Compte Rendu de TME
Apprentissage et Reconnaissance de Forme

Julien Denes, Michael Trazzi

Mars 2018

Avant-propos

L'ensemble du travail présenté dans ce rapport, y compris ce document lui-même, son fichier source LaTeX, les figures et surtout le code sur lequel il repose sont disponible à l'adresse suivante : https://github.com/mtrazzi/arf/.

TME 1 - Arbres de décision, sélection de modèles

Calcul de l'entropie

```
def entropie(vect):
    _, counts = np.unique(vect, return_counts=True)
    p_y = np.array(counts / len(vect))
    return (-np.sum(p_y * np.log(p_y)))
```

```
def entropie_cond(list_vect):
    n = len(list_vect)
    total_nb = sum(len(part) for part in list_vect)
    p = np.array((1,n))
    H = np.array((1, n))
    for i in range(n):
            p[i] = len(list_vect[i]) / total_nb
            H[i] = entropie(list_vect[i])
    return (np.sum(H * p))
```

Quelques expériences préliminaires

Q 1.4 Sur les données IMDB, nous avons testé des profondeurs d'arbres allant de 5 à 50. Plus la profondeur est grande, et plus on surapprend notre base de données d'apprentissage.

Q 1.5 Pour une profondeur de 5 (resp. 50), on obtient un score de 0.736429038587 (resp. 0.900152605189). On a bien un surapprentissage de nos données dans le cas de la profondeur de 50.

Q 1.6 Le score ainsi défini n'est pas un indicateur fiable : il indique uniquement notre capacité a surapprendre la base d'apprentissage, mais ne tient pas compte du pouvoir de généralisation.

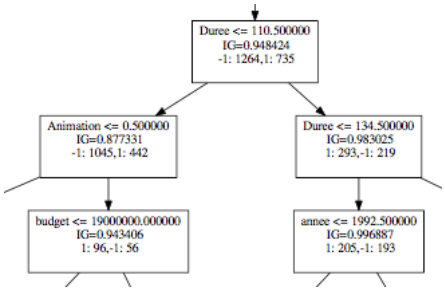

FIGURE 1 – une partie de ce qu'on obtient avec profondeur 5

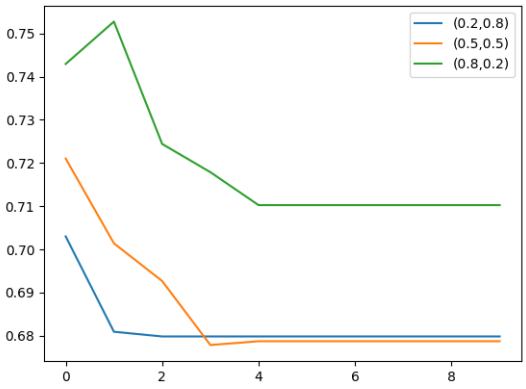

FIGURE 2 – Evolution du score en fonction de (profondeur-1)/5 avec toute la base d'apprentissage

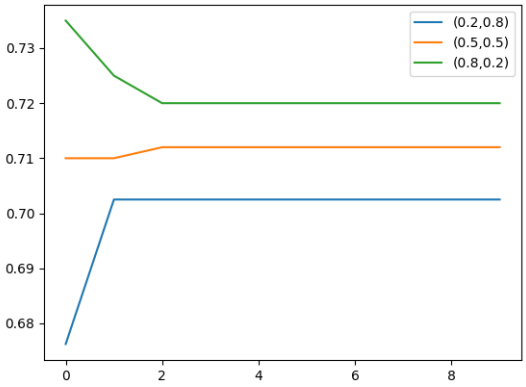

FIGURE 3 – Idem avec peu d'exemples (1000 au lieu de 4000)

Sur et sous apprentissage

Q 1.7 (cf. Figures 2 et 3 pour les courbes)

Q 1.8 Avec peu d'exemples d'apprentissage, le score se stabilise tres rapidement en fonction de la profondeur. Avoir un arbre tres profond avec seulement 1000 exemples ne fait pas varier le score. A contrario, avec l'ensemble de la base d'apprentissage, le score continue a varier jusqu'a une prondeur de 25.

