

Étude de cas:

Analyse de marché du transport aérien canadien avec R

Atelier d'introduction à R

BEAUCHEMIN, DAVID

CABRAL CRUZ, SAMUEL

GOULET, VINCENT

Dans le cadre du colloque R à Québec

25 mai 2017

Table des matières

Table des figures	2
Liste des codes sources	3
Liste des tableaux	4
Préface	5
Introduction	6
Étude de cas	9
2.1 Extraction, traitement, visualisation et analyse des données . . .	9
2.1.1 Extraction	9
2.1.2 Traitement	11
2.1.3 Visualisation et analyse des données	19
2.2 Création de fonctions utilitaires	25
2.3 Conception de graphiques en R	32
2.4 Outils d'analyse statistique en R	38
2.5 Ajustement de distributions statistiques sur données empiriques .	45
2.6 Simulation et analyse de rentabilité	45
Conclusion	46
A Code source du projet	48

Table des figures

1.1	Interface de l'outil OpenFlights	6
2.1	Extrait du fichier airports.dat	10
2.2	Structure des fichiers de données géospatiales	15
2.3	Exemple de carte géographique produite avec <i>ggmap</i>	21
2.4	Exemple de carte géographique produite avec <i>leaflet</i>	22
2.5	Densité de la population canadienne	25
2.6	Passage de paramètres graphiques à la commande <i>plot</i>	34
2.7	Tracer une courbe avec la commande <i>plot</i>	35
2.8	Tracer une courbe avec la commande <i>curve</i>	36
2.9	Distribution des altitudes des aéroports canadiens	38

Liste des codes sources

1.1	Environnement de travail	8
2.1	Extraction des données	11
2.2	Filtrer les données	12
2.3	Traitement standard de données géospatiales en R	16
2.4	Exemple de requête SQL	17
2.5	Fonctionnalités avancées de SQL	18
2.6	Fonctionnalités avancées de SQL	18
2.7	Fonctions de visualisation de données	20
2.8	Générer une carte du trafic aérien avec <i>ggmap</i>	21
2.9	Générer une carte du trafic aérien avec <i>leaflet</i>	22
2.10	Structure pour la définition d'une fonction	25
2.11	L'instruction <i>return</i> et le retour standard d'une fonction R	26
2.12	Définir des valeurs par défauts dans les fonctions utilitaires	26
2.13	Passage d'arguments à une fonction	28
2.14	L'assignation et les valeurs par défaut	29
2.15	Retour multiple par l'entremise d'une liste	30
2.16	Gestion des erreurs sous R	30
2.17	Utilisation de la commande <i>plot</i>	33
2.18	Utilisation de la commande <i>curve</i>	33
2.19	<i>hist</i> , <i>density</i> , <i>lines</i> , <i>abline</i> , <i>legend</i> et <i>mtext</i>	37
2.20	Fonctions relatives à la distribution Normale	40
2.21	Génération de nombres aléatoires	40
2.22	Fonctions de densité et de répartition empiriques	41
2.23	Tests d'indépendance et de corrélation entre distributions	42
2.24	Régression linéaire sur données empiriques	43
A.1	Benchmark.R	48
A.2	CaseStudyDevQ1.R	49
A.3	CaseStudyDevQ2.R	56
A.4	CaseStudyDevQ3.R	60
A.5	CaseStudyDevQ4.R	60
A.6	CaseStudyDevQ5.R	62
A.7	CaseStudyDevQ6.R	64

Liste des tableaux

2.1	Liste des distributions statistiques disponibles en R	39
2.2	Comparaison entre les coefficients réels et estimés par régression linéaire	44

Préface

Dans le cadre du colloque "R à Québec" qui se tiendra le 25 et 26 mai 2017 sur le campus de l'Université Laval, une séance d'introduction au langage de programmation R sera offerte aux participants. Cette séance vise principalement la compréhension et la pratique permettant de maîtriser les rudiments de cet environnement de programmation. [12] Cette séance sera divisée en deux parties. En ce qui concerne la première partie, les fondements du langage seront visités d'une manière théorique sous la forme d'un exposé magistral. La deuxième partie, tant qu'à elle, se concentrera davantage sur la mise en pratique des notions abordées lors de la première partie grâce à la complétion d'une étude de cas cherchant à faire l'analyse de marché du transport aérien canadien. Ce document correspond en fait à la documentation complète de cette deuxième partie de formation.

Étant donné qu'il s'agit tout de même d'une formation pour débutants, la majorité du code sera déjà fournie, mais il n'en vaut pas moins la peine de parcourir ce projet si ce n'est que pour constater la puissance et la simplicité du langage. De plus, il est souvent difficile de mettre en perspective les innombrables fonctionnalités d'un langage lorsque nous commençons à l'utiliser. Cet étude de cas nous fournit ainsi un bel exemple d'enchaînement de traitements jusqu'à l'aboutissement ultime qui consiste à répondre aux questions que nous nous posions avant même d'amorcer l'analyse.

D'autre part, il est important de préciser que le code qui sera présenté ne correspond pas toujours à la manière la plus efficiente d'accomplir une tâche donnée. L'objectif principal étant ici la transmission de connaissances dans un dessin éducatif plutôt que d'une réelle analyse de marché. Nous tenons aussi à mentionner que bien qu'il s'agisse d'une formation s'adressant à des débutants, plusieurs notions qui seront mises en valeur font plutôt état de niveau intermédiaire et avancé, mais apportées toujours de manière simplifiée et accessible à quiconque qui n'aurait jamais travaillé avec R.

Nous tenons à remercier Vincent Goulet de nous avoir fait confiance dans l'élaboration de cette partie de la formation ainsi que tous les membres du comité organisationnel de l'évènement. Nous croyons sincèrement que R est un langage d'actualité qui se révèle un atout à quiconque oeuvrant dans un domaine relié de près ou de loin aux mathématiques.

Introduction

Dans le cadre de cette étude de cas, nous nous placerons dans la peau d'un analyste du département de la tarification oeuvrant au sein d'une compagnie canadienne se sp  cialisant dans le transport de colis par voies a  riennes en mettant    profit le jeu de donn  es d'*OpenFlights*. [7]



FIGURE 1.1 – Interface de l'outil OpenFlights

Parmi les bases de donn  es disponibles, nous retrouvons :

- airports.dat** Donn  es relatives    tous les a  roports du monde
- routes.dat** Donn  es relatives    tous les trajets possibles dans le monde
- airlines.dat** Donn  es relatives    toutes les compagnies a  riennes du monde

Ainsi, notre mandat consistera, dans un premier temps,    analyser les bases de donn  es mises    notre disposition afin de cr  er des fonctions utilitaires qui permettront de facilement int  grer les informations qu'elles contiennent lors de la tarification d'une livraison sp  cifique. Une fois cette tarification compl  t  e, nous devrons fournir des chartes pour facilement estimer les prix d'une livraison qui s'av  reront   tre des outils indispensables au d  partement de marketing et au reste de la direction. Apr  s avoir transmis les documents en question, votre gestionnaire voulant s'assurer que la nouvelle tarification sera efficace et profitable vous demandera d'analyser les prix de la concurrence pour en extrapoler

leur tarification. Finalement, vous serez appelé à comparer ces deux tarifications et la compétitivité de votre nouvelle tarification comparativement au reste du marché en procédant à une analyse stochastique.



OpenFlights

OpenFlights est un outil en ligne permettant de visualiser, chercher et filtrer tous les vols aériens dans le monde. Il s'agit d'un projet libre entretenu par la communauté via GitHub. [5] L'information rendu disponible y est étonnamment très complète et facile d'approche ce qui en rend ce jeu de données très intéressant pour quiconque qui désire s'initier à l'analyse statistique. <https://openflights.org/>

Bien qu'on n'en soit toujours qu'à l'introduction, nous tenons dès lors introduire des notions de programmation qui comparativement à celles qui suivront sont d'autre un peu plus général. Tout d'abord, afin de maximiser la portabilité des scripts que vous créerez dans le futur, il est important de rendre votre environnement de travail indépendant de la structure de dossier dans laquelle il se trouve. Pour ce faire, nous devons donc utiliser le principe de liens relatifs plutôt qu'absolus. En R, deux fonctions bien spécifiques nous fournissent les outils afin de rendre cette tâche possible. Il s'agit de *getwd* et *setwd*. Comme leurs noms l'indiquent, elles servent respectivement à extraire le chemin vers l'environnement de travail et à le modifier.

De manière similaire qu'au sein d'un invité de commandes traditionnel, il est possible d'utiliser "." (cd ..) afin de revenir à un niveau supérieur dans la structure de dossiers. Dans la plupart des cas, le code source d'un projet sera souvent isolé du reste du projet en le plaçant dans un sous-dossier dédié.¹

Bref, comme le code source du présent projet se retrouve à l'intérieur du sous-dossier *dev* et que nous pourrions vouloir avoir accès à d'autres parties du répertoire au sein du code, le code suivant nous permettra de placer notre racine de projet à un niveau de dossier supérieur et de stocker ce chemin dans la variable *path*. Avec cette variable, tous les appels subséquents à des portions du répertoire pourront donc ce faire de manière relative puisque c'est cette variable *path* qui changera d'une architecture à un autre, tandis que la structure du répertoire restera toujours la même.

La deuxième notion que nous tenons à introduire immédiatement est celle de reproductibilité d'une analyse statistique. Comme vous le savez probablement, l'aléatoire pur n'existe pas en informatique, d'où la raison pour laquelle nous utiliserons plutôt le terme de nombres pseudo-aléatoires. Bien que cela peut

1. Il s'agit ici d'une excellente pratique de programmation et je dirais même indispensable si vous utilisez un gestionnaire de versions.

sembler étrange à première vue, il existe tout de même un point positif à tout ceci, soit la possibilité de fixer une racine au générateur de nombre pseudo-aléatoire (GNPA) ce qui aura comme impact de toujours produire les mêmes résultats pour autant que le GNPA utilisé soit le même. Comme nous pouvons le voir dans le code ci-dessous, l'instruction *set.seed* se chargera de fournir une valeur de départ aux calculs du GNPA.

Code Source 1.1 – Environnement de travail

```
1 ##### Setting working directory properly #####  
2 setwd('.')  
3 path <- getwd()  
4 set.seed(31459)
```



Code source du projet

Le code source du projet se retrouve dans son intégrité en annexe à ce document. N'hésitez pas à vous y référer au besoin.

Étude de cas

2.1 Extraction, traitement, visualisation et analyse des données

Cette section est certainement la plus importante de toutes, elle vise à faire un traitement adéquat et pertinent des données afin de pouvoir les réutiliser facilement dans les sections suivantes. Une mauvaise application des concepts d'extraction, de traitement et de visualisation des données peut entraîner des interprétations abérantes des phénomènes que nous cherchons à analyser.

2.1.1 Extraction

Les données d'OpenFlights possèdent l'avantage d'être téléchargeables directement via le web pour les rendre disponibles à notre environnement de travail. Pour ce faire, nous mettons à profit la fonction *read.csv*. Bien que le nom de la fonction indique qu'elle permet de lire un fichier présenté dans un format *.csv*, nous pouvons tout aussi bien utiliser cette fonction pour extraire des fichiers *.dat*. La différence principale entre ces deux types de fichiers et que les fichiers *.csv* utilisent un caractère d'encadrement des informations qui se trouve à être les doubles guillemets dans la majorité des cas. De plus, les fichiers *.csv* utilisent comme leur nom l'indique la virgule à titre de séparateur bien que celui-ci puisse être modifié pour un symbol différent.[2] Lorsque nous jetons un coup d'oeil à la structure des fichiers *.dat* disponibles à la [Figure 2.1](#), nous constatons que ceux-ci respectent à la fois les deux caractéristiques que nous venons de mentionner rendant ainsi l'utilisation de la fonction *read.csv* si naturelle.

Dans la même figure, on constate aussi l'absence d'une ligne servant à présenter les en-têtes de colonnes. Ceci pourra dans certains cas vous jouer de mauvais tours en ignorant la première ligne de données ou encore de considérer les titres comme étant des entrées en soi.¹ Bien qu'il serait possible de travailler avec des données sans nom, il s'agit ici d'une très mauvaise pratique à proscrire. Pour remédier à la situation, nous assignerons donc des noms aux colonnes grâce à l'attribut *colnames* d'un objet *data.frame* en lui passant un vecteur de noms.

1. La deuxième situation étant bien moins dramatique et plus facilement identifiable.

```

1,"Goroka Airport","Goroka","Papua New Guinea","GKA","AYGA",-6.081689834590001,145.391998291,5282,
2,"Madang Airport","Madang","Papua New Guinea","MAG","AYMD",-5.20707988739,145.789001465,20,10,"U"
3,"Mount Hagen Kagamuga Airport","Mount Hagen","Papua New Guinea","HGU","AYMH",-5.826789855957031,
4,"Nadzab Airport","Nadzab","Papua New Guinea","LAE","AYNZ",-6.569803,146.725977,239,10,"U","Pacif
5,"Port Moresby Jacksons International Airport","Port Moresby","Papua New Guinea","POM","APY", -9.
6,"Wewak International Airport","Wewak","Papua New Guinea","WWK","AYWK",-3.58383011818,143.6690063
7,"Narsarsuaq Airport","Narsarsuaq","Greenland","UAK","BGBW",61.1604995728,-45.4259986877,112,-3
8,"Godthaab / Nuuk Airport","Godthaab","Greenland","GOH","BGGH",64.19090271,-51.6781005859,283,-3
9,"Kangerlussuaq Airport","Sondrestrom","Greenland","SFJ","BGSF",67.0122218992,-50.7116031647,165,
10,"Thule Air Base","Thule","Greenland","THU","BGT",76.5311965942,-68.7032012939,251,-4,"E","Amer
11,"Akureyri Airport","Akureyri","Iceland","AEY","BIAR",65.66000366210938,-18.07270050048828,6,0,"I
12,"Egilsstaðir Airport","Egilsstaðir","Iceland","EGS","BIEG",65.2833023071289,-14.401399612426758
13,"Hornafjörður Airport","Hofn","Iceland","HFN","BIHN",64.295601,-15.2272,24,0,"N","Atlantic/Reyk
14,"Húsavík Airport","Húsavík","Iceland","HZK","BIHU",65.952301,-17.426001,48,0,"N","Atlantic/Reyk
15,"Ísafjörður Airport","Ísafjörður","Iceland","IFJ","BIIS",66.05809783935547,-23.135299682617188,

```

FIGURE 2.1 – Extrait du fichier airports.dat

Par défaut, lors de l'importation, la fonction `read.csv` retournera un `data.frame` en transformant les chaînes de caractères sous la forme de facteurs (*factors*). Cette action sera complètement transparente à l'utilisateur puisque l'affichage des variables ne sera pas impacté étant donné que R aura créé des formats d'affichage qui associe à chaque facteur la valeur unique correspondante. Le seul impact réel réside dans la possibilité d'utiliser des fonctions à caractères mathématiques sur les données peu importe si ces dernières sont numériques ou non. Parmi ce genre de fonctions, nous pouvons penser à des fonctions d'agrégation (*clustering*) ou tout simplement à l'utilisation de la fonction `summary` permettant d'afficher des informations génériques sur le contenu d'un objet. Il est important de comprendre que les données ne sont toutefois plus représentées comme des chaînes de caractères, mais bien pas un indice référant à la valeur textuelle correspondante.

La manière de représenter des valeurs manquantes variera souvent d'une base de données à une autre. Une fonctionnalité très intéressante de la fonction `read.csv` est de pouvoir automatiquement convertir ces chaînes de caractères symboliques en `NA` ayant une signification particulière dans R. Dans le cas présent, les valeurs manquantes sont représentées par `\\n` ou `" "` correspondant à un simple retour de chariot et un espace vide respectivement. Il suffit donc de passer cette liste de valeurs à l'argument `na.strings`.



read.csv

La fonction *read.csv* possède plusieurs autres arguments très intéressants dans des situations plus pointue. Pour en savoir plus, nous vous invitons à consulter la documentation officielle.

