
TP n° 2 : Apprentissage de métrique et gradient stochastique

Introduction

Le but de cette séance est la mise en œuvre des techniques d'apprentissage de métrique basées sur des algorithmes de type descente de gradient stochastique (SGD) vus dans le TP précédent. Deux fichiers Python sont fournis (`SGD_metric_learning_pairs.py` et `SGD_metric_learning_triplets.py`) pour guider les réponses, avec néanmoins certaines parties manquantes qu'il faut compléter.

Définitions et notations

On rappelle ici le cadre de l'apprentissage supervisé, et l'on présente les notations que l'on utilisera :

- \mathcal{Y} l'ensemble des étiquettes des données (*labels* en anglais) : $\mathcal{Y} = \{1, \dots, C\}$
- $\mathbf{x} = (x_1, \dots, x_p)^\top \in \mathcal{X} \subset \mathbb{R}^p$ est une observation, un exemple, un point (ou un *sample* en anglais). La j ème coordonnée de \mathbf{x} est la valeur prise par la j ème variable (*feature* en anglais).
- $\mathcal{D}_n = \{(\mathbf{x}_i, y_i) \in \mathcal{X} \times \mathcal{Y}, i = 1, \dots, n\}$, ensemble d'apprentissage contenant n exemples et leurs étiquettes.
- Il existe un modèle probabiliste qui gouverne la génération des observations selon des variables aléatoires X et $Y : \forall i \in \{1, \dots, n\}, (\mathbf{x}_i, y_i) \stackrel{i.i.d}{\sim} (X, Y)$ et l'on note \mathbb{E} l'espérance associée.
- On cherche à construire à partir de l'ensemble d'apprentissage \mathcal{D}_n une fonction appelée classifieur, $\hat{f} : \mathcal{X} \rightarrow \mathcal{Y}$ qui pour un nouveau point $\mathbf{x}_{\text{new}} \in \mathcal{X}$ fournit une étiquette $\hat{f}(\mathbf{x}_{\text{new}})$.
- On mesure la performance d'un classifieur, pour une perte $\ell : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$, par le risque $\mathbb{E}(\ell(\hat{f}(\mathbf{x}), y))$. En pratique, cette quantité n'est pas calculable, on se sert donc de contrepartie empirique du type $\frac{1}{n} \sum_{i=1}^n \ell(\hat{f}(\mathbf{x}_i), y_i)$.

Rappels sur l'apprentissage de métrique

Beaucoup de méthodes en apprentissage statistique s'appuient sur une mesure de distance entre les observations. C'est le cas par exemple de la classification par plus proches voisins ou encore du clustering K -Means. Le choix d'une distance (ou métrique) appropriée est crucial pour la performance de ces méthodes. L'apprentissage de métrique [Kul12, BHS13] consiste à apprendre automatiquement une mesure de distance (ou de similarité) à partir de paires d'observations que l'on sait similaires ou dissimilaires pour la tâche considérée.

On s'intéresse ici à la famille de distance dite "de Mahalanobis" qui est définie de la manière suivante :

$$\forall (\mathbf{x}, \mathbf{x}') \in \mathcal{X} \times \mathcal{X}, \quad d_{\mathbf{M}}^2(\mathbf{x}, \mathbf{x}') = (\mathbf{x} - \mathbf{x}')^T \mathbf{M} (\mathbf{x} - \mathbf{x}'), \quad (1)$$

avec $\mathbf{M} \in \mathbb{S}_+^p$, où \mathbb{S}_+^p est l'ensemble des matrices $p \times p$ symétriques semi-définies positives. Ceci permet de garantir que la fonction $d_{\mathbf{M}}$ est une pseudo-distance, c'est-à-dire qu'elle est positive, symétrique et satisfait l'inégalité triangulaire (par contre elle ne satisfait pas toujours la séparation : on peut trouver deux points $\mathbf{z} \neq \mathbf{z}'$ avec $d_{\mathbf{M}}(\mathbf{z}, \mathbf{z}') = 0$). On peut noter que si $\mathbf{M} = \mathbf{I}_p$, $d_{\mathbf{M}}$ correspond à la distance Euclidienne. L'objectif est ici d'optimiser la matrice \mathbf{M} en utilisant l'ensemble d'apprentissage afin d'obtenir une distance adaptée au problème considéré.

Approche basée sur les paires

Dans cette partie, on va considérer une formulation de l'apprentissage de métrique qui utilise des paires d'exemples similaires ou dissimilaires. On travaillera à partir du fichier `SGD_metric_learning_pairs.py`.

