Rapport TME ML

Master 1 ANDROIDE : Jérôme Arjonilla, Arthur Esquerre-Pourtère

2020

TME 1 - Arbres de décision, sélection de modèles

Entropie

Nous avons codé les fonctions "entropie" et "entropie_cond", elles sont disponibles dans le fichier "decisiontree.py"

Si la différence entre l'entropie et l'entropie conditionelle vaut 0, cela signifie que l'attribut ne permet pas de faire baisser l'entropie du tout, le "gain d'informations" est donc nul.

Pour la première partition, le meilleur attribut est l'attribut "drama" car c'est celui qui a la plus grande différence entre l'entropie et l'entropie conditionnelle, autrement dit c'est l'attribut qui permet de minimiser le plus l'entropie.

Question 1.4 :

On remarque que plus la profondeur de l'arbre est importante, et moins il y a d'exemple à séparer. Ce résultat est logique, en effet, car les questions sont de plus en plus précises, et permettent de moins séparer les données.

Question 1.5 :

Les scores de bonnes classification augmentent au fur et à mesure que la profondeur augmente. C'est normal puisque ce score est obtenu en testant sur les données utilisées pour l'apprentissage. Toutefois si on testait sur d'autres données, ce score n'augmenterait plus et diminuerait même au bout d'une certaine profondeur. Car si la profondeur est trop grande, l'algorithme va apprendre par coeur les données et n'arrivera plus à généraliser.

Question 1.6

Ces scores ne sont pas un indicateur fiable du comportement de l'algorithme, en effet on test notre algorithme sur les données d'apprentissage, ce qu'il ne faut pas faire. En testant de cette manière, on ne peut pas détecter le sur-apprentissage. Pour avoir un meilleur indicateur il faudrait séparer les données de test des données d'apprentissage. Si les données sont en nombre insuffisant, on pourrait utiliser la validation croisée.

Sur et sous apprentissage

Question 1.7

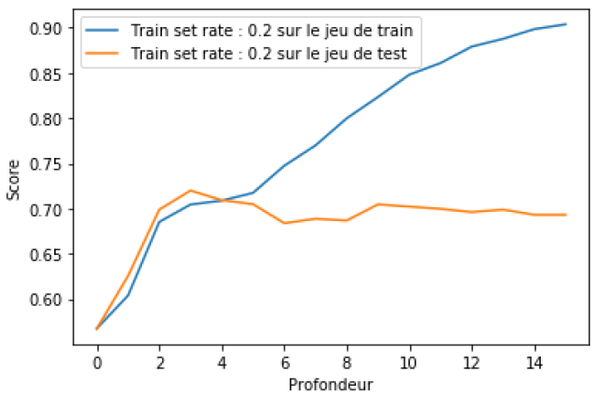

Figure 1: Score sur le jeu de test et le jeu de train avec train_set_rate qui vaut 0.2

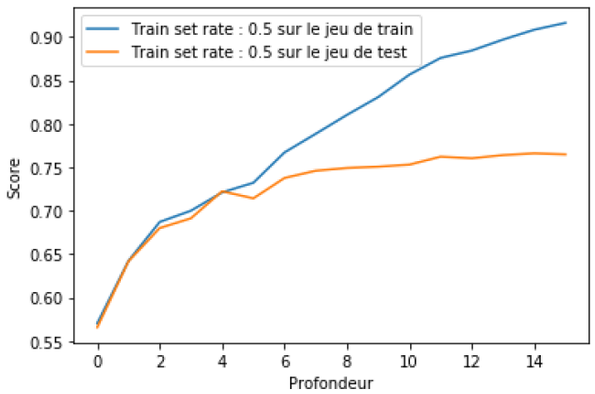

Figure 2: Score sur le jeu de test et le jeu de train avec train_set_rate qui vaut 0.5

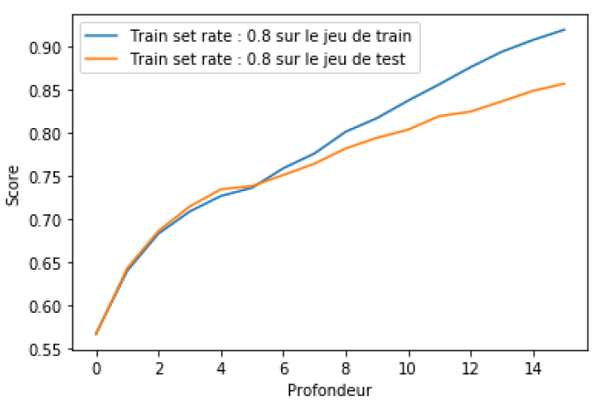

Figure 3: Score sur le jeu de test et le jeu de train avec train_set_rate qui vaut 0.8

Question 1.8

L'algorithme fait beaucoup plus d'erreurs sur le jeu de données de test quand il y a peu d'exemples d'apprentissage (quand train_set_rate vaut 0.2).On voit que le score semble atteindre son maximum à partir de la profondeur 3 et il diminue lentement après.