<https://stat.ethz.ch/R-manual/R-devel/library/utils/html/read.table.html>

Comme nous venons de le démontrer, l'extraction des données peut facilement devenir une tâche ingrate si nous n'avons aucune connaissance sur la manière dont l'information y a été entreposée. La règle d'or est donc de toujours avoir une idée globale de ce que nous cherchons à importer afin de bien paramétrer les fonctions. Si nous assemblons les différents aspects que nous venons d'aborder, nous aboutissons donc au code suivant :

Code Source 2.1 – Extraction des données

```
1 airports <- read.csv("https://raw.githubusercontent.com/jpatokal/
  openflights/master/data/airports.dat", header = FALSE, na.
  strings=c('\\N', ''))
```

2.1.2 Traitement

Une fois en possession du jeu de données, il fut nécessaire de nettoyer ce dernier pour en rendre son utilisation plus simple selon nos besoins. Parmi les différentes modifications apportées nous retrouvons :

- ▶ Conserver que les observations relatives aux aéroports Canadiens
- ▶ Filtrer les variables qui seront pertinentes dans le cadre de l'analyse que nous menons.²
- ▶ Alimentation des valeurs manquantes avec des sources de données externes (si possible) ou appliquer un traitement approximatif justifiable en documentant les impacts possibles sur le restant de l'analyse.

Nous considérons pertinent d'apporter quelques précisions sur le fonctionnement de R avant d'expliquer les traitements sus-mentionnés. Tout d'abord, R est un langage interprété orienté objet à caractère fonctionnel optimisé pour le traitement vectoriel. Ces simples mots ne sont pas à prendre à la légère puisque ce n'est qu'en s'appropriant ce mode de penser que les futurs développeurs que vous êtes parviendront à utiliser R dans toute sa puissance, sa simplicité et son élégance. Par sa sémantique objet, R permet de définir des attributs aux objets créés. Comme il sera possible de le voir plus loin, l'accès à ces attributs se

2. On ne devrait jamais travailler avec des informations superflues. Faire une pré-sélection de l'information ne fait qu'alléger les traitements et augmente de manière significative la compréhensibilité du programme.

fera à l'aide de l'opérateur `$`. Vous vous demandez probablement : Comment savoir si nous sommes en présence d'un objet ? C'est simple, tout dans R est un objet ! Le langage R permet aussi de mimer le paradigme fonctionnel puisque les fonctions (qui sont en fait des objets) sont des valeurs à part entière qui peuvent être argument ou valeur d'une autre fonction. De plus, il est possible de définir des fonctions dites anonymes qui se révéleront très pratiques. Finalement, par son caractère vectoriel, la notion de scalaire n'existe tout simplement pas en R. C'est pour cette raison que l'utilisation de boucles est à proscrire (ou du moins à minimiser le plus possible). En effet, l'utilisation d'une boucle revient en quelque sorte à la création d'un nouveau vecteur et à la mise en place de processus itératifs afin d'exécuter la tâche demandée. Heureusement, par un raisonnement vectoriel, il est très simple de convertir ces traitements sous une forme vectorielle dans la plupart des cas. [6] Pour accéder à une valeur précise d'un vecteur, nous utiliserons l'opérateur `[]` en spécifiant les indices correspondants aux valeurs désirées, un vecteur booléen d'inclusion/exclusion ou encore un vecteur contenant les noms des attributs nommés qui nous intéressent.

Avec ces outils en mains, il devient ainsi très facile de filtrer les aéroports canadiens à l'aide de l'attribut que nous avons nommé *country* du data.frame *airports*. Par un raisonnement connexe, la fonction *subset* nous offre aussi la possibilité de conserver que certaines variables contenues dans une table tout en appliquant des contraintes sur les observations à conserver. Le ?? dévoile au grand jour la dualité qui peut exister entre la multitude des fonctionnalités présentes en R.

Code Source 2.2 – Filtrer les données

```
1 airportsCanada <- airports[airports$country=='Canada',]
2 airportsCanada2 <- subset(airports, country == 'Canada')
3 all.equal(airportsCanada, airportsCanada2)
4
5 airportsCanada[is.na(airportsCanada$IATA), c("airportID", "name", "
6   IATA", "ICAO")]
7 subset(airportsCanada, is.na(IATA), select = c("airportID", "name",
8   "IATA", "ICAO"))
```

Nous ne devons pas être surpris qu'il y ait autant de possibilités différentes de parvenir au même résultat, il s'agit là d'une des principales caractéristiques d'un logiciel libre puisque la responsabilité du développement continu ne dépend plus que d'une seule personne ou entité, mais bien de la communauté d'utilisateurs au complet. Ceci peut toutefois sembler mélangeant pour des nouveaux utilisateurs et la question suivante arrivera assez rapidement lorsque vous commencerez à développer vos propres applications : Quelle est la meilleure manière d'accomplir une tâche X ? La bonne réponse est tout aussi décevante que la prémisse étant donné que chaque fonction aura été développée dans un besoin précis si ce n'est que de rendre l'utilisation de fonctionnalité de base plus aisée et facile d'approche... C'est pourquoi nous conseillons plutôt d'adopter un mode de penser

itératif, créatif et ouvert qui consiste à utiliser les fonctions qui vous semblent, à la fois, les plus simples, les plus versatiles et les plus efficaces. À partir du moment où vous constaterez qu’une de ces trois caractéristiques n’est plus au rendez-vous, il suffira d’amorcer des recherches pour bonifier vos connaissances et améliorer vos techniques. C’est un peu le but de ce document de vous faire faire une visite guidée pour vous offrir un coffre d’outil qui facilitera vos premiers pas en R.



subset

Bien que la fonction *subset* simplifie énormément l’écriture de requêtes afin de manipuler des bases de données, celle-ci souffre par le fait même de devenir rapidement inefficace lors de traitements plus complexes. D’autres packages tels que *dplyr* et *sqldf* deviendront dans ces situations des meilleures alternatives.

<https://www.rdocumentation.org/packages/raster/versions/2.5-8/topics/subset>

Après avoir fait une présélection des données qui nous seront utiles dans le reste de l’analyse, nous avons constaté que certaines variables n’étaient pas toujours totalement alimentées. Tout d’abord, la variable IATA n’était pas toujours définie pour tous les aéroports canadiens contrairement à ICAO. Étant donné la faible proportion des valeurs manquantes et du fait qu’une valeur fictive n’aurait qu’un impact minimal dans le cas de l’analyse, nous avons décidé de remplacer les valeurs manquantes par les 3 dernières lettres du code ICAO. En regardant les aéroports canadiens possédant les deux codes, nous observons que cette relation est respectée dans plus de 80% des cas. Une autre alternative aurait été de simplement prendre le code ICAO, mais le code IATA semblait beaucoup plus facile d’accès puisqu’il s’agit du code communément utilisé pour le transport des particuliers.

Les vrais problèmes au niveau des données résidaient davantage dans l’absence d’informations sur les fuseaux horaires de certains aéroports ainsi qu’un accès indirect à la province de correspondance de tous les aéroports. Heureusement, ce genre d’information ne dépend que de l’emplacement de l’entité dans le monde, ce qui rend la tâche beaucoup plus simple lorsque nous avons accès aux coordonnées géospatiales.

i

Adresses et coordonnées géospatiales

Dans la situation où seule l'adresse de l'entité aurait été disponible, nous aurions été contraint d'utiliser des techniques de géocodage qui permettent de transformer une adresse en coordonnées longitude/latitude et parfois même altitude. Ce genre de transformation est devenu beaucoup plus accessible avec l'avancement de la technologie et plusieurs APIs sont disponibles gratuitement sur le web pour procéder à ce genre de transformation. Encore une fois, il vaut mieux bien se renseigner pour identifier l'interface qui répondra le mieux à nos besoins en considérant notamment :

- ▶ Format de l'intrant
- ▶ Format de retour
- ▶ Limitation du nombre de requêtes sur une période de temps donnée
- ▶ Efficacité de l'outil
- ▶ Méthode d'interpolation
- ▶ Précision des valeurs

<https://www.programmableweb.com/news/7-free-geocoding-apis-google-bing-yahoo-and-mapquest/2012/06/21>

Bien que nous savons qu'il est possible de populer les valeurs manquantes à l'aide de données géographiques encore faut-il disposer de ses dites données. Encore une fois, grâce à de bonnes recherches vous parviendrez à trouver une source qui contiendra ce dont vous cherchez ou du moins un élément de réponse qui vous permettra d'en extrapoler la valeur ce qui sera déjà préférable à des données manquantes. Statistiques Canada possède une bibliothèque géographique très garnie et c'est notamment sur leur site que nous avons pris le fichier `.shp` qui définit les provinces et territoires du Canada. [16] En ce qui concerne les fuseaux horaires, nous avons trouvé ceux-ci sur un site dédié à cette fin qui mentionne ne plus être maintenu à jour, mais dont la dernière mise-à-jour a été fait le 28 mai 2016. Étant donné que les fuseaux horaires n'ont pas tendance à changer souvent dans les pays industrialisés comme le Canada, ceci ne consistait pas en un enjeu majeur. [17]



ArcGIS et les fichiers *.shp*

Le premier fichier ayant l'extension *.shp* fut créé dans le but d'être utilisé conjointement avec la suite de logiciel ArcGIS. Il s'agit de la première suite logiciel commercialisable visant le traitement des données géospatiales. Étant des pionniers dans le domaine, plusieurs aspects des outils visant à faire des traitements géospatiaux proviendront directement de leur travaux. Les fichiers *.shp* sont aujourd'hui vu comme un standard pour transporter ce type de données.

<https://www.arcgis.com/features/index.html>

Pour être en mesure de bien travailler avec ce genre de fichier nous devons en comprendre leur fonctionnement. Tout d'abord, lorsque vous téléchargerez un *.zip* de données géospatiales, vous devriez toujours obtenir la structure suivante de fichiers :





Name	Date modified	Type	Size
 gpr_000b11a_e.dbf	5/13/2017 10:48 PM	DBF File	2 KB
 gpr_000b11a_e.prj	5/13/2017 10:48 PM	PRJ File	1 KB
 gpr_000b11a_e.shp	5/13/2017 10:48 PM	SHP File	53,066 KB
 gpr_000b11a_e.shx	5/13/2017 10:48 PM	SHX File	1 KB

FIGURE 2.2 – Structure des fichiers de données géospatiales

Tel qu'illustré à la Figure 2.2, un dossier de données géospatiales se divisera minimalement sous la forme de quatre fichiers :

- .shp*** Contient l'information géographique représentée sous la forme de points, segments et/ou polygones
- .dbf*** Contient l'information rattachée à tous les entités définies dans le *.shp*
- .prj*** Contient les informations sur la projection associée (le modèle mathématique permettant d'interpréter les informations du *.shp* [10])
- .shx*** Contient les index des enregistrements du *.shp*

Cette structure peut donner l'impression que leur utilisation conjointement avec R sera compliqué, mais c'est loin d'être le cas grâce aux paquetages *rgdal* et *sp*. Pour conclure sur ce point, notons que la désignation *ShapeFile* au sens large désigne l'ensemble de la structure de fichier et non pas seulement le *.shp* lui-même. [1]

Le paquetage *rgdal* n'aura qu'une utilité bien précise, soit celle d'aller extraire les informations contenues dans le *ShapeFile*. Cepenat, il possède des dépendances

directement dans le paquetage *sp* ce qui explique pourquoi le seul appel de *rgdal* entraîne du même coup l'appel de *sp*. Les rôles de *sp* sont plutôt de transformer les informations des objets R sous une forme compatible au *ShapeFile* que nous aurons lu. Notez bien la transformation de la projection sous une base commune en passant ainsi de *NA* vers

```
"+proj=longlat"
```

(projection choisie en fonction des données contenues) à

```
"+proj=longlat +datum=NAD83 +no_defs +ellps=GRS80 +towgs84=0,0,0"
```

soit la projection du *ShapeFile*. La nécessité que nos points soit sous la même projection que celle du *ShapeFile* provient du fait que nous voulons superposer ces derniers pour ensuite en extraire l'information correspondante. Les deux fonctions indispensables ici sont *CRS* qui retourne un objet de classe *Coordinate Reference System* à partir d'une chaîne de caractère passée en argument et *over* qui se chargera de faire la superposition de points géométriques sur un couche (correspondant ici au *ShapeFile* vu selon une certaine projection) qui contient les attributs envers lesquels nous avons un intérêt. Le retour de la fonction *over* sera finalement un *data.frame* de même longueur que le nombre de points donnés en argument que nous pourrions facilement combiner le jeu de données initial. Cette recette ne risque pas de varier beaucoup d'un *ShapeFile* à un autre vous pourrez donc littéralement reprendre le code ci-dessous.

Code Source 2.3 – Traitement standard de données géospatiales en R

```
1  # Step 1 – Import the Packages
2  library(sp)
3  library(rgdal)
4  # Step 2 – Read the ShapeFile
5  prov_terr.shape <- readOGR(dsn=paste(path,"/Reference/prov_terr",
6  sep=""),layer="gpr_000b11a_e")
7  # Step 3 – Create the Spatial Points to be overlaid
8  unknown_prov <- airportsCanada[,c("airportID","city","longitude",
9  "latitude")]
10 sppts <- SpatialPoints(unknown_prov[,c("longitude","latitude")])
11 # Step 4 – Set the Spatial Points on the same projection as the
12   ShapeFile
13 proj4string(sppts) <- CRS("+proj=longlat")
14 sppts <- spTransform(sppts, proj4string(prov_terr.shape))
15 # Step 5 – Extract the Desired Information by overlaying the
16   Spatial Points on the ShapeFile
17 merged_prov <- cbind(airportsCanada, over(sppts, prov_terr.shape))
```

Maintenant que nous disposons de l'information requise pour compléter notre base de données, nous devons combiner la table primaire avec les sous-tables créées lors de nos extractions et refaire un dernier filtre final pour se débarrasser de tout ce qui ne sera plus utile. Bien que les fonctionnalités de base de R vous permettrait d'accomplir la tâche, nous profitons de cette étape du processus

pour vous introduire les paquetages *sqldf* et *dplyr*.

Le langage SQL (*Structured Query Language*) fut inventé en 1974 et ce dernier fut normalisé en 1986 devenant ainsi un standard dans l'exploitation de base de données relationnelles. Devenir familier avec les langages normalisés tel que le SQL ne peut qu'être à votre avantage puisque ceux-ci vous permettront d'écrire des tronçons de code qui pourront facilement être transportés avec peu de modifications d'un environnement à un autre. Leur caractère normalisé impose aux environnements voulant respecter les standards de l'industrie d'être en mesure de compiler ces instructions bien qu'il y ait des fonctionnalités permettant de répliquer leur comportement ou du moins offrir un paquetage permettant leur interprétation. [14] Nous conseillons fortement à tous les analystes de données de s'approprier les rudiments du SQL très tôt dans leur cheminement en raison de sa simplicité et sa flexibilité. Les requêtes SQL sont habituellement constituées des quatre instructions suivantes :

- | | |
|-----------------|--|
| Select | Déclare les variables que nous voulons conserver |
| From | Indique la source des données |
| Where | Mentionne les conditions que les observations doivent respecter pour se retrouver dans l'extrait |
| Order by | Spécifie la manière de trier l'extrait |

La syntaxe rudimentaire rend sa compréhension presque immédiate, et ce, même à des personnes ignorant même qu'il s'agit en fait d'une requête SQL. Dépendamment des noms de variables contenues dans les relations exploitées, les requêtes peuvent parfois se lire aussi bien qu'une liste d'épicerie écrite en anglais. Le [Code Source 2.4](#) fournit un exemple de l'utilisation du langage SQL avec R rendu disponible par le paquetage *sqldf*.

Code Source 2.4 – Exemple de requête SQL

```
1 library(sqldf)
2 sqldf("SELECT name,IATA,altitude,province
3       FROM airportsCanada
4       WHERE province = 'New Brunswick'
5       ORDER BY name")
```

En nous fiant à la requête ci-dessus, nous pourrions la transformer de manière textuelle sous la forme suivante :

1. Sélectionne les variables *name*, *IATA*, *altitude* et *province*
2. Dans la relation *airportsCanada*
3. Dont la province est *New Brunswick*
4. En triant le tout par *name*

Toutefois, les fonctionnalités de SQL ne s'arrêtent pas ici. Grâce à des intructions très compactes, nous pourrions rendre le comportement de la requête bien plus

complexe. Parmi les fonctionnalités qui font partie de notre quotidien, nous retrouvons `*` qui lorsqu'utiliser au sein de l'instruction `select` permettra d'extraire l'ensemble des variables de la relation sans avoir à les écrire une à la fois. La fonction `coalesce` servira à extraire la première valeur non manquante parmi une liste de variables fournie en argument. Nous attirons au passage votre attention sur le mot clé `as` qui a pour effet d'attribuer un nom à l'expression sous-jacente. Finalement, le bon vieux `left join` rendant si simple la fusion conditionnelle de deux tables en conservant toutefois les observations de la relation mère malgré le fait qu'il n'y ait pas eu correspondance dans la table à fusionner. Les conditions de cette fusion seront explicitées avec l'instruction `on` qui n'aura pas de signification tangible sans la présence de `join`. Le [Code Source 2.5](#) présente une requête combinant tous ces fonctionnalités que vous serez en mesure de retrouver dans le code source du projet.