Introduisons tout d'abord quelques notations. On associe à toute paire d'exemples $(\mathbf{x}_i, \mathbf{x}_j)$, $1 \leq i, j \leq n$, un label y_{ij} défini par

$$y_{ij} = \begin{cases} 1 & \text{si } y_i = y_j, \\ -1 & \text{sinon.} \end{cases}$$

On va formuler l'apprentissage de métrique sous la forme du problème d'optimisation suivant :

$$\min_{\mathbf{M} \in \mathbb{S}_+^p} \frac{1}{n^2} \sum_{i,j=1}^n \ell(d_{\mathbf{M}}^2(\mathbf{x}_i, \mathbf{x}_j), y_{ij}) = \frac{1}{n^2} \sum_{i,j=1}^n \ell_{i,j}(\mathbf{M}),$$

où $\ell : \mathbb{R}_+ \times \{-1, 1\} \rightarrow \mathbb{R}$ est une fonction de perte sur les paires. L'algorithme ci-dessous est une descente de gradient stochastique appliquée à ce problème. Notons qu'une étape de projection est nécessaire pour maintenir la matrice itérée dans l'ensemble admissible \mathbb{S}_+^p . Cette projection est définie par

$$\Pi_{\mathbb{S}_+^p} : \mathbb{S}^p \rightarrow \mathbb{S}_+^p, \\ \mathbf{M} \mapsto \Pi_{\mathbb{S}_+^p}(\mathbf{M}) = \arg \min_{\mathbf{S} \in \mathbb{S}_+^p} \|\mathbf{S} - \mathbf{M}\|_2^2,$$

où \mathbb{S}^p est l'ensemble des matrices symétriques. La projection $\Pi_{\mathbb{S}_+^p}$ est décrite dans la fonction `psd_proj` disponible dans `SGD_metric_learning`.¹

Algorithme 1 : Algorithme du gradient stochastique pour l'apprentissage de métrique

Data : les observations et leurs étiquettes $\mathcal{D}_n = \{(\mathbf{x}_i, y_i) : 1 \leq i \leq n\}$

nombre maximal d'itérations : T

suite de pas de gradient : $(\gamma_t)_{t=1, \dots, T}$

Result : \mathbf{M}_T

initialiser (aléatoirement) $\mathbf{M}_0 \in \mathbb{S}_+^p$; initialiser $t = 0$

while $t \leq T$ **do**

tirer aléatoirement (i, j) dans $\{1, \dots, n\}^2$
 $\mathbf{M}_{t+1} \leftarrow \Pi_{\mathbb{S}_+^p}[\mathbf{M}_t - \gamma_t \nabla \mathbf{M} \ell_{i,j}(\mathbf{M})]$
 $t \leftarrow t + 1$

optionnellement : $\bar{\mathbf{M}}_T \leftarrow \frac{1}{T} \sum_{t=1}^T \mathbf{M}_t$

- 1) Une fonction de perte très utilisée en apprentissage de métrique est basée sur la perte hinge :

$$\ell(d_{\mathbf{M}}(\mathbf{x}_i, \mathbf{x}_j), y_{ij}) = \max(0, 1 + y_{ij}(d_{\mathbf{M}}^2(\mathbf{x}_i, \mathbf{x}_j) - 2)). \quad (2)$$

Quelle est l'interprétation de cette fonction de perte ?

- 2) Dans le fichier `SGD_metric_learning_pairs.py` (voir ci-dessous), coder cet algorithme pour la fonction de perte hinge donnée par (2) en complétant la fonction `sgd_metric_learning_pairs`. Un sous-gradient est calculé de la manière suivante :

$$\nabla_{\mathbf{M}} \ell_{i,j}(\mathbf{M}) = \begin{cases} 0 & \text{si } 1 + y_{ij}(d_{\mathbf{M}}^2(\mathbf{x}_i, \mathbf{x}_j) - 2) \leq 0, \\ y_{ij}(\mathbf{x}_i - \mathbf{x}_j)(\mathbf{x}_i - \mathbf{x}_j)^T & \text{sinon.} \end{cases}$$

Astuce : pour deux vecteurs $\mathbf{a}, \mathbf{b} \in \mathbb{R}^p$, on peut calculer $\mathbf{ab}^T \in \mathbb{R}^{p \times p}$ avec la fonction `np.outer(a, b)`.

- 3) Tester l'algorithme sur les jeux de données Iris et Digits (on peut passer de l'un à l'autre en changeant le 1 en 0 en ligne 14 du fichier) et adapter le pas et le nombre d'itérations afin de le faire converger. Notez que pour le jeu de données Digits, vous pouvez modifier le nombre de classes considérées afin de changer la difficulté du problème.

