
TP n° 1 : Perceptron de F. Rosenblatt

Le fichier Python “`tp_perceptron_source.py`” est disponibles sur le site du cours. Il contient les fonctions utiles pour ce TP.

- DÉCOUVERTE DE PYTHON -

Consulter les pages suivantes pour démarrer ou bien trouver quelques rappels de Python :

- <http://www.python.org>
- <http://scipy.org>
- <http://www.numpy.org>
- <http://scikit-learn.org/stable/index.html>
- <http://www.loria.fr/~rougier/teaching/matplotlib/matplotlib.html>
- <http://jrjohansson.github.io/>

- INTRODUCTION -

Définitions et notations

On rappelle ici le cadre de la classification binaire supervisée, et l’on présente les notations que l’on utilisera :

- \mathcal{Y} l’ensemble des étiquettes des données (*labels* en anglais). On traite le cas binaire : il y a donc deux classes. Il est pratique de raisonner avec $\mathcal{Y} = \{-1, 1\}$ pour représenter les étiquettes, car on va considérer des signes au cours de ce travail².
- $\mathbf{x} = (x_1, \dots, x_p)^\top \in \mathcal{X} \subset \mathbb{R}^p$ est une observation, un exemple, un point (ou un *sample* en anglais). La j^{e} coordonnée de \mathbf{x} est la valeur prise par la j^{e} variable explicative (*feature* en anglais).
- $\mathcal{D}_n = \{(\mathbf{x}_i, y_i), i = 1, \dots, n\}$ un ensemble d’apprentissage contenant n exemples et leurs étiquettes,
- Il existe un modèle probabiliste qui gouverne la génération des nos observations selon des variables aléatoires X et $Y : \forall i \in \{1, \dots, n\}, (\mathbf{x}_i, y_i) \stackrel{i.i.d}{\sim} (X, Y)$.
- On cherche à construire à partir de l’ensemble d’apprentissage \mathcal{D}_n une fonction appelée classifieur, $\hat{f} : \mathcal{X} \mapsto \{-1, 1\}$ qui pour un nouveau point $\mathbf{x}_{\text{new}} \in \mathcal{X}$ fournit une étiquette $\hat{f}(\mathbf{x}_{\text{new}})$.

Génération artificielle de données

Dans un but d’expérimentation et de visualisation, il est plus aisé de travailler sur des données générées artificiellement. Ainsi on considère ici des variables explicatives (*features* en anglais) de dimension deux. Cela consiste à prendre $p = 2$ dans le formalisme ci-dessus.

- 1) Étudiez la fonction `rand_gauss(n, mu, sigmas)` qui engendre n observations selon la loi normale multi-dimensionnelle de moyenne le vecteur `mu` et de matrice de covariance la matrice diagonale de diagonale `sigmas` $= [\sigma_1, \sigma_2]$, *i.e.*, la matrice de variance covariance est : $\begin{pmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \end{pmatrix}$. Générer ensuite plusieurs jeux de données à l’aide des fonctions `rand_bi_gauss`, `rand_clown` et `rand_checkers` pour différents paramètres d’entrée `n, mu, sigma`. Que renvoient ces fonctions ? À quoi correspond la seconde variable de sortie ?