Code Source 2.5 – Fonctionnalités avancées de SQL

```
1 airportsCanada <- sqldf("
2   SELECT
3     a.*,
4     COALESCE(a.tzFormat,b.TZID) AS tzMerged,
5     c.PRENAME AS provMerged
6   FROM airportsCanada a
7   LEFT JOIN merged_tz b
8     ON a.airportID = b.airportID
9   LEFT JOIN merged_prov c
10    ON a.airportID = c.airportID
11  ORDER BY a.airportID")
```

Il serait faux de dire que ceci correspond à une bonne introduction à SQL sans parler de la capacité d'imbriquer des requête SQL. C'est à ce moment que toute la puissance du langage se révèle à nous. Le [Code Source 2.6](#) montre un exemple standard d'imbrication qui a été exploité pour créer la relation `routesCanada` en ne conservant que les routes aériennes empruntées pour les vols internes au Canada.^{3 4}

Code Source 2.6 – Fonctionnalités avancées de SQL

```
1 routesCanada <- sqldf("
2   SELECT *
3   FROM routes
4   WHERE sourceAirportID IN (SELECT DISTINCT airportID
5                             FROM airportsCanada)
6     AND destinationAirportID IN (SELECT DISTINCT airportID
7                                  FROM airportsCanada)")
```

3. Le mot clé *DISTINCT* spécifie de ne conserver qu'une seule observation pour chaque modalité retrouvée

4. L'utilisation de la case dans les exemples n'a été utilisé que pour bien faire la différence entre les instructions SQL des informations spécifiques aux relations traitées. Le SQL n'est pas sensible à la case.



Structured Query Language (SQL)

Le langage SQL regorge de plusieurs autres fonctionnalités qui ne seront pas abordées dans ce document. Parmi ces dernières, nous retrouvons *GROUP BY*, *HAVING*, les fonctions d'aggrégation numérique tel quel *SUM*, *AVG*, *MIN*, *MAX*, etc. et nous pourrions continuer ainsi encore longtemps.

<https://www.w3schools.com/sql/>

Avant de passer à la prochaine section, il serait injuste de présenter *sqldf* avec autant de précisions sans toucher un mot sur les paquetages *plyr* et *dplyr*. Ces derniers visent à reproduire les opérations permises par le langage SQL avec une notation aussi simpliste, mais en optimisant ces opérations en tenant compte du fonctionnement intrinsèque de R, soit le calcul matriciel. Une différence majeure avec le SQL provient du mode de pensée se rapprochant davantage d'un mode procédural pour *plyr* que du mode fonctionnel pour le SQL. Ces packages deviendront des outils très pertinents lorsque vous commencerez à faire face à des problèmes de temps d'exécution inraisonables.



plyr ou *dplyr* ?

Le paquetage *dplyr* est en fait une seconde version du paquetage *plyr* visant à optimiser le temps de calcul, simplifier son utilisation à l'aide d'une syntaxe plus intuitive et à rendre ses fonctions plus cohérentes entre elles. De plus, *dplyr* concentre son développement autour de la classe objet *data.frame*. Pour toutes ces raisons, l'utilisation de *dplyr* serait à préconiser si vous travaillez avec des *data.frame* qui consistent du même coup en la classe standard de R pour représenter les bases de données...

<https://blog.rstudio.org/2014/01/17/introducing-dplyr/>

2.1.3 Visualisation et analyse des données

La visualisation des données est une étape cruciale dans l'interprétation de ces dernières. En effet, seule une connaissance approfondie des données vous permettra d'en percevoir les secrets les plus précieux qui y résident. Afin de visualiser les données directement contenues dans une relation, le langage R met à notre disposition différentes fonctions qui sont décrites ci-dessous.

- | | |
|--------------------|---|
| <i>View</i> | Permet d'ouvrir la relation dans l'outil de visualisation de RStudio. Ce dernier permettra aussi d'appliquer des transformations de faible complexité comme le filtre sur une variable ou le tri. [?] |
| <i>head</i> | Renvoie en console un extrait des premières observations d'une relation (par défaut, 10 observations sont renvoyées) |

- summary** Compilation de statistiques pertinentes au sujet des différentes variables contenues dans une relation. Pour les variables quantitatives, le minimum, le 1^{er} quintile, la moyenne, la médiane, le 3^{ime} quintile et le maximum seront calculés, tandis qu’une simple analyse de fréquence des différentes modalités sera produite dans le cas d’une variable qualitative.
- table** Au même titre que le comportement de *summary* pour les variables qualitative, la fonction *table* renvoie un vecteur comptabilisant le nombre d’occurrences de chaque valeur unique.

Code Source 2.7 – Fonctions de visualisation de données

```
1 View(airportsCanada)
2 head(airportsCanada)
3 summary(airportsCanada)
4 nbAirportCity <- table(airportsCanada$city)
```

Ces fonctions ressemblent beaucoup plus à des outils pour optimiser le temps de développement qu’à des traitements que nous chercherons à laisser en production compte tenu de leur affichage très rudimentaire. De plus, il sera facile de se perdre dans le contenu présenté plus la relation possèdera des variables. Pour contrer ce problème, la production de graphiques sera la plupart du temps une solution plus qu’intéressante. Cependant, toujours dans un objectif de cohérence avec la structure du code source du projet, nous n’aborderons pas immédiatement la création de graphique en R. Nous nous contenterons plutôt d’introduire les méthodes de visualisation de données géospatiales pour faire le pont avec la [sous-section 2.1.2](#).

Au moment de l’analyse, deux paquetages ont retenus notre attention pour la production de cartes géographiques qui faciliteront la transmission de connaissances sommaires au sujet du jeu de données. Nos critères de sélection étaient encore une fois la simplicité des requêtes, la beauté de l’extrant final et la flexibilité des instructions pour les adapter à un contexte précis.

Le paquetage *ggmap*, nous a permis de produire la [Figure 2.3](#). Si cette dernière vous semble familière, ce n’est pas sans raison ! Le paquetage *ggmap* vise en fait à rendre la visualisation de données géospatiales sur des supports statiques disponibles en ligne tels que ceux de *Google Maps* en les combinant avec la puissance des fonctionnalités du paquetage *ggplot2*. [?]

En jetant un coup d’œil au [Code Source 2.8](#), nous voyons qu’il est possible des cartes très rapidement avec seulement quelques lignes de code. Malgré la facilité d’utilisation de *ggmap*, nous ressentons rapidement ses limitations lorsque nous espérons produire des cartes interactives similaires à celles que nous retrouvons dans la plupart des applications web et mobiles modernes.

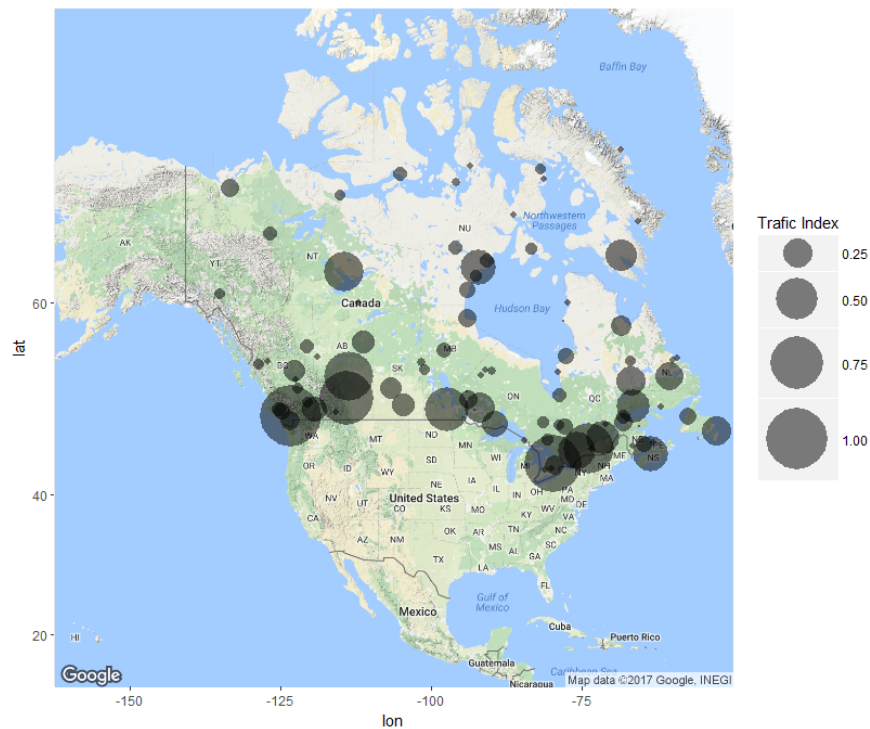


FIGURE 2.3 – Exemple de carte géographique produite avec *ggmap*

Code Source 2.8 – Générer une carte du trafic aérien avec *ggmap*

```

1 # install.packages("ggmap")
2 library(ggmap)
3 map <- get_map(location = "Canada", zoom = 3)
4 TrafficData <- subset(airportsCanada, as.numeric(paste(combinedIndex)
5   ) > 0.05)
6 lon <- as.numeric(paste(TrafficData$longitude))
7 lat <- as.numeric(paste(TrafficData$latitude))
8 size <- as.numeric(paste(TrafficData$combinedIndex))
9 airportsCoord <- as.data.frame(cbind(lon, lat, size))
10 mapPoints <-
11   ggmap(map) +
12     geom_point(data=TrafficData, aes(x=lon, y=lat, size=size), alpha=0.5,
13       shape=16)
14 (mapTraffic <-
15   mapPoints +
16   scale_size_continuous(range = c(0, 20), name = "Traffic Index"))

```

Pour ce faire, le paquetage *leaflet* viendra à notre secours avec un faible coût en complexité compte tenu de la flexibilité impressionnante rajoutée. Le [Code](#)

Source 2.9 est à l'origine de la vue statique présentée à la ?? extraite de la carte interactive qu'il génère.

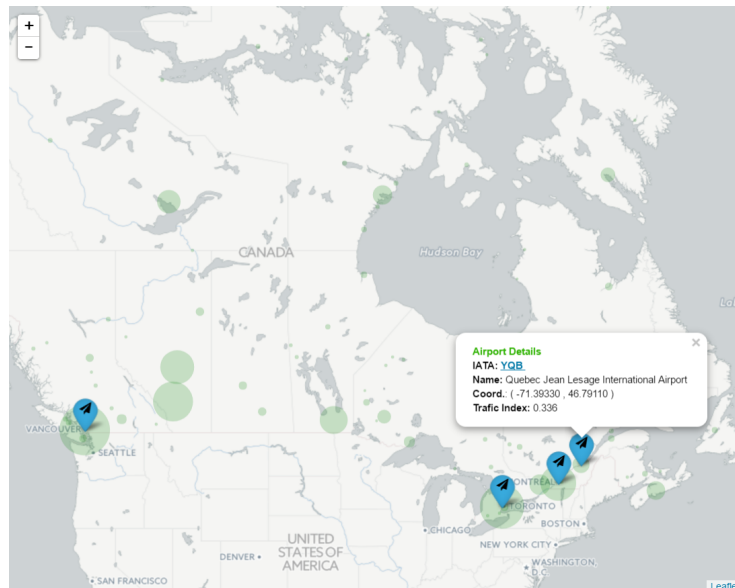


FIGURE 2.4 – Exemple de carte géographique produite avec *leaflet*

Code Source 2.9 – Générer une carte du trafic aérien avec *leaflet*

```

1 # install.package("leaflet")
2 library(leaflet)
3 url <- "http://hiking.waymarkedtrails.org/en/routebrowser/1225378/
  gpx"
4 download.file(url, destfile = paste(path, "/Reference/worldRoutes.
  gpx", sep=""), method = "wget")
5 worldRoutes <- readOGR(paste(path, "/Reference/worldRoutes.gpx", sep=
  ""), layer = "tracks")
6 markersData <- subset(airportsCanada, IATA %in% c('YUL', 'YVR', 'YYZ',
  'YQB'))
7 markersWeb <- c("https://www.aeroportoquebec.com/fr/pages/accueil"
  ,
  "http://www.admtl.com/",
  "http://www.yvr.ca/en/passengers",
  "https://www.torontopearson.com/")
8
9
10
11
12 # Defining the description text to be displayed by the markers
13 descriptions <- paste("<b><FONT COLOR=#31B404> Airport Details</FONT>
  </b> <br>",
14                       "<b>IATA: <a href=", markersWeb, ">", markersData$
  IATA, "</a></b><br>",
15                       "<b>Name: </b>", markersData$name, "<br>",

```

```

16         "<b>Coord.</b>: (" , markersData$longitude , " , " ,
           markersData$latitude , " ) <br>" ,
17         "<b>Traffic Index:</b>" , markersData$
           combinedIndex)
18
19 # Defining the icon to be add on the markers from fontawesome
    library
20 icons <- awesomeIcons(icon = 'paper-plane' ,
21                       iconColor = 'black' ,
22                       library = 'fa')
23
24 # Combinaison of the different components in order to create a
    standalone map
25 (mapTraffic <- leaflet(worldRoutes) %>%
26   addTiles(urlTemplate = "http://{s}.basemaps.cartocdn.com/light_
    all/{z}/{x}/{y}.png" ) %>%
27   addCircleMarkers(stroke = FALSE, data = TrafficData , ~as.numeric(
    paste(longitude)) , ~as.numeric(paste(latitude)) ,
28     color = 'black' , fillColor = 'green' ,
29     radius = ~as.numeric(paste(combinedIndex))*30 ,
    opacity = 0.5) %>%
30   addAwesomeMarkers(data = markersData , ~as.numeric(paste(
    longitude)) , ~as.numeric(paste(latitude)) , popup =
    descriptions , icon=icons))
31
32 # Resizing of the map
33 mapTraffic$width <- 874
34 mapTraffic$height <- 700
35
36 # Export of the map into html format
37 # install.packages("htmlwidgets")
38 library(htmlwidgets)
39 saveWidget(mapTraffic , paste(path , "/Reference/leafletTraffic.html" ,
    sep = "" ) , selfcontained = TRUE)

```

Le fonctionnement des deux paquetages est sensiblement le même. Nous commençons par extraire une carte qui servira de support directement à partir du web. Nous passons ensuite les informations géographiques nécessaires au constructeur du paquetage utilisé pour créer une instance. Nous ajoutons ensuite des composantes à cette instance à l'aide de méthode conçues spécifiquement à cette fin. Sans entrer davantage dans les détails, il est intéressant de mentionner les particularités que le paquetage *leaflet* offre en sus des fonctionnalités graphiques traditionnelles. Tout d'abord, les *markers* peuvent être personnalisés de fond en comble. Dans l'exemple présent, nous avons mis à profit la banque de symboles et d'outils CSS (*Cascading Style Sheets*) *fontawesome* [4] qui est célèbre auprès des utilisateurs L^AT_EX pour la diversité et la qualité de ses icônes. Un autre aspect encore plus pratique est la présentation d'informations supplémentaires lorsque l'utilisateur appuie sur le marqueur offrant ainsi une manière simple de stocké beaucoup d'information au sein du même objet sans alourdir indument sa lisibilité. L'ajout des ces informations et le formatage se résume par le passage de commande *html* directement à l'argument *popup*. Vous savez maintenant comment nous avons procédé pour exposer le code IATA, le nom,

les coordonnées géographiques ainsi que l'indice de trafic aérien sur chacun des marqueurs auxquels l'icône *fa-paper-plane* a été assigné. Le dernier point intéressant de *leaflet* est la capacité de créer des *widgets html* indépendant rendant le partage de l'information encore plus simple sans nécessité de recompiler le code source à chaque fois qu'un utilisateur aura envie de visionner l'objet. [8]



Est-ce tout ce que peut accomplir *leaflet* ?

