Dmytro SAVCHUK

IA
&
MACHINE LEARNING

Rapports de TP

Dmytro SAVCHUK

TP n°1 :

L'algorithme des K plus proches voisins

L'objectif de ce 1er TP est de mettre en application l'un des algorithmes de classification les plus utilisés en Machine Learning : l'algorithme des K plus proches voisins.

Le principe de fonctionnement de cet algorithme est le suivant : pour chaque nouvelle donnée, l'algorithme va déterminer dans quelle classe la ranger, en fonction des K données les plus proches de celle que l'on souhaite classer. L'algorithme choisira alors de classer cette donnée dans la classe majoritaire sur ses K plus proches voisins.

La mise en œuvre de cet algorithme se fait de la façon suivante :

1) On crée le modèle *kppv*, associé à la méthode des K plus proches voisins qui prend en paramètre la valeur de K :

   kppv = neighbors.KNeighborsClassifier(K)

2) On réalise l'apprentissage du modèle (sur les données dites d'entrainement) sur les bases d'apprentissage X et Y :

   kppv.fit(X,Y)

   où X contient toutes les données à classer, et Y les classes attribuées à chaque donnée de X.

3) A partir de l'apprentissage effectué, on réalise la prédiction des nouvelles valeurs, soit la prédiction des classes Y sur les données de test Xtest :

   Ypred = kppv.predict(Xtest)

I. Test de l'algorithme dans le plan

Dans un premier temps, nous allons nous intéresser à un exemple basique qui consiste à placer des points de différentes couleurs dans un plan en 2D, puis, à l'aide de l'algorithme, prédire la couleurs attribuée à de nouveaux points.

Ici, les classes utilisées seront donc des classes de couleurs : rouge, bleu, vert et magenta.

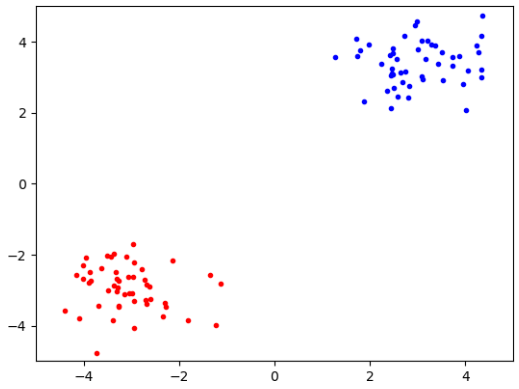
Pour la phase d'apprentissage, les données seront générées par des clics de souris, avec donc un choix de 4 couleurs en fonction du clic utilisé.

Les données de test seront ensuite générées par le programme, et disposés dans le plan de telle sorte à former une grille.

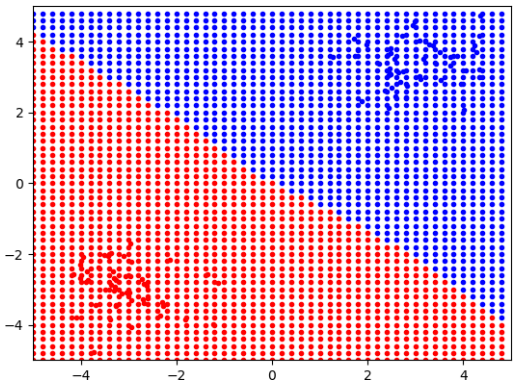
Le but est donc, pour chaque point de la grille, d'attribuer une couleur à chaque point de la grille, à l'aide de l'algorithme des K plus proches voisins.

Pour cela, nous allons tester différents cas de figure pour l'algorithme.

1) Exemple 1 : répartition équitable et linéairement séparable (2 classes, K=1)



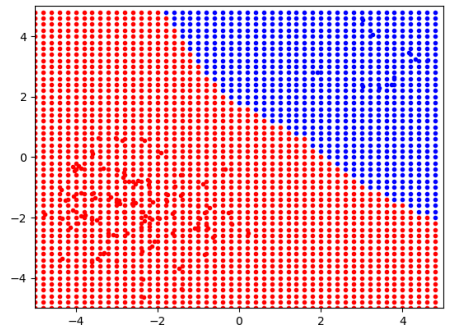
Données d'apprentissage



Prédiction des données de test

Dans cet exemple, où les points bleus et rouges sont clairement répartis équitablement (50 points rouges, 50 points bleus) de chaque côté du plan, on observe que les données de test sont séparées en 2 zones, délimitées par une frontière de séparation qui sépare donc le plan en 2 moitiés quasiment égales.

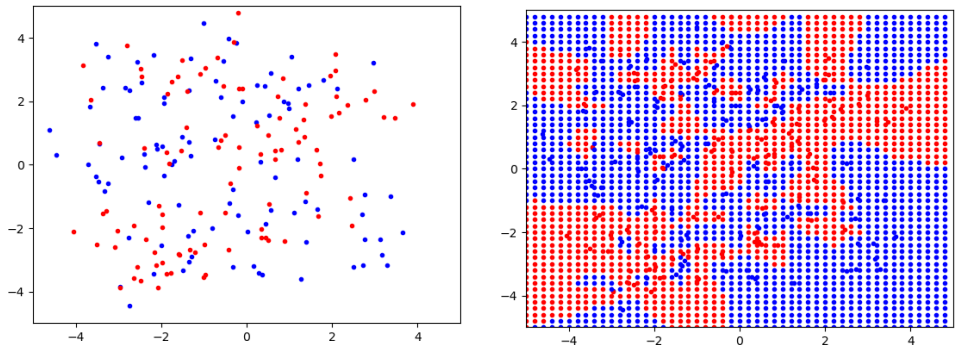
2) Exemple 2 : répartition non équitable et linéairement séparable (2 classes, K=1)



Cette fois-ci, les points bleus et rouges sont toujours répartis de chaque côté du plan, mais dans des proportions différentes (10 points bleus, 90 points rouges).

On observe alors que la frontière de séparation forme une parabole, autour de la zone de prédiction des points bleus. Cette zone est d'ailleurs nettement plus petite que la zone rouge, du fait d'un bien plus faible nombre de points bleus.

3) Exemple 3 : répartition équitable non séparable (2 classes, K=1)

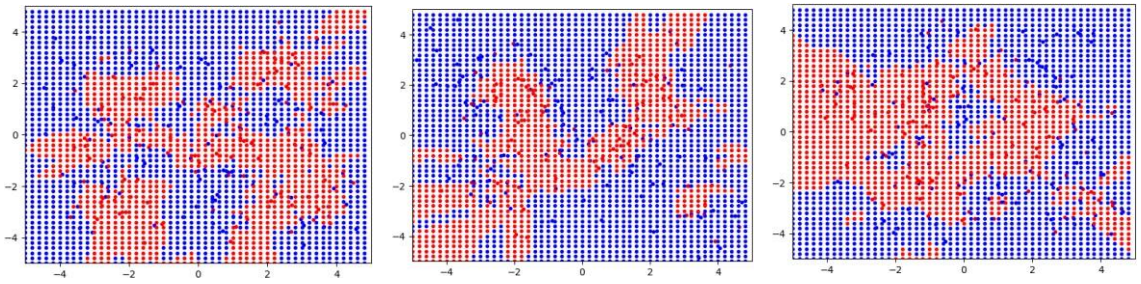


Dans cet exemple, il y a autant de points rouges que de points bleus, mais ceux-ci sont mélangés entre eux, et non plus séparés distinctement comme dans les exemples précédents.

