TP3 : k-NN et DBSCAN

Hormis k, quels autres parametres ou modifications pourrait-on apporter afin de modifier le comportement de l'algorithme ?

- On peut assigner des poids aux features du dataset.
- On peut assigner un poids aux points en fonction de leurs distances du point que l'on cherche (les points les plus près sont plus importants).
- On peut stocker les données d'entraînement sous forme d'arbre binaire, regroupant ainsi les points identiques et rendant l'algorithme de prédiction plus rapide (kd-tree) et moins proie à l'overfitting.

Quelle modification apporter afin d'utiliser l'algorithme en régression plutôt qu'en classification ?

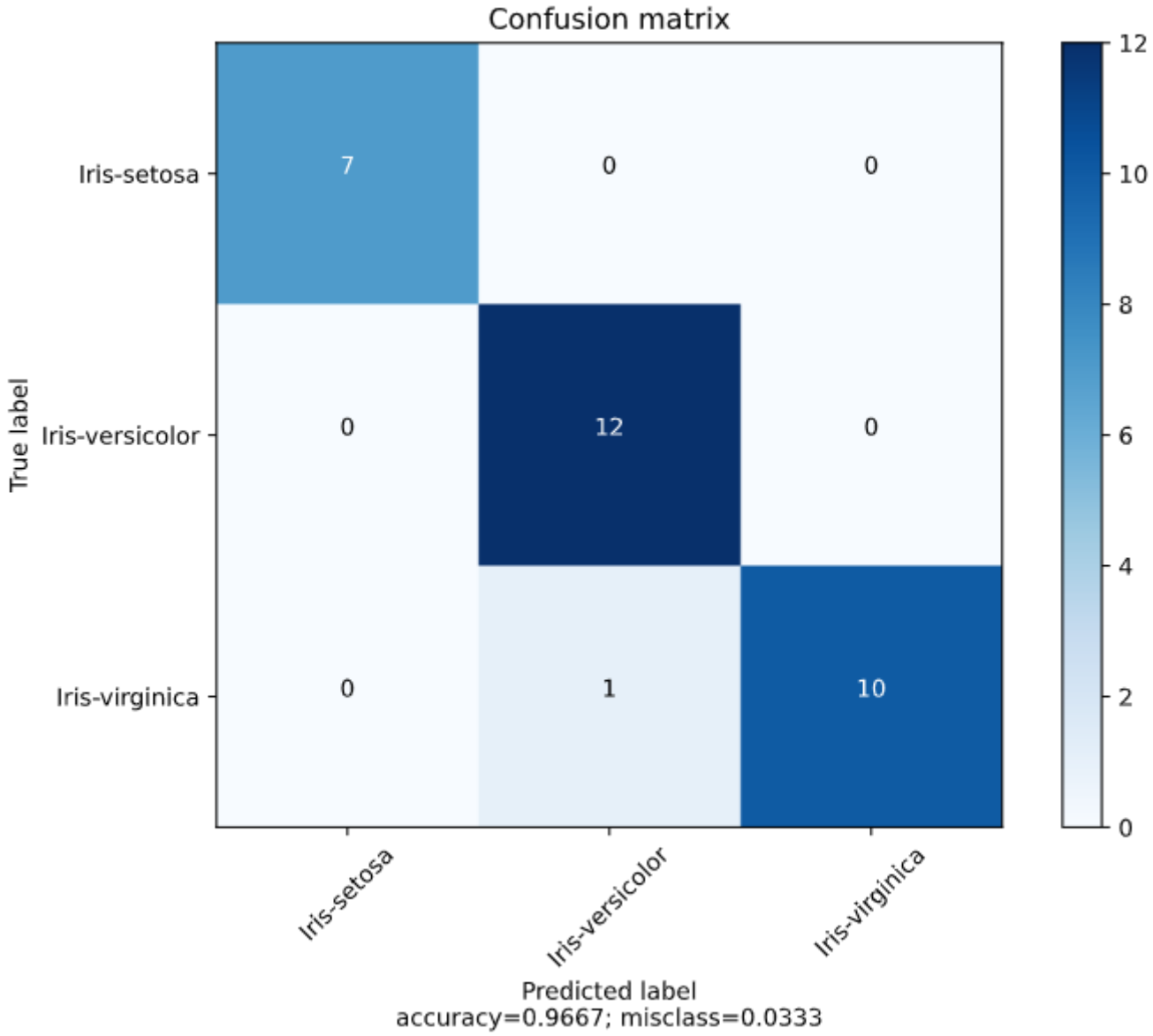
On associe un Chiffre par classe. Par exemple 0, 1, et 2 à chaque classe.

Implémenter dans le langage de votre choix une version des kplus proches voisins

Voir le code sur github.

