Rattrapage OS 202 2024

Parallélisation de prédiction

Le sujet tourne autour de quelques algorithmes simples issus de la science des données. Les trois parties sont indépendantes.

Quelques définitions et notations issues de la science des données

Dans les diverses données traitées, on peut distinguer deux types de variables :

variable explicative (notée pour la ième variable explicative X_i)

- Variable dont les valeurs serviront de paramètres pour prédire la variable d'intérêt
- On notera χ_i l'ensemble des valeurs pouvant être prises par la variable X_i

et

variable d'intérêt (notée en général Y):

- Variable dont on veut pouvoir prédire les valeurs en fonction d'une ou plusieurs variables explicatives
- On notera $\mathcal{Y}$ l'ensemble des valeurs pouvant être prises par Y.

Pour entraîner nos algorithmes, on se basera sur un

ensemble de données d'entraînement

- consistant en un ensemble de K_1 échantillons où chaque échantillon est consistué
  - d'une valeur $x_i^{(k)} \in \chi_i$ pour chaque variable explicative X_i
  - d'une valeur connue $y^{(k)} \in \mathcal{Y}$ pour la variable d'intérêt Y.

qui servira pour construire un modèle statistique permettant de prédire pour un ensemble de variables explicatives (par forcément *toutes* les variables explicatives) la valeur que devrait prendre la variable d'intérêt.

Enfin, pour évaluer la performance de notre modèle prédictif, on utilisera un

benchmark

- constitué de deux ensembles de données :

- un ensemble de K_2 échantillons $x_i^{(k)} \in \chi_i$ pour chaque variable explicative X_i qu'on utilisera pour essayer de prédire pour chaque échantillon la valeur $y'^{(k)} \in \mathcal{Y}$ que devrait prendre la variable Y
- Un ensemble de K_2 valeurs $y^{(k)} \in \mathcal{Y}$ donnant la valeur qu'on aurait dû trouver pour chaque échantillon donné dans l'ensemble précédent.

qui nous permettra de tester notre modèle prédictif en regardant le pourcentage de bonnes prédictions (quand $y'^{(k)} == y^{(k)}$ pour $k \in K_2$).

Note : Dans notre cas, on va partitionner notre ensemble de données (avec un partitionnement tiré au hasard) en deux sous-ensembles servant l'un pour l'entraînement et l'autre pour le benchmark.

I. Ce qu'il faudra restituer

- Les fichiers pythons où on a parallélisé du code,
- Un fichier texte, markdown ou pdf contenant vos analyses et justifications de la parallélisation du code.

Le tout devra être envoyé sur les deux adresses e-mail suivantes :

- `xjuvigny@gmail.com`
- `juvigny@onera.fr`

IMPORTANT : **Ne pas envoyer de fichiers compressés mais directement vos fichiers pythons et votre document (pdf, markdown, texte, ...)**

Vous pourrez quitter la salle dès réception effective de votre mail et après avoir signé la feuille d'émargement.

II. Configuration de votre ordinateur

- Donner le nombre de coeurs physiques et logiques possédés par votre ordinateur.
- Quelle est la différence entre coeurs physiques et coeurs logiques ?
- Donner la taille de vos différents niveaux de mémoire cache.

III. Données exploitées dans le cadre de l'examen

Les données exploitées sont tirées d'une base de donnée *Open source* fournie par le réseau de transport électrique (RTE) et météo france, qui donne sur plusieurs années, par demi-heure, le mode de production de l'électricité (variable d'intérêt) :

- **Décarbonnée** Très peu de dioxyde de carbonne émis par les centrales (nucléaires, éoliens, solaire)
- **Mixte** Du dioxyde de carbonne est émis de façon modérée par les centrales (mises en marche des centrales à charbon et à gaz)

- **Carbonnée** Beaucoup de dioxyde de carbonne émis car utilisation intensive des centrales à charbon et à gaz pour pouvoir répondre à la demande électrique.

ainsi que divers autres données associées (variables explicatives):

- *Date et heure* : Année, mois et jour et heure (par demi-heure)
- *Position dans l'année* : variable qui croît linéairement allant de 0 le premier janvier à 1 au 31 décembre de la même année.
- *Mois* : Le mois de l'année
- *Demi heure* : La demi-heure considérée dans la journée
- *Jour* : Le jour de la semaine (lundi, mardi, etc.)
- *Jour férié* : Variable booléenne pour savoir si c'est un jour férié (vrai) ou non (faux)
- *Type de jour férié* : permet de différencier les jours fériés : premier janvier, paques, premier mai, ascension, 8 mai, etc. et non férié...
- *Vacances zone A* : variable booléenne vrai si le jour est un jour de vacances scolaires en zone A
- *Vacances zone B* : variable booléenne vrai si le jour est un jour de vacances scolaires en zone B
- *Vacances zone C* : variable booléenne vrai si le jour est un jour de vacances scolaires en zone C
- *Température* : La température en °C
- *Nébulisoté* : Pourcentage du ciel visible couvert par des nuages
- *Humidité* : Pourcentage d'humidité
- *Précipitation* : Précipitation en mm