1. Avec la contribution majeure de Alexandre Gramfort et Joseph Salmon.

2. Noter que pour d’autres méthodes, il peut être plus pratique de prendre $\mathcal{Y} = \{0, 1\}$.

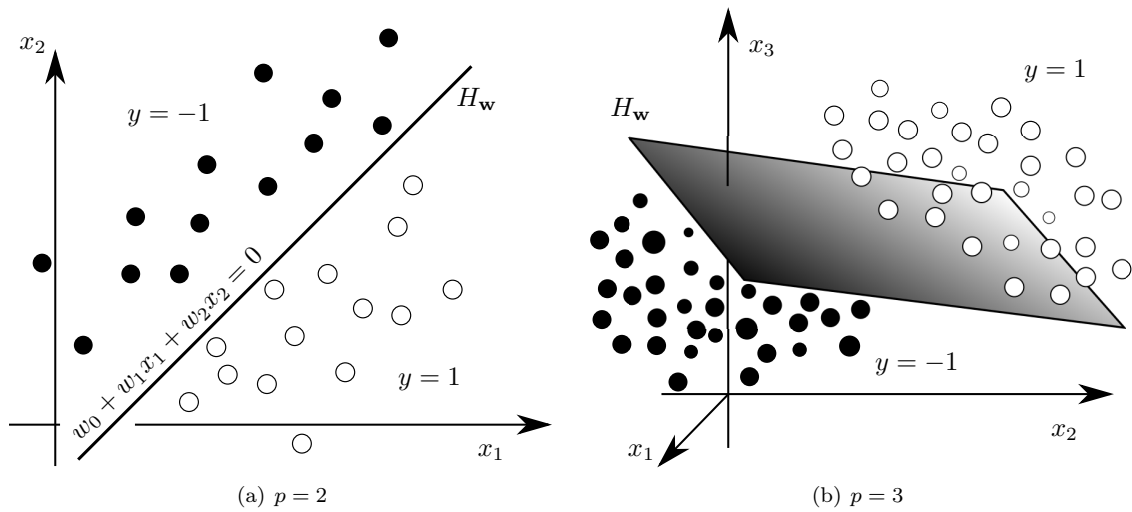


FIGURE 1 – Exemple de séparation de classe (1 pour les points blanc, -1 pour les noirs) par des hyperplans affines, en dimension $p = 2$ et $p = 3$

- 2) Conservez quelques jeux de données afin de les utiliser dans la suite : pour chacun, il faudra sauvegarder sous forme d'un tableau `numpy` à deux colonnes `X` les données, et dans un vecteur `Y` les labels correspondants à chaque exemple.
- 3) Utilisez la fonction `plot_2d` disponible dans `tp_perceptron_source.py` et qui permet de visualiser quelques jeux de données en fonction des étiquettes associées. Changer la couleur de la classe des -1 .

- PERCEPTRON -

Les classifieurs linéaires (affines)

Un classifieur linéaire est un classifieur qui associe à chaque observation \mathbf{x} une étiquette dans \mathcal{Y} (ici $\mathcal{Y} = \{-1, 1\}$) selon sa position par rapport à un hyperplan affine. Chaque classifieur linéaire est donc lié à un hyperplan (affines) de \mathbb{R}^p que l'on définit pour un certain vecteur directeur (aussi dit *vecteur normal*) $\mathbf{w} = (w_0, w_1, \dots, w_p)^\top \in \mathbb{R}^{p+1}$ par

$$H_{\mathbf{w}} = \left\{ \mathbf{x} \in \mathbb{R}^p : \hat{f}_{\mathbf{w}}(\mathbf{x}) := w_0 + \sum_{j=1}^p w_j x_j = 0 \right\}.$$

Pour classer une observation \mathbf{x} (*i.e.*, affecter une étiquette 1 ou -1) on utilise alors $\text{sign}(\hat{f}_{\mathbf{w}}(\mathbf{x}))$, où la fonction sign est définie par

$$\text{sign}(x) = \begin{cases} 1, & \text{si } x \geq 0, \\ -1, & \text{si } x < 0 \end{cases}.$$

Ainsi, la fonction $\mathbf{x} \mapsto \text{sign}(\hat{f}_{\mathbf{w}}(\mathbf{x}))$ est le classifieur binaire de frontière linéaire (affine) définie par \mathbf{w} .

L'objectif du perceptron est de trouver un hyperplan qui sépare le mieux possible les données en deux groupes. On aimerait donc que de chaque côté de l'hyperplan séparateur, les étiquettes soient le plus possible homogènes. Le vecteur \mathbf{w} est appelé vecteur de poids. Le coefficient w_0 est l'ordonnée à l'origine (*intercept* en anglais). Pour plus de détails sur l'intérêt de cette méthode on pourra se référer à [HTF09].

Utilisez les données artificielles déjà construites pour illustrer les questions suivantes.

- 1) A quoi correspond la frontière de décision du perceptron en dimension $p = 2$? Trouvez (à la main) une bonne séparatrice sur les trois jeux de données simulés. Quand est-ce que $\hat{f}_{\mathbf{w}}(\mathbf{x})$ est grand? négatif? positif? Quelle est la signification géométrique de cette fonction? À quoi correspond w_0 ?

- 2) Vérifiez que la fonction `predict(x,w)` prend en entrée un vecteur $\mathbf{x} \in \mathbb{R}^p$ et un vecteur poids $\mathbf{w} \in \mathbb{R}^{p+1}$ et renvoie le vecteur de prédiction $\hat{f}_{\mathbf{w}}(\mathbf{x})$. Vérifiez ensuite que `predict_class(x,w)` renvoie bien l'étiquette prédite $\text{sign}(\hat{f}_{\mathbf{w}}(\mathbf{x}))$.

