A2DI - TP n°6

Dans ce 6$^{\text{ième}}$ TP, nous allons tester l'algorithme de la régression logistique pour un problème à 2 classes ($\sharp\mathcal{C} = \ell = 2$) sur un jeu de données synthétiques puis un autre composé de données réelles pour 4 classes ($\sharp\mathcal{C} = \ell = 4$).

Le jeu de données réelles est `20newsgroup` (même jeu qu'au TP n°5). Nous pourrons donc comparer une approche discriminative avec le modèle génératif du classifieur naïf Bayésien.

Exercice n°1 : Régression logistique & convergence

Ce 1$^{\text{er}}$ exercice est l'occasion de s'exercer à la **descente de gradient** et à ses variantes et d'observer à quel point le choix des hyper-paramètres est important pour bien converger.

Questions :

1. Utilisez la fonction `datagen` écrite de lors l'exercice n°1 du TP n°3. Cette fonction sert à générer un jeu de données synthétique avec chevauchement entre classes.

   Comme dans le TP n°3, on se contente de diviser les données en ensemble de train et de test via cette fonction (sans validation croisée). L'ensemble d'apprentissage comprend 80% des données.

   On rappelle que les exemples sont tirés uniformément dans le carré unitaire, c'est à dire que $\mathbf{x}$ est de taille 2 et $0 \leq x_1, x_2 \leq 1$. Pour permettre le chevauchement, au moment de déterminer la classe d'un exemple, vous devrez :

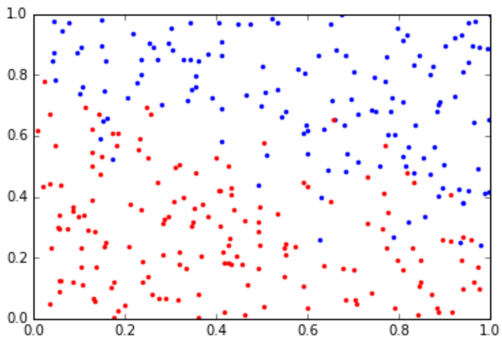
   — déterminer la distance d entre ce point généré et la droite d'équation $x_2 = -0.5x_1 + 0.75$,

   — calculer $r = e^{-\frac{d^2}{2\sigma^2}}$ avec $\sigma = 0.05$,

   — échantillonner une perturbation $Z \sim \text{Ber}\left(\frac{r}{2}\right)$,

   — Si l'échantillon $z = 1$ alors, permuter la valeur de la classe.

   Attention, pour la régression logistique, les classes doivent être 0 et 1 (et non -1 et 1). L'appel à cette fonction avec $n = 300$ doit aboutir à un dataset de la forme suivante :

2. Créez une variable `X_plus` en concaténant `X` avec une ligne de 1 de sorte à obtenir :

$$\mathbf{X}_+ = \begin{pmatrix} \begin{bmatrix} \mathbf{x}^{(1)} \end{bmatrix} & \cdots & \begin{bmatrix} \mathbf{x}^{(n)} \end{bmatrix} \\ 1 & \cdots & 1 \end{pmatrix}. \tag{1}$$

3. Nous allons commencer par utiliser le modèle de la régression logistique avec une descente de gradient classique. On rappelle que la régression logistique est un modèle linéaire qui cherche à trouver une droite séparatrice pour nos données. Cette droite est paramétrée par un vecteur normal $\mathbf{w}$ et une constante b, appelée *intercept*. Pour alléger l'écriture, on pose

$$\boldsymbol{\theta} = \begin{bmatrix} w_1 \\ \vdots \\ w_d \\ b \end{bmatrix}. \tag{2}$$

La descente de gradient classique est, par opposition à la stochastique, qualifiée de *batch* (fournée). En cours, nous avons justifié que la procédure suivante permet de converger vers $\boldsymbol{\theta}^*$ qui minimise la NLL du modèle de la régression logistique.

Initialiser $\boldsymbol{\theta}_0$ et η.
while pas convergé **do**
Calculer les prédictions : $\mathbf{pred} \leftarrow \mathrm{sgm}\left(\mathbf{X}_+^T.\boldsymbol{\theta}_t\right)$.
Calculer les erreurs : $\mathbf{err} \leftarrow \mathbf{pred} - \mathbf{c}$.
Calculer le gradient : $\mathbf{g} \leftarrow \mathbf{X}_+.\mathbf{err}$
Mettre à jour les paramètres : $\boldsymbol{\theta}_{t+1} \longleftarrow \boldsymbol{\theta}_t - \eta \times \mathbf{g}$.
end while

Il s'agit de la version la plus basique avec un *learning rate* η fixe.

Implémentez cette procédure en python de sorte à ce qu'elle retourne tout l'historique des valeurs prises par le vecteur $\boldsymbol{\theta}$. Cet historique sera conservé dans un `2D numpy array` appelé `thetas` et de taille 3×526.