1. Une preuve de ce résultat est donnée en annexe du sujet pour les curieux.

- 4) La factorisation de Cholesky de la matrice \mathbf{M} donne une matrice \mathbf{L} telle que $\mathbf{M} = \mathbf{L}\mathbf{L}^T$. On peut ainsi interpréter la distance apprise comme une distance Euclidienne après transformation linéaire des données par la matrice \mathbf{L} :

$$d_{\mathbf{M}}^2(\mathbf{x}, \mathbf{x}') = (\mathbf{x} - \mathbf{x}')^T \mathbf{M} (\mathbf{x} - \mathbf{x}') = (\mathbf{L}\mathbf{x} - \mathbf{L}\mathbf{x}')^T (\mathbf{L}\mathbf{x} - \mathbf{L}\mathbf{x}') = \|\mathbf{L}\mathbf{x} - \mathbf{L}\mathbf{x}'\|_2^2.$$

Utiliser le code fourni pour transformer les données et comparer visuellement les deux représentations des données en dimension 2 (en utilisant une ACP).

- 5) La distance apprise peut être utilisée pour prédire si deux exemples appartiennent ou non à la même classe. C'est une approche très populaire dans le domaine de la vérification faciale par exemple. Étant donné un seuil $\tau \geq 0$, on prédit $y_i = y_j$ si $d_{\mathbf{M}}(\mathbf{x}_i, \mathbf{x}_j) \leq \tau$ et $y_i \neq y_j$ sinon. La valeur de τ a un impact sur le nombre de faux positifs et de vrais positifs. Pour un échantillon aléatoire de paires d'apprentissage, calculer les valeurs de distance et utilisez **Scikit-Learn** pour comparer la courbe ROC de la distance apprise avec celle de la distance Euclidienne. Calculer également l'aire sous la courbe ROC (AUC).

http://scikit-learn.org/stable/modules/model_evaluation.html#receiver-operating-characteristic-roc

http://scikit-learn.org/stable/modules/generated/sklearn.metrics.roc_curve.html

<http://scikit-learn.org/stable/modules/generated/sklearn.metrics.auc.html>

D'autre part, appliquer un **knn** sur les données transformées obtenues en question 4) et comparer (sur une partie test) la performance de prédiction par rapport à la distance euclidienne.

- 6) Modifiez votre code afin de pouvoir utiliser plusieurs paires à chaque itération pour calculer le gradient (mini-batch SGD). Quel est l'effet de la taille du mini-batch sur la convergence de l'algorithme? On utilisera l'approche qui consiste à tirer b paires aléatoirement (Bonus : comparer avec l'autre approche qui consiste à tirer m observations aléatoirement, puis à former les $m(m-1)/2$ paires associées).
- 7) On a vu en cours qu'utiliser une régularisation basée sur la norme trace de \mathbf{M} permet de favoriser des solutions de rang faible et ainsi la réduction de la dimension des données. Cette norme est définie par $\|\mathbf{M}\|_* = \sum_{i=1}^p \lambda_i(\mathbf{M})$, où $\lambda_i(\mathbf{M})$ est la i -ème valeur propre de \mathbf{M} . Quand \mathbf{M} est symétrique semi-définie positive, on a une expression plus simple et facilement dérivable : $\|\mathbf{M}\|_* = \text{tr}(\mathbf{M}) = \sum_{i=1}^p \mathbf{M}_{i,i}$. On veut donc résoudre la version régularisée du problème précédent :

$$\min_{\mathbf{M} \in \mathbb{S}_+^p} \quad \frac{1}{n^2} \sum_{i,j=1}^n \ell_{i,j}(\mathbf{M}) + \alpha \text{tr}(\mathbf{M}),$$

où $\alpha \geq 0$ est un paramètre de régularisation. Modifiez votre code afin de prendre en compte la régularisation. Comment varie le nombre de valeurs propres nulles en fonction de α ? Peut-on trouver un bon compromis entre performance et réduction de dimension sur les datasets Iris et Digits?

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn import decomposition
from sklearn import metrics
import itertools

plt.close('all')

#####
#           Loading and visualizing the data
#####

if 1: # use iris
    iris = datasets.load_iris()
    X = iris.data
    y = iris.target
```

```

else: # use digits
    digits = datasets.load_digits()
    X = digits.data
    y = digits.target