Fonction de coût

Afin de mesurer l'erreur commise sur l'ensemble d'un jeu de données \mathcal{D}_n il est nécessaire de se fixer une fonction de perte $\ell : \mathbb{R} \times \mathcal{Y} \mapsto \mathbb{R}^+$ qui mesure le coût d'une erreur lors de la prédiction d'un exemple. Le coût que l'on veut minimiser (en fonction de \mathbf{w}) est $\mathbb{E} [\ell(\hat{f}_{\mathbf{w}}(\mathbf{x}), y)]$, l'espérance de la fonction de perte sur l'ensemble des données. Trois fonctions de perte sont utilisées habituellement et définies ci-dessous :

- le pourcentage d'erreur : $\text{ZeroOneLoss}(\hat{f}_{\mathbf{w}}(\mathbf{x}), y) = |y - \text{sign}(\hat{f}_{\mathbf{w}}(\mathbf{x}))|/2$,
- l'erreur quadratique : $\text{MSELoss}(\hat{f}_{\mathbf{w}}(\mathbf{x}), y) = (y - \hat{f}_{\mathbf{w}}(\mathbf{x}))^2$,
- l'erreur *hinge* (i.e., charnière en français) : $\text{HingeLoss}(\hat{f}_{\mathbf{w}}(\mathbf{x}), y) = \max(0, 1 - y \cdot \hat{f}_{\mathbf{w}}(\mathbf{x}))$.

Cette partie a pour but d'étudier ces différentes fonction de pertes. Ces trois fonctions sont codées dans le fichier source (ainsi que les gradients associés).

- 3) Supposons que $\mathbf{x} \in \mathbb{R}^p$ et $y \in \mathbb{R}$ soient fixes. Quelle est la nature des fonctions

$$\begin{aligned} \mathbb{R}^{p+1} &\rightarrow \mathbb{R}^+ \\ \mathbf{w} &\mapsto \ell(\hat{f}_{\mathbf{w}}(\mathbf{x}), y) \end{aligned}$$

pour les trois pertes étudiées : constante, linéaire, quadratique, constante par morceaux, linéaire par morceaux, quadratique par morceaux, etc. ?

- ALGORITHME DE DESCENTE DE GRADIENT STOCHASTIQUE -

Dans le cas général, il est bien sûr impossible de faire une recherche exhaustive de l'espace \mathbb{R}^{p+1} où évolue \mathbf{w} afin de trouver le coût minimum. De plus on ne peut pas observer $\mathbb{E} [\ell(\hat{f}_{\mathbf{w}}(\mathbf{x}), y)]$, on peut donc seulement tenter de minimiser sa contrepartie empirique : $\sum_{i=1}^n \ell(\hat{f}_{\mathbf{w}}(\mathbf{x}_i), y_i)$.

La méthode du perceptron consiste à utiliser une variante de l'algorithme de descente du gradient, une méthode usuelle et générale d'optimisation de fonction différentiable. La méthode de descente de gradient est itérative : à chaque étape, le poids courant est corrigé dans la direction du gradient, mais en sens opposé. L'algorithme converge dans le cas général vers un minimum local pour peu que le pas soit bien choisi. De plus, le minimum atteint est global pour les fonctions convexes.

La méthode du gradient stochastique est une variante qui propose de ne pas utiliser le gradient complet, qui requiert de calculer une somme sur les n observations, mais plutôt de tirer (aléatoirement ... ou non) un couple (\mathbf{x}_i, y_i) sur laquelle on calcul un gradient. On peut aussi montrer que cet algorithme converge sous certaines conditions cf. [SSBD14, page 157] ou [Bot98].

L'algorithme du perceptron est décrit de la manière suivante :

Algorithme 1 : Perceptron (version cyclique)

Data : les observations et leurs étiquettes $\mathcal{D}_n = \{(\mathbf{x}_i, y_i) : 1 \leq i \leq n\}$; le pas de gradient : ϵ ;
le nombre maximal d'itérations : n_{iter} ;

Result : \mathbf{w}

initialiser (aléatoirement) \mathbf{w} ; initialiser $j = 0$

```
while  $j \leq n_{\text{iter}}$  do
   $\mathbf{w} \leftarrow \mathbf{w}$ 
  for  $i = 1$  to  $n$  do
     $\mathbf{w} \leftarrow \mathbf{w} - \epsilon \nabla_{\mathbf{w}} \ell(\hat{f}_{\mathbf{w}}(\mathbf{x}_i), y_i)$ 
   $j \leftarrow j + 1$ 
```

Remarque La méthode du gradient stochastique est aussi disponible dans `sklearn` sous le nom `SGDClassifier` (SGD est l'abréviation *Stochastic Gradient Descent*). Une description est donnée sur la page : <http://scikit-learn.org/stable/modules/sgd.html>.

