TP n°2: Classification par SVM et kNN

Le compte-rendu doit être rédigé en Jupyter-Notebook, Markdown ou LaTeX. Il doit comporter à la fois les codes mais également **et surtout** *des commentaires et interprétations clairs et pertinents concernant (i) vos choix, (ii) vos résultats en rapport avec les notions vues en cours.*

1 Descriptif du TP

Dans ce second TP, nous allons développer et manipuler nos premiers codes de classification des données de la base `MNIST` vue en long et en large au TP1.

Spécifiquement, nous réaliserons dans un premier temps la classification de deux chiffres de la base `MNIST` au moyen des diverses approches de classification de type SVM:

1. classification linéaire, et analyse de l'hyperplan séparateur
2. classification par noyau non-linéaire (rbf)
3. classification linéaire ou non-linéaire en utilisant une représentation HOG (histogramme des gradients orientés)

avant de généraliser à l'ensemble des chiffres de la base. En parallèle, nous étudierons également l'algorithme kNN dont nous mettrons en évidence les limites lorsque les données sont de trop grandes dimensions ou trop nombreuses.

2 Classification par SVM

2.1 Classification binaire linéaire

En utilisant les résultats du TP1, commencez par sélectionner une sous-partie à 2 chiffres de la base de données `MNIST`: par exemple l'ensemble des `'3'` et des `'4'`. Ceux-ci formeront notre jeu de données `X,y`. N'oubliez pas, bien sûr, de filtrer à la fois la matrice des données `X` ainsi que le vecteur des étiquettes `y`. Vérifiez que votre filtrage fonctionne en imprimant quelques images. On divisera ensuite ces données en un ensemble d'entrainement `X_train,y_train` et un ensemble de validation `X_test,y_test` dont vous êtres libres de choisir les tailles (on pourra utiliser `np.shape()` pour évaluer la taille de `X`).

Vous allez maintenant utiliser l'algorithme SVM. Pour cela, il vous faut charger la librairie:

```
from sklearn import svm
```

La bibliothèque `sklearn` utilise une façon très générale de construire et utiliser des classifieurs. Dans le cas de SVM, on pourra créer une "classe" classifieur via

```
classifier = svm.SVC([...])
```

Explorer la documentation de `svm.SVC` afin de construire un classifieur binaire **linéaire** nommé `classifier`.

Une fois générés, l'entrainement des classifieurs (SVM, kNN, réseaux de neurones), fournis par `sklearn`, s'effectue toujours par la méthode

```
classifier.fit(training_data_matrix,training_label_vector)
```

Entrainez votre classifieur sur votre propre base de données d'**entrainement**.

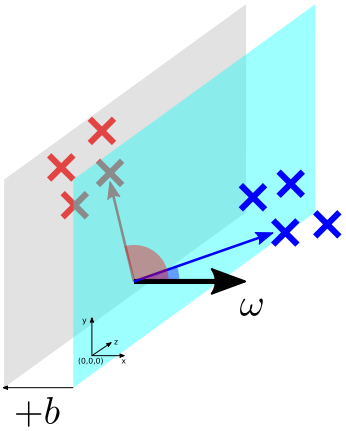
Une fois entrainé, il est ensuite possible de tester le classifieur sur n'importe quelle base de données et d'en évaluer les performances. Pour cela,

- pour n'importe quel ensemble de données `data_matrix`, on peut obtenir les étiquettes prédites par le modèle via `classifier.predict(data_matrix)`
- pour des données pré-étiquetées `(data_matrix,label_vector)`, on peut récupèrer le `score` de classification correcte via `classifier.score(data_matrix,label_vector):`

```
y_est = classifier.predict(data_matrix)
classifier.score(data_matrix,label_vector)
```

Testez votre classifieur SVM linéaire (i) d'une part sur la base de données d'entrainement, et (ii) d'autre part sur la base de données de test. Que remarquez-vous? Est-ce cohérent? Vérifiez ensuite que le nombre de données communes, en pourcentage, au vecteur `label_vector` et au vecteur `y_yest` correspond bien au score retourné par `classifier.score`.

