Faculté des Sciences et Ingénierie

Sorbonne Université

Apprentissage et Reconnaissance de Formes

Rapport des TMEs

Auteur :
Ahmed Tidiane BALDE

Auteur :
Viniya CHANEMOUGAM

15 avril 2018

Table des matières

1 Préambule **2**

2 Arbres de décision, sélection de modèles **3**
2.1 Préliminaires . . . 3
2.2 Sur et sous apprentissage . . . 3
2.3 Validation croisée . . . 4

3 Estimation de densité **5**
3.1 Méthode des histogrammes . . . 5
3.2 Méthode à noyaux . . . 5

4 Descente de gradient **6**
4.1 Optimisation de fonctions . . . 6
4.1.1 $f(x) = xcos(x)$ . . . 6
4.1.2 $-log(x) + x^2$ . . . 7
4.1.3 $f(x_1, x_2) = 100 * (x_2 - x_1^2)^2 + (1 - x_1)^2$ . . . 8
4.2 Régression logistique . . . 9
4.2.1 Expériences . . . 9

5 Perceptron **11**
5.1 Expériences . . . 11
5.1.1 2D Artificial data . . . 11
5.1.2 Type de descente de gradient . . . 11
5.1.3 USPS data . . . 12
5.1.4 2D Artificial data and Projection . . . 14

6 SVM **16**
6.1 Expériences . . . 16
6.1.1 Introduction au module Scikit-learn . . . 16
6.1.2 SVM and GridSearch . . . 18
6.1.3 Apprentissage multi-classe . . . 19
6.1.4 String kernel . . . 19

7 Conclusion **20**

Introduction

1 Préambule

Dans ce rapport, nous aborderons les différents travaux pratiques réalisés dans le cadre de l'UE ARF (Apprentissage et Reconnaissance des formes, certains brièvement et d'autres particulièrement comme les derniers $TMEs$ 3, 4, 5.

Des arbres de décisions aux algorithmes d'apprentissage supervisé comme la regression *linaire*, *logistique*, le *perceptron* et enfin le SVM, utilisant tous cette fameuse technique d'optimisation qu'est la *descente de gradient*, sans oublier l'*estimation de densité*, nous avons appris au courant de cette première partie de l'UE de nombreux modèles.

Dans les sections qui suivent, nous présenterons donc chacun d'eux, les expériences réalisées ainsi que les résultats obtenus dont nous prendrons soin de commenter.

Analyse

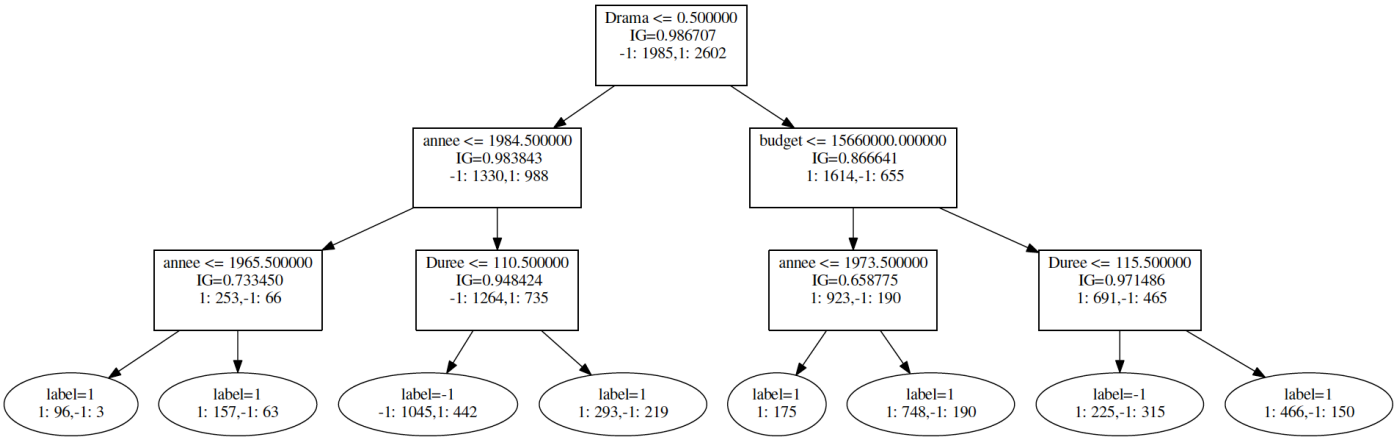
2 Arbres de décision, sélection de modèles

Les arbres de décisions font partie de ces classifieurs puissants capable de classifier tout et n'importe quoi. C'est pourquoi d'ailleurs on dit que ce sont des décideurs universels. L'apprentissage de l'arbre se fait de manière recursive gloutonne $top - down$. A l'aide de l'entropie de *Shannon* dénotant le désordre qu'on a dans un ensemble on choisit le test optimal, dimension et seuillage à chaque noeud.

Premièrement nous implémenterons l'entropie, car nécessaire pour le calcul du partitionnement optimal et ensuite nous réaliserons des expériences sur la profondeur de l'arbre, le sur et sous apprentissage et enfin sur la validation croisée.

2.1 Préliminaires

Apprentissage d'un arbre de décision avec profondeur 3 :



Le nombre d'exemples séparé au niveau de feuilles diminue en fonction de la profondeur de l'arbre et cela est normal car à chaque étape on se concentre sur un sous-ensemble de l'arbre. Quant à l'évolution du score en fonction de la profondeur, nous observons une augmentation. En effet, la profondeur de l'arbre spécialise la classification. Nous devrons donc faire attention à ne pas surprendre car plus la profondeur est élevée plus l'algorithme perd en généralisation.

Profondeur	3	5	10	20	50
Score	0.718	0.736	0.821	0.898	0.900

FIGURE 1 – Evaluation du score en fonction de la profondeur de l'arbre

Rappelons tout de même que ces évaluations ont été réalisées sur les données d'apprentissage, donc les résultats ne sont pas fiables. Afin de palier à ce problème, on divise donc la base en deux sous-ensembles, l'un pour l'apprentissage et l'autre pour le test.
Ainsi, nous pourrons espérer avoir des résultats plus fiables.

2.2 Sur et sous apprentissage

Après avoir mis en oeuvre les différents partitionnements $(0.2, 0.8)$, $(0.5, 0.5)$ et $(0.8, 0.2)$, nous observons les courbes d'erreurs en apprentissage et en test suivantes, par rapport à la profondeur de l'arbre.

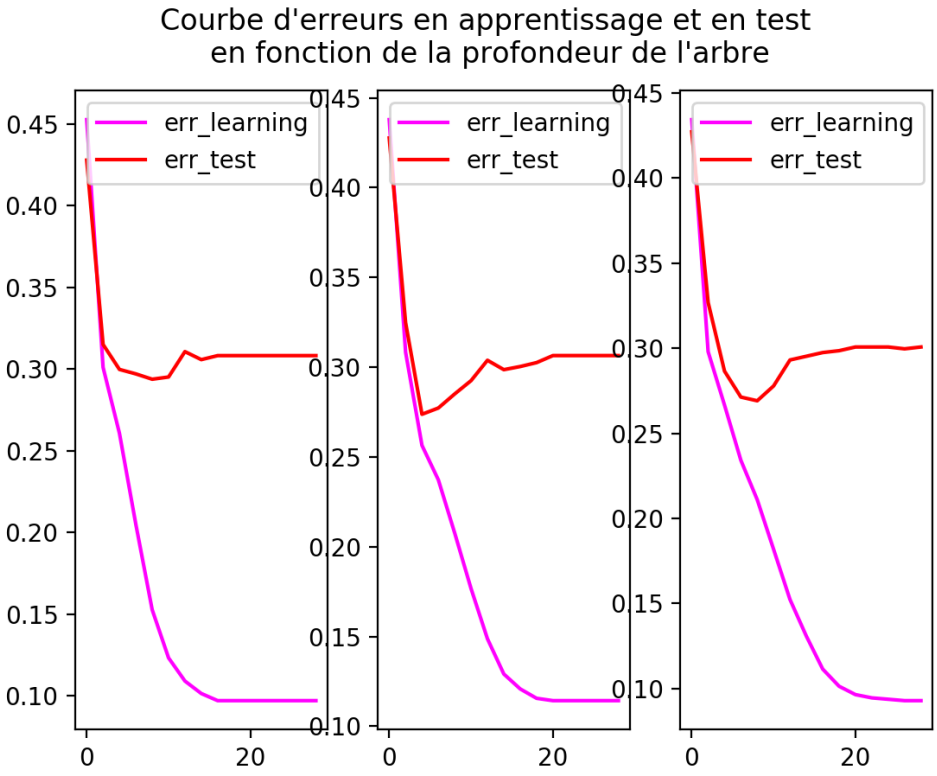


FIGURE 2 – Respectivement les 3 partionnements $0.2vs0.8$, $0.5vs0.5$ et $0.8vs0.2$

Notons l'évolution de la courbe d'erreur en test en fonction de la profondeur. On remarque que cette dernière tend à faire plus d'erreurs lorsque la profondeur est très grande. En effet comme indiqué plus haut, l'algorithme sur-apprend lorsque la profondeur est très élevée.

