TP 3 - Descente de gradient, classifieurs linéaires

Dans ce TP, vous devrez coder vos premiers classifieurs. Nous suivrons les conventions suivantes (les mêmes que celles du module `scikit-learn`) : un classifieur est muni d'une fonction `fit(data,labels)` qui permet d'apprendre les paramètres du classifieur sur la matrice d'exemples `data` et le vecteur d'étiquette `labels` ; d'une fonction `predict(data)` qui renvoie la prédiction pour chaque exemple de la matrice `data` sous forme d'un vecteur. Pour implémenter un classifieur, vous pouvez dans la suite soit utiliser une classe munie de ces deux méthodes (cf squelette dans le code joint au TP), soit utiliser une fonction `fit(data,labels)` qui renvoie les paramètres estimés du modèle et une fonction `predict(data,params)` qui prend en entrée ces derniers.
Attention ! toutes ces fonctions prennent bien une matrice d'exemples en paramètre et non pas un unique exemple ! Le fichier `tools.py` contient un ensemble de fonctions pour la génération de jeux de données artificielles et leur visualisation. Le fichier `tp3-etu.py` contient quelques exemples d'utilisation et un squelette de code en implémentation objet.

1 Données et classifieur bayésien naïf

Pour ce TP, vous utiliserez deux types de données : des données artificielles en 2D afin de visualiser les résultats et des données réelles pour expérimenter.

Q 1.1 Données artificielles
La fonction `gen_arti` permet de générer 3 types de données en fonction du paramètre `data_type` (à 0,1 ou 2). La fonction `plot_data` permet de visualiser des données 2D. Expérimenter en explorant les paramètres et les différents jeux de données disponibles et en visualisant les données.

Q 1.2 Données réelles USPS
Ces données sont des images de chiffres manuscrits, toutes de taille 16×16. La fonction `load_usps` permet de charger les données à partir d'un fichier, la fonction `get_usps` permet de récupérer l'ensemble des exemples correspondant à une liste de chiffres donnée. Charger les données et visualiser quelques exemples. Dans la suite, nous ferons essentiellement de la classification binaire (entre deux classes) : soit entre deux chiffres, soit entre un chiffre contre tous les autres. A votre avis, quel est le problème le plus facile ? Quelles sont les paires de chiffres les plus difficiles à différencier ?

Q 1.3 Le modèle du classifieur bayésien naïf fait les hypothèses suivantes
- indépendance des dimensions : $p(\mathbf{x}|y) = \prod_{i=1}^{d} p(x_i|y)$
- pour chaque dimension i, $p(x_i|y)$ suit une loi normale de paramètre (μ_i, σ_i) (ou une bernoulli dans le cas discret binaire).

Pour classer un exemple $\mathbf{x}$, la règle de décision consiste à trouver le label qui maximise la probabilité a posteriori

$$\hat{y} = argmax_y p(y|\mathbf{x}) = argmax_y \frac{p(\mathbf{x}|y)p(y)}{p(\mathbf{x})} = argmax_y p(y) \prod_{i=1}^{d} p(x_i|y)$$

L'apprentissage de ce modèle se fait en estimant les paramètres des gaussiennes de chaque dimension pour chaque label par maximum de vraisemblance. Pour des raisons de précisions numériques, on utilise souvent le logarithme de la probabilité : $\hat{y} = argmax_y log(p(y)) + \sum_{i=1}^{d} log(p(x_i|y))$. Par ailleurs, les probabilités estimées à 0 sont remplacées par une valeur très faible, 10^{-10} par exemple (pourquoi ?).

Q 1.3.1 Implémenter un classifieur bayésien naïf. Vous utiliserez un vecteur μ pour stocker pour chaque classe la moyenne estimée pour chaque dimension, un vecteur σ pour stocker pour chaque classe la variance pour chaque dimension, et un réel pour stocker la probabilité a priori de chaque classe. Dans les fonctions, n'utiliser qu'une seule boucle `for` si besoin sur les labels mais surtout pas sur les

données : utiliser les fonctions de numpy pour cela ; utiliser les fonctions `np.mean` et `np.std(ddof=1)` pour estimer la moyenne et la variance corrigée des données.

