# Intelligence artificielle et Apprentissage
# TP : classification SVM, méthodes de décomposition et classification de défauts de rails
# Version Python

F. Lauer

## 1 Classification SVM

Il existe beaucoup de logiciels implémentant les SVM. Nous utiliserons ici les fonctions de la bibliothèque python scikit-learn (basée sur le logiciel LibSVM) :

```
from sklearn import svm
```

L'interface est similaire à celle utilisée pour l'algorithme Kppv : apprentissage du modèle avec `model.fit(X, Y)` et prédictions avec `model.predict(Xtest)`. Les hyperparamètres sont donnés en amont lors de la création du modèle avec une syntaxe différente selon le noyau choisi :

```
model = svm.LinearSVC(C=C)         # noyau linéaire
model = svm.SVC(C=C, kernel='rbf', gamma=1/(2*sigma**2) )  # noyau gaussien
model = svm.SVC(C=C, kernel='poly', degree=D)   # noyau polynomial de degré D
```

où `SVC` signifie "support vector classifier" et `sigma` représente le paramètre $\sigma$ du noyau gaussien $K(x, x') = \exp(-\|x - x'\|^2/2\sigma^2) = \exp(-\gamma\|x - x'\|^2)$.

### 1.1 Cas séparable

Dans `svm2D.py` :

1. Créez un jeu de données linéairement séparable avec la souris.
2. Complétez la fonction `monprogramme` pour apprendre l'hyperplan de séparation optimal sur les données et visualiser la classification du plan obtenue au travers d'une grille de points de test.
   *On obtient ce classifieur en utilisant un noyau linéaire et $C = \infty$ (=`math.inf` en python).*
3. Tracez la droite de séparation en récupérant le vecteur normal $\boldsymbol{w}$ et le biais $b$ avec :

   ```
   w = model.coef_[0]
   b = model.intercept_
   ```

4. Calculez la marge $\Delta$ et tracez en tirets les frontières de la marge.
   *Indice : les points sur le bord de la marge satisfont $|\boldsymbol{w}^T\boldsymbol{x} + b| = 1$.*

### 1.2 Cas non séparable

1. Créez maintenant un jeu de données non linéairement séparable. Sur ces données, l'hyperplan de séparation optimal n'est plus défini.
2. Utilisez maintenant une SVM "à marge souple", avec une valeur de $C$ finie et tracez les mêmes droites que précédemment.
3. Calculez également la marge et observez son évolution lorsque vous changez la valeur de $C$.

### 1.3 Classification non linéaire

Ces étapes sont à réaliser dans la fonction `monprogrammeNL` qui reprend le même fonctionnement mais avec un paramètre `sigma` supplémentaire.

1. Utilisez maintenant un noyau gaussien. Dans ce cas, il n'est plus possible d'afficher aussi facilement la frontière entre les catégories qui n'est plus une droite. Pour cela il faudrait tracer la courbe de niveau pour $g(x) = 0$. Nous nous contenterons donc de visualiser la frontière grâce à la grille de points de test.
2. Faites varier $C$ et $\sigma$ au clavier et observez leur effet sur la surface de séparation. Affichez également le nombre de vecteurs support (`model.n_support_`) et observez son évolution en fonction de $C$ et $\sigma$.

## 2 Problème multi-classe et techniques de validation croisée : application à la classification de défauts de rails

Dans cette partie du TP, nous allons traiter un problème multi-classe à partir de données réelles provenant de défauts de rails. Pour cela, nous commencerons par créer des classifieurs bi-classes permettant de distinguer un défaut parmi les autres dans le but d'appliquer la méthode de décomposition *un-contre-tous*.