4. Testez votre implémentation avec $\eta = 0.02$ puis $\eta = 0.1$. Pour visualisez les résultats en tapant :

```
from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure()
ax = fig.gca(projection='3d')
ax.plot(thetas[0],thetas[1],thetas[2],'-o')
plt.draw()
```

5. Passons à la méthode de Newton où η_t sera mis à jour à l'aide de la matrice Hessienne. La procédure devient :

Initialiser $\boldsymbol{\theta}_0$ et η.
while pas convergé **do**
Calculer les prédictions : $\mathbf{pred} \leftarrow \mathrm{sgm}\left(\mathbf{X}_+^T.\boldsymbol{\theta}_t\right)$.
Calculer les erreurs : $\mathbf{err} \leftarrow \mathbf{pred} - \mathbf{c}$.
Calculer le vecteur $\mathbf{s} \leftarrow \mathbf{pred} \odot (1 - \mathbf{pred})$ ($\odot$ symbolise le produit terme à terme).
Créer la matrice $\mathbf{S} \leftarrow \mathrm{diag}\,(\mathbf{s})$.
Calculer la matrice Hessienne selon $\mathbf{H} \leftarrow \mathbf{X}_+\mathbf{S}\mathbf{X}_+^T$.
Calculer le gradient : $\mathbf{g} \leftarrow \mathbf{X}_+.\mathbf{err}$
Mettre à jour les paramètres : $\boldsymbol{\theta}_{t+1} \longleftarrow \boldsymbol{\theta}_t - \eta \times \mathbf{H}^{-1}.\mathbf{g}$.
end while

Implémentez cette procédure en python de sorte à conserver l'historique des valeurs de $\boldsymbol{\theta}$ de la même manière que précédemment.
Indications : Pour faire le produit terme à terme, vous pouvez utiliser `numpy.multiply`. Pour obtenir $\mathbf{S}$, la fonction `numpy.diag` est prévue pour ce genre de situations. Néanmoins, elle fonctionne si `s` est enregistré sous forme de `1D numpy array`. Si ce n'est pas le cas, vous pouvez utiliser la méthode `flatten` de cette classe. Enfin l'inversion de matrice est disponible via `numpy.linalg.inv`.

6. Testez votre implémentation avec $\eta = 0.1$ puis visualisez la trajectoire de $\boldsymbol{\theta}$ avec le même affiachage 3D que précédemment.

7. Passons enfin à la descente stochastique. La procédure devient :

> Initialiser $\boldsymbol{\theta}_0$ et η.
> **while** pas convergé **do**
>   Tirer une permutation σ au hasard.
>   Appliquer la permutation aux exemples : $\mathbf{X}_+ \leftarrow \sigma(\mathbf{X}_+)$
>   Appliquer la même permutation aux exemples : $c \leftarrow \sigma(c)$
>   **for** pour i de 1 à n **do**
>   Caluler la prédiction pour le $i^{\text{ème}}$ exemple : pred $\leftarrow \text{sgm}\left({\mathbf{x}_+^{(i)}}^T . \boldsymbol{\theta}_t\right)$.
>   Calculer son erreur : $err \leftarrow \text{pred} - c^{(i)}$.
>   Calculer son gradient : $\mathbf{g} \leftarrow \mathbf{x}_+^{(i)} \times err$
>   Mettre à jour les paramètres : $\boldsymbol{\theta}_{t+1} \longleftarrow \boldsymbol{\theta}_t - \eta \times \mathbf{g}$.
>   **end for**
> **end while**

Implémentez cette procédure en python de sorte à conserver l'historique des valeurs de $\boldsymbol{\theta}$ mais aussi le nombre d'« époques ».
Indications : Pour la permutation, vous aurez besoin de `numpy.random.permutation`.

8. Testez votre implémentation avec $\eta = 1.5$ puis visualisez la trajectoire de $\boldsymbol{\theta}$ avec le même affiachage 3D que précédemment.
9. Modifiez la procédure en calculant le *learning rate* selon :
$$\eta_t = \frac{20}{(t+100)^{0.6}}. \tag{3}$$
L'itération t s'obtient selon
$$t = \text{nombre d'époques} \times n + i. \tag{4}$$
Visualisez la trajectoire du vecteur $\boldsymbol{\theta}$.
10. Comparer les 3 algorithmes d'optimisation en superposant au dataset les 3 frontières séparatrices obtenues. Comparer aussi le nombre d'« époques » au nombre d'itérations nécessaires à Newton ou à la méthode basique.

Exercice n°2 : Discriminatif vs Génératif

Dans cet exercice, nous reprenons le même dataset qu'au TP N°5. Nous avions alors tester un modèle génératif appelé classifieur naïf Bayésien. Aujourd'hui, nous comparons avec la régression logistique qui est un modèle discriminatif. Nous allons en réalité utiliser une régression *softmax* car ce problème est à 4 classes.

On rappelle qu'un modèle est **génératif** s'il permet d'estimer la jointe $p_{X,Y}$ tandis qu'un modèle **discriminatif** se contente d'estimer $p_{Y|X}$.

1. Relancez vos programmes correspondant aux questions 1 à 3 du TP n°5. Vous avez alors chargé les données, puis répartis ces dernières en 5 plis pour la validation croisée.
2. Nous allons utiliser l'implémentation de la régression logistique fournie par le module `scikit-learn`. Tapez :
```
from sklearn import linear_model
logreg = linear_model.LogisticRegression(C=1e5,multi_class='multinomial',
                                         solver='newton-cg')
```
Les paramètres chosis précisent que nous souhaitons faire du multi-classe avec la méthode de Newton et avec une régularisation quasi-nulle (`C` très grand).

Lancez ensuite apprentissage et test sur chaque pli avec les méthodes `logreg.fit` et `logreg.predict`.
Indication : il est nécessaire de transposer la matrice des exemples d'apprentissage pour utiliser ces fonctions.
3. Calculez le taux de reconnaissance moyen et comparez au NBC.