IV. Classificateur de bayes

Le code **séquentiel** correspondant se trouve dans le fichier python `bayes.py`

On note Y la variable d'intérêt qu'on veut prédire (une donnée $y \in \mathcal{Y}$ vaut donc soit **Carbonné**, **Mixte** ou **Décarbonné**)

Le principe du classificateur de bayes est le suivant :

- On choisit n variables explicatives $X_1, \cdots, X_n$ où $n \in \{1, 2, \cdots\}$
- Pour chaque combinaison de données $x_i \in X_i$ et $y \in Y$, on calcule la fréquence d'apparition $f_{x_1,\cdots,x_n,y}$ où $x_1, \cdots, x_n$ et y apparaissent simultanément (c'est à dire sur la même "ligne" du tableau de donnée). On obtient ainsi une loi jointe.
- Pour un jeu de donnée $x'_1, \cdots, x'_n$, on cherche à prédire une valeur y' correspondant au mode de production le plus probable. Pour cela on cherche y' tel que $f_{x'_1,\cdots,x'_n,y'} \geq \max_{y \in Y} f_{x'_1,\cdots,x'_n,y}$

Question IV.1

Quelle est la complexité algorithmique et de stockage pour calculer une loi jointe sachant que chaque variable X_i peut prendre n_i valeurs et Y peut prendre trois valeurs ?

IV.2 Classificateur de Bayes naïf

Afin de diminuer drastiquement la complexité algorithmique et de stockage, on fait l'hypothèse approximative que les variables explicatives X_i sont indépendantes relativement à la variable d'intérêt Y. On a donc :

$$f_{x_1,\cdots,x_n,y} = \prod_{i=1}^{n} f_{x_i y}$$

IV.3 A Faire

- Paralléliser les diverses classifications de Bayes mises en oeuvre dans le fichier `bayes.py` et valider votre code. Pourquoi est-ce difficile d'obtenir exactement le même résultat qu'en séquentiel ?
- Mesurer les temps avec un nombre divers de processus et établir une courbe d'accélération que vous commenterez (dans la mesure des capacités de votre machine).

V. Arbre de décision

Afin d'améliorer nos prédictions, on va maintenant utiliser un *arbre de décision*. Un arbre de décision est un arbre binaire où chaque noeud correspond au test sur une variable X_i où on passe sur le fils gauche ou le fils droit selon une valeur seuil stockée dans le noeud de l'arbre.

La construction de l'arbre se base sur l'*indice d'impureté de Gini* qui calcule à partir d'un échantillon de données (contenant les variables d'intérêt) la *probabilité de choisir la mauvaise prédiction* (en supposant que la bonne prédiction est au moins contenue dans un échantillon).

Exemple Soit $y_i \in \mathcal{Y}$ (Y étant la variable d'intérêt) et m le nombre d'échantillons. Alors si $y_i = y_j$ pour $i, j \in \{1, \cdots, m\}$ l'indice d'impureté de gini est nul (puisqu'on est sûr de faire la bonne prédiction dans ce cas)

L'algorithme de construction de l'arbre est donc l'algorithme récursif suivant :

- Pour un noeud donné, trouver la variable X_i et une valeur de seuil η_i permettant d'effectuer une partition en deux fils (gauche contenant les $x_i < \eta_i$ et droit contenant les $x_i \geq \eta_i$) qui permettent de minimiser la moyenne pondérée des valeurs d'impureté de Gini des deux fils (on teste pour cela toutes les variables et les valeurs de seuils possibles !)

- Pour le fils gauche et le fils droit, si l'indice de Gini du fils est non nul (le noeud contient plusieurs prédictions possibles) ou si le nombre d'échantillons contenu dans le fils est supérieur à une valeur $n_{\min}$ donnée, alors on appelle récursivement l'algorithme de construction de l'arbre pour ce fils.

Pour plus de détails, l'algorithme est mis en oeuvre et testé dans le fichier python `decision_tree.py`.

Pour effectuer nos prédictions, on n'a plus pour chaque échantillon qu'à parcourir l'arbre en fonction des variables et des seuils sélectionnés jusqu'à arriver à une feuille de l'arbre qui nous donnera une prédiction.

V.1 Question sur le parallélisme de l'algorithme

- Expliquer comment on pourrait au mieux paralléliser l'algorithme de construction de l'arbre.
- Pourquoi l'intérêt est limité de paralléliser un tel algorithme ?

Note : Il n'est pas demandé de paralléliser le code python correspondant !

VI. Forêt d'arbres

La construction d'un arbre de décision à partir d'un grand nombre d'échantillon peut se révéler très coûteux en terme de CPU. L'idée d'une forêt d'arbres est de construire p arbres de décisions à partir d'un sous-échantillonage aléatoire donné pour chaque arbre.

Pour prédire, on calcule la prédiction de chaque arbre puis on prend la prédiction majoritaire trouvée parmi les p arbres.

Le code python correspondant (utilisant le fichier `decision_tree.py`) est le fichier `random_forest.py`

VI.1 A faire

- Paralléliser le code utilisant une forêt d'arbre
- Faire plusieurs tests en faisant varier le nombre de processus
- Calculer l'accélération obtenue.