces moyennes. Ceci nous donne un aperçu des couleurs dominantes sur l'ensemble des espèces :



Pour la distribution, le choix d'une boîte à moustache est plus judicieux :



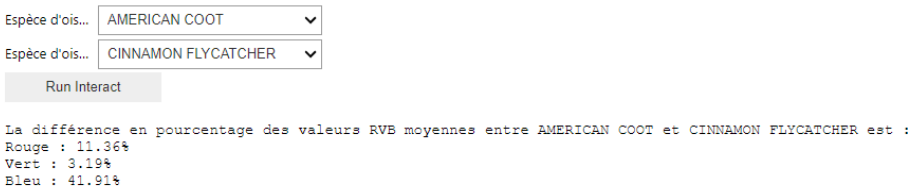
Il peut être intéressant de connaître le pourcentage de la répartition RVB pour une espèce d'oiseau sélectionnée. Aussi, nous avons créé cet outil permettant de sélectionner une espèce et d'obtenir la répartition des couleurs :



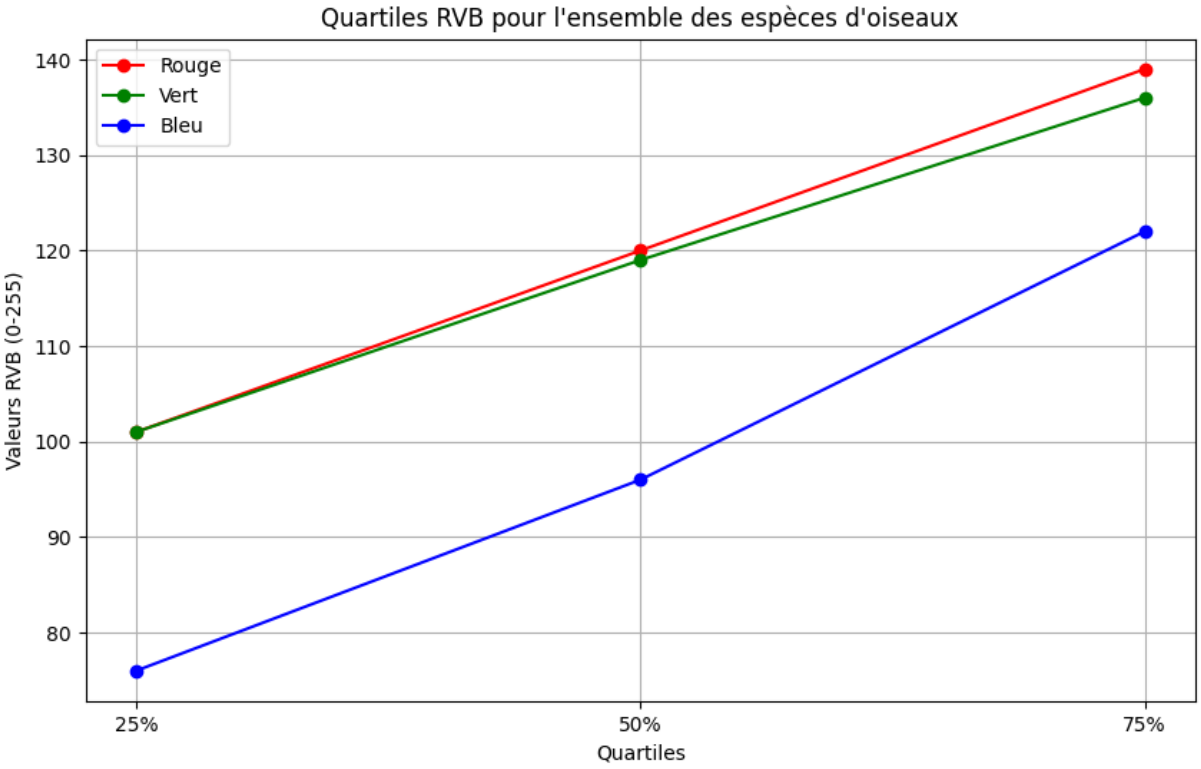

```
Pour l'espèce MCKAYS BUNTING, les pourcentages des couleurs RVB sont :
Rouge : 35.94%
Vert : 33.65%
Bleu : 30.41%
```

En se basant sur l'outil précédent, nous avons créé l'outil suivant permettant d'obtenir la différence couleur par couleur, entre deux espèces sélectionnées. Bien qu'au niveau statistique, de telles

informations ne soient pas pertinentes, cet outil pourrait nous permettre de comprendre pourquoi un modèle confond deux espèces. En effet, si la différence est très basse, il pourrait y avoir beaucoup d'erreurs entre deux classes. Ce genre de défaut serait mis en évidence par un tableau croisé entre les valeurs exactes et les valeurs prédites.



Enfin, nous analysons les images de différentes espèces d'oiseaux et calculons les valeurs totales de rouge, vert et bleu pour chaque espèce. Voici le graphique pour visualiser les quartiles de ces valeurs :



Cette analyse spectrale sommaire pourra trouver une utilité ultérieure si nous décidons de l'incorporer dans des modèles de machine learning (sur des histogrammes d'intensité). D'autre part, l'application de cette analyse sur des oiseaux détourées serait également plus judicieuse.

5. Conclusions des premières analyses descriptives

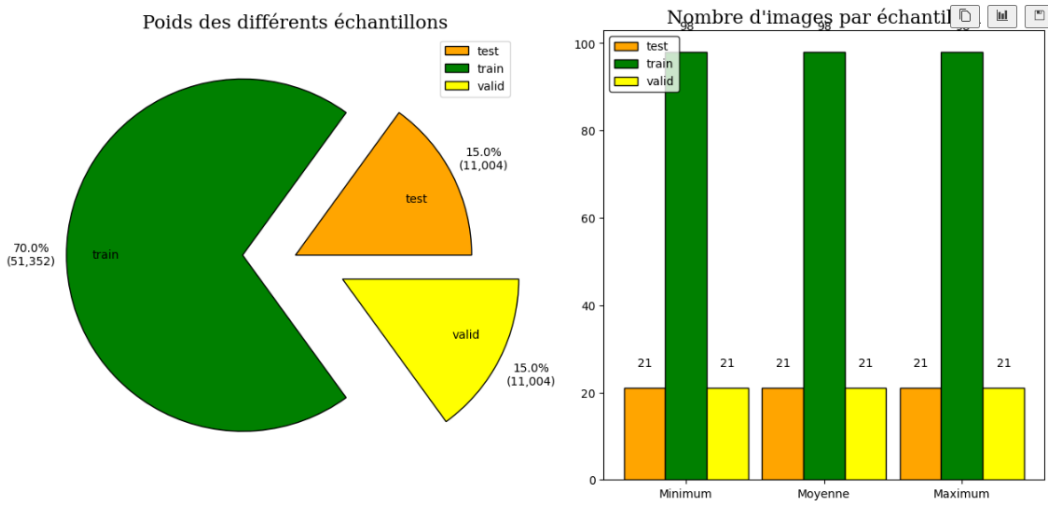
Plusieurs ajustements à réaliser ont été relevés par le contrôle de données : une classe doit être supprimée, une autre renommée et une image redimensionnée. De plus, l'analyse du poids de chaque échantillon dans le jeu de données nous a permis de constater que les jeux de test et de validation contiennent très peu d'images en comparaison du jeu d'entraînement. Enfin, les différents graphiques de répartition nous montrent que la distribution d'images par classe est hétérogène.

La première étape de pré-processing consiste à nettoyer l'intégralité de la base de données. Nous pensions ensuite rééquilibrer les sets afin de s'approcher d'un ratio 70-15-15 pour les sets d'entraînement, de test et de validation. Cependant, comme cité précédemment, la suite de l'analyse

nous a montré qu'un ré-échantillonnage était nécessaire. Plusieurs choix sont envisageables et le rééquilibrage des sets de données en dépend directement :

- Oversampling : c'est notre première idée. Dans le cas de la reconnaissance d'images, il est souvent préférable d'augmenter les données plutôt que de les réduire. Dans ce cas, nous utiliserons plusieurs manipulations d'images (rotation, translation, zoom, luminosité…) afin d'en dupliquer une partie. Il faut alors réaliser le rééquilibrage des sets avant d'effectuer l'opération pour ne pas risquer de trouver des images dupliquées dans les sets de test ou de validation. Nous avons développé une classe qui utilise la bibliothèque Pillow afin de réaliser cette opération. Ce n'est que plus tard, lors de la lecture du cours 151.2 que nous avons appris que Keras propose cette fonctionnalité d'une manière bien plus complète et optimisée.
- Undersampling : puisque nous travaillons dans un cas d'étude aux moyens limités, nous avons finalement décidé de réduire le nombre d'images afin d'optimiser le temps d'apprentissage de nos modèles. Dans ce cas, le rééquilibrage des sets de données devrait être réalisé après l'undersampling afin d'avoir une bonne répartition 70-15-15.
- Undersampling + Oversampling : une fois un modèle optimisé, nous envisageons de l'entraîner sur un set de données plus important. Pour cela, nous pouvons combiner l'oversampling à l'undersampling. Il faut alors réduire les 29 classes représentées par plus de 200 images avant d'augmenter les autres.

Les graphes ci-dessous donnent un aperçu du set de données après les corrections :



Concernant l'analyse RVB effectuée, nous envisageons de l'utiliser pour entraîner un modèle de machine learning basé sur la répartition et la quantité de couleur des images. Dans ce contexte, il est indispensable d'extraire les oiseaux en effaçant l'arrière-plan dans le but de ne conserver que les couleurs pertinentes. De plus, nous supposons que les modèles de deeplearning seraient également plus performants sur des images détourées. Cette hypothèse pourra être confirmée en comparant les modèles entraînés avec et sans arrière-plan.

III. Pré-processing

1. Nettoyer les données

Afin de démarrer le codage de notre projet dans un contexte CI/CD, les opérations de nettoyage ont été automatisées. Ainsi, l'intégration de nouvelles données provenant de sources diverses pourront facilement être traitées et intégrées à notre modèle.

La classe CleanDB a été développée dans le but de nettoyer notre set de données actuel ainsi que les données futures. Elle permet, par l'intermédiaire de différentes méthodes de classe, le redimensionnement d'image, la suppression de classe et l'équilibrage entre sets d'entraînement, de test et de validation.

La classe Oversampling permet de sur-échantillonner un set d'entraînement donné. Il ne sera cependant pas utilisé, la fonctionnalité existant déjà dans Keras.

La classe Undersampling permet de sous-échantillonner un set d'entraînement donné et de vérifier la répartition finale. Par défaut, toutes les classes sont ajustées sur celle la moins représentée. Elle donne cependant le choix de choisir un seuil sur lequel aligner les classes excédentaires.

2. Supprimer l'arrière-plan

Nous cherchons à entraîner un modèle capable de reconnaître des oiseaux, il est donc judicieux de ne pas garder le décor. En effet, cela pourrait rendre le modèle confus, puisqu'il n'y a pas de corrélation entre le décor et l'espèce d'oiseau.

En effectuant des recherches sur le web, notamment sur GitHub, j'ai découvert REMBG qui affirme pouvoir supprimer l'arrière-plan d'images. Le projet a été recommandé plus de 13.000 fois par les utilisateurs GitHub et des mises à jour récentes démontrent qu'il est encore maintenu. Il peut donc être considéré comme fiable.

Une fois toutes les étapes de configuration réalisées sur mon PC (CUDA, cuDNN et TensorRT), le programme s'est avéré fonctionner très rapidement (45 minutes pour traiter plus de 81 000 images de résolution 224x224 pixels). En plus de proposer une interface en ligne de commande, REMBG permet d'importer ses fonctionnalités dans notre propre code, ce qui m'a permis de bien l'intégrer à mon notebook et de pouvoir afficher une barre de progression. Le modèle u2net s'avère déjà très performant, avec seulement 2% de mauvais détourages sur 2500 images. J'ai également testé isnet-general-use et sam, deux autres modèles proposés par REMBG, mais u2net reste meilleur pour cette tâche.

J'ai constaté que les 2% d'images mal détourées sont souvent très vides, sans oiseau ni décor. J'ai donc essayé de détecter la présence d'un oiseau sur une image traitée afin de déterminer l'efficacité du détourage.

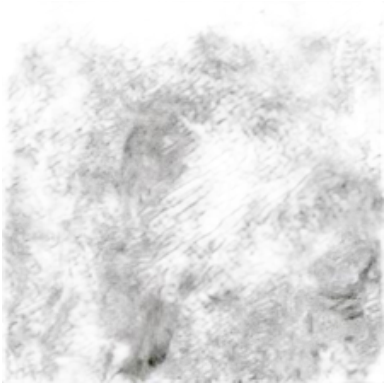
Pour tester si un oiseau a bien été isolé du décor, de manière automatisée, j'ai d'abord suivi la piste de Tensorflow Lite avec le projet d'Object Recognition, qui tourne très rapidement sur le CPU et que j'avais déjà utilisé lors de précédents projets. J'ai testé les différentes versions du modèle Efficientnet_lite ainsi que MobileNet V1. Sachant que les labels n'indiquent pas "oiseau" mais plutôt les noms de différentes espèces, j'ai extrait une liste contenant uniquement les classes qui sont liées aux oiseaux :

```
BIRD_LABELS = [
"cock", "hen", "ostrich", "brambling", "goldfinch", "house finch",
"junco", "indigo bunting", "robin", "bulbul", "jay", "magpie",
"chickadee","water ouzel", "kite", "bald eagle", "vulture", "great
grey owl", "black grouse", "ptarmigan", "ruffed grouse", "prairie
chicken", "peacock", "quail", "partridge", "African grey", "macaw",
"sulphur-crested cockatoo", "lorikeet","coucal", "bee eater",
"hornbill", "hummingbird", "jacamar", "toucan","drake","red-breasted
merganser", "goose", "black swan", "white stork", "black
stork","spoonbill", "flamingo", "little blue heron", "American egret",
"bittern","crane", "limpkin", "European gallinule", "American coot",
"bustard","ruddy turnstone", "red-backed sandpiper", "redshank",
"dowitcher","oystercatcher", "pelican", "king penguin", "albatross"
]
```

J'ai ensuite configuré le modèle pour simplement indiquer si une de ces classes était détectée, et donc confirmer ou non la présence d'oiseaux. L'idée était intéressante mais malheureusement, le modèle n'a pas dépassé les performances d'un modèle aléatoire. En effet, comme évoqué précédemment, seulement 2% des images traitées sont considérées comme mauvaises, mais le modèle Tensorflow en a compté plus de 50%. Malgré différents réglages de seuil de détection, je n'ai pas pu aboutir à un résultat satisfaisant.

