

Rapport TP3

PHILIPPON Louka SCHNEIDER Adrien

Problématique

On cherche à mettre au point un réseau de neurones capable de classer des images représentant des nombres.

Outils

Pour résoudre ce problème nous avons :

- load_digits dataset de scikit-learn
- matplotlib pour la visualisation
- MLPClassifier de scikit-learn

Apprentissage

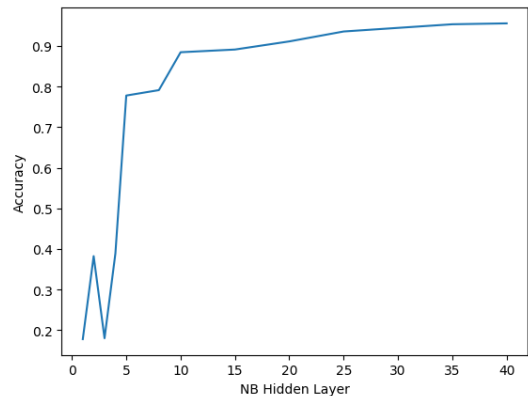
Hidden layers

On cherche à déterminer le meilleur nombre de *hidden layers* nécessaire pour entraîner notre réseau de neurones en un minimum de temps.

```
from sklearn.neural_network import MLPClassifier
import numpy as np

C = np.array([1,2,3,4,5,8,10,15,20,25,30,35,40])
res = []

for e in C:
    clf1 = MLPClassifier(
        hidden_layer_sizes=e,
        activation="tanh",
        solver="sgd",
        batch_size=1,
        alpha=0,
        learning_rate="adaptive",
        verbose=1)
    clf1.fit(X_train,y_train)
    res.append(clf1.score(X_test,y_test))
```



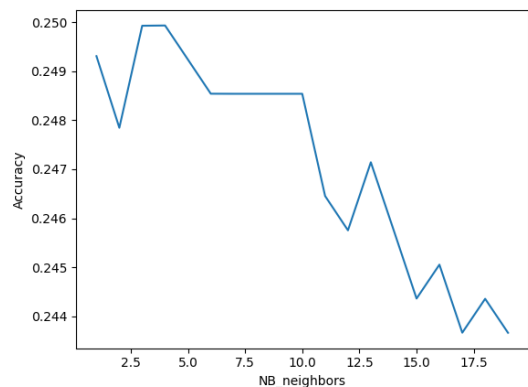
On observe bien qu'on atteint un seuil d'efficacité à partir d'une valeur de 10 *hidden layers*

Neighbors number

De la même manière on veut minimiser le nombre de voisins.

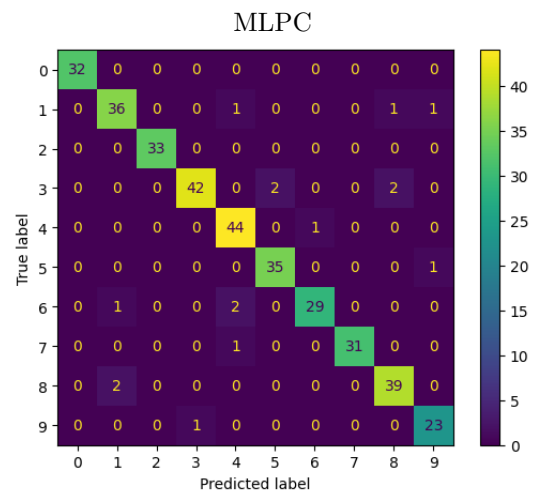
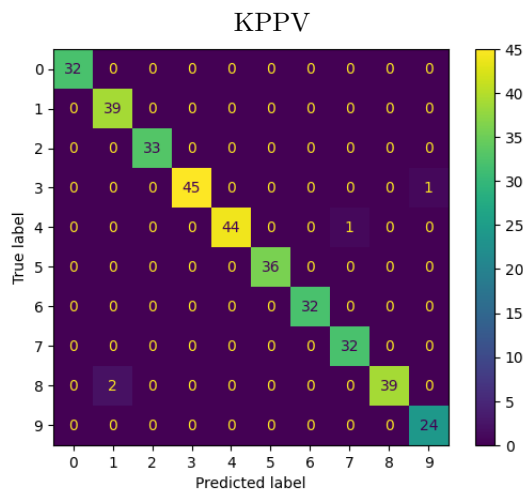
```
from sklearn.neighbors import
KNeighborsClassifier

res = []
val = 0
for e in range(1,20):
    clf2 = KNeighborsClassifier(n_neighbors=e)
    for f in range(10):
        clf2.fit(X_train,y_train)
        for g in range(10):
            val += clf2.score(X_test,y_test)
    val /= 400
    res.append(val)
```