Bien entendu, les exemples présentés dans ce document font l'éloge que de deux applications grossières des capacités de ces deux paquetages. Vous serez en mesure d'aisément trouver plusieurs autres exemples d'applications sur le web. Pour l'instant, voici quelques pages d'intérêt qui ont servies à créer la carte interactive pour amorcer vos recherches :

<https://rstudio.github.io/leaflet/>
<https://rstudio.github.io/leaflet/markers.html>
<https://rstudio.github.io/leaflet/popups.html>
<http://rgeomatic.hypotheses.org/550>
<https://www.r-bloggers.com/interactive-mapping-with-leaflet-in-r/>
<http://stackoverflow.com/questions/38837112/how-to-specify-radius-units-in-addcirclemarkers-when-using-leaflet-in-r>
<http://stackoverflow.com/questions/31562383/using-leaflet-library-to-output-multiple-popup-values>
<https://gis.stackexchange.com/questions/171827/generate-html-file-with-r-using-leaflet>

En terminant, il est possible de valider nos résultats en comparant ceux-ci avec la densité de la population canadienne. Nous devrions être en mesure d'observer une augmentatin du trafic aérien dans les zones où la densité de population est plus intense. Ainsi, nos cartographies sont cohérentes avec la Figure 2.5.

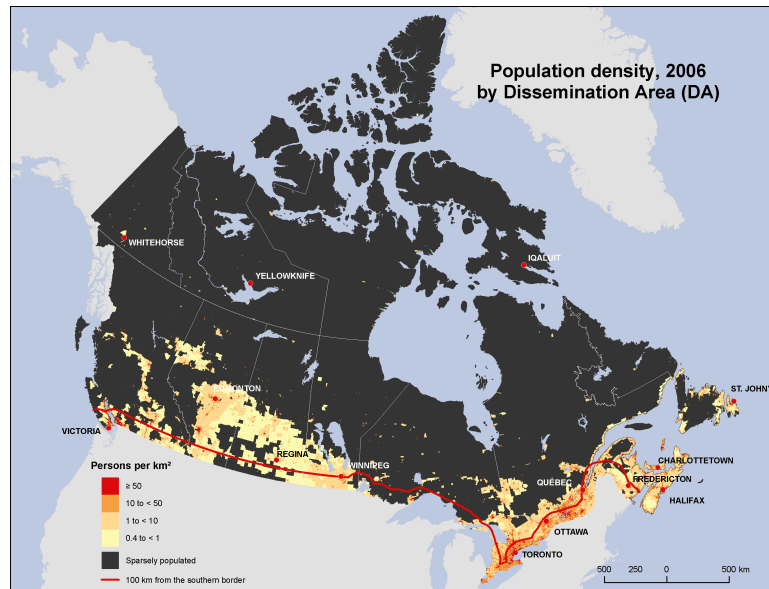


FIGURE 2.5 – Densité de la population canadienne

2.2 Création de fonctions utilitaires

Cette section servira principalement à faire la revue des concepts les plus importants dans la création de fonction utilitaires. Lorsque nous parlons de fonctions utilitaires, nous faisons référence à des fonctions définies par l'utilisateur afin de favoriser la compréhension de son programme et favoriser la réutilisation de tronçons de programme. Dans le cadre du projet, nous avons pris l'initiative de construire les trois fonctions suivantes :

airportsDist Calculer la distance en Km entre deux aéroports

arrivalTime Calculer l'heure d'arrivée d'un colis posté au moment du calcul

shippingCost Calculer le coût d'une livraison

Lorsque nous voulons définir une fonction, la structure présentée par le [Code Source 2.10](#) sera toujours utilisée.

Code Source 2.10 – Structure pour la définition d'une fonction

```
1 # nom_de_la_fonction <- function( liste_des_arguments )
2 # {
3 #   corps_de_la_fonction
4 #   ...
5 #   valeur_retournee_par_la_fonction
6 # }
```

À partir du [Code Source 2.10](#), nous pouvons dès lors déduire plusieurs éléments de théorie. Tout d’abord, le mot clé *function* sera toujours nécessaire pour mentionner à R que nous sommes en train de définir une fonction, et ce, qu’elle soit anonyme ou non. D’autre part, la valeur retournée par une fonction sera toujours la valeur de la dernière expression évaluée au sein de son corps qui sera délimitée par les accolades. Bien entendu, il est possible de contourner ce processus standard en introduisant le mot clé *return* qui aura pour effet d’entreprendre les processus de retour à l’exécution du programme principal tout en ignorant le reste de l’exécution que la fonction aurait pu engendrer. C’est exactement ce que le [Code Source 2.11](#) cherche à expliciter. Bien que la seule différence entre les deux fonctions soit la présence de l’instruction *return*, ces deux fonctions auront un comportement bien différent puisque la première retournera l’addition des deux paramètres qu’elle aura reçus pendant que la seconde ne arrêtera son exécution au croisement de l’instruction *return* pour renvoyer la valeur du premier argument, soit 5 et 2 respectivement. En théorie, nous chercherons donc à éviter l’utilisation du *return* ou d’autres modificateurs de flux du même genre et il n’y aura donc qu’une entrée et une sortie possible pour chaque fonction. En pratique, ce genre d’instruction peuvent simplifier grandement l’écriture du code, mais leur utilisation restera réservée à des situations bien particulière.

Code Source 2.11 – L’instruction *return* et le retour standard d’une fonction R

```

1 ftest1 <- function(a,b)
2 {
3   a+b
4 }
5 ftest2 <- function(a,b)
6 {
7   return(a)
8   a+b
9 }
10 ftest1(2,3)
11 ftest2(2,3)
```

L’exemple du [Code Source 2.11](#) combiné à la structure générique présentée précédemment, nous accorde un environnement idéal pour introduire les notions d’argumentation. Comme mentionné ci-dessus, le passage des arguments se fera à l’intérieur des parenthèses suivants le nom des fonctions. Il s’agit en fait de la même syntaxe que toutes les autres fonctions que nous avons déjà utilisées dans la section précédente. En d’autres mots, une fois une fonction utilitaire définie correctement par l’utilisateur, celle-ci sera équivalente aux autres fonctions rendues disponibles par les différents paquetages. Si nous examinons le [Code Source 2.11](#), nous voyons que la fonction *ftest1* et *ftest2* prennent 2 paramètres à titre d’arguments nommé *a* et *b*. Une fois les arguments déclarés dans l’en-tête de fonction, nous pourrons les utiliser comme bon nous semble à l’intérieur du corps en utilisant leur étiquette.

Code Source 2.12 – Définir des valeurs par défauts dans les fonctions utilitaires

```
1 ftest3 <- function(a=2,b=3)
2 {
3   a+b
4 }
5 ftest3()
```

Comme plusieurs autres langages de programmation, la méthode entreprise pour définir des paramètres par défaut revient simplement à en faire la définition directement dans l'en-tête de la fonction grâce à l'opérateur d'égalité. Bien que la définition de paramètre par défaut peut sembler anodine pour un nouveau programmeur, vous apprendrez rapidement au cours de votre carrière que vos programmes ne doivent jamais contenir de chiffres magiques. Nous désignons par chiffre magique, tout nombre (et par extension toute expression) constant présent dans un programme sur lequel un utilisateur donné ne pourrait avoir une influence sur celui-ci sans directement modifier le code source. Malgré le fait que vous soyez convaincus que votre programme ne vous sera jamais utile dans un autre dessein que celui pour lequel qui vous a initialement amené à le créer, vous serez souvent influencé par le contexte dans lequel vous opérez. En plus d'être inefficace, ce genre de pratique vont directement à l'encontre du but premier de la définition de fonction au sens élargi, soit la réutilisation du code. Un moyen simple d'ajouter de la flexibilité à une fonction sera alors la définition de paramètres par défaut. Vous ne pourrez que retirer du positif d'adopter des bonnes pratiques de programmation dès vos débuts dans le domaine. Sur le long terme et à l'aide d'une documentation adéquate de vos programmes (et fonctions), vous ne pourrez que bénéficier de votre rigueur même si cette dernière vous aura fait perdre du temps précieux au début de votre apprentissage.

D'accord, mais qu'entendons-nous par documentation "adéquate"? Trop souvent, la mauvaise documentation d'un programme ne vient pas d'un mal intentionnellement causé par le développeur, mais bien d'une mauvaise éducation sur ce qui caractérise une bonne documentation. Premièrement, le fait qui vous semble le plus évident au moment du procédé de documentation ne le sera pas nécessairement au futur utilisateur. Par le fait même, une documentation devrait être aussi monotone à lire qu'à écrire. Deuxièmement, une documentation ne devrait pas correspondre à un paragraphe sans structure précise ou encore à un enchaînement de faits complètement désorganisés qui n'auront un sens logique que pour celui qui les aura écrit. Troisièmement, un utilisateur s'attendra à retrouver le même type d'information dans la documentation de deux entités différentes qui son toutefois du même genre.

Lorsque nous mettons ces considérations en perspective, on vient donc rapidement à la conclusion qu'une structure standard devrait toujours être utilisée. En plus d'offrir un cadre rigide sur la manière de créer notre documentation, ces outils auront l'avantage de produire des fichiers de référence complets qui posséderont toutes les aspects pratiques d'une documentation professionnelle. Un bon exemple de ce genre d'outils est *Doxygen* [3] qui est très populaire pour

la documentation de script écrit en C/C++. Le principe derrière cet outil a justement été repris pour l'adapter au code R dans le cadre du développement du packaging *roxygen2* [13]. Nous croyons que l'utilisation de ces balises est indispensable même si aucune documentation officielle ne sera jamais générée. Il s'agit simplement d'une excellente habitude de travail et cela vous aidera à structurer votre documentation selon un modèle standard et reconnu par la communauté.



Doxygen et Roxygen, ça respire quoi en hiver ?

Le principe de ces outils est extrêmement rudimentaire. De manière intuitive, nous utiliserons les commentaires afin de faire la documentation de nos programmes. Ce sera toujours le cas ! La principale différence provient de l'introduction de balises qui guideront la présentation de l'information lors de la production de la documentation officielle disponible sous plusieurs formats (html, PDF, \LaTeX , etc.) À titre d'exemple, nous utiliserons la balise `param` pour décrire un paramètre, `return` pour décrire le retour et `examples` pour donner des exemples d'utilisation dans le cadre de la documentation d'une fonction. Dans bien des cas, \LaTeX sera derrière le formatage de cette documentation. Il est bon de savoir que l'écriture d'une telle documentation sera un pré-requis à tous ceux qui seraient tentés de créer un packaging et de le publier sur *Comprehensive \TeX Archive Network (CTAN)*.
<https://cran.r-project.org/doc/manuals/R-exprs.html#Marking-text>

En reprenant les fonctions *fctest1*, *fctest2* et *fctest3*, nous pouvons faire quelques tests en variant le nombre d'arguments envoyés et le comportement résultant.

Code Source 2.13 – Passage d'arguments à une fonction

```
1 > fctest1(3)
2 Error in fctest1(3) : argument "b" is missing, with no default
3 > fctest2(3)
4 [1] 3
5 > fctest2(b=5)
6 Error in fctest2(b = 5) : argument "a" is missing, with no default
7 > fctest3(3)
8 [1] 6
9 > fctest3(3,5)
10 [1] 8
11 > fctest3(b=5)
12 [1] 7
13 > fctest3(b=5,3)
14 [1] 8
15 > fctest3(3,5,4)
16 Error in fctest3(3, 5, 4) : unused argument (4)
```

Comme le montre le [Code Source 2.13](#), nous pourrions admettre comme règle que tout argument ne possédant pas de valeur par défaut doit absolument avoir une valeur d'attribuer lors de l'appel de la fonction. De plus, nous observons que la notion d'argument nommé n'a pas vraiment de signification en R. Ainsi, tous les arguments seront traités de manière positionnelle à moins d'indication contraire par la spécification du nom de l'argument dans l'appel de la fonction. Nous pouvons toutefois remarquer un cas particulier avec l'appel de `fctest2(3)` qui fournira bel et bien la valeur de 3 même si aucune valeur n'a été fournie pour le paramètre `b` et qu'il n'y a aucune valeur par défaut. Ceci s'explique par le fait que R détectera une erreur de valeur manquante qu'au moment de l'exécution plutôt qu'au moment de l'appel de la fonction. Ainsi, puisque cette `fctest2` retournera la valeur de `a` et que son exécution n'ira jamais évaluer la commande `a+b`, R n'aura jamais remarqué l'absence d'une valeur pour `b`. De manière similaire, une erreur sera produite si nous fournissons à `fctest2` qu'une valeur à `b`. L'appel `fctest3(b=5,3)` expose la flexibilité tout aussi incroyable que dangereuse des procédés d'assignation de valeurs lors de l'appel de fonction en R. Cette flexibilité de pouvoir alterné l'ordre pour spécifier les valeurs à nos paramètres vient du fait que R traitera ces deux processus d'assignation de manière indépendante. Dans un premier temps, l'ensemble des valeurs assignées à des paramètres en spécifiant leur nom sera extrait du vecteur de paramètres fourni par l'appel et les valeurs restantes seront attribuées de manière positionnelle sur les arguments n'ayant pas reçu de valeur. Il faut toutefois faire attention à ceci puisque qu'aucune discrimination ne sera effectuée par rapport aux paramètres ayant des valeurs par défaut comme illustré par le [Code Source 2.14](#).

Code Source 2.14 – L'assignation et les valeurs par défaut

```

1 > fctest4 <- function(a,b=3,c,d)
2 + {
3 +   a+b+c+d
4 + }
5 > fctest4(c=2,1,3)
6 Error in fctest4(c = 2, 1, 3) : argument "d" is missing, with no
   default

```

Votre oeil déjà très aguerri a probablement remarqué que les fonctions définies dans le cadre de cette étude de cas utilisaient une technique de retour multiple par l'entremise d'une liste. Cette technique deviendra intéressante dans les cas où une fonction doit effectuer plusieurs sous-calculs correspondant à des entités distinctes d'un calcul donné. À titre d'exemple, bien qu'une fonction soit destinée à exécuter une tâche précise, son utilisateur pourrait parfois être intéressé par la valeur d'un des calculs intermédiaire réalisé. L'avantage de la liste est la possibilité intrinsèque d'attribuer des noms aux différentes valeurs renvoyées. En plus d'ajouter beaucoup de valeur à vos fonctions sans nécessairement rendre le code source beaucoup plus complexe, ce type de retour vous aidera grandement dans le débogage de ces dernières lors de leur développement. Cette technique possède toutefois les désavantages d'imposer une certaine rigueur au

niveau de leur utilisation en obligeant l'utilisateur à récupérer la liste dans un objet et d'ensuite faire l'extraction de la valeur désirée avec l'opérateur `$`. Le [Code Source 2.15](#) offre un exemple concret de cette notion de retour multiple.

Code Source 2.15 – Retour multiple par l'entremise d'une liste

```
1 > ftest5 <- function(a,b=3,c,d)
2 + {
3 +   returningList <- list()
4 +   returningList$value <- a+b+c+d
5 +   returningList$params <- c(a,b,c,d)
6 +   returningList
7 + }
8 > (x <- ftest5(c=2,1,3,4))
9 $value
10 [1] 10
11
12 $params
13 [1] 1 3 2 4
14
15 > x$value
16 [1] 10
```

Le dernier thème qu'il nous reste à aborder au sujet des fonctions consiste en la gestion des erreurs. Lorsque nous voulons définir les limites d'utilisation d'une fonction, il est préférable de parfaitement connaître ce qu'elle ne peut accomplir. Nous définirons ensuite des validations pour s'assurer que nous ne sommes pas de ce genre de cas particuliers et nous renverrons à l'utilisateur un message lui permettant de corriger son appel. La simplicité de R pour générer ce genre de traitement enlève toute raison possible de ne pas le faire. Ce procédé se résume en quatre étapes qui sont :

1. Identifier une limitation du programme
2. Faire la validation nécessaire pour détecter la survenance de cette limitation
3. Composer un message concis fournissant toute l'information nécessaire pour corriger l'appel
4. Soulever l'erreur à l'exécution à l'aide de l'instruction `stop` en fournissant le message composer à l'étape précédente en argument

Code Source 2.16 – Gestion des erreurs sous R

```
1 > ftest6 <- function(a,b)
2 + {
3 +   if(b == 0)
4 +   {
5 +     stop("The value of b is not valid. A division by 0 would be
6 +       generated.")
7 +   }
8 +   a/b
9 + }
```

```
9 > ftest6(3,4)
10 [1] 0.75
11 > ftest6(3,0)
12 Error in ftest6(3, 0) :
13   The value of b is not valid. A division by 0 would be generated.
```



Comment jouer avec le feu sans se brûler ?

