

## TME 5 - Perceptron et SVMs

Vous devez avoir fini le TME 2 avant d'aborder celui-là. Vous aurez besoin des fonctions de coût et de leur gradient que vous avez implémentés lors de la séance précédente.

### Perceptron et classe Linéaire

Implémentez la fonction `perceptron_loss(w, x, y)` qui rend le coût perceptron ( $\max(0, -y < \mathbf{x} \cdot \mathbf{w} >)$ ) et son gradient `perceptron_grad(w, x, y)`.

Afin d'être plus générique et plus flexible, plutôt que d'implémenter une version script de la descente de gradient, nous allons coder une classe `Lineaire` qui va regrouper les différentes variantes possibles. Un squelette de classe est fourni dans le fichier source : il contient principalement le constructeur qui initialise les paramètres du modèle (le coût utilisé `loss`, son gradient `loss_g`, le nombre d'itération de la descente du gradient `max_iter`, le pas `eps`, le vecteur de poids `w`). Complétez le squelette en implémentant :

- la méthode `predict(self, datax)` qui permet d'inférer le label des données `datax` (i.e. qui calcule  $\text{sign}(< \mathbf{w} \cdot \mathbf{x} >)$ )
- la méthode `score(self, datax, datay)` qui permet de calculer le pourcentage de bonne classification sur le jeu de données passé en paramètre.
- la méthode `fit(self, datax, datay)` permet de faire la descente de gradient pendant `max_iter` itérations avec un pas `eps` en utilisant le coût `loss` et le gradient `loss_g`. Ainsi pour faire un perceptron, il suffira de passer lors de la construction de l'objet le coût et le gradient du coût perceptron. Votre méthode devra également conserver en mémoire l'historique des coûts en fonction des itérations.

Comme dans le précédent TME, vos méthodes doivent prendre en entrée des matrices de données (et non pas une seule donnée). Testez votre classe sur l'exemple de la semaine dernière.

### Données USPS

Les données USPS sont des données de chiffres manuscrits représentés par une matrice de pixel en niveau de gris de taille  $16 \times 16$ . Le code pour les charger vous est fourni ainsi qu'une fonction de visualisation (`load_usps` et `show_usps`).

Chargez les données et visualisez quelques exemples. Isolez 2 classes (par exemple les 6 vs 9) et entraînez un perceptron dessus. Visualisez la matrice de poids obtenue à l'aide de la fonction `show_usps`. Que remarquez vous ? Comment interpréter le résultat ?

Observez la matrice de poids obtenue lorsque vous entraînez le perceptron avec une classe (6 par exemple) contre toutes les autres classes.

En utilisant les données de test, tracer les courbes d'erreurs en apprentissage et en test en fonction du nombre d'itérations (vous pouvez modifier la fonction `fit` de la classe `Linéaire` afin qu'elle prenne également des données test en entrée). Observez-vous du sur-apprentissage ?

### Mini-batch et descente stochastique

Modifiez votre méthode `fit` afin de pouvoir faire une descente stochastique (les exemples sont ordonnés aléatoirement et un seul exemple est pris en compte lors du calcul du gradient et de la mise-à-jour des poids) et mini-batch (l'ensemble des exemples est divisé en petits batchs aléatoire de  $m$  exemples, le gradient est moyenné sur chaque batch avant de faire la mise-à-jour). Afin de comparer équitablement

les différentes variantes, le paramètre `max_iter` dénotera le nombre d'époques et non le nombre de mise-à-jour du gradient (une époque étant le fait d'avoir vu une fois tous les exemples). Vous pouvez remarquer que la descente stochastique correspond à une descente mini-batch de taille 1, et une descente batch à une descente mini-batch de taille le nombre d'exemples.

Comparez la vitesse de convergence en particulier en fonction du bruit dans le jeu de données.

## Projections et pénalisation

Afin d'augmenter l'expressivité du modèle linéaire, nous allons utiliser des projections. Codez la fonction `proj_poly(datax)` qui renvoie la projection polynomiale de degré 2 des données :

$(1, x_1, x_2, \dots, x_d, x_1^2, x_1x_2, \dots, x_d^2)$ .

Codez également la fonction `proj_biais(datax)` qui permet d'ajouter une colonne de 1 en première colonne des données afin d'introduire un biais.

Modifiez votre constructeur pour qu'il prenne potentiellement une projection en paramètre, votre fonction `fit` afin qu'elle projette si besoin les données avant de faire l'apprentissage, ainsi que votre fonction `predict`.

Testez votre projection polynomiale sur les données artificielles de la fonction `gen_arti` de type 1 et 2. Tracez les frontières de décision.

