TP 6

Méthodes à noyaux

Machines à vecteurs de support (SVM)

Les machines à vecteurs de support (SVM : *Support Vector Machines*) sont une classe des méthodes d'apprentissage statistique basées sur le principe de la maximisation de la marge (séparation des classes). Il existe plusieurs formulations (linéaires, versions à noyaux) qui peuvent s'appliquer sur des données séparables (linéairement) mais aussi sur des données non séparables.
Les avantages des SVM :

- Très efficaces en dimension élevée.
- Ils sont aussi efficaces dans le cas où la dimension de l'espace est plus grande que le nombre d'échantillons d'apprentissage.
- N'utilisent pas tous les échantillons d'apprentissage, mais seulement une partie (les vecteurs de support). En conséquence, ces algorithmes demandent moins de mémoire.

Désavantages :

- Si le nombre d'attributs est beaucoup plus grand que le nombre d'échantillons, les performances seront moins bonnes.
- Comme il s'agit de méthodes de discrimination entre les classes, elles ne fournissent pas directement des estimations de probabilités.

Scikit-learn

Dans Scikit-learn, les SVM sont implémentées dans le module `sklearn.svm`. Dans cette partie nous allons nous intéresser à la version à noyaux (Scikit-learn utilise la bibliothèque LibSVM déjà discutée).
Relisez le TP SVM linéaire (partie Scikit-learn) pour vous remettre dans le contexte (quels sont les classes Python utilisées et leurs paramètres). La documentation de scikit-learn sur les SVM vous sera utile.
Nous allons reprendre la classification sur les données Iris.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm, datasets
from sklearn.model_selection import train_test_split

# Chargement des données
iris = datasets.load_iris()
X, y = iris.data, iris.target
```

```
# On conserve 50% du jeu de données pour l'évaluation
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.5)
```

Question

Refaites la classification de la base de données `iris` mais avec un noyau gaussien. Testez l'effet du paramètre d'échelle du noyau (`gamma`) et du paramètre de régularisation `C`.

Comme dans le TP précédent, nous pouvons afficher la frontière de décision en ne conservant que deux variables explicatives :

```
X, y = iris.data[:,:2], iris.target
# On conserve 50% du jeu de données pour l'évaluation
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.5)
clf = svm.SVC(C=0.1, kernel='rbf', gamma=0.25)
clf.fit(X_train, y_train)

# Pour afficher la surface de décision on va discrétiser l'espace avec un
pas h
h = .02
# Créer la surface de décision discretisée
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))

# Surface de décision
Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)

plt.contourf(xx, yy, Z, cmap=plt.cm.coolwarm, alpha=0.8)
# Afficher aussi les points d'apprentissage
plt.scatter(X_train[:, 0], X_train[:, 1], label="train", edgecolors='k',
c=y_train, cmap=plt.cm.coolwarm)
plt.scatter(X_test[:, 0], X_test[:, 1], label="test", marker='*', c=y_test,
cmap=plt.cm.coolwarm)
plt.xlabel('Sepal length')
plt.ylabel('Sepal width')
plt.title("SVM RBF")
```

Question

Que constatez-vous par rapport au TP précédent ?

Jeu de données Digits

Reprenons notre base de données Digits de chiffres manuscrits.