Q 1.9 Mes resultats ne me semblent pas fiables (seulement 5 points pour chaque courbe, base d'apprentissage de seulement quelques milliers d'exemples). Pour les ameliorer il faudrait prendre plus d'exemples (10 000 000), plus de répartitions (au moins 5) et plus de points (100 profondeurs différentes).

TME 2 - Estimation de densité - Expérimentations

Méthode des histogrammes

Méthode à noyaux

Différence entre faible et forte discrétisation

Rôle des parametres des méthodes a noyaux

Choix automatique des meilleurs paramètres

Estimation de la qualité du modèle

TME 3 - Descente de gradient

L'ensemble des implémentations de ce qui est décrit dans la suite sont codées dans le fichier `TME3/tme3-etu.py`. A noter que celui-ci s'appuie entre autres sur `arftools.py`, que l'on utilise pour sa fonction `gen_arti` pour tester notre modèle.

Optimisation de fonctions

On étudie dans cette partie les trois fonctions $f_1(x) = x\cos(x)$, $f_2(x) = x^2 - \log(x)$ et $f_3(x_1, x_2) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2$.

L'algorithme de descente de gradient est testé sur ces trois fonctions. On visualise ce-dessous les trajectoires d'optimisation sur celles-ci. Pour f_1 et f_2, cet affichage est logiquement en 2D (Figures 4 et 5). L'algorithme de descente de gradient est initialisé à $x = 5$, avec un pas de 0.1 et 30 itérations. La fonction f_3 est quant à elle affichée en 3D. L'algorithme est initialisée à $(x_1, x_2) = (5, 5)$ avec un pas de 0.001 et 20 itérations (Figure 6).

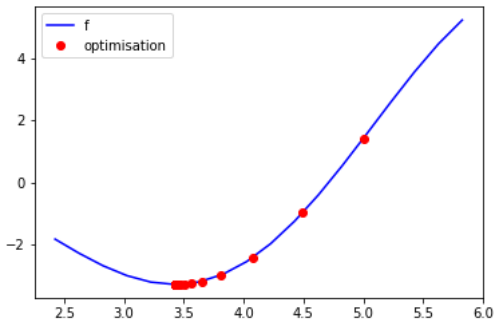

FIGURE 4 – Trajectoire d'optimisation de f_1

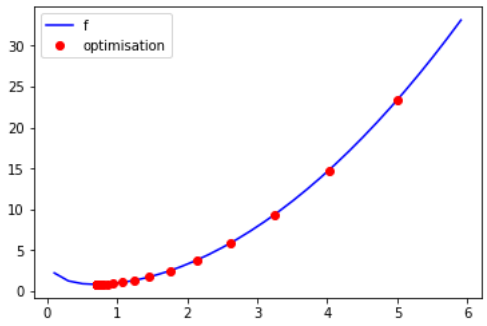

FIGURE 5 – Trajectoire d'optimisation de f_2

Les Figures 7, 8 et 9 montrent, pour chacune des 3 fonctionc, la valeur de l'optimum estimé pour chacune et celle du gradient, en fonction du nombre d'itérations. On constate que pour les 3

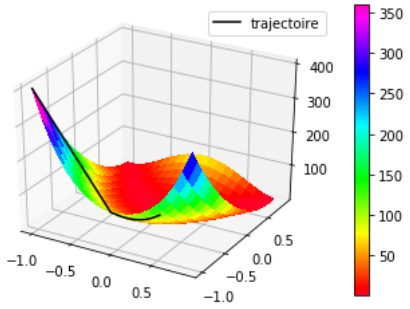

FIGURE 6 – Trajectoire d'optimisation de f_3

fonctions, l'optimum est obtenu assez rapidement (en une dizaine d'itérations). Les réglages sont identiques à ceux des figures précédentes.