On constate alors que l'algorithme crée un grand nombre de zones, en fonction de la couleur de chaque point d'apprentissage. Cela est dû au fait que K=1, ainsi l'algorithme utilise uniquement le plus proche voisin de chaque point pour lui prédire sa couleur.

4) Exemple 4 : répartition équitable non séparable (2 classes, K>1)

Pour étudier l'influence de K sur l'algorithme, on peut alors recommencer l'exemple précédent en faisant varier la valeur de K.



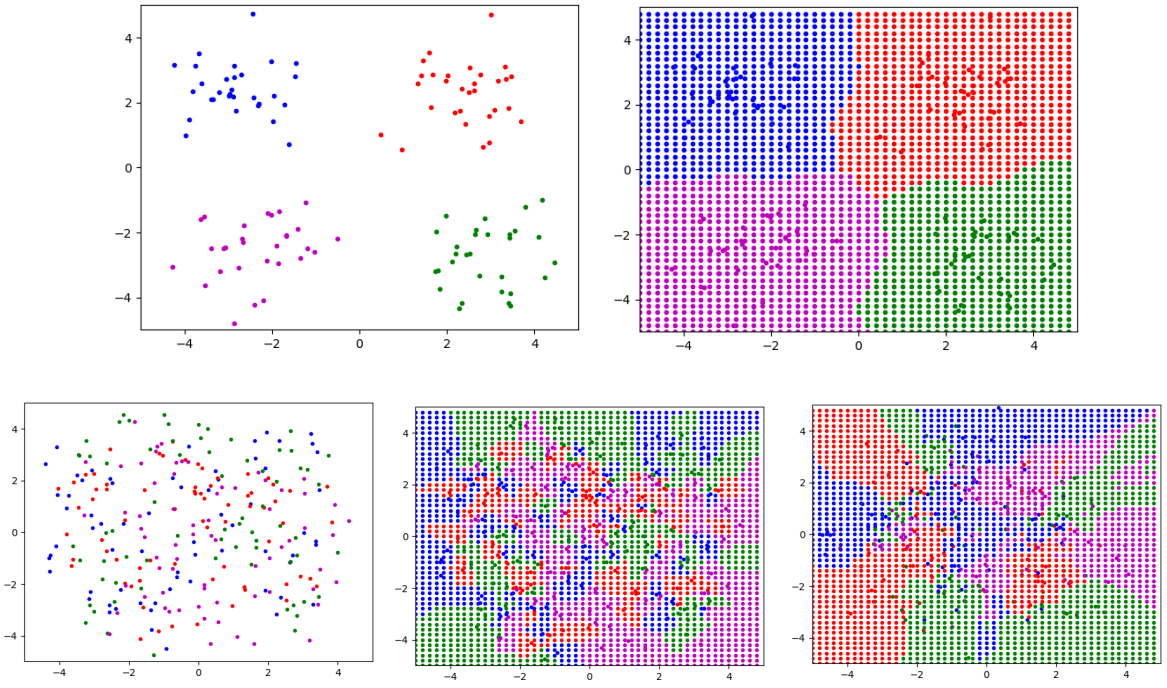
K=5 *K=10* *K=15*

On constate donc que plus K est grand, moins on observe de régions. En effet, l'algorithme se base alors sur un plus grand nombre de voisins pour réaliser ses estimations, et choisit ensuite la couleur majoritaire, qui n'est donc pas forcément la couleur de chacun des voisins. La prédiction se fait donc par régions, et non plus par points comme dans l'exemple précédent avec K=1.

Néanmoins, il faut être vigilant sur le réglage de K, et éviter de faire du surapprentissage en choisissant une valeur de K trop élevée.

5) Exemple 5 : répartition équitable (4 classes, K=1)

Pour ce dernier exemple, nous allons tester l'algorithme avec un plus grand nombre de classes, dans un premier temps avec une répartition linéairement séparable, puis non séparable dans un second temps, mais toujours avec une répartition équitable:



K=1 *K=15*

On constate donc, que ce soit dans le cas séparable ou non séparable, que l'algorithme se comporte de la même façon que lorsque nous utilisions 2 classes.

Dans le cas de la répartition séparable, l'algorithme découpe le plan en 4 régions à peu près égales, et dans le cas non séparable l'algorithme va créer un certains nombre de régions, selon le nombre de K plus proches voisins que l'on souhaite calculer.

II. Application à la reconnaissance de caractères

La deuxième partie de ce TP consiste à appliquer l'algorithme des K plus proches voisins à un exercice de reconnaissance de caractères manuscrits.

La base de données totale comporte 70000 exemples de caractères : 60000 exemples d'apprentissage, et 10000 exemples de test.

Dans un premier temps, on crée un classifieur *Kppv* à partir de 10000 images de la base d'apprentissage. Ainsi, la taille du modèle utilisé est :

$$\begin{aligned} Taille\ du\ modèle &= nb\ images * (nb\ pixels + 1) * 8 \\ &= 10000 * (28 * 28 + 1) * 8 \\ &= 62{,}8\ Mo \end{aligned}$$

L'objectif est de déterminer le meilleur classifieur possible pour la reconnaissance de caractères, c'est-à-dire le classifieur avec le taux d'erreur le plus faible. Ainsi, pour chaque image X, l'algorithme va comparer la valeur de Y associée avec la valeur de Y prédite.

Pour réaliser cette recherche, on commence par créer, à partir du set de données d'apprentissage, un set de données destiné à la validation (10000 images) :

```
# chargement de la base mnist:
Xtrain, Xtest, ytrain, ytest = load_mnist(11000, 1000)
Xapp = Xtrain[:10000,:]
Yapp = ytrain[:10000]
Xval = Xtrain[10000:,:]
Yval = ytrain[10000:]
```

Ainsi, nous disposons de :

- 10000 images pour l'apprentissage
- 1000 images pour la validation
- 1000 images pour le test

Où chaque image de chaque jeu de données est unique. Cette étape est primordiale puisqu'il est nécessaire de réaliser l'évaluation des différents classifieurs sur un jeu de données différent de celui utilisé pour le test final, avec le classifieur que l'on aura choisi.

Comme notre modèle dépend uniquement du paramètre K, rechercher le meilleur classifieur revient à chercher la valeur de K pour lequel l'algorithme commet le moins d'erreur.

Nous allons donc, à travers une boucle for, réaliser l'apprentissage d'un jeu de données de 10000 images (tirées aléatoirement dans la base d'apprentissage), puis réaliser la prédiction sur les données de validation, en faisant varier K de 1 jusqu'à 10.

```
Kmax = 0
Error_min = 0.1
for K in range(1,11):
    kppv = neighbors.KNeighborsClassifier(K)
    kppv.fit(Xapp, Yapp)
    Ypred = kppv.predict(Xval)
    """ Evaluation de l'erreur de prédiction --> risque du classifieur """
    Error = np.mean(Ypred != Yval)
    print("Erreur de validation (K=",K,") = ", Error)
    if Error < Error_min :
        Error_min = Error
        Kmax = K

print("Meilleur K : ", Kmax)
```

Etant donné que les données d'apprentissage utilisées sont issues d'un tirage aléatoire sur la base d'apprentissage, les résultats varient d'une simulation à l'autre, mais globalement les résultats sont de l'ordre de :

Valeur de K	**Pourcentage d'erreur**
1	5,3 %
2	5,4 %
3	5,2 %
4	5,0 %
5	5,1 %
6	5,2 %
7	5,3 %
8	5,2 %

9	5,3 %
10	5,3 %

Ici, le meilleur classifieur obtenu est celui avec K=4.