```
from sklearn.datasets import load_digits
digits = load_digits()
X, y = digits.data, digits.target
print(X.shape)
print(y.shape)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
```

Question

Réalisez une classification par une SVM linéaire et une SVM à noyau gaussien du jeu de données Digits. Comment est choisi le paramètre `gamma` dans scikit-learn ? Testez différentes valeurs de ce paramètre pour évaluer son influence. En particulier, testez les paramètres `gamma='auto'` et `gamma='scale'`. À quoi correspondent-ils ?

Question :

Réalisez une analyse en composante principale (ACP) et gardez les 2 premières composantes principales (voir la documentation Scikit-learn). Ensuite faites une classification avec un noyau gaussien et affichez les points de test ainsi que la surface de décision (reprendre le code du TP SVM linéaire). Comparez avec une SVM linéaire.

Question

Réalisez une recherche par grille afin de déterminer sur le jeu de données Digits complet (sans l'ACP) :

- le meilleur noyau à utiliser,
- la meilleure valeur de `C`,
- la meilleure valeur de `gamma` (ou le degré du polynôme pour un noyau polynomial).
- la meilleure valeur de `n_components` de l'ACP

Question

(optionnel) Combien de composantes faut-il garder au minimum dans l'ACP pour classer correctement au moins 97% des images ? À quel facteur de réduction de dimension cela correspond-il ?

TP 5

SVM linéaires

Les machines à vecteurs de support (SVM : *Support Vector Machines*) sont une classe de méthodes d'apprentissage statistique basées sur le principe de la maximisation de la marge (séparation des classes). Il existe plusieurs formulations (linéaires, versions à noyaux) qui peuvent s'appliquer sur des données séparables (linéairement) mais aussi sur des données non séparables.
Les avantages des SVM :

- Très efficaces en dimension élevée.
- Ils sont aussi efficaces dans le cas où la dimension de l'espace est plus grande que le nombre d'échantillons d'apprentissage.
- Pour la décision, n'utilisent pas tous les échantillons d'apprentissage, mais seulement une partie (les vecteurs de support). En conséquence, ces algorithmes demandent moins de mémoire.

Désavantages :

- Si le nombre d'attributs est beaucoup plus grand que le nombre d'échantillons, les performances sont moins bonnes.
- Comme il s'agit de méthodes de discrimination entre les classes, elles ne fournissent pas d'estimations de probabilités.

Jeu de données Iris

Dans Scikit-learn, les SVM sont implémentées dans le module `sklearn.svm`. Dans cette partie nous allons nous intéresser à la version linéaire (Scikit-learn utilise les bibliothèques libLinear et libSVM).

Nous allons utiliser le jeu de données Iris déjà rencontré dans les séances précédentes. Pour pouvoir afficher les résultats, on va utiliser seulement les premiers deux attributs (longueur et largeur des sépales).

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm, datasets
from sklearn.model_selection import train_test_split

# Chargement des données
iris = datasets.load_iris()
```

Pour commencer, nous ne conservons que les deux premiers attributs du jeu de données :

```
X, y = iris.data[:, :2], iris.target
# On conserve 50% du jeu de données pour l'évaluation
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.5)
```

Nous pouvons maintenant entraîner une machine à vecteur de support linéaire :

```
C = 1.0 # paramètre de régularisation
lin_svc = svm.LinearSVC(C=C)
lin_svc.fit(X_train, y_train)
```

Question

Calculez le score d'échantillons bien classifiés sur le jeu de données de test.

Visualisons maintenant la surface de décision apprise par notre modèle :

```
# Créer la surface de décision discretisée
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
# Pour afficher la surface de décision on va discrétiser l'espace avec un
pas h
h = max((x_max - x_min) / 100, (y_max - y_min) / 100)
xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))

# Surface de décision
Z = lin_svc.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)

plt.contourf(xx, yy, Z, cmap=plt.cm.coolwarm, alpha=0.8)
# Afficher aussi les points d'apprentissage
plt.scatter(X_train[:, 0], X_train[:, 1], label="train", edgecolors='k',
c=y_train, cmap=plt.cm.coolwarm)
plt.scatter(X_test[:, 0], X_test[:, 1], label="test", marker='*', c=y_test,
cmap=plt.cm.coolwarm)
plt.xlabel('Sepal length')
plt.ylabel('Sepal width')
plt.title("LinearSVC")
```

Question :

Testez différentes valeurs pour le paramètre C. Comment la frontière de décision évolue en fonction de C ?

Question

D'après la visualisation ci-dessus, ce modèle vous paraît-il adapté au problème ? Si non, que peut-on faire pour l'améliorer ?

Nous verrons dans le prochain TP que scikit-learn permet de manipuler des machines à vecteurs de support avec des noyaux non-linéaires dans la classe `SVC`.

