Fouille de Données Massives
TD - Une approche par Boosting

M2 Informatique - SISE (2022-2023)

Guillaume Metzler

Institut de Communication (ICOM)
Université de Lyon, Université Lumière Lyon 2
Laboratoire ERIC UR 3083, Lyon, France

guillaume.metzler@univ-lyon2.fr

Abstract

Le but de ce dernier TD est de vous faire travailler sur le boosting et sur l'approximation des méthodes à noyaux. Plus précisément, nous allons chercher à approximer des méthodes à noyaux en se basant sur différents concepts :

- le *boosting*, plus précisément le *gradient boosting*
- les *Random Fourier Features* (RFF) qui constituent une méthode d'approximation des noyaux qui sont *invariants par translation*
- les *landmarks*

Nous allons voir comment combiner ces différentes approches pour créer des approximations performantes et rapides qui peuvent constituer de bonnes alternatives à l'apprentissage d'un noyau gaussien.

Avant d'attaquer ce TD, il faudra terminer le TD précédent afin de bien en saisir l'intérêt mais aussi pour avoir pu appréhender cette notion de *landmarks* qui apparaît de façon indirecte dans le sujet de TD précédent. Ces *landmarks* étant utilisés pour construire notre première approximation des méthodes à noyaux en considérant un sous-ensemble du jeu de données pour la construction du modèle.

1 Contexte

Dans la suite, nos données $\mathbf{x}$ sont des observations qui sont décrites par d variables, *i.e.* ce sont des vecteurs de $\mathcal{X} = \mathbb{R}^d$. A chaque donnée est associée une étiquette que l'on notera y et qui ne prendra ses valeurs dans l'ensemble $Y = \{-1, +1\}$.

On commence par rappeler/présenter les différents outils nécessaires à la compréhension du sujet en revenant sur les méthodes à noyaux puis en présentant les *Random Fourier Features* (RFF).

1.1 Méthodes à noyaux

On se place dans le contexte d'apprentissage où l'on dispose d'un ensemble $S = \{(\mathbf{x}_j, y_j)\}_{j=1}^m$ où $\mathbf{x}_j \in \mathcal{X} = \mathbb{R}^d$ et $y_j \in \mathcal{Y} = \{-1, +1\}$. Notre objectif est d'apprendre un séparateur un linéaire f, *i.e.* un hyperplan, qui dépend d'un paramètre $\mathbf{w}$:

$$f_{\mathbf{w}}(\mathbf{x}) = \langle \mathbf{w}, \mathbf{x} \rangle.$$

Le paramètre $\mathbf{w}$ est obtenu en minimisant une certaine fonction de loss, par exemple la hinge loss dans le cas d'un algorithme de type SVM.

Ces méthodes linéaires se révèlent peu efficaces en pratique, notamment lorsque le jeu de données ne présente aucune relation linéaire entre l'*Input Space* $\mathcal{X}$ et l'*Output space* $\mathcal{Y}$, c'est pourquoi on utilise souvent des méthodes non linéaires comme les méthodes à noyaux.
Ces méthodes à noyaux consistent à projeter les données de $\mathcal{X}$ dans un autre espace $\mathcal{Z}$ dans lequel la tâche pourra se résoudre linéairement. L'espace $\mathcal{Z}$, appelé espace latent, sera potentiellement de dimension infinie. La question est de savoir comment apprendre une telle projection ! C'est à ce moment qu'intervient l'*astuce du noyau* qui ne nécessite alors pas d'expliciter une telle transformation. Il suffit de définir une fonction $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ avec des bonnes propriétés (PSD) pour que la transformation sous-jacente devienne implicite. Ce résultat est une conséquence du Théorème de Mercer [Mercer, 1909], qui dit que si les hypothèses sur k sont vérifiées, alors il existe une fonction $\varphi : \mathcal{X} \to \mathcal{Z}$ telle que

$$k(\mathbf{x}, \mathbf{x}') = \langle \varphi(\mathbf{x}), \varphi(\mathbf{x}') \rangle_{\mathcal{Z}}. \tag{1}$$

