# Machine Learning

## Compte rendu de TME

Victor PIRIOU - Célina KHALFAT

# TME 1 - Arbres de décision, sélection de modèles

## Entropie

L'entropie permet de mesurer le désordre : quantifier l'aléatoire. L'entropie conditionnelle de X sachant Y représente l'incertitude qu'il reste sur X lorsque l'on connaît Y. Ici, notre Y est la variable représentant le vote : on cherche à prévoir le vote d'un film. Pour cela, on calcule dans un premier temps, l'entropie de chaque attribut binaire :

```
Entropie
{'Sci-Fi': 0.564976837274551, 'Crime': 0.7256620117131098, 'Romance': 0.7540946843905093,
'Animation': 0.2634038042219715, 'Music': 0.1926285852674641, 'Comedy': 0.9487805443246595, 'War':
0.2653268692430364, 'Horror': 0.5347013539632184, 'Film-Noir': 0.040466328209344704, 'Thriller':
0.9104758524619008, 'Western': 0.13310901662996316, 'Mystery': 0.49425468209811785, 'Short':
0.00786173698629584, 'Drama': 0.9999176833887986, 'Action': 0.8103265236207233, 'Documentary':
0.058438060605145864, 'Musical': 0.17702726334400468, 'History': 0.22865199105519707, 'Family':
0.45026771320199777, 'Adventure': 0.6915486835207777, 'Fantasy': 0.5144914462944487, 'Sport':
0.19699814392010093, 'Biography': 0.3061179094676092, 'couleur': 0.2013306211270578}
```

Ainsi que leur entropie conditionnelle de chaque attribut sachant le vote.

```
Entropie Conditionnelle
{'Sci-Fi': 0.9809464709356692, 'Crime': 0.9862397337182898, 'Romance': 0.9869061262363985,
'Animation': 0.9846725825766344, 'Music': 0.9868922312099724, 'Comedy': 0.9709077861783149, 'War':
0.9778090284609612, 'Horror': 0.9642153203055335, 'Film-Noir': 0.9833321472296909, 'Thriller':
0.9754570883513247, 'Western': 0.984875569644032, 'Mystery': 0.9867824646754102, 'Short':
0.9863737670007015, 'Drama': 0.9262469118518663, 'Action': 0.9704709047945151, 'Documentary':
0.985653165935511, 'Musical': 0.9844489392750518, 'History': 0.9732710207451594, 'Family':
0.9850273935122518, 'Adventure': 0.9866370234204362, 'Fantasy': 0.9846264056214122, 'Sport':
0.9869073344864076, 'Biography': 0.9658241922244407, 'couleur': 0.18312206830137276}
```

Et dans un deuxième temps, on calcule le gain d'information pour chaque attribut, qui correspond à la différence entre son entropie et l'entropie conditionnelle de l'attribut sachant le vote.

```
Gain d'information
{'Sci-Fi': -0.4159696336611183, 'Crime': -0.2605777220051799, 'Romance': -0.2328114418458892,
'Animation': -0.7212687783546629, 'Music': -0.7942636459425083, 'Comedy': -0.022127241853655333,
'War': -0.7124821592179248, 'Horror': -0.4295139663423151, 'Film-Noir': -0.9428658190203462,
'Thriller': -0.06498123588942395, 'Western': -0.8517665530140688, 'Mystery': -0.49252778257729235,
'Short': -0.9785120300144056, 'Drama': 0.07367077153693236, 'Action': -0.16014438117379182,
'Documentary': -0.9272151053303651, 'Musical': -0.8074216759310472,
'History': -0.7446190296899623, 'Family': -0.534759680310254, 'Adventure': -0.29508833989965855,
'Fantasy': -0.4701349593269636, 'Sport': -0.7899091905663066, 'Biography': -0.6597062827568315,
'couleur': 0.018208552825685037}
```

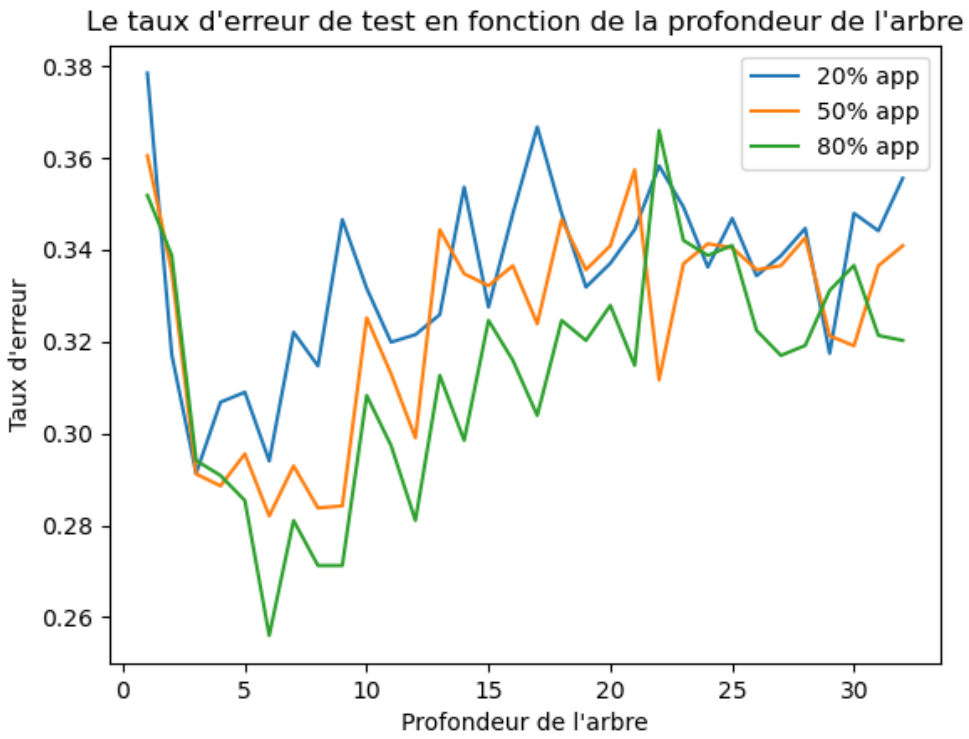
L'attribut dont la valeur maximise le gain d'information est l'attribut est considéré comme le split optimal : l'attribut qui permet à la fois de bien partitionner en groupe homogène et dont la distribution est proche de celle du vote.
Ainsi le meilleur attribut pour la première partition est : “Drama”.

## Quelques expériences préliminaires

On constate que le nombre d'exemples séparés diminue à mesure que la profondeur augmente.  Cela est normal car une plus grande profondeur implique un plus grand nombre de nœuds, donc un plus grand nombre de tests/splits déjà effectués.
Lorsqu'on calcule les scores de bonne classification, plus la profondeur est grande, plus le score est élevé. Cela est normal, c'est dû au fait qu'on est de plus en plus précis, et donc qu'on a moins de chance de se tromper. Par ailleurs, ces scores ne sont pas un indicateur fiable car on teste sur les données sur lesquelles on s'est entraîné. Pour obtenir un indicateur plus fiable il faudrait tester notre programme sur des données sur lesquelles on ne s'est pas entraîné.

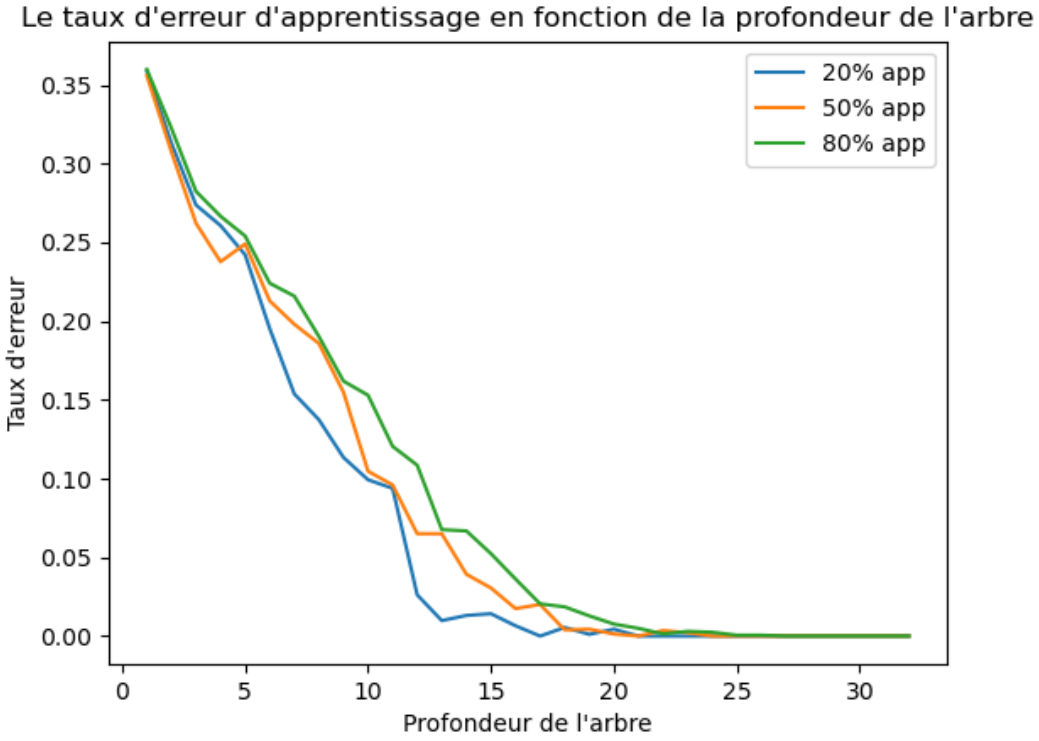
## Sur et sous-apprentissage

Pour obtenir un indicateur plus fiable, on partitionne nos données de sorte à pouvoir apprendre sur une partie et tester notre modèle sur l'autre partie. On a testé trois partitionnement différents, 80% des données d'apprentissage avec 20% de données de tests, 50% de données d'apprentissages et de tests et 80% de données d'apprentissages et 20% de données de tests.




Lorsqu'il y a peu d'exemples d'apprentissage, on remarque que le taux d'erreur de test a tendance à être plus élevé. L'erreur décroît fortement jusqu'à une profondeur de 3 pour le partage (0.2,0.8), de 6 pour les partages (0.5,0.5) et (0.8,0,2) puis remonte légèrement pour se stabiliser autour de 0.33. En général, on privilégie la partition 80% de données pour l'apprentissage et 20% pour les données de tests.
Lorsqu'il y a beaucoup d'exemples d'apprentissage, globalement la courbe a la même forme mais l'erreur de test est en moyenne moins élevée.
Un fort taux d'erreur de test peut avoir plusieurs origines :

- On descend trop loin dans les profondeurs de l'arbre, on perd en généralisation en sélectionnant les données sur de mauvais critères. C'est le sur-apprentissage.
- On ne descend pas assez dans les profondeurs de l'arbre, on ne teste pas assez les données. C'est le sous-apprentissage.
- Le pourcentage de données sur lesquelles on apprend est trop faible par rapport aux données sur lesquelles on teste.



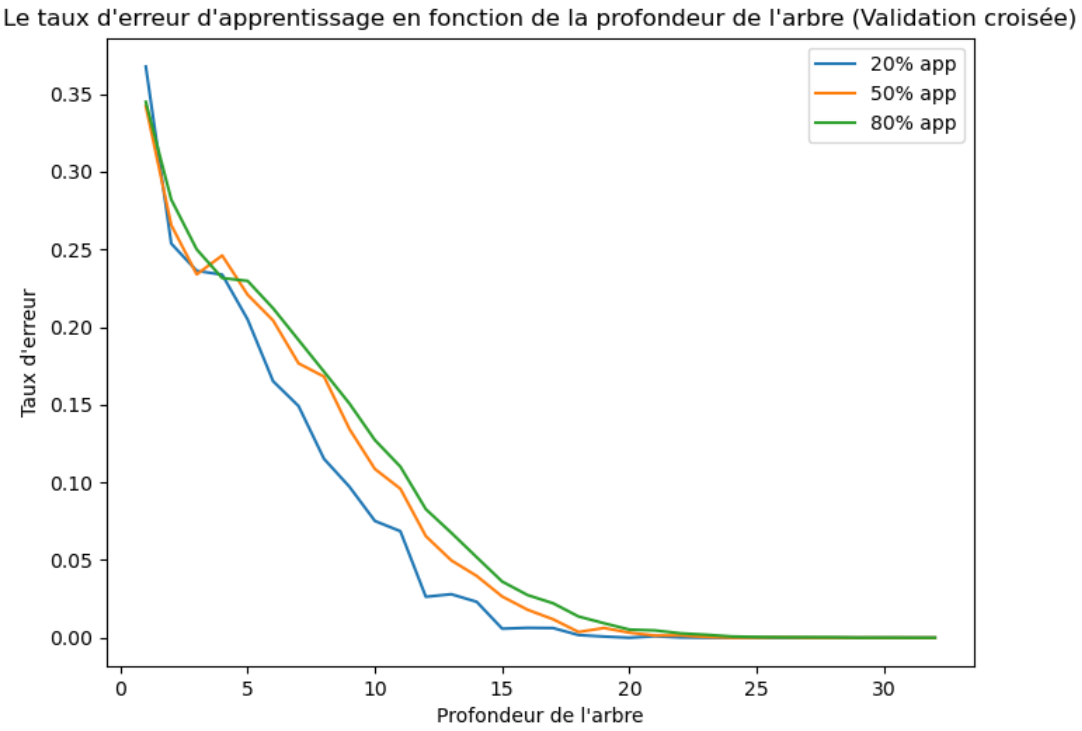

Pour l'erreur d'apprentissage, on n'observe pas le même comportement. En effet, le taux d'erreur d'apprentissage ne cesse de diminuer à mesure que la profondeur de l'arbre augmente, jusqu'à devenir nul. C'est un résultat attendu puisqu'ici le modèle est testé avec les données grâce auxquelles il a été entraîné. Par conséquent, à partir d'une certaine profondeur il devient possible d'apprendre par cœur les données. En somme, le test est biaisé.

