Université de Paris

Perceptron moyenné

Guillaume Wisniewski
guillaume.wisniewski@u-paris.fr

janvier 2020

Nous allons décrire, dans ce document, un algorithme d'apprentissage permettant de déterminer les poids d'un classifieur linéaire à partir d'un ensemble d'apprentissage. Cet algorithme est une généralisation de l'algorithme du perceptron qui permet d'obtenir de très bonnes performances sur de très nombreuses tâches de TAL.

L'algorithme du perceptron, introduit dans le dernier cours, est un algorithme d'apprentissage par *correction d'erreurs* : lors de l'apprentissage, dès une étiquette erronée est prédite, les poids sont mis à jour afin de corriger cette erreur ou, du moins, réduire l'impact de celle-ci, en diminuant le score de l'étiquette prédite par erreur et en augmentant le score de l'étiquette qui aurait dû être prédite. Cette approche fonctionne lorsque les données sont linéairement séparables : dans ce cas, l'algorithme d'apprentissage converge après un nombre d'itérations polynomial par rapport au nombre d'exemples.

Si les données ne sont pas linéairement séparables (ce qui est en général le cas), il faut introduire un critère d'arrêt en spécifiant, par exemple, un nombre maximum de corrections devant être effectuées. Cette approche est toutefois problématique : si les 999 premiers exemples de l'ensemble d'apprentissage correctement classifiés mais que l'étiquette prédite pour 1 000^{e} exemple est fausse, les paramètres du perceptron seront corrigés sans tenir compte du fait que ceux-ci permettent de classifier correctement les 999 premiers exemples. Il est, de plus, tout à fait possible, que, après cette correction, ces exemples ne soient plus correctement classifiés : l'algorithme d'apprentissage du perceptron n'a pas de « mémoire » et, suivant sa position dans l'ensemble d'apprentissage, une erreur aura un impact plus ou moins grand sur le vecteur de poids.

Une généralisation simple de l'algorithme d'apprentissage du perceptron permet d'éviter ce problème : le perceptron moyenné. Cette généralisation, décrite dans l'algorithme 1 consiste à conserver le vecteur de poids à chaque itération et à utiliser, lors de l'inférence (c.-à-d. après l'apprentissage) un perceptron dont le vecteur de poids est constitué par la moyenne de tous les vecteurs de poids vus lors de l'apprentissage. En pratique, il n'est pas nécessaire de conserver les valeurs de tous les vecteurs de paramètres, mais uniquement la somme de ceux-ci (le vecteur $\mathbf{a}$ dans l'algorithme 1) et il n'est pas nécessaire de diviser celle-ci par le nombre d'itération

puisque un perceptron de paramètre $\mathbf{w}$ fera exactement les mêmes prédictions qu'un perceptron de paramètres $\alpha \times \mathbf{w}$ quelque soit le réel α. Il est primordial de noter que l'apprentissage du perceptron moyenné est exactement le même que l'apprentissage d'un perceptron « normal » et repose sur les corrections successives du vecteur de poids $\mathbf{w}$, le vecteur de poids moyenné $\mathbf{a}$ n'est utilisé que lors de l'inférence, une fois que l'apprentissage est terminée.

Le nombre d'itération *MAX_EPOCH* est un hyper-paramètre de l'algorithme qui est fixé avant l'exécution de celui-ci et dont la valeur est généralement déterminée par validation croisée.

Algorithm 1 Algorithme d'apprentissage du perceptron moyenné.

