Un perceptron simple avec Python et Numpy

Introduction

Le **perceptron** est l'ancêtre des réseaux de neurones. Il a été inventé par Frank Rosenblatt en 1958 dans le cadre de la conception par IBM d'une machine dédiée à la reconnaissance d'images.

L'architecture du perceptron est très simple puisqu'il s'agit en réalité d'un sommateur qui effectue le **produit scalaire** de deux vecteurs.

Historiquement, le percpetron n'a pas rempli les espoirs qui étaient placé en lui, notamment parce que M. Minsky et S. Papert ont montré que les possibilités du système étaient limités à faire de la **classification linéaire** et qu'il ne pourrait jamais résoudre des problèmes de séparation complexes.

> **N.B.** A la suite de cette démonstration, il faudra attendre plus de vingt ans pour que G. Hinton améliore l'idée du perceptron avec la notion de couches multiples qui donnera naissance aux réseuax de neurones modernes.

Le perceptron est donc à la fois le premier classificateur linéaire binaire de l'histoire et le premier système à implémenter la notion d'**apprentissage**.

L'algorithme fait partie des techniques d'**apprentissage supervisé**, c'est-à-dire que l'on cherche à associer des jeux de données à des catétories pré-définies.

Dans le mécanisme du perceptron, les données d'entrée sont directement reliés à la sortie.

Description

Description mathématique

D'un point de vue moderne, le perceptron est un algorithme qui associe un **vecteur** de valeurs entrée, appellées « **caractéristiques** » du problème, à une valeur scalaire en sortie, généralement booléenne 0/1, et appelée **réponse**.

Le perceptron effectue la tâche relativement simple de calculer l'équation :

$y = \sum_{i=1}^{n} w_i x_i$ (noté aussi $y = \vec{W} \cdot \vec{X}$)

où :

- $\vec{X}$ représente le vecteur des données d'entrée
- $\vec{W}$ représente un vecteur de coefficients de pondération appelés **poids**

Si le résultat de l'équation est positif, l'exemple sera présumé appartenir à la classe 1 ; s'il est négatif, à la classe 0.

Mathématiquement parlant, le problème consiste à trouver les valeurs du vecteur $\vec{W}$ qui satisfont l'équation, sachant que `X` et `y` sont connus.

Le problème n'est pas très éloigné de la régression linéaire, car l'on cherche également l'équation d'une droite (ou d'un hyerplan dans le cas général). Mais l'hypothèse est différente. Si l'on estime dans le cas de la régression que les points sont alignés de manière cohérente le long de la droite, on espère dans le cas de la classification qu'il en seront le plus éloignés possible.

Description algorithmique

Le perceptron est un algorithme itératif, qui trouve une solution par **approximations successives**. De ce point de vue, il est proche dans l'esprit de la méthode de Newton pour trouver les racines d'un polynôme.

Les données initiales sont :

- $D = \{(x_i, d_i), i \in \mathbb{N}\}$ un ensemble de couples où x_i est un *exemple* que l'on souhaite classer et d_i la classe d'appartenance (0 ou 1)
    - chque exemple est lui-même un vecteur de données de taille `n` $x_i = x_{i,j}, 1 \leq i \leq n$
- `r` est une variable appelée **pas d'apprentissage** qui paramètre la vitesse d'évolution du système
- $W = \{w_i, i \in \mathbb{N}\}$ un vecteur de même taille `n` que les exemples x_i
- γ, une mesure de précision estimant la qualité du résultat de l'apprentissage

Dans sa version la plus simple, l'algorithme peut s'écrire comme suit :

1. Initialiser le vecteur `W` avec des valeurs arbitraires. Par exemple, une solution est de mettre tous les poids à zéro, une autre est de leur attribuer des valeurs aléatoires
2. Tant que l'approximation ϵ est supérieure à γ, itérer sur une varaible `t`:
    1. Pour chaque exemple X_j :
        - Calculer le produit scalaire $y_j(t) = \overrightarrow{W(t)}.\overrightarrow{X_j}$

- Calculer le produit scalaire $y_j(t) = f[w(t) \cdot x_j]$
- Actualiser les valeurs de poids :
  $w_i(t+1) = w_i(t) + r * (d_j - y_j(t)) * x_{j,i}$

2. Calculer l'estimateur ϵ : $\epsilon = \frac{1}{n} \sum_{k=1}^{n} |d_k - y_k(t)|$

On peut affiner l'algorithme du perceptron an ajoutant une **fonction d'activation**. Bien que cela ne soit pas très utile dans le cas du perceptron, nous verrons plus tard que les fonctions d'activation jouent un rôle trsè important dans les réseaux de neurones multi-couches.

Une fonction d'activation est, comme son nom le laisse deviner, une fonction qui indique si un neurone et actif ou non, c'est-à-dire par analogie avec un interruteur, ouvert ou fermé. L'évolution des réseaux de neurones conduit en fait à créer des fonctions d'activation moins tranchées, mais la version simple est appelée **fonction d'Heavyside**, qui est juste un interrupteur :

$$H(x) = \begin{cases} 1 & \text{si } x > 0 \\ 0 & \text{si } x \leq 0 \end{cases}$$

Le perceptron en Python

Implémentation en Python

Pour implémenter un perceptron en Python, on peut suivre les étapes suivantes :

Etape 1 : importer les données

Comme dans tous les systèmes d'analyse de données, il est d'abor nécessaire de comprendre et de mettre en forme les données. Celles-ci peuvent provenir de sources extrêmement différentes (bases de données, APi, fichiers CSV, etc.) et peuvent être structurées de manières diverses.