Il arrivera parfois où la génération d'erreurs sera inévitable, mais pour lesquelles nous voudrions appliquer un traitement particulier. Nous appelons ce processus la gestion d'exception. Similairement à la majorité des autres langage de programmation, R inclus des méthodes *try/catch* pour palier au problème. Nous avons mis cette technique en pratique dans la dernière partie de cette étude de cas.

<https://stat.ethz.ch/R-manual/R-devel/library/base/html/try.html>

<https://stat.ethz.ch/R-manual/R-devel/library/base/html/try.html>

2.3 Conception de graphiques en R

Avant même d’aborder les fonctionnalités graphiques de R, vous devons préciser qu’elles sont quasi infinies. C’est donc pour cette raison que nous nous contenterons de ne faire qu’une revue globale des types de graphiques qui combleront amplement vos besoins pour faire vos premiers pas. Advenant le cas où ces connaissances ne seront plus suffisantes, il existe énormément d’exemples sur les forums de la communauté pour appaiser votre curiosité.

Pour débiter, la fonction *plot* est de loin la fonction la plus rudimentaire de faire un graphique avec R. Cette fonction ne possède que trois arguments : *x*, *y* et *...*. Naturellement, nous devons fournir des valeurs d’abscisse et d’ordonnée à la commande *plot* via les arguments *x* et *y* et la fonction s’occupera de produire un graphique à points traditionnel. En partant directement du jeu de données *airports.dat*, nous pouvons être tentés d’essayer cette commande en représentant les couples longitude/latitude de chaque aéroport dans le monde. Bien entendu, le résultat obtenu sera peu élégant ne représentant que l’essentiel.

C’est à ce moment que l’argument *...* entre en scène. Nous n’avons pas discuter de ce type d’argument dans la section précédente puisque nous considérons plus intuitif de le présenter à l’aide d’un exemple de son utilisation la plus commune, le passage d’options graphiques au sein de la commande *plot*. Il ne sera toutefois pas rare de retrouver cet argument dans bon nombre de fonction, mais sa nécessité sera souvent moindre que dans le cas de la création de graphique. Cet argument possède la propriété particulière d’absorber tous les paramètres qui seront passés à la fonction et qui n’auront pas été assignés à un argument. Ces mêmes paramètres pourront donc ensuite être transmis à une autre fonction au sein du corps de la fonction.

C’est exactement ce qui se produit dans le cas de la commande *plot* qui enverra tous les paramètres supplémentaires à la fonction *par* étant la commande gérant tous les aspects des graphiques en R. Heureusement, il existera des comportements par défaut pour tous les arguments de cette fonction. Il sera inconcevable et surtout inutile à quiconque d’apprendre l’ordre réel dans lequel ses arguments se présentent. Le passage des paramètres se fera donc en nommant chaque argument sur lequel nous voulons imposer un comportement différent.



par magie !

La fonction *par* vous sera de grands secours à plusieurs reprises. Une utilisation fréquente de cette fonction est de modifier la division de la fenêtre d’affichage de R. En modifiant la valeur de l’argument *mfrow*, nous pourrions ainsi combiner plusieurs graphiques intimement reliés sur la même fenêtre graphique facilitant du même coup leur comparaison.

Par exemple, `par(mfrow = c(2,2))` divisera la fenêtre en 2 lignes et 2 colonnes pour ainsi accueillir 4 graphiques distincts.

C’est précisément ce que nous avons fait dans la deuxième version de notre graphique ([Figure 2.3](#)) en spécifiant le nom des axes (*xlab* et *ylab*) ainsi qu’un titre au graphique (*main*). Nous avons aussi modifié le type de point pour passer de points vides à des points remplis (*pch*) tout en réduisant la taille de ces derniers pour obtenir une meilleure résolution (*cex*). Finalement, nous avons utilisé une police en gras pour le titre du graphique et les axes (*font* et *font.lab*) en plus de venir augmenter la taille de ces derniers (*cex.main* et *cex.lab*). Référez-vous au [Code Source 2.17](#) pour plus de détails.

Code Source 2.17 – Utilisation de la commande *plot*

```

1 plot(airports$longitude, airports$latitude)
2 plot(airports$longitude, airports$latitude, cex = 0.1, xlab="Longitude
   ", ylab="Latitude", main="Spatial Coordinates of All the Airports
   ", pch = 20, font = 2, cex.main = 1.5, font.lab = 2, cex.lab = 1.5)

```

Dans le cas où nous aurions plutôt voulu faire la représentation d’une fonction continue, nous pourrions encore une fois utiliser la commande *plot* en modifiant l’argument *type*. Bien que cette pratique peut nous sembler justifiée, elle pourra jouer de mauvais tours à un utilisateur non-averti. Comme le montre la [Figure 2.7](#), dépendamment de l’espacement des valeurs points calculés, nous pourrions perdre toute l’information sur l’allure réelle de la courbe que nous cherchons à visualiser.

Il sera donc préférable d’utiliser la commande *curve* pour ce genre de tâche afin de simplifier le code source en ne précisant que les extrêmes de l’étendu sur lequel nous voulons tracer la fonction en spécifiant au besoin le nombre de valeur à calculer dans l’intervalle.

Code Source 2.18 – Utilisation de la commande *curve*

```

1 fquad <- function(x, a=2,b=3,c=4)
2 {
3   a*x**2+b*x+c

```

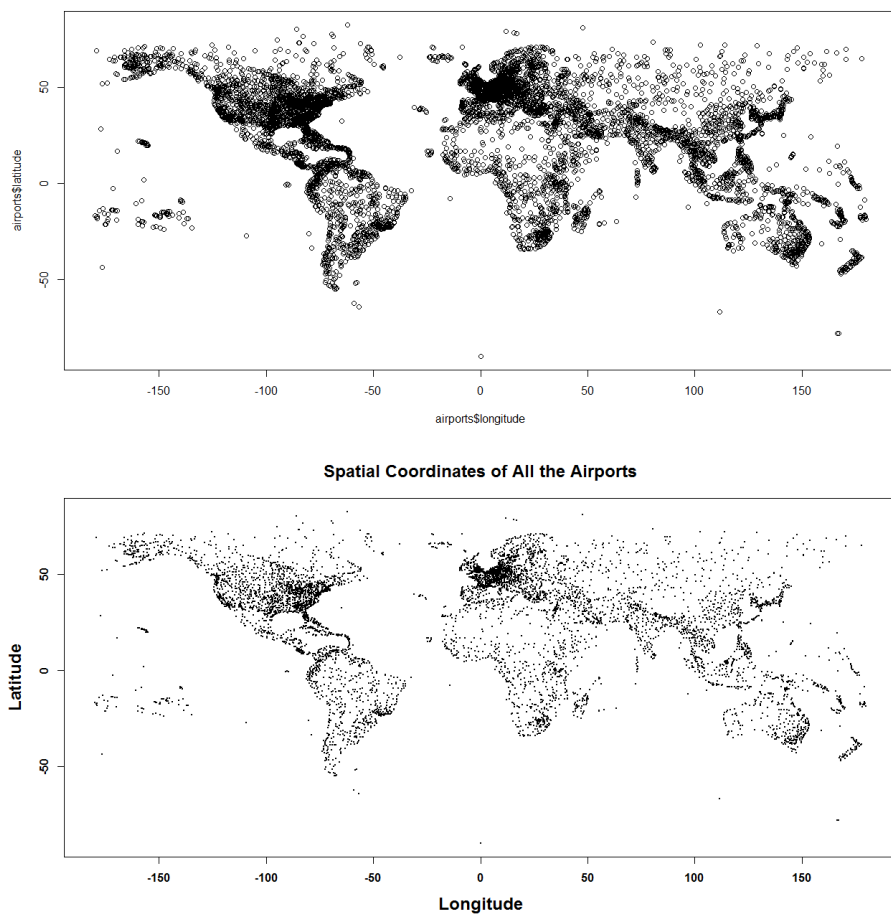


FIGURE 2.6 – Passage de paramètres graphiques à la commande *plot*

```

4 }
5 fquad(2)
6 par(mfrow = c(2,2))
7 plot(x <- seq(-10,10,10),fquad(x,2,3,4),type = "l",ylab = "fquad(x)"
8      ,xlab = "x",main = "dx = 10")
9 plot(x <- seq(-10,10,5),fquad(x,2,3,4),type = "l",ylab = "fquad(x)"
10     ,xlab = "x",main = "dx = 5")
11 plot(x <- seq(-10,10,2),fquad(x,2,3,4),type = "l",ylab = "fquad(x)"
12     ,xlab = "x",main = "dx = 2")
13 plot(x <- seq(-10,10),fquad(x,2,3,4),type = "l",ylab = "fquad(x)",
14     ,xlab = "x",main = "dx = 1")
15
16 par(mfrow = c(1,1))
17 curve(fquad(x),from = -10,to = 10)

```

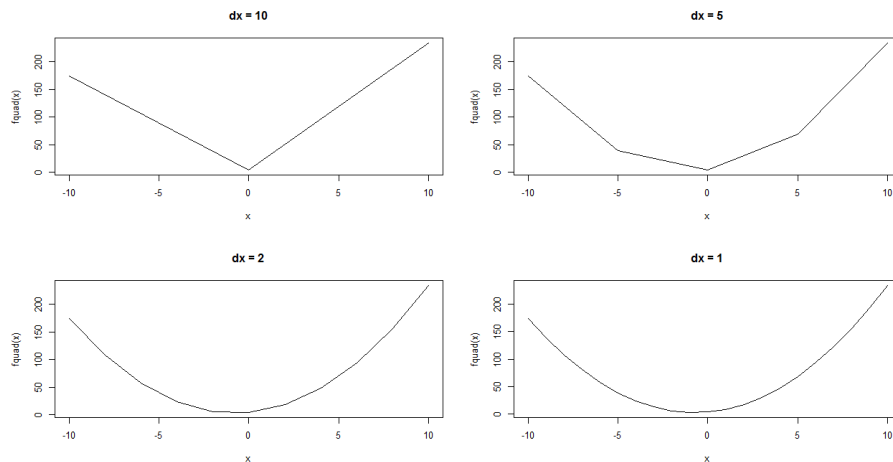


FIGURE 2.7 – Tracer une courbe avec la commande *plot*

Un autre type de graphique fréquemment utilisé dans les analyses statistiques sont les histogrammes. Ces derniers permettent de rapidement avoir une idée globale sur le type de distribution à laquelle nous sommes confrontés. L'argument *breaks* de la commande *hist* est de loin le plus important puisqu'il permettra d'obtenir un visuel beaucoup plus précis de la situation en réduisant la taille des regroupements effectués. En ne spécifiant qu'un seul nombre à cet argument, nous indiquons à R de diviser les données pour obtenir ce même nombre de groupes d'étendue équivalente. Dans le cas où un vecteur de nombre lui serait fourni, R comprendra plutôt qu'il doit regrouper les données en utilisant ces nombres à titre de bornes pour les différents intervalles. Un autre argument bien intéressant est *freq*. Cet argument booléen contrôlera l'affichage de la hauteur des colonnes de l'histogramme. Le nombre d'observations sera affiché si sa valeur est vraie (valeur par défaut) ou sous la forme d'une probabilité sinon.

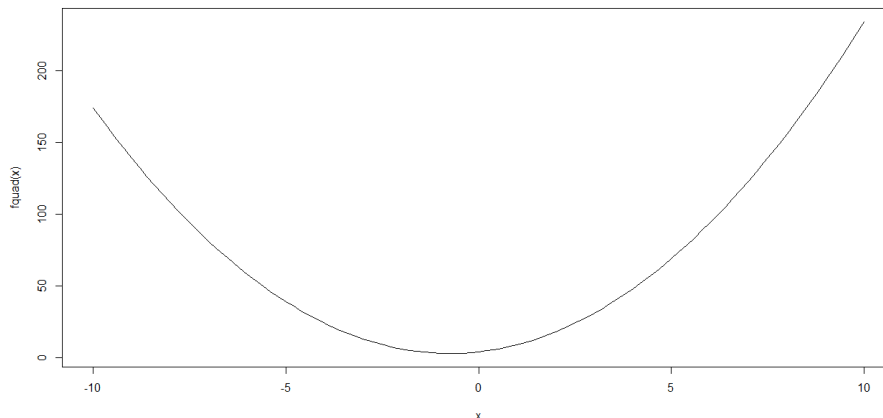


FIGURE 2.8 – Tracer une courbe avec la commande *curve*



Excel et les histogrammes

Si vous êtes habitués de travailler avec *Excel*, vous avez probablement une mauvaise impression de la valeur ajoutée d'utiliser des histogrammes. Ceci vient du fait que *Excel* travaille plutôt avec des graphiques à bâtons. La différence entre ces deux types de graphique réside dans le fait qu'un les colonnes d'un histogramme posséderont à la fois une largeur et une hauteur, tandis que les diagrammes à bâtons ne possèdent qu'une notion de hauteur et sont plutôt destinés à représenter la distribution d'une variable qualitative.

La fonction *density* est aussi très intéressante d'un côté pratique pour estimer la fonction de densité empirique sous-jacente. Cette fonction possède un argument *adjust* avec lequel nous contrôlerons le degré de lissage employé. La valeur par défaut de cet argument est 1 et plus sa valeur sera faible, plus nous nous rapprochons de la distribution discrète, tandis qu'une valeur supérieure aura pour effet de lisser davantage la fonction obtenue.

Bon nombre des fonctionnalités graphiques de R peuvent être combinées au sein d'un même graphique. Il s'agira d'un comportement natif dans certains cas (les commandes *points* et *lines*) ou d'un comportement induit par l'argument *add* comme c'est possible de le faire avec *curve*. Il sera possible de facilement tracer la fonction de densité renvoyée par *density* grâce à la commande *lines*.

La commande *abline* simplifiera grandement l’affichage de fonctions linéaires. L’utilisation de celle-ci pourra se faire de trois manières différentes. La première consiste à spécifier les arguments *a* et *b* pour produire la représentation d’une droite d’équation $y = ax + b$. La deuxième permettra plutôt de tracer une droite d’équation $y = h$ en attribuant une valeur à l’argument *h*. La dernière et non la moindre qui est, selon moi, la plus commode d’entre toutes permet de créer des droites d’équation $x = v$. L’ajout de ce genre de droites permettra de faire ressortir des valeurs d’abscisses ayant une signification particulière dans le cadre de votre analyse.

Certains autres fonctions vous permettront de rajouter de l’information afin de faciliter la lecture de vos graphiques. Parmi ces fonctions, la plus importante sera *legend* qui comme son nom l’indique, s’occupera de générer une légende au graphique que nous venons de produire. Cette fonction est tout autant paramétrisable que le graphique sous-jacent. Nous pouvons tout de même identifier des arguments plus communs que d’autres. L’argument *bty* permettra de supprimer l’encadrement de la légende en lui attribuant la valeur "n". Nous préciserons aussi un type de points avec *pch* ou un type de ligne avec *lty* sur lesquels nous pourrions affecter la même couleur que la courbe correspondante à l’aide de *col*. La fonction *mtext* s’occupera plutôt d’ajouter du texte à des endroits précis sur le graphique pour noter des observations ou ajouter des explications sur des aspects qui nous semble plus surprenant.

L’ensemble des points discutés ci-dessus ont été repris dans le [Code Source 2.19](#) pour produire la [Figure 2.9](#).

