Intelligence artificielle & Machine learning
TP : classification SVM, méthodes de décomposition et classification de défauts de rails

Juliette Bluem - 5A

11 novembre 2022

Table des matières

Partie I : Classification SVM

Un classifieur SVM est un séparateurs à vaste marge (en anglais support-vector machine).
Les SVM sont des classificateurs qui reposent sur deux principes majeurs : l'idée de marge maximale et la notion de transformation de l'espace par une fonction noyau.
En effet, la marge est la distance entre la frontière et les point les plus proches. Ces points sont les vecteurs supports.
La difficulté dans ce classificateur est de trouver cette frontière à partir d'une base d'apprentissage.
La deuxième idée clé des SVM est de transformer l'espace de représentation des données d'entrées en un espace de plus grande dimension (possiblement de dimension infinie), dans lequel il est probable qu'il existe une séparation linéaire. Ceci est réalisé grâce à une fonction noyau.

Dans la première partie de ce TP, nous créons nous même notre jeu de données. Sur une grille, nous plaçons des points. Ces points ont des coordonnées (ordonnée et abscisse) et une couleur. Nous formons ainsi une base d'apprentissage.
Nous avons ensuite une fonction qui prédit la couleur de tous les autres points de la grille. Cet ensemble de points devient notre base de test. Il va de soit que c'est cette fonction qui fait réellement l'objet du TP. Nous la modifierons régulièrement afin de comprendre les différents paramètres de classification SVM ainsi que ses réactions aux différentes bases d'apprentissage.

Pour réaliser cela, nous utiliserons les fonctions de la bibliothèque python scikit-learn.

1 Cas séparable

Nous avons commencé par créer un jeu de données linéairement séparable. C'est à dire que notre ensemble de points appartenant à deux catégories (bleu ou rouge) pourra être séparé en deux par une droite.

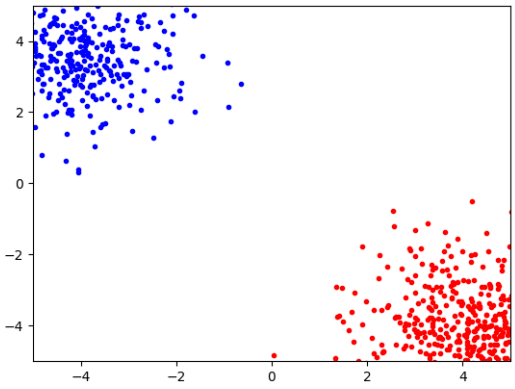

FIGURE 1 – Exemple grille pour base apprentissage cas linéairement séparables

Nous avons ensuite complété notre fonction afin, dans un premier temps, d'apprendre les processus de catégorisation. Pour cela, nous utilisons un noyau linéaire ($C = \infty$)

```
1     # Aprentissage du modele SVM
2         svm2d = svm.LinearSVC(C=math.inf)
3         svm2d.fit(Xapp, Yapp)
```

Dans un second temps la fonction prédira la couleur des points de la base de test.

```
    # Prediction de la categorie pour tous les points de test
        Ypred = svm2d.predict(Xtest)
```

Nous voulons visualiser ces prédictions.

```
    # Tracage du resultat
        XtestAbRouge = Xtest[Ypred == 2, 0]
        XtestOrdRouge = Xtest[Ypred == 2, 1]
        plt.plot(XtestAbRouge, XtestOrdRouge, 'or', alpha = 0.2)

        XtestAbBleu = Xtest[Ypred == 1, 0]
        XtestOrdBleu = Xtest[Ypred == 1, 1]
        plt.plot(XtestAbBleu, XtestOrdBleu, 'ob', alpha = 0.2)
```

En d'autres termes, nous mettons en rouge les points de la base de test appartenants à la catégorie 2 et en bleu ceux de la catégorie 1.