La première tâche est de construire une fonction `getData` qui rendra deux tableaux correspondant, pour le premier (X) aux exemples d'apprentissage, et pour le second (y) aux étiquettes associées à ces exemples.

on voudrait par exemple pour écrire :

```
X, y = Perceptron.getData(source)
```

`getData` peut naturellement s'appuyer sur divers modules (`reader` pour CSV, `request` pour les API, etc.) en fonction de la source et du format des données.

Etape 2 : Prédire la classe d'un exemple

La fonction de prédiction `predict` est la plus simple à éctire, car elle fait uniquement appel au produit scalaire.

On souhaite pouvoir écrire :

```
classe = Perceptron.predict(example, weights)
```

où :

- `example` est un échantllon (un vecteur de données)
- `weights` est un vecteur dont les valeurs sont les coefficients (poids)

Cette fonction ne devrait pas rendre uniquement le résultat du produit scalaire, mais le résultat de la *fonction d'activation*.

On peut donc écrire deux fonctions qui exécuteront le produit scalaire et l'activation (idéalement, ces fonctions seraient dans des modules distincts).

Naturellement, si vous voulez tester cette fonction à ce stade du développement, il y a de fortes chances que vous n'obteniez pas de bons résultats.

Etape 3 : Fonction d'apprentissage

C'est évidemment le cœur de l'algorithme. Nous allons chercher à écrire une fonction `fit` pour calculer le vecteur des poids.

Cette fonction admet (au moins) les paramètres suivants :

- un tableau Numpy contenant les exemples d'apprentissage
- un tableau Numpy contenant les étiquettes associées aux exemples
- la valeur du pas d'apprentissage
- le nombre d'itération (époques) et/ou la précision admise

La fonction `fit` devra :

1. Initialiser le vecteur des poids
2. Itérer sur le nombre d'époques (le plus simpole) ou jusqu'à atteindre la précision voulue :
    - A chaque itération, examiner tous les exemples du jeu de données :
        1. calculer la réponse en fonction des poids
        2. calculer l'écart (l'erreur) entre la réponse attendue et la réponse calculée
        3. réactualiser toutes les valeurs du vecteur des poids

En option, vous pouvez mélanger les exemples avant d'entamer la phase

En option, vous pouvez mélanger les exemples avant d'entamer la phase d'apprentissage.

Etape 4 : Amélioration du dispositif

A ce stade, le perceptron n'est pas optimal car il tend à calculer une équation qui passe par l'origine. On peut améliorer les performances en ajoutant aux caractéristiques des exemples une valeur arbitraire, par exemple initialisée à zéro et qui correspondra au décalage de l'origine (l'« *intercept* » de la régression linéaire).

dans ce cas, à chaque époque, il faudra recalculer cet *intercept* en lui ajoutant le produit du pas d'apprentissage par la valeur de l'erreur

scikit-learn

Le perceptron fait partie des modèles linéaires connus de scikit-learn.

Documentation

La classe `Perceptron` admet une série de paramètres d'instanciation, parmi lesquels :

- `eta0` : valeur du pas d'apprentissage
- `tol` : seuil d'approximation en dessous duquel l'approntissage est estimé correct
- `max_iter` : nombre maxiaml d'itérations (appelées **époques** ou **epochs** en anglais)
- `shuffle` : mélange les exemples d'apprentissage (donne généralement de meilleurs résultats)
- `warm_start` : reprned une phase d'apprentissage avec un nouveau jeu d'exemples en tenant compte des résultats précédents

Les principales méthodes associées à la classe `Perceptron` sont :

- `fit` : méthode exécutant la phase d'apprentissage, construit le **modèle**
- `predict` : rend la classe d'appartenance d'un exemple, d'apès le modèle appris
- `score` : évalue le score de l'apprentissage sur un jeu de données

Exemple :

```
1 from sklearn.linear_model import Perceptron
2 X = [[1,1], [2,2], [3,3]]
3 y = [1,0,1]
4 model = Perceptron(tol=1e-3)
5 Perceptron()
```

```
6 model.score(X, y)
```

Exercices

[A] Sonar

- Jeu de données

Des mesures d'écholocation sont destinées à prédire si un objet sous-marin renvoyant un écho est une mine ou un rocher.

Ecrivez en Python un perceptron qui répondra aux questions posées par de nouvelles mesures

[B]

Addenda

Classes multiples

Comme souvent, il est possible de disciminer plus de deux classes avec un perceptron, à condition que l'on reste dans le domaine de la classification linéaire.

Dans ce cas, il suffit d'ajouter autantde neurones que l'on a de classes ; ces neurones sont considérés comme indépendants dans la phase d'apprentissage. Lors de la prédiction, le neurone ayant la prlus grande valeur se considéré comme le « gagnant ».