Modélisation et Outils Mathématiques
– TP de Statistique –
Application au problème du voyageur de commerce

4IF, INSA de Lyon, 2019/2020
Irène Gannaz

INSA-Lyon, Département Informatique
version 1.00, 2020 – rédigé avec LaTeX

Environnements. Ce TP a été construit pour les environnements Linux et Windows mais je ne peux fournir les éléments nécessaires pour Mac (cela dépend de problèmes posés par Mac aux développeurs `R` et est indépendant de ma volonté).

Introduction

Les objectifs de ce TP sont les suivants :

- Proposer une visualisation pour comparer des algorithmes,
- Mettre en œuvre des procédures de tests statistiques,
- Introduire la notion de tests multiples,
- Ajuster un modèle de régression linéaire,
- Analyser la pertinence d'un modèle de régression (validité de l'ajustement et des hypothèses),
- Réaliser une sélection de variables.

L'ensemble des codes durant ces deux séances sera développé avec le **logiciel `R`.** A la fin de la séance, vous devrez rendre via Moodle à l'adresse `http://moodle.insa-lyon.fr/` :

- un compte-rendu rédigé, au format .pdf, qui contiendra les résultats **commentés**, les graphiques et les tableaux ;
- les codes que vous avez développés pendant les séances permettant de retrouver les résultats du compte-rendu.

Votre compte-rendu devra être mis dans une archive avec une extension .zip ou .tar.gz. N'oubliez pas les noms sur le compte-rendu !

Il est fortement conseillé de rédiger votre compte-rendu avec `Rmarkdown`. Des fiches d'introduction au logiciel `R` et à `Rmarkdown` sont disponibles sous Moodle. Est également fourni sur Moodle un fichier `TPstat_CR.Rmd` qui donne un schéma de compte-rendu en `Rmarkdown`.

Chaque tâche devant être réalisée est précédée d'un • dans le sujet.

Packages à installer

- Installer les paquets :
  - `maps`, `sp` qui permettent l'affichage de cartes,
  - `Rcpp`, qui permet d'intégrer des codes en C++ dans R, et est utilisé dans certaines fonctions utilisées ici,
  - `TSP` qui proposent des calcul de parcours hamiltoniens sur des graphes,
  - `microbenchmark` qui fournit des fonctions de comparaison de codes,
  - `multcomp` qui propose des procédures de tests multiples (utile dans la comparaison de codes par `microbenchmark`),

  tous disponibles sur le CRAN, site de dépôts de R. Rappelons que les packages s'installent à l'aide de la fonction `install.packages` puis que la fonction `library` permet de faire appel aux packages dans la session.

  Attention : Pour ceux qui sont sur leur machine personnelle Windows, il est nécessaire d'avoir `Rtools` pour installer `Rcpp`. `Rtools` n'est pas un package et doit être installé séparément (cf lien sur moodle).
- Récupérer les documents disponibles sur `Moodle`. Notamment les fichiers `DonneesGPSvilles.csv` et `DonneesGraphes.csv`, et le paquet `TSPpackage`. Installer le package `TSPpackage`.

Remarque sur l'aléatoire

Chaque exécution de votre code donnera des résultats différents si vous ne fixez pas de graine. Comme vous l'avez vu dans le TP de probabilités en 3IF (sauf si vous ne l'avez pas fait), une graine est une initialisation dans la génération d'aléatoire. La fixer permet de s'assurer que ce qui sera généré sera identique à chaque exécution du programme.

Lorsque vous allez faire vos commentaires, vous allez vous baser sur une réalisation donnée. **Afin que les commentaires restent valables, il est conseillé de fixer la graine** en début de votre `.Rmd` comme suit :

```
set.seed(287)
```

avec tout nombre de votre choix à la place de 287.

Comme nous allons étudier des temps de calcul, ceux-ci peuvent varier entre deux lancers de votre `.Rmd`. Ceci n'est a priori pas critique mais vérifier tout de même que les commentaires sont cohérents avant de rendre le compte-rendu (quitte à relancer sinon).

0 Visualisation de chemins

Cette section a pour but de visualiser des chemins sur un exemple et de donner quelques lignes de commande `R` pouvant être utiles dans la suite. Aucun code ne vous est demandé.

Récupération des coorddonnées

Le fichier `DonneesGPSvilles.csv` contient les coordonnées des 22 villes françaises (les chefs-lieux des régions de métropole avant la réforme territoriale de 2015).