BONUS. Pour finir cette section, tentons de voir ce qu'il se passe au sein de la "boîte noire" `classifier` une fois entrainée. Rappelons tout d'abord que, dans ce cas binaire, SVM construit un hyperplan séparateur entre les deux classes



En particulier, SVM estime un vecteur $\omega \in \mathbb{R}^p$ orthogonal à l'hyperplan: il est accessible via `classifier.coef_[0]` (le `[0]` est nécessaire puisque pour plus de deux classes, `classifier.coef_` contient l'ensemble des vecteurs ω pour tous les hyperplans). SVM estime également l'ordonnée à l'origine $b \in \mathbb{R}$ via `classifier.intercept_[0]`. Rappelons alors (voir le dessin ci-dessus) que ces deux attributs permettent d'évaluer si une donnée $x \in \mathbb{R}^p$ se trouve d'un côté ou de l'autre de l'hyperplan selon que

$$\omega^\mathsf{T} x + b = \|\omega\| \|x\| \cos(\angle(\omega, x)) + b \lessgtr 0.$$

Validez qu'en effet:

- tous les points x **d'une même classe** (disons les `'3'`), et qui sont correctement classés, ont une valeur de $\omega^\mathsf{T} x + b$ de signe constant; et que les points correctement classés de l'autre classe ont un signe opposé;
- que les points x mal classés ont un signe opposé à ce qu'il devrait être.

Regardez précisément la valeur absolue des scores pour les données bien et mal classées: que cela signifie-t-il selon vous (regardez les images correspondantes) lorsqu'une donnée a un score de valeur absolue élevée? Cette information pourrait-elle être utile en pratique et pourquoi?

Astuce: `a@b` *calcule le produit scalaire* $a^\mathsf{T} b$ *entre deux vecteurs* `a` *et* `b`

2.2 Classification générique

Reproduisez les étapes de construction du classifieur SVM, désormais pour:

1. un SVM à noyau non-linéaire (par exemple `rbf`): référez vous à la documentation pour vous aider
2. un SVM linéaire ou non-linéaire, mais désormais sur les 10 classes de la base `MNIST`.

Affinez vos choix en explorant (et en **comprenant bien!**) les différentes options offertes par `svm.SVC` de sorte à augmenter les scores de classification. Quelles sont les choix qui vous semblent être les plus judicieux, et pourquoi?

2.3 Classification des représentations HOG

Au lieu de travailler sur les données `MNIST` "brutes", nous allons maintenant plutôt en extraire une représentation "orientée image" pertinente: l'histogramme des gradients orientés (HOG) déjà vus en cours. Nous avons besoin pour cela de charger la bibliothèque

```
from skimage.feature import hog
```

Celle-ci nous permet d'utiliser la fonction `hog(image)` qui transforme une **image** (attention, pas sa version vectorisée!) `image` en une description sous forme d'histogramme de gradients. Pour effectuer un premier test, prenez un vecteur de la base de données `X` (donc de taille 784), transformez le (grâce à `reshape`) en une image de taille 28×28, vérifiez visuellement que l'image est correcte (via `imshow`) et ensuite extrayez avec `hog` le vecteur de ses gradients orientés. Quelle est la taille de ce vecteur? Jouez avec les paramètres de `hog` et voyez l'effet sur le vecteur de représentations: quel est l'impact des options principales?

Une fois des paramètres pertinents choisis, créez une nouvelle matrice `X_hog` qui contiendra (en lignes) les représentations successives des données de `X`. On travaillera désormais, non plus avec la matrice `X`, mais avec la matrice `X_hog`.

Une fois `X_hog` établie, reprenez l'ensemble des questions des Sections 2.1–2.2 cette fois-ci appliquées à `X_hog`. Comparez les résultats (notamment les scores) obtenus pour différents classifieurs, différentes représentations. Que concluez-vous? Ces conclusions vont semblent-elles logiques étant donné le problème de classification auquel vous avez affaire?