Les modèles linéaires `LinearSVC()` et `SVC(kernel='linear')`, comme nous l'avons déjà dit, produisent des résultats légèrement différents à cause du fait qu'ils optimisent des fonctions de coût différentes mais aussi à cause du fait qu'ils gèrent les problèmes multi-classe de manière différente (linearSVC utilise *One-vs-All* et SVC utilise *One-vs-One*).

```
lin_svc = svm.LinearSVC(C=C).fit(X_train, y_train)
svc = svm.SVC(kernel='linear', C=C).fit(X_train, y_train)

titles = ['SVC with linear kernel', 'LinearSVC (linear kernel)']

fig = plt.figure(figsize=(12, 4.5))

for i, clf in enumerate((svc, lin_svc)):
    plt.subplot(1, 2, i + 1)
    plt.subplots_adjust(wspace=0.4, hspace=0.4)
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
    # Utiliser une palette de couleurs
    Z = Z.reshape(xx.shape)
    plt.contourf(xx, yy, Z, cmap=plt.cm.coolwarm, alpha=0.8)
    # Afficher aussi les points d'apprentissage
    plt.scatter(X_train[:, 0], X_train[:, 1], label="train", edgecolors='k',
c=y_train, cmap=plt.cm.coolwarm)
    plt.scatter(X_test[:,  0],  X_test[:,  1],  label="test",  marker='*',
c=y_test, cmap=plt.cm.coolwarm)
    plt.xlabel('Sepal length')
    plt.ylabel('Sepal width')
    plt.title(titles[i])
plt.show()
```

Pour l'instant, nous n'avons exploité que deux variables explicatives. Néanmoins, l'intérêt des machines à vecteur de support linéaires est qu'il est souvent plus facile de trouver des hyperplans séparateurs dans des espaces de grande dimension.

Question

Réalisez l'optimisation d'une nouvelle machine à vecteur de support linéaire mais en utilisant les quatre attributs du jeu de données Iris. Le score de classification en test a-t-il augmenté ? Pourquoi ?

Jeu de données Digits

Le jeu de données Digits est une collection d'images de chiffres manuscrits (nous l'avons déjà utilisé dans le TP sur les forêts aléatoires). Elles peuvent se charger directement depuis scikit-learn :

```
from sklearn.datasets import load_digits
digits = load_digits()
X, y = digits.data, digits.target
```

Question :

Utilisez les données Digits pour construire un classifieur LinearSVC et évaluez-le. Si le temps d'apprentissage est trop long, sélectionnez une partie plus petite de la base d'apprentissage (par exemple 10000 échantillons). Pour quelle valeur de C on obtient le meilleurs résultats de généralisation ?

TP 4
Les forêts aléatoires

Méthodes d'agrégation

Les méthodes ensemblistes (ou d'agrégation) pour les algorithmes d'apprentissage statistique (en anglais : *ensemble learning*) sont basées sur l'idée de combiner les prédictions de plusieurs prédicteurs (ou classifieurs) pour une meilleure généralisation et pour compenser les défauts éventuels de prédicteurs individuels.
En général, on distingue deux familles de méthodes de ce type :

1. Méthodes par moyennage (*bagging*, forêts aléatoires) où le principe est de faire la moyenne de plusieurs prédictions en espérant un meilleur résultat suite à la réduction de variance de l'éstimateur moyenne.
2. Méthodes adaptatives (*boosting*) où les paramètres sont itérativement adaptés pour produire un meilleur mélange.

Dans la suite nous explorerons chacune de ces classes d'algorithme en Scikit-learn et présenterons quelques comparaisons.