Code Source 2.19 – *hist*, *density*, *lines*, *abline*, *legend* et *mtext*

```

1 Altitude <- as.numeric(paste(airportsCanada$altitude))
2 hist(Altitude)
3 hist(Altitude, xlim = c(0,5000))
4 hist(Altitude, xlim = c(0,5000), breaks = 100)
5 hist(Altitude, xlim = c(0,5000), breaks = 100, freq = FALSE, col = "
  gray", border = grey(0.8), font = 2, font.lab = 2)
6 lines(density(Altitude, adjust = 4), lwd = 2, col = "blue")
7 lines(density(Altitude, adjust = 1), lwd = 2, col = "purple")
8 lines(density(Altitude, adjust = 0.25), lwd = 2, col = "red")
9 altitudeAvg <- round(mean(Altitude), 1)
10 abline(v = altitudeAvg, lwd = 2)
11 legend(2500, 0.0015, legend = c("4", "1", "0.25"), title = "Density
  Adjustment \n Factor", col = c("blue", "purple", "red"), bty = "n",
  title.col = "black", lty = 1, lwd = 3, y.intersp = 0.5, cex = 1.25)
12 mtext(paste("Average: \n", altitudeAvg), at = altitudeAvg, cex = 0.75)

```

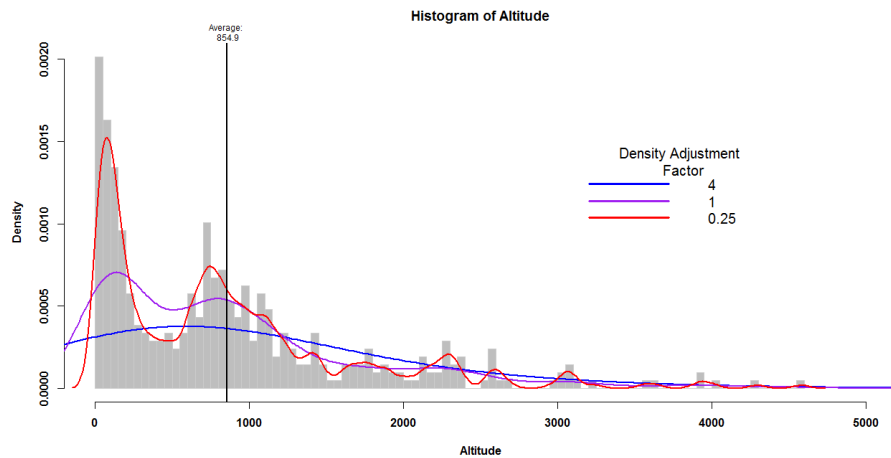


FIGURE 2.9 – Distribution des altitudes des aéroports canadiens



Vers l'infini et plus loin encore !

Vous aurez compris qu'il ne s'agit que d'un TRÈS bref aperçu des capacités graphiques de R. Il existe des structures standard pour générer d'autres types de graphique tels que les diagrammes en pointes de tarte (*pie*) ou encore les boîtes à moustaches (*boxplot*). Certains d'entre vous trouveront peut-être que la génération de graphique est un processus lent et ardu, mais il s'agit ici du coût à payer pour avoir autant de flexibilité. Ces mêmes personnes seront toutefois heureuse d'apprendre que plusieurs paquets intègrent des modules de visualisation standard pour les objets qui leur sont propres. Il serait par contre un peu prétentieux de définir des modifier les options d'affiche par défaut des objets dont l'existence ne dépend aucunement de leur utilisation.

2.4 Outils d'analyse statistique en R

Un des aspects du langage R sur lequel sa réputation s'est bâtie est la variété des outils statistiques qu'il place à la disposition de son utilisateur. Sans même avoir à importer une quelconque librairie à partir de CRAN, plusieurs distributions statistiques sont disponibles. La [Tableau 2.4](#) fait la revue des ces distributions et de leur identifiant R correspondant. [9]

D'autres distributions deviendront aussi disponible via des paquets dédiés à

Distribution	identifiant R
Bêta	beta
Binomiale	binom
Binomiale négative	nbinom
Chi Deux	chisq
Exponentielle	exp
Fisher	f
Gamma	gamma
Géométrique	geom
Hypergéométrique	hyper
Normale	norm
Poisson	pois
Student	t
Uniforme	unif
Weibull	weibull

TABLE 2.1 – Liste des distributions statistiques disponibles en R

cette fin. Le paquetage *actuar* donne accès à plusieurs distributions supplémentaires communément utilisées en actuariat. La distribution Pareto en est un bon exemple.

Un aspect particulièrement intéressant de ces implementation de distribution statistique (qu'elles soient disponibles par défaut en R ou via l'importation d'un paquetage) est la constance dans la structure de ces fonctions. Pour chacune des distribution, nous retrouverons en outre les trois fonctions qui suivent :

$d\langle ID_R \rangle$	Calcule la valeur de la fonction de densité de la distribution ayant l'identifiant R $\langle ID_R \rangle$
$p\langle ID_R \rangle$	Calcule la valeur de la fonction de répartition de la distribution ayant l'identifiant R $\langle ID_R \rangle$
$q\langle ID_R \rangle$	Renvoie le quantile associé à la valeur fournie en argument selon la fonction de répartition de la distribution ayant l'identifiant R $\langle ID_R \rangle$
$r\langle ID_R \rangle$	Permet de générer des valeurs aléatoires suivant la distribution ayant l'identifiant R $\langle ID_R \rangle$

De plus, les arguments de ces fonctions se présenteront toujours sous le même format. Nous devons soit fournir la valeur à laquelle nous voulons évaluer la fonction ou encore un nombre d'observation à générer dans le cas des fonctions préfixée par "r" et les paramètres de la loi utilisée. À des fins d'optimisation des performances, le logarithme de ces fonctions sera souvent nécessaire et c'est ce qui explique la présence de l'argument *log*.⁵ Finalement, nous serons parfois

5. Plusieurs propriétés statistiques découlent du logarithme des fonctions de densité et de répartition tel que la fonction génératrice de moments pour ne nommer que cette dernière.

intéressé par la fonction de survie d'une distribution donnée correspondant au complément de la fonction de répartition. En attribuant la valeur faux à l'argument *lower.tail*, les fonctions préfixée par "p" renverront ainsi la valeur de la fonction de survie. Un exemple d'utilisation de ces fonctions est présenté par le [Code Source 2.20](#).

Code Source 2.20 – Fonctions relatives à la distribution Normale

```

1 > set.seed(2017)
2 > mean <- 6
3 > sd <- 2
4 > x <- 0:12
5 > dnorm(x, mean, sd)
6 [1] 0.002215924 0.008764150 0.026995483 0.064758798
7 [5] 0.120985362 0.176032663 0.199471140 0.176032663
8 [9] 0.120985362 0.064758798 0.026995483 0.008764150
9 [13] 0.002215924
10 > pnorm(x, mean, sd)
11 [1] 0.001349898 0.006209665 0.022750132 0.066807201
12 [5] 0.158655254 0.308537539 0.500000000 0.691462461
13 [9] 0.841344746 0.933192799 0.977249868 0.993790335
14 [13] 0.998650102
15 > r <- seq(0, 1, 0.1)
16 > qnorm(r, mean, sd)
17 [1] -Inf 3.436897 4.316758 4.951199 5.493306
18 [6] 6.000000 6.506694 7.048801 7.683242 8.563103
19 [11] Inf
20 > rnorm(10, mean, sd)
21 [1] 8.868403 5.845416 7.478274 2.482791 5.860350
22 [6] 6.903811 2.083267 5.996951 5.469328 9.126445

```

Ceux qui sont familiers avec les distributions statistiques auront remarqués qu'à l'aide des fonctions décrites ci-dessus nous aurons donc deux manières de générer des nombres aléatoires. La première qui est aussi la plus évidente sera d'utiliser les fonctions préfixée avec "r". La seconde utilisera le théorème de la réciproque consistant à générer des valeurs aléatoires suivant une loi uniforme de paramètre $a := 0$ et $b := 1$ pour ensuite trouver le quantile correspondant de la fonction de répartition de la loi pour laquelle nous voulons générer des nombres aléatoires grâce aux fonctions préfixée par "q". Ces deux techniques sont mise à profit dans le [Code Source 2.21](#).

Code Source 2.21 – Génération de nombres aléatoires

```

1 > y1 <- rnorm(1000, mean, sd)
2 > summary(y1)
3      Min.   1st Qu.   Median     Mean   3rd Qu.    Max.
4  0.07041  4.70800   6.02800   6.06200   7.35500  12.59000
5 > sd(y1)
6 [1] 1.96455
7 > r <- runif(1000)
8 > y2 <- qnorm(r, mean, sd)
9 > summary(y2)

```

```

10      Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
11 -0.1347  4.7670   6.0830   6.0910  7.5070  12.2400
12 > sd(y2)
13 [1] 1.966951

```



Théorème de la réciproque

Ce sont les 4 propriétés des fonctions de répartition qui rendent possible l'application du théorème de la réciproque. Ces propriétés sont définies comme suit (où F désigne la fonction de répartition d'une variable aléatoire X quelconque) :

1. F_X est croissante
2. Elle est partout continue à droite
3. $\lim_{x \rightarrow -\infty} F_X(x) = 0$
4. $\lim_{x \rightarrow \infty} F_X(x) = 1$

Étant donné que ces propriétés seront toujours respectées pour toute fonction de répartition, nous pourrions appliquer cette méthode peu importe la distribution qu'elle soit clairement définie ou non! https://fr.wikipedia.org/wiki/Fonction_de_r%C3%A9partition#Th.C3.A9or.C3.A8me_de_la_r.C3.A9ciproque

En présence de données empiriques, la première étape d'une analyse statistique sera de dresser le portrait statistique de ces données. Nous avons déjà parlé de la fonction *summary* à la [sous-section 2.1.1](#). Nous rajouterons ici les fonctions *mean* et *sd* retournant respectivement la moyenne et l'écart-type d'un jeu de données empiriques comme nous l'avons fait montré dans le [Code Source 2.21](#).

Afin de valider l'ajustement d'une distribution donnée sur les données empiriques, nous serons souvent contraint à identifier les fonctions de densité et de répartition sous-jacentes. Il existe plusieurs façons de faire. Celle qui nous semble toutefois la plus pertinente et polyvalente exploite le comportement de la fonction *ecdf*. Cette dernière permet de construire une fonction de répartition empirique à partir des observations fournies en argument. Nous pouvons ensuite construire une fonction de densité empirique en évaluant cette fonction de répartition à deux points autour de la valeur désirée et en divisant ensuite le résultat par la largeur de l'intervalle évalué. Les instructions permettant de construire ces fonctions sont fournies par [Code Source 2.22](#).

Code Source 2.22 – Fonctions de densité et de répartition empiriques

```

1 empCDF <- ecdf(compData$weight)
2 empPDF <- function(x, delta=0.01)
3 {
4   (empCDF(x+delta/2)-empCDF(x-delta/2))/delta
5 }

```

En plus de dresser le portrait statistique du jeu de données, nous pourrions être tentés d’effectuer des tests statistiques sur ces mêmes données. Parmi les tests disponibles, nous retrouvons notamment :

- ▶ Test de normalité (Test de Shapiro-Wilk)
- ▶ Test de comparaison de deux variances (Test F)
- ▶ Test de Student
- ▶ Test du Khi carré
- ▶ Test de Wilcoxon
- ▶ ANOVA (Analyse de variance)
- ▶ Test de corrélation

Il n’est toutefois pas indispensable de connaître l’utilité de tous ces tests, les situations dans lesquelles ils devraient être utilisés ni la mécanique mathématique sous-entendue puisque la plupart des méthodes statistiques inclueront déjà les appels nécessaires de ceux-ci. Ce sera le cas de la fonction *lm* comme nous le verrons plus loin. [15]

Dans le cadre de notre étude de cas, nous avons performé les tests du Khi carré et de corrélation afin de s’assurer que les variables explicatives du poids et de la distance soient indépendantes et sans corrélation. Dans le cas où ce genre de phénomène serait apparu entre nos variables, nous aurions été contraint d’utiliser des modèles de régression plus complexes tel que les modèles linéaires généralisés.

Lorsque nous effectuons un test statistique, nous cherchons toujours à répondre à une question binaire représentée sous la forme de deux hypothèses H_0 et H_1 complémentaires. Une valeur nommée la *p-value* sera ensuite calculée en acceptant l’hypothèse H_0 comme vraie. Cette valeur correspondra à la probabilité d’observer un résultat équivalent ou encore plus extrême du test que nous venons d’exécuter en considérant l’hypothèse nulle comme vraie. En d’autres mots, cette valeur nous indiquera la probabilité de se tromper en rejetant l’hypothèse nulle en considérant l’hypothèse nulle comme vraie initialement. Ainsi, à partir du moment où la *p-value* sera inférieure au seuil de crédibilité que l’on s’était fixé (habituellement 5%), nous considérerons l’hypothèse nulle comme fausse.

Dans le cas du test du Khi carré, l’hypothèse nulle suppose que les deux distributions sont indépendantes. Le test de corrélation suppose tant qu’à lui que la valeur théorique de corrélation est équivalente à 0. Comme nous pouvons le voir avec le [Code Source 2.23](#), nous pouvons ne pouvons pas rejeter ces deux hypothèses.

Code Source 2.23 – Tests d’indépendance et de corrélation entre distributions

```
1 > weightsBinded <- as.numeric(cut(compData$weight,25))
2 > distancesBinded <- as.numeric(cut(compData$distance,25))
```

```

3 > contingencyTable <- table(weightsBinded, distancesBinded)
4 > chisq.test(contingencyTable)
5
6 Pearson's Chi-squared test
7
8 data: contingencyTable
9 X-squared = 248.38, df = 391, p-value = 1
10
11 Warning message:
12 In chisq.test(contingencyTable) :
13 Chi-squared approximation may be incorrect
14 > contingencyTable <- rbind(contingencyTable[1:4,], colSums(
15 contingencyTable[5:18,]))
16 > contingencyTable <- cbind(contingencyTable[,1:14], rowSums(
17 contingencyTable[,15:24]))
18 > (independencyTest <- chisq.test(contingencyTable))
19
20 Pearson's Chi-squared test
21
22 data: contingencyTable
23 X-squared = 72.814, df = 56, p-value = 0.06495
24
25 > cor.test(compData$weight, compData$distance, method = "pearson")
26
27 Pearson's product-moment correlation
28
29 data: compData$weight and compData$distance
30 t = -0.7801, df = 99998, p-value = 0.4353
31 alternative hypothesis: true correlation is not equal to 0
32 95 percent confidence interval:
33 -0.008664731 0.003731121
34 sample estimates:
35 cor
36 -0.0024669

```

Il est pertinent de remarquer que le test du Khi carré possède des limitations importantes dans le cas de distributions devenant un peu trop clairsemée. Ce test nécessite une efficacité statistique d'au minimum 5 observations à toutes les intersections des deux variables catégoriques. C'est pour cette même raison que nous combinons les dernières lignes et colonnes de la table de contingences. Malgré tout, le test offre toujours une *p-value* d'environ 6% ce qui reste supérieur à notre seuil de 5% et nous ne pouvons donc pas rejeter notre hypothèse nulle. Dans le cas du test de corrélation, nous voyons que la valeur 0 est comprise dans notre intervalle de confiance autour de la valeur de corrélation empirique déterminée, ce qui nous permet d'affirmer qu'aucune corrélation n'existe entre ces deux variables. La *p-value* de 43% aurait été suffisante pour arriver à la même conclusion.

Pour terminer cette section, jettons un coup d'oeil à la régression linéaire qui fut accomplie dans le but de modéliser la distribution ayant menée à générer les données ([Code Source 2.24](#)).

Code Source 2.24 – Régression linéaire sur données empiriques

```

1 > profitMargin <- 1.12
2 > avgTaxRate <- sum(table(airportsCanada$province)*as.numeric(paste
   (taxRates$taxRate)))/length(airportsCanada$province)
3 > compModel <- lm(price/(profitMargin*avgTaxRate) ~ distance +
   weight, compData)
4 > summary(compModel)
5
6 Call:
7 lm(formula = price/(profitMargin * avgTaxRate) ~ distance + weight,
8     data = compData)
9
10 Residuals:
11      Min       1Q   Median       3Q      Max
12 -30.7903  -4.6585   0.0305   4.6462  29.9563
13
14 Coefficients:
15             Estimate Std. Error t value Pr(>|t|)
16 (Intercept)  3.227e+01  7.509e-02  429.7   <2e-16 ***
17 distance     2.820e-02  9.206e-05   306.4   <2e-16 ***
18 weight       7.252e-01  9.479e-03    76.5   <2e-16 ***
19 ---
20 Signif. codes:
21 0 '***', 0.001 '**', 0.01 '*', 0.05 '.', 0.1 ' ', 1
22
23 Residual standard error: 6.89 on 99997 degrees of freedom
24 Multiple R-squared:  0.499, Adjusted R-squared:  0.499
25 F-statistic: 4.98e+04 on 2 and 99997 DF, p-value: < 2.2e-16

```

L'appel de la fonction *lm* est assez rudimentaire. Il suffit de fournir une formule de regression contenant les variables explicatives avec lesquelles nous tenons à faire la regression et nous spécifions le nom de la table contenant ces variables. Nous remarquons ici la technique du retour multiple abordée à la 2.2. Nous voyons aussi que pour chaque coefficient un test de Student a été effectué pour déterminer à quel point l'estimé était significativement différent de 0. D'autre part, le test de Fisher permet de savoir s'il existe réellement une relation entre les variables explicatives choisies et la variable réponse analysée. [11]

Lorsque l'on compare les valeurs réellement utilisées dans le A et les coefficients estimés, nous voyons que ces derniers sont très proches les uns des autres. La Tableau 2.2 fait la revue de ces valeurs.