Pour tracer la droite frontière de ces deux catégories, nous devons exprimer x_2 en fonction de x_1. Or nous ne disposons que de w et b.
w est le vecteur normal et b le biais.

$$w.x + b = 0$$
$$<=> \begin{pmatrix} w_1 & w_2 \end{pmatrix} . \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + b = 0$$
$$<=> w_1 x_1 + w_2 x_2 + b = 0$$
$$<=> x2 = \frac{-b - w_1 x_1}{w_2}$$

Ainsi, nous pouvons tracer cette droite sur notre grille de points de test.

```
    # Recuperation de w et b
        w = svm2d.coef_[0]
        b = svm2d.intercept_

    # Determination de l'equation de la droite de separation
        x2 = (-b - w[0] * r1)/w[1]

    # Affichage de la droite
        plt.plot(r1,x2, 'k')
```

Le principe de ce classifieur se reposant sur la marge maximale, nous décidons de la montrer en même temps que la frontière.
Pour la calculer, nous effectuons une translation de la droite de séparation.

```
    # Marges superieure et inferieure
    delta1 = (1 - b - w[0] * r1)/w[1]
        delta2 = (-1 - b - w[0] * r1)/w[1]
```

Après tous ces tracés et ces colorations, nous obtenons cette grille :

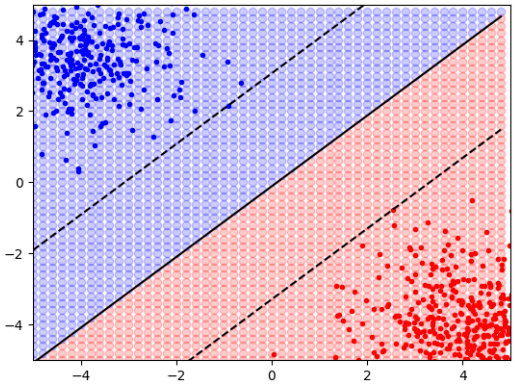

FIGURE 2 – Prédiction cas jeu linéairement séparable

2 Cas non séparable

Dans le cas d'un jeu non séparable, la frontière sera toujours une droite, mais la marge ne sera plus stricte et la frontière plus optimale.
En d'autres termes, la grille sera toujours séparée en deux couleurs, mais certains points de la base d'apprentissage seront visuellement "mal classés".

Prenons un exemple. On choisi un noyau $C = 2$, et les points suivants :

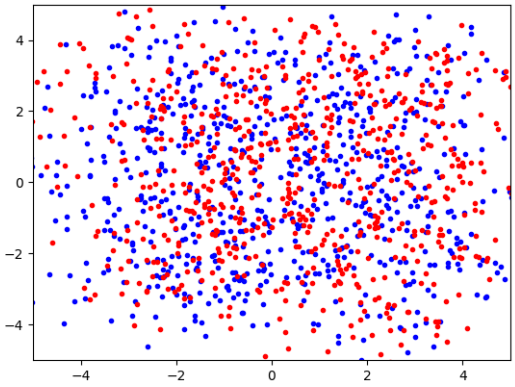

FIGURE 3 – Base d'apprentissage avec jeu non séparable linéairement

Nous appelons le même programme que précédemment.

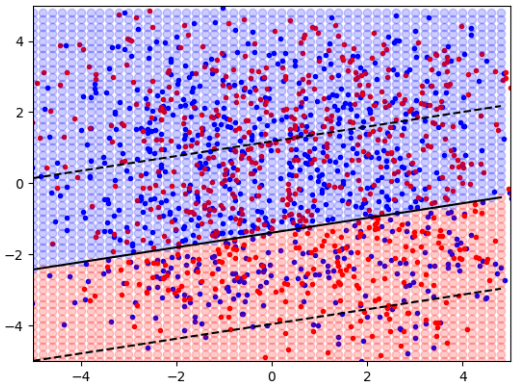

FIGURE 4 – Prédiction, frontière et marge avec jeu non séparable linéairement

Si nous analysons le résultat, nous pouvons dire qu'un point a plus de chance d'être bleu si son ordonnée est élevée. Seulement, le C choisi est faible, et par conséquent, il favorise la maximisation de la marge et conduit donc à plus d'erreur.
En effet, en faisant varier cet hyperparamètre, nous voyons que plus ce dernier est faible, plus la marge est importante.