Bagging

Les méthodes de type *bagging* construisent plusieurs instances d'un estimateur, calculées sur des échantillons aléatoires tirés de la base d'apprentissage (et éventuellement une partie des attributs, également sélectionnés de façon aléatoire), et ensuite combine les prédictions individuelles en réalisant leur moyenne pour réduire la variance de l'estimateur. Leur avantage principal réside dans le fait qu'ils construisent une version améliorée de l'algorithme de base, sans demander de modification de cet algorithme. Le prix à payer est un coût de calcul plus élevé. Comme elles réduisent le sur-apprentissage, les méthodes *bagging* fonctionnent très bien avec des prédicteurs « forts ». Par contraste, les méthodes *boosting* sont mieux adaptées à des prédicteurs faibles (*weak learners*).
Dans Scikit-learn, les méthodes de *bagging* sont implémentées via la classe `BaggingClassifier` et `BaggingRegressor`. Les constructeurs prennent en paramètres un estimateur de base et la stratégie de sélection des points et attributs :

- `base_estimator` : optionnel (default=None). Si None alors l'estimateur est un arbre de décision.
- `max_samples` : la taille de l'échantillon aléatoire tiré de la base d'apprentissage.
- `max_features` : le nombre d'attributs tirés aléatoirement.
- `bootstrap` : boolean, optionnel (default=True). Tirage des points avec remise ou non.
- `bootstrap_features` : boolean, optionnel (default=False). Tirage des attributs avec remise ou non.
- `oob_score` : boolean. Estimer ou non l'erreur de généralisation OOB (*Out of Bag*).

Le code suivant construit un ensemble des classifieurs. Chaque classifieur de base est un `KNeighborsClassifier` (c'est-à-dire k-plus-proches-voisins), chacun utilisant au maximum 50% des points pour son apprentissage et la moitié des attributs (*features*) :

```
from sklearn.ensemble import BaggingClassifier
from sklearn.neighbors import KNeighborsClassifier
bagging = BaggingClassifier(KNeighborsClassifier(), max_samples=0.5,
max_features=0.5)
```

Dans cet exemple nous allons utiliser la base de données digits, qui contient 10 classes (images des chiffres en écriture manuscrite). Il y a 1797 éléments, chaque élément a 64 attributs (8 pixels par 8).

```
from sklearn.datasets import load_digits
digits = load_digits()

# Affichage des 10 premières images
import matplotlib.pyplot as plt
fig = plt.figure()
for i, digit in enumerate(digits.images[:10]):
    fig.add_subplot(1,10,i+1)
    plt.imshow(digit)
plt.show()
```

Pour ce TP, nous allons utiliser comme classifieur de base un arbre de décision DecisionTreeClassifier. Ce classifieur nous permet d'établir des performances de référence (c'est un ensemble à 1 modèle).

```
import numpy as np
from sklearn import tree
from sklearn.ensemble import BaggingClassifier

X, y = digits.data, digits.target
clf = tree.DecisionTreeClassifier()
clf.fit(X, y)
accuracy = clf.score(X,y)
print(accuracy)
```

Sur la base d'apprentissage accuracy = 1 . Pour plus de réalisme, découpons la base de données en un jeu d'apprentissage et un je de test afin de voir le comportement de généralisation de l'arbre sur des données différentes des celles d'apprentissage :

```
from sklearn.model_selection import train_test_split
# 90% des données pour le test, 10% pour l'apprentissage
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.90)

clf = tree.DecisionTreeClassifier()
clf.fit(X_train, y_train)

Z = clf.predict(X_test)
accuracy = clf.score(X_test,y_test)
print(accuracy)
```

Question :

Construire la variance de la valeur accuracy sur 100 tirages pour la séparation apprentissage/test. Que pouvons-nous conclure ?

Pour comparer, construisons mainntenant un classifieur *bagging* sur nos données, toujours basé sur les DecisionTreeClassifier :

```
clf = BaggingClassifier(tree.DecisionTreeClassifier(), max_samples=0.5, max_features=0.5, n_estimators=200)
```

L'apprentissage et l'évaluation de cet ensemble se font de la façon habituelle :

```
clf.fit(X_train, y_train)
Z = clf.predict(X_test)
accuracy=clf.score(X_test,y_test)
```

Question :

Calculer la variance de la valeur `accuracy` sur 100 tirages pour la séparation apprentissage/test. Comparer avec la variance du classifieur de base. Que pouvons-nous conclure ?