La méthode suivante, bien plus simple, consiste à calculer la proportion de pixels transparents. S'il y a plus de 80% de pixels transparents, cela signifie que l'oiseau a été mal détouré.

Mauvais détourage



Mauvais détourage

Bon détourage

Pourtant, si l'on regarde de plus près sur un fond blanc (contrairement au fond noir de Windows File Explorer), on se rend compte que ce n'est pas de la transparence mais plutôt de la "saleté" ou du bruit. J'ai donc pensé à une méthode très simple mais efficace : calculer la luminosité moyenne de l'image. Pour cela, je la convertis en niveaux de gris, puis en array numpy. Enfin, je calcule la moyenne des valeurs des pixels. Les résultats obtenus sont très satisfaisants, avec une bonne détection des mauvaises images.

Le programme actuel a donc cette structure :

SUPPRIMER ARRIÈRE PLAN

▼

CONVERTIR EN NOIR ET BLANC

▼

↙ **CALCULER LUMINOSITÉ MOYENNE** ↘

Luminosité < 12

▼

On sauvegarde l'image originale qui sera traitée par un autre algo

Luminosité > 12

▼

On sauvegarde l'image détourée

Exemples de détourages réussis

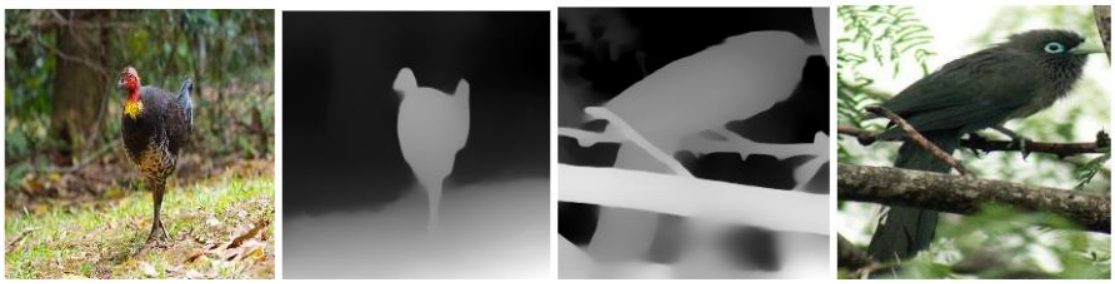
Les bonnes images sont enregistrées dans un nouveau dossier, avec la même structure que le dataset original (on ne traite que les images du set de *train*, car les sets *test* et *valid* doivent contenir un arrière-plan).

Il faut ensuite s'occuper des images mal détourées. Quand la luminosité n'est pas satisfaisante, on conserve l'image originale, dans l'attente d'un meilleur traitement. Les noms de fichiers n'étant pas uniques (1.jpg, 5.jpg, 111.jpg… dans chaque dossier), j'ai concaténé le nom de la classe et le nom du fichier (par exemple "ALBATROSS_131.jpg").

Afin de détourer les 2% d'images restantes, on utilise un algorithme différent : Depth-Anything. Ce projet GitHub permet de créer ce que l'on appelle une depth map (carte de profondeur) :

Cartes de profondeur

Une carte comme celle-ci est très facile à lire : plus les pixels sont blancs, plus l'élément est proche de la caméra ; plus les pixels sont noirs, plus l'élément est éloigné. L'idée est de créer un masque : tous les pixels dont la valeur dépasse un certain seuil (par exemple 128) sont conservés sur l'image d'origine. Les autres sont rendus transparents. L'objectif est de garder uniquement les oiseaux, qui sont souvent au premier plan, comme sur les images ci-dessus. Il faut néanmoins prendre en considération que l'oiseau n'est pas forcément le seul élément du premier plan, voire n'est pas au premier plan du tout, comme on peut le voir sur les images ci-dessous.



Cartes de profondeur

Ici, on voit sur les images qu'il y a d'autres éléments proches de la caméra. En choisissant de garder ce qui est proche, il restera forcément une partie du décor. Il est possible d'ajuster le seuil, que j'ai réglé à 100 (sur 256), ce qui signifie que l'on supprime surtout les parties les plus sombres et donc éloignées. Nous pouvons d'ailleurs noter que ce programme est très impressionnant dans sa capacité à créer des depth maps très détaillées, même dans des scènes complexes. Bien que le décor n'ait pas complètement disparu, ce résultat est préférable à l'image originale.

Via une simple commande, le script crée des depth maps pour toutes les images qui ont été mises de côté dans la partie précédente et les stocke dans un autre dossier, avec cette-fois ci la structure : "ALBATROSS_131_depth.png".

Depth-Anything utilise pytorch et torchvision et peut travailler sur le GPU, pour un temps d'inférence optimisé.

Maintenant que nous avons les images et leur depth map, il faut supprimer l'arrière-plan, tâche qui est confiée au dernier script. En utilisant la carte de profondeur, il va créer un masque. Pour tous les pixels inférieurs dont le seuil de luminance est inférieur à 100, les pixels du masque seront nuls ("False"). Par la suite, il suffit d'appliquer ce masque à notre image, en rendant transparents tous les pixels "False".

Le résultat

Sachant que ces images ne représentent que 1 à 2% du dataset *train*, je considère que le détourage est satisfaisant. À chaque fois qu'une image est traitée, on la stocke dans le bon dossier avec le bon nom, et on supprime les dossiers temporaires. Une fois toutes ces procédures réalisées, toutes les images du set d'entraînement ont été détourées de manière automatique.

Nous essayerons d'entraîner le modèle à la fois avec ou sans arrière-plan et nous comparerons les résultats.

IV. Nettoyage et formatage des données

1. Mise en production des scripts de formatage du dataset

Pendant cette deuxième phase de projet, la classe CleanDB a largement été améliorée et débuguée. En effet, elle n'était apparemment pas adaptée à la configuration de nos différents postes de travail. De plus, elle restait assez dépendante de paramètres passés ou du nommage de fichiers. Une faute de frappe ou d'orthographe pouvait alors faire planter le script. Enfin, un paramètre a été ajouté pour permettre de contrôler le caractère aléatoire du split *train*/*test*/*valid*.

L'ensemble de ces bugs ont été corrigés. La suite logique serait d'embarquer le script dans un conteneur docker afin de contrôler complètement la chaîne d'exécution.

En outre, les fonctions de sous-échantillonnage ont également été débuguées et légèrement améliorées pour offrir plus d'options.

Dans l'ensemble, tout le code du projet a été relu, commenté plus précisément, nettoyé et optimisé.

2. Changer la distribution des données

Introduction

Comme nous le verrons plus loin, les premiers modèles n'étaient pas prometteurs. On pouvait constater un sur-apprentissage évident et supposer un nombre de classes trop important. Enfin, la quantité de données disponibles pour chaque classe semblait trop faible.

Revoir le sous-échantillonnage

La première solution testée a été de supprimer certaines classes (celles pour lesquelles nous avions le moins d'images). En imposant un nombre minimum de 160 images par classe, nous obtenons 395 classes, chacune mieux représentée que lors des premiers essais.

La classification des oiseaux

Définir la classification

La deuxième option explorée a consisté à rassembler les espèces d'oiseaux par groupes. De cette manière, on peut à la fois réduire le nombre de classes et augmenter la quantité de données pour chaque classe.

La manière la plus logique d'appliquer cette méthode est de s'intéresser à la classification des oiseaux. Les oiseaux font partie du règne *Animalia*. Ils sont d'abord répartis en ordres, puis en familles au sein de ces ordres et enfin en espèces. Le plus connu de ces ordres est l'ordre des Passereaux qui regroupe la plupart des oiseaux de nos jardins.

Le nombre d'ordres semble diverger en fonction des sources mais on en compte une trentaine environ, ce qui équivaut au même nombre de classes, ce qui est bien plus raisonnable que les 523 initiales.

Récupérer les données

La première étape pour pouvoir entraîner un modèle sur les données réparties de cette manière est de trouver ou de créer une table de correspondance exhaustive entre les 523 espèces du dataset et les ordres. Nous pensions pouvoir la trouver sous forme de fichier excel ou csv, sur un site spécialisé, ou encore via l'API de l'un de ces sites (les plus connus étant eBird et Avibase).

Après des recherches infructueuses sur plusieurs sites d'ornithologie, nous avons essayé de demander à plusieurs modèles LLM de les classer. Le résultat n'était pas du tout satisfaisant, nous avons alors envisagé d'écrire un script de web scrapping pour récupérer les données.

Le site Avibase propose une base de données qui semble très bien renseignée. Après quelques requêtes effectuées sur cette base avec différentes espèces de notre dataset, un nouveau problème nous est apparu : la base est pleine de fautes d'orthographe ou de typographie. De plus, certains noms de d'espèces du dataset sont en fait des familles. Il arrive également qu'ils soient incomplets. Écrire et débuguer un script pour cette tâche, alors que nous ne sommes pas experts en scrapping, serait sans doute plus long que de récupérer les données à la main.

Nous avons donc récupéré le nom réel de chaque espèce, l'ordre et la famille pour les 523 classes du dataset.

Bien que le travail soit fastidieux, cela nous a permis de découvrir l'existence de la classe Loony Bird qui ne contient que des photos de personnalités politiques. Nous l'avions ratée lors de l'exploration des données.

Écrire le script

Une fois la table de correspondance complétée, écrire le script pour réorganiser les photos par ordres ou par familles a été rapide. Il a cependant fallu ajouter à chaque image le nom de son espèce afin de ne pas perdre d'information. En effet, le nom des fichiers est par défaut un nombre qui a été donné pendant l'exécution du script cleanDB. Plusieurs fichiers d'espèces différentes ont donc le même nom.

V. Modèle From scratch

1. Première approche à travers les réseaux denses

Premier modèle

Le premier modèle est construit avec une architecture simple qui comprend une couche d'entrée, deux couches denses intermédiaires avec des unités de Dropout pour la régularisation, et une couche de sortie avec une activation softmax pour la prédiction de plusieurs classes.

Des générateurs d'images sont utilisés pour charger et prétraiter les images à partir des dossiers *train*, *test* et *valid*. Les callbacks *ReduceLROnPlateau* et *EarlyStopping* ont été utilisés pour optimiser le processus d'apprentissage en ajustant le taux d'apprentissage et en arrêtant l'entraînement lorsque la perte de validation ne s'améliore plus.

Les modèles sont testés sur 10 classes pour des raisons de temps de calcul. Si un modèle donne de bons résultats, il sera entraîné sur un nombre plus important de classes.

Le premier modèle a atteint une précision de 0.43 sur le set de test et une précision de validation de 0.45. Ces valeurs sont moyennes, ce modèle a du mal à généraliser à partir des données d'entraînement. De plus, nous voyons avec le *EarlyStopping* qu'il a atteint ses limites dans ces conditions d'entraînement.

```
...
Epoch 11: early stopping
4/4 [==============================] - 0s 29ms/step - loss: 1.5554 - acc: 0.4400 - mean_absolute_error: 0.1395
[1.5553895235061646, 0.4399999976158142, 0.1394689679145813]
0.4583333432674408
```

Cela peut être dû à plusieurs raisons, c'est pourquoi nous avons décidé d'explorer d'autres architectures de modèles, d'ajuster les hyperparamètres pour améliorer les performances et d'augmenter les données pour améliorer la précision du modèle.

Deuxième modèle

Dans le deuxième modèle, la taille du lot (*batch_size*) a été augmentée de 16 à 20. Cela signifie que le modèle est mis à jour plus fréquemment lors de chaque époque, ce qui peut accélérer la convergence de l'apprentissage, mais aussi rendre l'apprentissage plus instable.

L'optimiseur est modifié, c'est *Adam* qui est utilisé avec un taux d'apprentissage de 0.01. Dans le premier modèle, *Adam* avait été utilisé avec son taux d'apprentissage par défaut (0.001). Un taux d'apprentissage plus élevé peut accélérer l'apprentissage, mais il peut aussi empêcher la convergence vers le minimum global. Nous avons choisi cinq epochs, ce qui a été suffisant pour voir le mauvais score de ce modèle, qui a été interrompu par le *EarlyStopping* au bout de quatre epochs.

```
Epoch 4: early stopping
3/3 [==============================] - 0s 133ms/step - loss: 1.9489 - acc: 0.2600 - mean_absolute_error: 0.1627
[1.948917031288147, 0.25999999046325684, 0.1627383530139923]
0.25
```

Le résultat est beaucoup moins bon que pour le premier modèle. Il faut donc explorer d'autres pistes.