Les données sont disponibles dans le fichier `defautsrails.dat` qui peut être chargé avec :

```
data = np.loadtxt("defautsrails.dat")
X = data[:,:-1]  # tout sauf la dernière colonne
y = data[:,-1]   # uniquement la dernière colonne
```

Cela permet d'obtenir :

— une matrice de 140 exemples `X` : chaque ligne correspond à un exemple et représente avec 96 descripteurs une fenêtre de mesure des signaux des capteurs magnétiques ;

— un vecteur `y` de 140 étiquettes associées aux exemples de `X` : chaque étiquette est un entier entre 1 et 4 représentant un type de défaut de rail différent (joint avant aiguillage, joint éclissé, joint soudé, écaille).

### 2.1 Classifieurs binaires

1. Dans un nouveau programme `defautsrails.py`, écrivez une boucle pour créez les 4 classifieurs binaires SVM linéaires nécessaires à la méthode de décomposition *un-contre-tous* à partir de toutes les données.
2. Complétez la boucle pour calculer, pour chaque classifieur binaire séparément, son taux de reconnaissance sur la base d'apprentissage.

### 2.2 Combinaison des classifieurs binaires

Nous allons maintenant créer le classifieur multi-classe général basé sur la méthode un-contre-tous. Pour éviter les ambiguïtés, la classe prédite par ce classifieur correspond au classifieur binaire conduisant à la valeur réelle maximale en sortie. Autrement dit :

$$f(x) = \arg \max_{j=1,\dots,4} g_j(x)$$

où $g_j(x)$ est la sortie réelle du classifieur chargé de distinguer la classe $j$ :

$$f_j(x) = \text{signe}(g_j(x))$$

Pour récupérez les valeurs de $g_j(x)$, que l'on peut voir comme des scores pour chaque catégorie, il faut appeler `model.decision_function` au lieu de `model.predict` :

```
score = model.decision_function(X)
```

Note : les fonctions `predict` et `decision_function` attendent une matrice de points de test ; pour faire une prédiction pour un seul point `x`, il faut donc passer par exemple par `score = model.decision_function([x])`.

1. Complétez `defaultsrails.py` pour calculer la prédiction multi-classe à partir des classifieurs binaires et calculez l'erreur d'apprentissage du classifieur multi-classe global. *Vous pourrez utiliser*

```
indiceDuMax = np.argmax( u ) # pour un vecteur u
indicesColonneMax = np.argmax( U ) # pour une matrice U
```

## 2.3 Estimation de l'erreur de généralisation par validation croisée

Le faible nombre d'exemples disponibles ne permet pas d'en retenir pour créer une base de test indépendante de la base d'apprentissage. Pour évaluer les performances en généralisation du classifieur, nous allons donc utiliser la technique de validation croisée *Leave-One-Out* (LOO).

Complétez le programme pour réaliser les tâches suivantes.

1. L'apprentissage du classifieur multi-classe (et donc des 4 classifieurs binaires) de l'exercice précédent doit être répété pour chaque $i$ de 0 à 139 avec l'exemple d'indice $i$ exclu de la base d'apprentissage. Pour visualiser l'évolution de la procédure (qui peut être un peu longue), pensez à afficher $i$ à chaque itération. Il n'est pas nécessaire de mémoriser tous les classifieurs obtenus (voir question suivante).
   *On peut utiliser* `X_i = np.delete(X, i, axis=0)` *pour récupérer une copie de* `X` *sans la* $i$*ème ligne et* `y_i = np.delete(y, i)` *pour un vecteur.*
2. Après chaque apprentissage, le classifieur obtenu est testé sur l'exemple d'indice $i$ (non utilisé pour l'apprentissage) et le nombre d'erreurs multi-classes ainsi faites est enregistré (par contre il n'est pas nécessaire de mémoriser tous les classifieurs). L'erreur de validation croisée sera le nombre d'erreurs à la fin de la procédure divisé par le nombre d'exemples dans la base complète.
3. A combien est estimée l'erreur de généralisation de votre classifieur multi-classe ?
4. Déterminez également les erreurs de validation croisée pour chacun des classifieurs binaires. Quelles informations apportent ces erreurs par rapport à l'erreur du classifieur global ?
5. Quel est le temps de calcul de la procédure complète ?