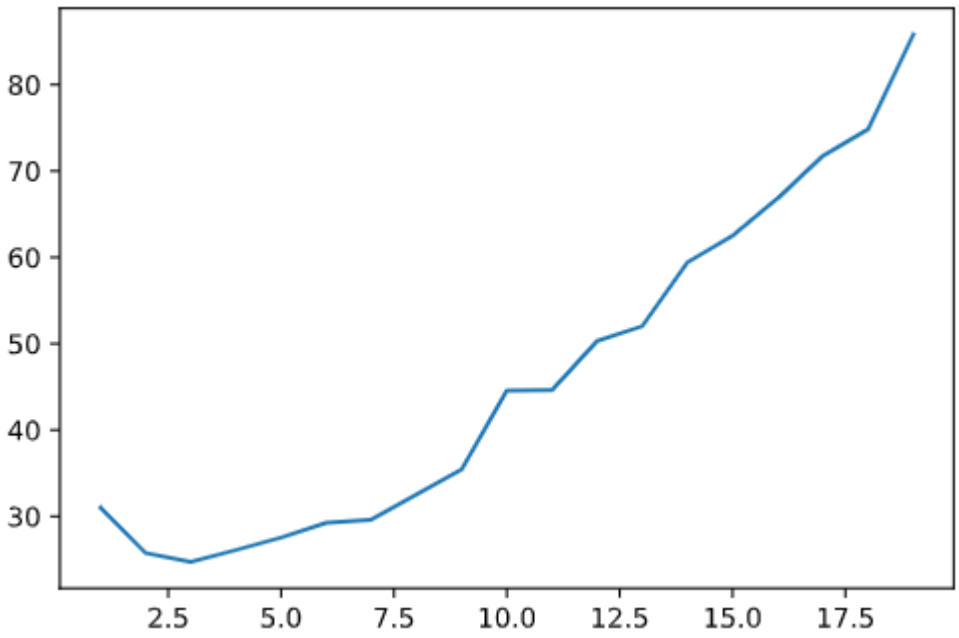
Résultats

Confusion matrix

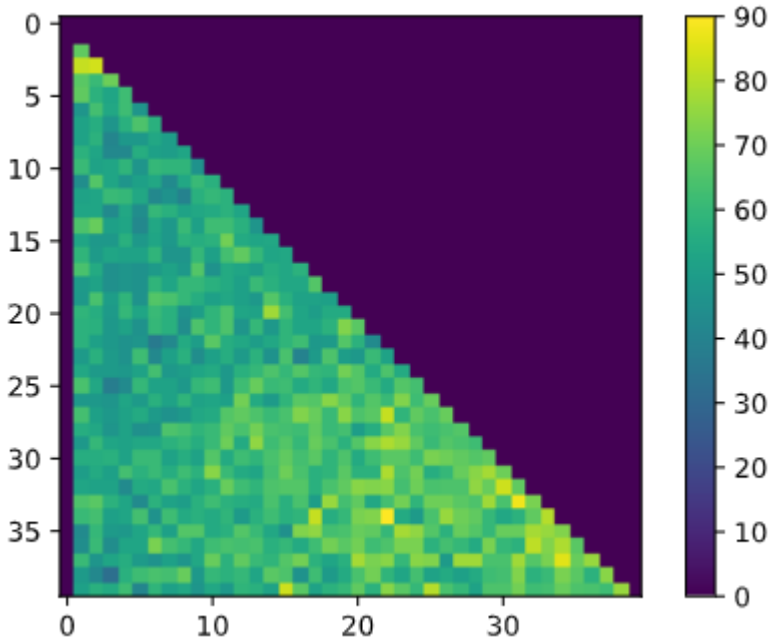


Erreur en fonction de K sur 50 runs par valeurs de K

Pour le choix de k on avait donc 3.



Cependant, avec les nouveaux sets de données, et la méthode de kd-tree, les paramètres les plus optimisés sont k = 6 et leaf_size entre 25 et 35 :



Les 0 (violet) correspondent aux valeurs impossibles ou non testé (k > leaf_size, k = 0, ou leaf_size = 0)

Améliorations

pondération on fonction de la distance

Augmentation dans la précision d'environ 4%.

```
distance = self.__get_distance(point, x_train)
order = np.argsort(distance)
y_train = y_train[order][:self.k_neighbours]
distance = distance[order][:self.k_neighbours]
```