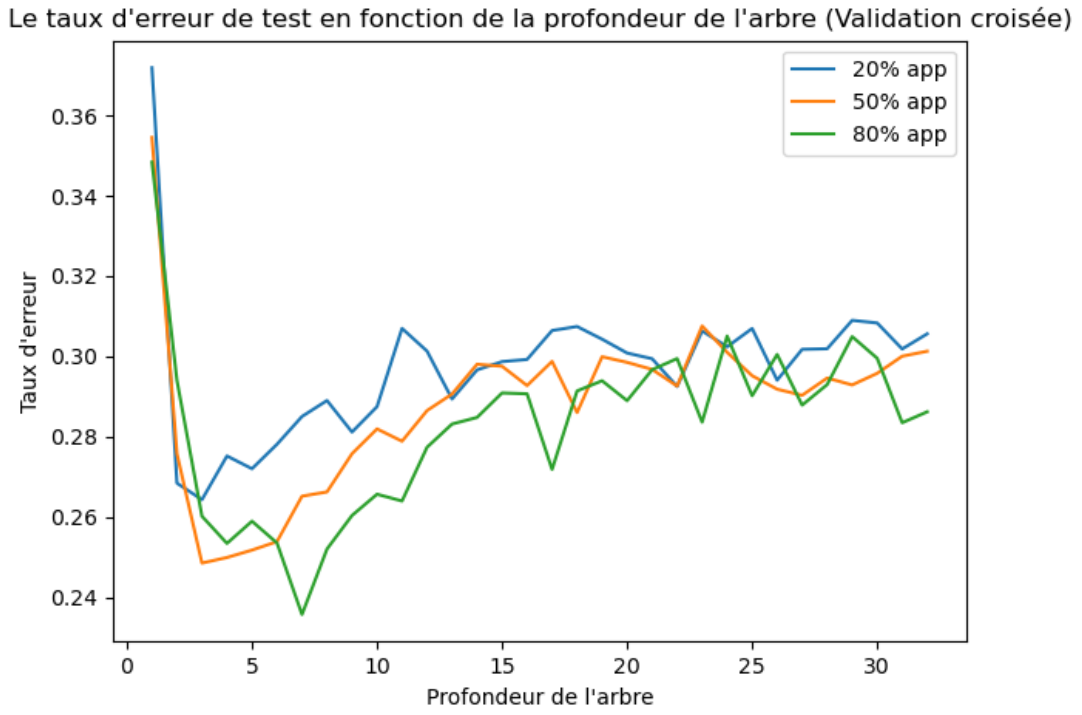
Cependant, ces résultats ne sont ni stables ni fiables car ils dépendent d'une seule partition. On peut les améliorer en utilisant la validation croisée.

## Validation croisée : sélection du modèle

La validation croisée consiste à séparer nos données en k échantillons : on apprend sur k-1 échantillons et le dernier échantillon sert d'ensemble de validation. On répète cette méthode de sorte à ce que chaque échantillon a été une fois l'ensemble de validation. Ainsi, pour chaque ensemble de validation, on obtient un score : on a k scores. On peut estimer la performance de notre modèle grâce à la moyenne et l'écart type des k scores.

Par ailleurs, bien que la validation croisée peut aider à détecter le sur-apprentissage, elle ne peut pas le réguler.




On constate que les courbes des taux d'erreur d'apprentissage et de test sont plus "lisses", ce qui est normal car pour chaque profondeur, l'erreur ne dépend plus d'une seule partition mais d'un ensemble de partitions. En clair, avec la validation croisée, le taux d'erreur est une moyenne des taux d'erreur des différentes partitions.

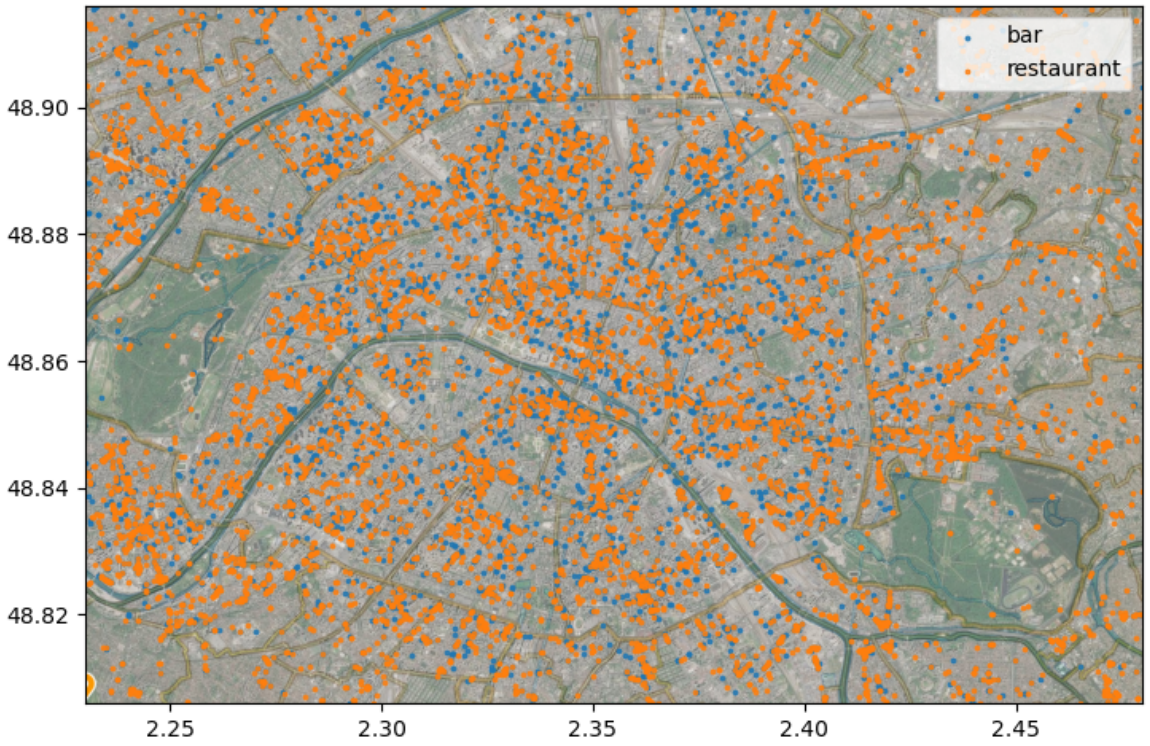
# TME 2 - Estimation de densité

## Log-vraisemblance

La fonction log étant définie sur ]0, + ∞[, on ajoute une très petite valeur à chacune des densités pour s'assurer qu'aucune d'entre elles ne soient nulles.

## Données

Nous allons utiliser le jeu de données POI qui contient différents types de points d'intérêt de Paris.

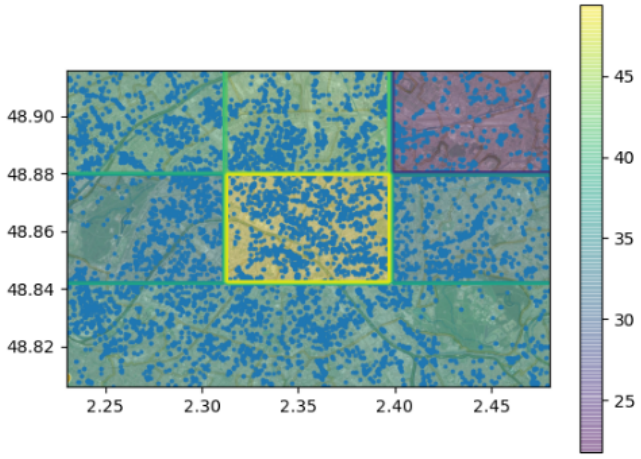



Par la suite, nos expérimentations seront basées sur la base de données des restaurants.

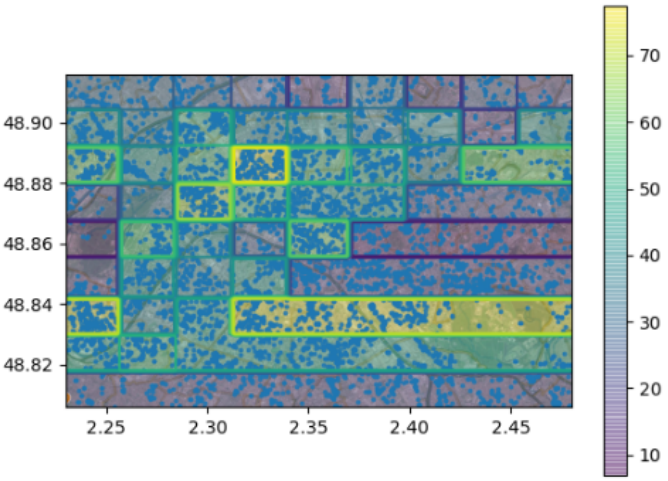
## Méthode par histogramme

Pour vérifier que notre estimateur est correct, on vérifie que la somme des valeurs de notre grille vaut 1 et que sa taille est la même que celle de l'histogramme, ou égale à la plus petite majoration possible de la taille de l'histogramme.

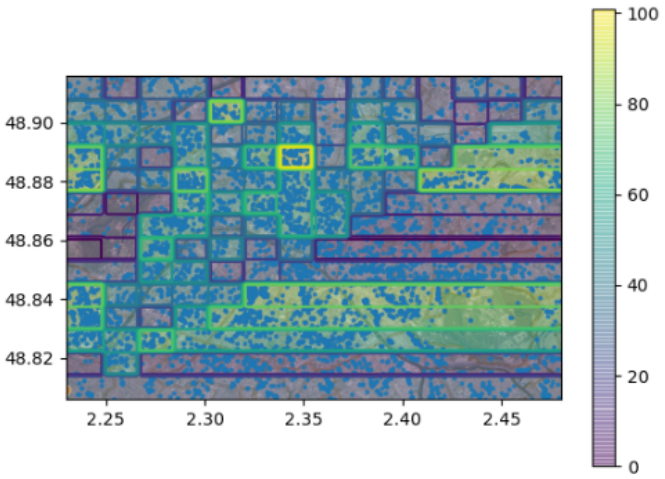
```
def check_size_and_density(h,step):
    somme = 0
    n,m = h.shape
    for i in range(n):
        for j in range(m):
            somme += h[i,j] * ((xmax-xmin)*(ymax-ymin))/(step**2)

     print("densite = ",somme," (valeur attendue : 1)  |  taille : ",n,"x",m," (taille
attendue: ",step,"x",step, ')')
```