2.3 Validation croisée

Lorsqu'un ensemble de validation explicite n'est pas disponible, la validation croisée nous permet de tester l'efficacité de notre modèle sur un ensemble de validation hypothétique. On prend l'ensemble des données, on le divise en k partitions, et pendant k itérations, on initialise chacune d'elle comme base de test et le reste comme base d'apprentissage. Ainsi nous aurons des résultats plus fiables et plus stables car nous utilisons la totalité de la base de données à notre disposition.

3 Estimation de densité

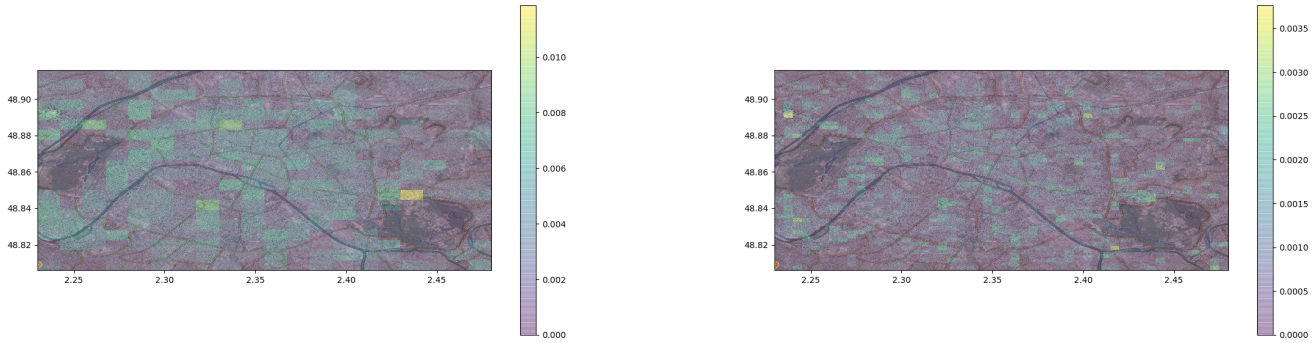
Dans ce TME, nous voulons estimer la loi de densité géographique des différents point d'intérêts (Point of interest) dans la région parisienne. Pour cela, nous utiliserons l'*API* **Google places** contenant un dictionnaire dont chaque clé représente un centre d'intérêt, et la valeur, plusieurs informations telles que les coordonnées, son nom, etc...

Afin d'y parvenir, deux types de méthodes s'offrent à nous :

— la méthode des histogrammes et
— la méthode à noyaux

3.1 Méthode des histogrammes

La méthode des histogrammes, en l'occurrence, consiste tout simplement à la discrétisation des coordonnées et à compter le nombre d'observations présentent dans chaque région, sans oublier de normaliser.



steps=20 steps=50

FIGURE 3 – Estimation de densité par la méthode des histogrammes, POI='restaurant'

On remarque que lorsque le pas de discrétisation est grand, la méthode par histogrammes est très précise. Cependant, lorsque celui-ci est faible, elle tend naturellement à généraliser.

3.2 Méthode à noyaux

La méthode ci-haut par définition produit une estimation discrète, c'est alors qu'intervient la *méthode à noyaux* qui permet de retrouver la continuité. Intéressons nous à la fenêtre de *parzen*. Le concept est simple, considérer une fenêtre autour d'une observation et faire le comptage de toutes les observations présentent dans celle-ci.

Cependant, la taille de la fenêtre joue un rôle important. Si nous considérons une petite fenêtre pour pouvoir faire l'hypothèse que les éléments ont la même densité, on risque au final de n'avoir que peu d'éléments dans la fenêtre. Tout de même quand le nombre d'échantillons tend vers l'infini, nous pouvons nous permettre de prendre une très petite fenêtre de *parzen*.

Une autre approche consiste à l'utilisation d'un noyau, *gaussien*, *laplace*,... Si nous considérons le cas d'un noyau *gaussien*, on remplacerait donc la boîte centrée en x de taille h que nous avions, par une gaussienne centrée en x.

4 Descente de gradient

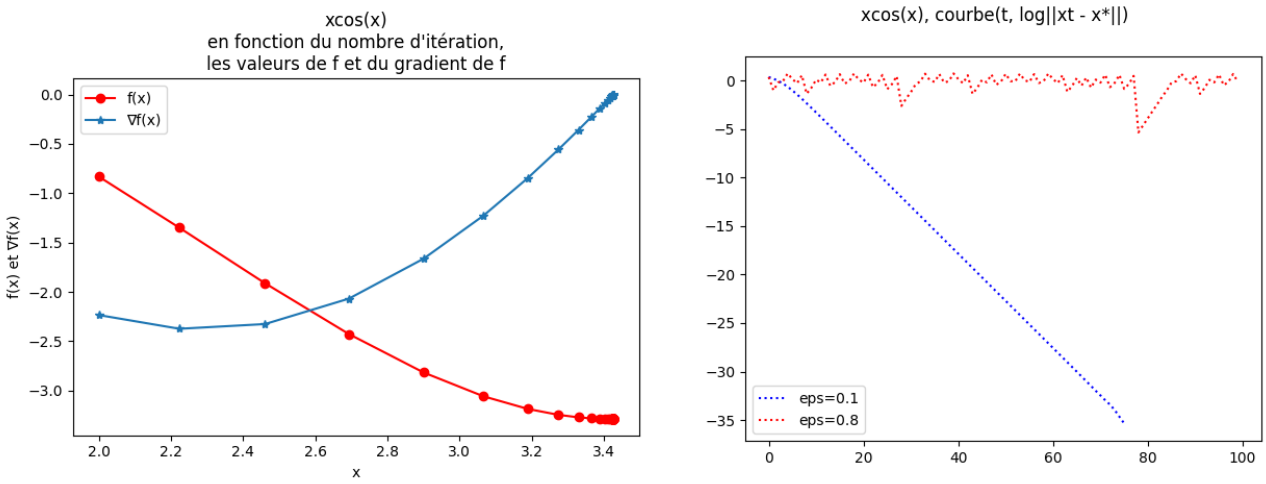
Approcher le minimum d'une fonction convexe est une opération essentielle en machine learning car cela nous permet de réaliser diverses choses telles qu'en l'occurrence minimiser l'erreur lors de la classification. En régression linéaire, cela nous aurait permis de minimiser la différence entre la valeur prédite et la valeur réelle. C'est donc à cette fin qu'intervient la *descente de gradient*.