- 4) Décrire un pseudo algorithme “Perceptron version aléatoire”, une variante qui visite les observations en effectuant un tirage aléatoire uniforme (faire le cas avec remise et sans remise). On ne demande pas de re-coder cette fonction ici. C’est cette version qui est généralement appelée méthode de descente de gradient stochastique. Remarquez que c’est celle qui est proposée par défaut dans la fonction `gradient`.
- 5) On va observer graphiquement avec `plot_gradient` l’évolution de $\frac{1}{n} \sum_{i=1}^n \ell(\hat{f}_{\mathbf{w}}(\mathbf{x}_i), y_i)$ selon \mathbf{w} suivant les étapes de l’algorithme. La descente de gradient simple est aussi appelée *batch* (i.e., lot ou stock en français), et consiste à calculer le vrai gradient $\frac{1}{n} \sum_{i=1}^n \nabla_{\mathbf{w}} \ell(\hat{f}_{\mathbf{w}}(\mathbf{x}_i), y_i)$. Lancer l’exemple avec et sans l’option stochastique. Pourquoi le nombre d’itérations est `niter * len(y1)` pour le cas stochastique ?
- 6) Expérimentez sur différents jeux de données : utiliser soit les fonctions fournies dans le fichier source, soit la fonction de `sklearn`. Étudiez les performances selon les points suivants : le nombre d’itérations, la fonction de coût, la difficulté du problème (si les classes sont facilement séparables ou non par un hyperplan). Observez-vous des comportements étranges, si oui quelle en est la raison ?
- 7) Étudiez comme précédemment le comportement de l’algorithme de descente de gradient, avec l’option `stoch=True` désactivée.
- 8) Question optionnelle : étudiez numériquement la vitesse de convergence dans le cas suivant : les \mathbf{x}_i sont des points uniformément repartis sur les segments $\{0\} \times [0, M]$ (alors les y_i valent 1) ou bien sur le segment $\{\delta\} \times [0, M]$ (alors les y_i valent -1). De plus la proportion de 1 est égale à $1/2$. On regardera l’impact de δ, M et n sur le temps de convergence du perceptron.
- 9) Proposez/codez des variantes de conditions d’arrêt pour l’algorithme.
- 10) Afficher sur une même figure l’évolution des frontières en fonctions des itérations. On pourra utiliser la fonction `plot_2d` et son argument `alpha_choice` pour cela.
- 11) Question optionnelle : trouver une fonction de perte telle que l’algorithme du perceptron soit équivalent à la version donnée par l’Algorithme 2 (qui est la version initiale de l’algorithme). Interprétez la condition suivante : $\hat{f}_{\mathbf{w}}(\mathbf{x}_i) \cdot y_i \leq 0$.

Algorithme 2 : Perceptron “classique” (version cyclique)

Data : les observations et leurs étiquettes $\mathcal{D}_n = \{(\mathbf{x}_i, y_i) : 1 \leq i \leq n\}$; le pas de gradient : ϵ ;
le nombre maximal d’itérations : n_{iter} ;

Result : w

initialiser (aléatoirement) \mathbf{w} ; initialiser $j = 0$

```

while  $j \leq n_{\text{iter}}$  do
     $\mathbf{w} \leftarrow \mathbf{w}$ 
    for  $i = 1, \dots, n$  do
        if  $\hat{f}_{\mathbf{w}}(\mathbf{x}_i) \cdot y_i \leq 0$  then
             $\mathbf{w} \leftarrow \mathbf{w} + \epsilon \begin{pmatrix} 1 \\ \mathbf{x}_i \end{pmatrix} \cdot y_i$ 
     $j \leftarrow j + 1$ 

```

Pour aller plus loin on pourra consulter hagan.okstate.edu/4_Perceptron.pdf, et [Sha11, SSBD14] pour un point de vue plus moderne sur la technique du gradient stochastique.

Perceptron : linéaire ... seulement ?

- 12) Quelle est la formule analytique d’une ellipse, d’une hyperbole et d’une parabole en 2D ?
- 13) Proposez une méthode pour réussir à classer le jeu de données `clown` en créant des interactions d’ordre deux. En pratique, peut-on généraliser au delà de l’ordre deux facilement ? On pourra utiliser la fonction `poly2` du fichier source, qui plonge les données bi-dimensionnelles dans l’espace des fonctions polynomiales de degré 2 en les données.
- 14) Sur le jeu de données `clown`, faites quelques expériences en transformant vos données et tracez les frontières de décision.

Références

- [Bot98] L. Bottou. Online learning and stochastic approximations. *On-line learning in neural networks*, 17(9) :142, 1998. [3](#)
- [HTF09] T. Hastie, R. Tibshirani, and J. Friedman. *The elements of statistical learning*. Springer Series in Statistics. Springer, New York, second edition, 2009. <http://www-stat.stanford.edu/~tibs/ElemStatLearn/>. [2](#)
- [Sha11] S. Shalev-Shwartz. Online learning and online convex optimization. *Foundations and Trends in Machine Learning*, 4(2) :107–194, 2011. <http://www.cs.huji.ac.il/~shais/papers/OLsurvey.pdf>. [4](#)
- [SSBD14] S. Shalev-Shwartz and S. Ben-David. *Understanding machine learning : From theory to algorithms*. Cambridge University Press, 2014. [3](#), [4](#)