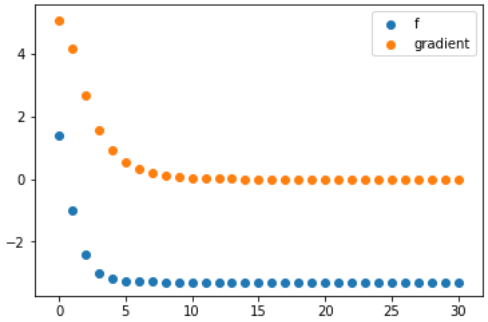

FIGURE 7 – Valeur de f_1 et de son gradient

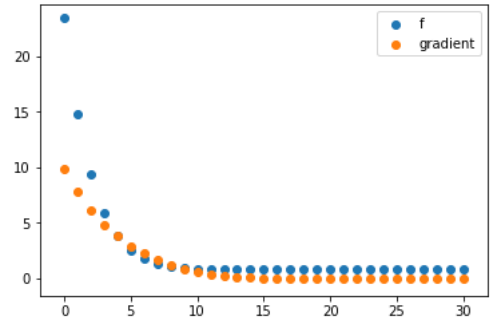

FIGURE 8 – Valeur de f_2 et de son gradient

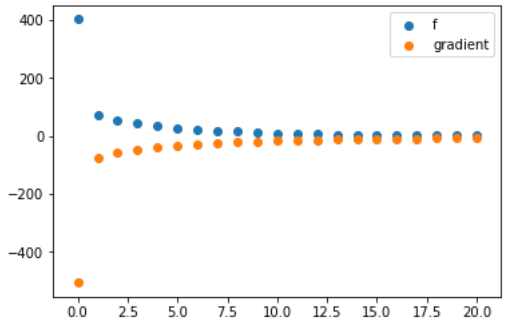

FIGURE 9 – Valeur de f_3 et de son gradient (moyenne sur x_1 et x_2)

On propose enfin d'afficher les courbes de log distance entre la valeur estimée de la fonction et la valeur optimale globale, en fonction de l'itération. Les résultats sont proposés dans les Figures 10, 11 et 12. On semble constater que f_1 et f_2 ont une log distance proportionnelle en fonction du nombre d'itération, c'est à dire que la distance diminue exponentiellement. La forme de la fonction de f_3 est moins facilement interprétatble, et semble "accélérer sa diminution" avec l'augmentation de l'itération (entre 40 et 50).

Régression logistique

La classe correspondant au classifieur basé sur la régression logistique est nommé `Learner` dans notre implémentation sous Python. Il dispose d'une méthode d'initialisation, des méthodes `fit`,