À contrario, lorsque l'ensemble d'apprentissage est plus grand, sur le jeu de test l'algorithme fait moins d'erreurs et son score augmente lorsqu'on augmente la profondeur.

Par contre sur le jeu de données de train, l'erreur diminue toujours lorsqu'on augmente la profondeur, quelque soit la taille du jeu de données d'apprentissage (train_set_rate), en effet lorsqu'on teste sur le jeu de données de train, on ne peut pas détecter le sur-apprentissage, c'est pourquoi il ne faut jamais évaluer notre modèle sur ce jeu de données.

Question 1.9

Nos résultats ne sont pas très stables et donc pas très fiables, on pourrait les améliorer en utilisant la validation croisée.

Validation croisée : sélection de modèle

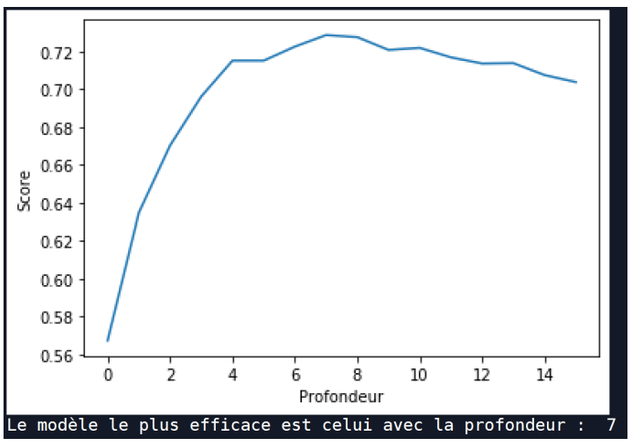

Figure 4: Score sur le jeu de test, en utilisant la validation croisée

En utilisant la validation croisée on trouve que le modèle est le plus efficace lorsqu'on a une profondeur qui vaut 7. La validation croisée nous a permis d'avoir des résultats plus stables et donc plus fiables.

TME 2 - Estimation de densité

Pour ce TME, nous nous sommes intéressés à l'estimation de points d'intérêt (POI) dans Paris. Les résultats présentés ci-dessous correspondent aux points d'intérêt "night_club".

Méthode des histogrammes

Si on discrètise trop (avec un steps trop petit), on obtient de trop grandes "cases" et il y a une risque de sous-apprentissage. Au contraire si on ne discrètise pas assez (un nombre de steps grand), il y a un risque de sur-apprentissage.

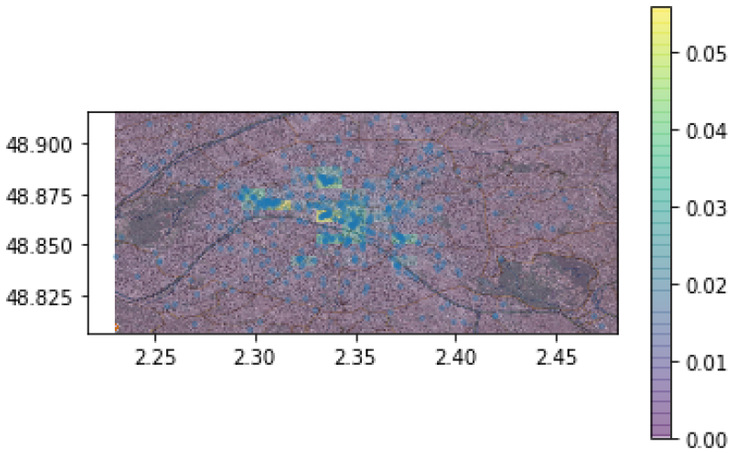

Figure 5: Estimation de densité en utilisant la méthode des histogrammes, steps=20

Méthode à noyaux

Par la méthode à noyaux, nous avons 2 paramètres à chaque fois. Le paramètre n présent à la fois pour Parzen et pour le noyau gaussien sert en fait pour déterminer le nombre de points d'intérêt pour pouvoir les afficher. Il vaut mieux avoir une valeur plus élevée pour que ça soit plus précis, mais attention le nombre de points d'intérêt est n**2, si n est trop grand le temps de calcul augmente beaucoup.

Pour les fenêtres de Parzen, le paramètre hn fait varier la taille de l'hypercube. Si il est trop grand, on sous-apprend et s'il est trop petit il y a un risque de sur-apprentissage,

en particulier si on a peu de données.