4.1 Optimisation de fonctions

Nous avons donc implémenté une fonction $optimize(fonc, dfonc, xinit, eps, max\_iter)$ qui à partir d'une fonction, de son gradient, du pas eps du gradient ainsi que la valeur initiale trouve le minimum *local* dans le cas d'une fonction non convexe et *global* dans le cas convexe.
Cette implémentation fut testée sur les différentes fonctions suivantes :

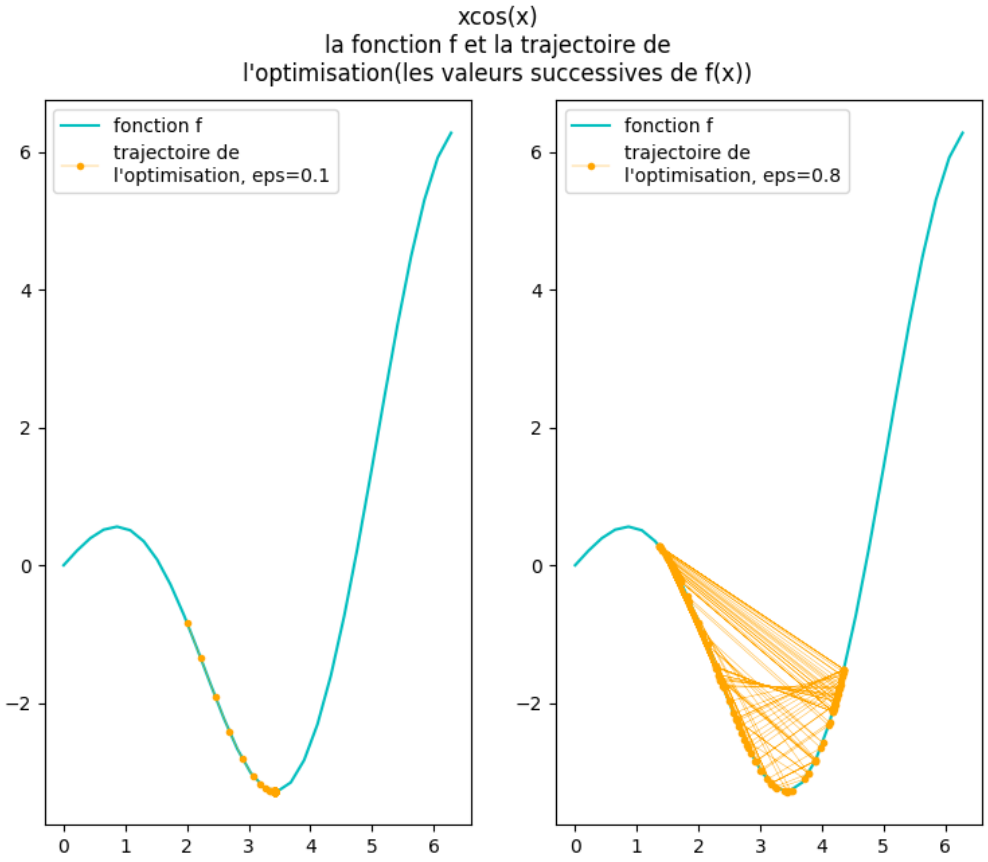
- ⋆ 1d, $f(x) = xcos(x)$
- ⋆ 1d, $-log(x) + x^2$
- ⋆ 2d, la fonction $Rosenbrock(banana)$ définie par : $f(x_1, x_2) = 100 * (x_2 - x_1^2)^2 + (1 - x_1)^2$

4.1.1 $f(x) = xcos(x)$



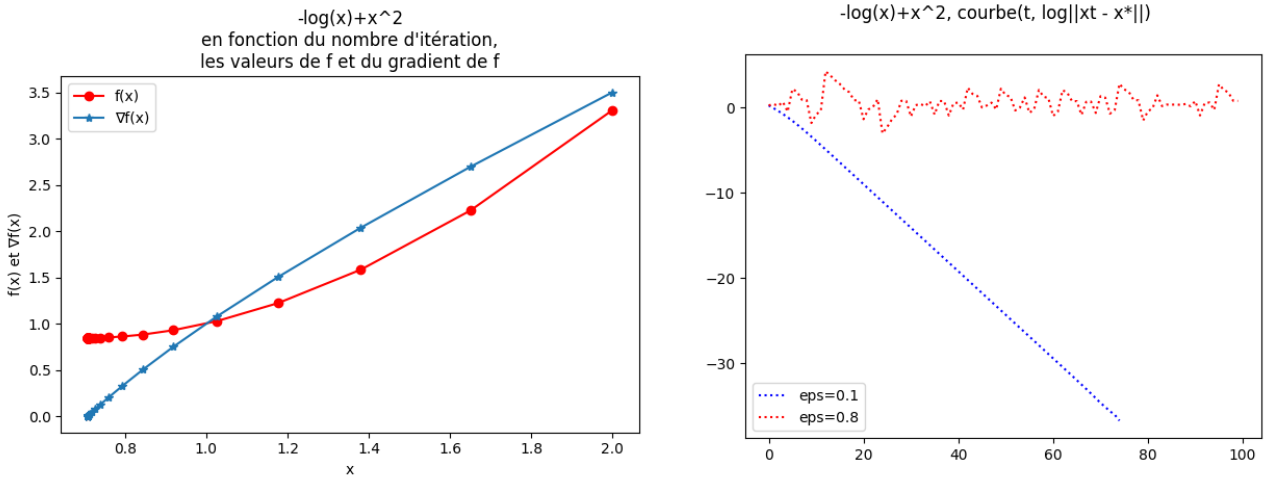
Nous remarquons qu'en fonction du nombre d'itérations, le gradient finit par s'annuler et cela indépendamment du point initial. Et naturellement $(t, log(||x^t - x^*||))$ décroit de façon plus stable quand le pas du gradient $eps = 0.1$ contrairement à quand il est de 0.8.

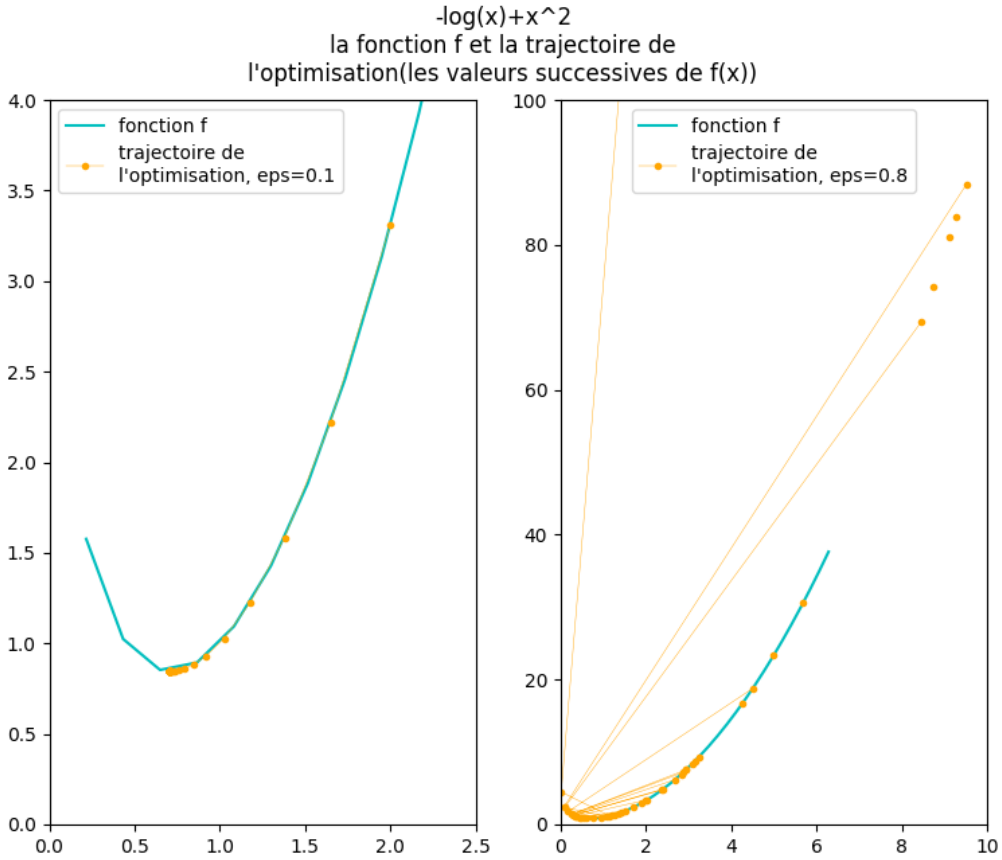
Nous pouvons observer plus clairement sur la figure ci-bas l'effet qu'a le pas du gradient lors de la recherche des valeurs optimales qui minimisent la fonction.
En effet, quand *epsilon* est trop grand, la recherche de minimum devient très difficile car à plusieurs reprises, la trajectoire va dans le sens complètement opposé à celle de l'optimisation .