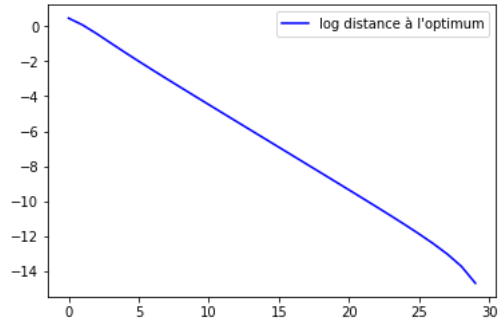

FIGURE 10 – Log distance de f_1 à l'optimum

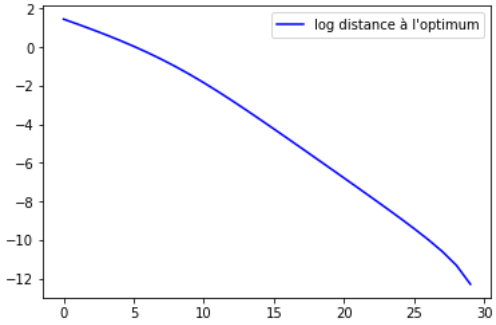

FIGURE 11 – Log distance de f_2 à l'optimum

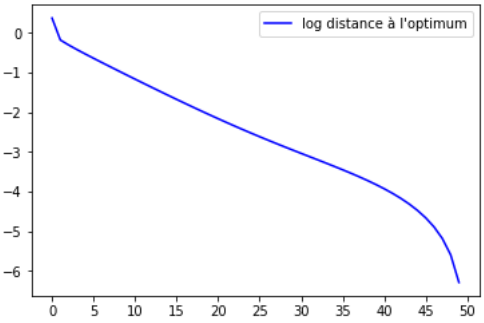

FIGURE 12 – Log distance de f_3 à l'optimum

`predict` et `score` comme définies dans l'énoncé, ainsi que des méthodes `loss` qui calcule l'erreur des associée à ce modèle aux données passées en entrée, et d'une méthode `grad\_loss` qui fournit le gradient de cette fonction d'erreur.

On commence par tester ce classifieur sur les données générées artificiellement grâce à la fonction `gen_arti` empruntée aux outils du TME4 : celui-ci se révèle très performant sur ces données, avec une erreur de 8% en entrainement et 9% en test (voir Figure 13).

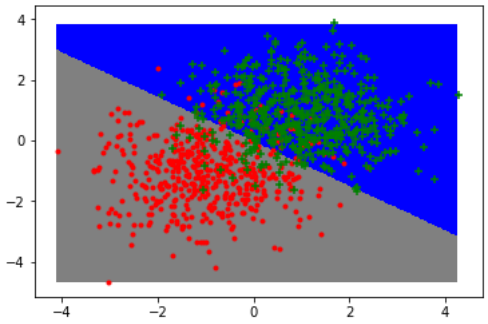

FIGURE 13 – Frontière de décision de la régression logistique

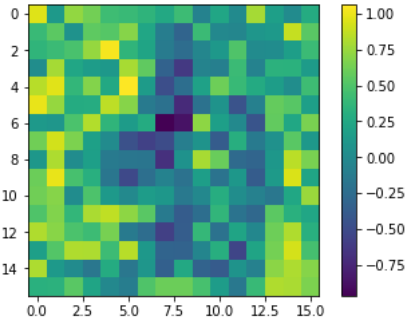

FIGURE 14 – Affichage graphique des 256 poids du classifieur, sous forme de matrice 16×16

On teste ensuite notre classifieur sur les données USPS. Celui-ci s'avère remarquablement performant : par exemple sur la reconnaissance des "6" contre les "9", il obtient une erreur de 0.0% en entrainement et 0.1% en test. Pour la reconnaissance des "1" contre toutes les autres classes, il obtient un score de classification incorrecte de 1.4% en entrainement et 2.5% en test.

On propose ci-dessous un affichage graphique du vecteur de poids déterminé par l'algorithme pour

la classification 6 vs. 9. Celui-ci étant de taille 256, on l'a reformaté sous forme d'une matrice 16×16 pour retrouver les dimension des images. Cet affichage est proposé en Figure 14. Pour cette classification, nous avons transformé les labels "6" en -1 et les labels "9" en $+1$ (après avoir isolé les seules images de 6 et de 9). On pourrait donc s'attendre à trouver des poids faibles sur les pixels traçant un 6, et des poids forts pour ceux traçants un 9. La Figure 14 ne semble pas montrer de tels motifs, sûrement à cause du chevauchements des formes "6" et "9".

TME 4 et 5 - Perceptron

L'ensemble des implémentations de ce qui est décrit dans la suite sont codées dans le fichier `TME4/tme4-etu.py`. A noter que celui-ci s'appuie entre autres sur `arftools.py`.

Implémentation

La bonne implémentation des fonctions de coût MSE et Hinge ainsi que de leur gradient respectif est bien vérifiée par l'affichage de leurs isocontours (voir Figures 15 et 16).

