Apprentissage statistique et sélection de modèle
TP noté : classification SVM et méthodes de décomposition
Version Python

F. Lauer

Pré-requis

Pour réaliser des calculs scientifiques (par exemples sur des matrices) en Python, nous utilisons la bibliothèque `numpy`, ainsi qu'une autre qui permet de tracer des figures dans un environnement similaire à Matlab :

```
import numpy as np
import matplotlib.pyplot as plt
```

Les algorithmes de machine learning sont implémentés en python dans la bibliothèque scikit-learn (`sklearn`). En règle générale, seule la partie utilisée de la bibliothèque est importée. Par exemple

```
from sklearn import neighbors
```

importe les outils liés aux plus proches voisins. Pour les SVM, nous utiliserons donc :

```
from sklearn import svm
```

De manière générale (pour les SVM ou les autres algorithmes), il y a ensuire 3 étapes majeures : la création du modèle pendant laquelle les hyperparamètres sont fixés, l'apprentissage à partir des données, et la prédiction sur d'autres données. Pour les SVM, la création du modèle est obtenue par (une des trois lignes) :

```
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 σ du noyau gaussien $K(x,x') = \exp(-\|x-x'\|^2/2\sigma^2) = \exp(-\gamma\|x-x'\|^2)$.

L'apprentissage du modèle est réalisé avec :

```
model.fit(X, Y)
```

où

$$\texttt{X} = \begin{bmatrix} \boldsymbol{x}_1^T \\ \boldsymbol{x}_2^T \\ \vdots \\ \boldsymbol{x}_m^T \end{bmatrix}, \quad \texttt{Y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix}$$

contiennent la base d'apprentissage.

Pour calculer un vecteur de prédictions `Ypred` sur les points stockés en lignes dans la matrice `Xtest`, on utilise ensuite :

```
Ypred = model.predict(Xtest)
```

Les fonctions `.fit()` et `.predict()` sont les méthodes standards pour l'apprentissage et la prédiction pour tous les modèles de classifieur ou de régression dans scikit-learn.

Installation. Pour installer ces bibliothèques sous Ubuntu :