De façon globale, il faut donc être prudent dans le cas d'un jeu de donnée non séparable linéairement.

3 Classification non linéaire

Dans ce dernier cas, le jeu de données constituant la base d'apprentissage est séparable, mais la frontière ne peut pas prendre la forme d'une droite.

Nous utilisons donc non plus un noyau fini ou infini, mais un noyau gaussien.
Dans notre programme, nous changeons donc la phase d'apprentissage afin de le définir.

```
1     # Apprentissage du modele SVM non lineaire
2         svm2d = svm.SVC(C = C, kernel= 'rbf', gamma=1/(2*sigma**2))
```

Pour visualiser le changement, nous prenons une nouvelle grille et rentrons le jeu de données d'apprentissage suivant :

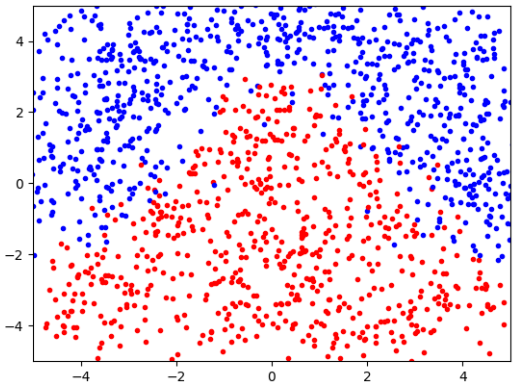

FIGURE 5 – Base d'apprentissage cas non linéaire

Après apprentissage, la phase de prédiction a lieu. Cette dernière n'a pas changé par rapport aux cas précédents. Dans notre cas, nous avons $C = 1$ et $\sigma = 1$. Nous obtenons la prédiction suivante :

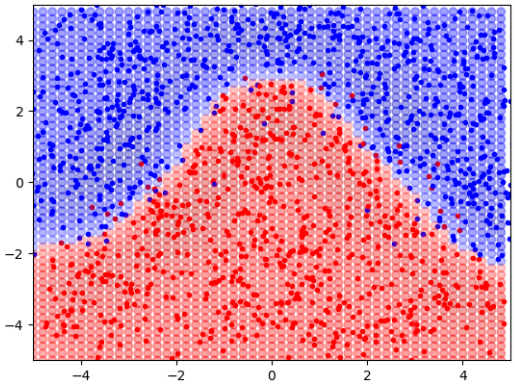

FIGURE 6 – Prédiction cas non linéaire

Nous pouvons visualiser la frontière. Cette dernière n'est donc plus une droite, mais une courbe (que nous ne traçons pas).
Cette méthode de tracé ne permet que difficilement de visualiser la marge, mais après de nombreux tests, nous voyons que σ joue sur la frontière en elle même : plus il augmente, plus la frontière épousera la forme des points de données. Tandis que C influe la marge : si il est élevé, il marge va diminuer.

Selon le niveau de précision que nous souhaitons, il est donc important de correctement définir ces paramètres.

Partie II : Problème multi-classe et techniques de validation croisée

La classification multi-classes consiste en un problème de répartition des données en plusieurs catégories. La classification binaire en fait partie. En effet, elle sépare des données en deux groupes.
Nous verrons dans ce TP que nous pouvons utiliser une classification bi-classes (ou binaire) pour résoudre une problème 4-classes.

Lors de cet exercice, nous utilisons un jeu de données réél qui consiste à répartir des défauts de rails en quatre catégories.