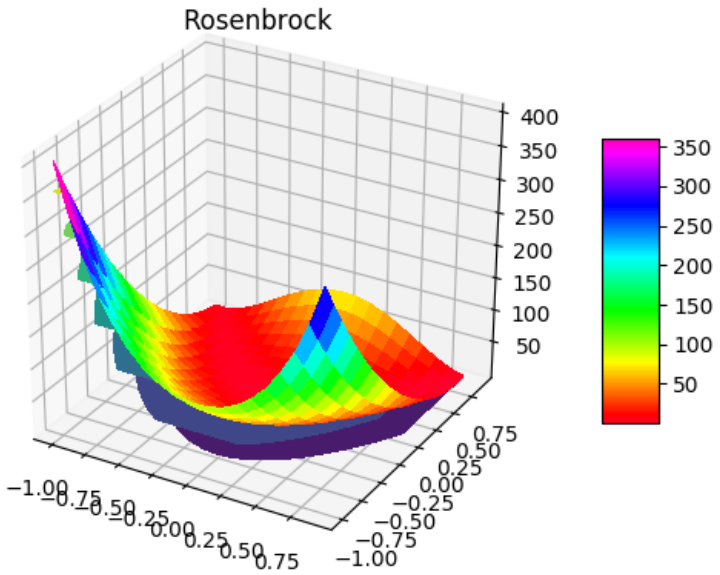
Dans les fonctions suivantes, nous remarquons les mêmes effets, le gradient qui s'annule ainsi qu'une trajectoire vers l'optimisation dont la stabilité et la performance dépend fortement du pas du gradient *epsilon*.

4.1.2 $-log(x) + x^2$





4.1.3 $f(x_1, x_2) = 100 * (x_2 - x_1^2)^2 + (1 - x_1)^2$



4.2 Régression logistique

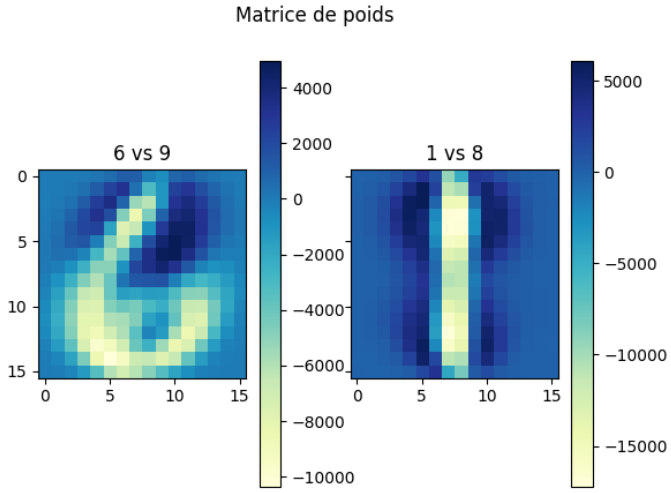
La *régression logistique* est l'un modèles les plus performants en *apprentissage supervisé*. Comme nous l'avons vu auparavant, c'est un faux ami car ce n'est pas une régression mais une classification. Dans le but de l'implémenter, nous avons utilisé la descente de gradient qui met à jour en l'occurrence le paramètre poids w.

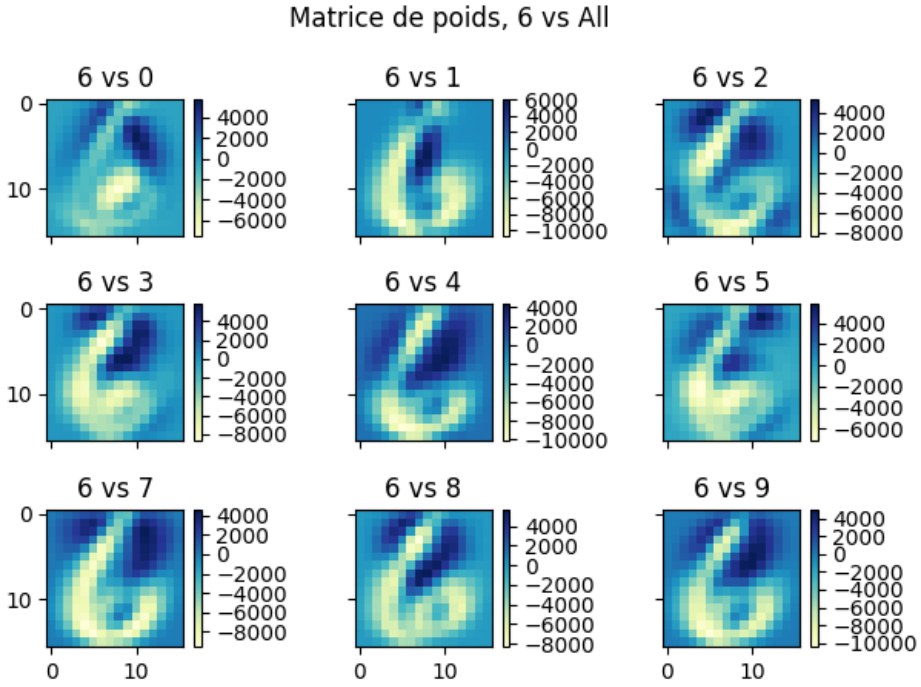
Cet algorithme a été testé sur les données $USPS$(chiffres manuscrits) en ne choisissant que deux classes parmi les 10.

Par la suite, la plupart des algorithmes d'apprentissage supervisé que nous implémenterons serons testés sur ces mêmes données et avec les chiffres $6vs9$ ou $1vs8$ dans un premier temps et dans un second temps 6 contre toutes les autres classes.

4.2.1 Expériences

Ces résultats ci-dessous correspondent à la matrice de poids obtenu lorsque nous appliquons la *régression logistique* sur les données $USPS$.





Error	6vs0	6vs1	6vs2	6vs3	6vs4	6vs5	6vs7	6vs8	6vs9
Train	0.248	0.213	0.378	0.310	0.370	0.404	0.272	0.352	0.293
Test	0.268	0.244	0.413	0.348	0.400	0.442	0.302	0.413	0.299

FIGURE 4 – Evaluation de la régression logistique sur les données USPS

En test, *la régression logistique* fait moins d'erreurs lorsqu'il s'agit de classifier les chiffres 6 et les 1 et plus d'erreurs pour classifier les 6 et 5. Cela nous semble cohérent du fait que le chiffre 5 d'un certain point de vue, pourrait se confondre au 6, il a donc du mal a décidé si certains pixels allumés sont des 6 ou des 5.

Ci-dessous, comparons les résultats de la *régression logistique* avec ceux du modèle *Naïve Bayes*.

Error	6vs0	6vs1	6vs2	6vs3	6vs4	6vs5	6vs7	6vs8	6vs9
Train	0.035	0.014	0.044	0.016	0.035	0.147	0.0007	0.019	0.004
Test	0.058	0.036	0.059	0.041	0.048	0.163	0.006	0.029	0.002

FIGURE 5 – Evaluation de la classifieur Bayésien naïf sur les données USPS

Comme la *régression logistique*, le modèle *naïve Bayes* fait beaucoup plus d'erreurs en classifiant le $6vs5$, par contre bien moins d'erreurs avec $6vs9$.

Notons surtout l'efficacité de ce modèle par rapport à la *régression logistique*. Il a un score de plus de 99% en général.

5 Perceptron

Dans ce TME, nous mettrons en oeuvre le *perceptron*, un algorithme d'*apprentissage supervisé* de classifieurs binaires. On peut aussi le considérer comme étant le réseaux de neurones le plus simple avec une seul couche. Il permet dans sa version simplifiée de trouver la séparatrice de tout problème linéairement séparable. La fonction de coût la plus adapté est le *Hinge loss* qui consiste à optimiser le poids courant seulement si la valeur prédite est différente de la valeur réelle, en d'autres termes seulement s'il y'a erreur.

Par la suite, nous remarquerons que la fonction coût *Mean Square Error* (*mse*) est moins adaptée car elle tend à pénaliser certains exemples bien classés mais tout simplement éloignés de la droite séparatrice. Ce qui est naturellement normal car la fonction *mse* appliquée au perceptron, représente une régression linéaire.

Tout comme nous l'avons vu pour la régression logistique, le *perceptron* procède par *descente de gradient* afin de trouver le paramètre w optimal.

