L2 mini-projets
Challenge areal-groupe satellite
Florian GUERIN, Eric AUBINAIS, François BALDUCCHI, Léandre BISCHOFF, Baptiste BRIOUZE,
Petru-Florin CISLARU
Adresse du challenge : https://codalab.lri.fr/competitions/399
Repo github : https://github.com/SatelliteISS/SatelliteISS2(le dossier PREPRO)
Présentation youtube de notre défi : https://www.youtube.com/watch?v=e9S77Ap30x4&feature=youtu.be
ID de la dernière soumision : 8250

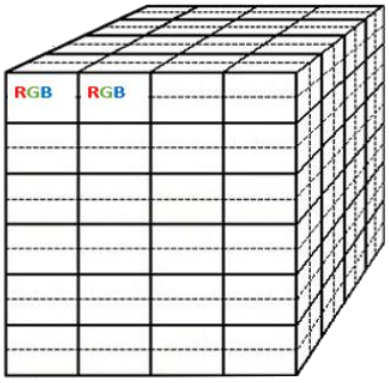
SATELLITE

Notre projet consiste à la reconnaissance et au tri d'images de paysages prises par des satellites. Que ce soit pour les applications militaires (repérage de sites sensibles...), écologiques (déforestation, fonte des glaces...) ou économiques (surfaces viables pour l'agriculture…), la reconnaissance de paysages est nécessaire vu le nombre de données exploitables. Le but de notre équipe était donc de réussir à s'approprier le code du défi codalab pour réussir à créer un algorithme de reconnaissance aussi efficace que possible. Pour se faire, nous avons formé trois binômes, les binômes preprocessing se chargeant de prétraiter les images, le groupe prédiction chargé des algorithmes d'apprentissage et pour finir le groupe visualisation chargé de représenter nos résultats des façons les plus simples et explicites possibles. Le jeu de donné que l'on nous a fourni était composé de 9100 images de 128*128 pixels en colorisation standard RGB réparties en 13 classes (bord de mer, montagne, lac…). Notre meilleur résultat avec les techniques pré-conseillées était de 0,8646 cependant nous préférons garder le score de notre algorithme qui est certe légèrement plus faible, mais compte tenu de l'évolution potentielle de notre code ainsi que pour sa partie innovante nous préférons garder un résultat de 0,8364.

Sources :
https://scikit-learn.org/stable/auto_examples/classification/plot_classifier_comparison.html
https://scikit-learn.org/stable/auto_examples/model_selection/plot_confusion_matrix.html#sphx-glr-auto-examples-model-selection-plot-confusion-matrix-py
https://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html

Preprocessing

Le but du binôme préprocessing, est de prétraiter les données pour l'intelligence artificielle. Très généralement, le prétraitement des données consiste à transformer des tableaux énormes, en des plus petits ne contenant que l'essentiel.



Dans notre cas, nos données brutes sont des images satellites, transformées en des gros tableaux de nombre représentant leur résolution ainsi que la couleur de chaque pixel. Chaque image est ainsi vu comme un tableau tri-dimensionnel dont deux dimensions représentent la résolution. Ainsi, pour chaque couple des deux premières dimensions, une troisième dimension représentant les composantes rgb est ajoutée. Dans notre cas, nous avons des tableaux de tailles 128x128x3 où le « 128x128 » représente la résolution de l'image.

Notre but va donc être de transformer tous ces tableaux 128x128x3 en des tableaux ne conservant que l'essentiel de taille 32x32.

Originalement, nous utilisions des algorithmes PCA qui appliquaient une régression linéaire sur nos images brutes. Cependant, nous avons voulu utiliser notre propre code.

Dans un premier temps, nous cherchons à « compacter les couleurs ». C'est-à-dire réduire le nombre de dimensions des tableaux : passer de 3 à 2. En effet, plus nous avons de dimensions, plus le travail sera énorme pour l'intelligence artifielle (de l'ordre de n^3 pour le tri-dimensionnel contre n^2 pour le bi-dimensionnel). De plus, étant donné que nous n'avons pas qu'une seule image à traiter, cela permet aussi d'être plus précis et ainsi d'éviter le sur-apprentissage.