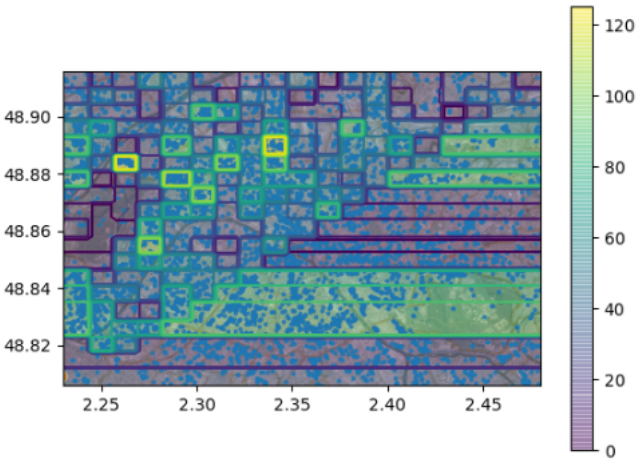

*3 intervalles*




*9 intervalles*



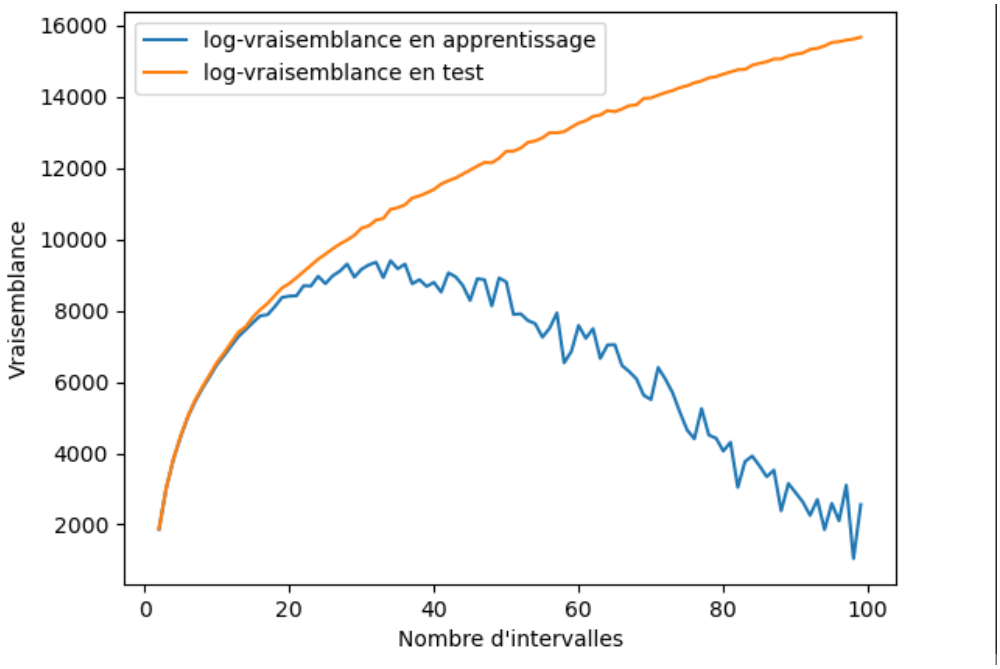

*14 intervalles*

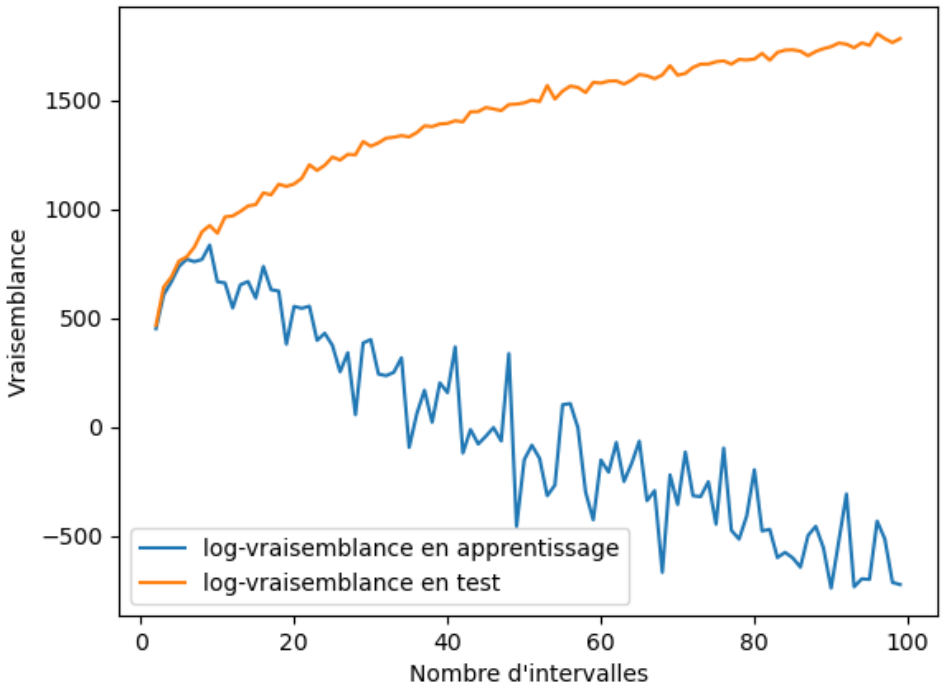



*19 intervalles*

On constate qu'avec un pas de discrétisation trop faible, on est en sous-apprentissage car on distingue trop difficilement les lieux entre eux. On pourrait segmenter plus précisément nos données.
Tandis qu'avec un pas de discrétisation trop grand, on est en sur-apprentissage car à chaque zone est associée un seul lieu. On ne généralise plus assez.

Maintenant, si on partitionne aléatoirement nos données en deux ensembles (apprentissage = 50\%, test = 50\%) et qu'on calcule la log-vraisemblance en apprentissage et en test, on peut afficher les courbes suivantes.



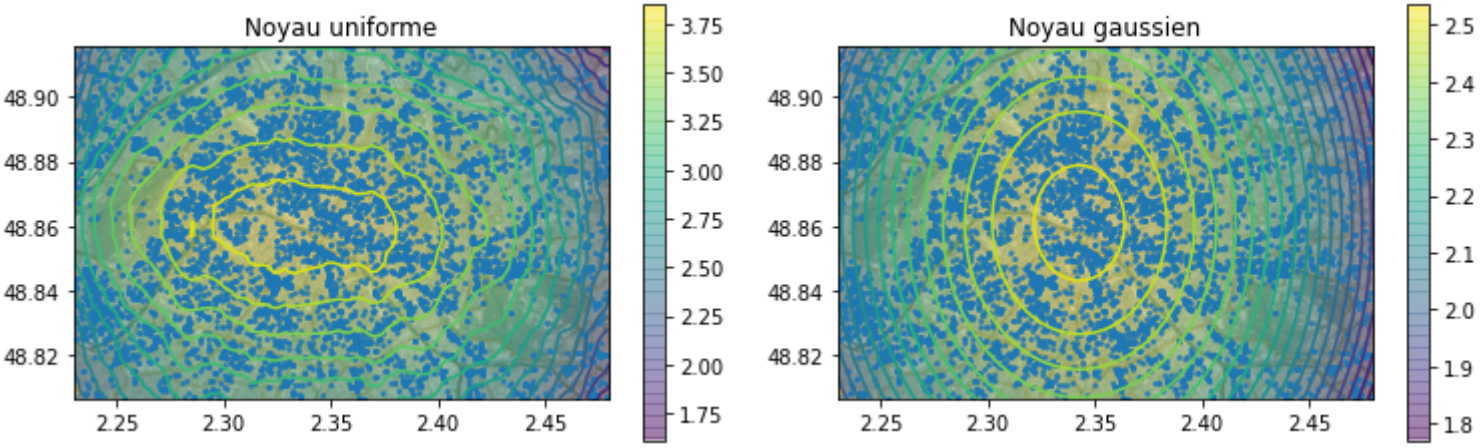

La première image correspond à l'estimation de la vraisemblance en apprentissage et en test sur les données des restaurants) et la deuxième à celle sur les clubs. Le meilleur hyper-paramètre pour les clubs est bien plus petit que celui pour les restaurants. Cela pourrait s'expliquer par le fait qu'on ne dispose pas assez de données pour apprendre le modèle, ni pour le tester de manière fiable. En effet, la taille de la base de données des discothèques (4888) est plus petite que celle de la base de données des restaurants (6914). La validation croisée nous apporterait certainement une estimation plus fiable.

## Méthodes des noyaux

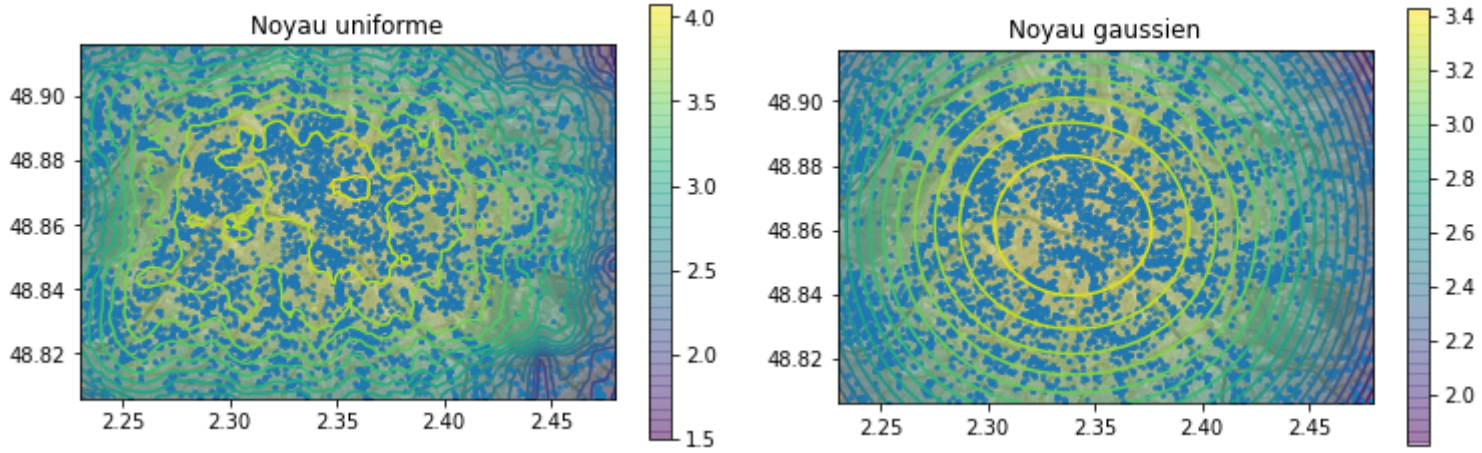
Nous avons les deux noyaux suivants :

```
def kernel_uniform (x):
     return np.intc(np.all(np.abs(x) <= 0.5, axis = 1))

def kernel_gaussian(x):
    return ((2*np.pi)**(-len(x[0])/2))*np.exp(-0.5*np.sum(x**2,axis=1))
```

Pour sigma = 0.1, nous obtenons :



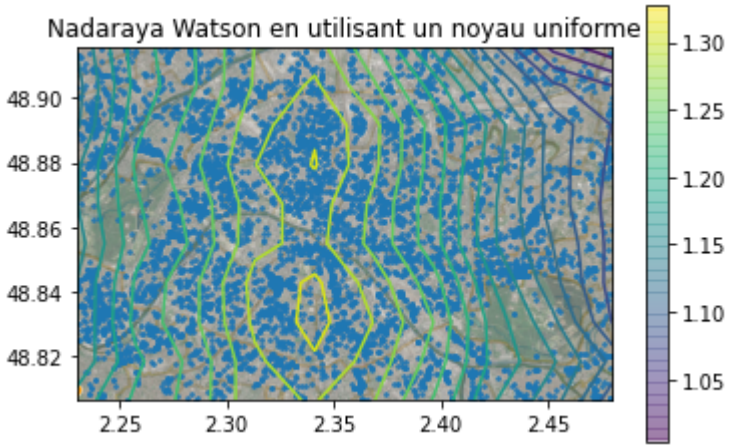

Pour sigma = 0.05, nous obtenons :

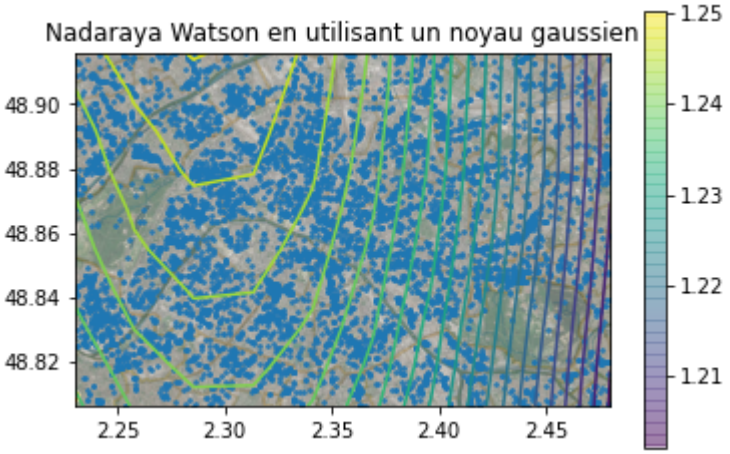



Prenons l'exemple du noyau uniforme, plus sigma est petit, plus nous sommes précis. Avec un sigma trop petit, nous pourrons être en sur-apprentissage. Alors que si sigma est trop grand, nous parlerons de sous-apprentissage.