Variable	Valeur réelle	Valeur estimée
distance	0.0275	0.0282
poids	0.7	0.7252

TABLE 2.2 – Comparaison entre les coefficients réels et estimés par régression linéaire



Lire des tables directement sur le web

Afin de récupérer les valeurs sur les niveaux de taxe pour chaque province canadienne, nous avons pris l'initiative de passer directement via le web. Cette méthode possède l'avantage de se mettre à jour directement avec l'information la plus récente si la structure de la page n'est pas modifiée. Afin de parvenir à ce résultat, les paquetages *XML*, *RCurl* et *rlist* fournissent des fonctions permettant d'interpréter la structure *HTML* d'une page web spécifiée par le passage du chemin *url* en argument à la fonction *readHTMLTable* pour y détecter les occurrences de balises du genre *<table>*.

http://web.mit.edu/~r/current/arch/i386_linux26/lib/R/library/XML/html/readHTMLTable.html

2.5 Ajustement de distributions statistiques sur données empiriques

2.6 Simulation et analyse de rentabilité

Conclusion

Bibliographie

- [1] A quoi correspondent les extensions *.shp, *.dbf, *.prj, *.sbn, *.sbx et *.shx? <http://www.portailsig.org/content/quoi-correspondent-les-extensions-shp-dbf-prj-sbn-sbx-et-shx>.
- [2] CSV vs. Delimited Flat Files : How to Choose. <http://www.thoughtspot.com/blog/csv-vs-delimited-flat-files-how-choose>.
- [3] Doxygen. <http://www.stack.nl/~dimitri/doxygen/>.
- [4] Font Awesome - The iconic font and CSS toolkit. <http://fontawesome.io/>.
- [5] GitHub. <https://github.com/>.
- [6] Introduction à la programmation en R. https://cran.r-project.org/doc/contrib/Goulet_introduction_programmation_R.pdf.
- [7] OpenFlights. <https://openflights.org/data.html>.
- [8] Package 'leaflet'. <https://cran.r-project.org/web/packages/leaflet/leaflet.pdf>.
- [9] Probabilités et Statistique avec R. <http://ljk.imag.fr/membres/Bernard.Ycart/mel/dr/node7.html>.
- [10] Projection (Système de). <http://www.emse.fr/tice/uved/SIG/Glossaire/co/Projection.html>.
- [11] Quick Guide : Interpreting Simple Linear Model Output in R. <http://feliperego.github.io/blog/2015/10/23/Interpreting-Model-Output-In-R>.
- [12] R à Québec 2017. <http://raquebec.ulaval.ca/2017/programme-r-a-quebec-2017>.
- [13] roxygen2. <http://roxygen.org/>.
- [14] Structured Query Language (SQL). https://fr.wikipedia.org/wiki/Structured_Query_Language.
- [15] Tests statistiques avec R. <http://www.sthda.com/french/wiki/tests-statistiques-avec-r>.
- [16] Statistics Canada. Boundary Files, Reference Guide. <http://www.statcan.gc.ca/pub/92-160-g/92-160-g2011002-eng.htm>.
- [17] Eric Muller. A shapefile of the TZ timezones of the world. <http://efele.net/maps/tz/world/>.

Annexe A

Code source du projet

Cette annexe présente les codes sources constituant l'ensemble du projet. Ceux-ci se divisent sous la forme de 6 parties correspondant aux différents thèmes abordés dans le présent document.

Code Source A.1 – Benchmark.R

```
1 # coding: utf-8
2 # CaseStudyRQuebec2017
3 # Authors : David Beauchemin & Samuel Cabral Cruz
4
5 # Source code for the creation of the benchmark.csv file
6
7 # Setting working directory properly
8 setwd('C:/Users/Samuel/Documents/ColloqueR/Dev')
9 getwd()
10 setwd("..")
11 (path <- getwd())
12
13 # Parameters of the simulation
14 n <- 100000
15 x <- matrix(c(runif(2*n)), ncol = 2, byrow = TRUE)
16
17 # Generate weights with a LogNormal distribution
18 mul <- log(3000)
19 sigma1 <- log(1.8)
20 exp(mul+sigma1**2/2)
21 exp(2*mul+4*sigma1**2/2)-exp(mul+sigma1**2/2)**2
22 weights <- round(qlnorm(x[,1], mul, sigma1)/1000,3)
23 hist(weights, breaks = 100, freq=FALSE)
24 mean(weights)
25 max(weights)
26
27 # Generate the errors on the weights
28 weightsTarifParam <- 0.7
29 weightsCost <- weights*weightsTarifParam
30 weightsError <- rnorm(n, mean(weightsCost), sd(weightsCost))
31 max(weightsError)
```

```

32 min(weightsError)
33 weightsFinalPrice <- weightsCost+weightsError
34 mean(weightsFinalPrice)
35 min(weightsFinalPrice)
36 var(weightsFinalPrice)
37
38 # Generate the distance with a LogNormal distribution
39 # routesCanada
40 # routesIATA <- cbind(paste(routesCanada$sourceAirport),paste(
    routesCanada$destinationAirport))
41 # routesDistance <- apply(routesIATA, 1, function(x) airportsDist(x
    [1],x[2])$value)
42 # max(routesDistance)
43 # mean(routesDistance)
44 mu2 <- log(650)
45 sigma2 <- log(1.4)
46 (distances <- round(qlnorm(x[,2],mu2,sigma2)))
47 hist(distances,breaks = 100,freq=FALSE)
48 mean(distances)
49 max(distances)
50
51 # Generate the errors on the distances
52 distancesTarifParam <- 0.0275
53 distancesCost <- distances*distancesTarifParam
54 distancesError <- rnorm(n,mean(distancesCost),sd(distancesCost))
55 distancesFinalPrice <- distancesCost+distancesError
56 mean(distancesFinalPrice)
57 var(distancesFinalPrice)
58 max(distancesFinalPrice)
59 min(distancesFinalPrice)
60
61 # Generate total price
62 baseCost <- 10
63 # taxRate <- sum(table(airportsCanada$province)*as.numeric(paste(
    taxRates$taxRate)))/length(airportsCanada$province)
64 taxRate <- 1.082408
65 profitMargin <- 1.15
66 (totalCost <- round((baseCost + weightsFinalPrice +
    distancesFinalPrice)*profitMargin*taxRate,2))
67 mean(totalCost)
68 var(totalCost)
69 max(totalCost)
70 min(totalCost)
71
72 # Export to csv format
73 (dataExport <- cbind(weights,distances,totalCost))
74 colnames(dataExport) <- c("Poids (Kg)","Distance (Km)","Prix (CAD $
    )")
75 write.csv(dataExport,paste(path,"/Reference/benchmark.csv",sep=''),
    row.names = FALSE, fileEncoding = "UTF-8")

```

Code Source A.2 – CaseStudyDevQ1.R

```

1 # coding: utf-8
2 # CaseStudyRQuebec2017
3 # Authors : David Beauchemin & Samuel Cabral Cruz

```

```

4
5 ##### Setting working directory properly #####
6 setwd('C:/Users/Samuel/Documents/ColloqueR/Dev')
7 getwd()
8 setwd('..')
9 (path <- getwd())
10 set.seed(31459)
11
12 ##### Question 1 – Data extraction, processing, visualization and
13   analysis #####
14
15 # 1.1 – Database extraction of airports.dat, routes.dat and
16   airlines.dat.
17 airports <- read.csv("https://raw.githubusercontent.com/jpatokal/
18   openflights/master/data/airports.dat", header = FALSE, na.
19   strings=c("\N",""))
20 routes <- read.csv("https://raw.githubusercontent.com/jpatokal/
21   openflights/master/data/routes.dat", header = FALSE, na.strings
22   =c("\N",""))
23 airlines <- read.csv("https://raw.githubusercontent.com/jpatokal/
24   openflights/master/data/airlines.dat", header = FALSE, na.
25   strings=c("\N",""))
26
27 # 1.2 – Columns names assignation base on the information
28   available on the website.
29 colnames(airports) <- c("airportID", "name", "city", "country", "
30   IATA", "ICAO",
31   "latitude", "longitude", "altitude", "
32   timezone", "DST",
33   "tzFormat", "typeAirport", "Source")
34 colnames(routes) <- c("airline", "airlineID", "sourceAirport", "
35   sourceAirportID",
36   "destinationAirport", "destinationAirportID", "
37   codeshare",
38   "stops", "equipment")
39 colnames(airlines) <- c("airlineID", "name", "alias", "IATA", "ICAO", "
40   Callsign", "Country", "Active")
41
42 # 1.3 – Keeping the Canada information of the dataset
43 airportsCanada <- airports[airports$country=='Canada',]
44 airportsCanada2 <- subset(airports, country == 'Canada')
45 all.equal(airportsCanada, airportsCanada2)
46
47 # 1.4 – Extraction of the general information about the
48   distributions of the variables in the dataset and
49   understanding of the signification of those variables and the
50   different modalities they can take.
51
52 # Those are the principal R function to easily visualize
53   information.
54 View(airportsCanada)
55 head(airportsCanada)
56 summary(airportsCanada)
57 nbAirportCity <- table(airportsCanada$city)
58 (nbAirportCity <- sort(nbAirportCity, decreasing=TRUE))[1:10]
59

```

```

44 # The relevance of the data.
45 # 1.5 – Correct the modalities of the variables and make a
      selection of those we seem useful for the rest of the treatment
      .
46 # We observe that the variables typeAirport and Source are useless
      in our situation since we only use information on air transport
      .
47 # A similar reasoning is applicable for the country variable which
      will only have the modality Canada.
48 airportsCanada <- subset(airportsCanada, select = -c(country,
      typeAirport, Source ))
49
50 # As seen in the summary, we don't have the IATA for 27 airports.
51 airportsCanada[is.na(airportsCanada$IATA),c("airportID","name","
      IATA","ICAO")]
52 subset(airportsCanada, is.na(IATA), select = c("airportID","name","
      IATA","ICAO"))
53
54 # The first option, is to simply ignore these airports in the rest
      of the analysis.
55 # However, all these airports have a well-defined ICAO code which
      will allow a default value to be assigned.
56 # Since 82% of the IATA is the last three characters of the ICAO, we
      will simply use the derivate IATA from the ICAO.
57 sum(airportsCanada$IATA==substr(airportsCanada$ICAO,2,4),na.rm =
      TRUE)/sum(!is.na(airportsCanada$IATA))
58
59 # We are now able to fill the missing IATA and we will delete the
      ICAO since it will be useless.
60 airportsCanada$IATA <- as.character(airportsCanada$IATA)
61 # We fill the NA with the substring ICAO.
62 airportsCanada$IATA[is.na(airportsCanada$IATA)] <- substr(
      airportsCanada$ICAO[is.na(airportsCanada$IATA)],2,4)
63 airportsCanada$IATA <- as.factor(airportsCanada$IATA)
64 airportsCanada <- subset(airportsCanada, select = - ICAO)
65 View(airportsCanada)
66
67 # Finally, we are missing more than fifty time zone.
68 missingTZ <- airportsCanada[is.na(airportsCanada$timezone),]
69
70 # Since the TZ depend only on the geographical position, two
      options are available to us :
71 # 1) Deduce the information from other close airport;
72 # 2) Locate the real time zone by mapping tools.
73 # We will use the second option which may seem more complex but
      with the proper tools become more easy and accurate.
74
75 # install.packages("sp")
76 # install.packages("rgdal")
77 library(sp)
78 library(rgdal)
79 tz_world.shape <- readOGR(dsn=paste(path,"/Reference/tz_world",sep=
      ""),layer="tz_world")
80 unknown_tz <- airportsCanada[is.na(airportsCanada$tzFormat),c("
      airportID","name","longitude","latitude")]
81 sppts <- SpatialPoints(unknown_tz[,c("longitude","latitude")])
82 proj4string(sppts) <- CRS("+proj=longlat")

```

```

83 sppts <- spTransform(sppts, proj4string(tz_world.shape))
84 merged_tz <- cbind(unknown_tz, over(sppts, tz_world.shape))
85
86 # We can also note that we only have information derived from the
      province in which each airport is located with the city.
87 # Since we want to apply taxes by province in our situation, we
      will need a better data to access this informaiton.
88 # We will again use mapping techniques to extract the province as a
      function of the x and y coordinates.
89 prov_terr.shape <- readOGR(dsn=paste(path, "/Reference/prov_terr",
      sep=""), layer="gpr_000b11a_e")
90 unknown_prov <- airportsCanada[, c("airportID", "city", "longitude", "
      latitude")]
91 sppts <- SpatialPoints(unknown_prov[, c("longitude", "latitude")])
92 proj4string(sppts) <- CRS("+proj=longlat")
93 sppts <- spTransform(sppts, proj4string(prov_terr.shape))
94 merged_prov <- cbind(airportsCanada, over(sppts, prov_terr.shape))
95
96 # install.packages("sqldf")
97 library(sqldf)
98 airportsCanada <- sqldf("
99     select
100     a.*,
101     coalesce(a.tzFormat, b.TZID) as tzMerged,
102     c.PRENAME as provMerged
103 from airportsCanada a
104 left join merged_tz b
105 on a.airportID = b.airportID
106 left join merged_prov c
107 on a.airportID = c.airportID
108 order by a.airportID")
109 airportsCanada <- data.frame(as.matrix(airportsCanada))
110
111 # Since the timezone, DST and city are now useless, we remove them
      from the dataset.
112 # Plus, we withdraw tzFormat because it's incomplet and we will use
      the tzmerge data to replace will a complete data.
113 airportsCanada <- subset(airportsCanada, select = -c(timezone, DST,
      tzFormat, city))
114 summary(airportsCanada)
115
116 # install.packages("plyr")
117 library(plyr)
118 airportsCanada <- rename(airportsCanada, c("tzMerged"="tzFormat", "
      provMerged"="province"))
119 summary(airportsCanada)
120 routesCanada <- sqldf("
121     select *
122 from routes
123 where sourceAirportID in (select distinct airportID
124                           from airportsCanada)
125     and destinationAirportID in (select distinct airportID
126                                  from airportsCanada)")
127 routesCanada <- data.frame(as.matrix(routesCanada))
128
129 # This code will give the same result :
130 # x <- routesCanada[!is.na(match(routesCanada$sourceAirportID,