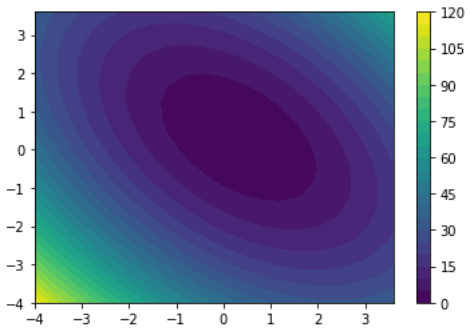

FIGURE 15 – Isocontours de la fonction d'erreur MSE

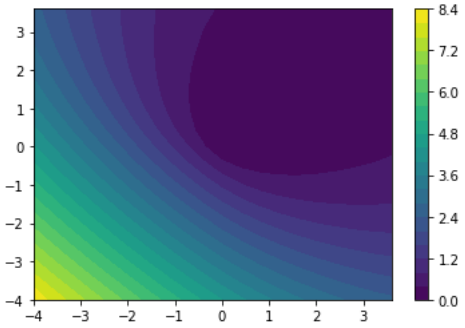

FIGURE 16 – Isocontours de la fonction d'erreur Hinge

On vérifie également bien que l'implémentation de `Lineaire` est correcte grâce au données de `gen_arti` : comme régression linéaire elle obtient une classification incorrecte de 14.5% en entrainement et 14.3% en test (voir Figure 17), et comme perceptron 9.6% en entrainement et 8.5% en test (Figure 18).

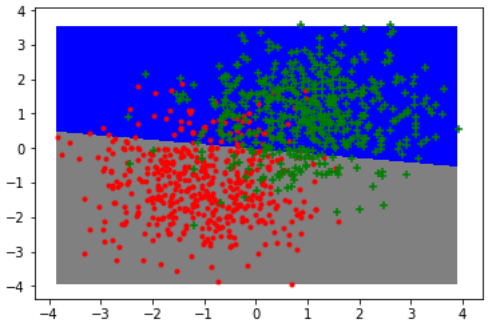

FIGURE 17 – Frontière de la régression linéaire

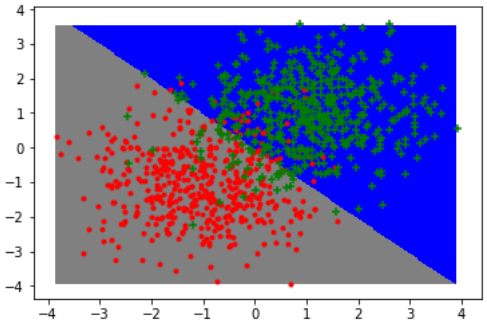

FIGURE 18 – Frontière du perceptron

On propose également d'afficher la trajectoire d'optimisation sur la surface d'isocontour des fonctions d'erreur pour vérifier la bonne convergence. Celle-ci semble bonne dans les Figures 19 et 20.

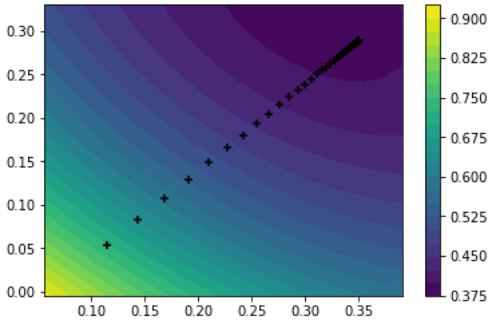

FIGURE 19 – Trajectoire d'optimisation pour la fonction MSE

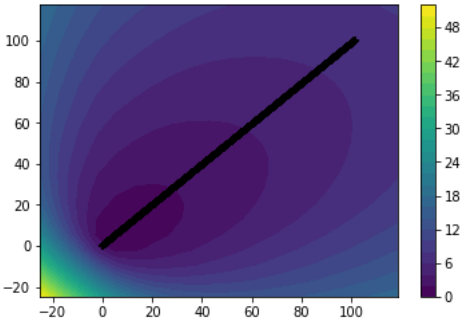

FIGURE 20 – Trajectoire d'optimisation pour la fonction Hinge

On implémente également dans notre classe `Lineaire` une bossibilité d'ajouter un biais : il suffif pour cela de fixer la variable `self.bias` sur True. L'ajout de ce biais permet (étonnamment) d'obtenir de moins bons résultats pour la régression linéaire (11.5% d'erreur en test) mais quelques peu meilleurs pour le perceptron (8.2%). Les frontières de décisions sont affichées dans les Figures 21 et 22.