Troisième modèle

- Augmentation des données : *ImageDataGenerator* est utilisé avec des paramètres d'augmentation des données pour l'ensemble d'entraînement. Cela inclut la rotation, le décalage en largeur et en hauteur et le retournement horizontal des images.
- Taux de dropout : le taux de dropout est augmenté de 0.2 à 0.5.

Le score final est meilleur que le précédent mais moins bon que pour le premier modèle.

```
3/3 [==============================] - 0s 28ms/step - loss: 1.9083 - acc: 0.3000 - mean_absolute_error: 0.1602
[1.9082660675048828, 0.29999998211860657, 0.16015444695949554]
0.32500001788139343
```

Quatrième modèle

- Modification de l'optimiseur *Adam* pour *SGD* (descente de gradient stochastique).
- Ajout d'une régularisation L2 aux couches denses.
- Ajout des couches de normalisation par lots après les couches denses.
- Augmentation du nombre de neurones dans les couches denses de 1024 à 2048 et de 512 à 1024.

Les résultats restent trop bas.

```
Epoch 5/5
80/80 [==============================] - 17s 210ms/step - loss: 11.5105 - acc: 0.2441 - mean_absolute_error: 0.1529 - val_loss: 8.5280 - val_acc: 0.3500 - val_mean_abs
3/3 [==============================] - 0s 28ms/step - loss: 8.4772 - acc: 0.2400 - mean_absolute_error: 0.1479
[8.477151870727539, 0.23999999463558197, 0.14787811040878296]
0.3499999940395355
```

Cinquième modèle

- Modification de la fonction d'activation *ReLU* par *LeakyReLU*. Contrairement à *ReLU*, *LeakyReLU* permet aux neurones de passer un petit gradient négatif lorsqu'ils sont inactifs.
- Ajout de l'initialisation de *He* pour les poids des couches denses.
- Ajout d'un taux de déclin à l'optimiseur *Adam*.
- Augmentation du nombre d'epochs à 10.

Le résultat obtenu est meilleur, sans être satisfaisant.

```
Epoch 10/10
80/80 [==============================] - 18s 219ms/step - loss: 36.0050 - acc: 0.2815 - mean_absolute_error: 0.1433 - val_loss: 14.7080 - val_acc: 0.4000 - val_mean_ab
3/3 [==============================] - 0s 56ms/step - loss: 9.1759 - acc: 0.3600 - mean_absolute_error: 0.1214
[9.17594051361084, 0.35999998450279236, 0.12135462462902069]
0.4000000059604645
```

Après plusieurs essais, les réseaux de neurones denses ne semblent pas répondre aux attentes du projet. Nous décidons de nous tourner vers les CNN.

2. Réseaux de neurones convolutifs

Sixième modèle

- Modification de l'architecture du modèle pour utiliser un réseau de neurones convolutifs (CNN) avec une construction fonctionnelle.
- Ajout de deux couches convolutives avec une activation *ReLU* et des couches de *pooling_max* après chaque couche convolutive.
- Ajout d'une couche *Flatten.*
- Ajout d'une couche Dense avec une activation *ReLU.*

Le score est étonnement bas. En effet, les CNN ont été conçus spécifiquement pour le traitement d'images. Ils sont vus comme les meilleurs extracteurs de features pour la classification d'images.

```
Epoch 4: early stopping
3/3 [==============================] - 2s 1s/step - loss: 2.3095 - acc: 0.1000 - mean_absolute_error: 0.1800
[2.309499979019165, 0.09999999403953552, 0.18000002205371857]
0.10000000149011612
```

Septième modèle

- Construction du CNN séquentiel.
- Ajout de plusieurs couches convolutives avec une activation *ReLU* et des couches de normalisation par lots après chaque couche convolutive.
- Ajout des couches de *pooling_max* après chaque bloc de couches convolutives.

Les résultats, bien meilleurs que précédemment, montrent que nous sommes sur la bonne piste.

```
Epoch 10/10
80/80 [==============================] - 143s 2s/step - loss: 1.1196 - acc: 0.6154 - mean_absolute_error: 0.1018 - val_loss: 0.7068 - val_acc: 0.8250 - val_mean_absolu
3/3 [==============================] - 6s 3s/step - loss: 0.6718 - acc: 0.8000 - mean_absolute_error: 0.0760
[0.6718435287475586, 0.7999999523162842, 0.07595556974411011]
0.824999988079071
```

Nous n'obtiendrons pas de meilleurs résultats lors des entraînements suivants. Cependant, voici quelques pistes que nous avons suivies. Cette liste n'est pas exhaustive : l'ensemble des essais réalisés se trouve dans le fichier dédié sur le dépôt github :

« *reco_oiseau_jan24bds/notebooks/model_deep_02_from_scratch.ipynb* »

Huitième modèle

- Augmentation du nombre d'epochs de 10 à 40. Cela donne plus de temps au modèle pour apprendre.

L'entraînement a été arrêté à la onzième epoch avec un résultat moins bon. Cela pourrait éventuellement montrer que le *EarlyStopping* n'est pas assez patient.

```
Epoch 11: ReduceLROnPlateau reducing learning rate to 1.0000000474974514e-05.
...
Epoch 11: early stopping
3/3 [==============================] - 1s 141ms/step - loss: 0.7737 - acc: 0.7200 - mean_absolute_error: 0.0820
[0.7736566066741943, 0.7199999690055847, 0.08202610909938812]
0.7250000238418579
```

Neuvième modèle

- Réduction de la taille du lot de 20 à 16.

Le résultat obtenu est meilleur que le précédent.

```
Epoch 10: ReduceLROnPlateau reducing learning rate to 0.00010000000474974513.
100/100 [==============================] - 108s 1s/step - loss: 1.1570 - acc: 0.5962 - mean_absolute_error: 0.1034 - val_loss: 1
...
Epoch 14: early stopping
4/4 [==============================] - 0s 97ms/step - loss: 0.4736 - acc: 0.9000 - mean_absolute_error: 0.0613
[0.47355496883392334, 0.8999999761581421, 0.06125110015273094]
0.7916666865348816
```

Dixième modèle

- Suppression de l'augmentation des données pour l'ensemble d'entraînement. Le modèle s'est donc entraîné sur les images originales sans modifications. L'idée est d'observer l'influence de l'augmentation de données.

```
Epoch 10: early stopping
4/4 [==============================] - 0s 57ms/step - loss: 0.8334 - acc: 0.7400 - mean_absolute_error: 0.0603
[0.8334342837333679, 0.7400000095367432, 0.060277972370386124]
0.7291666865348816
```

Le modèle est moins bon, ce qui montre l'efficacité de l'augmentation de données.

VI. Modèles de machine learning

1. Préambule

En parallèle de la réalisation des modèles de deep learning, nous avons décidé de tester des modèles de machine learning afin d'obtenir une première base de comparaison mais également d'analyser les capacités et les limites de ces modèles sur ce type de problématique.

L'ensemble des analyses ont été réalisées à partir des jeux de données générés par la classe CleanDB.

Comme vu précédemment, nous disposons de deux bases d'images :

- La base originale
- La base détourée

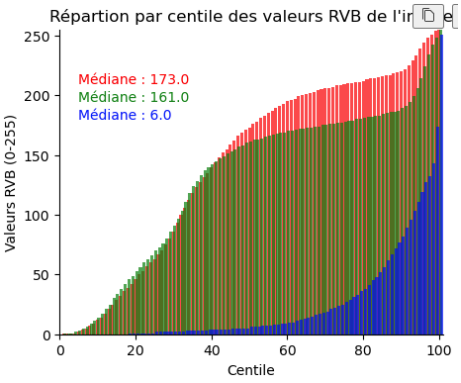
Pour cette partie, chaque base se compose de 394 espèces comportant chacune 160 images (112 d'entraînement, 24 de test et 24 de validation) soit un total de 63 040 images par base.

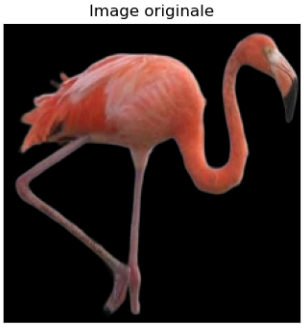
2. Modèles sur intensité des couleurs RVB

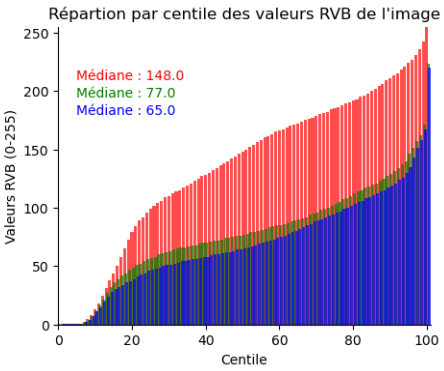
Création de la base

La première tâche consiste donc à créer un jeu de données exploitable par les modèles de machine learning. Le jeu de données utilisé pour cette modélisation a été obtenu à partir de la répartition en centiles de l'intensité des couleurs pour chaque image. Nous avons créé une fonction permettant d'obtenir la répartition en centiles des valeurs RVB d'une image.

Exemples sur images détourées (détourage exclu des métriques) :







Nous avons créé une autre fonction permettant de balayer l'ensemble des dossiers et lancer la fonction précédente. Le temps de traitement est d'environ 170 images/secondes.

Nous obtenons ainsi deux jeux de données (originales et détourées) composées chacun de 63 040 lignes et 315 colonnes :

- Centiles pour couleur Rouge (101), Verte (101) et Bleue (101)
- Moyenne et écart-type pour chaque couleur (6)
- Trois variables calculées supplémentaires (écart entre les moyennes des couleurs)
- Source des données (échantillon, classe et nom de l'image)

Pré-processing et réduction de dimension

Une fois les jeux de données créés, nous avons mis en place une fonction permettant de préparer les datasets pour le machine learning :

- Séparation des variables explicatives de la cible (= classe)
- Séparation des données (*train*, *test* et *valid*)
- Normalisation des données à partir du jeu d'entraînement

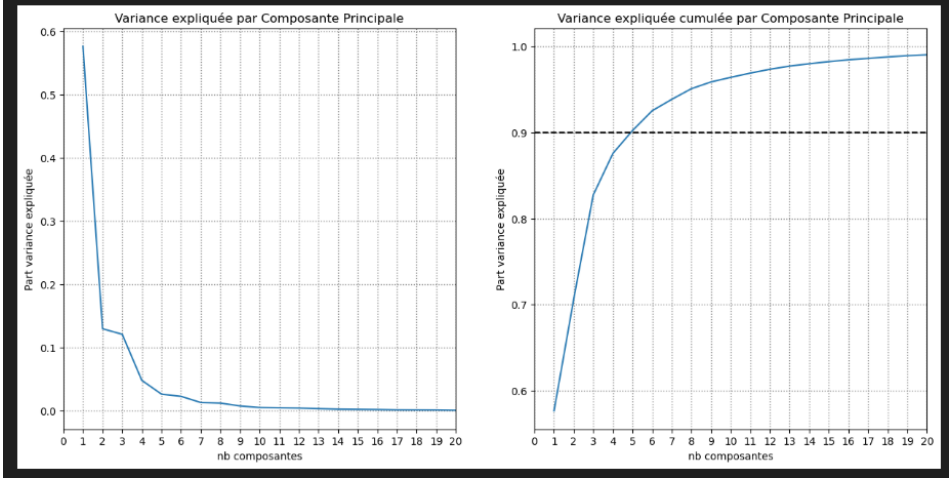
Le nombre de variables en entrée (312) est élevé et le risque de corrélation fort. Une réduction de dimension s'avère nécessaire. Nous avons créé une classe initialisée à partir du jeu de données d'entraînement et des jeux de données à transformer. Elle possède trois méthodes :

- Suppression des variables de variance nulle
- Affichage des résultats de l'ACP (part de la variance expliquée par composante principale)
- Exécution de l'ACP (entraînement et transformation avec nombre de classes choisi)

Exemple de traitement du jeu de données d'images détourées :

```
Nombre d'images par échantillon :
 echantillon
train    44128
test      9456
valid     9456
Name: count, dtype: int64

Nombre de variable avec variance nulle :  0
```




```
Part de la variance expliquée conservée pour 10 composantes : 96.40%
```

On constate qu'en ne prenant que les 10 premières composantes, nous conservons plus de 96% de la variance. Cela confirme donc tout l'intérêt de la réduction de dimension.

Pour finir, toutes ces fonctions et classes ont permis de mettre en place une pipeline.

Modélisation et tests

Pour la partie modélisation nous avons décidé de tester quatre modèles :

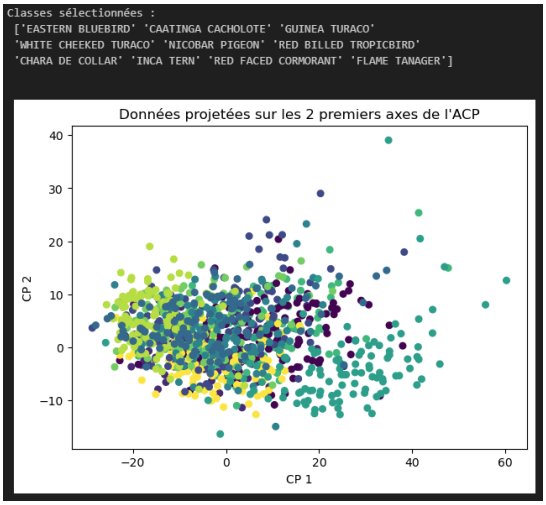
- Régression logistique
- Bagging
- Random Forest Classifier
- XGBoost Classifier

Nous avons créé une classe Modèle initialisée à partir des différents jeux de données (X_train, X_test, X_valid, y_train, y_test et y_valid) obtenus suite aux étapes de pré-processing et de réduction de dimension. Cette classe possède cinq méthodes : une méthode par modèle et une globale permettant de lancer les quatre premières à la suite. Chaque modèle possède sa propre grille de recherche des meilleurs hyperparamètres.