Une fois ce meilleur classifier obtenu, on réalise un nouvel apprentissage, cette fois-ci sur la base d'apprentissage complète (60000 images), en utilisant la meilleure valeur de K obtenue (c'est-à-dire la plus faible).

Puis on réalise l'évaluation du modèle sur l'échantillon de test (10000 images), qui contient des images que le classifieur n'a encore pas rencontré.

```
Xtrain, Xtest, ytrain, ytest = load_mnist(60000, 1000)

kppv = neighbors.KNeighborsClassifier(Kmax)
kppv.fit(Xtrain, ytrain)
Ypred = kppv.predict(Xtest)

""" Evaluation de l'erreur de prédiction --> risque du classifieur """
Error = np.mean(Ypred != ytest)
print("Erreur de test = ", Error)
```

Finalement, on obtient un résultat final autour de 2,4 % d'erreur avec notre meilleur classifieur, ce qui démontre bien l'efficacité de l'algorithme des K plus proches voisins, et sa simplicité de mise en œuvre.

Enfin, on constate également un temps de calcul beaucoup plus long lors de la dernière étape, du fait que l'on travaille avec la base d'apprentissage complète : pour chaque image, l'algorithme recherche les K plus proches voisins parmi les 60000 images de la base d'apprentissage, ce qui donne un temps de calcul total relativement long.

Dmytro SAVCHUK

TP n°2 :

L'algorithme SVM (Support Vector Machine)

L'objectif de ce TP est de mettre en application l'algorithme de classification SVM (Support Vector Machine).

Le but de cet algorithme est de trouver le meilleur hyperplan de séparation linéaire H, pour un jeu de données séparé en plusieurs classes :

$$H = \{\boldsymbol{x} | \boldsymbol{w}^T \boldsymbol{x} + b = 0\}$$

A cet hyperplan sont associées des marges Δ, qui correspondent aux exemples xi de chaque catégorie les plus proches de l'hyperplan, de part et d'autre de celui-ci, de telle sorte à ce que les marges se retrouvent chacune à la même distance de H :

$$\Delta = \min_{\boldsymbol{x}_i \in S} dist(\boldsymbol{x}_i, H) = \min_{\boldsymbol{x}_i \in S} \frac{|\boldsymbol{w}^T \boldsymbol{x}_i + b|}{\|\boldsymbol{w}\|}$$

Comme pour l'algorithme des K plus proches voisins, la mise en œuvre de cet algorithme se déroule en 3 étapes :

1) La création du modèle : *model = svm.LinearSVC(C)*

2) L'apprentissage du modèle : *model.fit(Xapp, Yapp)*

3) La prédiction : *Ypred = model.predict(Xtest)*

Le modèle dépend d'un hyperparamètre nommé C. Le réglage de sa valeur permet d'utiliser l'algorithme SVM selon ses 2 versions :

- L'algorithme SVM à marge dure, qui impose que tous les exemples soient bien classés. (C=∞).

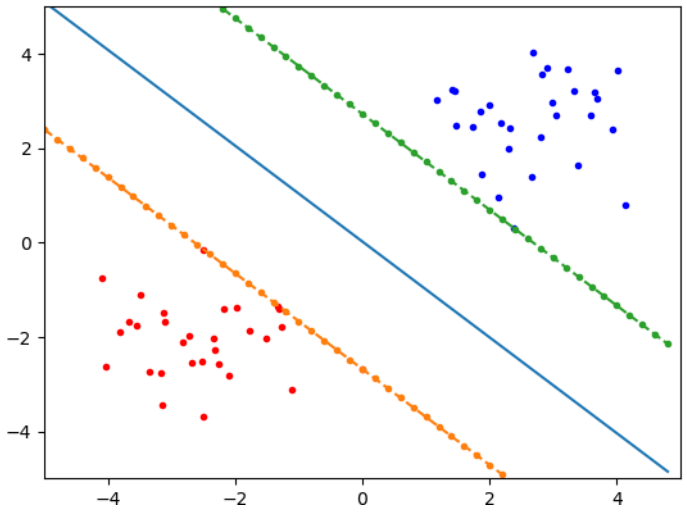
- L'algorithme SVM à marge douce, qui tolère des erreurs de classification pour rendre l'algorithme réalisable et optimal (C>0).

I. Test de l'algorithme dans le plan

Dans cette première partie, nous allons tester l'algorithme SVM dans un plan en 2D, où nous plaçons des points de couleur rouge et des points de couleur bleue.

L'algorithme doit ensuite définir l'hyperplan de séparation H et les marges correspondantes au jeu de données défini par l'utilisateur.

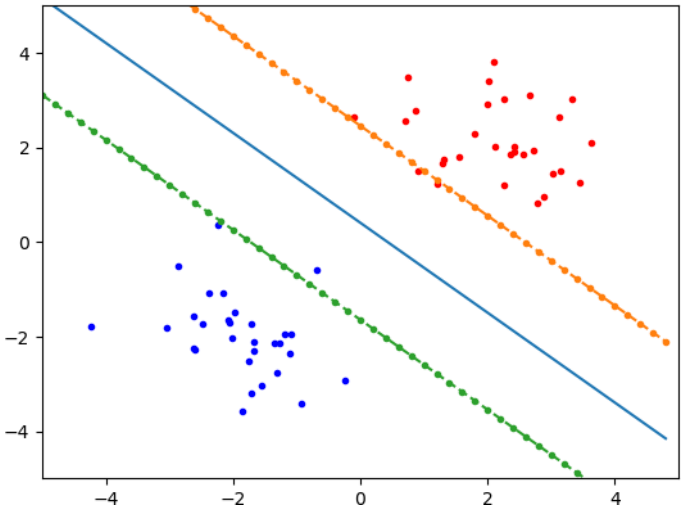
Dans un premier temps, nous allons tester l'algorithme à marge dure (C=∞), dans un cas linéairement séparable :



On observe donc la création de l'hyperplan H, et les 2 marges, à égale distance de part et d'autre de cet hyperplan. Ces marges correspondent aux points de chaque catégorie les plus proches de l'hyperplan, de telle sorte qu'aucun point ne se trouve à l'intérieur des marges.

Ces points xi se trouvant sur la marge sont appelés vecteurs supports. Le déplacement ou la suppression d'un point xi en dehors de cette marge n'a aucun impact sur la solution.

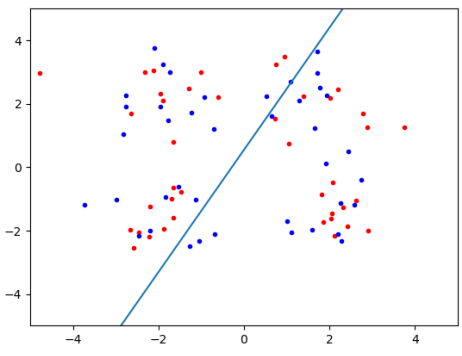
Le même exemple, en utilisant l'algorithme SVM à marge douce (C>0) nous donne :



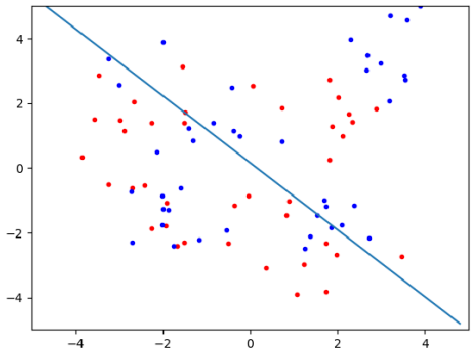
Où l'on observe ici une tolérance aux erreurs : certains points se retrouvent à l'intérieur des marges. En effet, l'algorithme à marge douce recherche le meilleur compromis entre la qualité d'erreur et la marge. Le règlement de ce compromis s'effectue par variation de l'hyperparamètre C.