Nous pouvons récupérer les données :

```
villes <- read.csv('DonneesGPSvilles.csv',header=TRUE,dec='.',sep=';',quote="\"")
```

Une synthèse du contenu peut être obtenue en appliquant la fonction `str` :

```
str(villes)
```

Nous pouvons ensuite extraire une matrice `coord` dont chaque ligne correspond à une ville, telle que la première colonne donne la latitude et la seconde la longitude. On obtient la matrice `dist` des distances à vol d'oiseau entre les villes en appliquant la fonction `distanceGPS` à la matrice précédente.

```
coord <- cbind(villes$longitude,villes$latitude)
dist <- distanceGPS(coord)
```

Calcul de chemins

Le chemin par plus proches voisins à l'aide de la fonction est obtenu par la fonction `TSPnearest`. Cette fonction retourne une liste contenant le chemin obtenu ainsi que sa longueur.

La fonction `TSPsolve(couts,methode)` permet elle de calculer la longueur du chemin hamiltonien sur un graphe dont les coûts entre les sommets sont donnés par une matrice `couts` par une méthode `methode` donnée. En particulier la méthode `'branch'` permet d'appliquer l'algorithme Branch&Bound que vous avez implémenté en TP de AAIA.

En appliquant `TSPnearest` puis `TSPsolve` avec la méthode `'branch'` sur les distances entre les villes, on constate qu'en effet la longueur du chemin obtenu par la procédure des plus proches voisins est plus importante.

Visualisation

Le chemin optimal est

```
pathOpt <- c(1,8,9,4,21,13,7,10,3,17,16,20,6,19,15,18,11,5,22,14,12,2)
```

où les villes sont numérotées dans l'ordre du fichier initial.

La fonction `plotTrace` permet enfin de représenter sur une carte le chemin par plus proches voisins ou le chemin optimal.

1 Comparaison d'algorithmes

Nous souhaitons comparer la méthode Branch&Bound codée lors du TP de AAIA avec des méthodes approchées. Nous allons considérer trois méthodes proposées par le paquet `TSP` de `R`. Ces méthodes sont données respectivement par les arguments `'repetitive_nn'`, `'nearest_insertion'` et `'two_opt'` dans la fonction `TSPsolve`. Une description peut être trouvée dans [4]. La méthode des plus proches voisins (également non optimale) est donnée par `'nearest'` et la méthode Branch&Bound (retournant un chemin de longueur minimale) par `'branch'`. Ces deux méthodes sont issues des TP de AAIA.

Les mesures seront réalisées sur des graphes de $n = 10$ sommets dont les coordonnées cartésiennes sont des lois uniformes sur [0,1]. Les graphes seront générés comme suit :

```
sommets <- data.frame(x = runif(n), y = runif(n))
  couts <- distance(sommets)
```

`sommets` contient la liste des coordonnées cartésiennes des sommets du graphe, `sommets$x` et `sommets$y` donnant respectivement les abscisses et les ordonnées des points. `couts` est la matrice des distances cartésiennes entre les sommets, `couts[i,j]` donnant la distance entre les sommets i et j.

1.1 Longueur des chemins

- Récupérer les longueurs des chemins hamiltoniens donnés par les 5 méthodes sur 50 réalisations des graphes aléatoires décrites ci-dessus. Représenter côte-à-côte les boîte à moustache (boxplots) de ces longueurs, pour chaque méthode. On pourra utiliser la fonction `boxplot`. Commenter.
- Soit m_{nn} et m_b les espérances respectives des algorithmes des plus proches voisins et de Branch&Bound. Réaliser le test de (H_0) $m_{nn} - m_b \leq 0$ contre (H_1) $m_{nn} - m_b > 0$. Conclure.

  Nous souhaitons comparer 2 à 2 les longueurs moyennes obtenues par les algorithmes. Pour cela nous voulons tester

  $$(H_0)\ m_i = m_j \quad \text{contre} \quad (H_1)\ m_i \neq m_j, \quad i \neq j,$$

  avec $(m_i)_{i=1,\dots,5}$ l'espérance de la $i^{\text{ème}}$ méthode appliquée.

  Nous devons donc réaliser 10 tests simultanés. Si chaque test est réalisé avec une probabilité α de se tromper, sur 10 tests la probabilité de se tromper ($i.e$ d'avoir au moins un faux positif) sera de $p = 1 - (1-\alpha)^{10}$. Afin d'avoir une probabilité α de nous tromper sur les 10 tests, nous réalisons chaque test avec un rique $\alpha/10$. En effet la probabilité devient alors $p = 1 - (1-\alpha/10)^{10} \leq \alpha$.

  La procédure décrite ci-dessus s'appelle **la correction multiple de Bonferroni**. Elle consiste lorsqu'on réalise K tests statistique à prendre un risque α/K pour chaque test afin de garantir un rique α au final.

  On pourra lire par exemple le chapitre 8 de [3] pour plus de précisions sur les tests multiples.