Les modèles sont entraînés sur le jeu d'entraînement et les hyperparamètres sont validés sur le jeu de test. Une fois le meilleur modèle défini, ce dernier est réentraîné sur le jeu d'entraînement ainsi que le jeu de test. Les résultats du modèle sont ensuite évalués à partir du jeu de validation.

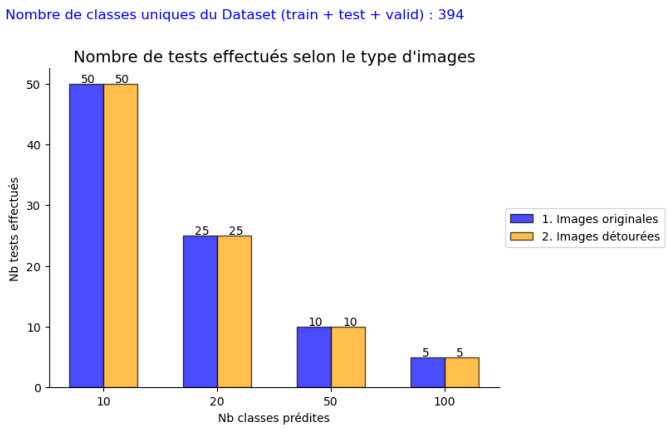
Pour la mise en place des tests, nous avons implémenté une fonction permettant la création de jeux de données à partir d'un nombre de classes choisi aléatoirement ou selon un noyau (seed) défini. Cela permet d'évaluer les différents modèles selon 10, 20, 50 ou 100 classes à prédire.

Exemple pour 10 classes :



Évaluation des modèles

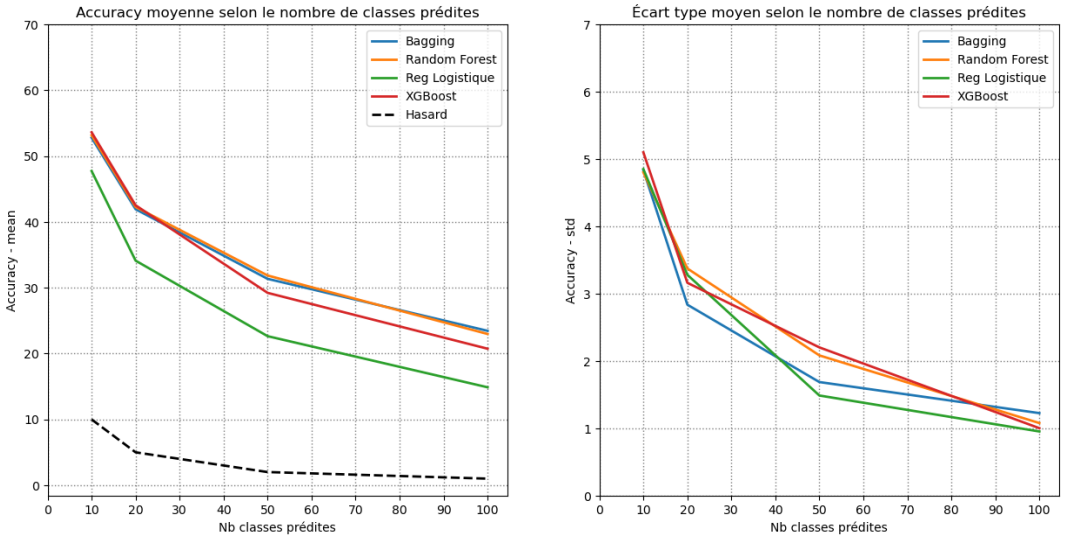
Nous avons réalisé 90 évaluations par jeu de données (images originales et détourées).



Par exemple, 50 tests ont été réalisés pour 10 classes à prédire. Ces classes ont été choisies aléatoirement parmi les 394 présentes dans le jeu de données. Afin d'obtenir une comparaison fiable entre la base d'images originales et détourées, les classes à prédire sont les mêmes entre ces deux jeux de données.

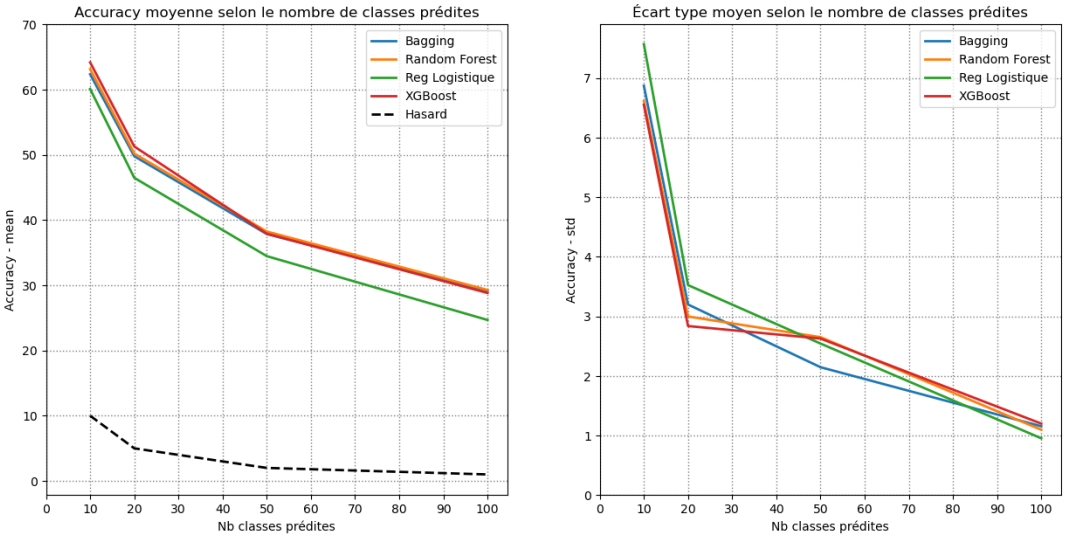
Résultats obtenus :

Résultats tests sur image : 1. originale



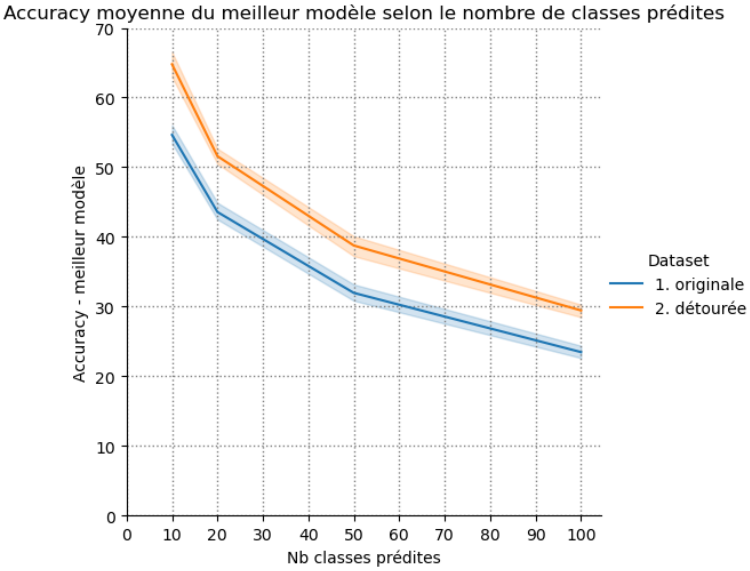
Sur les images originales (non détourées), nous arrivons à un peu moins de 55% d'accuracy de moyenne pour 10 classes à prédire. Plus le nombre de classes à prédire augmente, plus le taux d'accuracy moyen obtenu diminue. On note que sur cette source de données, le modèle de régression logistique est beaucoup moins performant que les autres.

Résultats tests sur image : 2. détourée



Sur les images détourées, nous arrivons à un peu moins de 65% d'accuracy de moyenne pour 10 classes à prédire. Comme pour les images originales, plus le nombre de classes à prédire augmente, plus le taux d'accuracy moyen obtenu diminue. On note que sur cette source de données, le modèle XGBoost obtient de meilleurs résultats.

Comparaison des résultats obtenus selon la source de données :



Comme on pouvait s'y attendre, les résultats obtenus à partir de la base d'images détourées sont bien meilleurs (+10 pts d'accuracy pour 10 classes à prédire par rapport à la base originale). En effet, on se base sur la distribution des couleurs pour effectuer la prédiction. Les images non-détourées présentent, en plus de l'oiseau, un décor qui biaise la répartition des couleurs.

3. Modèles sur palette de couleurs

Suite aux premiers résultats obtenus et l'apprentissage de la Computer Vision, nous avons voulu tester si des données encore plus pertinentes pouvaient être extraites des images.

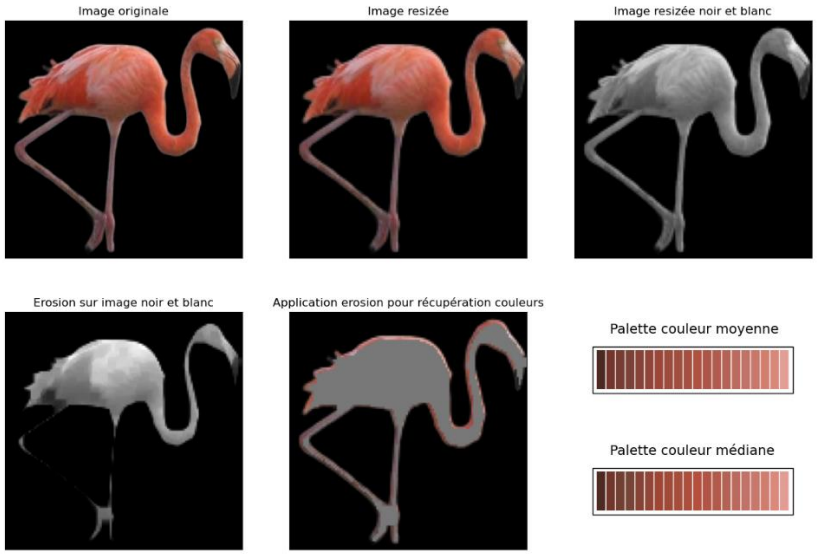
Dans la précédente modélisation, chaque couleur était traitée indépendamment. L'idée était donc pour cette seconde modélisation d'essayer de conserver la combinaison de ces trois intensités.

Création de la base

Une nouvelle fonction permet l'extraction des couleurs principales de l'image (de la plus foncée à la plus claire). L'image est ainsi découpée en x couleurs de poids égaux.

Étapes d'extraction pour une image détourée :

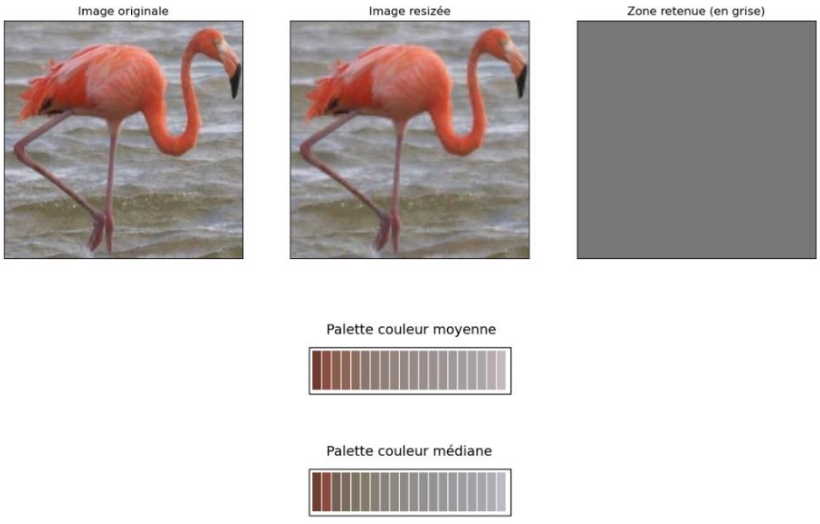
Lecture image (224, 224, 3) >> Redimensionnement (120, 120, 3) >> Passage en noir et blanc >> Définition de l'érosion >> Application de l'érosion >> Découpage en x couleurs selon les données extraites (ici 20)



Pour les images originales, le traitement est plus léger :

Lecture image (224, 224, 3) >> Redimensionnement (120, 120, 3) >> Exclusion des pixels totalement blancs ou noirs (car certaines images originales n'ont pas d'arrière-plan) >> Découpage en x couleurs selon données extraites (ici 20)

NB : On constate que pour l'exemple donné, l'ensemble de l'image (arrière-plan compris) est pris en compte pour le calcul de la palette (cf. zone retenue).



Pour chaque couleur extraite, différents indicateurs sont calculés à partir des pixels qui la composent :

- Les indicateurs de position des couleurs R, V, B + gris (moyenne, médiane, Q1, Q3)
- Les indicateurs de dispersion des couleurs R, V, B + gris (écart-type et intervalle interquartile)

La fonction permettant de balayer l'ensemble des dossiers a ensuite été reprise. Pour la constitution de la base, le paramètre a été fixé à 20 couleurs à extraire par image. Le temps de traitement est forcément plus long que pour la première modélisation (environ 6 images/secondes soit 27 fois plus long que pour le premier jeu de données).