Dans un second temps, nous allons nous intéresser à un cas où les données sont non linéairement séparables. Ici, seul l'algorithme à marge douce est utilisable, puisqu'il est impossible de trouver une solution où les points sont parfaitement séparés de part et d'autre de chaque marge, sans erreur de classification.

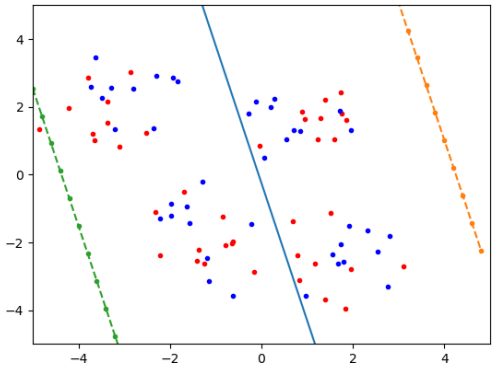
L'objectif est donc de trouver la valeur de C qui nous offre la solution la plus optimale :



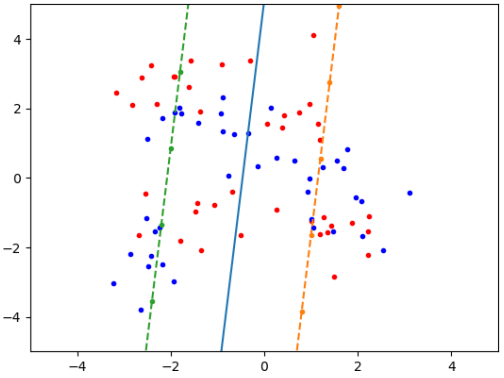
C=1



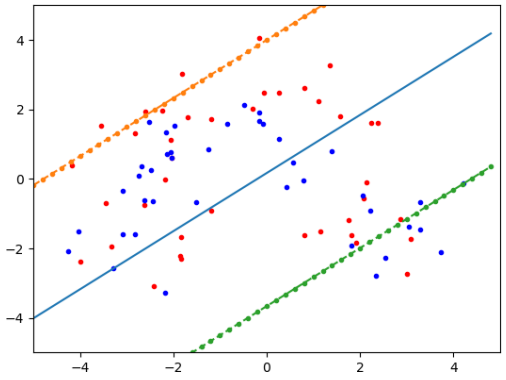
C=64



C=128



C=512



C=1024

Où l'on constate que la solution optimale semble être atteinte pour une valeur de C environ égale à 0. Pour des valeurs de C plus petites ou plus grandes, on peut observer des solutions moins efficaces, dues aux phénomènes de sous-apprentissage et sur-apprentissage.

Par la suite, on peut également tester l'algorithme dans un fonctionnement non linéaire. Pour cela, on fait appel à la méthode des noyaux. Ici, nous utilisons un noyau gaussien, de la forme :

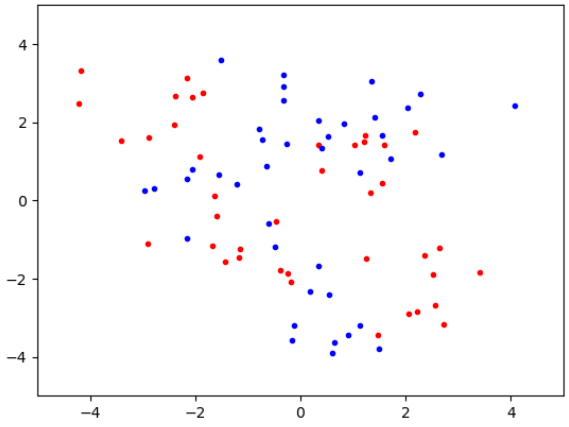
$$K(\boldsymbol{x}, \boldsymbol{x}') = \exp\left(\frac{-\|\boldsymbol{x} - \boldsymbol{x}'\|_2^2}{2\sigma^2}\right)$$

Concrètement, cela revient à changer la définition du modèle :

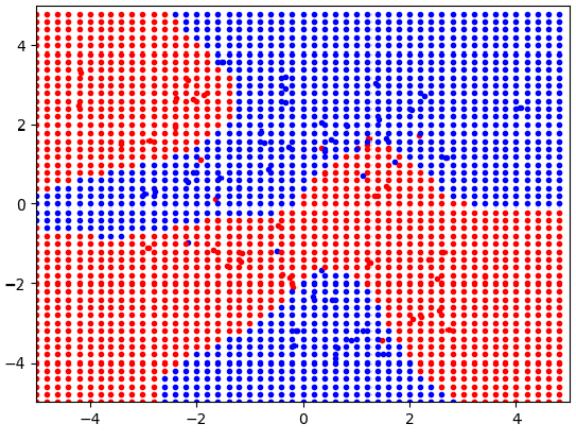
*modelNL = svm.SVC(C=C, kernel='rbf', gamma=1/(2*sigma**2))*

Le modèle non linéaire dépend donc de 2 hyperparamètres : C et σ (paramètre du noyau gaussien).

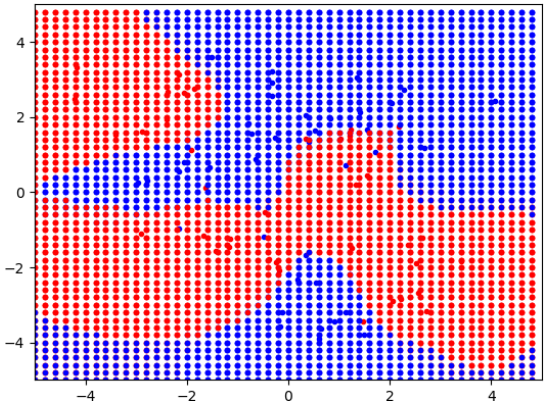
Pour mieux comprendre le fonctionnement de cet algorithme de classification non linéaire, revenons à l'exemple de la grille de points à colorier. En effet, la frontière de séparation des données n'est plus une droite, mais une courbe. Ainsi il est plus simple de visualiser la séparation à l'aide de la grille.