    # on ne garde que les 5 premieres classes par simplicité
    X = X[y < 5]
    y = y[y < 5]

# standardize data
X -= X.mean(axis=0)
X /= X.std(axis=0)
X[np.isnan(X)] = 0.

def plot_2d(X, y):
    """ Plot in 2D the dataset data, colors and symbols according to the
    class given by the vector y (if given); the separating hyperplan w can
    also be displayed if asked"""
    plt.figure()
    symlist = ['o', 's', '*', 'x', 'D', '+', 'p', 'v', 'H', '^']
    collist = ['blue', 'red', 'purple', 'orange', 'salmon', 'black', 'grey',
               'fuchsia']

    labs = np.unique(y)
    idxbyclass = [y == labs[i] for i in range(len(labs))]

    for i in range(len(labs)):
        plt.plot(X[idxbyclass[i], 0], X[idxbyclass[i], 1], '+',
                 color=collist[i % len(collist)], ls='None',
                 marker=symlist[i % len(symlist)])
    plt.ylim([np.min(X[:, 1]), np.max(X[:, 1])])
    plt.xlim([np.min(X[:, 0]), np.max(X[:, 0])])
    plt.show()

#####
#           Displaying labeled data
#####

# on utilise PCA pour projeter les donnees en 2D
pca = decomposition.PCA(n_components=2)
X_2D = pca.fit_transform(X)
plot_2d(X_2D, y)

#####
#           Stochastic gradient for metric learning
#####

def psd_proj(M):
    """ projection de la matrice M sur le cone des matrices semi-definies
    positives"""
    # calcule des valeurs et vecteurs propres
    eigenval, eigenvect = np.linalg.eigh(M)
    # on trouve les valeurs propres negatives ou tres proches de 0
    ind_pos = eigenval > 1e-10
    # on reconstruit la matrice en ignorant ces dernieres

```

```

M = np.dot(eigenvec[:, ind_pos] * eigenval[ind_pos][np.newaxis, :],
            eigenvec[:, ind_pos].T)
return M

def hinge_loss_pairs(X, pairs_idx, y_pairs, M):
    """Calcul du hinge loss sur les paires
    """
    diff = X[pairs_idx[:, 0], :] - X[pairs_idx[:, 1], :]
    return np.maximum(0., 1. + y_pairs.T * (np.sum(np.dot(M, diff.T) * diff.T,
                                                    axis=0) - 2.))

def sgd_metric_learning_pairs(X, y, gamma, alpha, n_iter, n_eval, M_ini,
                              random_state=42):
    """Stochastic gradient algorithm for metric learning with pairs

    Parameters
    -----
    X : array, shape (n_samples, n_features)
        The data
    y : array, shape (n_samples,)
        The targets.
    gamma : float | callable
        The step size. Can be a constant float or a function
        that allows to have a variable step size
    alpha : float
        The regularization parameter
    n_iter : int
        The number of iterations
    n_eval : int
        The number of pairs to evaluate the objective function
    M_ini : array, shape (n_features, n_features)
        The initial value of M
    random_state : int
        Random seed to make the algorithm deterministic
    """
    rng = np.random.RandomState(random_state)
    n_samples = X.shape[0]

    # tirer n_eval paires aleatoirement
    pairs_idx = rng.randint(0, n_samples, (n_eval, 2))
    # calcul du label des paires
    y_pairs = 2.0 * (y[pairs_idx[:, 0]] == y[pairs_idx[:, 1]]) - 1.0
    M = M_ini.copy()
    pobj = np.zeros(n_iter)

    if not callable(gamma):
        def gamma_func(t):
            return gamma
    else:
        gamma_func = gamma

    for t in range(n_iter):
        pobj[t] = np.mean(hinge_loss_pairs(X, pairs_idx, y_pairs, M))
        gradient = np.zeros((n_features, n_features))

```

```

    # TODO QUESTION 2
    # idx_for_gradient = rng.randint(0, n_samples, 2)

    M -= gamma_func(t) * gradient
    M = psd_proj(M)
    return M, pobj

n_features = X.shape[1]

M_ini = np.eye(n_features)
M, pobj = sgd_metric_learning_pairs(X, y, 0.002, 0.0, 10000, 1000, M_ini)
# M, pobj = sgd_metric_learning_pairs_minibatch(X, y, 0.002, 0.0, 10000, 1000,
#                                              5, 0, M_ini)

plt.figure()
plt.plot(pobj)
plt.xlabel('t')
plt.ylabel('cost')
plt.title('hinge stochastic with pairs')
plt.show()

# on regarde le nombre de valeurs propres egales a 0
e, v = np.linalg.eig(M)
print "Nb de valeurs propres non nulles de M: ", np.sum(e < 1e-12), "/", \
      e.shape[0]

# calcul de la factorisation de cholesky
# on ajoute de tres faibles coefficients sur la diagonale pour eviter
# les erreurs numeriques
L = np.linalg.cholesky(M + 1e-10 * np.eye(n_features))
# on projette lineairement les donnees
X_proj = np.dot(X, L)

