R avancé: 2021-2022
masedki.github.io

TP1: Profilage de code

L'ensemble des séances se fera sur le serveur auquel vous pouvez vous connecter par l'url suivante

http://cesp-oxygene3.vjf.inserm.fr:8787

Exercise 1 (Temps d'exécution)

On travaille avec un `data.frame` ayant 151 colonnes. La première colonne contient un identifiant des individus (ID) et les autres 150 colonnes contiennent des valeurs numériques. On souhaite retrancher à chaque colonne sa moyenne (pour obtenir un tableau centré en colonne).

1. Écrire un script qui crée un `data.frame` nommé `data_df` ayant $n = 10^5$ observations et 151 colonnes. La première colonne contient les labels des observations de sorte que l'observation i possède le label "g.i". Les autres colonnes sont générées par des réalisations indépendantes de $\mathcal{N}(0, 1)$.
2. Écrire un script qui effectue les taches suivantes:
   - Copier la variable `data_df` dans une autre variable `datanew_df`.
   - Calculer la moyenne par colonne de l'objet `data_df` (pour les 150 variables continues) en utilisant la fonction `apply`.
   - Utiliser une boucle `for` pour parcourir chaque colonne numérique de l'objet `datanew_df` afin d'en retirer sa moyenne et que `datanew_df` soit centré en colonnes.
   - Effectuer le profilage de code par la fonction `profvis` en sauvegardant le resultat dans un fichier html. Quelle est la partie à améliorer?
3. Optimisation du calcul de la moyenne par colonnes
   - Proposer quatre façons d'obtenir la moyenne par colonnes (avec `apply`, `colMeans`, `lapply`, `vapply`).
   - Utiliser le profilage de code, en sauvegardant le resultat, (macro et micro) pour sélectionner la façon la plus rapide d'obtenir la moyenne par colonnes. Expliquer pourquoi cette approche est plus rapide.
4. Profilage de code 2
   - Reprendre le code original est remplacer l'appel à la fonction `apply` par `vapply`.
   - Effectuer le profilage de code par la fonction `profvis`, en sauvegardant le resultat. Quelle est la partie à améliorer?
5. Optimisation de la fonction de normalisation
   - Proposer une meilleure façon de soustraire la moyenne par colonne (utiliser la fonction `lapply`).
   - Vérifier avec le profilage de code (uniquement macro), en sauvegardant le resultat, que votre solution est plus efficace.

Exercise 2 (Memoire)

Il peut être difficile de voir les causes de lenteur d'un code, mais on peut voir leurs effets de bords et particulièrement ceux causés par des grandes allocations de mémoire.

1. Générer un `data.frame` qui contient une colonne composée de 3×10^4 observations issues d'une distribution uniforme. On veut calculer la somme cumulée de ces données sans utiliser la fonction

`cumsum()`. Écrire un script qui calcule la somme cumulée et qui la stocke dans une seconde colonne du `data.frame`. Utiliser une boucle `for`.

2. Faire le profilage de code.

3. Le profilage nous indique que le temps de calcul est principalement causé par $ et $< −. On pourrait sûrement réduire le temps de calcul en évitant ces appels. Pour cela, à la place de travailler sur les colonnes du `data.frame`, on peut travailler sur un vecteur temporaire.

   Il s'avère que écrire une fonction qui prend un vecteur comme argument et retourne un vecteur n'est pas seulement pratique. Cela permet de créer des variables temporaires qui évitent les appels de type $ et $< − dans une boucle. Cette approche est implémentée dans la fonction `csum` définie à la fin de l'exercice. Cette approche est-elle plus rapide?

4. Tester la fonction `csum2` qui est identique à la fonction `csum` sauf qu'elle fait une pré-allocation de mémoire pour le vecteur `sum`. Ici considérer $n = 10^6$.

```
  csum <- function(x_num) {
    if (length(x_num) < 2)
      return(x_num)

    out_num <- x_num[1]
    for (i in 2:length(x_num))
      out_num[i] <- out_num[i-1] + x_num[i]

    out_num
  }
  data_df$sum <- csum(data_df$value)
```

Exercise 3 (Suppression des valeurs manquantes)

On considère le problème de suppression des lignes qui contiennent des valeurs manquantes dans un jeu de données. Voici quatre façons de supprimer les valeurs manquantes.

```
funAgg <- function(x) {
  res <- NULL
  n <- nrow(x)
  for (i in 1:n) {
    if (!any(is.na(x[i,]))) res <- rbind(res, x[i,])
  }
  res
}

funLoop <- function(x) {
  res <- x
  n <- nrow(x)
  k <- 1
  for (i in 1:n) {
    if (!any(is.na(x[i,]))) {
      res[k, ] <- x[i,]
      k <- k + 1
    }
  }
  res[1:(k-1), ]
}
```

```r
funApply <- function(x) {
  drop <- apply(is.na(x), 1, any)
  x[!drop, ]
}

funOmit <- function(x) {
  drop <- F
  n <- ncol(x)
  for (i in 1:n)
    drop <- drop | is.na(x[, i])
  x[!drop, ]
}
```

1. Générer une matrice de taille $10^6 \times 20$ composée de réalisations indépendantes de $\mathcal{N}(0, 1)$. Remplacer toutes les valeurs supérieures à deux par `NA`.
2. Programmer les 4 fonctions précédantes.
3. Utiliser le profilage de code pour détecter la meilleur façon d'enlever les valeurs manquantes.