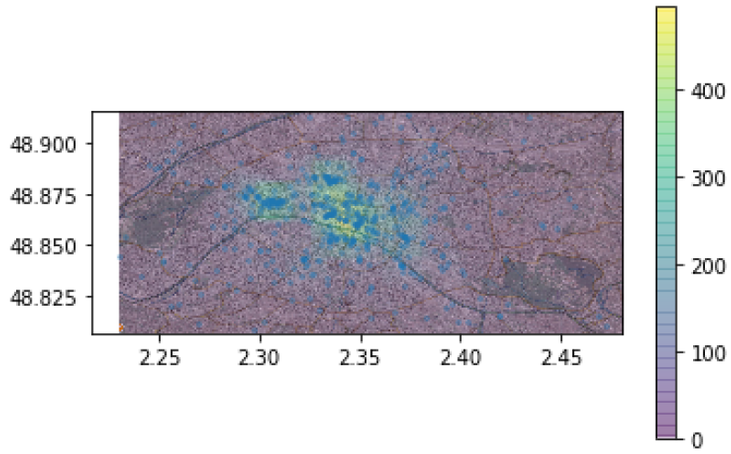

Figure 6: Estimation de densité en utilisant la méthode de fenêtres de Parzen, hn=0.02, n=100

Enfin, pour le noyau gaussien, on utilise le paramètre alpha, si alpha est élevé, on prend plus en compte les points éloignés et on risque de sous-apprendre et si alpha est faible les points éloignés sont peu pris en compte en on risque de sur-apprendre.

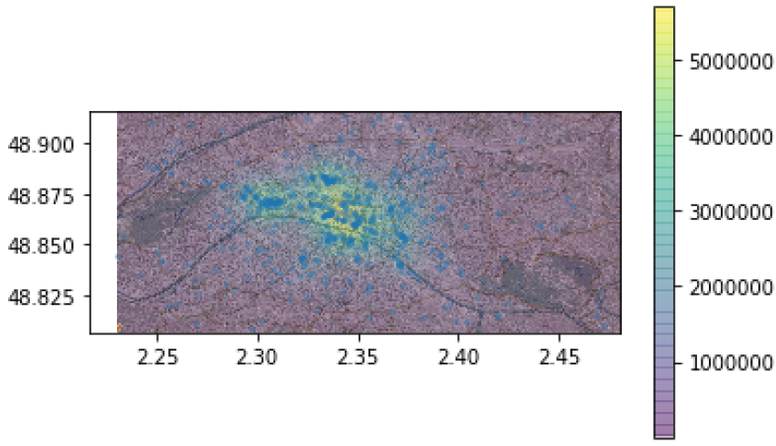

Figure 7: Estimation de densité en utilisant la méthode du noyau gaussien, alpha=0.008, n=100

Évaluation des modèles et choix des hyper-paramètres

Pour la méthode des histogrammes et la méthode des noyaux, il faut choisir minutieusement les valeurs de ces hyper-paramètres pour éviter, ni de sous-apprendre, ni de sur-apprendre. Pour choisir de manière optimale ces hyper-paramètres, on pourrait évaluer notre modèle en calculant la log-vraisemblance des points d'intérêts de notre modèle. Le but serait donc d'utiliser un gridSearch pour maximiser cette log-vraisemblance et améliorer notre modèle.

TME 3 - Descente de gradient, Perceptron

Durant ce TME, la plupart de nos résultats ont été lancé sur 100 runs indépendant, afin d'obtenir des résultats plus précis.

Perceptron

La courbe de l'erreur lorsque notre fonction de coût est "MSE" :

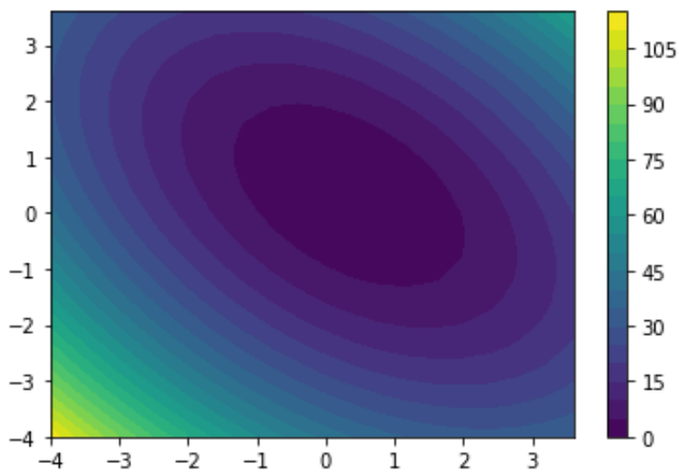

Figure 8: Erreur MSE

La courbe de l'erreur lorsque notre fonction de coût est "hinge" :