L'idée ici est donc de supprimer les trois composantes rgb de chaque pixel pour ne garder plus qu'une seule composante, qui représentera donc une « tonalité de couleur » pour chaque pixel. Pour faire ceci, nous faisons une moyenne pondérée sur les composantes rgb de chacun des pixels. Pour y parvenir, nous utilisons des « poids » pour chacune des composantes, permettant ainsi l'unicité des nouvelles valeurs. En effet, si nous avions fait une moyenne classique, il y aurait plusieurs cas où certains pixels différent se seraient retrouver être les mêmes. Par exemple le pixel (100,100,100) et le pixel (0,45,255) auraient tous les deux donner la même valeur.

```python
def reduct(X):
    XX = np.array(X)
    YY = []

    a = 0.299
    b = 0.587
    c = 0.114

    for j in range(len(XX)):
        tmp = []
        for i in range(int(len(XX[j])/3)):
            new_value = a*XX[j][3*i] + b*XX[j][3*i+1] + c*XX[j][3*i+2]
            tmp.append(new_value)
        YY.append(np.array(tmp))
    return np.array(YY)
```

La seconde partie de notre code préprocessing est de réduire la taille du tableau, passer d'une taille « 128x128 » à une taille « 32x32 ». On procède de la même manière que pour le compactage des couleurs mais cette fois ci en prenant des « carrés » de 4x4 pour réduire le tableau. Cependant, cette fois-ci la pondération ne se fait pas avec des constantes initialisées au début mais lors de l'algorithme afin d'optimiser le préprocessing, et que cela dépende ainsi de comment sont organiser les 4x4 pixels. Le but de cet algorithme est donc de réduire la taille des tableaux considérablement.

```python
def transform(X):
    XX = reduct(X)
    YY = []
    for k in range(len(XX)):
        nn = np.array(XX[k][:]).reshape(128,128)
        new_image = []
        for j in range(32):
            for i in range(32):
                new_value = 0
                for u in range(4):
                    for v in range(4):
                        new_value += nn[4*j+u][4*i+v]
                new_value /= 16.
                new_image.append(new_value)
        YY.append(np.array(new_image))
    return np.array(YY)
```