  Soient `results` un vecteur avec toutes les longueurs obtenues et `methods` un vecteur de même taille, tel que `methods[i]` donne le nom de l'algorithme utilisé pour obetnir `results[i]`. Alors

  ```
  pairwise.t.test(results,methods,adjust.method='bonferroni')
  ```

  va réaliser les tests souhaités avec la procédure de Bonferroni. Plus précisément, `pairwise.t.test` retourne une p-valeur modifiée, qui peut s'interpréter directement comme une p-valeur usuelle, au vu du risque global.
- Réaliser les tests. Conclure.

  Remarque : Une approche plus rigoureuse pour considérer ce genre de problèmes est le modèle de l'ANOVA (voir par exemple [2], chapitre 16). Il s'agit d'une régression linéaire de la variable expliquée (ici le temps) par rapport à des variables explicatives qualitatives (ici les méthodes appliquées). Des procédures de tests multiples spécifiques à ce contexte (et donc plus puissantes) peuvent alors être proposées, par exemple Tukey-HSD.

1.2 Temps de calcul

Un *microbenchmark* est une comparaison sur un jeu de simulation des temps d'exécution de deux fonctions. En `R`, la fonction `microbenchmark` issue de la fonction homonyme permet de réaliser facilement cette comparaison. Voici ci-après un exemple d'utilisation :

```
microbenchmark(sqrt(x),x^0.5, times=100, setup={x <- runif(1)})
```

La ligne ci-dessus compare les temps des fonctions `sqrt(x)` et `x^0.5` sur 100 exécutions, avec `x<-runif(1)` effectué avant chaque exécution.

Avec le package `multcomp`, les temps moyens d'exécutions des fonctions sont comparées 2 à 2 ; si m_1 et m_2 sont les espérances des temps des algorithmes 1 et 2, le test appliqué est (H_0) $m_1 = m_2$ contre (H_1) $m_1 \neq m_2$ avec un risque $\alpha = 5\%$. La dernière colonne de `microbenchmark` range les fonctions selon leur temps d'exécution : a pour les algorithmes les plus rapides, b ensuite, puis c, etc. Une même lettre est attribuée aux algorithmes tel que (H_0) n'est pas rejetée. Inversement 2 lettres différentes signifient que (H_0) est rejetée.

Ces tests sont réalisés en respectant le principe des tests multiples, décrit plus haut.

- Appliquer la fonction `microbenchmark` pour comparer les temps des 5 méthodes étudiées sur 20 graphes de $n = 10$ sommets dont les coordonnées sont des lois uniformes sur [0,1]. Conclure.

2 Etude de la complexité de l'algorithme Branch&Bound

Nous nous intéressons maintenant exclusivement à l'algorithme Branch&Bound. Nous souhaitons étudier sa complexité, en fonction du graphe sur lequel il est appliqué. Pour cela nous allons mettre en œuvre des modèles de régressions linéaires gaussiens. Rappelons que des exemples d'ajustement, d'analyse de pertinence et d'étude de la validité des hypothèses sont fournis dans le polycopié de cours.

2.1 Comportement par rapport au nombre de sommets : premier modèle

Dans un premier temps, nous considérons les graphes générés auparavant, c'est-à-dire,

```
sommets <- data.frame(x = runif(n), y = runif(n))
  couts <- distance(sommets)
```

Nous construisons un modèle de régression linéaire simple du temps d'exécution de Branch&Bound en fonction du nombre de sommets n.

Introduisons

```
seqn <- seq(4,20,1)
```

- Construire `temps`, matrice telle que la i$^{\text{ème}}$ ligne soit obtenue par

```
microbenchmark(TSPsolve(couts, method = 'branch'),
               times = 10,
               setup = { n <- seqn[i]
                         couts <- distance(cbind(x = runif(n), y = runif(n)))}
              )$time
```

- Représenter `temps` en fonction de `n`, puis `log(temps)^2` en fonction de `n` :

```
par(mfrow=c(1,2)) # 2 graphiques sur 1 ligne
matplot(seqn, temps, xlab='n', ylab='temps')
matplot(seqn, log(temps)^2, xlab='n', ylab=expression(log(temps)^2))
```

- Ajuster le modèle linéaire de `log(temps)^2` en fonction de `n` puis récupérer les principales statistiques :

```
vect_temps <- log(as.vector(temps))^2
vect_dim <- rep(seqn,times=10)
temps.lm <- lm(vect_temps ~ vect_dim)
summary(temps.lm)
```

  Commenter.

- Réaliser l'étude graphique des hypothèses sur les résidus :

```
par(mfrow=c(2,2)) # 4 graphiques, sur 2 lignes et 2 colonnes
plot(temps.lm)
```

  Conclure.

  On souhaite vérifier l'adéquation des résidus à une loi normale, c'est-à-dire tester (H_0) « les résidus suivent une loi normale » contre (H_1) « les résidus ne suivent pas une loi normale ». Nous avons vu en cours que ceci peut se faire avec un test du χ^2, mais nous allons utiliser ici le test de Shapiro-Wilk qui est plus puissant (car spécifique à la loi normale).