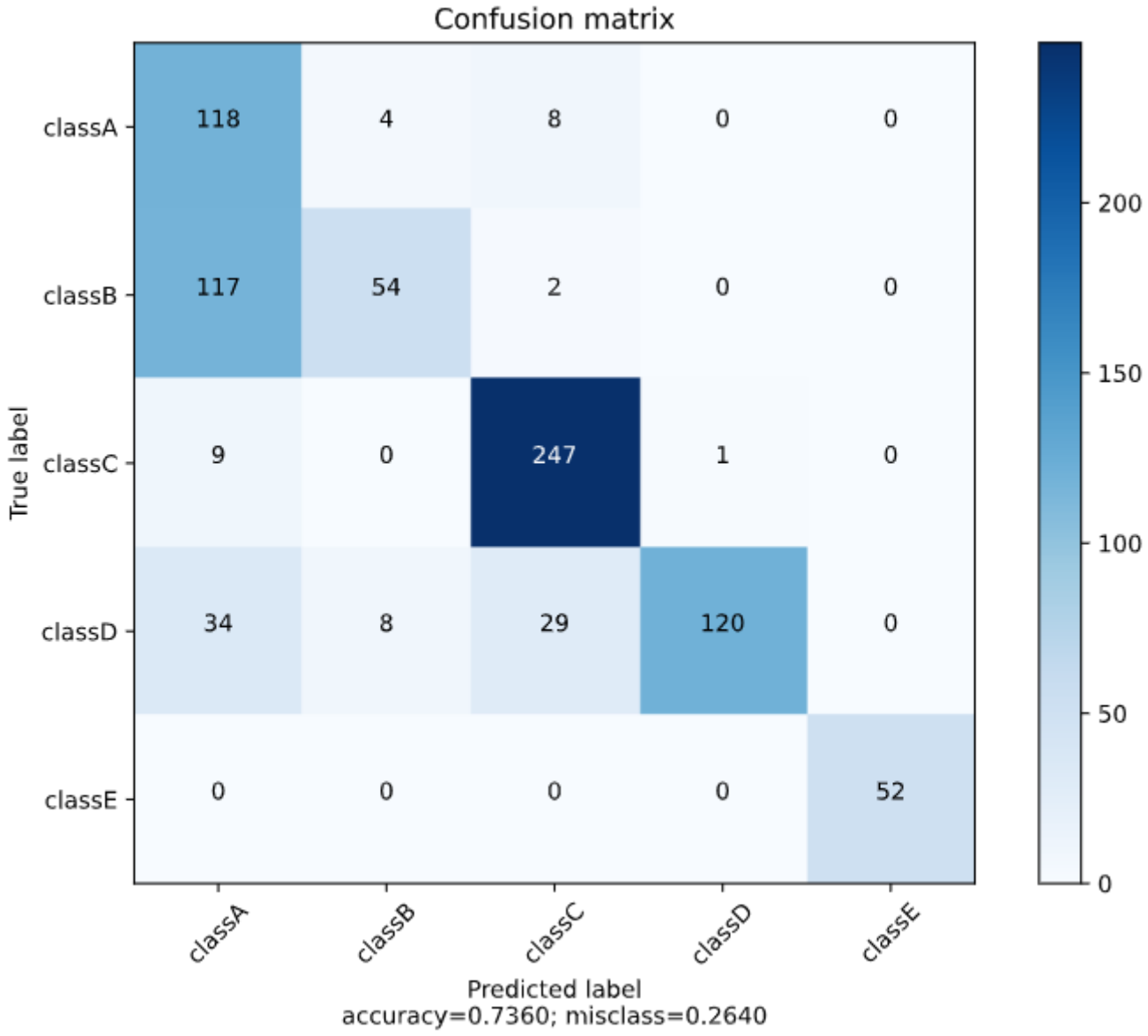
```
# mean value of the k nearest neighbours (same weight)
return np.argmax(np.bincount(y_train))

# Try to give different weights according to distance
class_choice = np.zeros(5) # nombre de classes
for i in range(len(y_train)):
    class_choice[y_train[i]] += 1/distance[i]
return np.argmax(class_choice)
```