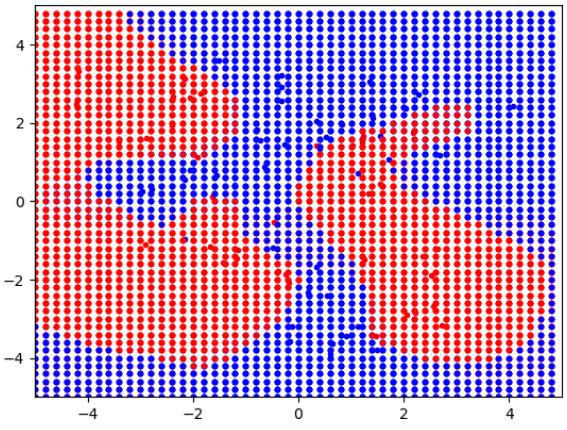
Jeu de données



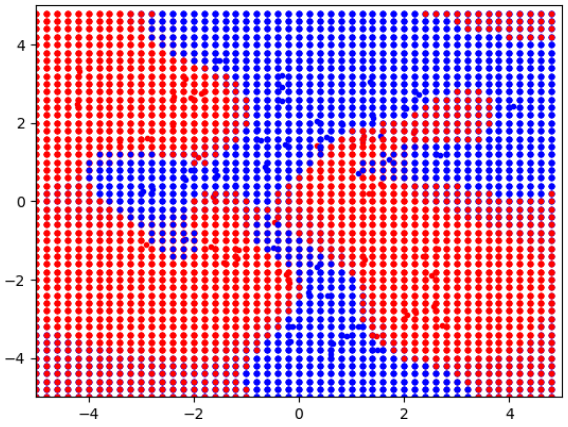
C=1 ; σ=1



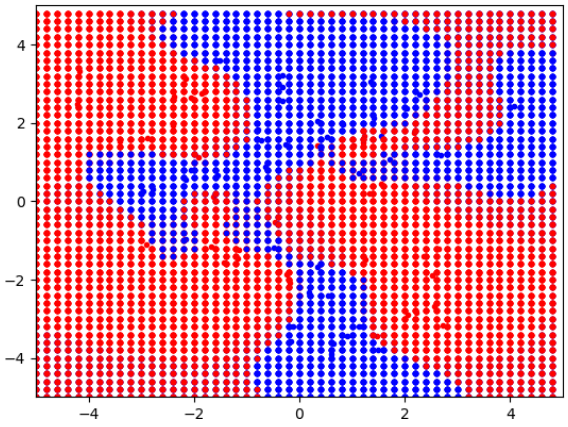
C=4 ; σ=1



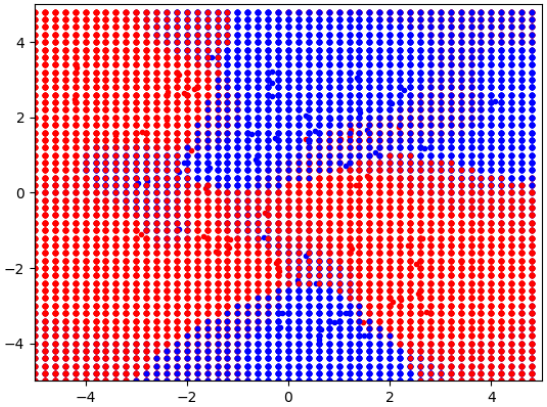
C=32 ; σ=1



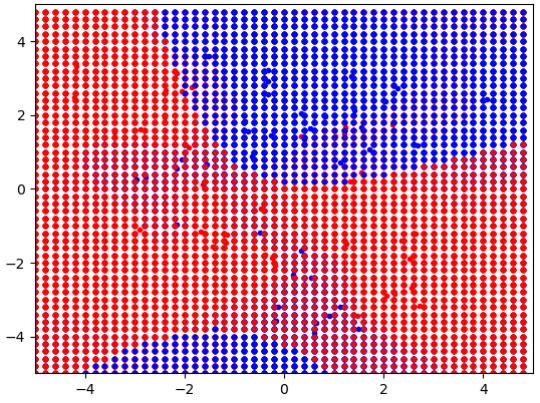
C=128 ; σ=1



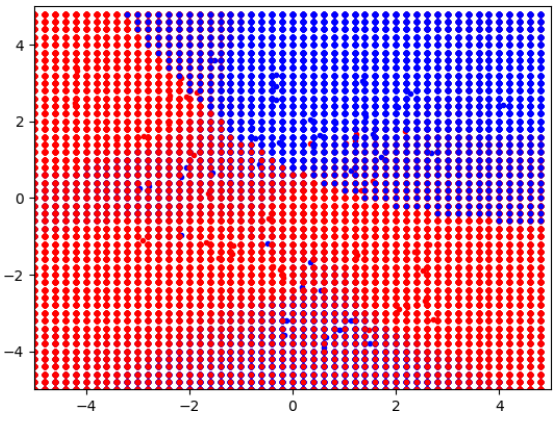
C=512 ; σ=1



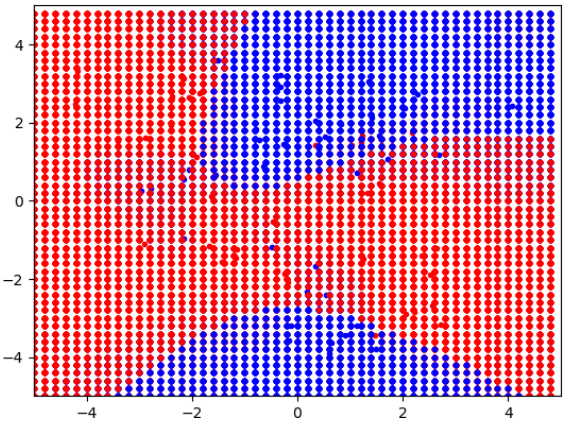
C=1 ; σ=2



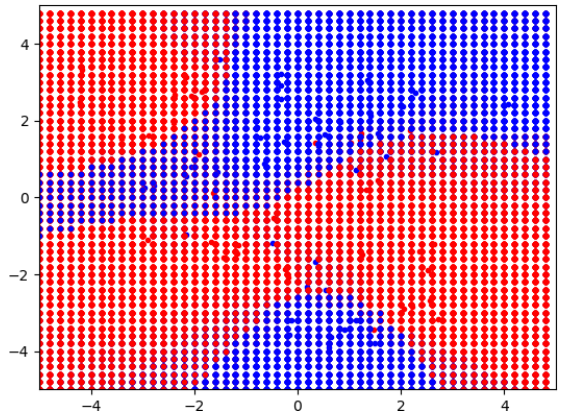
C=1 ; σ=4



C=1 ; σ=8



C=32 ; σ=4



C=256 ; σ=4

En observant les résultats ci-dessus, on peut alors en conclure que σ joue sur la complexité du classifieur : plus σ est grand et plus les zones sont larges, et plus le nombre de vecteurs supports est important.

II. Application à la classification de défauts de rails

1) Classifieurs binaires

Dans cette deuxième partie, nous allons nous intéresser à un problème multi-classes à partir de données de défauts de rails, en utilisant la méthode de décomposition *un contre tous*.

Les données utilisées comportent 140 exemples x, et 140 étiquettes y associées respectivement à chaque exemple x. Chaque étiquette y est un nombre entier entre 1 et 4, chacun de ces 4 numéros représentant un type de défaut particulier.

La première étape consiste, à partir de la matrice y, à implémenter et évaluer 4 classifieurs binaires, chacun d'eux étant spécialement dédié à la détection d'un seul défaut.

```
for k in range(0,4):
    Yk = 2*(y==k+1)-1
    model = svm.LinearSVC(C=1)
    model.fit(X, Yk)
    Ypred = model.predict(X)
    Error = np.mean(Ypred != Yk)
    print("Erreur modèle", k+1, " = ", Error)
    G[:,k] = model.decision_function(X)
```

Concrètement, pour chacun des 4 défauts, on commence par créer une matrice Yk composée de valeurs -1 ou +1 (+1 si le numéro de défaut de y correspond bien au numéro de défaut étudié pour ce classifieur, -1 s'il s'agit d'un des 3 autres défauts).

On crée ensuite un classifieur binaire SVM linéaire à partir de la matrice Yk, puis on réalise les prédictions des valeurs de la matrice Yk à l'aide de la base d'apprentissage.

Les résultats obtenus pour chaque classifieur lors de l'évaluation de l'erreur de reconstruction de la matrice Yk sont de l'ordre de :

- Classifieur 1 : 2.1 %
- Classifieur 2 : 0.7 %
- Classifieur 3 : 0.7 %
- Classifieur 4 : 0.0 %

Enfin, la dernière étape de la boucle consiste à calculer, pour chaque classifieur, le score *gj(x)* attribué à chaque valeur. Tous les résultats sont ensuite contenus dans une matrice G de dimensions 140 x 4.