# TODO QUESTION 4: projeter X_proj en 2D avec une PCA

# TODO QUESTION 5: tirer paires aleatoires
# calculer les distances et tracer les courbes ROC

# AIDE POUR LE BONUS DE LA QUESTION 6:

# def combs(a, r):
#     """ compute all r-length combinations of elements in array; a faster
#     than np.array(list(itertools.combinations(a, r)))
#     """
#     a = np.asarray(a)
#     dt = np.dtype([(' ', a.dtype)] * r)
#     b = np.fromiter(itertools.combinations(a, r), dt)
#     return b.view(a.dtype).reshape(-1, r)

```

Approche basée sur les triplets

Dans cette partie, on va expérimenter avec une formulation d'apprentissage de métrique qui utilise cette fois des triplets d'exemples de la forme $(\mathbf{x}_i, \mathbf{x}_j, \mathbf{x}_k)$ avec $y_i = y_j \neq y_k$. On travaillera à partir du fichier `SGD_metric_learning_triplets.py`.

On note \mathcal{R} l'ensemble de ces triplets et on considère le problème d'optimisation suivant :

$$\min_{\mathbf{M} \in \mathbb{S}_+^p} \sum_{(\mathbf{x}_i, \mathbf{x}_j, \mathbf{x}_k) \in \mathcal{R}} \ell(\mathbf{M}, \mathbf{x}_i, \mathbf{x}_j, \mathbf{x}_k),$$

où $\ell : \mathbb{S}_+^p \times \mathbb{R}^p \times \mathbb{R}^p \times \mathbb{R}^p \rightarrow \mathbb{R}$ est une fonction de perte sur les triplets.

- 8) Une fonction de perte très utilisée pour les triplets est encore une fois basée sur la perte hinge :

$$\ell(\mathbf{M}, \mathbf{x}_i, \mathbf{x}_j, \mathbf{x}_k) = \max(0, 1 - d_{\mathbf{M}}^2(\mathbf{x}_i, \mathbf{x}_k) + d_{\mathbf{M}}^2(\mathbf{x}_i, \mathbf{x}_j)).$$

Quelle est l'interprétation de cette perte ?

- 9) Après avoir trouvé comment calculer un sous-gradient de cette perte, implémentez une version triplet de l'algorithme de descente de gradient stochastique. Vous pourrez vous aider de la fonction `generate_triplets` qui permet de générer un ensemble de triplets. Comparez la performance des deux approches.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn import decomposition
from sklearn import metrics

plt.close('all')

#####
#           Loading and visualizing the data
#####

if 1: # use iris
    iris = datasets.load_iris()
    X = iris.data
    y = iris.target
else: # use digits
    digits = datasets.load_digits()
    X = digits.data
    y = digits.target

    # on ne garde que les 5 premières classes par simplicité
    X = X[y < 5]
    y = y[y < 5]

# standardize data
X -= X.mean(axis=0)
X /= X.std(axis=0)
X[np.isnan(X)] = 0.

def plot_2d(X, y):
    """ Plot in 2D the dataset data, colors and symbols according to the
        class given by the vector y (if given); the separating hyperplan w can
        also be displayed if asked"""
    plt.figure()
```

```

symlist = ['o', 's', '*', 'x', 'D', '+', 'p', 'v', 'H', '^']
collist = ['blue', 'red', 'purple', 'orange', 'salmon', 'black', 'grey',
          'fuchsia']

labs = np.unique(y)
idxbyclass = [y == labs[i] for i in range(len(labs))]

for i in range(len(labs)):
    plt.plot(X[idxbyclass[i], 0], X[idxbyclass[i], 1], '+',
             color=collist[i % len(collist)], ls='None',
             marker=symlist[i % len(symlist)])
plt.ylim([np.min(X[:, 1]), np.max(X[:, 1])])
plt.xlim([np.min(X[:, 0]), np.max(X[:, 0])])
plt.show()

#####
#           Displaying labeled data
#####

# on utilise PCA pour projeter les donnees en 2D
pca = decomposition.PCA(n_components=2)
X_2D = pca.fit_transform(X)
plot_2d(X_2D, y)

#####
#           Stochastic gradient for metric learning
#####

def psd_proj(M):
    """ projection de la matrice M sur le cone des matrices semi-definies
    positives"""
    # calcule des valeurs et vecteurs propres
    eigenval, eigenvect = np.linalg.eigh(M)
    # on trouve les valeurs propres negatives ou tres proches de 0
    ind_pos = eigenval > 1e-10
    # on reconstruit la matrice en ignorant ces dernieres
    M = np.dot(eigenvect[:, ind_pos] * eigenval[ind_pos][np.newaxis, :],
               eigenvect[:, ind_pos].T)
    return M

def hinge_loss_triplets(X, triplets_idx, M):
    """Calcul du hinge loss sur les triplets
    """
    diffp = X[triplets_idx[:, 0], :] - X[triplets_idx[:, 1], :]
    diffn = X[triplets_idx[:, 0], :] - X[triplets_idx[:, 2], :]
    return np.maximum(0., 1. - np.sum(np.dot(M, diffn.T) * diffn.T, axis=0) +
                     np.sum(np.dot(M, diffp.T) * diffp.T, axis=0))

def generate_triplets(X, y, n_triplets, random_state=42):
    """Generation de triplets
    """
    rng = np.random.RandomState(random_state)
    n_samples = X.shape[0]
    triplets_idx = np.zeros((n_triplets, 3), dtype=int)