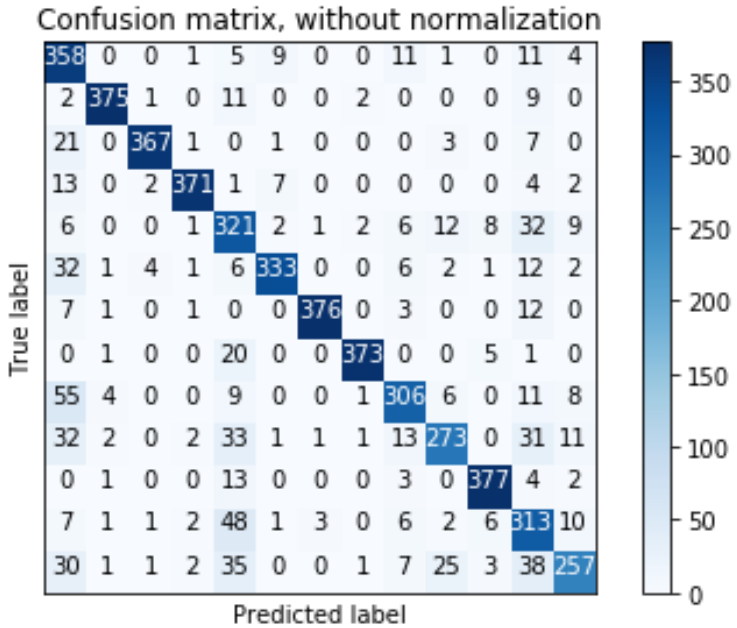
Modèle prédictif

Notre binôme doit coder un modèle qui prend des données d'images prises par des satellites et prédit quel paysage cela représente. Ces données ont été auparavant traitées par le binôme Preprocessing. Pour cela, nous allons utiliser la bibliothèque Python sklearn, qui permet de faire du machine learning en apprentissage supervisé efficacement avec une grande variété de méthodes. Nous utiliserons également DataManager pour gérer nos données (ensemble de train et de test). Le modèle initial est celui du Starting Kit, nous avons rajouté des modèles différents de sklearn, comme le modèle Gaussian et d'autres par la suite. Nous avons rajouté la possibilité de sélectionner le modèle voulu au moment de créer l'objet pour faciliter les comparaisons avec les hyper paramètres souhaités. Ceci est la classe BasicClassifier. Cette classe nous a servi à optimiser les hyper-paramètres des différents classifieurs. Nous avons ensuite fait la classe baselineModel, qui utilise un ensemble de classifiers différents, notamment un pipeline utilisant d'abord Prepro puis un ensemble de Gaussian (contenus dans BasicClassifier) qui s'entraînent en parallèle et qui votent ensuite pour chaque image donnée à quelle classe elle appartient. La classe qui obtient le plus de vote est considérée comme la bonne réponse car nous utilisons la méthode de vote "soft" qui prend également en compte le pourcentage de certitude de chaque classifier, contrairement à la méthode "hard" qui n'utilise que l'avis des classifiers, celle-ci est donc généralement moins efficace.

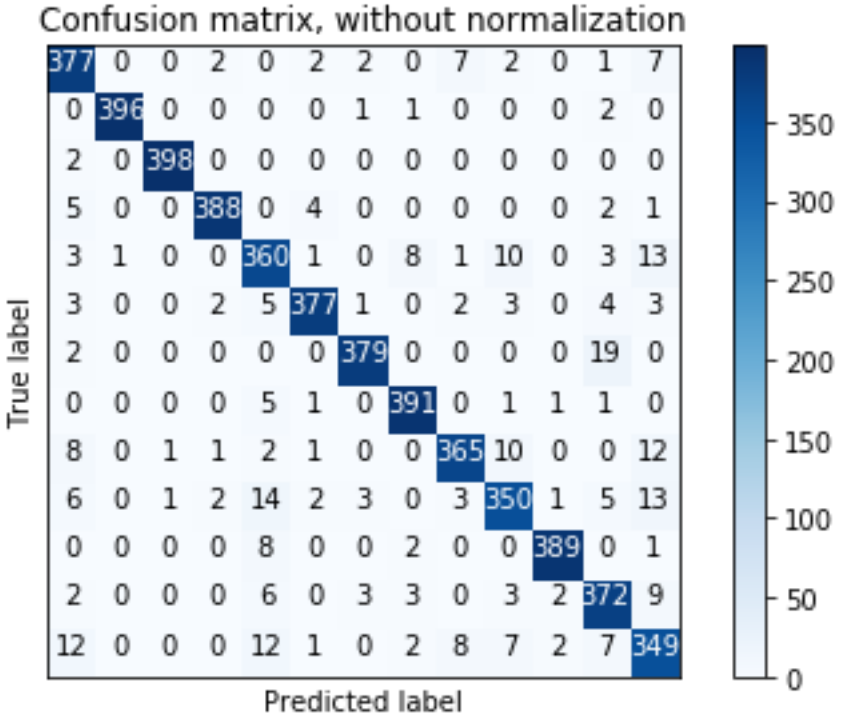
Algorithme :

- Prendre les données traitées
- Les charger et séparer avec DataManager
- Charger une instance du modèle
- Entraîner le modèle avec les données Train (chaque classifier s'entraîne, ce qui peut prendre du temps)
- Vérifier les résultats avec Test
- Lancer ensuite une cross-validation (long temps)
- Établir son score en fonction des erreurs de prédiction

Matrice de confusion du modèle Tree de sklearn avec 10 en depth_max



Matrice de confusion du modèle RandomForest de sklearn avec 10 en depth_max et n_estimators=10 (10 arbres) :



La méthode RandomForest est pour l'instant la plus efficace, mais a une forte tendance à overfitter les données (les apprendre par cœur). Le modèle Gaussian étant très long à entraîner, nous n'avons pas eu le temps de récupérer ses taux de réussite. Nous en rajouterons par la suite.