On voit que si l'on met trop de voisins on perd en précision, on choisira donc d'en prendre 3 pour le reste de l'étude

Comparaison



On observe une efficacité similaire entre les 2 algorithmes ce qui peut être surprenant mais cohérent si la base d'entraînement possède des classes bien séparées

Rejet

Voici les fonctions qui permettent de faire le filtre en fonction du seuil

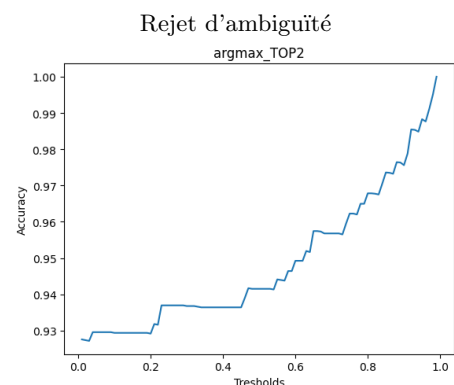
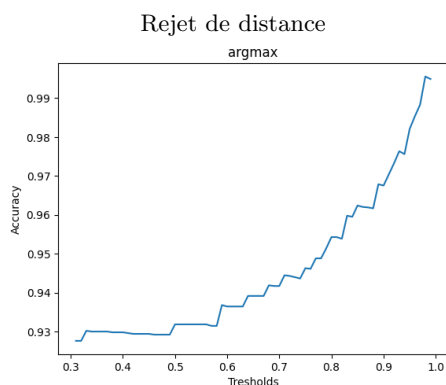
fonction de rejet

```
for _ in range(100):
    tmp = argmax_top2_reject_threshold(y_proba,
    tres)
    if np.unique(tmp, return_counts=True)[0][0]
    == -1:
        rejects.append(tmp)
        rejects_nb.append(np.unique(tmp,
    return_counts=True)[1][0])
        tress.append((tres))
        thresholds.append(tres)
    else:
        rejects_nb.append(0)
        tress.append(tres)
    tres += 0.01
```

fonction de selection

```
for reject in rejects:
    count+=1
    for e in range(len(reject)):
        if reject[e] != -1:
            X_accept.append(X_test[e])
            y_accept.append(y_test[e])
    X_scores.append(clf1.score(X_accept, y_accept
    ))
X_accept = []
y_accept = []
```

On obtient donc ces graphes :



Le rejet d'ambiguïté semble être plus efficace, on le conservera pour la suite de l'étude.

Cascade de classifieurs

On a plus qu'à récupérer les valeurs rejeté et les passer dans l'algorithme de kppv pour voir si l'on peut obtenir de meilleurs résultats

On adapte donc le code de cette façon :

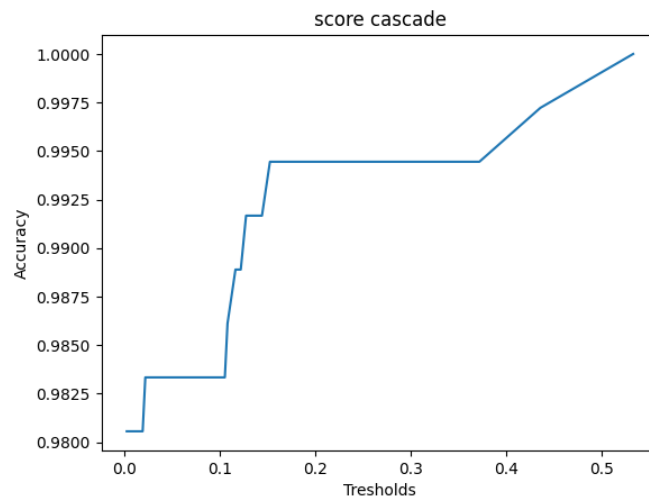
```
for reject in rejects:
    count+=1
    for e in range(len(reject)):
        if reject[e] != -1:
            X_accept.append(X_test[e])
            y_accept.append(y_test[e])
        else:
            X_reject.append(X_test[e])
            y_reject.append(y_test[e])
    X_scores.append(clf1.score(X_accept, y_accept))

y_predict = clf2.predict(X_reject)

X_control = np.concatenate((X_reject, X_accept))
y_control = np.concatenate((y_predict, y_accept))

X_scores_2.append(clf2.score(X_control, y_control))
```

Ce qui nous donne au final ces valeurs de précisions



On observe donc une bien plus grande précision avec un taux de rejet élevé