```



```

triplets_idx[:, 0] = rng.randint(0, n_samples, (n_triplets,))
for i in range(n_triplets):
    same_idx = np.where(y == y[triplets_idx[i, 0]])[0]
    diff_idx = np.where(y != y[triplets_idx[i, 0]])[0]
    triplets_idx[i, 1] = same_idx[rng.randint(0, same_idx.shape[0])]
    triplets_idx[i, 2] = diff_idx[rng.randint(0, diff_idx.shape[0])]
return triplets_idx

def sgdc_metric_learning_triplets(X, y, gamma, alpha, n_iter, n_eval, M_ini,
                                  random_state=42):
    """Stochastic gradient algorithm for metric learning with triplets

    Parameters
    -----
    X : array, shape (n_samples, n_features)
        The data
    y : array, shape (n_samples,)
        The targets.
    gamma : float | callable
        The step size. Can be a constant float or a function
        that allows to have a variable step size
    alpha : float
        The regularization parameter
    n_iter : int
        The number of iterations
    n_eval : int
        The number of triplets to evaluate the objective function
    M_ini : array, shape (n_features, n_features)
        The initial value of M
    random_state : int
        Random seed to make the algorithm deterministic
    """
    rng = np.random.RandomState(random_state)
    # n_samples = X.shape[0]
    # generate n_eval triplets
    triplets_idx = generate_triplets(X, y, n_eval, random_state=random_state)
    M = M_ini.copy()
    pobj = np.zeros(n_iter)

    if not callable(gamma):
        def gamma_func(t):
            return gamma
    else:
        gamma_func = gamma

    for t in range(n_iter):
        pobj[t] = np.mean(hinge_loss_triplets(X, triplets_idx, M))
        gradient = np.zeros((n_features, n_features))

        # TODO GRADIENT

        M -= gamma_func(t) * gradient
        M = psd_proj(M)
    return M, pobj

```

```

n_features = X.shape[1]

M_ini = np.eye(n_features)
M, pobj = sgd_metric_learning_triplets(X, y, 0.002, 0.0, 10000, 1000, M_ini)

plt.figure()
plt.plot(pobj)
plt.xlabel('t')
plt.ylabel('cost')
plt.title('hinge stochastic with triplets')
plt.show()

# on regarde le nombre de valeurs propres egales a 0
e, v = np.linalg.eig(M)
print "Nb of nonzero eigenvalues of M: ", np.sum(e < 1e-12), "/", e.shape[0]

# calcul de la factorisation de cholesky
# on ajoute de tres faibles coefficients sur la diagonale pour eviter
# les erreurs numeriques
L = np.linalg.cholesky(M + 1e-10 * np.eye(n_features))
# on projette lineairement les donnees
X_proj = np.dot(X, L)

# on utilise PCA pour projeter les donnees en 2D
X_proj_2D = pca.fit_transform(X_proj)

plot_2d(X_proj_2D, y)

# TODO PCA AND ROC

```

Question bonus : OASIS

L'algorithme OASIS [CSSB09] est un algorithme en ligne pour apprendre une similarité bilinéaire $S_{\mathbf{M}}(\mathbf{x}, \mathbf{x}') = \mathbf{x}^T \mathbf{M} \mathbf{x}'$, où $\mathbf{M} \in \mathbb{R}^{p \times p}$ n'est pas nécessairement semi-définie positive (ni même forcément symétrique!). A chaque itération t , OASIS reçoit un triplet $(\mathbf{x}_i, \mathbf{x}_j, \mathbf{x}_k)$ et résout le sous-problème suivant :

$$\begin{aligned}
\mathbf{M}^t = \arg \min_{\mathbf{M}, \xi} \quad & \frac{1}{2} \|\mathbf{M} - \mathbf{M}^{t-1}\|_{\mathcal{F}}^2 + C\xi \\
\text{s.t.} \quad & 1 - S_{\mathbf{M}}(\mathbf{x}_i, \mathbf{x}_j) + S_{\mathbf{M}}(\mathbf{x}_i, \mathbf{x}_k) \leq \xi \\
& \xi \geq 0
\end{aligned}$$

où $C \geq 0$ est un paramètre dit "d'agressivité".