Nous obtenons ainsi deux jeux de données (originales et détourées) composés chacun de 63 040 lignes et 483 colonnes :

- Indicateurs de position et dispersion (6) pour chaque couleur extraite (20) par RVB et gris (4) soit 480 colonnes (6*20*4)
- Source des données (échantillon, classe et nom de l'image)

Pré-processing et réduction de dimension

Pour le pré-processing et la réduction de dimension, les fonctions et classes déjà définies précédemment ont également été reprises. Le nombre de composantes conservées est plus important que pour la première modélisation :

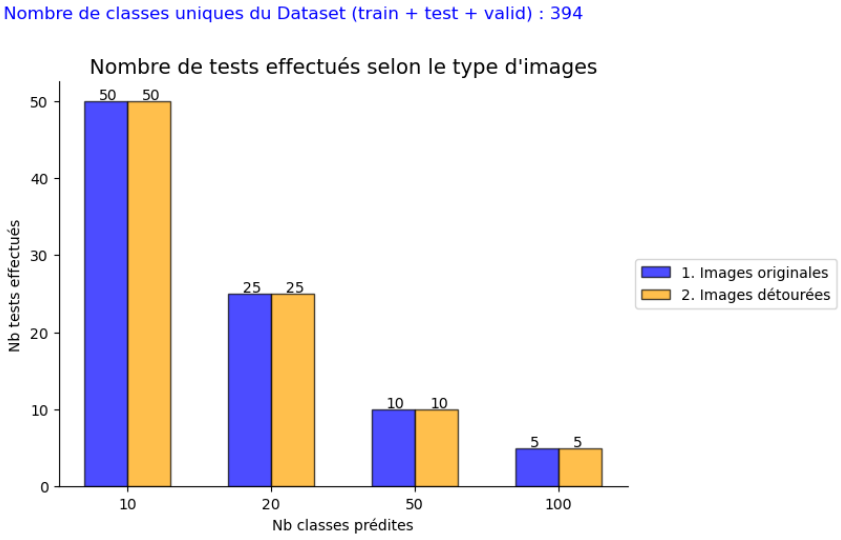
- 20 vs 11 pour la base d'images originales
- 24 vs 10 pour la base d'images détourées

Modélisation et tests

Cette partie reprend exactement les mêmes concepts que pour la première modélisation. Il a été possible à nouveau de reprendre l'ensemble des fonctions et classes déjà définies.

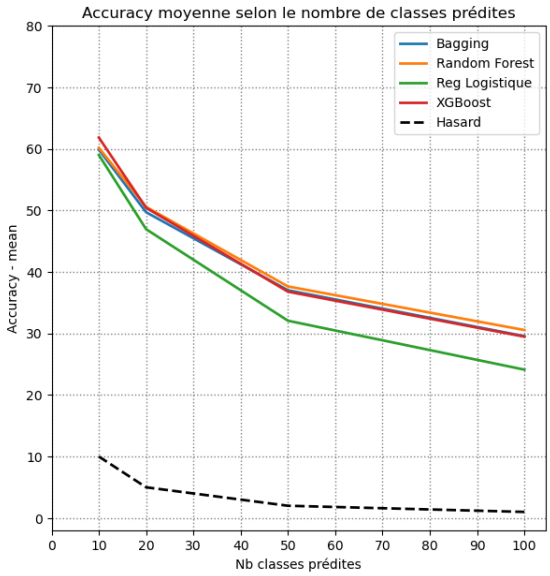
Évaluation des modèles

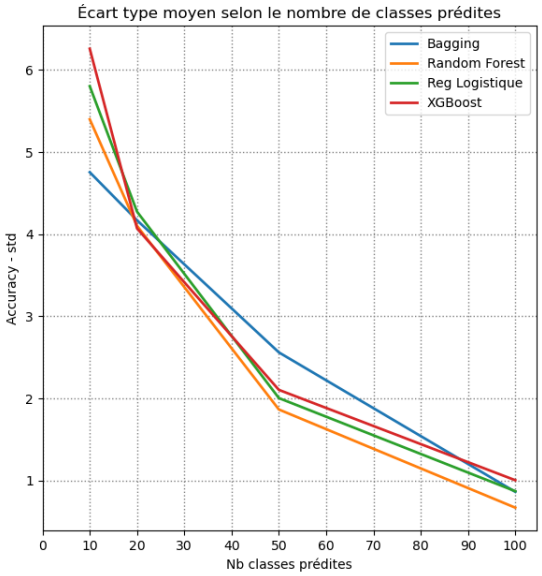
Comme pour la première modélisation, 90 évaluations par jeu de données (images originales et détourées) ont été réalisées.



Résultats obtenus :

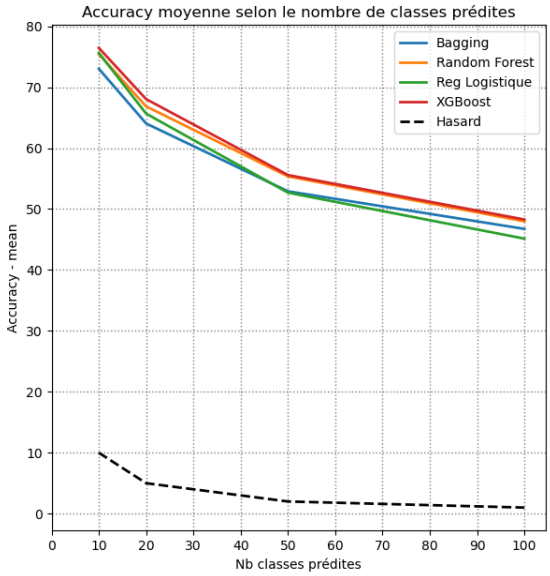
Résultats tests sur image : 1. originale

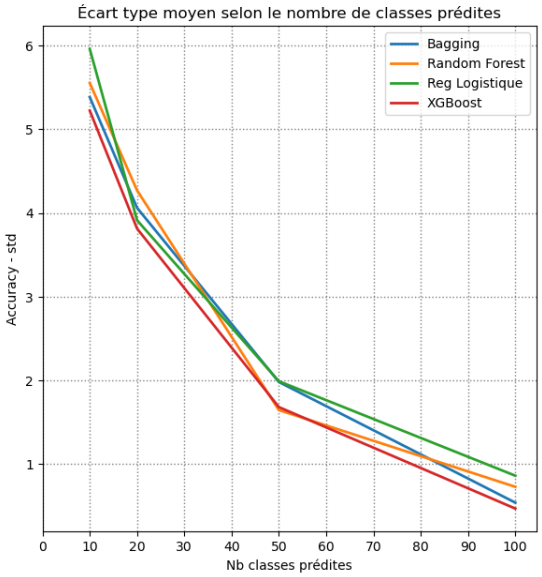




Sur les images originales (non détourées), nous arrivons à environ 62% d'accuracy de moyenne pour 10 classes à prédire soit +7 pts par rapport à la première modélisation. Le meilleur modèle pour 10 classes à prédire est le XGBoost mais c'est également celui avec la plus grande variation. On note également que, sur ce jeu de données, le Random Forest Classifier se comporte mieux que les autres modèles pour un nombre élevé de classes à prédire.

Résultats tests sur image : 2. détourée

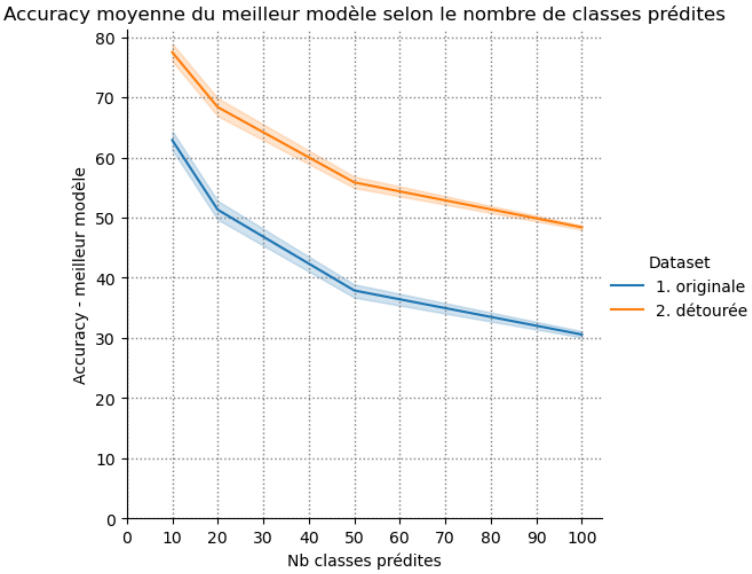




Sur les images détourées, nous arrivons à un peu plus de 76% d'accuracy de moyenne pour 10 classes à prédire (soit +11 pts par rapport à la première modélisation) et presque 50% d'accuracy moyenne pour 100 classes à prédire (soit +20 pts par rapport à la première modélisation).

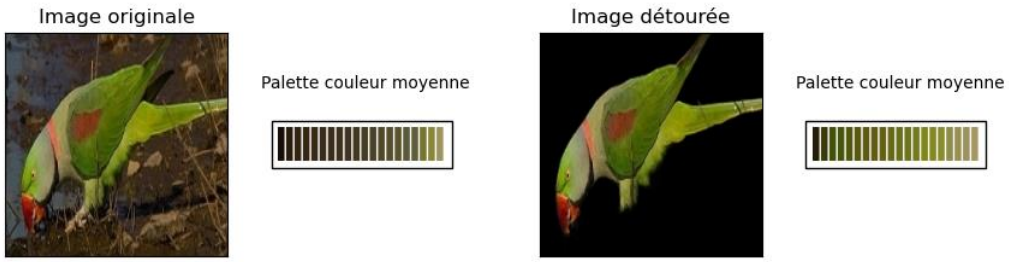
Le meilleur modèle, quel que soit le nombre de classes à prédire, est le XGBoost (meilleure accuracy moyenne et variation la plus faible).

Comparaison des résultats obtenus selon la source de données :



L'écart entre les bases d'images originales et détourées s'est encore accentué par rapport à la première modélisation (+15 pts vs +10 pts). À noter également que cet écart se creuse au fur et à mesure que le nombre de classes à prédire augmente, ce qui n'était pas le cas lors de la première modélisation (écart stable). Comme pour la première modélisation, le décor biaise la palette de couleurs.

Exemple :



4. Modèles sur K-means des couleurs

Après les résultats encourageants obtenus avec la modélisation sur la palette de couleurs, nous souhaitions tester une autre façon d'extraire les couleurs d'une image. Il nous a semblé intéressant d'utiliser un algorithme de clustering.

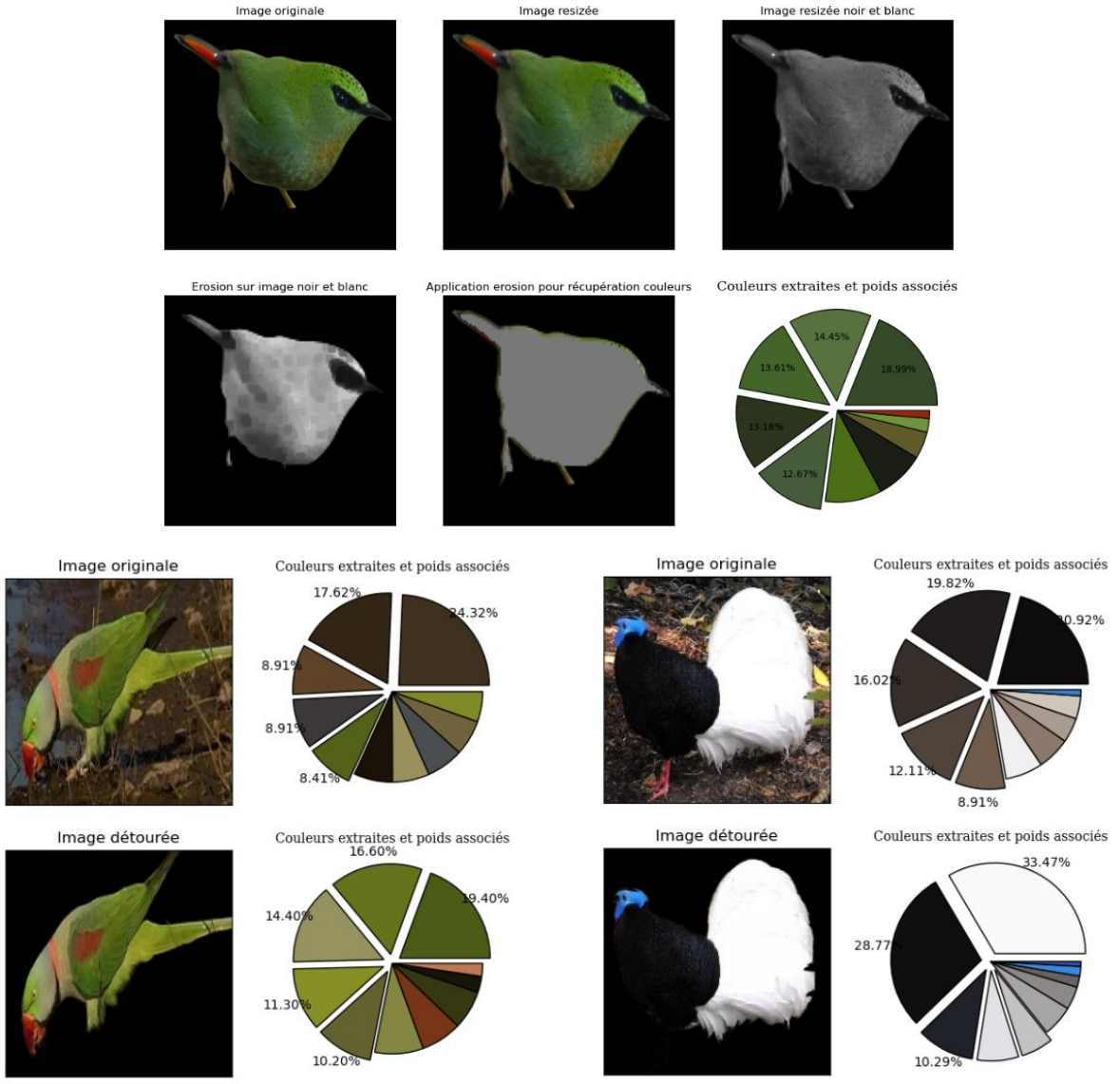
Création de la base

Une nouvelle fonction permettant l'extraction des couleurs principales de l'image via la méthode des K-means a été créée. L'image est ainsi découpée en un nombre de couleurs défini. Cette fonction reprend les principales étapes déjà implémentées lors de la création de la palette de couleurs : seule la dernière étape (le découpage) est différente.