5.1 Expériences

5.1.1 2D Artificial data

Sur les différentes donnée de type 0, 1, 2, avec un peu de bruit, $eps = 0.3$ (comme dans la plupart des expériences que nous mènerons), on a :

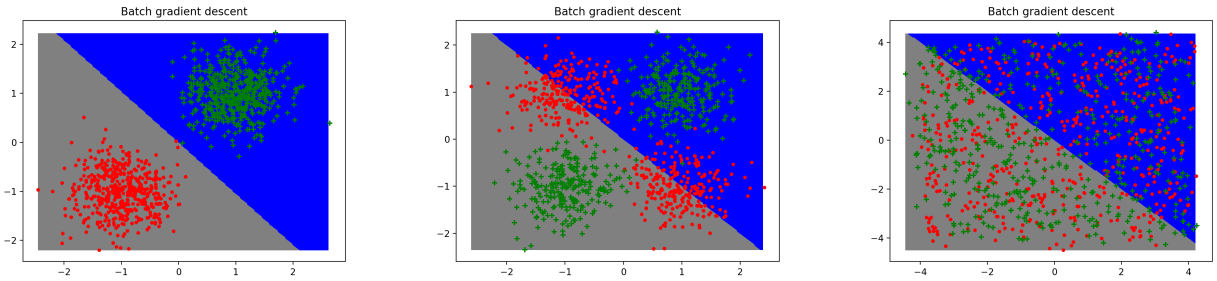


FIGURE 6 – Avec Hinge loss

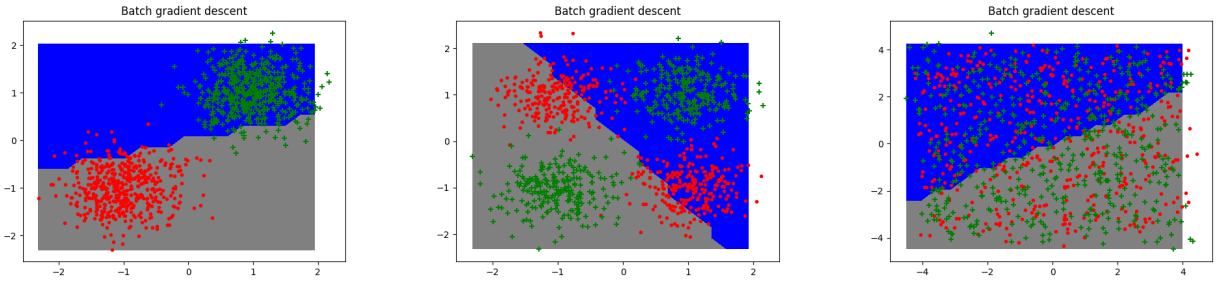


FIGURE 7 – Avec Mean Square Error

Plus particulièrement sur les données linéairement séparable, nous remarquons que le perceptron est plus performant contrairement à la régression linéaire. Ce qui est normal, car la régression linéaire n'est pas faite pour de la classification. On peut d'ailleurs remarquer qu'elle cherche la droite la plus proche des points.

5.1.2 Type de descente de gradient

Dans les expériences précédentes, nous avons utilisé que la descente de gradient de type *bash*. Cependant quand nous avons des données gigantesques, on peut attendre très longtemps afin de voir les résultats étant donnée que

l'on utilise directement toute la base.

C'est alors que nous introduisons la descente de gradient *stochastique* qui, quant à elle, itère sur un seul exemple à chaque itération. Ainsi, nous pouvons voir l'évolution de notre algorithme au fur et à mesure des itérations

Nous avons vu que le premier type *bash* est plus stable, plus rapide mais itère sur toutes les données à la fois. Le second à une meilleure tolérance au bruit mais plus lent car itère sur un seul exemple à la fois. La descente de gradient *hybride* ou encore $mini - bash$ combine les deux derniers types avec un paramètre *bash* étant le nombre d'exemples à considérer pour chaque itération. Quand *bash* est égal au nombre d'exemples du dataset, on se ramène à une descente de gradient *bash* et quand *bash* égal à 1 cela correspondrait à la descente de gradient *stochastique*.

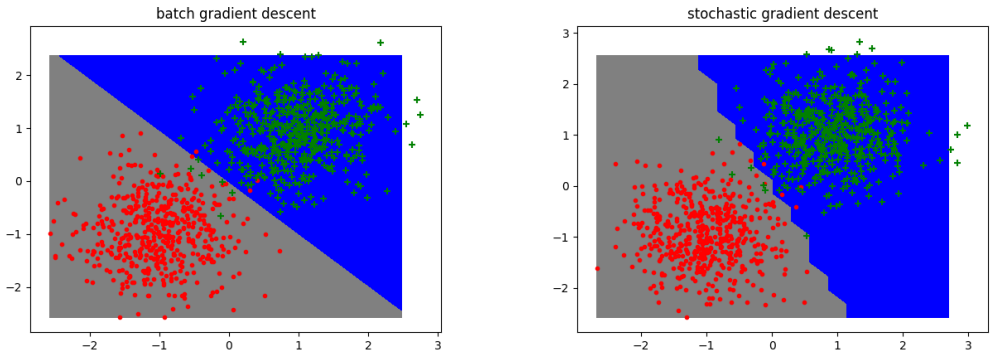
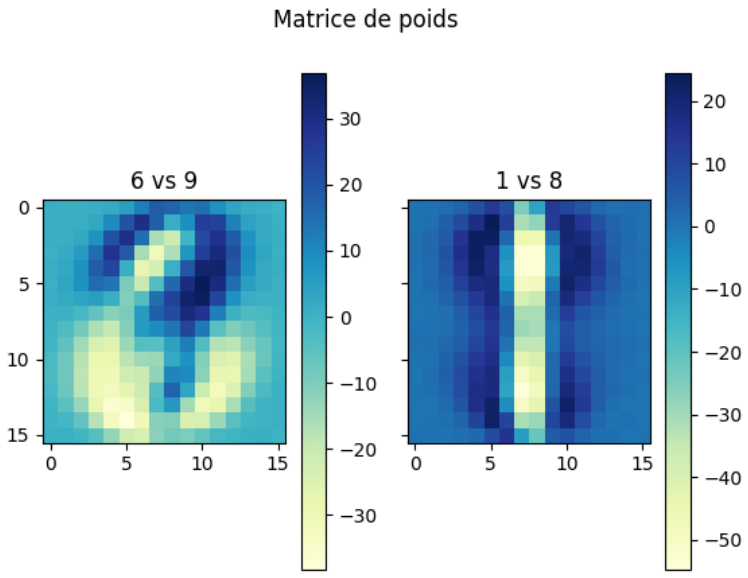


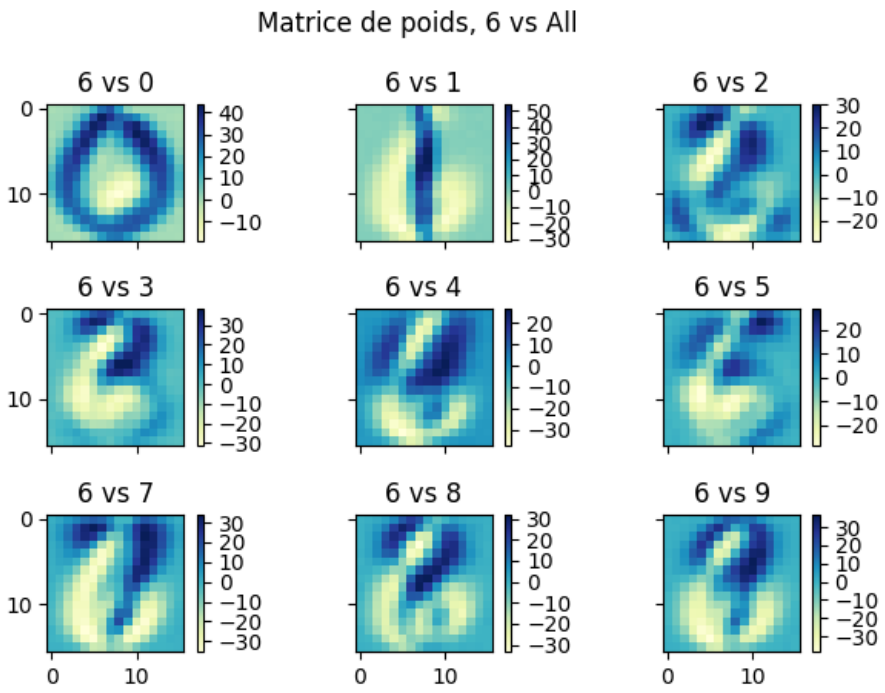
FIGURE 8 – Avec du bruit, $eps = 0.5$

5.1.3 USPS data

Dans un premier temps, nous avons effectué cette expérience en prenant deux classes : $6vs9$ ainsi que $1vs8$.