Q 1.3.2 Tester les performances sur les jeux de données artificielles. Tracer les frontières de décision grâce à la fonction `plot_frontiere(data,predict,step=50)`. Commenter les résultats.

Q 1.3.3 Tester les performances sur les données réelles. Utiliser `show_usps` pour visualiser le vecteur μ d'une classe comme une matrice : à quoi correspond chaque cellule de la matrice ? qu'indique une valeur élevée ? une valeur faible ?

2 Descente de gradient

L'algorithme de descente de gradient est un algorithme itératif qui permet d'approcher le minimum d'une fonction convexe f donnée. Il consiste à calculer à chaque étape t le gradient de la fonction convexe au point courant $\mathbf{x}^t$ et de mettre-à-jour le point courant en se déplaçant dans le sens opposé au gradient : $\mathbf{x}^{t+1} = \mathbf{x}^t - \epsilon * \nabla f(\mathbf{x}^t)$.

Q 2.1 Coder une fonction `optimize(fonc,dfonc,xinit,eps,max_iter)` qui implémente l'algorithme du gradient avec en paramètre : `fonc` la fonction à optimiser, `dfonc` le gradient de cette fonction, `xinit` le point initial, `eps` le pas de gradient, `max_iter` le nombre d'itérations. Cette fonction doit rendre un triplet `(x_histo,f_histo,grad_histo)`, respectivement la liste des points $\mathbf{x}^t$,$f(\mathbf{x}^t)$ et $\nabla f(\mathbf{x}^t)$. Pour cela vous pouvez utiliser une liste pour stocker au fur et à mesure les points, puis transformer chaque liste en un tableau `np.array`.

Q 2.2 Testez votre implémentation sur les 2 fonctions suivantes :

- en 1d sur la fonction $f(x) = x\ \cos(x)$
- en 2d sur la fonction Rosenbrock (ou banana), définie par : $f(x_1, x_2) = 100*(x_2 - x_1^2)^2 + (1 - x_1)^2$.

Tracer :

- en fonction du nombre d'itérations, les valeurs de f et de la norme du gradient de f
- sur un même graphe avec deux couleurs différentes, la fonction f et la trajectoire de l'optimisation (les valeurs successives de $f(\mathbf{x}^t)$)
- la courbe $(t, \log(||\mathbf{x}^t - \hat{\mathbf{x}}||))$, avec $\hat{\mathbf{x}}$ l'estimation finale. Que remarquez vous ?

3 Applications : Classification plug-in et Régression logistique

On suppose dans la suite que le problème de classification est binaire à valeur dans $\{-1, +1\}$. Nous noterons $\eta(\mathbf{x}) = P(y = 1|\mathbf{x})$ et $X = \{\mathbf{x}^i, y^i\}_{i=1}^N$ l'ensemble d'apprentissage. Dans cette partie, vous allez implémenter deux approches de classification binaire, la classification par plug-in en utilisant une régression linéaire, et la régression logistique. Vous utiliserez la descente de gradient pour optimiser les paramètres du classifieur.

Asstuce d'implémentation : comme dans le cas de la régression linéaire, nous utiliserons des fonctions linéaires dont le résultat peut s'exprimer par produit scalaire. Afin d'incorporer le biais w_0 dans le produit scalaire et de soulager les notations et les lignes de codes, il est conseillé d'ajouter à la matrice des données une colonne de 1 en première colonne par exemple. La transformation par $f_{\mathbf{w}}$ correspond alors exactement à un produit scalaire : $f_{\mathbf{w}}(\mathbf{x}) =< \mathbf{w}, (1, x_1, x_2, \ldots, x_d) >$.