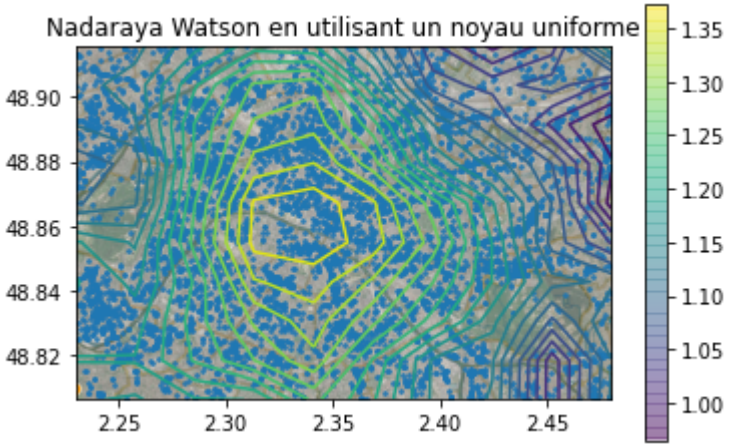
## Nadaraya-Watson

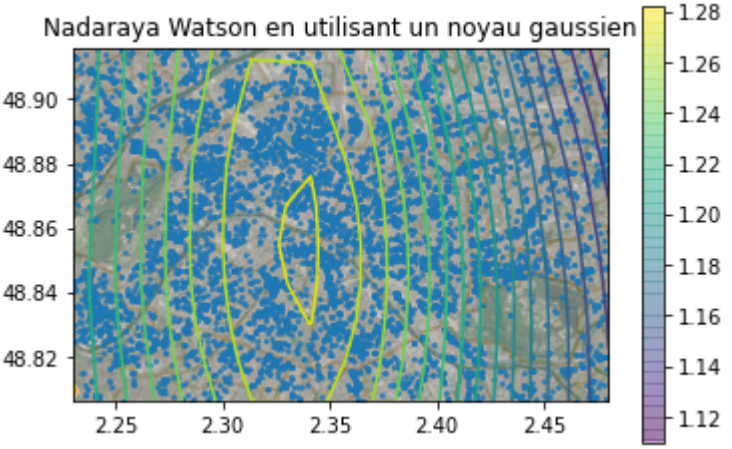
Pour sigma = 0.1, nous obtenons :




Pour sigma = 0.05, nous obtenons :



# TME 3 - Descente de gradient

## Implémentation des fonctions de coûts et de leurs gradients

On a implémenté la fonction de coût aux moindres carrés et la fonction de coût de la régression logistique ainsi que leurs gradients.
Pour vérifier que nos fonctions de gradients étaient correctes, on a implémenté une fonction *grad_check(f, f_grad, N)* qui nous permet de vérifier que nos fonctions de gradients sont correctes.
Pour cela, cette fonction tire N points au hasard, et nous renvoie un dictionnaire qui compare les résultats obtenus, à l'aide des fonctions de gradient et à l'aide de la formule de la dérivée, et la somme des écarts obtenus.
Par exemple, pour N=10, la régression logistique comme fonction f et son gradient pour f_grad, on obtient :

```
({0: {'gradient': 44.0, 'taylor': 44.00000000046588},
  1: {'gradient': 97.0, 'taylor': 97.00000000094632},
  2: {'gradient': 10.999816284359671, 'taylor': 10.99981629462121},
  3: {'gradient': 37.0, 'taylor': 37.00000000037562},
  4: {'gradient': 73.0, 'taylor': 73.00000000043383},
  5: {'gradient': 93.0, 'taylor': 93.00000000109775},
  6: {'gradient': 16.999999296210618, 'taylor': 16.999999296629653},
  7: {'gradient': 20.999999984076624, 'taylor': 20.99999998428359},
  8: {'gradient': 25.999999999867168, 'taylor': 26.00000000008151},
  9: {'gradient': 72.0, 'taylor': 72.00000000011642}},
 1.4537691939153774e-08) #La somme des écarts obtenus
```

On peut alors conclure que notre fonction qui calcule le gradient de la fonction logistique est correcte. Lorsque nous faisons les mêmes expériences avec la fonction de coût aux moindres carrés, on obtient des écarts un tout peu plus grands, avec une somme des écarts obtenus d'environ 0.2 pour N=10.

## Implémentations des régressions

Nous avons une classe *Regression,* qui prend comme arguments *x_train et y_train* qui correspondent à l'ensemble d'apprentissage ainsi qu'un string *type*. Si *type* vaut "linéaire', nous utiliserons la fonction *mse* comme fonction de coût et *mse_grad* comme gradient, sinon la fonction de coût utilisée sera *reglog* et son gradient *reglog_grad*.
Elles possèdent les fonctions suivantes :

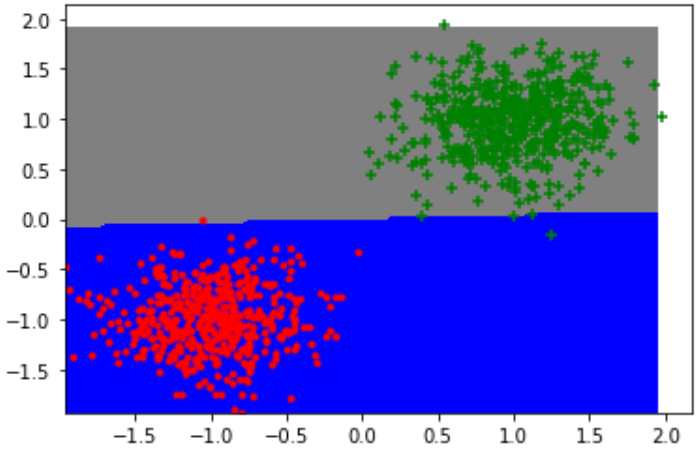
- fit (self, eps, iter, init) retourne le paramètre optimal w trouvé, la liste des w et les valeurs de la fonction de coût au fur et à mesure des itérations pour l'ensemble d'apprentissage.
- predict (self,x_test) retourne les étiquettes y prédites (1 ou -1).
- score (self, x_test, y_test) correspond à un 0-1 loss. Elle retourne le nombre d'étiquettes mal prédite.

## Expérimentations

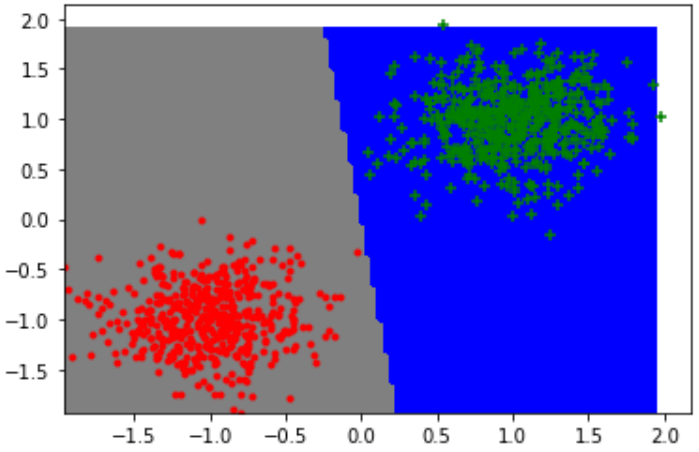
On a partitionné nos données en deux ensembles : l'ensemble d'apprentissage et l'ensemble de tests. L'ensemble d'apprentissage comporte 80% des données. Nous allons dans un premier temps nous intéresser aux cas linéairement séparables, pour cela nous aurons epsilon = 0.1.

### Descente de gradient : régression linéaire

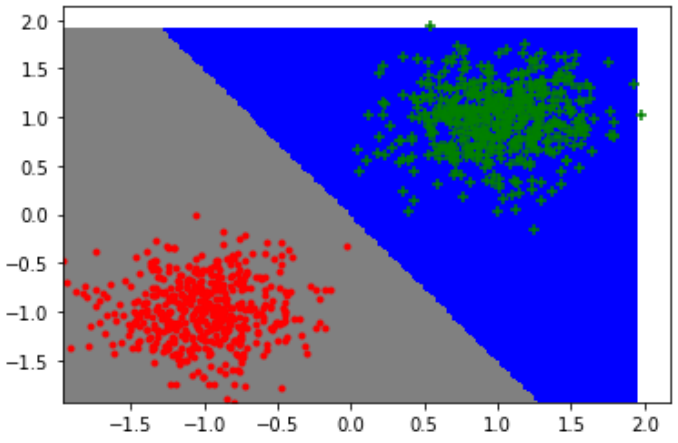
Observons l'évolution de sa frontière de décision avec le w0 initial à [0, -1].



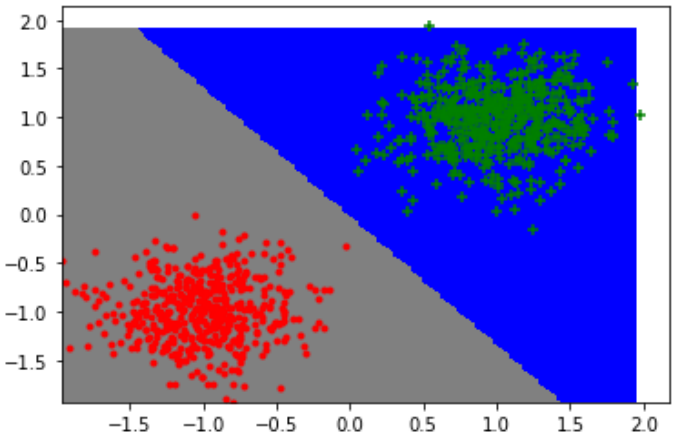

*Itération 1*



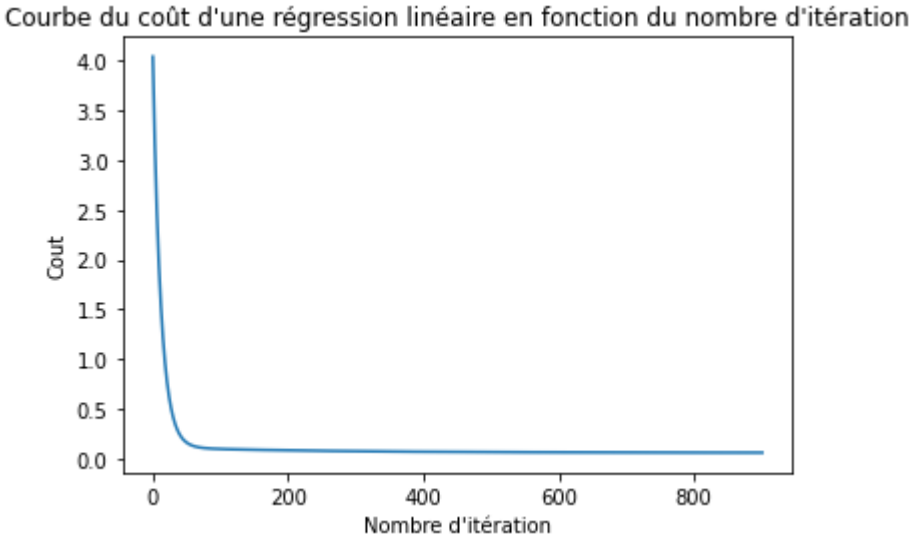

*Itération 151*



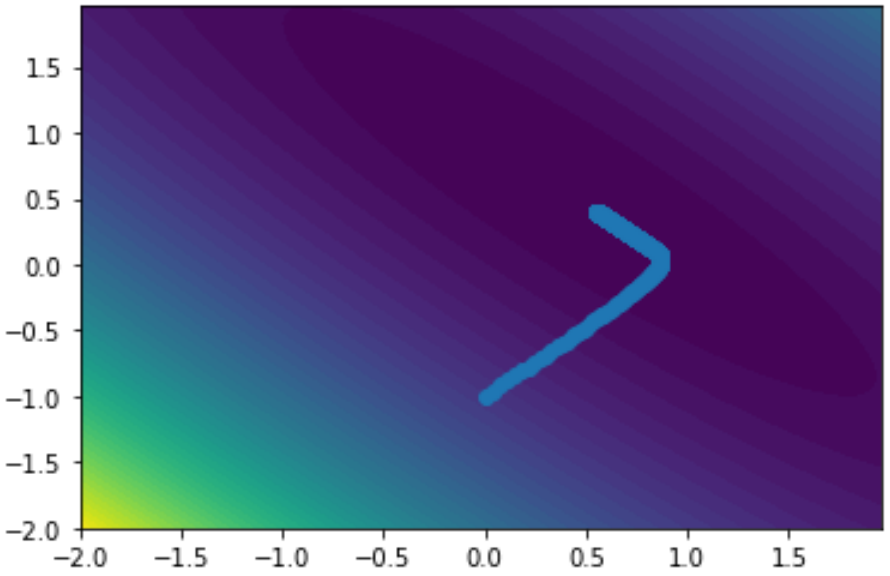

*Itération 201*




*Itération 901*




On peut voir que très rapidement on converge vers le minimum de la fonction de coût.