Question :

Construire le graphique `accuracy` vs `n_estimators`. Que constatez-vous ?

Question :

Faites varier les parametres `max_samples` et `max_features`. Pour quelles valeurs on obtient le meilleur résultat ? On pourra notamment utiliser `GridSearchCV` pour réaliser une recherche systématique.

Forêts aléatoires

L'algorithme des forêts aléatoires propose une optimisation des arbres de décision. Il utilise le même principe que le *bagging*, mais avec une étape supplémentaire de randomisation dans la sélection des attributs des nœuds dans le but de réduire la variance de l'estimateur obtenu. Les deux objets Python qui implémentent les forêts aléatoires sont `RandomForestClassifier` et `RandomForestRegressor`.
Les paramètres les plus importants sont :

- `n_estimators` : integer, optional (default=10). Le nombre d'arbres.
- `max_features` : le nombre d'attributs à considérer à chaque split.
- `max_samples` : la taille de l'échantillon aléatoire tiré de la base d'apprentissage.
- `min_samples_leaf` : le nombre minimal d'éléments dans un nœud feuille.
- `oob_score` : boolean. Estimer ou non l'erreur de généralisation OOB (*Out of Bag*).

Par la suite nous allons refaire la classification sur la base Digits en utilisant un classifieur `RandomForestClassifier`. Comme d'habitude, on sépare les données en gardant 10% pour l'apprentissage et 90% pour le test.

```
digits = load_digits()
X, y = digits.data, digits.target

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.90)
```

On peut désormais créer et entraîner notre modèle :

```
from sklearn.ensemble import RandomForestClassifier
clf = RandomForestClassifier(n_estimators=200)
clf.fit(X_train, y_train)
```

Puis réaliser les prédictions et calculer le score de test :

```
y_pred = clf.predict(X_test)
accuracy = clf.score(X_test,y_test)
print(accuracy)
```

Question :

Comment la valeur de la variable `accuracy` se compare avec le cas *bagging* qui utilise le même nombre d'arbres (200 dans notre cas) ?

Question :

Construire la variance de la valeur accuracy sur 100 tirages pour la séparation apprentissage/test. Que pouvons-nous conclure en comparant avec la séction précedente (*bagging*) ?

Question :

Construire le graphique accuracy vs n_estimators. Que constatez-vous ? A partir de quelle valeur on n'améliore plus ?

Question :

Regardez dans la documentation les *ExtraTreesClassifier* et refaites la classification avec ce type de classifieur. Comparez avec *RandomForestClassifier*.

Boosting

Le principe du *boosting* est d'évaluer une séquence de classifieurs faibles (*weak learners*) sur plusieurs versions légèrement modifiées des données d'apprentissage. Les décisions obtenues sont alors combinées par une somme pondérée pour obtenir le modèle final.
Avec scikit-learn, c'est la classe AdaBoostClassifier qui implémente cet algorithme. Les paramètres les plus importants sont :

- n_estimators : integer, optional (default=10). Le nombre de classifieurs faibles.
- learning_rate : contrôle la vitesse de changement des poids par itération.
- base_estimator : (default=DecisionTreeClassifier) le classifieur faible utilisé.

Dans la suite nous allons refaire la classification sur la base Digits en utilisant un classifieur RandomForestClassifier :

```
from sklearn.ensemble import AdaBoostClassifier

digits = load_digits()
X, y = digits.data, digits.target

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.90)

# AdaBoost basé sur 200 arbres de décision
clf                                                                    =
AdaBoostClassifier(base_estimator=tree.DecisionTreeClassifier(max_depth=5),
n_estimators=200, learning_rate=2)
clf.fit(X_train, y_train)
accuracy = clf.score(X_test, y_test)
print(accuracy)
```

Question :

Le paramètre max_depth contrôle la profondeur de l'arbre. Essayez plusieurs valeurs pour voir l'impact de l'utilisation d'un classifieur faible vs plus fort (max_depth élevé ou éliminer le paramètre). Testez aussi l'effet du paramètre learning_rate et le nombre de classifieurs.