Codez la fonction `proj_gauss(datax, base, sigma)` qui réalise une projection gaussienne de `datax` sur les points  $(\mathbf{b}_1, \dots, \mathbf{b}_b)$  de `bases` avec un paramètre `sigma` : pour une donnée  $\mathbf{x}$  de `datax`, sa représentation est  $(e^{-\|\mathbf{x}-\mathbf{b}_1\|^2/2\sigma}, e^{-\|\mathbf{x}-\mathbf{b}_2\|^2/2\sigma}, \dots, e^{-\|\mathbf{x}-\mathbf{b}_b\|^2/2\sigma})$  et expérimentez sur les 3 jeu de données artificiels. Vaut-il mieux beaucoup de points ou peu de points pour la base de projection ? Un grand ou petit  $\sigma$  ? Quels sont les points qui ont le plus de poids ? Représentez les sur les figures et tracez les frontières.

Vous pouvez également étudié l'effet d'une marge et d'une pénalisation sur les poids de la projection gaussienne : introduisez une nouvelle fonction de coût `hinge_loss(w, x, y, alpha, lambda)` qui retourne  $\max(0, \alpha - y \langle \mathbf{w}, \mathbf{x} \rangle) + \lambda \|\mathbf{w}\|^2$  et son gradient `hinge_loss_grad`. Observez l'effet de `alpha` et `lambda` en particulier sur le problème de l'échiquier.

Quel(s) lien(s) avec le SVM ?

## SVM et Grid Search

Le module `scikit-learn` (`sklearn` : <http://scikit-learn.org/stable/documentation.html>) est le principal module d'apprentissage statistique de python. Le sous-module `sklearn.linear_model` propose en particulier une implémentation du perceptron, `sklearn.neighbors` une implémentation des  $k$ -nn, `sklearn.tree` des arbres de décision. Explorer rapidement ses implémentations, comparer par exemple pour le perceptron les résultats de vos tmes précédents avec les implémentations de `sklearn`.

*Vérifier bien d'avoir la dernière version installée ! sinon exécuter `pip install -U sklearn --user`*  
Le module `sklearn.svm` propose une implémentation des SVMs. Sur les jeux de données des précédents TMEs (2d artificiels et reconnaissance de chiffres), explorer plusieurs noyaux (linéaire, gaussien, polynomial) et plusieurs paramétrages des noyaux. Vous étudier en particulier les frontières de décisions et les vecteurs supports - les points dont les coefficients sont non nuls.

Comment évolue le nombre de ces derniers en fonction du noyau et de son paramétrage ? Est-ce normal ? Que retrouvez vous dans le cas linéaire ?

Afin de trouver les meilleurs paramètres, on opère par validation croisée sur une grille des paramètres (grid search). Pour les différents noyaux et différents nombre d'exemples d'apprentissage, opérer un grid search afin de trouver les paramètres optimaux. Tracer les courbes d'erreurs en apprentissage et en test. Les résultats sont-ils cohérents ?

Afin de visualiser les frontières de décisions en 2D, vous pouvez utiliser ce bout de code :

```
svm = sklearn.svm.SVC(probability = True, ...)
```

```

...
def plot_frontiere_proba(data, f, step=20):
    grid, x, y = make_grid(data=data, step=step)
    plt.contourf(x, y, f(grid).reshape(x.shape), 255)

plot_frontiere_proba(data, lambda x: svm.predict_proba(x)[: , 0], step=50)

```

## String Kernel

Le string kernel est un noyau défini sur les mots  $\Sigma^*$  d'un alphabet  $\Sigma$ . Pour un mot  $s$ , on dénote  $|s|$  la longueur du mot :  $s = s_1, s_2, \dots, s_{|s|}$ ,  $s[i : j]$  le sous-mot  $s_i, \dots, s_j$  de  $s$ . On dit que  $u$  est une sous-séquence de  $s$  s'il existe une suite d'indices  $\mathbf{i} = (i_1, \dots, i_{|u|})$  avec  $1 \leq i_1 < i_2 < \dots < i_{|u|} \leq |s|$  tels que  $u_j = s_{i_j}$  pour  $j = 1, \dots, |u|$ , ce que l'on notera  $u = s[\mathbf{i}]$ . La longueur  $l(\mathbf{i})$  de la sous-séquence dans  $s$  est  $i_{|u|} - i_1 + 1$ .

La projection utilisée est l'ensemble des coordonnées  $\{\phi_u(s) = \sum_{\mathbf{i}:u=s[\mathbf{i}]} \lambda^{l(\mathbf{i})}\}$   $u \in \Sigma^*$  avec  $\lambda \leq 1$ .

Ainsi on peut définir le noyau  $K_n(s, t) = \sum_{u \in \Sigma^n} \langle \phi_u(s), \phi_u(t) \rangle = \sum_{u \in \Sigma^n} \sum_{\mathbf{i}:u=s[\mathbf{i}]} \sum_{\mathbf{j}:u=t[\mathbf{j}]} \lambda^{l(\mathbf{i})+l(\mathbf{j})}$ . Programmer un string kernel, visualiser la matrice de similarité sur des exemples de textes de différents auteurs, puis tester l'apprentissage.