Le principe est le suivant : nous allons considérer pour chacune des quatre catégories possible que soit un défaut en fait partie, soit non. Nous allons ensuite faire le lien entre les quatre classements.
Donnons un exemple plus concret : on considère un défaut A. Il n'est pas dans la catégorie 1, il est dans la 2, il n'est pas dans la 3 et n'est pas dans la 4. Nous pouvons lier les informations est dire que le défaut A est de catégorie 2. C'est la méthode un-contre-tous.
C'est ce raisonnement que nous allons mettre en pratique grâce à la classification SVM et les mêmes outils Python que dans la partie précédente.

1 Classifieurs binaires

Afin d'appliquer la méthode un contre tous, nous créons 4 classifieurs SVM linéaires.

```
1  # Creation des 4 modeles (un pour chaque valeur de y)
2  svm1 = svm.LinearSVC(C=math.inf, max_iter=100000)
3  svm2 = svm.LinearSVC(C=math.inf, max_iter=100000)
4  svm3 = svm.LinearSVC(C=math.inf, max_iter=100000)
5  svm4 = svm.LinearSVC(C=math.inf, max_iter=100000)
```

Nous pouvons maintenant réaliser les mêmes étapes que dans la première partie afin de calculer, pour chacun des quatre classifieur, son taux de reconnaissance sur notre base d'apprentissage, default rails.

```
1  # Liste de nos quatre classifieurs
2  svmtab = [svm1, svm2, svm3, svm4]
3
4  # Boucle de calcul de taux d'apprentissage
5  for i in range(4): # Parcours pour chacun des modeles
6      # Si y = [1, 2, 3 ou 4], on met 1, sinon : -1 :
7      yi = 2*(y==(i+1))-1
8
9      # On applique le modele (enfin les 4)
10     svmtab[i].fit(X, yi)
11     Ypred = svmtab[i].predict(X)
12
13     # On calcule l'erreur de chacun de nos 4 modeles
14     Err = np.mean(Ypred != yi)*100
15     print("Erreur modele classification binaire", i, " :")
16     print(Err)
```

Autrement-dis, pour chaque itération de notre boucle, nous réalisons un apprentissage sur notre jeu de donnée et une colonne de -1 et 1 (1 : fait parti de la catégorie, -1 : n'en fait pas parti). Nous réalisons ensuite une prédiction sur le jeu de données. Enfin, nous calculons l'erreur de notre modèle afin d'obtenir l'équivalent de son taux de reconnaissance (sont complément à un).

2 Combinaison des classifieurs binaires

Nous allons maintenant combiner nos quatre classifieurs afin d'appliquer la méthode un-contre-tous.
La catégorie prédite par ce nouveau classifieur correspond au classifieur binaire dont la valeur est maximale.

```
1     # On transforme nos 4 modeles en 1, on les combine :
2     G[:,i] = svmtab[i].decision_function(X)
3     # Prediction du model globale
4     ypred = []
5     ypred.append(np.argmax(G, axis=1) + 1)
```

Nous calculons ensuite son erreur.

```
1     # Et son erreur
2     print("Erreur␣modele␣combinaison␣des␣classifieurs␣binaire␣:")
3     Err = np.mean(ypred != y)*100
4     print(Err)
```

En faisant tourner le programme, voici ce que nous obtenons :

```
Erreur modele classification binaire 0  :
0.0
Erreur modele classification binaire 1  :
0.0
Erreur modele classification binaire 2  :
0.0
Erreur modele classification binaire 3  :
0.0
Erreur modele combinaison des classifieurs binaire :
0.0
```

FIGURE 7 – Taux de reconnaissance des classifieurs binaire

Nous observons un taux de reconnaissance de 100% pour chacun de nos quatre classifieurs. C'est tout à fait normal, nous faisons une prédiction sur les mêmes données que celles utilisées pour l'apprentissage. Cela ne laisse pas de place à l'erreur.

Nous devrions diviser le jeu de données en deux bases : une pour l'apprentissage et une pour le test, mais la taille de notre jeu ne nous le permet pas. En effet, nous n'aurions pas suffisemment de données.
Pour évaluer les performances en généralisation du classifieur, nous allons donc utiliser la technique de validation croisée Leave-One-Out (LOO).