A partir de cela, nous sommes capables de construire un nouveau classifieur linéaire, dans l'espace latent , mais qui donnera un classifieur non linéaire dans l'espace de départ, *i.e.* l'Input Space $\mathcal{X}$:

$$f(\mathbf{x}) = \sum_{j=1}^{m} \alpha_j k(\mathbf{x}_j, \mathbf{x}) = \langle \mathbf{w}, \varphi(\mathbf{x}) \rangle_{\mathcal{Z}}. \tag{2}$$

Sur le plan pratique, cela nécessite de calculer une matrice de *similarités* $\mathbf{K} \in \mathbb{R}^{m \times m}$ entre les points de l'ensemble S :

$$\mathbf{K} = \begin{pmatrix} k(\mathbf{x}_1, \mathbf{x}_1) & k(\mathbf{x}_1, \mathbf{x}_2) & \dots & k(\mathbf{x}_1, \mathbf{x}_m) \\ k(\mathbf{x}_2, \mathbf{x}_1) & k(\mathbf{x}_2, \mathbf{x}_2) & \dots & k(\mathbf{x}_1, \mathbf{x}_m) \\ \vdots & \vdots & \ddots & \vdots \\ k(\mathbf{x}_m, \mathbf{x}_1) & k(\mathbf{x}_m, \mathbf{x}_2) & \dots & k(\mathbf{x}_m, \mathbf{x}_m) \end{pmatrix}.$$

Dans certains contextes, il est nécessaire d'inverser une telle matrice pour trouver notre classifieur linéaire. Ainsi, même si notre méthode permet d'apprendre des classifieurs complexes, elle ne passe pas forcément à l'échelle. En effet, pour l'apprentissage de la plupart des modèles, nous serons amenés à inverser cette matrice, ce qui n'est pas réalisable dans un contexte de *Big Data*.

1.2 Random Fourier Features

Ali Rahimi et Ben Recht [Rahimi and Recht, 2007] proposent une approche différente : approximer le produit interne ci-dessus dans (1) avec une projection aléatoire $u : \mathbb{R}^d \to \mathbb{R}^p$ où p est potentiellement très petit, *i.e.* :

$$k(\mathbf{x}, \mathbf{x}') = \langle \varphi(\mathbf{x}), \varphi(\mathbf{x}') \rangle_{\mathcal{Z}} \simeq \langle u(\mathbf{x}), u(\mathbf{x}') \rangle.$$

Si l'on revient au Théorème de Mercer, nous pouvons alors écrire notre séparateur linéaire de façon analogue à celui décrit en (2) mais en utilisant cette fois-ci notre approximation :

$$\begin{aligned} f(\mathbf{x}) &= \sum_{j=1}^{m} \alpha_i k(\mathbf{x}_j, \mathbf{x}), \\ &= \sum_{j=1}^{m} \alpha_j \langle \varphi(\mathbf{x}), \varphi(\mathbf{x}_j) \rangle_{\mathcal{Z}}, \\ &\simeq \sum_{j=1}^{m} \alpha_j \langle u(\mathbf{x}), u(\mathbf{x}_j) \rangle. \\ &= \langle \mathbf{w}, \varphi(\mathbf{x}) \rangle_{\mathcal{Z}}. \end{aligned}$$

Notre fonction u va donc constituer une approximation de cette fonction φ et on va donc projeter nos données à l'aide de la transformation u et directement apprendre

un séparateur linéaire sur ces données projetées, ce qui sera beaucoup plus rapide !
Il reste savoir comment définir une telle projection pour des méthodes à noyaux.