Dans un second temps, en classifiant la classe 6 avec toutes les autres classes.



Error	6vs0	6vs1	6vs2	6vs3	6vs4	6vs5	6vs7	6vs8	6vs9
Train	0.35	0.047	0.059	0.015	0.072	0.159	0.006	0.038	0.006
Test	0.321	0.064	0.084	0.038	0.113	0.178	0.015	0.077	0.008

FIGURE 9 – Evaluation du perceptron implémenté sur les données USPS

On remarque que pour certains cas comme $6vs7$, il arrive bien à distinguer les deux chiffres, ainsi que pour le $6vs9$ ou il fait le moins d'erreurs. Contrairement au cas $6vs0$ ou il confond la plupart des pixels du 6 à ceux du 0 car ces deux chiffres se ressemblent.

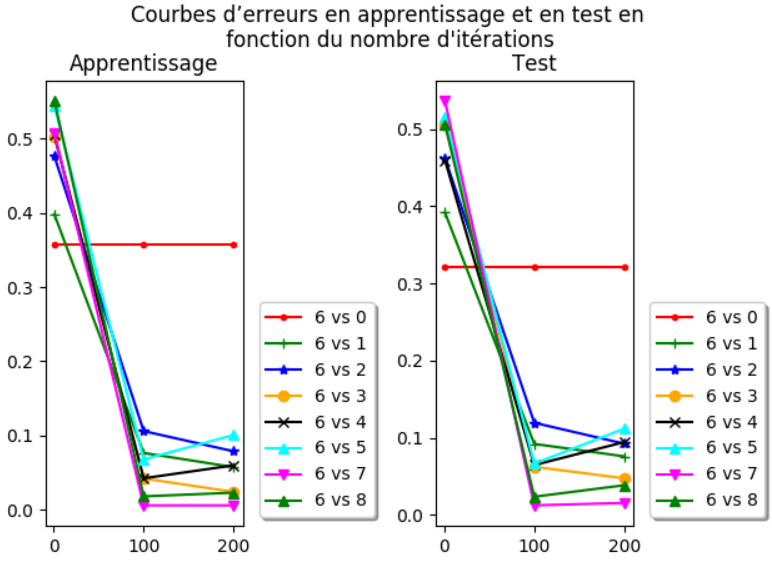


FIGURE 10 – Implemented perceptron

En affichant la courbe d'erreurs, on peut confirmer que notre *perceptron* se comporte de manière normale. Notons le cas $6vs0$ où il faisait beaucoup plus d'erreurs. En effet, la courbe d'erreur reste stable en fonction du nombre d'itérations.

5.1.4 2D Artificial data and Projection

Ci-haut, nous avons vu que le *percpetron* ne classifie que des données linéairement séparable. Pour palier à ce problème, nous implémenterons la *projection* aussi appelé *kerneltrick* qui consiste juste à projeter les données en entrée souvent en $2D$ en données de plus grande dimensions.

Ainsi nous pourrons classifier des problèmes comme le XOR.

Polynomial Kernel : Comme vu dans l'exemple de TD, pour des données en entrée en $2D$, nous pourrons décider d'avoir une sortie sous la forme suivante : $[1, x_1, x_2, x_1^2, x_2^2, x_1x_2]$, donc une sortie en $6D$.

En appliquant ce *kernel* à nos données 2D, nous observons les sorties ci-dessous.

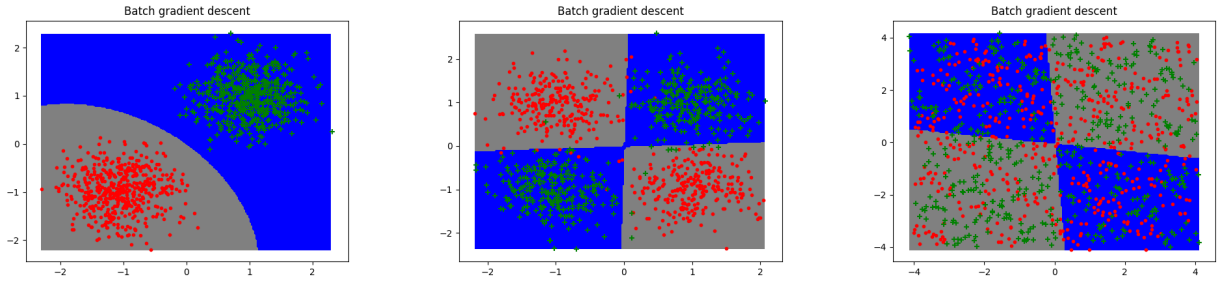


FIGURE 11 – Perceptron and polynomial Kernel

Comme attendu, il arrive bien à classifier les 2 gaussiennes et le problème du XOR.

Gaussian Kernel : Avec le *kernel gaussien*, on peut avoir après projection, un nombre de dimension égal au nombre d'exemples dans la base. Si 1000 exemples, on peut avoir jusqu'à 1000 dimensions.

Après implémentation et expérimentations, nous obtenons :

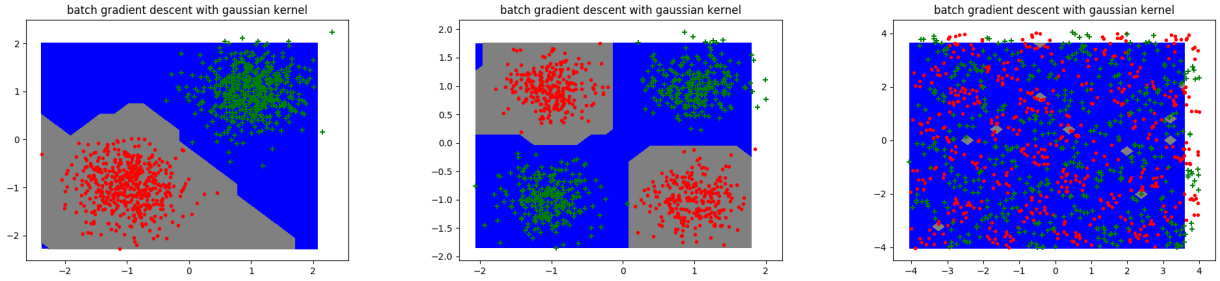
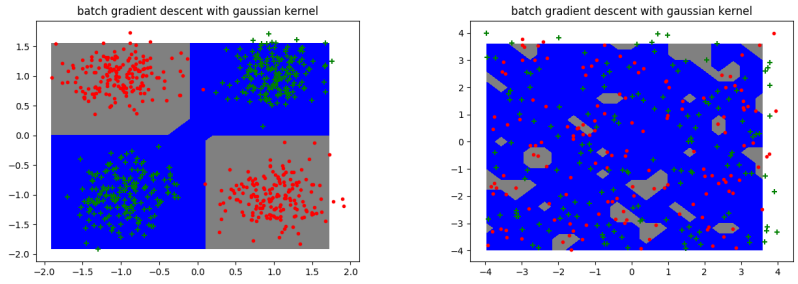


FIGURE 12 – Perceptron and gaussian kernel with 1000 exemples in the dataset

Le *gaussian kernel* s'en sort très bien dans la classification des deux gaussiennes et du XOR mais sur l'échiquier, il fait presque 100% d'erreurs. Nous ne nous attendions pas à cela.



(a) 700 exemples (b) 300 exemples

FIGURE 13 – Perceptron and gaussian kernel avec le nombre de points diminué

Après avoir diminué le nombre de points pour la base de projection, on remarque qu'il performe un petit peu mieux sur l'échiquier. Néanmoins, nous nous attendions à ce qu'il le fasse très bien. Cela peut être du à une erreur dans notre implémentation. Nous pouvons tout de même conclure après cette expérience qu'il vaut mieux peu de données pour le *gaussien kernel* étant donné le nombre d'exemples et de dimensions. Le calcul devient vite très coûteux. C'est l'un des inconvénients du modèle le plus connu.