Require: a training set $\left(\mathbf{x}^{(i)}, y^{(i)}\right)_{i=1}^{n}$, hyper-parameter *MAX_EPOCH*

1: $\mathbf{w} \leftarrow 0$
2: $\mathbf{a} \leftarrow 0$
3: **for** *epoch* = 1 to *MAX_EPOCH* **do**
4: shuffle training set
5: **for** $i = 1$ to n **do**
6: $y^* = \text{argmax}_{y \in \mathcal{Y}} \left\langle \mathbf{w}_y | \mathbf{x}^{(i)} \right\rangle$
7: **if** $y^* \neq y^{(i)}$ **then**
8: $\mathbf{w}_{y^{(i)}} \leftarrow \mathbf{w}_{y^{(i)}} + \mathbf{x}^{(i)}$
9: $\mathbf{w}_{y^*} \leftarrow \mathbf{w}_{y^*} - \mathbf{x}^{(i)}$
10: **end if**
11: $\mathbf{a} \leftarrow \mathbf{a} + \mathbf{w}$
12: **end for**
13: **end for**
14: Use $\mathbf{a}$ as the weight vector.

Implémentation « rapide » Dans l'algorithme 1, la mise à jour du vecteur $\mathbf{a}$ (ligne 11) a une complexité qui est linéaire par rapport au nombre de classe et au nombre de caractéristique et est particulièrement coûteuse : en pratique, le temps de calcul de cette opération devient prohibitif si celle-ci est réalisée à chaque itération de l'algorithme (c.-à-d. après chaque exemple).

Il est possible de réduire fortement la complexité de l'apprentissage en remarquant que le vecteur de paramètres n'est modifié que lorsqu'un exemple est mal classé. Plutôt que de réaliser l'opération à chaque étape, il est donc possible d'effectuer la mise à jour suivante du vecteur $\mathbf{a}$ uniquement lorsque le perceptron fait une erreur de prédiction de la manière suivante (avant de mettre à jour le vecteur $\mathbf{w}$) :

$$\mathbf{a} \leftarrow \mathbf{a} + n_{\text{last\_error}} \times \mathbf{w} \tag{1}$$

où $n_{\text{last\_error}}$ est le nombre d'exemples vus depuis la dernière fois où le perceptron a mal prédit un exemple du corpus d'apprentissage. Faire cette mise à jour revient à effectuer la mise à jour de $\mathbf{a} \leftarrow \mathbf{a} + \mathbf{w}$ chaque fois qu'un exemple aura été bien classé.

Cette première observation permet de réduire le nombre de mise à jour du vecteur **a** (qui ne sera plus modifié que lorsque l'algorithme fait une erreur, ce qui ne devrait pas arriver trop souvent). Mais, chaque mise à jour aura toujours la même complexité (linéaire par rapport au nombre de caractéristiques et d'étiquettes). Celle-ci peut encore être réduite en distinguant les composantes de **a** devant être mises à jour : lors de l'apprentissage, si une observation **x** est mal classée, les paramètres correspondant aux caractéristiques « actives » (pour les classes de l'étiquette prédite et de l'étiquette de référence) seront modifiés. Il est possible de ne mettre à jour que ces paramètres en gardant trace de la dernière fois que la valeur correspondant à une paire (étiquette, caractéristique) a été mise à jour.

L'algorithme d'apprentissage utilise ainsi sur les trois structures de données suivantes : deux dictionnaires de dictionnaires permettant de représenter les vecteurs **a** et **w** (la première clé représentant une classe et la seconde une caractéristique, la valeur associée étant la valeur de cette caractéristique pour cette classe) et un dictionnaire permettant de stocker pour une paire (étiquette, caractéristique) la dernière fois que celle-ci a été mise à jour.

Lors d'une erreur, les vecteurs **a** et **w** sont alors mis à jour de la manière suivante :

```
# x: observation prédite
# gold: étiquette de référence
# pred: étiquette prédite
for feat in x:
  # mise à jour de l'accumulateur
  a[gold][feat] += (n_examples - last_update[gold, feat]) * w[gold][feat]
  last_update[gold, feat] = n_examples

  a[pred][feat] += (n_examples - last_update[pred, feat]) * w[pred][feat]
  last_update[pred, feat] = n_examples

  # mise à jour de w
  w[gold][feat] += 1
  w[pred][feat] -= 1
```

où `n_examples` est le nombre d'itérations ayant été effectuées (d'exemples ayant été déjà parcouru). Il ne faut pas oublier, à la fin de l'apprentissage, d'effectuer une dernière mise à jour de **a**.