Étapes d'extraction pour une image détourée :

Lecture image (224, 224, 3) >> Redimensionnement (120, 120, 3) >> Passage en noir et blanc >> Définition de l'érosion >> Application de l'érosion >> Découpage en x couleurs selon la méthode des K-means (10 pour tester)

Quelques exemples de résultats de la fonction (paramètre fixé à 10 couleurs) :



Pour chaque couleur extraite, différents indicateurs sont calculés à partir des pixels qui la composent :

- Les indicateurs de position des couleurs R, V, B + gris (moyenne, médiane, Q1, Q3).
- Les indicateurs de dispersion des couleurs R, V, B + gris (écart-type et intervalle interquartile).
- Le poids en pourcentage de pixels par rapport à l'image totale.

Pour la constitution de la base, le paramètre a été fixé à 14 couleurs à extraire par image (seuil défini à partir de la méthode de coude sur une vingtaine d'images).

Bien que l'algorithme des K-means soit une méthode rapide, les temps de traitement s'avèrent particulièrement longs en comparaison aux deux précédentes modélisations (Un peu plus d'1 image/seconde soit 200 fois plus long que pour le premier jeu de données).

Pour cette raison, seules 10 classes ont été sélectionnées aléatoirement pour cette modélisation.

Nous obtenons ainsi deux jeux de données (originales et détourées) composés chacun de 1 600 lignes et 353 colonnes :

- Indicateurs de position et dispersion (6) pour chaque couleur extraite (14) par RVB et gris (4) soit 336 colonnes (6*14*4).
- Poids de chaque couleur (14).
- Source des données (échantillon, classe et nom de l'image).

Pré-processing et réduction de dimension

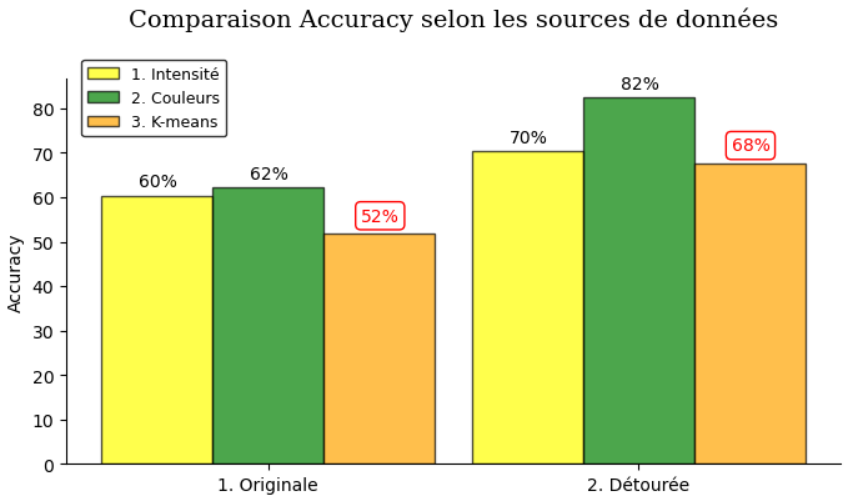
Pour le pré-processing et la réduction de dimension, les fonctions et classes déjà définies précédemment ont été utilisées. À noter que le nombre de composantes conservées est équivalent à la deuxième modélisation (palette de couleurs).

Modélisation et tests

Contrairement aux précédentes modélisations, il n'y aura donc qu'une seule évaluation portant sur les 10 classes sélectionnées lors de la création de la base. Les modèles utilisés restent les mêmes.

Évaluation et comparaison des modélisations

Pour cette modélisation, nous obtenons des résultats inférieurs aux deux premières, ce qui est plutôt déceptif car, les données extraites semblaient pertinentes lors de la visualisation graphique. Ce test ne sera donc pas élargi à plus de classes.



VII. Transfer learning

1. MLFlow

Nous avons souhaité appliquer le cours en créant un serveur de tracking MLFlow. L'objectif de cet outil est de suivre l'entraînement des modèles sous forme de *runs*, et de sauvegarder un certain nombre de paramètres et de métriques ainsi qu'afficher des graphiques. Nous avons utilisé la fonction *autolog()* qui permet de choisir automatiquement les meilleures métriques à suivre pendant l'entraînement.

Pour pouvoir tester différents paramètres pour le modèle, nous avons passé en arguments la *batch_size*, le nombre d'*epochs* ainsi que la *learning_rate*, cette dernière étant utile durant la phase 2, abordée plus loin. L'objectif est de pouvoir relancer le même script plusieurs fois en faisant varier ces paramètres, et comparer les différents *runs*. Puisque ces derniers ont tous le même nom, car ils proviennent du même fichier, nous avons commencé par ajouter des tags avec le nombre de classes, la présence d'augmentation de données ou non, etc. Par la suite, j'ai remplacé cette méthode par des descriptions indiquant les conditions d'entraînement de chaque modèle.

2. Transfer learning

La tâche principale de notre projet étant de classifier des images d'oiseaux, il est très intéressant de se baser sur des modèles déjà existants. Nous avons utilisé le transfer learning avec VGG16, MobileNetV2 et EfficientNet. La méthode que nous utilisons pour faire du transfer learning consiste à importer le modèle de base et à le « geler », c'est-à-dire garder les poids pré-entrainés sans les modifier durant l'entraînement. Nous avons ensuite ajouté les couches suivantes :

- *GlobalAveragePooling2D()* : permet de réduire la dimensionnalité du modèle en conservant les caractéristiques importantes.
- *Dense(1024, activation = 'relu')* : une couche dense avec 1024 neurones, avec l'activation *Rectified Linear Unit* permettant de capturer la non linéarité dans les données.
- *Dropout(rate = 0.2)* : supprime 20% des connexions neuronales de manière aléatoire, pour prévenir le surapprentissage.
- *Dense(512, activation = 'relu')* : la même couche que précédemment, avec moitié moins de neurones, pour alléger le modèle en lui permettant de se concentrer sur l'essentiel.
- *Dropout(rate = 0.2)* : même utilité que précédemment.
- *Dense(n_class, activation = 'softmax')* : dernière couche dense avec un nombre de neurones égal au nombre de classes, et qui via *softmax*, donne les probabilités d'une image d'appartenir à chaque classe.

Avant de lancer l'entraînement, il est possible de faire de l'augmentation de données, c'est-à-dire d'augmenter artificiellement le nombre d'images en appliquant des translations, rotations, symétries, etc. Le modèle ayant davantage d'images à étudier, cela peut améliorer ses performances. Néanmoins, cela augmente aussi fortement le temps d'entraînement. Les premiers modèles n'utilisent donc pas cette technique, afin de tester plus de combinaisons de paramètres en un temps réduit.

Avant d'aborder les paramètres utilisés ainsi que les premiers résultats, il faut évoquer l'entraînement en deux parties : comme précisé plus haut, on "gèle" d'abord les poids du modèle de base en entraînant uniquement les couches que l'on a ajoutées. Pour améliorer encore les résultats du modèle, nous pouvons « dégeler » un certain nombre de couches parmi les dernières, tout en gardant les couches ajoutées. Il faut, pour cette phase, réduire la learning rate.

3. Modèle VGG16

Note : les entraînements sur VGG16 n'ont pas été intégrés au tracking MLFlow.

La première tentative de création de modèle via transfer learning a naturellement été avec VGG16. Ce réseau neuronal convolutif est entraîné sur des millions d'images de catégories différentes. Il était donc plausible qu'il soit adapté à la reconnaissance d'oiseaux. Il s'agit du modèle utilisé dans le cours pour introduire et expliquer le principe du transfer learning, c'est pour cette raison qu'il a été choisi pour être testé en premier.

On se propose de réaliser une première approche en utilisant la génération d'images permettant de normaliser les données. De plus, on placera les mêmes couches en sortie de VGG16 que dans le cours. Enfin, toutes les couches de VGG16 sont gelées.

Cette première approche a pour vocation d'observer ce que l'on peut obtenir d'un modèle de transfer learning : nous n'avons aucune attente particulière.

La valeur de *val_loss* tourne autour de 4 pendant l'apprentissage, mais remonte à plus de 214 lors de l'évaluation du modèle. De plus, la *val_accuracy* est moyenne pendant l'apprentissage (maximum à 0.22), mais pourrait être améliorée avec plus d'*epochs*. En effet, elle n'a cessé de croître pendant l'apprentissage, on peut donc imaginer qu'elle continuerait sur cette tendance avec un apprentissage plus long.

Cependant, lors de l'évaluation sur le set de test, la valeur de précision est de 0.08, ce qui est mauvais.

L'ensemble de ces métriques montrent un surapprentissage. Pour le limiter, on pourra augmenter le nombre d'*epochs* qui était très réduit (cinq *epochs*) lors de ce premier apprentissage. On pourra ensuite intégrer des callbacks, puis dégeler certaines couches du modèle VGG16.

Le modèle VGG16 sera néanmoins laissé de côté au profit d'EfficientNet. En effet, nous travaillions en parallèle sur différents modèles. Lorsque nous avons obtenu ces résultats peu prometteurs, le modèle EfficientNet proposait déjà de biens meilleures performances avec notamment une précision dépassant 0.80.

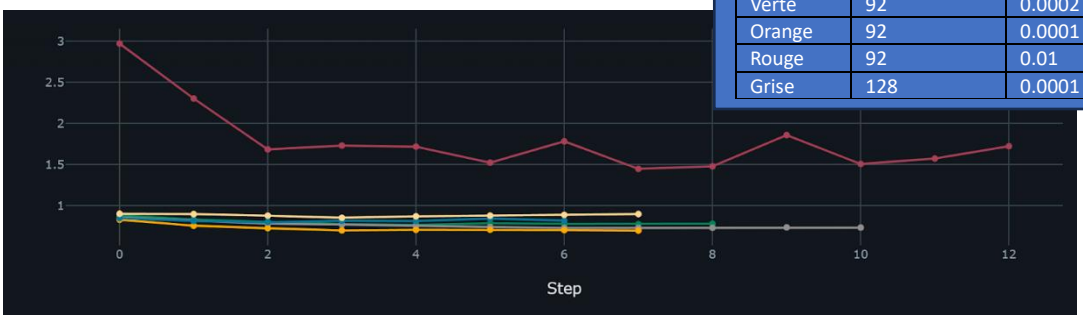
Par curiosité, nous avons tout de même entraîné un second modèle sur 443 classes, en dégelant quatre couches et en passant à huit *epochs*. Les résultats sont encore plus mauvais, avec une *val_accuracy* à 0.11.

4. MobileNetV2

Nous avons commencé par modifier la batch_size. En théorie, plus elle est petite, meilleure est la modélisation, parfois au détriment d'une meilleure généralisation.

Voici un graphique avec six *runs* sur MobileNetV2 :

Courbe	Batch_size	Learning_rate
Jaune	16	0.0001
Bleue	32	0.0001
Verte	92	0.0002
Orange	92	0.0001
Rouge	92	0.01
Grise	128	0.0001



On remarque tout de suite que le modèle s'améliore en augmentant la *batch_size*. En la passant à 128, on voit une amélioration. Cependant, la courbe orange montre des performances encore meilleures avec une *batch_size* de seulement 92. Cette différence est due au *learning_rate*, on en conclut que la valeur de *batch_size* peut être limitée à 92.

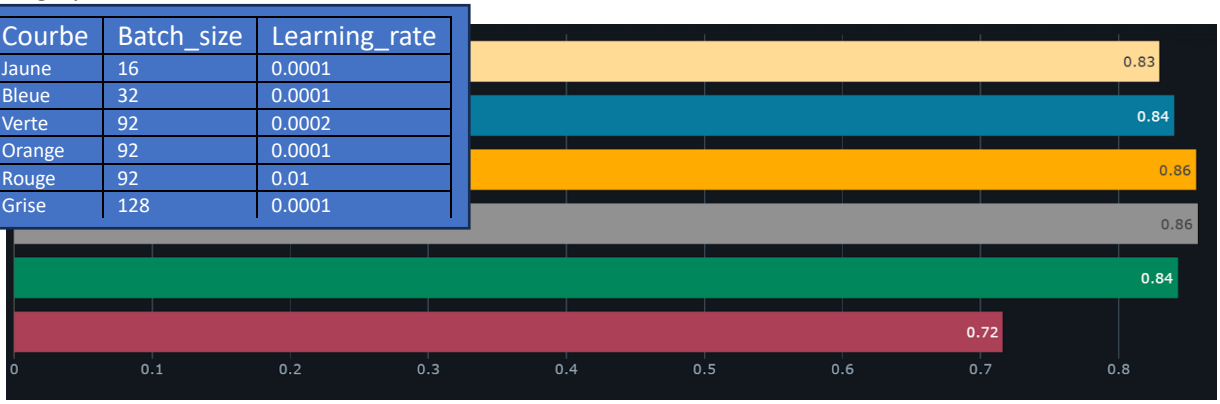
Le paramètre suivant à faire varier est la *learning_rate* pour la phase 2 de l'entraînement, consistant à dégeler les quatre dernières couches de MobileNetV2. La valeur par défaut est 1e-4, soit 0.0001. Nous avons souhaité tester des valeurs plus grandes.