La solution de ce sous problème s'écrit $\mathbf{M}^t = \mathbf{M}^{t-1} + \beta \mathbf{V}$, où $\mathbf{V} = \mathbf{x}_i(\mathbf{x}_j - \mathbf{x}_k)^T$ et

$$\beta = \min \left(C, \frac{\max(0, 1 - S_{\mathbf{M}}(\mathbf{x}_i, \mathbf{x}_j) + S_{\mathbf{M}}(\mathbf{x}_i, \mathbf{x}_k))}{\|\mathbf{V}\|_{\mathcal{F}}^2} \right).$$

Implémentez OASIS et testez sa performance sur les datasets utilisés précédemment.

Références

- [Ber99] D. P. Bertsekas. *Nonlinear programming*. Athena Scientific, 1999. 12
- [BHS13] Aurélien Bellet, Amaury Habrard, and Marc Sebban. A Survey on Metric Learning for Feature Vectors and Structured Data. Technical report, arXiv :1306.6709, June 2013. 1

- [CSSB09] Gal Chechik, Uri Shalit, Varun Sharma, and Samy Bengio. An Online Algorithm for Large Scale Image Similarity Learning. In *Advances in Neural Information Processing Systems 22*, pages 306–314, 2009. [10](#)
- [Kul12] Brian Kulis. Metric Learning : A Survey. *Foundations and Trends in Machine Learning*, 5(4) :287–364, 2012. [1](#)

Projection sur les matrices symétriques semi-définie positives

Soit $\mathbf{M} \in \mathbb{S}_+^p$. Pour tout $t \in \mathbb{R}$, on note $t_+ = \max(0, t)$ (resp. $t_- = \min(0, t)$), ainsi on a la décomposition en partie positive et négative suivante :

$$t = t_+ + t_-. \quad (3)$$

Définissons sa décomposition spectrale par $\mathbf{M} = \mathbf{U}^\top \mathbf{D} \mathbf{U}$ avec $\mathbf{U} \in \mathbb{R}^{p \times p}$ vérifiant $\mathbf{U}^\top \mathbf{U} = \mathbf{I}_p$ et $\mathbf{D} \in \mathbb{R}^{p \times p}$ est diagonale. On note $\mathbf{D} = \text{diag}(d_1, \dots, d_p) = \mathbf{D}_+ + \mathbf{D}_-$, avec $\mathbf{D}_+ = \text{diag}((d_1)_+, \dots, (d_p)_+) \in \mathbb{R}^{p \times p}$ (resp. $\mathbf{D}_- = \text{diag}((d_1)_-, \dots, (d_p)_-) \in \mathbb{R}^{p \times p}$) la matrice diagonale contenant les termes positifs (resp. négatifs) de \mathbf{D} . On notera au passage que $\mathbf{D}_- \mathbf{D}_+ = 0$.

On rappelle maintenant que $\Pi_{\mathbb{S}_+^p} : \mathbb{S}^p \rightarrow \mathbb{S}_+^p$, $\Pi_{\mathbb{S}_+^p}(\mathbf{M}) = \arg \min_{\mathbf{S} \in \mathbb{S}_+^p} \|\mathbf{S} - \mathbf{M}\|_2^2$, où \mathbb{S}^p est l'ensemble des matrices symétriques. On va montrer que $\Pi_{\mathbb{S}_+^p}(\mathbf{M}) = \mathbf{U}^\top \mathbf{D}_+ \mathbf{U}$.

Pour cela on rappelle que pour le produit scalaire associé à la norme $\|\cdot\|_2$, et définit sur \mathbb{S}_+^p par $\langle \mathbf{M}, \mathbf{N} \rangle = \text{tr}(\mathbf{M}^\top \mathbf{N})$, $\Pi_{\mathbb{S}_+^p}(\mathbf{M})$ (projection sur le convexe fermé \mathbb{S}_+^p) est la seule matrice vérifiant :

$$\forall \mathbf{N} \in \mathbb{S}_+^p, \quad \langle \mathbf{M} - \Pi_{\mathbb{S}_+^p}(\mathbf{M}), \mathbf{N} - \Pi_{\mathbb{S}_+^p}(\mathbf{M}) \rangle \leq 0. \quad (4)$$

Nous allons donc vérifier que $\mathbf{M}_+ = \mathbf{U}^\top \mathbf{D}_+ \mathbf{U}$ vérifie cette relation (cf. Figure 1 et [Ber99, Prop. 2.1.3] pour une preuve de cette propriété).