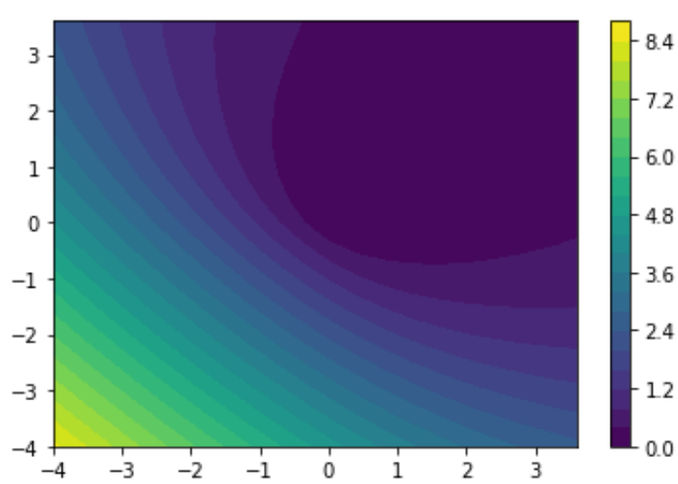

Figure 9: Erreur Hinge

En lançant nos fonctions sur l'exemple sur deux gaussiennes, on obtient :

- Hinge
  - Descente de gradient avec le coût hinge sans biais et un batch de 10
    - Erreur : train 0.118750, test 0.111290
  - Descente de gradient avec le coût hinge sans biais
    - Erreur : train 0.111070, test 0.104610
  - Descente de gradient avec le coût hinge avec biais et un batch de 10
    - Erreur : train 0.120060, test 0.111510
  - Descente de gradient avec le coût hinge avec biais
    - Erreur : train 0.103350, test 0.098720
- MSE
  - Descente de gradient avec le coût MSE sans biais et un batch de 10
    - Erreur : train 0.088930, test 0.082770
  - Descente de gradient avec le coût MSE sans biais
    - Erreur : train 0.087000, test 0.080000

- Descente de gradient avec le coût MSE avec biais et un batch de 10
  - Erreur : train 0.092330, test 0.083560
- Descente de gradient avec le coût MSE avec biais
  - Erreur : train 0.086000, test 0.084000

Grâce aux résultats, on observe que le pourcentage d'erreur est moins important lorsqu'il n'y a pas de batch et lorsque nous ajoutons un biais.

Ces résultat sont logiques, car on étudie sur une plus grande population avec une variable supplémentaire.

Lorsqu'on représente la frontière obtenus sur l'exemple, on obtient

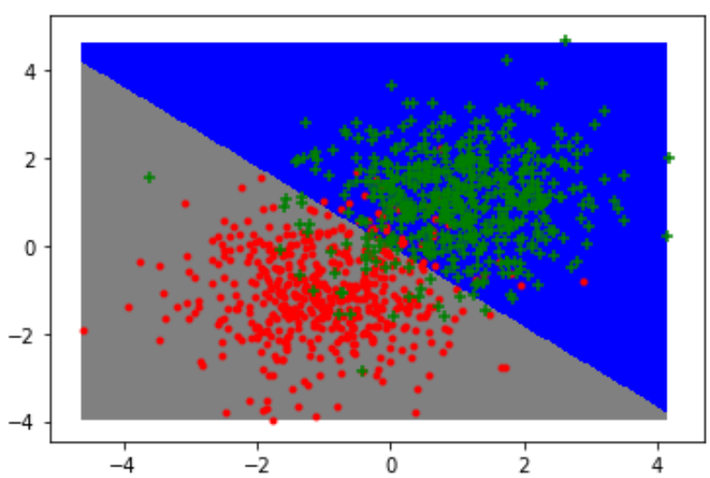

Figure 10: Frontière avec la fonction de coût hinge

Données USPS

Lorsqu'on utilise notre perceptron sans biais sur les données USPS avec 6 vs 9, on obtient :

- Erreur : train 0.000000, test 0.009510

Lorsqu'on utilise notre perceptron sans biais sur les données USPS avec 1 vs 8, on obtient :

- Erreur : train 0.001810, test 0.018837

Lorsqu'on utilise notre perceptron sans biais sur les données USPS avec 6 vs toutes les autres classes, alors en traçant les courbes d'erreurs en apprentissage et en test, on obtient :