- Réaliser le test de Shapiro-Wilk :

```
shapiro.test(residuals(modele))
```

  et conclure quant à la validité du modèle.

2.2 Comportement par rapport au nombre de sommets : étude du comportement moyen

Nous allons maintenant essayer de construire un modèle de régression linéaire simple du temps moyen d'exécution de l'algorithme Branch&Bound en fonction du nombre de sommets.

Introduisons

```
temps.moy <- rowMeans(temps)
```

- Réaliser l'ajustement du modèle de régression linéaire simple gaussien de `log(temps.moy)^2` en fonction de `seqn`. Récupérer les principales statistiques. Faire l'étude des résidus. Conclure quant à la validité du modèle.

2.3 Comportement par rapport à la structure du graphe

A l'aide du paquet `igraph` de `R`, nous avons généré 73 graphes avec différentes structure : complets, *Small-World*, *Stochastic Block models*...Chaque graphe généré a un chemin hamiltonien, *i.e.* il existe un chemin passant une unique fois par tous les sommets du graphes. Nous avons récupéré le temps moyen d'exécution de l'algorithme sur 20 graphes générés pour chaque structure considérée.

Si vous voulez des précisions sur les structures de graphes, référez-vous à [1], ou pour aller plus loin à [5].

- Récupérer le jeu de données 'DonneesTSP.csv' dans un data.frame nommé `data.graph`.

  Il est constitué de la mesure du temps moyen de l'algorithme Branch&Bound sur 20 exécutions sur 73 graphes similaires, noté `tps`. Il fournit également 6 moyennes de mesures réalisées sur ces familles de graphes :

  - `dim`, nombre de sommets,
  - `mean.long`, longueur moyenne entre deux sommets sur les chemins obtenus,
  - `mean.dist, sd.dist`, moyenne et écart-type des coûts des arêtes,
  - `mean.deg, sd.deg`, moyenne et écart-type des degrés des sommets,
  - `diameter`, le diamètre du graphe (longueur maximale du plus court chemin entre les sommets 2 à 2).

- Construire le modèle de régression de `log(tps)` par rapport à `sqrt(dim)` et toutes les autres variables de `data.graph`. Quelles sont les variables explicatives non pertinentes dans le modèle ?

  Pour faire la régression de `y` par rapport à toutes la variables contenues dans un jeu de données `dataset`, la commande est

  ```
  lm(y ~ ., data = dataset)
  ```

  Afin de choisir quelles variables sont pertinentes dans le modèle, on met ensuite en place une **sélection de variables**. L'idée est que plus les écarts au modèle sont petits, plus le modèle est pertinent. Les écarts au modèle sont mesurés par la somme des carrés des résidus, $\Delta = \sum_{i=1}^{nobs}(y_i - \widehat{y}_i)^2$ si la régression a lieu sur *nobs* mesures et $\widehat{y}$ est l'estimation de y donnée par la régression.

  Le problème est que plus il y a de variables, plus Δ sera faible. On pondère donc par le nombre de variables explicatives p du modèle, afin de trouver un compromis entre l'écart au modèle et sa complexité. Un critère usuel est AIC,

  $$AIC = 2(p-1) - 2\sum_{i=1}^{nobs}(y_i - \widehat{y}_i)^2.$$

  L'objectif est de maximiser l'AIC.

  Ceci est fait par la fonction `step` de `R`, qui à partir du modèle complet va enlever les variables une à une pour maximiser l'AIC. Si `modele.complet` est l'ajustement par rapport à toutes les variables explicatives obtenu à l'aide de la fonction `lm`, alors `step(modele.complet)` va réaliser une sélection de variable par AIC.

  Voir [2] pour plus de détails.

- Mettre en œuvre la sélection de variables. Quelles variables ont été exclues du modèle ? Toutes les variables conservées sont-elles pertinentes individuellement pour le modèle ? Au vu du test de Fisher, le modèle est-il pertinent (dans sa globalité) ?

- Faire l'étude des résidus. Le modèle est-il validé ?

Références

[1] Gabor Csardi and Tamas Nepusz. The igraph software package for complex network research. *InterJournal*, Complex Systems :1695, 2006.

[2] Julian J Faraway. *Linear models with R*. Chapman and Hall/CRC, 2016.

[3] Christophe Giraud. *Introduction to high-dimensional statistics*. Chapman and Hall/CRC, 2014.

[4] Michael Hahsler and Kurt Hornik. TSP – Infrastructure for the traveling salesperson problem. *Journal of Statistical Software*, 23(2) :1–21, December 2007.

[5] Eric D Kolaczyk and Gábor Csárdi. *Statistical analysis of network data with R*, volume 65. Springer, 2014.