Pour cela les auteurs [Rahimi and Recht, 2007] ont fait l'observation suivante. Considérons un vecteur $\boldsymbol{\omega} \in \mathbb{R}^d$

$$\boldsymbol{\omega} \sim \mathcal{N}(0, \mathbf{I})^d$$

et une fonction $h : \mathcal{X} \to$ telle que

$$h : \mathbf{x} \mapsto \exp(i\boldsymbol{\omega}^T \mathbf{x}),$$

où i désigne le nombre complexe vérifiant $i^2 = -1$. Regardons maintenant ce qu'il se passe si on prend l'espérance selon $\boldsymbol{\omega}$ du carré de cette fonction h :

$$\begin{aligned}
\mathbb{E}_{\boldsymbol{\omega}}[h(\mathbf{x})h(\mathbf{x}')^\star] &= \mathbb{E}_{\boldsymbol{\omega}}\left[\exp(i\boldsymbol{\omega}^T(\mathbf{x}-\mathbf{x}'))\right], \\
&= \int_{\mathbb{R}^d} p(\boldsymbol{\omega}) \exp(i\boldsymbol{\omega}^T(\mathbf{x}-\mathbf{x}'))d\boldsymbol{\omega}, \\
&= \exp\left(\frac{1}{2}(\mathbf{x}-\mathbf{x}')^T(\mathbf{x}-\mathbf{x}')\right).
\end{aligned}$$

En d'autres termes, l'espérance de $h(\mathbf{x})h(\mathbf{x}')^\star$ n'est rien d'autre que notre noyau gaussien. Plus précisément, on peut observer que certains noyaux, peuvent d'écrire comme la transformée de Fourier de certaines distributions connues, c'est ce que l'on appelle le théorème de Bochner [Rudin, 2017].

L'idée est ensuite d'approximer k notre noyau en tirant un nombre suffisamment important de vecteurs $\boldsymbol{\omega}$, disons R, selon une loi gaussienne afin d'approcher cette espérance et donc d'approximer notre noyau gaussien :

$$\begin{aligned}
k(\mathbf{x}, \mathbf{x}') &= \int_{\mathbb{R}^d} p(\boldsymbol{\omega}) \exp(i\boldsymbol{\omega}^T(\mathbf{x}-\mathbf{x}'))d\boldsymbol{\omega}, \\
&= \mathbb{E}_{\boldsymbol{\omega}}\left[\exp(i\boldsymbol{\omega})^T(\mathbf{x}-\mathbf{x}')\right], \\
&\simeq \frac{1}{R}\sum_{r=1}^{R} \exp(i\boldsymbol{\omega}_r^T(\mathbf{x}-\mathbf{x}')).
\end{aligned}$$

Malheureusement les quantités qui entrent en jeu pour le moment sont des nombres complexes ce qui les rend peu utilisables en pratique. Regardons comment on peut s'affranchir des nombres complexes par la suite et approximer notre noyau par une somme de fonction cos.

1.3 Vers un usage pratique

Tout d'abord, comme nous vecteurs $\boldsymbol{\omega}$ sont tirées selon une loi gaussienne et que les noyaux étudiés sont à valeurs réelles, nous avons

$$\begin{aligned}\exp(i\boldsymbol{\omega}^T(\mathbf{x}-\mathbf{x}')) &= \cos(\boldsymbol{\omega}^T(\mathbf{x}-\mathbf{x}')) - i\sin(\boldsymbol{\omega}^T(\mathbf{x}-\mathbf{x}')),\\ &= \cos(\boldsymbol{\omega}^T(\mathbf{x}-\mathbf{x}')),\end{aligned}$$

on se focalise donc uniquement sur la partie réelle de notre transformation. Ainsi, on pourra approcher notre noyau comme suit

$$k(\mathbf{x},\mathbf{x}') = \mathbb{E}_{\boldsymbol{\omega}}[\cos(\boldsymbol{\omega}^T(\mathbf{x}-\mathbf{x}'))].$$