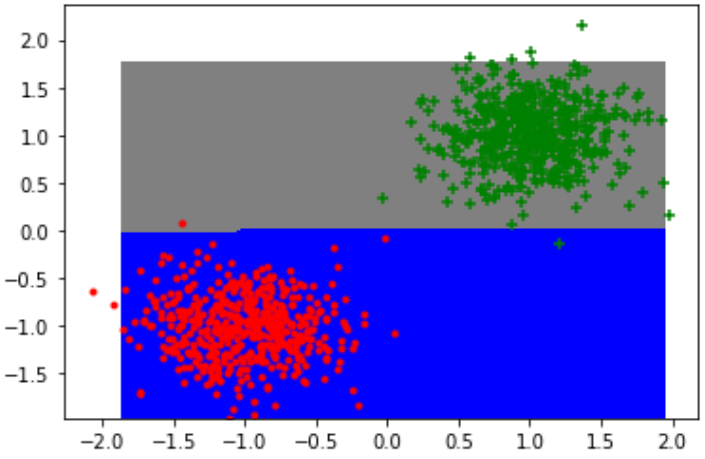

On peut également visualiser la fonction de coût dans l'espace des poids selon les deux dimensions du problème.
On suit la trajectoire du vecteur w.

La fonction score nous affirme qu'en test, il n'y a aucune étiquette qui est mal prédite.

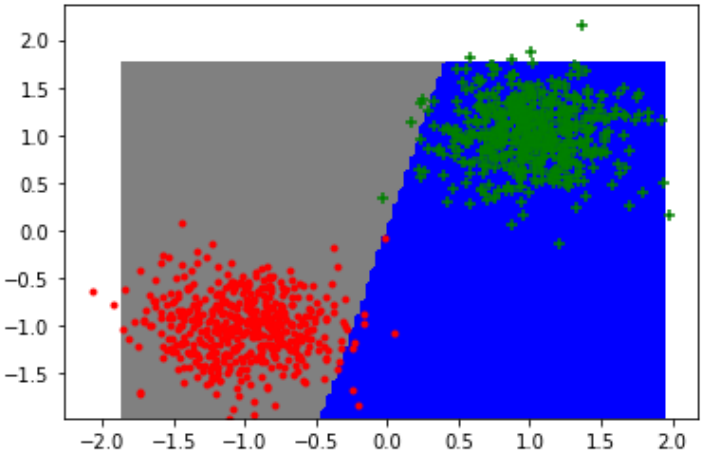
Par ailleurs, on peut noter que la fonction de coût *mse* n'est pas idéale pour la classification : un point peut être bien classé mais comptera beaucoup dans la fonction de coût car il sera loin de la frontière. La fonction de coût au moindre carré est utilisée d'avantages lorsqu'il faut établir  une relation linéaire entre plusieurs variables.

## Descente de gradient : régression logistique

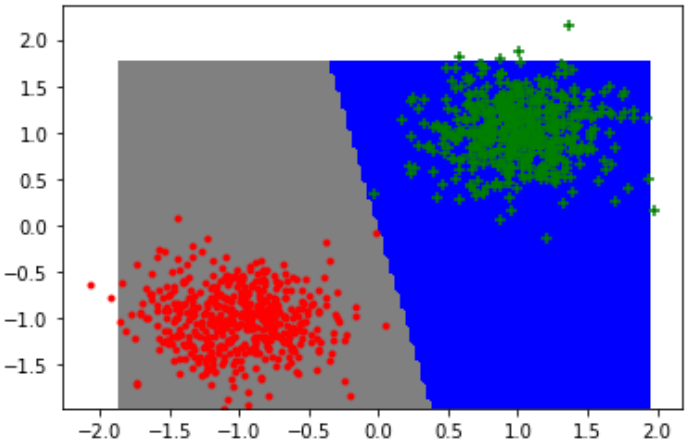
La fonction de régression logistique est totalement adaptée à la classification.
Observons l'évolution de sa frontière de décision avec le w0 initial à  [0, -1].



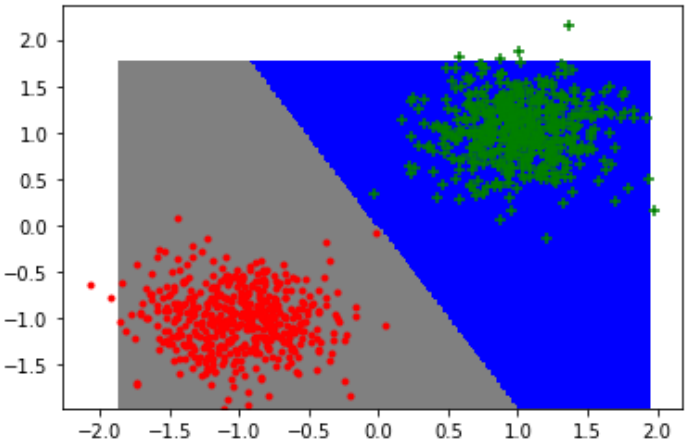
*Itération 1*



*Itération 151*

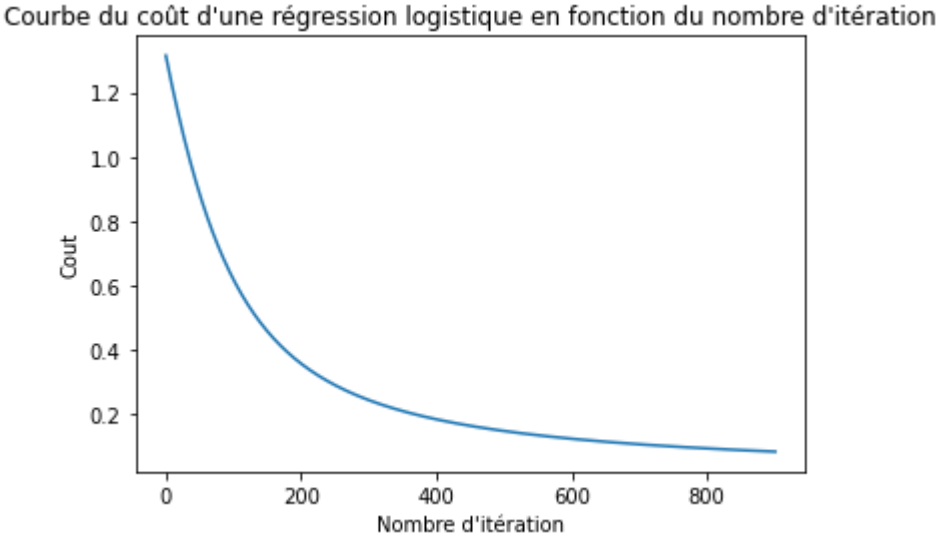


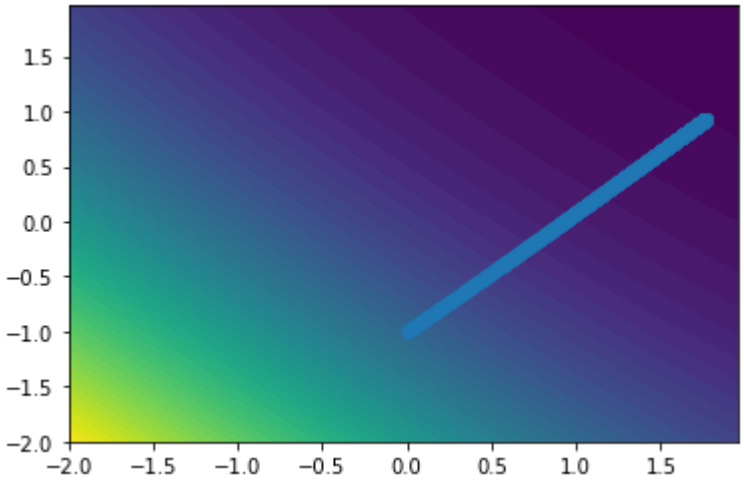
*Itération 201*



*Itération 901*

Cette courbe nous permet de voir que plus on fait d'itération, plus on s'approche du minimum de la fonction de coût.



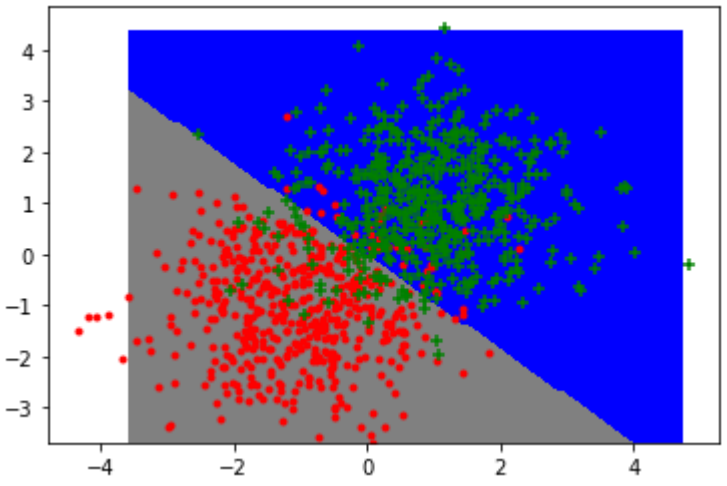


On peut également visualiser la fonction de coût dans l'espace des poids selon les deux dimensions du problème.
On suit la trajectoire du vecteur w.

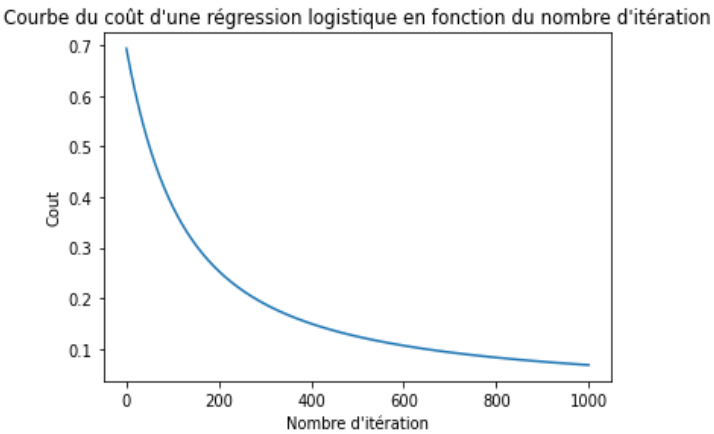
En test, il n'y a aucune étiquette qui est mal prédite.

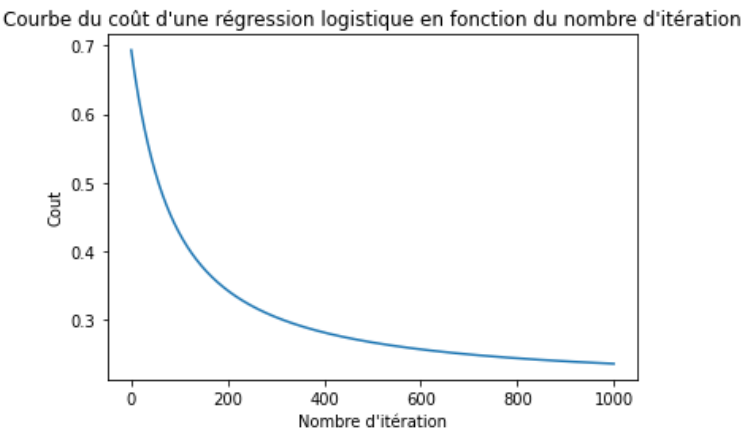
## Augmentation du bruit

Prenons l'exemple de la régression logistique (cela fonctionne de la même manière avec la régression linéaire).
Ajoutons du bruit, par exemple prenons epsilon = 1.



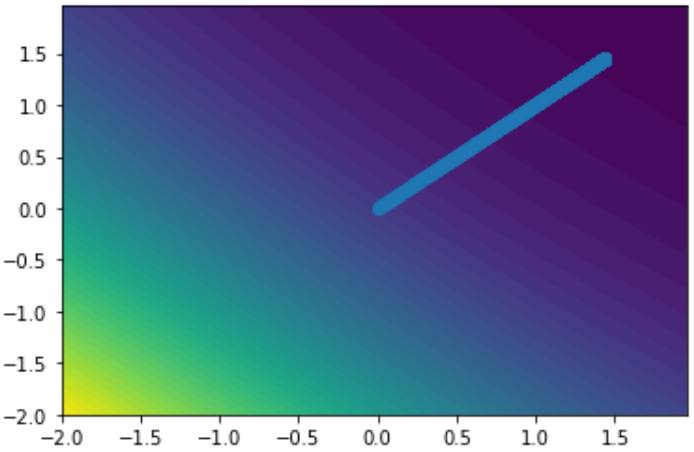
Nous pouvons observer que les données ne sont pas linéairement séparables. C'est pour cela que même au bout de 1 000 itérations, la droite ne sépare pas bien les données.