3 Estimation de l'erreur de généralisation par validation croisée

Dans cette partie, nous allons justement implémenter la méthode de validation croisée LOO.
Dans cette méthode, on divise toujours les données en bases d'entraînement et de test, mais cette fois d'une façon différente. Au lieu de diviser les données en deux sous-ensembles, comme son nom l'indique, on choisit une seule observation comme données de test et tout le reste est étiqueté comme données d'entraînement. Puis, la deuxième observation est sélectionnée comme données de test et le modèle est entraîné sur les données restantes. Ce processus se poursuit pour toutes les observations. À chaque fois, on entraîne le modèle sur les données et on regarde s'il a pu trouver la bonne réponse pour l'observation qu'on a laissé à côté.

Ainsi, dans notre cas, 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.
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. 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.

```
1     def loo(X,y,svmtab):
2         Erreur = 0
3         for i in range(140):
4             X_i = np.delete(X, i, axis=0)
5             y_i = np.delete(y, i)
6 
7             Gbis = np.zeros(4)
8             for j in range(4):
9                 yi = 2*(y_i==(j+1))-1
10                svmtab[j].fit(X_i, yi)
11                yPrevision = svmtab[j].predict(X_i)
12                Gbis[j] = svmtab[j].decision_function([X[i,:]])
13            fx = np.argmax(Gbis) + 1
14            Err = fx != y[i]
15            Erreur += Err
16 
17        monErreur = Erreur/140 * 100
18        print("Erreur de validation croisee :")
19        print(monErreur)
20        return monErreur
```

Nous appelons ensuite cette fonction avec nos données et nos classifieurs.

```
1     # Appel LOO :
2     loo(X,y, svmtab)
```

En lançant ce programme, nous obtenons un taux de reconnaissance de 88.6% :

```
Modele multiclass par validation croisée (LOO)
Erreur de validation croisée :
11.428571428571429
```

FIGURE 8 – Taux de reconnaissance modèle multi-class méthode LOO

Ces taux ont pour paramètre d'hyperplan $C = \infty$.

Maintenant que nous avons une fonction LOO toute faite, nous pouvons faire tourner l'algorithme pour différentes valeurs de C afin d'en determiner le meilleur.

```
    mistakes = []
    mesC = [0.01,0.1,1,5,10,100]
    for c in mesC:
        svm1 = svm.LinearSVC(C=c, max_iter=100000)
        svm2 = svm.LinearSVC(C=c, max_iter=100000)
        svm3 = svm.LinearSVC(C=c, max_iter=100000)
        svm4 = svm.LinearSVC(C=c, max_iter=100000)

        svmtabLoo = [svm1, svm2, svm3, svm4]
        print("pour C =", c)
        mistakes.append(loo(X,y,svmtabLoo))

    print("Le meilleur C est :")
    print(mesC[mistakes.index(min(mistakes))])
```

Nous obtenons ainsi :

```
pour C = 0.01
Erreur de validation croisée :
16.428571428571427
pour C = 0.1
Erreur de validation croisée :
10.714285714285714
pour C = 1
Erreur de validation croisée :
7.857142857142857
pour C = 5
Erreur de validation croisée :
9.285714285714286
pour C = 10
Erreur de validation croisée :
10.0
pour C = 100
Erreur de validation croisée :
10.714285714285714
Le meilleur C est :
1
```

FIGURE 9 – Choix du paramètre d'hyperplan grâce à la méthode LOO

Nous pourrions maintenant calculer l'erreur du modèle pour $C = 1$. Pour cela, il nous faudrait d'autres données ou alors refaire un découpage de notre jeu de défaut de rails.

Après ces différents tests, nous observons tout de même que le problème avec cette méthode, est qu'on augmente fortement le temps de calcul puisqu'on fait entraîner autant de modèles que le nombre de points dans le jeu complet. Il serait donc déconseillé d'utiliser cette méthode en travaillant avec un grand volume de donnée, ce ne sera pas facile de tirer des conclusions.