Enfin, pour palier à ce problème, notons que plusieurs travaux ont été réalisés afin de rendre le modèle *gaussien* plus performant sur des larges datasets.

6 SVM

Le SVM (Support Vector Machines) est aussi un modèle linéaire comme le *perceptron* mais plus puissant. Il a d'ailleurs été prouvé que pour des données linéairement séparables, il est celui qui arbore de meilleurs résultats. Le but de cet algorithme étant de trouver la frontière optimale qui maximisent la distance entre les points d'entraînements.

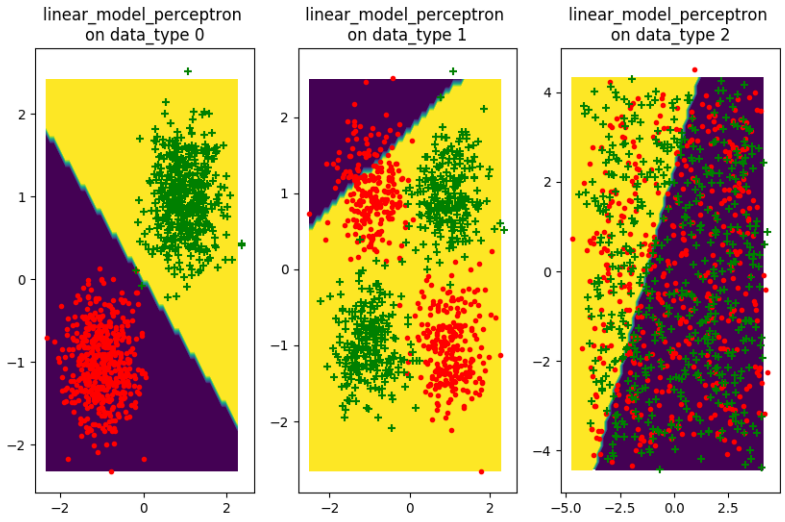
Tout au long de ce TME, nous utiliserons les outils du module $scikit - learn$ et nous comparerons les résultats obtenus aux classifieurs que nous avons implémenté dans les $TMEs$ précédents.

6.1 Expériences

6.1.1 Introduction au module Scikit-learn

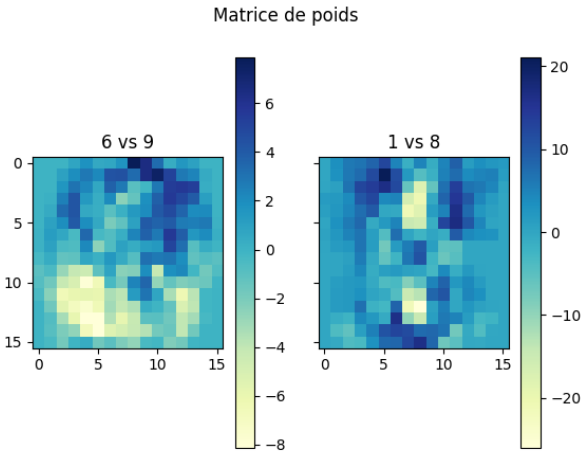
Pour nous familiariser avec le module du *scikit-learn*, nous avons commencé par tester le sous module $scklearn.linear\_model$ qui contient une implémentation du *perceptron*

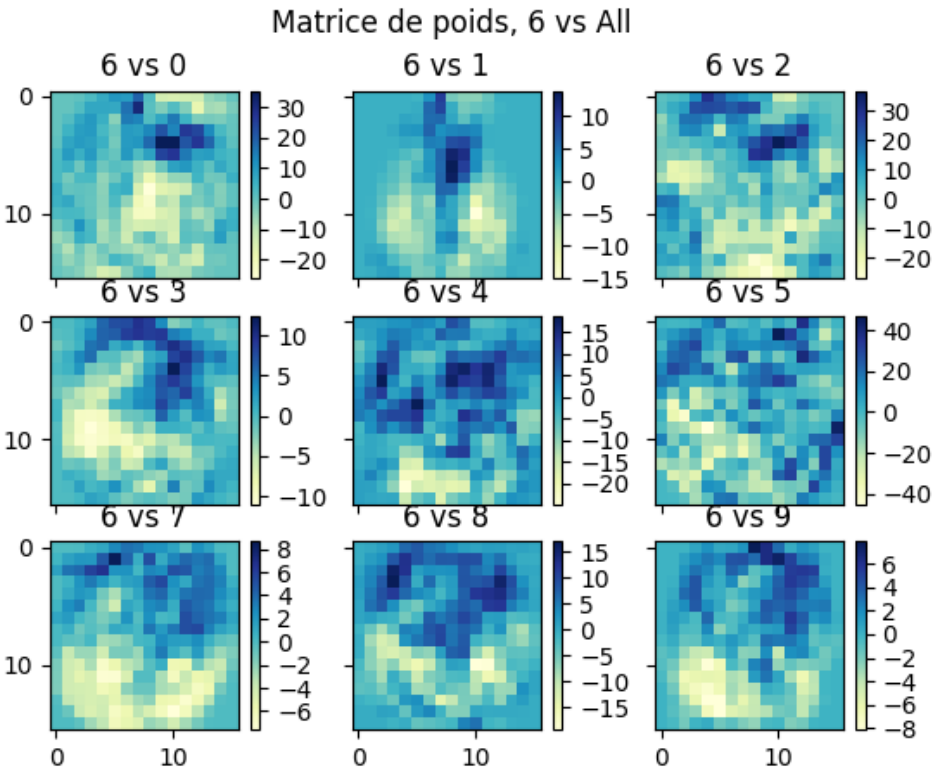
2D Artificial Data



Il classifie aussi bien que le *perceptron* que nous avons implémenté sur les données linéairement séparable.

USPS data : Appliquons le modèle sur les données USPS.





Error	6vs0	6vs1	6vs2	6vs3	6vs4	6vs5	6vs7	6vs8	6vs9
Train	0	0	0	0	0	0	0	0	0
Test	0.013	0.016	0.027	0.006	0.024	0.009	0.012	0.0089	0.000

FIGURE 14 – Evaluation du perceptron de sklearn sur les données USPS

Sur les données USPS les scores nous montrent que le *perceptron* de *sklearn.linear_model* est presque parfait. En *train* il ne fait absolument aucune erreur. Et en test, il en fait pas moins de 99%
Sur les graphiques, nous remarquons aussi que contrairement aux tests précédents sur les classifieurs implémentés, il ne confond pas certains chiffres comme le 0 et le 6. C'est pourquoi d'ailleurs que le graphique a l'air flou. Il n'attribue pas tous les pixels communs à un seul des deux chiffres.

6.1.2 SVM and GridSearch

GridSearch de *sklearn.model_selection* est un outil très puissant, nous permettant de trouver les paramètres optimaux d'un classifieur en lui passant un ensemble de paramètres sur lesquels itérés.