2) Combinaison des classifieurs binaires

A présent nous voulons construire le classifieur multi-classes généralisé, à partir des 4 classifieurs binaires élaborés à l'étape précédente, pour ensuite évaluer ce classifieur sur la reconstruction des valeurs de *y*.

Pour cela, nous calculons nos prédictions de la façon suivante :

$$f(x) = argmax\ g_k(x)$$

```
Ypredict = np.argmax(G,axis=1)+1
ErrApp = np.mean(Ypredict != y)
print("Erreur d'apprentissage : ", ErrApp)
```

Concrètement, pour chacune des 140 valeurs de X, on choisit le classifieur qui donne le meilleur score, et on utilise ce classifieur pour réaliser la prédiction de son étiquette *Ypredict*.

On compare ensuite les prédictions avec les données réelles de y, ce qui nous donne une erreur d'apprentissage du classifieur multi-classes global d'environ 0%.

3) Validation croisée

Cependant, nous avions vu dans le TP n°1 que la procédure générale pour évaluer un classifieur consiste à effectuer son apprentissage sur des données d'apprentissage, puis effectuer son évaluation sur des données de test, séparées des données d'apprentissage.

Or, ici, la base d'apprentissage est déjà suffisamment limitée en termes de quantité d'exemples, il est donc difficile de couper ce jeu de données en 2 pour en extraire des données de test.

Dans ce cas, une technique efficace permettant d'évaluer proprement notre classifieur consiste à effectuer une validation croisée, ici en l'occurrence la méthode *Leave-One-Out (LOO).*

Concrètement, cette technique consiste à répéter l'apprentissage des 4 classifieurs autant de fois que la matrice X possède d'exemples (140), en excluant à chaque fois l'exemple *xi* et son étiquette *yi* des matrices X et Y.

Le test se fait ensuite à chaque itération sur la valeur *xi* exclue. Ainsi le test s'effectue sur des données non prises en compte pour l'apprentissage.

Toutes les erreurs résultantes de chaque itération sont ensuite ajoutées, puis l'erreur de validation croisée LOO correspond ensuite à la moyenne de ces erreurs.

```
# Validation croisée

erreurs=0
for i in range(0,140):
    x_i = np.delete(X, i, axis=0)
    y_i = np.delete(y, i)
    G = np.zeros(4)

    for k in range(0,4):
        Yk = 2*(y_i==k+1)-1
        model = svm.LinearSVC(C=1)
        model.fit(x_i, Yk)
        G[k] = model.decision_function([X[i,:]])

    Ypred = np.argmax(G)+1
    erreurs += (Ypred != y[i])

ErreurLOO = erreurs/len(y)
print("Nombre erreurs = ", erreurs)
print("Erreur LOO = ", ErreurLOO)
```

Ce qui nous donne finalement une erreur d'apprentissage par validation croisée d'environ 7%.

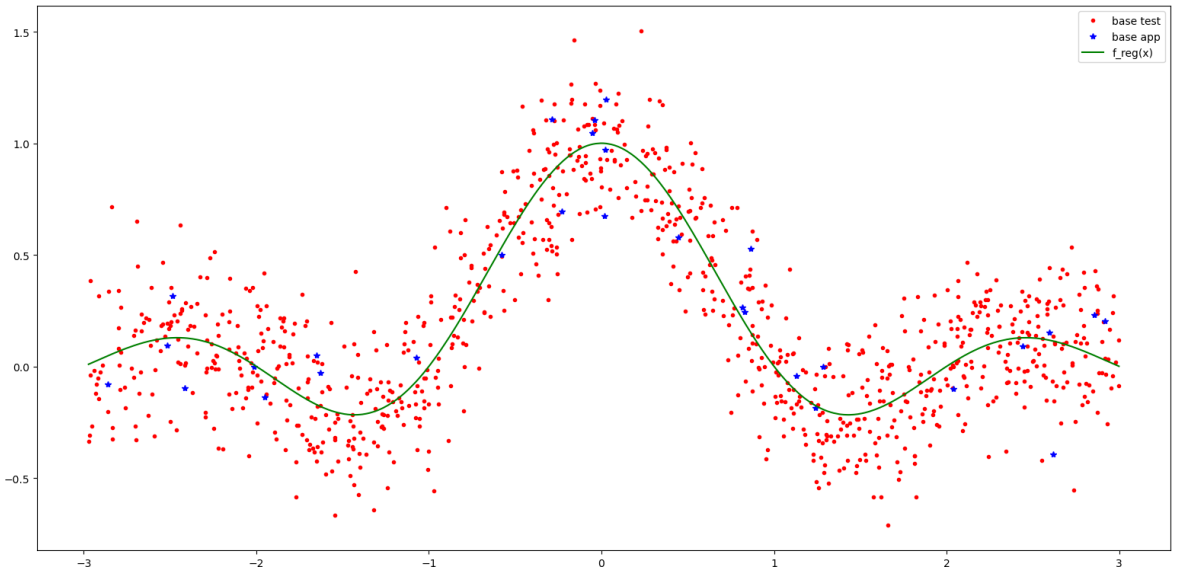
Dmytro SAVCHUK

TP n°3 :

Régression non linéaire

La régression non linéaire consiste à déterminer la fonction non linéaire qui permet d'approcher le plus efficacement possible un ensemble de données.

L'exemple que nous allons utiliser pour ce TP est le suivant :



Avec donc :

- En bleu, les données d'apprentissage (30 exemples).
- En rouge, les données de test (970 exemples).

(Ces données sont réparties dans le plan à l'aide de la fonction sinus cardinal)

- En vert, la fonction sinus cardinal qui représente la droite optimale qui approche ces données.

Kernel Ridge Regression

Nous allons ensuite appliquer une régression non linéaire par la méthode *Kernel Ridge Regression* (KRR).

Cette méthode doit permettre d'apprendre la fonction non linéaire sinus cardinal recherchée, en calculant :

$$f = \sum_{i=1}^{m} \beta_i K(x_i, \cdot)$$

Pour cela, la première étape consiste à calculer la matrice de noyau K avec un noyau gaussien :

$$K(\boldsymbol{x}, \boldsymbol{x}') = \exp\left(\frac{-\|\boldsymbol{x} - \boldsymbol{x}'\|_2^2}{2\sigma^2}\right)$$

```
def kernel(X1,X2,sigma):
    m1 = X1.shape[0]
    m2 = X2.shape[0]
    K = np.zeros((m1,m2))
    for i in range(m1):
        for j in range(m2):
            K[i,j] = math.exp(- np.linalg.norm(X1[i] - X2[j])**2 / (2*sigma**2))
    return K
```

Puis la deuxième étape consiste à calculer le vecteur β qui réalise l'apprentissage du modèle KRR à noyau gaussien à partir de la base d'apprentissage Xapp & Yapp, selon 2 hyperparamètres $\lambda \; et \; \sigma$:

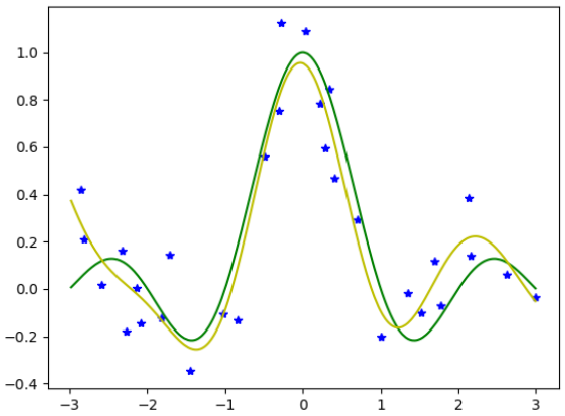
```
def krrapp(X,Y,Lambda,sigma):
    K = kernel(Xapp, Xapp, sigma)
    A = K+(np.eye(len(K))*Lambda)
    beta = np.linalg.solve(A,Y)
    return beta
```