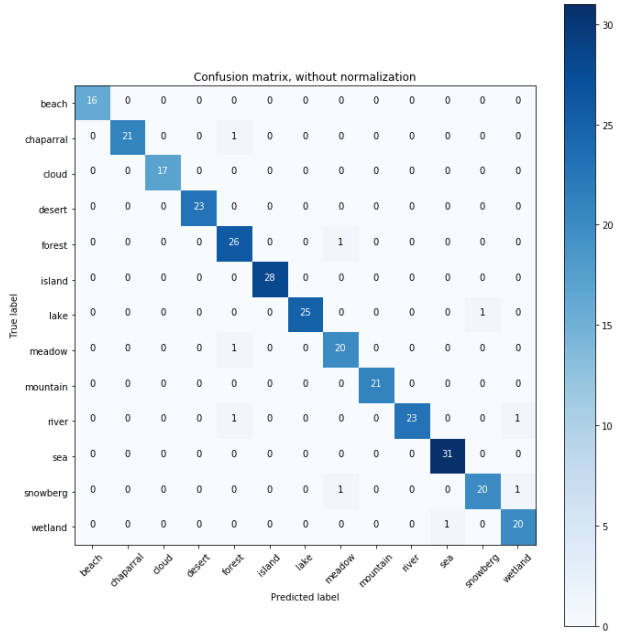
Method	**NaiveBayesor Gaussian classifier**	**Linear regression or SVM**	**Decision Tree (max_deapth = 10)**	**Random Forest (max_deapth = 10)**	**Nearest Neighbors**	**Voting Classifier**
Training	**0.81**	**0.92**	**0.85**	**0.94**	**0.91**	**0.97**
CV	**0.80 (+/- 0.02)**	**0.88 (+/- 0.03)**	**0.67 (+/- 0.02)**	**0.78 (+/- 0.01)**	**0.85 (+/- 0.02)**	**0.76 (+/- 0.03)**
Valid(ation)	**0.81**	**0.92**	**0.85**	**0.94**	**0.91**	**0.87**

Visualisation

L'objectif de la partie visualisation est de présenter les résultats obtenus sous différents graphes afin de les rendre plus compréhensible et apporter une vision d'ensemble pertinente aux données récupérées.

Premier graphe : matrice de confusion

Nous décidons tout d'abord de représenter nos résultats avec une matrice de confusion couplée à une heatmap. Pour ce faire nous utilisons sklearn.metrics d'où nous importons la fonction "confusion_matrix" qui prend en argument nos resultats prédits (Y_hat_train) et les réels(Y_train)




```
from sklearn.metrics import confusion_matrix
cnf_matrix = confusion_matrix(Y_train, Y_hat_train)
```

Le résultat de la fonction est une arraylist, que nous représenterons dans une matrice 2D avec la fonction "plot_confusion_matrix" qui prend en argument la matrice de confusion et nos différentes classes.

```
def plot_confusion_matrix(cm, classes,
                          normalize=False,
                          title='Confusion matrix',
                          cmap=plt.cm.Blues):
```

Analyse du graphe

Cette matrice de confusion nous montre très clairement que bien qu'imparfait, notre modèle semble réussir à prédire correctement la plupart de nos images et à bien les classer. Nous pouvons tout de même remarquer que les classes forest, meadow et wetland sont celles qui sont le plus à même à obtenir des falses positives.