Sur les données $2D$ (2 gaussiennes), nous obtenons les paramètres suivants :
$\{'C' : 1,' gamma' : 0.0001, kernel =' rbf', max_iter' : 500\}$

Quant aux données $USPS$, nous observons :
$\{'C' : 20,' gamma' : 0.01, kernel =' rbf', max_iter' : 500\}$

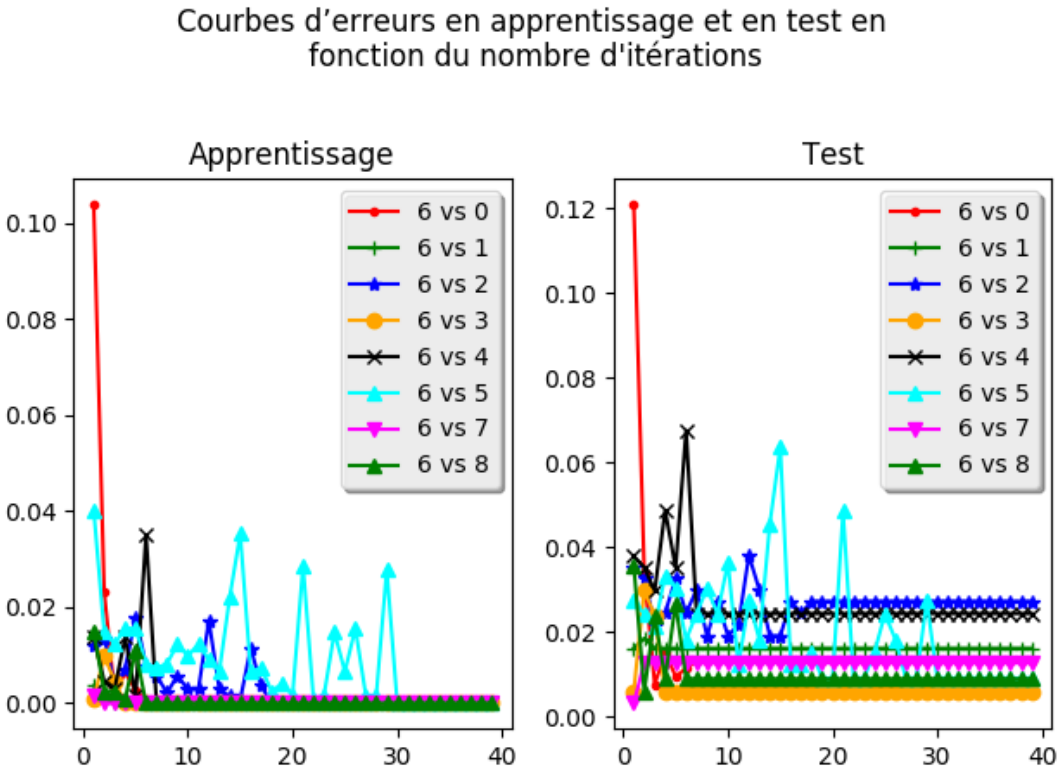
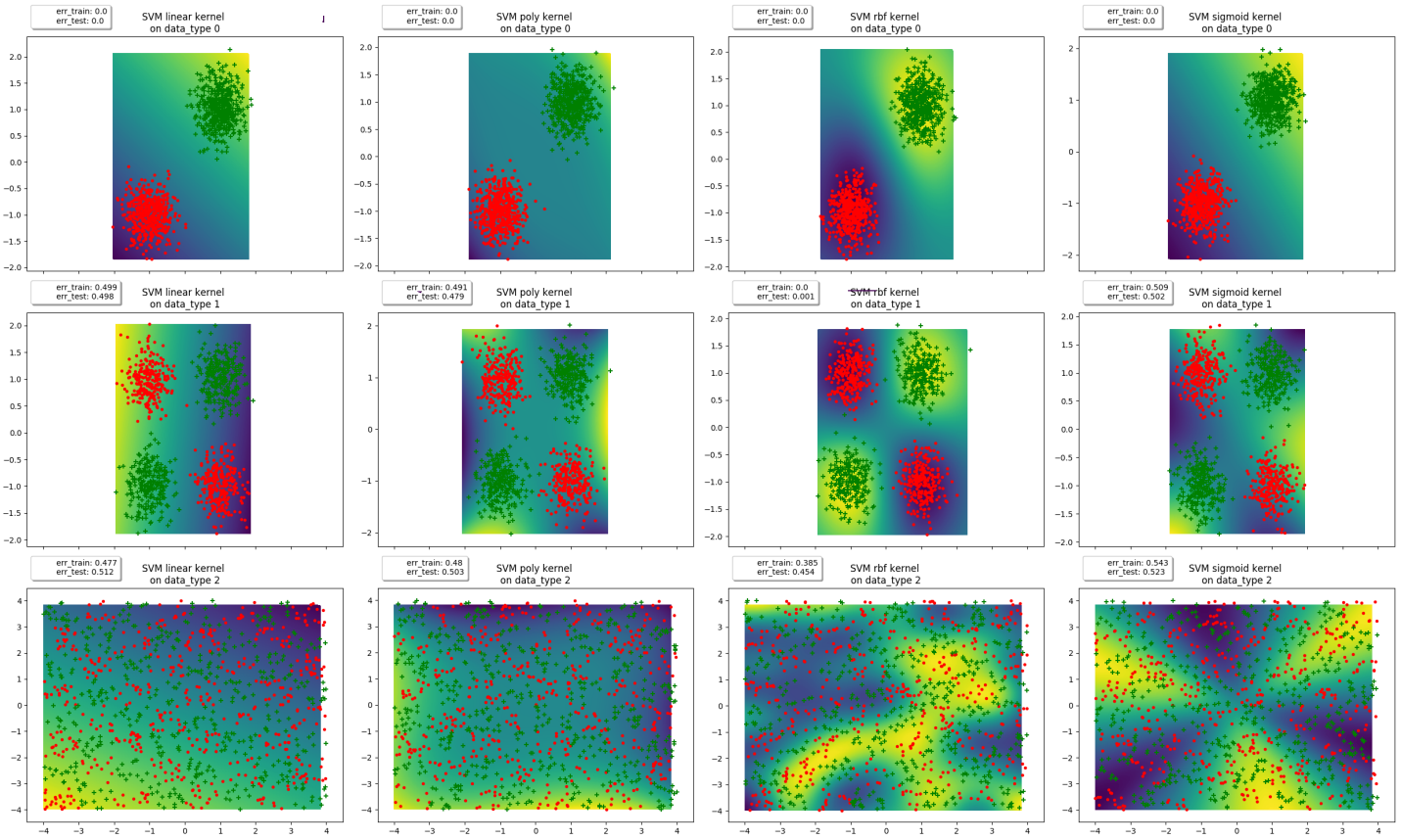


FIGURE 15 – Perceptron de sklearn

Les courbes d'erreurs en apprentissage et en test, nous semble cohérentes. Nous n'observons ni du sur-apprentissage car le modèle est performant en test, ni du sous-apprentissage.

De plus, nous avons testé les différents noyaux, *linaire*, *gaussien* (rbf) et *polynomial* et *sigmode*, aux différents types de données 0, 1 et 2 et avons obtenu les frontières ci-dessous.

Les différents noyaux se sont comportés comme attendu à l'exception du *rbf* sur l'échiquier, ou nous sommes restés sur notre faim. Nous n'avons pas pu décelé pourquoi est-ce qu'il n'arrivait pas à classifier le problème de l'échiquier, même en jouant avec les nombreux paramètres, la *pnalit*, le *gamma*, le **nombre d'itération**, etc...



6.1.3 Apprentissage multi-classe

Nous aborderons ici les deux méthodes courantes qui nous permettent de faire de l'apprentissage multi-classe : $One - versus - One$ et le $One - versus - All$ encore appelé $One - versus - Rest$.
L'idée derrière ces deux méthodes est de ramené l'apprentissage multi-classe à un problème de classification binaire. La première, apprend $k(k-1)/2$ classifieurs permettant ainsi de départager tout couple de labels y_i, y_j. La décision se fait en comptant tout simplement le nombre de $+1$ de chaque classifeurs. Celui ayant eu le nombre de vote le plus élevé est enfin prédit par le classifieur englobant.
La seconde, elle, apprend k classifieurs, si on a k labels. Classifier chaque classe $+1$ par rapport à toutes les autres -1. De plus, aulieu de prédire un label, cette méthode prédit plutôt un score et la décision se fait en choisissant le classifieur ayant le score le plus élevé.

Nous avons testé, ces deux méthodes sur les données $USPS$ à laide des outils de $sklearn$.

Méthode	$OnevsOne$	$OnevsAll$
Train	0	0.076
Test	0.011	0.100

FIGURE 16 – Evaluation des méthodes $OnevsOne$ et $OnevsAll$ sur les données USPS

6.1.4 String kernel

Le *string kernel* nous permet de calculer ici la similarité entre deux textes. Pour y arriver, il faudrait auparavant faire du prétraitement sur les textes en question à fin d'enlever tous les mots qui risquent de fausser le résultat tels que les stop words, ou même les ponctuations, enlever les accents, mettre en miniscule, etc...

7 Conclusion

Nous sommes donc venus à bout de cette synthèse au cours de laquelle, certaines expériences ont été réussies et d'autres non que nous planifions d'améliorer et aller plus loin dans les expériences.