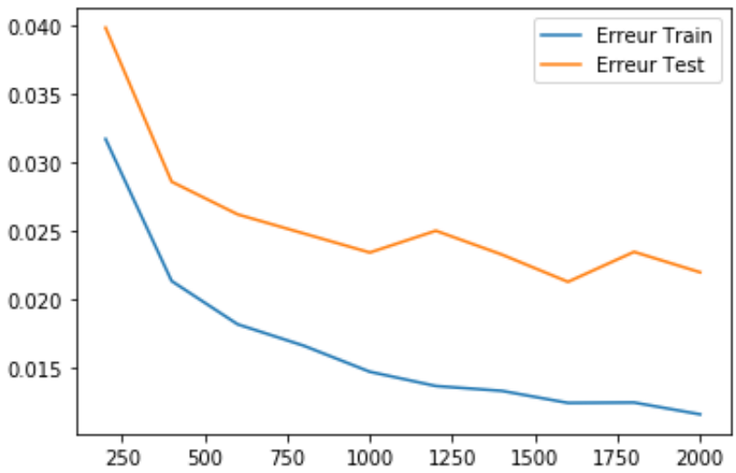

Figure 11: Erreur en test et en apprentissage en fonction du nombre d'itérations

Comme, on peut le voir dans le graphique, on n'observe pas de sur-apprentissage. Ce graphique a été obtenus en lançant 10 runs indépendant au lieu de 100.

De plus, on a le graphique ci dessous qui représente les valeurs des poids lorsqu'on étudie 6 vs toutes les autres classes :

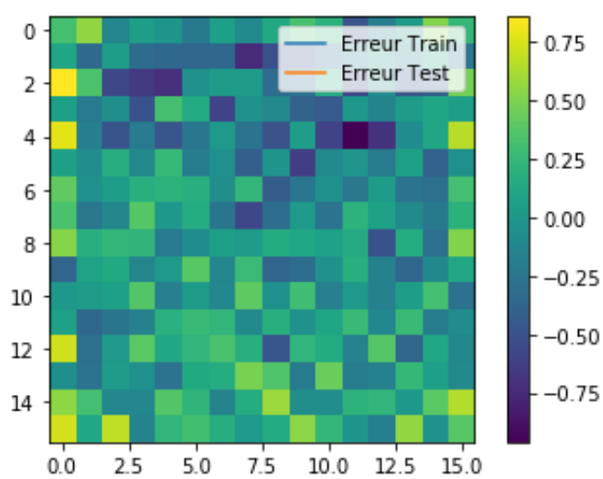

Figure 12: Valeur des poids lorsqu'on étudie 6 vs toutes les autres classes

Données 2D et projection

En testant notre perceptron avec 2 gaussiennes et sans biais, on obtient :

- Erreur : train 0.104410, test 0.11384

En testant notre perceptron avec 4 gaussiennes et sans biais, on obtient :

- Erreur : train 0.500010, test 0.500160

En testant notre perceptron sur l'échequier et sans biais, on obtient :

- Erreur : train 0.496730, test 0.498850

C'est normal qu'on obtienne des mauvais résultats pour l'échequier et pour les 4 gaussiennes car le perceptron est seulement efficace si le problème est linéaire, ce qui n'est pas le cas.

Après avoir effectué une projection polynomiale de la forme

$$\begin{cases} x1 = x1 \\ x2 = x1 * x2 \end{cases}$$

, alors, pour les 4 gaussiennes, on obtient :

- Erreur : train 0.337550, test 0.339100

De plus, on peut voir cette projection sur le graphique ci dessous

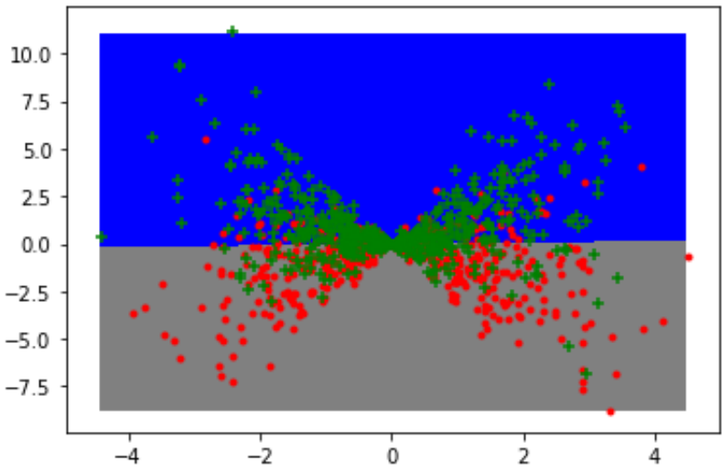

Figure 13: 4 gausiennes après la projection

TME 4 - SVM

Introduction

Dans cette partie, nous allons comparer notre perceptron avec le perceptron dans sklearn.

Lorsque nous utilisons 2 gaussiennes, nous obtenons une valeurs d'erreur en test 0.125000, et en apprentissage de 0.106000.

Lorsque nous utilisons 4 gaussiennes, nous obtenons une valeurs d'erreur en test 0.485000, et en apprentissage de 0.491000.