Puis la troisième étape consiste à effectuer les prédictions de la fonction de régression non linéaire :

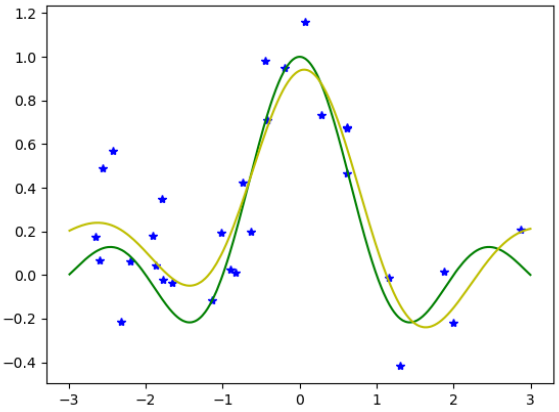
```
def krrpred(Xtest,Xapp,beta,sigma):
    Ktest = kernel(Xtest, Xapp, sigma)
    ypred = Ktest.dot(beta)
    return ypred
```

On peut alors faire varier les hyperparamètres λ *et* σ pour tenter d'avoir la meilleure régression possible :

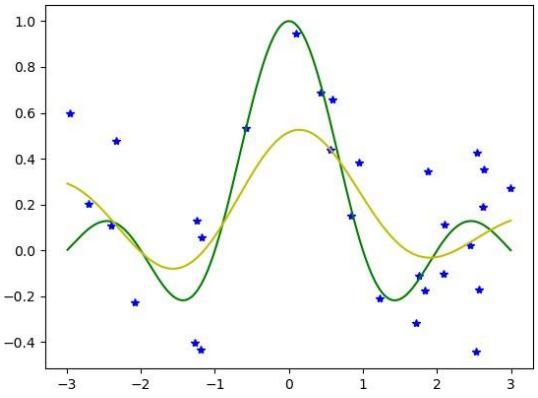
(avec en vert la fonction sinus cardinal, et en jaune la fonction de régression)



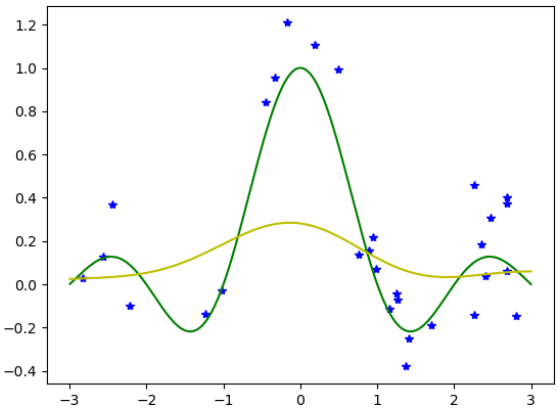
$\lambda = 0.01$; $\sigma = 1$ (Erreur = 5.1%)



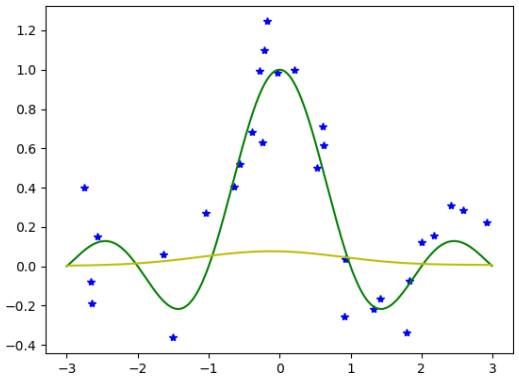
$\lambda = 0.1$; $\sigma = 1$ (Erreur = 5.1%)



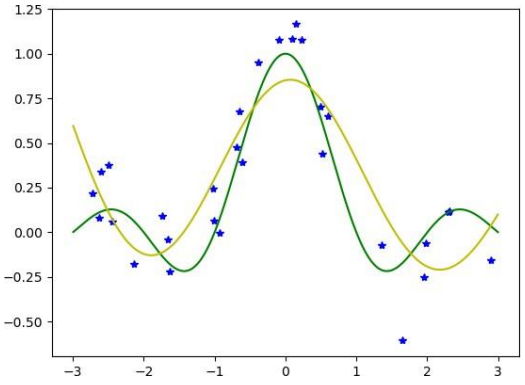
$\lambda = 1$; $\sigma = 1$ (Erreur = 8.9%)



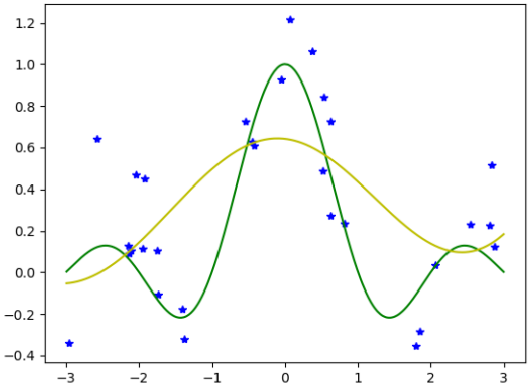
$\lambda = 10$; $\sigma = 1$ (Erreur = 13.6%)



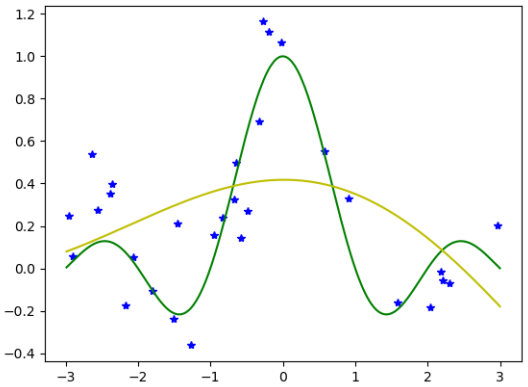
$\lambda = 100$; $\sigma = 1$ (Erreur = 18.7%)



$\lambda = 0.01$; $\sigma = 2$ (Erreur = 9.0%)



$\lambda = 0.01$; $\sigma = 3$ (Erreur = 13.3%)



$\lambda = 0.01$; $\sigma = 4$ (Erreur = 15.0%)

On peut alors remarquer l'influence de chacun des 2 hyperparamètres sur la fonction de régression :

- Plus λ est faible et plus les variations d'amplitude sont importantes. A l'inverse, plus λ est élevé et plus la fonction s'apparente à une droite à l'ordonnée 0.
- Plus σ est élevé et plus la forme sinusoïdale de la fonction se transforme peu à peu en une parabole qui représente la tendance globale des données.

Ainsi, les meilleurs résultats semblent être obtenus avec $\lambda = 0.01$ et $\sigma = 1$.

Dmytro SAVCHUK

TP n°4 :

K-means et spectral clustering

L'objectif de ce TP est de mettre en application 2 algorithmes de *clustering* :

- K-means
- Spectral clustering

L'opération de *clustering* consiste à créer des groupes pour y classer des données en fonction de critères spécifiés. Contrairement aux algorithmes de classification étudiés dans les premiers TP, nous ne cherchons pas à classer les données dans des groupes définis, mais bien à créer les groupes permettant de classer les données de la manière la plus optimale possible.

Nous allons tester les 2 algorithmes nommés ci-dessus, dans un plan en 2D dans lequel serons placés des points d'une seule et même catégorie (couleur).