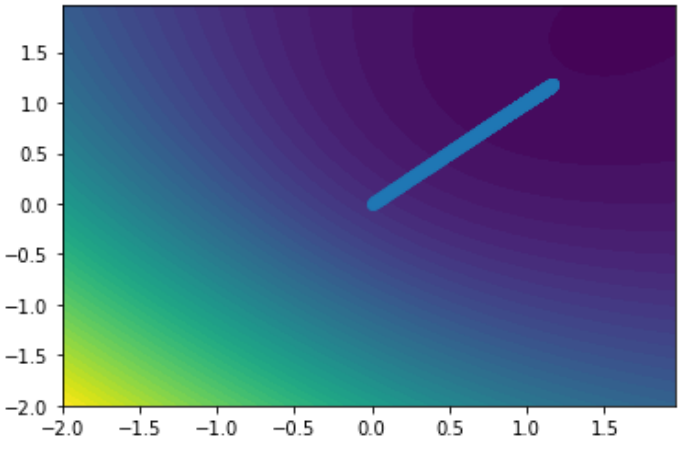


A gauche, nous retrouvons la courbe du coût d'une régression logistique pour un epsilon égale à 0.1 et à droite pour un epsilon égale à 1. On peut remarquer que lorsqu'il y a du bruit, il faut plus d'itération pour minimiser la fonction de coût.

Nous pouvons aussi le voir grâce au chemin parcouru par le vecteur w, on est plus loin du minimum de la fonction à droite :



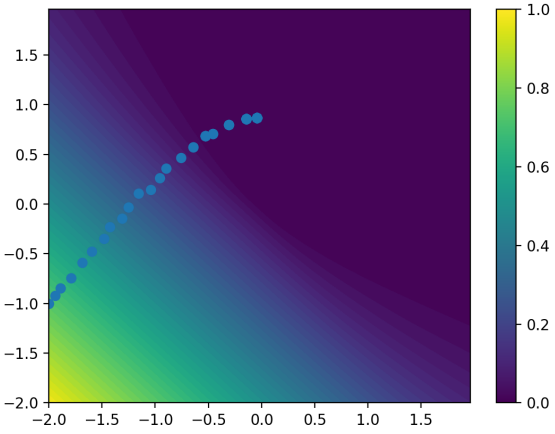

De plus, avec du bruit, le nombre d'étiquettes mal prédites tourne autour de 15-20 étiquettes, contre 0 sans bruit.

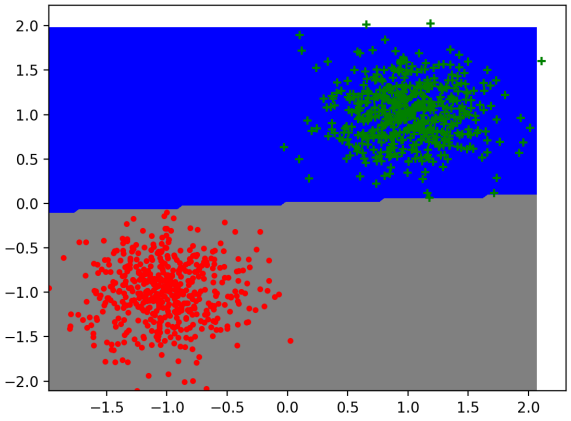
# TME 4 - Perceptron

## Perceptron et classe linéaire

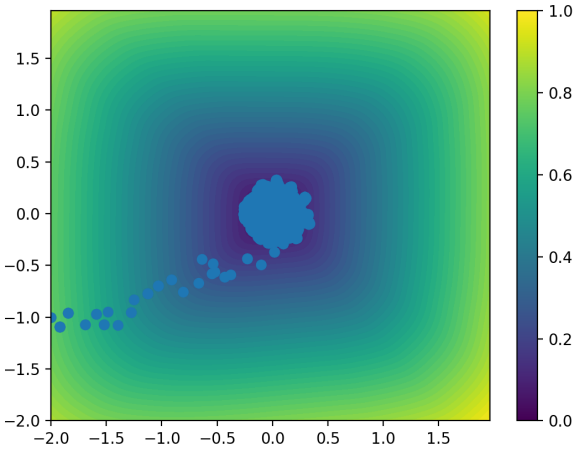
La classe Linéaire a été écrite de la même manière que la classe Regression du TME 3. Le constructeur prend directement la fonction de coût et la fonction de gradient. La fonction score renvoie le taux de bonne classification.

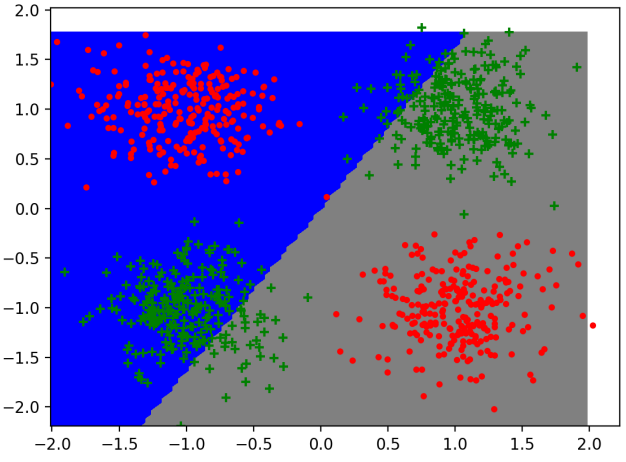
Test de la classe sur un ensemble de points générés selon 2 gaussiennes :



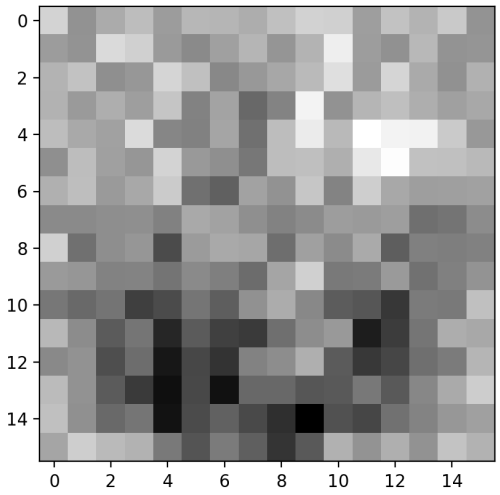

Test de la classe sur un ensemble de points générés selon 4 gaussiennes :



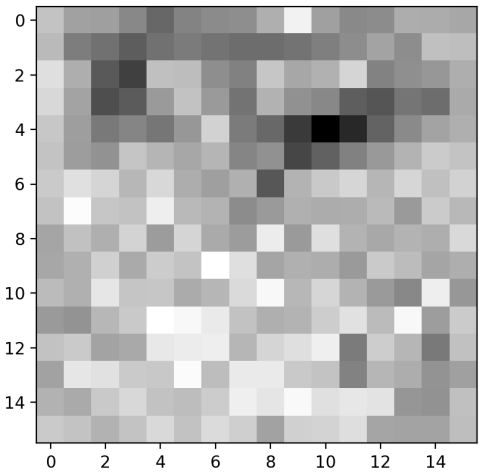



## Données USPS

Matrice de poids obtenue après avoir entraîné le perceptron sur la classe des 6 et 9 :




Matrice de poids obtenue après avoir entraîné le perceptron sur la classe des 6 et celle de tous les autres chiffres :



## Mini-batch et descente stochastique

Toutes ses variables ont permis de construire une fonction factorisée :

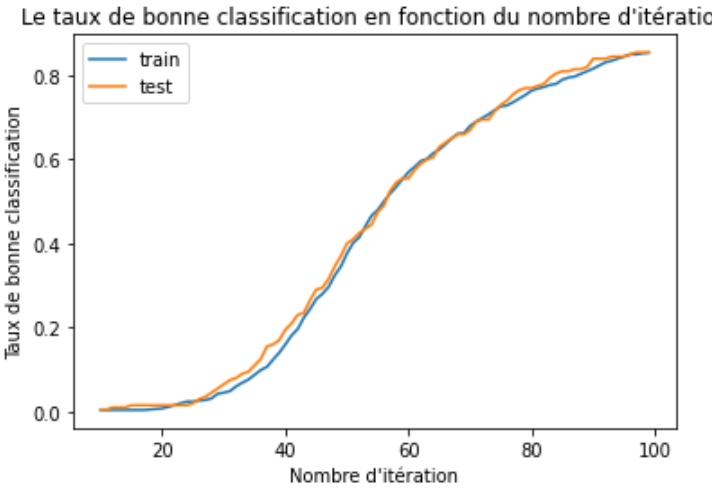
```
listesFusionnées = list(zip(datax, datay))
nb_elem = len(datay)//batch
random.shuffle(listesFusionnées)
datax, datay = zip(*listesFusionnées) #On obtient nos données mélangées
datax_batch = [datax[(k*nb_elem):min(((k+1)*nb_elem, len(datay)))] for k in range (batch+1)]
datay_batch = [datay[(k*nb_elem):min(((k+1)*nb_elem, len(datay)))] for k in range (batch+1)]
```

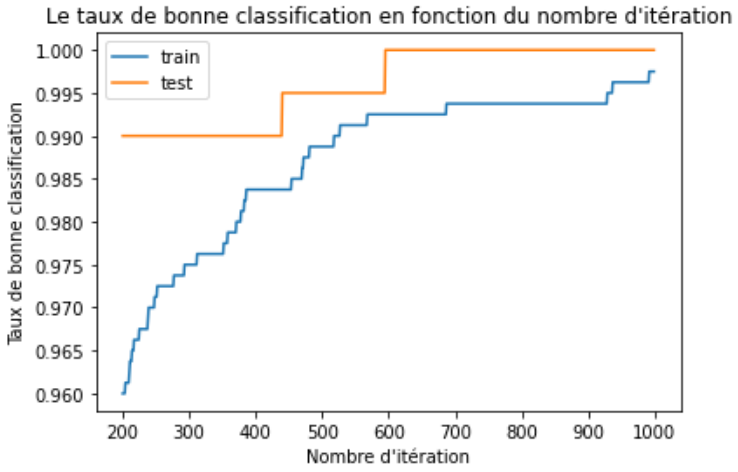
Si batch = 1, nous sommes dans le cas d'une descente de gradient batch.
Si batch = N, soit la taille des données, nous somme dans le cas d'une descente de gradient stochastique.
Si batch = k, nous sommes dans le cas d'une descente de gradient mini-batch avec k paquets de nb_elem exemples.

Nous allons comparer la vitesse de convergence sur les données 2D des deux gaussiennes en augmentant plus ou moins le bruit et en modifiant le type de descente de gradient.

**Descente de gradient batch**

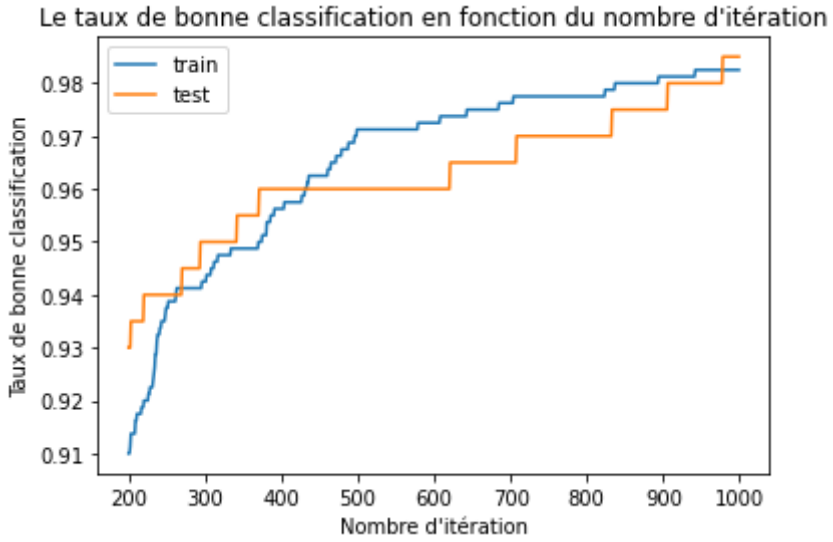
Pour un bruit égale à 0.1, on obtient :




Les deux courbes augmentent : plus on fait d'itération, moins on fait d'erreurs.
On observe qu'en test on converge vers les 600 itérations. En apprentissage comme en test, on atteint les 100% de bonnes classifications. Les données sont linéairement séparables, ainsi on finira par converger.