Lorsque nous utilisons l'échequier, nous obtenons une valeurs d'erreur en test 0.524000, et en apprentissage de 0.500000.

Les valeurs obtenus sont très proches de ceux qu'on avaient obtenus pour le TME3.

SVM et Grid Search

Avec les paramètres de base, on a :

Pour le kernel 'poly' une erreur en test de 0.115000 et en apprentissage de 0.100000, et la frontière suivante :

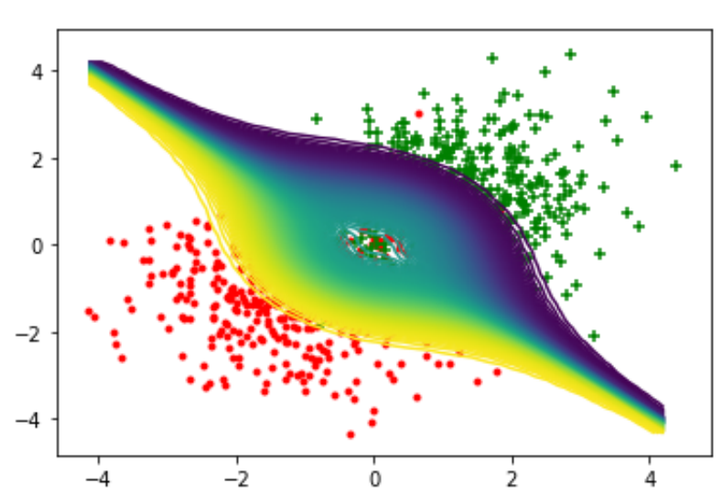

Figure 14: Frontière pour le kernel 'poly' avec les paramètres de base

Pour le kernel 'rnf' une erreur en test de 0.096000 et en apprentissage de 0.097000, et la frontière suivante :

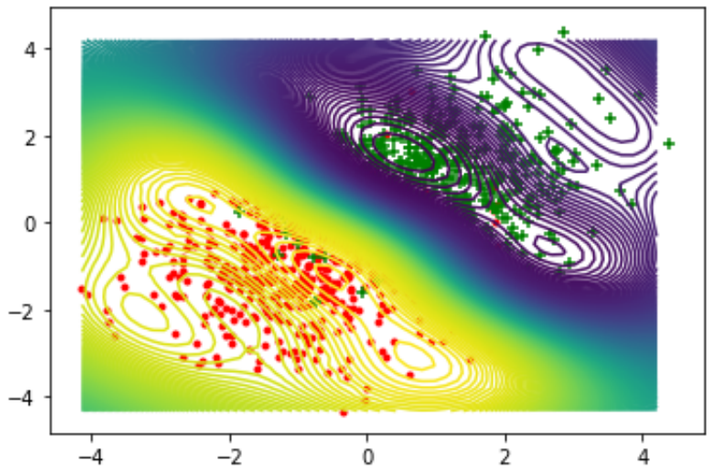

Figure 15: Frontière pour le kernel 'rnf' avec les paramètres de base

Pour le kernel 'sigmoide' une erreur en test de 0.115000 et en apprentissage de 0.129000, et la frontière suivante :

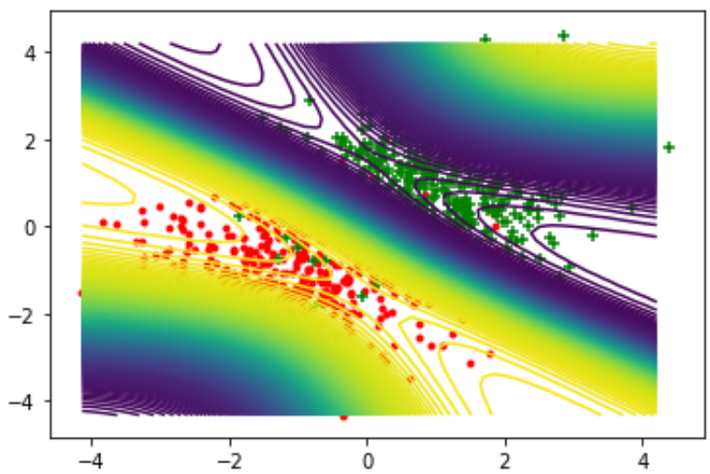

Figure 16: Frontière pour le kernel 'sigmoide' avec les paramètres de base

Pour le kernel 'linear' une erreur en test de 0.093000 et en apprentissage de 0.101000, et la frontière suivante :