Classification plug-in Dans ce contexte, le problème de classification est vu comme un problème de régression aux moindres carrés : il s'agit de trouver la fonction f^* qui minimise $\mathbb{E}[(f^*(\mathbf{x}) - y)^2]$ (soit trouver f^* qui approxime au mieux $2\eta(\mathbf{x}) - 1$). Nous supposerons f^* linéaire, paramétrée par un vecteur de poids $\mathbf{w}$: $f_{\mathbf{w}}(\mathbf{x}) = \sum_{i=1}^d w_i x_i + w_0$.

- Inférence : le signe de $f_{\mathbf{w}}(\mathbf{x})$ conduit à la classification de $\mathbf{x}$: $+1$ si positif, -1 sinon.
- Coût à optimiser : $L(\mathbf{w}, X) = \sum_{i=1}^N (f_{\mathbf{w}}(\mathbf{x}) - y)^2 = \sum_{i=1}^N (w_0 + \sum_{j=1}^d w_j x_j^i - y^i)^2$

Régression logistique Ce modèle considère que la probabilité $\eta(\mathbf{x})$ peut être modélisé par $\sigma\left(f_{\mathbf{w}}(\mathbf{x})\right) = \frac{1}{1+e^{-f_{\mathbf{w}}(\mathbf{x})}}$, $f_{\mathbf{w}}$ fonction linéaire ; ainsi, $log\left(\frac{\eta(\mathbf{x})}{1-\eta(\mathbf{x})}\right)$ est approximée par $f_{\mathbf{w}}(x)$).
Par ailleurs, $P(y=-1|\mathbf{x}) = 1-\eta(\mathbf{x}) = \sigma(-f_{\mathbf{w}}(\mathbf{x}))$. Comme dans le cas du classifieur bayésien naïf, on utilise le maximum de vraisemblance pour trouver les paramètres optimaux (ou plus exactement minimiser la neg-log vraisemblance).

- Inférence : classer en $+1$ si $\sigma\left(f_{\mathbf{w}}(\mathbf{x})\right) > 0.5$, en -1 sinon ; soit classifier selon le signe de $f_{\mathbf{w}}(\mathbf{x})$.
- Coût à optimiser : $L(\mathbf{w}, X) = \sum_{i=1}^{N} log\left(1+e^{\left(-y^i f_{\mathbf{w}}(\mathbf{x}^i)\right)}\right) = -\sum_{i=1}^{N} log\left(1+e^{\left(-y^i\left(w_0+\sum_{j=1}^{d} w_j x_j^i\right)\right)}\right)$

Q 3.1 Calculer les gradients du coût pour les deux modèles. En déduire un algorithme d'optimisation.

Q 3.2 Implémenter les fonctions `mse(w,data,label)` et `grad_mse(w,data,label)` qui calculent le coût en w et le gradient du coût pour la classification plug-in.

Q 3.3 Implémenter les fonctions `reglog(w,data,label)` et `grad_reglog(w,data,label)` qui calculent le coût en w et le gradient du coût pour la régression logistique.

Q 3.4 Sur les données artificielles, tracer en 2D les deux fonctions de coût en fonction de w_1 et w_2 (considérer un biais w_0 nul). Que remarquez vous pour la classification plug-in ? Pourquoi n'y a-t-il pas cet effet pour la régression logistique ?

Q 3.5 Utiliser l'algorithme du gradient pour expérimenter sur les données artificielles les deux méthodes. Observer l'évolution du coût. Visualiser les frontières de décision. A quoi correspondent des valeurs élevées de $f_{\mathbf{w}}$? des valeurs faibles ? Est-ce la même signification pour les deux modèles ?

Q 3.6 Expérimenter sur les données réelles. Afficher le vecteur de poids comme une matrice (comme pour le classifieur bayésien naïf). Commenter et analyser vos résultats.