```
sudo apt install python3-numpy python3-scipy python3-matplotlib python3-sklearn
```

Sous Windows, la gestion des paquets et dépendances est plus complexe, et il est habituel d'installer une distribution telle qu'Anaconda contenant l'ensemble des paquets utiles.

1 Classification SVM

Remarque : il est possible que lors de l'utilisation de `svm.LinearSVC` pour la classification linéaire, un message d'avertissement soit affiché relatif à la convergence. Cela signifie que l'algorithme d'optimisation utilisé par le logiciel n'a pas totalement convergé vers la solution optimale et que le modèle ne correspond donc pas exactement à sa définition mathématique. Pour éviter cela, vous pouvez augmenter le nombre d'itérations avec par exemple `svm.LinearSVC(C=C, max_iter = 10000)`.

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 Δ 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 σ 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 σ.

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 ?

2.4 Sélection de modèle

Bien que nous considérions des SVM linéaires, il reste néanmoins un hyperparamètre à régler : C. L'approche standard consisterait soit à couper la base en deux pour créer une base de validation (mais le nombre de données est ici insuffisant), soit à utiliser une validation croisée pour estimer la qualité de chaque valeur de C testée (mais la validation croisée est déjà utilisée ici pour estimer les performances en généralisation du modèle final).

1. Proposez une méthode permettant d'effectuer la sélection de modèle (le réglage de C) dans ce cas précis sans remettre en cause l'estimation de la qualité du modèle final. Appliquez cette méthode et donnez une estimation du risque obtenu.
2. En réalité, nous avons 4 hyperparamètres C_k, $k = 1, \ldots, 4$, un pour chaque classifieur binaire associé à une catégorie k. Modifiez votre programme pour effectuer la sélection de modèle indépendamment pour chaque catégorie.
3. Plutôt que d'optimiser les hyperparamètres de chaque classifieur binaire indépendamment, une autre approche serait de les considérer simultanément pour optimiser les performances du classifieur multi-classe global. Comparez cette approche avec la précédente, dans un premier temps sans l'implémenter et en discutant de son bénéfice potentiel et de ses difficultés de mise en œuvre. Celle-ci est-elle applicable à n'importe quel problème ?
4. Implémentez cette dernière approche d'optimisation simultanée des hyperparamètres et comparez les temps de calculs et résultats obtenus avec celle consistant à régler les C_k de manière indépendamment les uns des autres.

3 Programmation quadratique

Nous allons ici essayer d'implémenter l'apprentissage SVM nous-mêmes.

1. Créez une fonction du type `(w,b) = apprendSVM(X,Y,C)` qui réalise l'apprentissage SVM en utilisant la bibliothèqe `quadprog` pour résoudre le problème de programmation quadratique

dans le primal déterminant directement $\boldsymbol{w}$ et b :

$$\begin{aligned}\min_{w\in\mathbb{R}^d,\ b\in\mathbb{R},\xi_i\in\mathbb{R}} \ & \frac{1}{2}\|\boldsymbol{w}\|^2 + C\sum_{i=1}^{m}\xi_i \\ s.c. \quad & y_i(\langle \boldsymbol{w}, \boldsymbol{x}_i\rangle + b) \geq 1 - \xi_i \\ & \xi_i \geq 0\end{aligned}$$

Vous pouvez installer `quadprog` *avec* `pip3 install quadprog -user`*, puis regarder un exemple d'utilisation dans* `exempleQuadprog.py` *sur ARCHE.*

2. Vérifiez le fonctionnement sur les données générées à la souris avec `svm2D.py` après avoir également créé la fonction
`y = predictionSVM(X, w, b)` qui calcule la classification des points stockés dans la matrice `X` avec une SVM de paramètres `w` et `b`.
3. Quelles sont les limites de cette approche ?
4. Pour palier à ces limites, implémentez maintenant, toujours en utilisant `quadprog`, la forme duale de l'apprentissage SVM dans une fonction `alpha = apprendSVMdual(X,Y, C, noyau, sigma)` :

$$\begin{aligned}\max_{\boldsymbol{\alpha}\in\mathbb{R}^m} \ & -\frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m}\alpha_i\alpha_j y_i y_j K(\boldsymbol{x}_i,\boldsymbol{x}_j) + \sum_{i=1}^{m}\alpha_i \\ s.c. \ & \sum_{i=1}^{m}\alpha_i y_i = 0 \\ & 0 \leq \alpha_i \leq C\end{aligned}$$

Vous pourrez commencer par écrire une fonction pour calculer la matrice $\boldsymbol{K}$ des $K(\boldsymbol{x}_i, \boldsymbol{x}_j)$ correspondant par exemple au noyau linéaire si `noyau == "lineaire"` et au noyau gaussien sinon.

5. Quelles sont les limites de cette approche basée sur la forme duale ?
6. Complétez la fonction pour calculer également la valeur de b : `(alpha, b) = apprendSVMdual(X,Y, C, noyau, sigma)`. Pour cela, il faut faire appel aux *conditions de relâchement supplémentaires de Karush-Kuhn-Tucker (KKT)* qui stipulent qu'à l'optimum le produit des variables duales et des contraintes associées est nul. Ainsi,

$$\alpha_i \neq 0 \quad \Rightarrow \quad y_i(\langle \boldsymbol{w}, \boldsymbol{x}_i\rangle + b) - 1 + \xi_i = 0$$

et $\xi_i \neq 0 \Rightarrow \mu_i = 0$ et donc $\alpha_i = C - \mu_i = C$. Donc, si $\alpha_i \neq C$, on ne peut avoir $\xi_i = 0$ et

$$\alpha_i < C \quad \Rightarrow \quad \xi_i \neq 0$$

En combinant ces deux résultats, il est possible de calculer b en choisissant l'indice i d'un point tel que $0 < \alpha_i < C$.

7. Implémentez aussi la fonction `y = predictionSVMdual(X, alpha, b, noyau, sigma)` et testez avec `svm2D.py` sur des données non linéairement séparables.