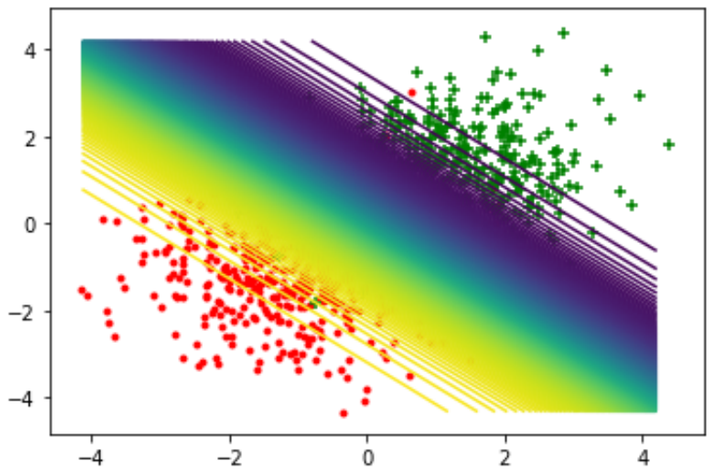

Figure 17: Frontière pour le kernel 'linear' avec les paramètres de base

Pour la suite (Graphique chiffre manuscrite et gaussienne), nous avons effectué un grid Search pour les variables 'C','gamma','kernel' et 'degree' avec comme valeurs :

- Pour la variable 'C' : [0.01, 0.1 , 1 , 10, 100, 1 000, 10 000]
- Pour la variable 'gamma' : [0.00001, 0.0001, 0.001, 0.01, 0.1, 1, 10]
- Pour la variable 'degree' (pour 'poly') : [1, 3, 5, 7, 9]
- Pour la variable 'kernel' : ['poly', 'rbf']

En effectuant un grid search lorsqu'il y a seulement 2 gaussiennes, on obtient comme valeur optimale :

- Kernel 'rnf' : 0.9311
- Kernel 'polynomial' : 0.9311

Pour le kernel rnf, on obtient la gridSearch suivante :

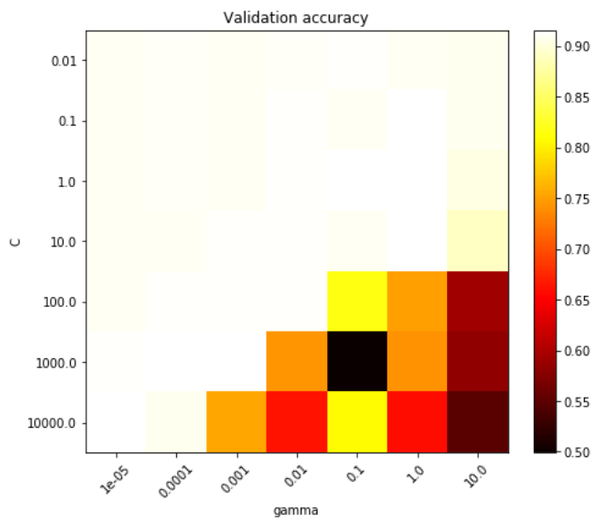

Figure 18: Grid search pour le kernel 'rnf'

Dans le cas polynomial, on a observe que les paramètres choisies forment une fonction linéaire, cela est compréhensible, en effet car dans ce cas ci, le problème est linéaire.

Reconnaissance des chiffres :

En effectuant un grid search pour les lettres manuscrites, on obtient les valeurs optimales (en test) :

- Kernel 'rbf' : 0.9731117095538744
- Kernel 'polynomial' : 0.9655

De plus, on obtient les paramètres optimaux suivants :

- Paramètres pour 'rbf' :
  - 'C' = 100
  - 'gamma' = 0.01
- Paramètres pour 'poly' :
  - 'C' = 0.1
  - 'gamma' = 0.1
  - 'degree' = 3.0

On peut voir dans les graphiques ci dessous l'erreur en fonction du nombre d'exemple :

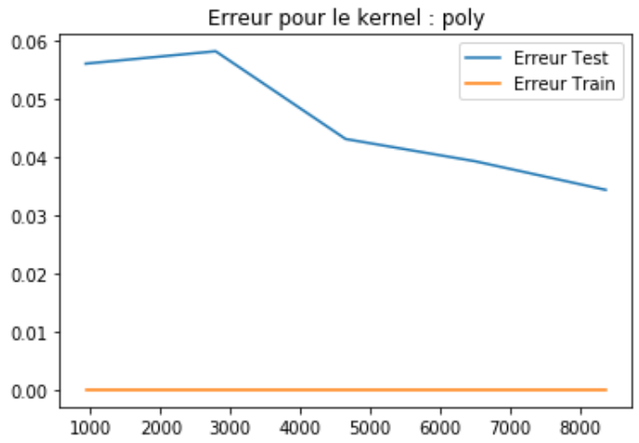

Figure 19: Erreur pour le kernel 'poly'