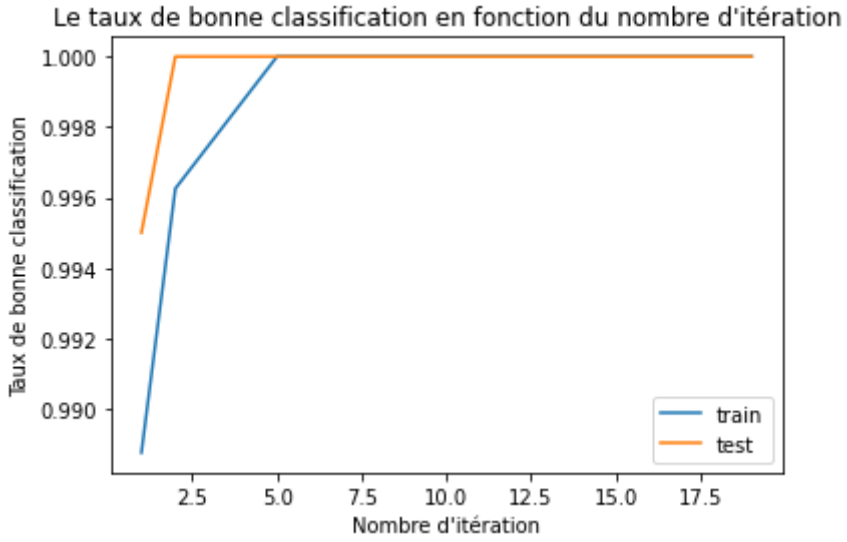
Pour un bruit égale à 0.5, on obtient :




La convergence est plus lente. Les 100% de bonnes classifications ne sont pas atteints, ni en apprentissage, ni en test, il faudrait peut être quelques itérations supplémentaires pour que cela puisse se produire.

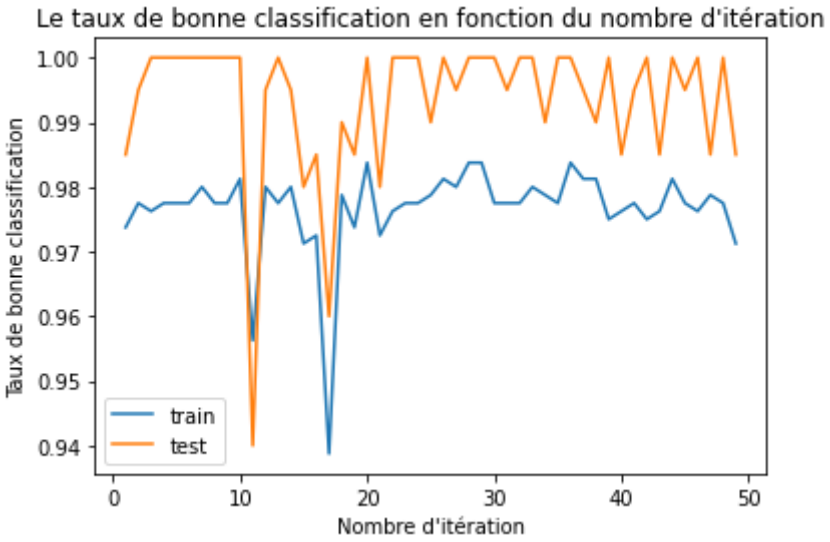
**Descente de gradient stochastique**

Pour un bruit égale à 0.1, on obtient :




En descente stochastique, lorsque les données sont linéairement séparables, on converge vers 100% de bonne classification en test et en apprentissage beaucoup plus rapidement qu'en batch, autour des 5 itérations. En effet, comme on modifie la frontière à chaque point, on peut plus rapidement trouver la bonne frontière.

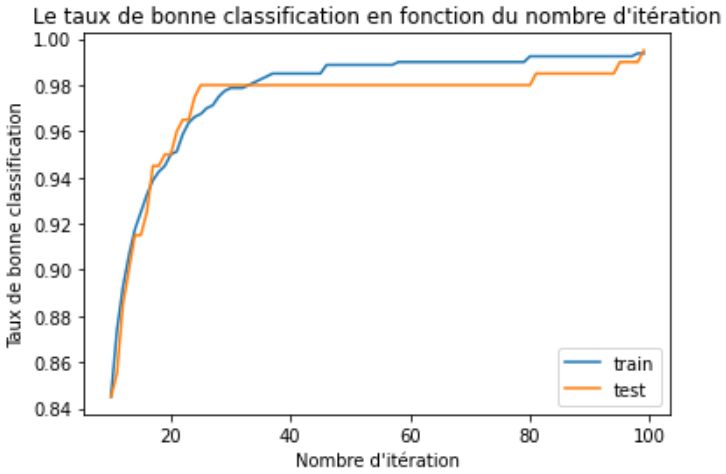
Pour un bruit égale à 0.5, on obtient :



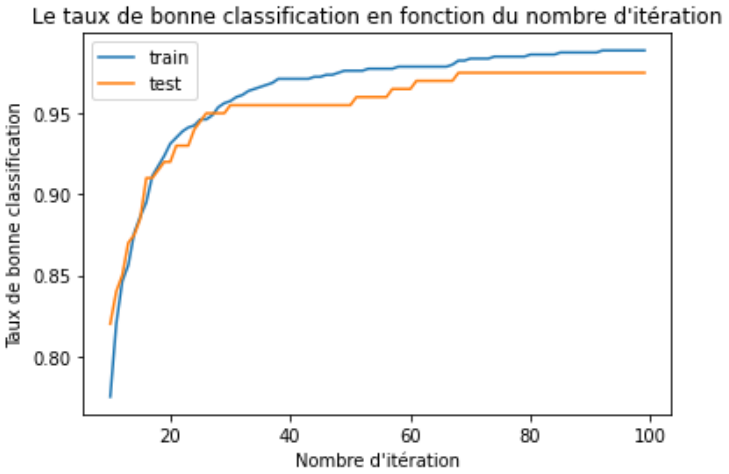

Lorsqu'on ajoute du bruit, les résultats fluctuent énormément car corriger un point mal placé peut en induire d'autres en erreur, alors qu'ils étaient de base bien placé.

**Descente de gradient pour 10 paquets**

Pour un bruit égale à 0.1, on obtient :

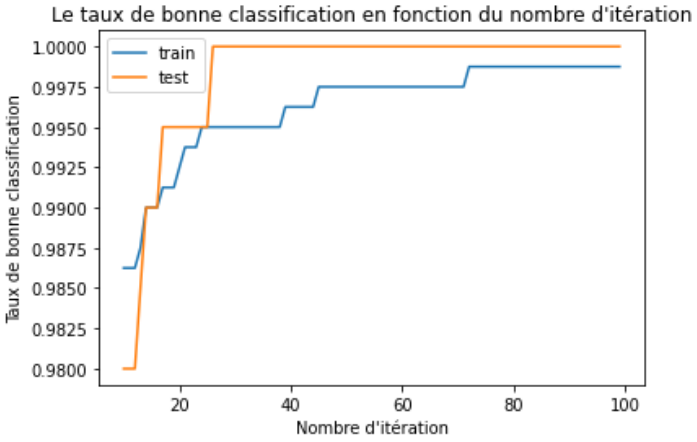



Pour un bruit égale à 0.5, on obtient :

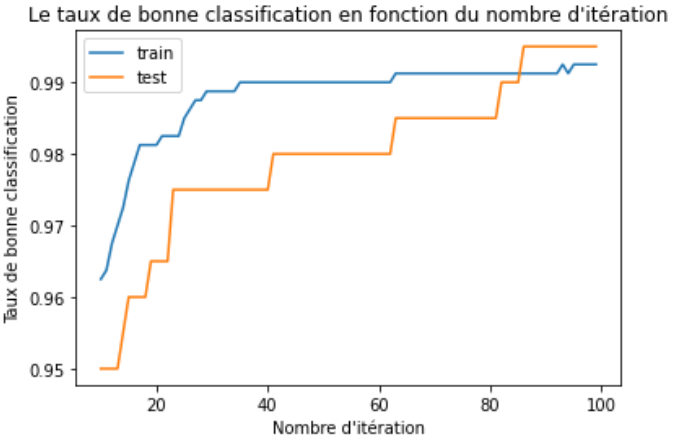


### Descente de gradient pour 40 paquets

Pour un bruit égale à 0.1, on obtient :



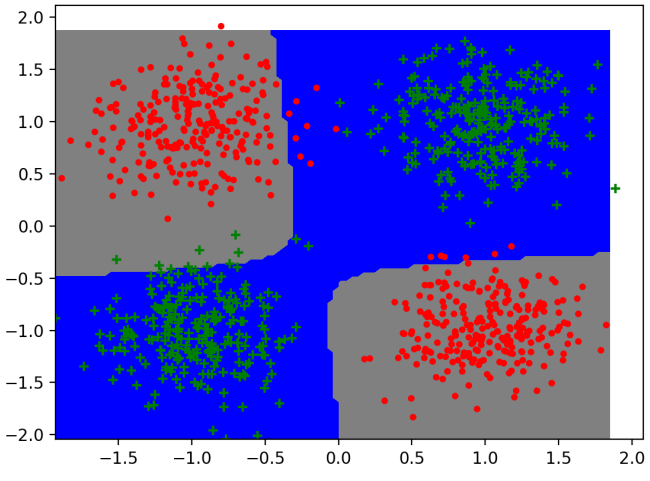

Pour un bruit égale à 0.5, on obtient :



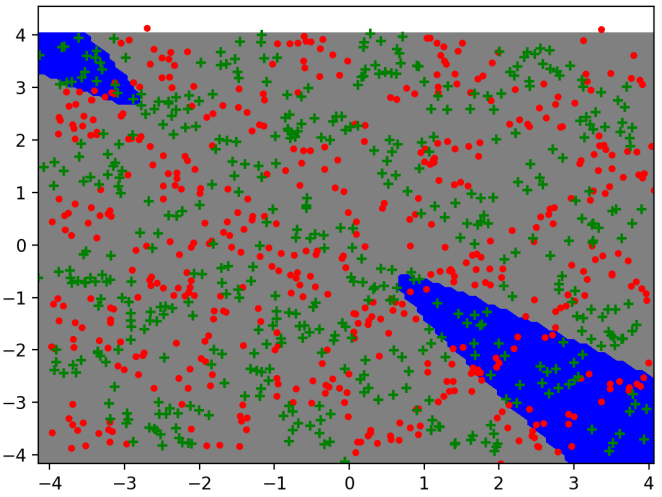

## Projection et pénalisation

### Projection polynomiale de degré 2

test sur les données (de test) de type 1 :



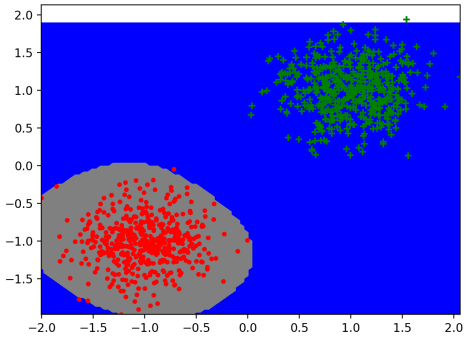

test sur les données (de test) de type 2:



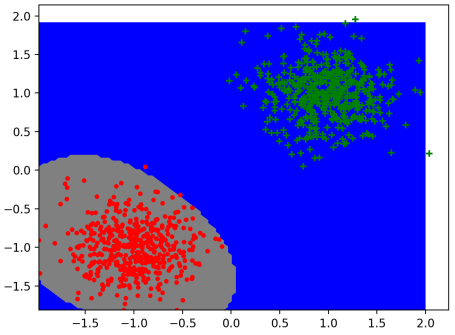

<u>Analyse :</u>
Avec une projection polynomiale de degré 2 sur les données de type 1, on arrive à séparer les données. Cependant, cette projection ne permet pas de séparer les données de type 2.

**Projection gaussienne**

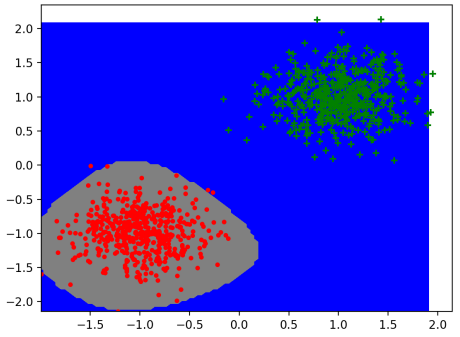
On fixe sigma à 0.35 et on fait varier le nombre de points de la base.