Second graphe : Courbe ROC

Les courbes ROC sont aussi une méthode efficace pour visualizer les performances d'un classsifieur, mais pour les afficher nous avons besoin des labels binairs.

Dans notre code nous transformons les tableaux de « vrais labels » et les tableaux des labels prédits qui sont des tableaux de nombre entre 0 et 12 représentant une classe en tableaux de zéros et uns.

For example for the class 0 we might have an array that looks like this:

Par exemple pour la class 0 nous pouvons avoir un tableau de ce genre:

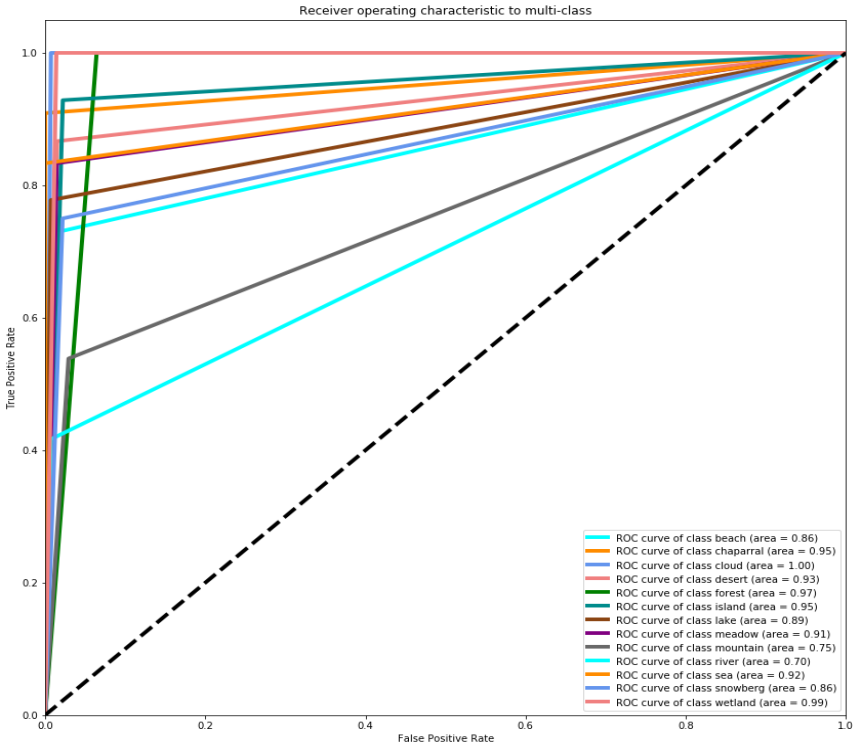
[7, 9, 0, 5, 2, ...]

Qui deviant ensuite:

[0, 0, 1, 0, 0, …]

Cela va nous permettre de calculer et d'afficher les courbes ROC pour chacune des classes.
On oeut calculer les courbes ROC en utilisant la function sklearn.metrics.roc_curve qui possède la signature suivante :
sklearn.metrics.roc_curve(*y_true*, *y_score*, *pos_label=None*, *sample_weight=None*, *drop_intermediate=True*)

Le graphique montrant les courbes ROC pour chacun de nos classifieurs:



Analyse du graphe:

Nous pouvons voir que les classes cloud, desert et d'autre on tune grande aire sous la courbe, ce qui veut dire que notre classifieur fait un bon travail de prediction pour ces classes.

De part nos graphes et nos analyses, nous pouvons voir que notre modèle de prédiction fonctionne bien qu'imparfait. Nous allons essayer lors des semaines restantes de le peaufiner afin d'obtenir le score le plus parfait possible. Nous allons nous pencher sur les prédictions des classes rivers et mountain afin de comprendre ce qui pourrait expliquer une moins bonne précision de la part de notre modèle.
Sur la partie visualisation en elle même, nous allons chercher a utiliser d'autre méthode tel que le t-SNE afin de pouvoir faire une analyse dimensionnelle de nos résultats et d'avoir une plus grande chance de corriger notre modèle.