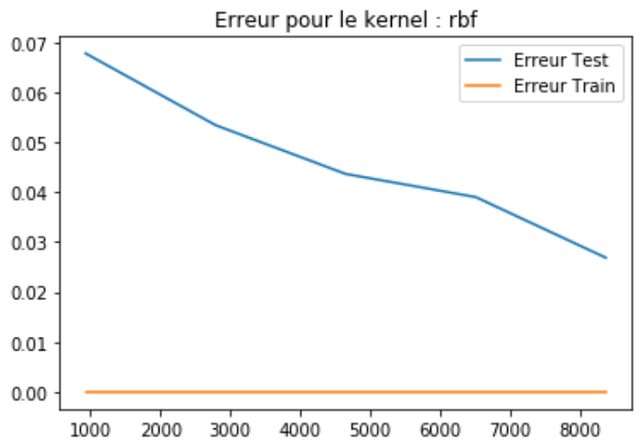

Figure 20: Erreur pour le kernel 'poly'

Comme on peut le voir dans les graphiques, nous n'observons pas de sur apprentissage.

On a aussi la validation accurency suivante pour le kernel 'rbf' :

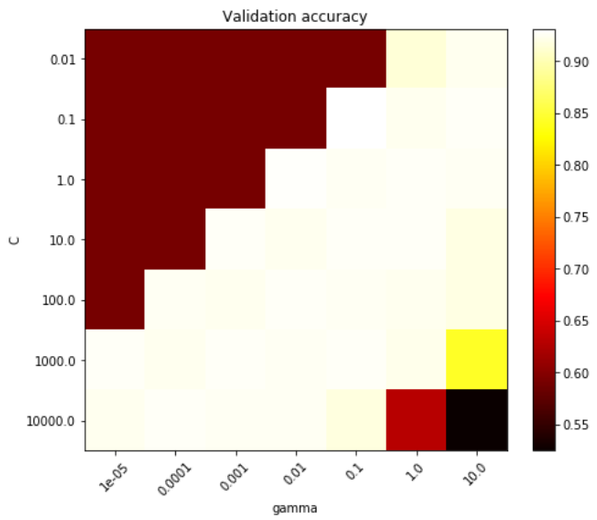

Figure 21: Grid search pour le kernel 'rnf' pour la reconnaissance de chiffre

Apprentissage Multi-Classe

Dans cette partie, on compare 2 méthodes permettent de traiter des cas multi-classes à partir de classifieurs binaires.

Nos résultats sont lancés sur les SVM avec les paramètres optimaux obtenus à la partie précédente.

Dans le cas One-VS-One, on obtient un pourcentage d'erreur de :

- Pour les données d'apprentissage : 0.000137
- Pour les données en test : 0.046836

Dans le cas One-VS-All, on obtient un pourcentage d'erreur de :

- Pour les données d'apprentissage : 0.000137
- Pour les données en test : 0.046836

On observe des résultats identiques, cela est suprenant, on aura pu penser que le cas One-VS-One donnerai des meilleurs résultats, car celui ci compare plus de classe.

String Kernel

Pour cette partie, nous avons dû dans un premier temps, déterminer toutes les combinaisons possibles de lettres de taille n pour chaque mot. Ceci est dans la fonction "determineAllU" de notre code.

Dans un second temps, nous avons dû, déterminer tous les chemins possibles pour un mot s en fonction d'une combinaison de mot u. Ceci est dans la fonction "determineChemin" de notre code.

Pour finir, nous dû créer les fonctions phi et le noyau K afin de pouvoir déterminer de déterminer la similarité. Ceci est dans la fonction "phi", "K".

En effectuant la matrice de similarité pour les mots : logarithm, algorithm, biorhythm, rhythm, competing, computing, on obtient la matrice suivante :

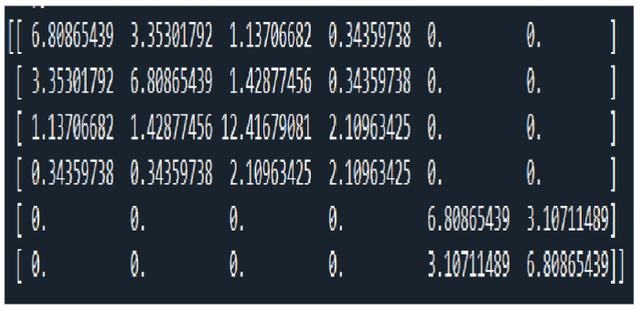

Figure 22: Matrice de similarité

La matrice a été obtenu avec les paramètes n= 4 et lambda =0.8. On peut observer que les mots logarithm et algorithm sont proche, en effet ils possèdent une valeurs de 3.35, et au contraire, ils sont éloignés des mots comme competing et computing.