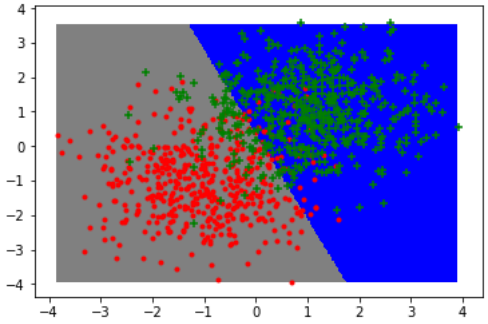

FIGURE 21 – Frontière de décision de la régression linéaire avec biais

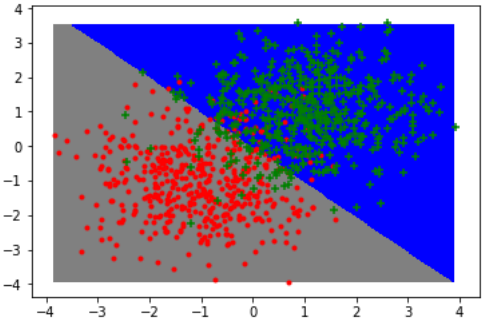

FIGURE 22 – Frontière de décision du perceptron avec biais

Données USPS

On commence par étudier la capacité de notre perceptron à distinguer les deux classes "6" et "9", comme on l'avait fait pour la régression linéaire au TME précédent. Comme pour cette dernière, le perceptron s'avère très efficace : son taux d'erreur en entrainement est de 0.2%, et de 0.7%. On affiche dans la Figure 23 le vecteur de 256 poids sous forme d'une matrice 16×16 de manière similaire à ce que nous avions fait dans le TME3. Cependant le résultat est cette fois-ci bien plus intéressant : on discerne clairement que les poids correspondant aux pixels placés sur le tracé du "6" (encodé en -1) sont fortement négatifs, tandis que ceux placés sur le tracés d'un "9" ($+1$) sont fortement positifs. Les autres sont proches de 0.

On réitère l'entrainement en tentant cette fois-ci de faire discerner le "6" de toutes les autres classes. les résultats sont encore bons, avec un taux d'erreur de 8.4% en entrainement et 9.1% en test. L'affichage du vecteur de poids (Figure 24) nous permet de constater cette fois-ci que tous les poids sont négatifs (on a encodé $+1$ si c'est un "6", -1 sinon) : on peut donc dire que le classifieurs cherche à reconnaitre tous les autres chiffres que le 6, comme le montre l'activation des poids sur ce qui semble être le tracé d'un "3" ou d'un "8".

Les courbes d'apprentissage (erreur en fonction du nombre d'itération), proposée pour chacun de ces deux problèmes (Figures 25 et 26) nous montre qu'il n'y a pas de sur-apprentissage dans ce

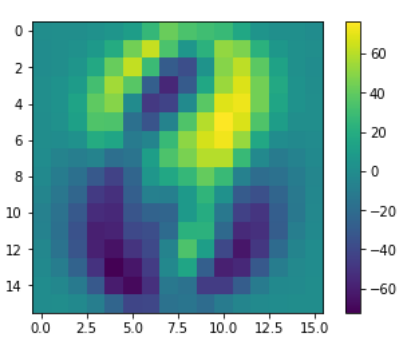

FIGURE 23 – Vecteur des 256 poids (sous forme 16×16) du perceptron pour 6 vs. 9

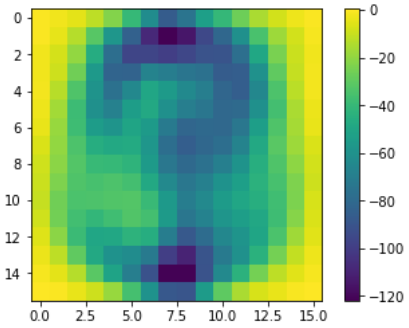

FIGURE 24 – Vecteur des 256 poids (sous forme 16×16) du perceptron pour 6 vs. le reste