2.4 Des scores aux probabilités

La librairie `sklearn` offre la possibilité de fournir, non seulement une prédiction de classe pour chaque donnée x, mais également un "niveau de confiance", ou "probabilité", de l'adéquation de chacune des classes à la donnée x: ainsi la classe prédite est celle estimée la plus **probable**. Ces probabilités, pour chaque classe, sont accessibles via la fonction: The `sklearn` library provides, not only the class prediction for every datum x, but also a "confidence level", or "probability", for each class to match x: as such, the predicted class is simply the most **probable**. These probabilities for each class are accessible with the function:

```
classifier.predict_proba(x)
```

qui généralise donc l'appel simple `classifier.predict(x)`. Testez la fonction `predict_proba`, et relevez notamment ce que vous obtenez pour les données mal classées par `predict()`. Pour ces données, tracez un graphique d'abscisses `'0'`, `'1'`, ..., `'9'` et d'ordonnées les probabilités associées à chaque classe: que remarque-t-on?

BONUS. Si vous avez traité la question BONUS de la Section 2.1, pouvez-vous imaginer une façon d'obtenir les probabilités d'appartenance d'une donnée x à chaque classe en fonction des valeurs $\omega^\mathsf{T} x + b$ associées à chaque hyperplan séparateur (pour rappel, dans un cas à plus de deux classes, SVM utilise plusieurs hyperplans pour divisier l'espace des données)?

3 Classification kNN

Nous terminons ce TP par l'utilisation d'un autre classifieur fourni par la librairie `sklearn`: l'algorithme kNN des k plus proches voisins. Avant toute chose, il est bon de rappeler que l'algorithme kNN ne passe pas bien à l'échelle, en ce sens qu'il demande, pour chaque donnée $x \in \mathbb{R}^p$ dont on souhaite estimer la classe, d'évaluer l'ensemble des distances $\|x - x_i\|$ pour tous les x_i de la base de données d'entrainement. Si à la fois la dimension p et le nombre n des données de la base d'apprentissage sont grands, le coût d'estimation est très vite prohibitif. Il est donc sage de ne **travailler qu'avec une sous-partie restreinte** de la base de données `X,y`. Effectuez cette opération en ne conservant dans un couple matrice-vecteur `X_red,y_red` une version "réduite" de `X,y` à un maximum de 1 000 éléments.

Une fois cette opération effectuée, nous devons charger le classifieur kNN de `sklearn` par l'opération

```
from sklearn.neighbors import KNeighborsClassifier
```

Le paramètre important de kNN est le nombre de voisins que l'algorithme utilise pour déterminer la classe par *vote majoritaire*. Si, par exemple, parmi les cinq plus proches voisins de `x`, trois d'entre eux sont des `'4'`, alors `'4'` sera choisi comme classe de `x`. Expliquez pourquoi il est risqué de prendre un seul voisin ou trop de voisins dans le vote. Expliquez également pourquoi on prendra en général un nombre impair, et même un nombre premier, de voisins. Le nombre de voisins est déterminé par la variable `n_neighbors` qui vaut 11 par défaut.

En utilisant la même démarche que dans la Section 2 (avec deux classes, puis avec toutes les classes; avec les données brutes puis leurs représentations HOG), évaluer les performances de kNN et comparez les aux performances de SVM. En prenant `n_neighbors=1`, qu'observez-vous en particulier concernant l'erreur sur la base d'entrainement? Expliquez en quoi ce phénomène est en fait évident. Regardez également la sortie de la fonction `predict_proba()` pour différentes valeurs de `n_neighbors`: expliquez ce que vous observez et ses conséquences pratiques.

Finalement, en jouant sur les différents paramètres à la fois pour SVM et pour kNN, en prenant à la fois en compte les performances mais aussi le temps de calcul, justifiez les avantages et les inconvénients de chaque méthode. Quelle serait celle que vous privilégieriez?

Astuce: pour évaluer le temps pris par un bout de code, vous pouvez utiliser le script suivant:

```
import time
t_begin = time.time()
# ...
# script
# ...
t_end = time.time() - t
```