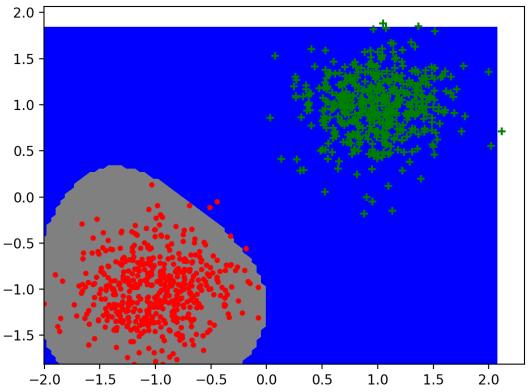
20 points



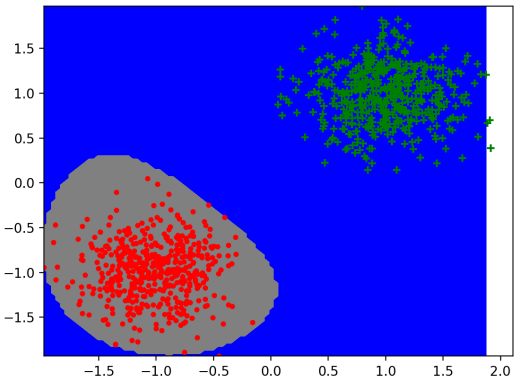
200 points



2 000 points



20 000 points



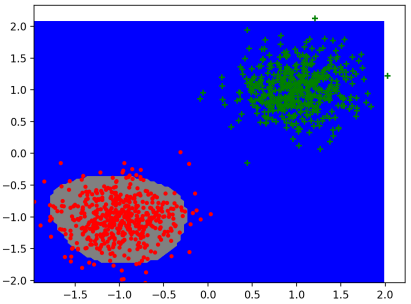
200 000 points

<u>analyse :</u>

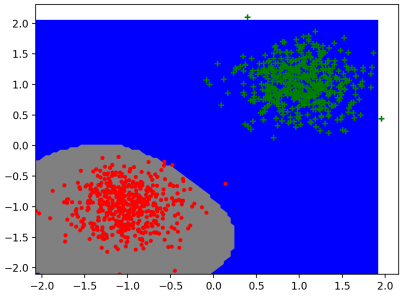
Plus le nombre de points de la base est grand, plus la projection est précise. Il vaut mieux prendre beaucoup de points.

Maintenant on fait varier sigma. Les points de la base sont au nombre de 500.

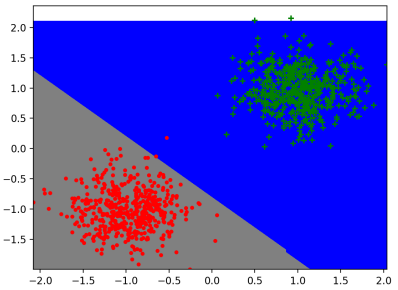
On fait varier sigma.



sigma = 0.1



sigma = 0.4



sigma = 3

<u>analyse:</u>

Plus sigma est petit, plus le nombre de points pris en compte est petit.
Plus sigma est grand, plus le nombre de points pris en compte est grand.

Le sigma optimal semble être compris entre 0.1 et 0.4.

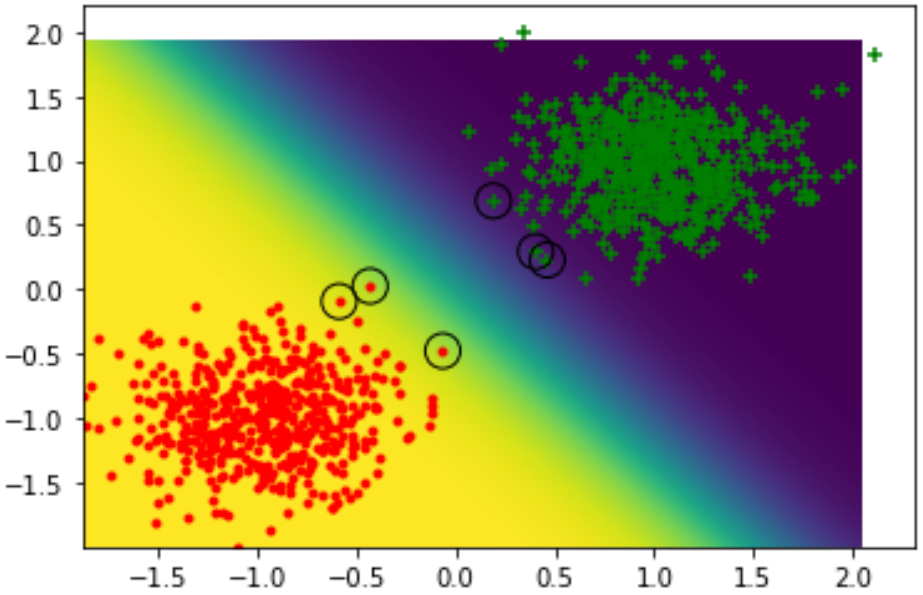
**Pénalisation :**

$\alpha$ contrôle la marge : plus $\alpha$ grand, plus les vecteurs de support sont éloignés de la frontière.

Plus $\lambda$ est grand, plus l'expressivité est pénalisée.

# TME 5 - SVM

## SVM et Grid Search




Voici un exemple sur le jeu des données 2d artificielles de 2 gaussiennes. En utilisant un SVM linéaire on obtient l'image ci-dessus. On peut repérer les vecteurs supports entourés qui sont au nombre de 6 dans cet exemple.

Pour trouver le meilleur noyau avec le meilleur paramétrage pour un jeu de données, nous allons utiliser une GridSearch afin de tester toutes les combinaisons possibles puis comparer les scores.
Pour cela nous avons une classe dont le constructeurs prends en paramètre une liste de *kernel*, une liste de degré pour le kernel polynomial et les valeurs que peut prendre gamma, valeurs qui correspondent aux coefficient pour les kernels : rbf, poly et sigmoid.
La classe GridSearch contient une fonction *test_model* qui pour un model donnée, des données d'apprentissage et de tests renvoie par cross validation (5 splits) le score en train, en test et le nombre de vecteurs supports ainsi qu'une fonction score qui teste toutes les combinaisons possibles et renvoie un dictionnaire avec les résultats.
Testons sur les données 2d artificielles (cela fonctionne de la même manière pour la reconnaissance de chiffres).

Avec ces paramètres :

```
kernels = {'linear', 'poly', 'rbf', 'sigmoid'} #les kernels testés
n_degres = 5 #on testera sur les degrés de 1 à 5 pour le kernel polynomial
gamma = {'scale', 'auto'}
gs = GridSearch(kernels, n_degres, gamma) #On crée un GridSearch
gs.score( x_train,  y_train.ravel(), x_test, y_test.ravel()))
```

On obtient :

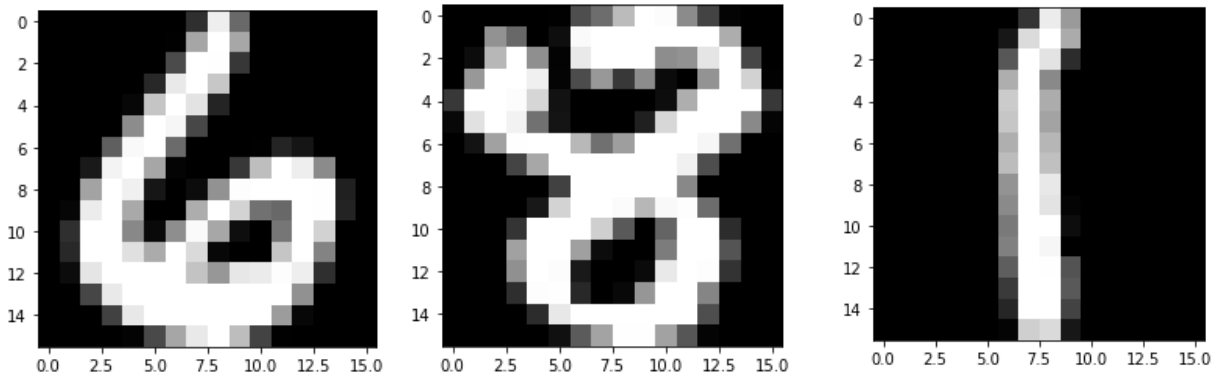
```
{('sigmoid', 'scale'): ([1.0, 1.0, 1.0, 1.0, 1.0], [1.0, 1.0, 1.0, 1.0, 1.0], 8),
('sigmoid', 'auto'): ([1.0, 1.0, 1.0, 1.0, 1.0], [1.0, 1.0, 1.0, 1.0, 1.0], 8),
('rbf', 'scale'): ([1.0, 1.0, 1.0, 1.0, 1.0], [1.0, 1.0, 1.0, 1.0, 1.0], 12),
('rbf', 'auto'): ([1.0, 1.0, 1.0, 1.0, 1.0], [1.0, 1.0, 1.0, 1.0, 1.0], 13),
('poly', 'scale', 1): ([1.0, 1.0, 1.0, 1.0, 1.0], [1.0, 1.0, 1.0, 1.0, 1.0], 8),
('poly', 'scale', 2): ([0.525, 0.46875, 0.50625, 0.51875, 0.51875], [0.575, 0.575, 0.525,
0.4, 0.575], 794),
('poly', 'scale', 3): ([1.0, 1.0, 1.0, 1.0, 1.0], [1.0, 1.0, 1.0, 1.0, 1.0], 38),
('poly', 'scale', 4): ([0.5, 0.475, 0.51875, 0.50625, 0.50625], [0.4, 0.5, 0.5, 0.45,
0.5], 795),
('poly', 'scale', 5): ([1.0, 1.0, 0.99375, 1.0, 0.99375], [1.0, 1.0, 1.0, 1.0, 1.0], 90),
('poly', 'auto', 1): ([1.0, 1.0, 1.0, 1.0, 1.0], [1.0, 1.0, 1.0, 1.0, 1.0], 8),
('poly', 'auto', 2): ([0.51875, 0.44375, 0.50625, 0.5125, 0.525], [0.55, 0.6, 0.525,
0.375, 0.575], 793),
('poly', 'auto', 3): ([1.0, 1.0, 1.0, 1.0, 1.0], [1.0, 1.0, 1.0, 1.0, 1.0], 34),
('poly', 'auto', 4): ([0.5, 0.475, 0.51875, 0.50625, 0.50625], [0.375, 0.5, 0.5, 0.475,
0.475], 794),
('poly', 'auto', 5): ([1.0, 1.0, 0.99375, 1.0, 0.99375], [1.0, 1.0, 1.0, 1.0, 1.0], 76),
'linear': ([1.0, 1.0, 1.0, 1.0, 1.0], [1.0, 1.0, 1.0, 1.0, 1.0], 6)}
```

En clé du dictionnaire nous avons les paramètres, la première valeur correspond au score sur chaque split de train, la deuxième le score sur les tests et la dernière valeurs aux nombres de vecteurs supports.
On peut observer une différence flagrantes sur le nombre de vecteurs supports, notamment les polynômes de degrés pair qui en ont énormément comparé aux autres.

Le taux de bruit choisi était faible. Lorsqu'on l'augmente, comme prévu, le nombre de vecteurs supports augmentent également.

## Apprentissage multi-classe

Nous allons utiliser ici les méthodes One-versus-one et One-versus-all sur la reconnaissances des chiffres.
Voici des exemples des USPS qui sont des données de chiffres manuscrits :



La classe MultiClass prend en paramètre un kernel, un degré, un gamma et un boolean qui spécifie si on utilise la méthode One-versus-one ou One-versus-all.
Si on ne précise par les valeurs de *n_degree*, *gamma* et du booléen, on leur attribue des valeurs par défauts.
La fonction *score* calcule la moyenne de bonne classification.
Testons avec différents kernel et différentes méthodes :

| **Paramétrage du kernel** | **Méthode** | **Score de bonne classification** |
|---|---|---|
| Linéaire | One-versus-one | 0.9267563527653214 |
| Sigmoid | One-versus-one | 0.7344294967613353 |
| Polynomial degré 2 | One-versus-one | 0.9382162431489786 |
| Polynomial degré 3 | One-versus-one | 0.9382162431489786 |
| Linéaire | One-versus-all | 0.9063278525161933 |
| Sigmoid | One-versus-all | 0.5640259093173892 |
| Polynomial degré 2 | One-versus-all | 0.9481813652217239 |
| Polynomial degré 3 | One-versus-all | 0.9481813652217239 |

On peut observer que la sigmoid n'est pas un bon choix de kernel pour la reconnaissance des chiffres. De plus, le score avec kernel polynomial converge dès le degré 2, que se soit pour la méthode One-versus-one ou One-versus-all.
Pour les kernels linéaires et polynomials, les deux méthodes donnent des résultats très similaires.