On pourra ainsi approcher notre noyau par une somme de cos

$$\begin{aligned}k(\mathbf{x},\mathbf{x}') &= \mathbb{E}_{\boldsymbol{\omega}}[\cos(\boldsymbol{\omega}^T(\mathbf{x}-\mathbf{x}'))],\\ &\simeq \frac{1}{R}\sum_{r=1}^{R}\cos(\boldsymbol{\omega}_r^T(\mathbf{x}-\mathbf{x}')),\\ &= \frac{1}{R}\sum_{r=1}^{R} z_{\boldsymbol{\omega}_r}(\mathbf{x}) z_{\boldsymbol{\omega}_r}(\mathbf{x}'),\\ &= \frac{1}{R} z_{\boldsymbol{\omega}}(\mathbf{x}) z_{\boldsymbol{\omega}}(\mathbf{x}'),\end{aligned}$$

où $z_{\boldsymbol{\omega}}(\mathbf{x}) = \begin{bmatrix} z_{\boldsymbol{\omega}_1}(\mathbf{x}) \\ z_{\boldsymbol{\omega}_2}(\mathbf{x}) \\ \vdots \\ z_{\boldsymbol{\omega}_R}(\mathbf{x}) \end{bmatrix}$ est le vecteur de la représentation de $\mathbf{x}$ dans un espace latent.

Au final, on retiendra que l'on peut approcher notre noyau par une somme de fonctions circulaires comme la fonction *cosinus*.

On va maintenant regarder un algorithme qui permet de faire cela de façon efficace, *i.e.* avec une approche de boosting, on va chercher une bonne combinaison de ces fonctions *cosinus*.

2 Une approche des RFF par Boosting

Dans ce qui précède, nous n'avons pas encore réglé le problème de la vitesse d'apprentissage mais simplement d'approximation d'un noyau.

Nous allons maintenant aller un cran plus loin et tenter d'approcher ce noyau en utilisant nos RFF ainsi que des landmarks.
Dans le TP précédent, les landmarks nous permis de calculer des similarités avec les poids de notre ensemble d'entraînement. Plus précisément, si on considère un ensemble de $l < m$ landmarks, sélectionnés dans notre ensemble d'entraînement, le noyau associé est alors de taille $l \times l$ et notre classifieur se présentait alors comme :

$$f(\mathbf{x}) = \sum_{k=1}^{l} \alpha_k y_k k(\mathbf{x}, \mathbf{x}_k).$$

Pour prédire l'étiquette d'une nouvelle donnée, nous n'avions alors qu'à calculer la similarité du nouveau point aux landmarks que l'on avait choisi aléatoirement.

On se propose maintenant de faire deux choses :

- apprendre des landmarks optimaux de façon itérative,
- approcher notre noyau par une somme de *cos*.

On va faire tout cela dans un contexte de *Gradient Boosting* et l'Algorithme 1 résume la procédure.

Objectifs Comprendre et Implémenter cet algorithme pour le faire fonctionner sur un jeu de données de classification binaire.

2.1 Compréhension de l'algorithme

On se focalise sur les différentes étapes de l'algorithme et on gardera à l'esprit l'objectif de ce TD.

1. En vous aidant de la présentation de l'algorithme de Gradient Boosting, identifier les différentes étapes de la procédure présentée en Algorithme 1.
   (a) Que représentent les deux étapes 3 et 4 de l'algorithme ?
   (b) Que représente l'étape 6 ?

Algorithm 1: Gradient Boosting avec RFF

Inputs : Ensemble $S = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$; Nombre d'itérations T;
Paramètres γ and λ

Output: sign $\Big(H^0(\mathbf{x}) + \sum_{t=1}^T \alpha^t \cos\big(\boldsymbol{\omega}^t \cdot (\mathbf{x}_i - \mathbf{x}^t)\big)\Big)$