cas-ci

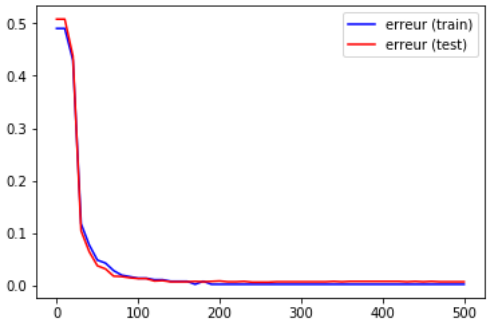

FIGURE 25 – Courbe d'apprentissage du perceptron pour 6 vs. 9

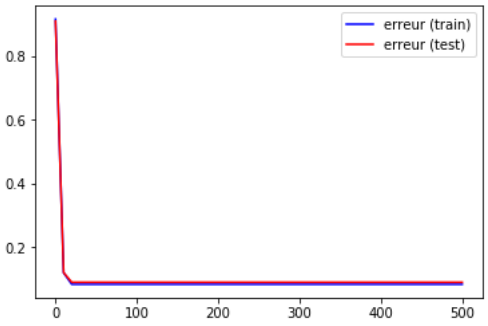

FIGURE 26 – Courbe d'apprentissage du perceptron pour 6 vs. le reste

Données 2D et projection

On cherche maintenant à ajouter des projections pour améliorer l'expressivité de notre perceptron. On commence par lui ajouter une projection gaussienne (en fixant, dans la classe `Lineaire`, la valeur `projec` sur "gauss"), en utilisant comme base un ensemble de 100 exemples de la base d'exemples tirés au hasard (et passés en valeur de `self.base`). Le taux d'erreur n'est pas vraiment amélioré : il est de 9.5% en apprentissage et 10.3% en test. La Figure 27 nous montre que la frontière n'est plus linéaire mais semble "overfitter" quelques-uns des points proches de la séparation.

Nous avons également appliqué une projection polynomiale sur les données 2D. On n'obtient pas là non plus de résultats significativement meilleurs : le taux d'erreur en apprentissage est de 9.5%, et 8.6% en test. Il n'y a donc cette fois-ci sûrement pas de surapprentissage, mais la forme conique de la courbe n'aide pas à séparer les poids des deux gaussiennes qui se chevauchent. La frontière avec cette projection est affichée dans la Figure 28.

TME 6 - SVM et Noyaux

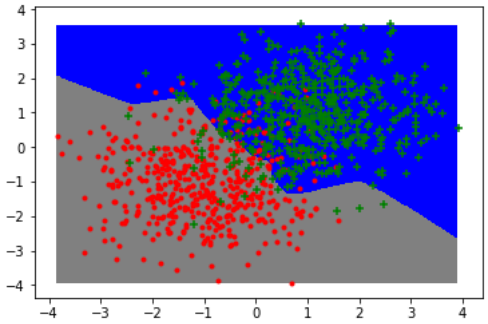

FIGURE 27 – Frontière du perceptron avec projection gaussienne

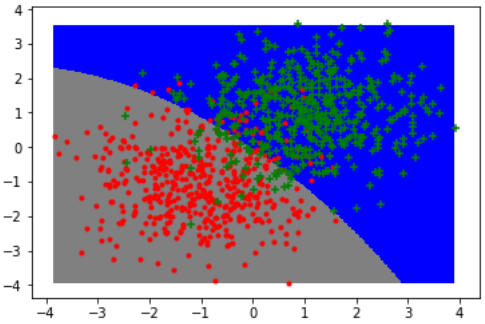

FIGURE 28 – Frontière du perceptron avec projection polynomiale