Utilistaion de l'algorithme KD-Tree

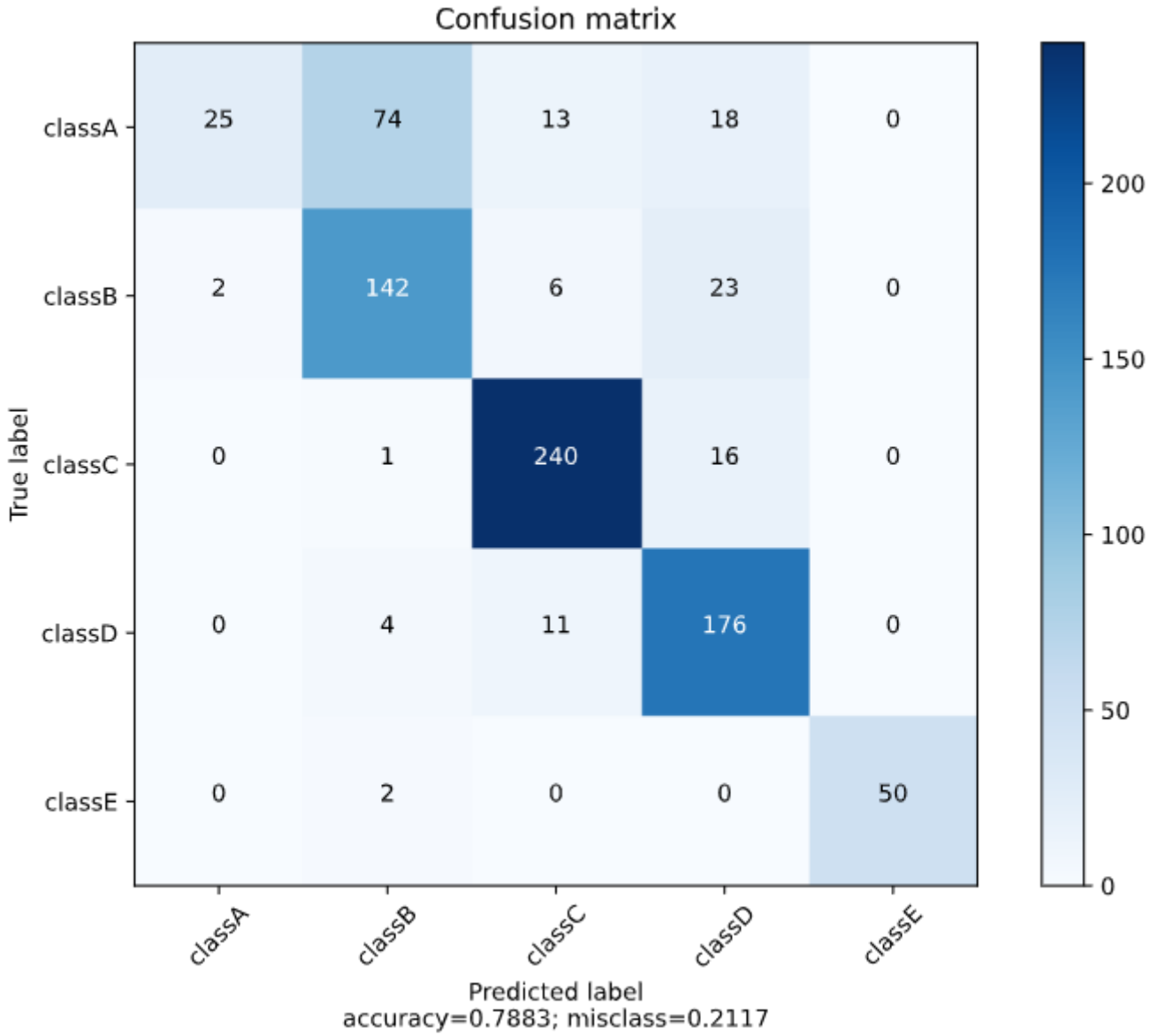
Les tests ont été réalisé avec le test sur les données *data* pour l'entrainement et les données *preTest* pour la validation. Les pourcentages sont donc bas.

Avec la méthode "brute", l'algorithme nécessite environ 5 voisins pour avoir l'erreur minimum. La matrice de confusion donne environ 72% de réussite.



Avec la méthode kd-tree , on a besoin de moins de voisins et a environ 78% de réussite. Cela peut peut être s'expliquer par le fait qu'on ne compare avec tous les voisins mais qu'avec les voisins les "plus proche".

Certains voisins (qui peuvent être les plus proche) ne sont pas utilisés pour la comparaison.



Centrer et réduire les données

Pour moi le fait de centrer et reduire les données n'a pas aidé. Au contraire, j'ai eu des pourcentages à 44% sur des datasets que je n'avais jamais vu, i.e. preTest, malgrés la simplicité de l'algorithme.

```
def reduce(data):
    mean = data.mean(axis=0)
    std = data.std(axis=0)
    return (data - mean) / std
```

Cela peut être du au fait de ne pas avoir des données qui sont centrées autour des mêmes valeurs.

Cependant, le fait d'utiliser l'algorithme kd-tree qui se base sur la variance de chaque feature pour classer les données joue peut-être ce rôle et prend donc en compte la dispersion des données et le problème d'échelle.

Séparation de données

Pour obtenir des valeurs les mieux calibrés possible, on prend un taille de donnée de test de 66% ce qui va représenter à peu près le pourcentage de donnée à tester par rapport aux données d'entraînements.

On obtient alors des données encourageantes :