1: $H^0 \leftarrow H^0(\mathbf{x}_i) = \frac{1}{2}\ln\frac{\sum_{j=1}^n (1+y_j)}{\sum_{j=1}^n (1-y_j)}$
2: **for** $t = 1, \dots, T$ **do**
3: $\quad \forall i = 1, \dots, n, \quad w_i = \exp(-y_i H^{t-1}(\mathbf{x}_i))$
4: $\quad \forall i = 1, \dots, n, \quad \tilde{y}_i = y_i w_i$
5: $\quad$ Tirer $\boldsymbol{\omega} \sim \mathcal{N}(0, 2\gamma)^d$
6: $\quad \mathbf{x}^t = \arg\min_{\mathbf{x}^t \in \mathbb{R}^d} \frac{1}{n}\sum_{i=1}^n \exp\Big(-\tilde{y}_i \cos\Big(\boldsymbol{\omega} \cdot (\mathbf{x}_i - \mathbf{x}^t)\Big)\Big)$
7: $\quad \boldsymbol{\omega}^t = \arg\min_{\boldsymbol{\omega} \in d} \lambda\|\boldsymbol{\omega}\|_2^2 + \frac{1}{n}\sum_{i=1}^n \exp\Big(-\tilde{y}_i \cos\Big(\boldsymbol{\omega} \cdot (\mathbf{x}_i - \mathbf{x}^t)\Big)\Big)\Big).$
8: $\quad \alpha^t = \frac{1}{2}\ln\frac{\sum_{i=1}^n \Big(1+y_i \cos\big(\boldsymbol{\omega}^t \cdot (\mathbf{x}_i - \mathbf{x}^t)\big)\Big) w_i}{\sum_{i=1}^n \Big(1-y_i \cos\big(\boldsymbol{\omega}^t \cdot (\mathbf{x}_i - \mathbf{x}^t)\big)\Big) w_i}$
9: $\quad \forall i = 1, \dots, n, \; H^t(\mathbf{x}_i) = H^{t-1}(\mathbf{x}_i) + \alpha^t \cos\big(\boldsymbol{\omega}^t \cdot (\mathbf{x}_i - \mathbf{x}^t)\big)$
10: **end for**=0

(c) Même question pour l'étape 8.

2. En regardant de plus le calcul effectué à l'étape 2, identifier la loss que l'on cherche à minimiser.

3. Est-il possible de modifier l'étape 6 de façon à ce que le landmark appris $\mathbf{x}^t$ ne se trouve plus dans un espace de dimension d mais dans un espace de dimension 1 ? Si oui, expliquer comment.

4. En utilisant le fait que

$$\alpha^t = \arg\min_{\alpha} \sum_{i=1}^n \exp\big[-y_i(H^{t-1}(\mathbf{x}_i) + \alpha h^t(\mathbf{x}_i))\big] = \arg\min_{\alpha} \sum_{i=1}^n w_i \exp\big[-y_i \alpha h^t(\mathbf{x}_i)\big]$$

et une inégalité de convexité sur la fonction exponentielle, montrer la relation obtenue à l'étape 8 de l'algorithme.

2.2 Implémentation

On va maintenant chercher à implémenter cet algorithme. Vous devrez alors créer une classe sous Python (ou des fonctions) qui permettent d'apprendre le modèle et de faire la prédiction.

On prendra également garde au fait que notre modèle dépend d'un hyper-paramètre λ que l'on devra chercher à tuner idéalement, ici on va simplement lui donner une valeur fixe au début, par exemple $\lambda = 1$.

Pour l'implémentation, vous devrez résoudre des problèmes d'optimisation, ce qui implique d'employer certains solveurs :

```
from scipy import optimize
from scipy.optimize import fminbound
```

En particulier, on utilisera la fonction `optimize.fmin_l_bfgs_b` de librairie `scipy`.

Enfin, pour l'implémentation, vous aurez besoin de calculer un gradient, de stocker toutes les valeurs des paramètres du modèle.

References

[Mercer, 1909] Mercer, J. (1909). Functions of positive and negative type, and their connection with the theory of integral equations. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 209(441-458):415–446.

[Rahimi and Recht, 2007] Rahimi, A. and Recht, B. (2007). Random features for large-scale kernel machines. *Advances in neural information processing systems*, 20.

[Rudin, 2017] Rudin, W. (2017). *Fourier analysis on groups*. Courier Dover Publications.