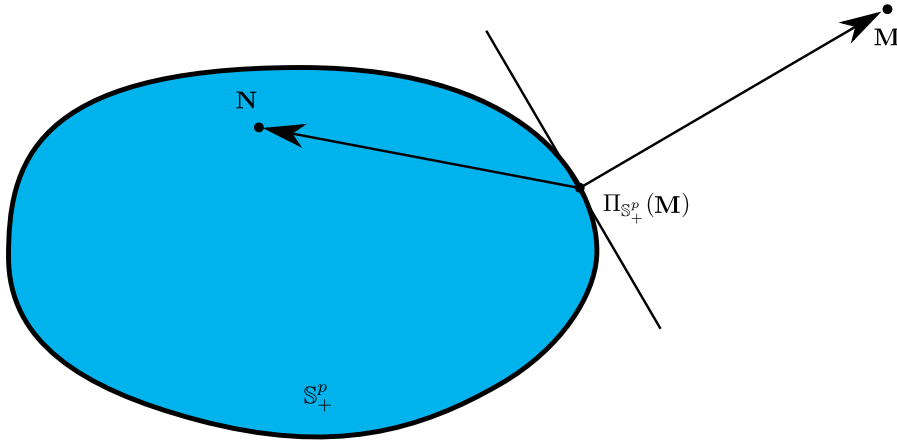


FIGURE 1 – Rappel visuel sur le théorème de projection sur les convexes fermés.

Prenons $\mathbf{N} \in \mathbb{S}_+^p$ quelconque

$$\begin{aligned} \langle \mathbf{M} - \mathbf{M}_+, \mathbf{N} - \mathbf{M}_+ \rangle &= \langle \mathbf{M} - \mathbf{U}^\top \mathbf{D}_+ \mathbf{U}, \mathbf{N} - \mathbf{U}^\top \mathbf{D}_+ \mathbf{U} \rangle, \\ &= \langle \mathbf{U}^\top \mathbf{D}_- \mathbf{U}, \mathbf{N} - \mathbf{U}^\top \mathbf{D}_+ \mathbf{U} \rangle, \\ &= \langle \mathbf{U}^\top \mathbf{D}_- \mathbf{U}, \mathbf{N} \rangle - \langle \mathbf{U}^\top \mathbf{D}_- \mathbf{U}, \mathbf{U}^\top \mathbf{D}_+ \mathbf{U} \rangle, \\ &= \langle \mathbf{U}^\top \mathbf{D}_- \mathbf{U}, \mathbf{N} \rangle - \langle \mathbf{U}, \mathbf{D}_- \mathbf{D}_+ \mathbf{U} \rangle, \\ &= \langle \mathbf{U}^\top \mathbf{D}_- \mathbf{U}, \mathbf{N} \rangle, \\ &= \text{tr}(\mathbf{U}^\top \mathbf{D}_- \mathbf{U} \mathbf{N}), \\ &= \text{tr}(\mathbf{D}_- \mathbf{U} \mathbf{N} \mathbf{U}^\top), \end{aligned}$$

Notons que $\mathbf{U} \mathbf{N} \mathbf{U}^\top$ est symétrique semi-définie positive (car \mathbf{N} l'est), donc sa décomposition spectrale donne : $\mathbf{U} \mathbf{N} \mathbf{U}^\top = \sum_{j=1}^p \lambda_j v_j v_j^\top$ avec $\lambda_j \geq 0$ pour $j = 1, \dots, p$. Ainsi, en notant $e_k = (0, \dots, 0, \underbrace{1}_{k^e}, 0, \dots, 0)^\top$ et en utilisant la linéarité de la trace :

$$\begin{aligned}
\langle \mathbf{M} - \mathbf{M}_+, \mathbf{N} - \mathbf{M}_+ \rangle &= \text{tr} \left(\mathbf{D}_- \sum_{j=1}^p \lambda_j v_j v_j^\top \right), \\
&= \text{tr} \left(\sum_{k=1}^p (d_k)_- e_k e_k^\top \sum_{j=1}^p \lambda_j v_j v_j^\top \right), \\
&= \text{tr} \left(\sum_{k=1}^p \sum_{j=1}^p \lambda_j (d_k)_- e_k^\top v_j v_j^\top e_k \right), \\
&= \sum_{k=1}^p \sum_{j=1}^p \underbrace{\lambda_j (d_k)_-}_{\leq 0} \underbrace{\text{tr} \left((v_j^\top e_k)^\top v_j^\top e_k \right)}_{\geq 0}, \\
&\leq 0.
\end{aligned}$$

On a donc montré que $\mathbf{M}_+ = \mathbf{U}^\top \mathbf{D}_+ \mathbf{U} = \Pi_{\mathbb{S}_+^p}(\mathbf{M})$.