La courbe rouge montre qu'une valeur de *learning_rate* trop grande est très néfaste pour la qualité de l'apprentissage. La courbe verte montre qu'une *learning_rate* deux fois supérieure à celle par défaut n'a pas d'impact significatif sur l'apprentissage.

Tous ces *runs* ont été réalisés sur la moitié du dataset détouré (262 classes) afin d'accélérer les temps d'entraînement. Les scores ne sont donc pas finaux.

En regardant la *validation_accuracy*, qui est plus parlante en termes de performance, on peut analyser le graphe suivant :

Courbe	Batch_size	Learning_rate
Jaune	16	0.0001
Bleue	32	0.0001
Verte	92	0.0002
Orange	92	0.0001
Rouge	92	0.01
Grise	128	0.0001



On obtient des résultats qui sont bons, excepté pour la courbe rouge qui a une *learning_rate* bien supérieure aux autres. Le modèle à retenir est représenté par la barre orange : il a été entraîné avec une *batch_size* de 92 et une *learning_rate* de 0.0001. Concernant le nombre d'*epochs*, si l'on compte la partie gelée puis dégelée, ce *run* totalise 14 *epochs*. Nous utilisons un *EarlyStopping*, avec une patience de 4, pour arrêter l'entraînement en l'absence d'amélioration significative de la *validation_loss*.

5. EfficientNet

Le second modèle utilisé est EfficientNet, qui est décliné en huit versions, de B0 à B7. Pour atteindre le meilleur résultat possible, nous avons d'abord choisi la version B7, censée être plus lourde, mais plus performante. Comme pour les modèles de transfer learning précédents, il suffit d'accoler les couches précédemment décrites à la suite du modèle B7. Avant d'afficher d'autres graphiques, comparons d'abord les résultats du premier *run* (entraîné sur 262 classes), avec le modèle MobileNetV2 :

- EfficientNetB7 : *validation_loss* -> 0.48 & *validation_accuracy* -> 0.91
- MobileNetV2 : *validation_loss* -> 0.69 & *validation_accuracy* -> 0.85

La montée en performances est significative, tout comme la durée d'entraînement : nous sommes passés de 14 minutes (avec les deux phases) pour MobileNetV2 à 54 minutes pour EfficientNetB7. Les entraînements sont réalisés sur une RTX 3070 avec 8Go de VRAM. Comme nous souhaitons privilégier la performance finale du modèle au temps d'entraînement, nous choisissons de conserver EfficientNet plutôt que MobileNetV2.

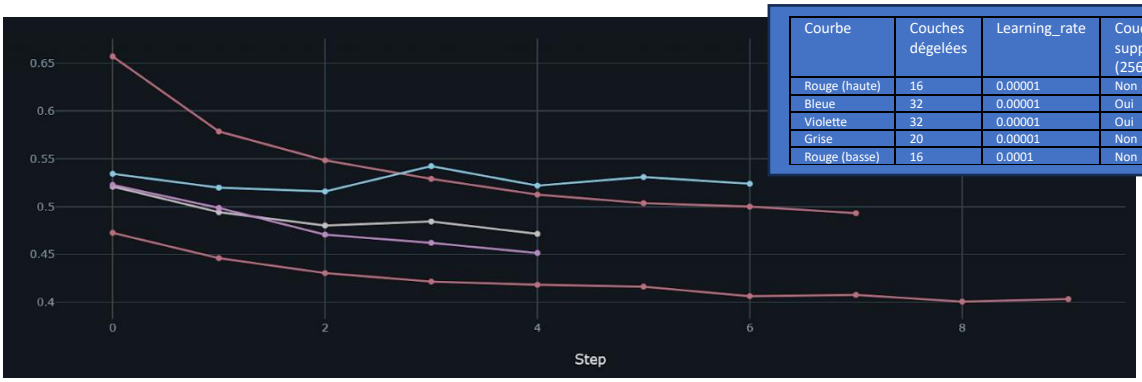
Pour la suite des tests, effectués sur l'entièreté du jeu de données (523 classes, 98 images chacune), les deux *runs* (un par phase) ont pris 1h08. Les résultats obtenus sont les suivants :

- *validation_loss* : 0.5
- *validation_accuracy* : 0.89

Les résultats, satisfaisants, montrent que malgré un nombre de classes doublé, le modèle reste performant pour différencier toutes les classes.

Comme expliqué auparavant, dégeler plusieurs couches du modèle de base pour les ajuster à notre problème de classification améliore le score du modèle, souvent avec 0.02-0.04 points en plus pour la *validation_accuracy*, ce qui n'est pas négligeable.

Il faut néanmoins choisir le bon nombre de couches que l'on souhaite modifier. Pour cela, le plus simple est de tester différentes valeurs et comparer les résultats :



Courbe	Couches dégelées	Learning_rate	Couche dense supplémentaire (256 neurones)	Augmentation de données
Rouge (haute)	16	0.00001	Non	Non
Bleue	32	0.00001	Oui	Non
Violette	32	0.00001	Oui	Oui
Grise	20	0.00001	Non	Oui
Rouge (basse)	16	0.0001	Non	Oui

En plus de changer le nombre de couches dégelées, nous avons fait varier plusieurs paramètres.

On peut noter les hyperparamètres de la couche rouge la plus basse qui a les meilleurs résultats : 16 couches dégelées, avec data augmentation, *learning_rate* par défaut (0.0001) et sans la couche dense supplémentaire. Elle a la meilleure *validation_loss*.

Nous avons ensuite entraîné le modèle sur le dataset réduit, avec non pas 523 classes et 98 images par classe pour *train*, mais 395 classes et 112 images à chaque fois. En théorie, avoir plus d'images et moins de classes améliore le résultat. Les résultats obtenus sont les suivants :

- *validation_loss* : 0.43
- *validation_accuracy* : 0.90

Ce n'est finalement pas beaucoup mieux. Pour approfondir, le même entraînement a été lancé sur les images originales. Les résultats obtenus sont les suivants :

- *validation_loss* : 0.26
- *validation_accuracy* : 0.94

Ces derniers résultats sont étonnants au premier abord. En effet, on imaginait avoir de meilleurs résultats en excluant le décor autour des oiseaux. On peut supposer que les habitats et donc les arrière-plans peuvent être corrélés aux espèces d'oiseaux. Il est rare de voir un pingouin dans le désert tout comme un pigeon sur la banquise. Le modèle a dû trouver des informations pertinentes dans l'arrière-plan pour certaines classes, remontant ainsi le score. Cela n'est bien sûr pas applicable entre plusieurs oiseaux dans la forêt par exemple. Comme on le verra plus loin, l'utilisation de la méthode GradCam pourra nous éclairer sur ce sujet. Cette méthode permet de créer une carte de chaleur qui montre les pixels de l'image que le modèle regarde pour réaliser sa prédiction.

Il est préférable de garder le modèle aux résultats légèrement moins bons car il a l'avantage de fonctionner correctement indépendamment de l'arrière-plan, ce qui permet de travailler sur des images d'oiseaux où l'environnement n'est pas nécessairement cohérent.

6. EfficientNetB0

Dans un souci d'optimisation du temps d'inférence, nous avons tenté d'utiliser la version la plus légère d'EfficientNet : B0. En théorie, accélérer le temps d'inférence se fait au prix de la performance du modèle. Pour vérifier cela, deux modèles ont été entraîné avec les même paramètres sur le dataset détouré et nettoyé, en utilisant B0 d'une part et B7 d'autre part.

En gelant puis dégelant les dernières couches, nous obtenons 0.905 en *validation_accuracy* pour B7, et 0.47 en *validation_loss*. B0 obtient une précision similaire (0.908) mais avec une valeur de perte moins importante, à 0.43.

Les temps d'entraînement sont similaires, B0 étant légèrement plus rapide. N'ayant pas de contrainte de temps, ce point est néanmoins négligeable dans notre cas.

Concernant les 20 pires classes, nous obtenons des résultats très similaires pour les deux modèles, avec un score F1 allant de 0.60 à 0.73.

Le temps d'inférence de la version B0 est trois fois plus rapide que B7, ce qui est un avantage considérable.

Le modèle entraîné avec EfficientNetB0 est donc similaire en tout point, sauf pour le temps d'inférence, pour quel il est trois fois plus performant. C'est pour cette raison que nous choisissons de le conserver, au détriment du modèle entraîné avec EfficientNetB7.

7. Inférence

Pour aller plus loin dans l'évaluation des modèles, il faut passer à l'étape de l'inférence, c'est-à-dire obtenir les prédictions sur des images de test. En effet, il faut réaliser cette opération sur des données dont le label est connu afin de vérifier la cohérence des prédictions. Le set de test contient des images qui n'ont pas été utilisées pour l'entraînement du modèle.

Pour réaliser cette étape, nous avons besoin de sauvegarder le modèle, c'est-à-dire son architecture et les poids optimaux trouvés lors de l'entraînement. La méthode *model.save()* n'a malheureusement pas permis de réaliser cette sauvegarde. Malgré des recherches sur Internet, nous n'avons pas trouvé la cause du problème.

Pour ne pas perdre trop de temps dans ces recherches, nous avons trouvé une autre solution : enregistrer uniquement les poids. La seule contrainte est de devoir redéfinir l'architecture du modèle lorsqu'on veut le reconstruire. Nous implémentons donc une classe proposant une méthode qui construit ce modèle. Si nous devons créer d'autres modèles de cette manière, il suffira alors d'ajouter des méthodes à cette classe.

Une fois les prédictions effectuées, nous obtenons deux listes : une avec les vraies classes et une avec les classes prédites. Pour simplifier l'analyse, il est possible de faire appel à la fonction *classification_report()*, permettant d'obtenir des scores pour chaque classe ainsi qu'un score global. Puisque la classification d'oiseaux n'est pas un domaine critique, comme par exemple le milieu médical, où il faut absolument éviter les faux positifs, nous nous concentrerons sur le score F1, qui mélange les métriques *accuracy* et *recall*.

Pour le modèle entraîné sur 395 classes aux images non détourées, nous obtenons un score F1 de 0.94. Pour les photos détourées, 0.91, soit légèrement mieux que durant l'entraînement.

Les classes qui ont le moins bon score doivent également être observées. En effet, une moyenne de 0.94 peut être constituée de scores très hauts et très bas. Le paramètre *output_dict* de la fonction *classification_report()* permet d'obtenir les valeurs pour chaque classe sous forme de dictionnaire. Nous avons trié dans l'ordre croissant les scores F1 de toutes les classes, puis avons affiché seulement les 20 premières classes. Voici le résultat pour les images non détourées :

RED TAILED HAWK, score F1: 0.6	PURPLE FINCH, score F1: 0.73
BREWERS BLACKBIRD, score F1: 0.77	NORTHERN GOSHAWK, score F1: 0.77
ASHY THRUSHBIRD, score F1: 0.78	AUSTRAL CANASTERO, score F1: 0.78
FASCIATED WREN, score F1: 0.78	EASTERN GOLDEN WEAVER, score F1: 0.79
GYRFALCON, score F1: 0.79	HOUSE FINCH, score F1: 0.79
BROWN HEADED COWBIRD, score F1: 0.8	CANARY, score F1: 0.8
RED BILLED TROPICBIRD, score F1: 0.8	ANTBIRD, score F1: 0.81
NORTHERN BEARDLESS TYRANNULET, score F1: 0.81	CRESTED COUA, score F1: 0.82
GUINEA TURACO, score F1: 0.82	MERLIN, score F1: 0.82
TEAL DUCK, score F1: 0.82	WOOD THRUSH, score F1: 0.82

à lire de gauche à droite, ligne par ligne, dans l'ordre croissant des scores

Bien que les résultats soient largement meilleurs qu'avec un modèle aléatoire, nous remarquons que certaines classes ont un score très inférieur à 0.94, ce qui montre que le modèle peut encore être amélioré. Le modèle détouré présente un tableau similaire, avec des classes au score bien inférieur à la moyenne également.

Lorsque l'on revient à un entraînement sur 523 classes, le score F1 obtenu est 0.90, sur la partie détourée cette fois-ci, qui est la plus pertinente, toujours avec la data augmentation.

NORTHERN BEARDLESS TYRANNULET, score F1: 0.55	AUCKLAND SHAQ, score F1: 0.58
ABBOTTS BOOBY, score F1: 0.61	FASCIATED WREN, score F1: 0.63
GREATER PEWEE, score F1: 0.65	ASHY THRUSHBIRD, score F1: 0.67
CRESTED SERPENT EAGLE, score F1: 0.67	GILDED FLICKER, score F1: 0.67
NORTHERN GOSHAWK, score F1: 0.67	BROWN HEADED COWBIRD, score F1: 0.68
NORTHERN FLICKER, score F1: 0.68	BREWERS BLACKBIRD, score F1: 0.7
ASHY STORM PETREL, score F1: 0.71	RED TAILED HAWK, score F1: 0.73
RED WINGED BLACKBIRD, score F1: 0.73	AUSTRALASIAN FIGBIRD, score F1: 0.74
CROW, score F1: 0.74	GREY CUCKOOSHRIKE, score F1: 0.74
IMPERIAL SHAQ, score F1: 0.74	MALAGASY WHITE EYE, score F1: 0.74

à lire de gauche à droite, ligne par ligne, dans l'ordre croissant des scores

Le score F1 du modèle est similaire à celui du modèle entraîné sur 395 classes, mais les 20 pires classes sont moins bien notées. Cela peut indiquer que l'augmentation du nombre de classes réduit la performance, notamment car cela peut accroître la confusion entre deux classes similaires.