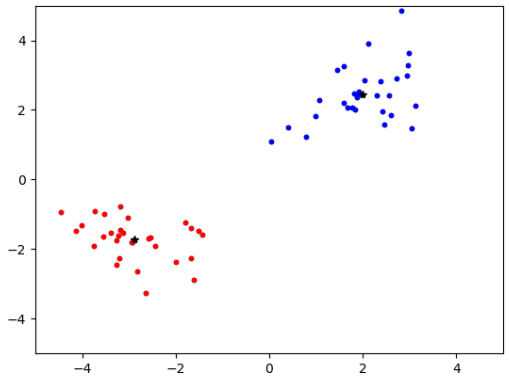
I. K-means

Le principe de cet algorithme est le suivant : l'algorithme sépare les données en K groupes, où chacun de ces groupes est caractérisé par son centre.

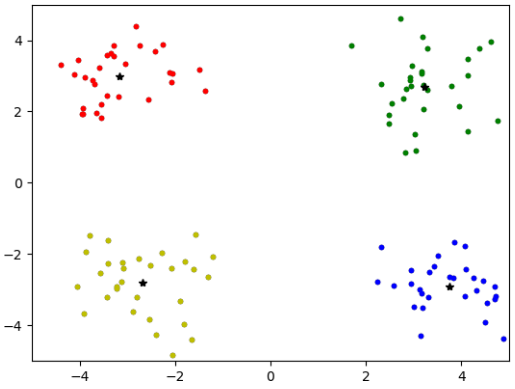
L'algorithme exécute ensuite plusieurs itérations d'une boucle, où pour chaque point l'on redétermine son groupe selon le centre le plus proche de ce point, jusqu'à ce que l'algorithme converge vers une solution, où la position des centres des K groupes est stabilisée.

```
clustering = KMeans(n_clusters=K, init='random', n_init=1)
clustering.fit(X)

y = clustering.labels_

couleurs = ['b','r','g','y','m','c']

for k in range(K):
    Xk = X[y==k,:]
    plt.plot(Xk[:,0], Xk[:,1], '.'+couleurs[k])
    # Calcul des centres
    centers = clustering.cluster_centers_[k,:]
    plt.plot(centers[0], centers[1], '*k')
```

Pour tester cet algorithme, on peut commencer par le tester dans un cas où les données sont déjà séparées en groupes distincts, et constater que l'algorithme effectue bien la création des groupes selon cette logique :



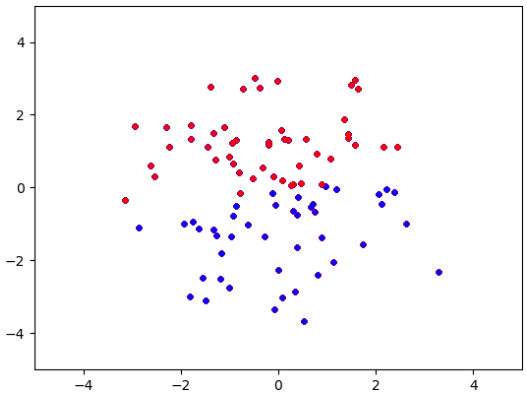
K=2

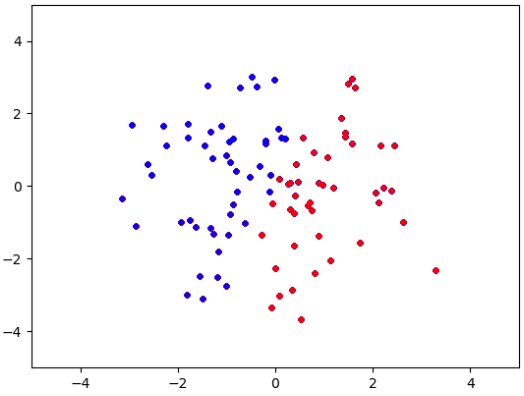


K=4

Le centre de chaque groupe est alors représenté par une étoile *****.

A présent, si l'on teste l'algorithme sur des données non distinctement séparées, on constate un aspect aléatoire du résultat, lorsqu'on exécute plusieurs fois l'algorithme sur un même jeu de données. Par exemple :





Où l'on constate bien ici, sur un exemple simple avec K=2 où il n'y a pas de solution unique évidente, que le résultat donné par l'algorithme varie, avec une séparation entre les 2 groupes qui peut être soit verticale, soit horizontale.

Pour régler ce phénomène aléatoire, il peut être utile de lancer l'algorithme avec un nombre d'initialisations élevées (par exemple 100), ainsi l'algorithme testera plusieurs solutions et ne gardera que la meilleure de celles-ci.

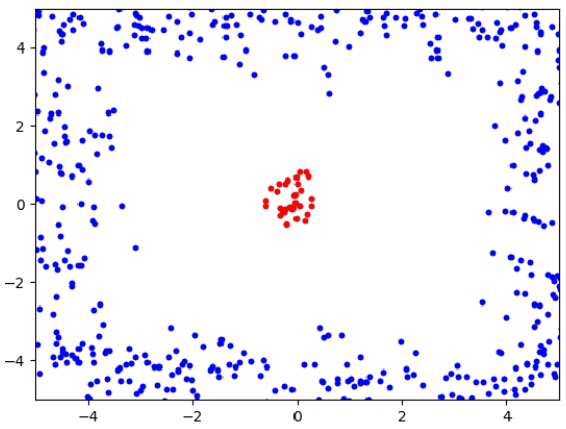
Cependant, la limite de cet algorithme est que la classification des données se fait en fonction du centre de groupe le plus proche, ainsi les groupes seront toujours répartis en zones, en fonction des distances les séparant des centres.

II. Spectral clustering

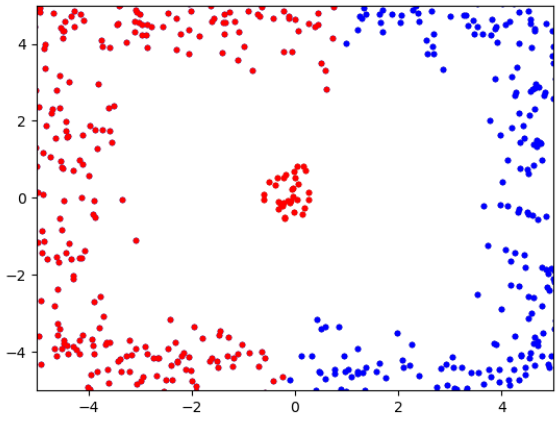
L'algorithme *Spectral Clustering* permet quant à lui d'aller un peu plus loin que l'algorithme *K-means*, puisque la classification des points s'effectue cette fois en évaluant la similarité avec les autres points, et non plus en cherchant la plus petite distance avec un centre de groupe.

```
clustering = SpectralClustering(n_clusters=K, gamma=1/(2*sigma**2))
clustering.fit(X)

y = clustering.labels_

plt.figure(1)
couleurs = ['b','r','g','y','m','c','k']

for k in range(K):
    Xk = X[y==k,:]
    plt.plot(Xk[:,0], Xk[:,1], '.'+couleurs[k])
```

Ainsi, cela nous donne, par exemple :



Spectral clustering



K-means

Dans cet exemple, on distingue bien 2 groupes distincts, mais avec l'un d'eux inclus dans l'autre. L'algorithme *Spectral clustering* parvient à distinguer ces 2 groupes en analysant la similarité entre les points, ce que n'est pas capable de faire l'algorithme K-means, qui persiste à placer chaque groupe d'un côté du plan.