8. Affichage des mauvaises classifications

Une autre approche pour étudier les images mal classées consiste à les afficher. Pour cela, un script récupère le chemin vers les images dont la classe a été mal prédite. On peut alors en afficher :

Le résultat est intéressant : ici, deux oiseaux sur les huit ont mal été détourés. En revanche, le traitement sur les autres oiseaux semble avoir été correctement effectué. On peut supposer que certains sont trop sombres, tandis que d'autres ressemblent à un oiseau d'une autre classe, rendant le modèle confus.

9. Double modèle

Nous avons souhaité essayer une méthode différente pour réaliser la classification. Sachant qu'un modèle au nombre de classes inférieur performe mieux, l'objectif est d'entraîner deux modèles, chacun sur la moitié du dataset, et de réaliser une double inférence. Nous avons donc lancé l'entraînement sur les 262 premières classes ainsi que les 261 dernières.

Le premier modèle obtient un score F1 de 0.92, soit un peu mieux que pour 523 classes, avec pour la pire classe 0.65.

Le second modèle obtient un score F1 de 0.97, avec cependant la pire classe notée à 0.57. Toutefois seules cinq classes ont un score F1 inférieur à 0.75.

Le principe du double modèle est de pouvoir réaliser une double inférence, comme un concours de modèles. L'image est présentée aux deux modèles. Chacun donne une prédiction avec un pourcentage de fiabilité associé. On choisit alors la prédiction avec la meilleure fiabilité.

La faille de cette méthode est la suivante : chaque modèle essayera à tout prix de classer l'image, même s'il ne connaît en fait pas la classe. On peut alors obtenir un haut score de la part d'un modèle qui ne connaît pas la classe réelle.

Nous réalisons les prédictions, avec quatre listes en sortie : deux qui correspondent aux classes prédites et deux aux probabilités associées à ces classes. Il suffit ensuite de parcourir les listes en ne gardant que la classe qui a la probabilité la plus haute entre les deux modèles, pour chaque image. On stocke l'identifiant de la classe (0<id<524) dans une nouvelle liste, qui ressemble à cela :

```
[0, 0, 0, 0, 0, 0, 0, 519, 0, 520, 0, 0, 32, 519, 283, 479, 519, 519,
354, 0, 0, 344, 1, 1, 344, 1, 344, 1, 1, 1, 1, 243, 378, 344, 1, 330,
1, 296, 1, 1, 344, 400, 2, 261, ...]
```

Normalement, nous devrions obtenir une suite de 0, puis de 1, puis de 2, etc. En effet, les images sont importées dans l'ordre. Or, ici, il y a beaucoup de classes mal prédites, ce qui présage une performance médiocre.

Pourtant, le score F1 obtenu est de 0.89, soit très similaire au modèle classique. En revanche, les pires classes sont cette fois-ci vraiment mauvaises :

GRAY PARTRIDGE, score F1: 0.17	GOLDEN CHEEKED WARBLER, score F1: 0.38
RED TAILED HAWK, score F1: 0.55	TOWNSENDS WARBLER, score F1: 0.56
BLACKBURNIAM WARBLER, score F1: 0.6	COMMON HOUSE MARTIN, score F1: 0.6
HEPATIC TANAGER, score F1: 0.61	ASHY THRUSHBIRD, score F1: 0.62
BREWERS BLACKBIRD, score F1: 0.62	CINNAMON ATTILA, score F1: 0.62
EMU, score F1: 0.62	ALTAMIRA YELLOWTHROAT, score F1: 0.63
GREY CUCKOOSHRIKE, score F1: 0.63	FAN TAILED WIDOW, score F1: 0.64
ABBOTTS BOOBY, score F1: 0.65	EASTERN WIP POOR WILL, score F1: 0.65
FASCIATED WREN, score F1: 0.65	ASIAN DOLLARD BIRD, score F1: 0.68
SQUACCO HERON, score F1: 0.68	TRICOLORED BLACKBIRD, score F1: 0.68

à lire de gauche à droite, ligne par ligne, dans l'ordre croissant des scores

On se rend compte que les confusions relevées plus haut dans les listes affectent très fortement les scores, avec des classes très mal prédites, bien plus que dans le modèle principal.

Au vu des métriques, de l'entraînement et de l'inférence plus complexes, il n'est pas pertinent de garder un tel modèle, mais l'expérience s'est avérée intéressante.

10. Changer d'approche : ordres et familles

Comme vu précédemment, il est possible d'adopter une autre approche pour la classification de ce dataset. En regroupant les images par familles, ou par ordres d'oiseaux, le nombre de classes est drastiquement réduit et la quantité d'images par classe augmentée.

Une fois les images redistribuées, nous avons entraîné deux modèles (familles et ordres) avec les hyperparamètres de notre meilleur modèle trouvé précédemment. Les résultats sont très encourageants :

- Classification par familles : *validation_accuracy* : 0.92
- Classification par ordres : *validation_accuracy* : 0.95

Malheureusement, dans le cadre de ce projet, nous manquons de temps pour améliorer ces modèles. Cependant, plusieurs pistes pourraient être explorées pour améliorer encore leurs performances :

- Chercher des hyperparamètres plus adaptés à chacun des cas de figure. En effet, nous avons utilisé ceux qui donnaient les meilleurs résultats pour une classification par espèces, mais peut-être ne sont-ils pas optimaux pour une distribution différente.
- Augmenter les données : cette redistribution pose un problème majeur. En effet, il y a un fort déséquilibre dans la quantité de données entre les familles et entre les ordres. Par exemple, l'ordre des passereaux regroupe énormément d'espèces et de familles différentes. On pourrait recourir à la génération de données, ou chercher d'autre sources de photos originales sur Internet.

L'idée originale était de proposer une réduction du nombre de classes afin d'obtenir des modèles plus précis. Cependant, nous avons déjà d'excellents résultats sur 523 classes. Toutefois, ces modèles pourraient servir dans un contexte de production comme double ou triple vérification d'une prédiction. Lors de la prédiction réalisée sur une image, on présente l'image aux trois modèles. Si les prédictions sont cohérentes (l'espèce appartient à la famille qui appartient à l'ordre), on peut donner une réponse fiable à l'utilisateur. En revanche, dans le cas contraire, on donne à l'utilisateur les trois prédictions en lui indiquant l'incohérence et les scores de chaque prédiction.

VIII. GradCam

Nous avons créé plusieurs modèles de deep learning en suivant différentes stratégies. Le problème de cette approche est l'explicabilité du modèle. En effet, nous ne sommes pas réellement capables d'expliquer ce que nos modèles font pour obtenir de tels résultats. La méthode GradCam va nous permettre de combler partiellement ce manque de connaissances.

1. Principe technique

Le but de GradCam est d'établir une carte de chaleur indiquant quelles parties de l'image ont permis de réaliser une prédiction. Le principe est d'observer la sortie du modèle pour chaque pixel et d'établir son importance, c'est-à-dire à quel point il a contribué à la prédiction.

Pour cela, on calcule le gradient du score de la classe de sortie par rapport aux cartes de caractéristiques de la dernière couche convolutive. En réalisant le produit matriciel entre ce gradient et la carte de caractéristiques de la dernière couche de convolution du modèle, on obtient la heatmap. On peut alors la superposer à l'image initiale pour se rendre compte des zones de l'image qui ont été utilisées par le modèle pour réaliser la prédiction.

2. Interprétation

La méthode GradCam permet de prouver qu'un modèle fonctionne en mettant en évidence les zones pertinentes d'une photo qui permettent la prédiction d'une classe. Par exemple, les deux images suivantes montrent que le modèle se base principalement sur la zone du bec et de la tête pour reconnaître un albatros.

On peut également montrer le dysfonctionnement du modèle sur certaines images. Dans l'exemple suivant qui a mal été prédit, on voit que le modèle porte beaucoup d'attention à l'extrémité du bec, ce qui peut sembler insuffisant pour identifier un oiseau. De plus, il utilise également d'autres pixels moins pertinents. En effet, la zone bleu clair à gauche de l'oiseau a trop été utilisée dans la prédiction alors qu'elle n'apporte aucune information.

IX. Critique méthodologique

Après deux mois de travail sur ce projet de datascience, nous pouvons réaliser une analyse de la méthodologie suivie et la critiquer. Nous avons suivi un processus presque complet, en partant de la collecte de donnée et en arrêtant juste avant la mise en production.

1. Liste de tâche

Très rapidement, une liste de tâches a été mise en place afin de se répartir le travail. C'était une très bonne manière de contrôler le partage du travail, nous permettant de pouvoir adapter la charge à l'avance ou au retard de chacun sur les cours. Cependant, au fur et à mesure du projet, nous avons cessé ce suivi d'activité sur cette liste. La raison est probablement la diversité des approches : les tâches étaient tellement différentes et exécutées en parallèle (machine learning vs deep learning par exemple) que nous n'avons pas jugé nécessaire de continuer ce suivi. C'était cependant une erreur. En effet, pour un suivi correct, une liste de tâche exhaustive doit être tenue tout au long du projet.

2. Répartition des tâches

La répartition des différentes tâches a été correctement réalisée tout au long du projet. Cela nous a permis d'avancer de manière constante, rapide et simultanée sur différents sujets. De cette manière, nous avons exploré un maximum de possibilités à travers plusieurs approches machine learning et deep learning tout en assurant une architecture de projet propre et entretenue.

3. Gestion des constantes

Le gros point noir du projet a été la duplication de code, ce qui nous a causé des problèmes. Plusieurs d'entre nous débutions en python, et même en développement et n'avaient donc pas de notion en bonnes pratiques. Les plus anciens ont mis en place certains process afin d'avoir un ensemble de fichiers cohérents sur le dépôt github.

Cependant, nous ne les avons pas suivis avec suffisamment de rigueur. Le problème majeur a été causé par un fichier regroupant les constantes, telles que les chemins menant à différentes bases de données (réduite, détourée, non détourée…). La bonne approche aurait été d'imposer un nombre réduit de noms de constantes cohérente, liées à des chemins cohérents, et d'être strict sur l'application de ces noms et chemins.

Chacun a cependant adapté ce fichier au cas par cas, avec des chemins et des noms de variables différents pour chaque machine. À cause de cette diversité, des entraînements ont été réalisés sur de mauvais dataset, n'ayant pas subi de preprocessing, sans que nous ne nous en rendions compte. C'est très tard, au moment de rendre le rapport, que nous avons pris connaissance de l'erreur. Dans un contexte réel, cette erreur aurait été critique et nous aurait sans doute coûté très cher.

4. Suite de test

Aucune suite de tests rigoureuse n'a été mise en place pour vérifier, notamment, la non-régression des scripts créées pendant le projet. Cette lacune n'est cependant pas due à un oubli, mais à un manque de temps et de compétence. En effet, mettre en place une suite de tests correcte demande de la rigueur, des connaissances en la matière et du temps.

Nous avons cependant conscience que ce « gain » de temps, n'en est pas réellement un. En effet, une fois en places, les tests permettent de gagner un temps considérable en débugging et en déploiement.

5. MLFlow

L'utilisation de l'outil MLFlow, instauré très tôt dans le projet, a été essentiel pour le suivi d'entraînement des modèles. Il nous a permis d'analyser les différentes combinaisons d'hyperparamètres et d'architecture afin de trouver la meilleure.

X. Conclusion

Ce projet visait à créer un modèle de classification d'espèces d'oiseaux en utilisant des photos. Nous avons basé notre travail sur un dataset contenant près de 90 000 images réparties entre 523 classes.

Nous avons d'abord exploré la base de données, ce qui nous a poussés à supprimer deux classes, puis à tester plusieurs manipulations de rééchantillonnage. Cette phase d'exploration a également été l'occasion de prévoir différentes stratégies pour la création de modèles.

Une fois le nettoyage terminé, nous avons travaillé sur différents modèles en parallèle : plusieurs tentatives ont été réalisées sur des modèles de machine learning basés sur les couleurs des images. Les résultats sont satisfaisants pour un nombre réduit de classes. Cependant, sans surprise, le machine learning ne suffit pas pour un dataset de 523 classes.

Les résultats sur les modèles de deep learning sont très hétérogènes. Le modèle basé sur VGG16 a très vite été abandonné tant ses performances étaient basses. En revanche, contrairement à nos attentes, nous avons réussi à construire un modèle *from scratch* assez performant.

C'est finalement avec EfficientNetB2 que nous obtenons les meilleurs résultats, avec un score de précision atteignant 95%. La classe subissant les pires prédictions obtient un score de précision de 83%, ce qui reste excellent. De plus, plusieurs classes ont un score de précision de 100%.

Pour faire suite à ce projet, plusieurs pistes peuvent être explorées. Tout d'abord, bien que nous ayons commencé par utiliser EfficientNetB7, censé être le plus performant des modèles EfficientNet, nous nous sommes rabattus sur des versions inférieures pour de meilleurs résultats. Nous pourrions donc tenter de comprendre pourquoi les modèles utilisant la version B7 donnent de moins bons résultats et trouver des hyperparamètres qui les rendraient meilleurs.

La chaîne de pré-processing étant complètement automatisée, nous pourrions alimenter le modèle en continu, en requêtant régulièrement des bases de données d'ornithologie et en réentraînant le modèle. De cette manière, nous pourrions augmenter le nombre de classes. En effet, on estime qu'il y a près de 10 000 espèces d'oiseaux dans le monde, il y a donc une belle marge d'amélioration.