```

```

    airportsCanada$airportID)) &
131 #       !is.na(match(routesCanada$destinationAirportID ,
    airportsCanada$airportID)),],
132 # routesCanada <- routesCanada[!is.na(match(routesCanada$
    sourceAirport , airportsCanada$IATA)) &
133 #       !is.na(match(routesCanada$destinationAirport ,
    airportsCanada$IATA)),],
134
135 summary(routesCanada)
136 unique(routesCanada$airline)
137 unique(routesCanada[,c("airline", "airlineID")])
138 unique(routesCanada$airlineID)
139 summary(airlines)
140 (airlinesCanada <- sqldf("
141     select *
142     from airlines
143     where IATA in (select distinct airline
144                     from routesCanada)")
145 routesCanada[is.na(routesCanada$airlineID),]
146 unique(routesCanada$airlineID)
147 unique(routesCanada[is.na(routesCanada$airlineID),]$airline)
148 summary(routesCanada$stops)
149 # As we can see, there are only two flights that are not direct.
150 # For the sake of simplicity, we will consider all flights as
    direct flights.
151 # Moreover, the notion of codeshare will not be useful since the
    delivery of
152 # merchandise can be done as much through an air agency as by
    private flight.
153 # In conclusion, we get rid of these variables.
154 routesCanada <- subset(routesCanada, select = -c(codeshare, stops))
155 summary(routesCanada)
156
157 # 1.6 – Create a map showing the different airports on a map of
    Canada.
158 # install.packages("ggmap")
159 library(ggmap)
160 map <- get_map(location = "Canada", zoom = 3)
161 lon <- as.numeric(paste(airportsCanada$longitude))
162 lat <- as.numeric(paste(airportsCanada$latitude))
163 airportsCoord <- as.data.frame(cbind(lon, lat))
164 (mapPoints <- ggmap(map) + geom_point(data=airportsCoord, aes(lon,
    lat), alpha=0.5))
165
166 # 1.7 – Create a second map showing all possible routes between
    these different airports.
167 summary(routesCanada)
168 summary(airportsCanada)
169 routesCoord <- sqldf("
170     select
171         a.sourceAirport,
172         a.destinationAirport,
173         b.longitude as sourceLon,
174         b.latitude as sourceLat,
175         c.longitude as destLon,
176         c.latitude as destLat
177     from routesCanada a

```

```

178   left join airportsCanada b
179     on a.sourceAirport = b.IATA
180   left join airportsCanada c
181     on a.destinationAirport = c.IATA")
182 lonBeg <- as.numeric(paste(routesCoord$sourceLon))
183 latBeg <- as.numeric(paste(routesCoord$sourceLat))
184 lonEnd <- as.numeric(paste(routesCoord$destLon))
185 latEnd <- as.numeric(paste(routesCoord$destLat))
186 routesCoord <- as.data.frame(cbind(lonBeg, latBeg, lonEnd, latEnd))
187 (mapRoutes <- mapPoints + geom_segment(data=routesCoord, aes(x=
    lonBeg, y=latBeg, xend=lonEnd, yend=latEnd), alpha=0.5))
188
189 # Calculate an airport ridership index based on the number of
    incoming routes.
190 arrivalFlights <- table(routesCanada$destinationAirport)
191 departureFlights <- table(routesCanada$sourceAirport)
192 totalFlights <- arrivalFlights + departureFlights
193 max(totalFlights)
194 mean(totalFlights)
195 var(totalFlights)
196 sd(totalFlights)
197 head(sort(totalFlights, decreasing = TRUE), n = 30)
198 totalFlightsCDF <- ecdf(totalFlights)
199 IATA <- names(totalFlights)
200
201 # Index drawing
202 curve(totalFlightsCDF(x-1), from = 0, to = 60, n = 100,
203       xlab = "Nombre de routes par aeroport",
204       ylab = "CDF")
205
206 # Calculate a combined index from the index.
207 combinedIndex <- round(totalFlights/max(totalFlights), 3)
208 combinedIndexTable <- data.frame(IATA,
209                                   as.numeric(paste(totalFlights)),
210                                   as.numeric(paste(combinedIndex)))
211 rownames(combinedIndexTable) <- NULL
212 colnames(combinedIndexTable) <- c("IATA", "totalFlights", "
    combinedIndex")
213 combinedIndexTable
214 airportsCanada <- sqldf("
215   select
216     a.*,
217     coalesce(b.combinedIndex, 0) as combinedIndex
218   from airportsCanada a
219   left join combinedIndexTable b
220     on a.IATA = b.IATA")
221 airportsCanada <- data.frame(as.matrix(airportsCanada ))
222
223 #1.11 – Create maps to visualize these indices using a bubble graph
    .
224 TrafficData <- subset(airportsCanada, as.numeric(paste(combinedIndex)
    ) > 0.05)
225 lon <- as.numeric(paste(TrafficData$longitude))
226 lat <- as.numeric(paste(TrafficData$latitude))
227 size <- as.numeric(paste(TrafficData$combinedIndex))
228 airportsCoord <- as.data.frame(cbind(lon, lat, size))
229 mapPoints <-

```

```

230   ggmap(map) +
231   geom_point(data=TraficData, aes(x=lon, y=lat, size=size), alpha=0.5,
             shape=16)
232   (mapTraffic <-
233     mapPoints +
234     scale_size_continuous(range = c(0, 20), name = "Trafic Index"))
235
236   # install package("leaflet")
237   library(leaflet)
238   url <- "http://hiking.waymarkedtrails.org/en/routebrowser/1225378/
         gpx"
239   download.file(url, destfile = paste(path, "/Reference/worldRoutes.
         gpx", sep=""), method = "wget")
240   worldRoutes <- readOGR(paste(path, "/Reference/worldRoutes.gpx", sep=
         ""), layer = "tracks")
241   markersData <- subset(airportsCanada, IATA %in% c('YUL', 'YVR', 'YYZ',
         'YQB'))
242   markersWeb <- c("https://www.aeroportdequebec.com/fr/pages/accueil"
         ,
243                   "http://www.admtl.com/",
244                   "http://www.yvr.ca/en/passengers",
245                   "https://www.torontopearson.com/")
246
247   # Defining the description text to be displayed by the markers
248   descriptions <- paste("<b><FONT COLOR=#31B404> Airport Details</FONT
         ></b> <br>",
249                         "<b>IATA: <a href=", markersWeb, ">", markersData$I
         ATA, "</a></b><br>",
250                         "<b>Name:</b>", markersData$name, "<br>",
251                         "<b>Coord.</b>: (", markersData$longitude, ",",
         markersData$latitude, ") <br>",
252                         "<b>Trafic Index:</b>", markersData$
         combinedIndex)
253
254   # Defining the icon to be add on the markers from fontawesome
         library
255   icons <- awesomeIcons(icon = 'paper-plane',
256                          iconColor = 'black',
257                          library = 'fa')
258
259   # Combinaison of the different components in order to create a
         standalone map
260   (mapTraffic <- leaflet(worldRoutes) %>%
261     addTiles(urlTemplate = "http://{s}.basemaps.cartocdn.com/light_
         all/{z}/{x}/{y}.png") %>%
262     addCircleMarkers(stroke = FALSE, data = TraficData, ~as.numeric(
         paste(longitude)), ~as.numeric(paste(latitude)),
263                      color = 'black', fillColor = 'green',
264                      radius = ~as.numeric(paste(combinedIndex))*30,
         opacity = 0.5) %>%
265     addAwesomeMarkers(data = markersData, ~as.numeric(paste(
         longitude)), ~as.numeric(paste(latitude)), popup =
         descriptions, icon=icons))
266
267   # Resizing of the map
268   mapTraffic$width <- 874
269   mapTraffic$height <- 700

```



```

270
271 # Export of the map into html format
272 # install.packages("htmlwidgets")
273 library(htmlwidgets)
274 saveWidget(mapTraffic, paste(path, "/Reference/leafletTraffic.html",
    sep = ""), selfcontained = TRUE)
275
276 #addMarkers(data = subset(airportsCanada,IATA %in% c('YUL','YVR','
    YYZ','YQB')), ~as.numeric(paste(longitude)), ~as.numeric(paste(
    latitude)), popup = ~IATA) %>%

```

Code Source A.3 – CaseStudyDevQ2.R

```

1 # coding: utf-8
2 # CaseStudyRQuebec2017
3 # Authors : David Beauchemin & Samuel Cabral Cruz
4
5 ##### Question 2 #####
6 # Distance calculation function between two airports.
7 # install.packages("geosphere")
8 library(geosphere)
9
10 airportsDist <- function(sourceIATA,destIATA)
11 {
12   # Verification of the sourceIATA and destIATA
13   sourceFindIndex <- match(sourceIATA,airportsCanada$IATA)
14   if(is.na(sourceFindIndex))
15   {
16     stop(paste("sourceIATA :",sourceIATA,"is not a valid IATA code"
17     ))
18   }
19   destFindIndex <- match(destIATA,airportsCanada$IATA)
20   if(is.na(destFindIndex))
21   {
22     stop(paste("destIATA :",destIATA,"is not a valid IATA code"))
23   }
24   sourceLon <- as.numeric(paste(airportsCanada$longitude)[
25     sourceFindIndex])
26   sourceLat <- as.numeric(paste(airportsCanada$latitude)[
27     sourceFindIndex])
28   sourceCoord <- c(sourceLon,sourceLat)
29   destLon <- as.numeric(paste(airportsCanada$longitude)[
30     destFindIndex])
31   destLat <- as.numeric(paste(airportsCanada$latitude)[
32     destFindIndex])
33   destCoord <- c(destLon,destLat)
34   airportDistList <- list()
35   airportDistList$source <- sourceIATA
36   airportDistList$dest <- destIATA
37   airportDistList$value <- round(distGeo(sourceCoord,destCoord)/
38     1000)
39   airportDistList$metric <- "Km"
40   airportDistList$xy_dist <- sqrt((sourceLon - destLon)**2 + (
41     sourceLat - destLat)**2)
42   airportDistList$sourceIndex <- sourceFindIndex
43   airportDistList$destIndex <- destFindIndex

```

```

37   airportDistList
38 }
39 airportsDist("AAA", "YQB")
40 airportsDist("YUL", "AAA")
41 airportsDist("YPA", "YQB")
42 airportsDist("YUL", "YQB")
43 airportsDist("YUL", "YQB")$value
44
45 # Function to establish the estimated time of arrival
46 # install.packages("lubridate")
47 library(lubridate)
48 arrivalTime <- function(sourceIATA, destIATA)
49 {
50   topSpeed <- 850
51   adjustFactor <- list()
52   adjustFactor$a <- 0.0001007194 # found by regression (not
      included)
53   adjustFactor$b <- 0.4273381 # found by regression (not included)
54   arrivalTimeList <- list()
55   arrivalTimeList$source <- sourceIATA
56   arrivalTimeList$dest <- destIATA
57   arrivalTimeList$departureTime <- Sys.time()
58   distance <- airportsDist(sourceIATA, destIATA)
59   cruiseSpeed <- (distance$value*adjustFactor$a + adjustFactor$b)*
      topSpeed
60   arrivalTimeList$avgCruiseSpeed <- cruiseSpeed
61   arrivalTimeList$flightTime <- ms(round(distance$value/cruiseSpeed
      *60, digits = 1))
62   arrivalTimeList$departureTZ <- paste(airportsCanada[distance$
      sourceIndex, "tzFormat"])
63   arrivalTimeList$arrivalTZ <- paste(airportsCanada[distance$
      destIndex, "tzFormat"])
64   arrivalTimeList$value <- with_tz(arrivalTimeList$departureTime +
      arrivalTimeList$flightTime,
65                                   tzone = arrivalTimeList$
      arrivalTZ)
66   arrivalTimeList
67 }
68 arrivalTime("AAA", "YYZ")
69 arrivalTime("YUL", "AAA")
70 arrivalTime("YUL", "YYZ")
71 arrivalTime("YUL", "YVR")
72 arrivalTime("YUL", "YYZ")$value
73 difftime(arrivalTime("YUL", "YVR")$value, Sys.time())
74 difftime(arrivalTime("YUL", "YYZ")$value, Sys.time())
75
76 # Import tax rates by province directly from the web
77 #install.packages("XML")
78 #install.packages("RCurl")
79 #install.packages("rlist")
80 library(XML)
81 library(RCurl)
82 library(rlist)
83 theurl <- getURL("http://www.calculconversion.com/sales-tax-
      calculator-hst-gst.html", .opts = list(ssl.verifypeer = FALSE))
84 tables <- readHTMLTable(theurl)
85 provinceName <- as.character(sort(unique(airportsCanada$province)))

```

```

86 taxRates <- as.data.frame(cbind(provinceName, as.numeric(sub("%", "",
      tables$'NULL'[-13,5]))/100+1))
87 colnames(taxRates) <- c("province", "taxRate")
88 taxRates
89
90 # Shipping cost calculation function
91 shippingCost <- function(sourceIATA, destIATA, weight,
92                           percentCredit = 0, dollarCredit = 0)
93 {
94
95   # Verification of the existence of the route between sourceIATA
      and destIATA
96   routeConcat <- as.character(paste(routesCanada$sourceAirport,
      routesCanada$destinationAirport))
97   if(is.na(match(paste(sourceIATA, destIATA), routeConcat)))
98   {
99     stop(paste("the combination of sourceIATA and destIATA (",
      sourceIATA, "-", destIATA, ") do not corresponds to existing
      route"))
100  }
101
102  if(weight < 0 || weight > 30)
103  {
104    stop("The weight must be between 0 and 30 Kg")
105  }
106
107  if(percentCredit < 0 || percentCredit > 1)
108  {
109    stop("The percentage of credit must be between 0 % and 100 %")
110  }
111
112  if(dollarCredit < 0)
113  {
114    stop("The dollar credit must be superior to 0 $")
115  }
116
117  minimalDist = 100
118  distance <- airportsDist(sourceIATA, destIATA)
119  if (distance$value < minimalDist)
120  {
121    # We verify if the distance of shipping is further than the
      minimal requirement
122    stop(paste("The shipping distance is under the minimal
      requirement of", minDist, "Km"))
123  }
124
125  # Pricing variables
126  distanceFactor <- 0.03
127  weightFactor <- 0.8
128  fixedCost <- 3.75
129  profitMargin <- 1.12
130
131  # Traffic Index
132  trafficIndexSource <- as.numeric(paste(airportsCanada[distance$
      sourceIndex, "combinedIndex"]))
133  trafficIndexDest <- as.numeric(paste(airportsCanada[distance$
      destIndex, "combinedIndex"]))

```

```

134
135 # Calculation of the base cost
136 baseCost <- fixedCost + (distance$value*distanceFactor + weight*
    weightFactor)/(trafficIndexSource*trafficIndexDest)
137
138 # Additional automated credits
139 automatedCredit <- 1
140 # Lightweight
141 automatedCredit <- automatedCredit * ifelse(weight < 2, 0.5, 1)
142 # Gold Member
143 automatedCredit <- automatedCredit * ifelse(baseCost > 100, 0.9,
    1)
144 # Partnership
145 automatedCredit <- automatedCredit * switch(sourceIATA,
146     "YUL" = 0.85,
147     "YHU" = 0.95,
148     "YMX" = 0.95,
149     "YYZ" = 0.9,
150     "YKZ" = 0.975,
151     "YTZ" = 0.975,
152     "YZD" = 0.975)
153 # The Migrator
154 if(distance$value > 3500)
155 {
156     automatedCredit <- automatedCredit * 0.8125
157 }
158 else if(distance$value >= 3000)
159 {
160     automatedCredit <- automatedCredit * 0.825
161 }
162 else if(distance$value >= 2500)
163 {
164     automatedCredit <- automatedCredit * 0.85
165 }
166 else if(distance$value >= 2000)
167 {
168     automatedCredit <- automatedCredit * 0.9
169 }
170
171 # Calculation of tax rate and control of text
172 taxRate <- as.numeric(paste(taxRates[match(airportsCanada[
    distance$sourceIndex, "province"], taxRates$province), "taxRate"
    ]))
173 price <- round(pmax(fixedCost*profitMargin*automatedCredit*
    taxRate, (baseCost*automatedCredit*profitMargin - dollarCredit
    )*(1 - percentCredit)*taxRate), 2)
174
175 # Return List
176 shippingCostList <- list()
177 shippingCostList$distance <- distance
178 shippingCostList$weight <- weight
179 shippingCostList$distanceFactor <- distanceFactor
180 shippingCostList$weightFactor <- weightFactor
181 shippingCostList$fixedCost <- fixedCost
182 shippingCostList$profitMargin <- profitMargin
183 shippingCostList$percentCredit <- percentCredit
184 shippingCostList$dollarCredit <- dollarCredit

```

```

185 shippingCostList$minimalDist <- minimalDist
186 shippingCostList$trafficIndex <- list(trafficIndexSource,
    trafficIndexDest)
187 shippingCostList$baseCost <- baseCost
188 shippingCostList$automatedCredit <- 1-automatedCredit
189 shippingCostList$taxRate <- taxRate
190 shippingCostList$price <- price
191 shippingCostList
192 }
193 shippingCost("YUL", "YVR", 1)
194 shippingCost("YUL", "YQB", 1)
195 shippingCost("YUL", "YVR", 30)
196 shippingCost("YUL", "YQB", 30)

```

Code Source A.4 – CaseStudyDevQ3.R

```

1 # coding: utf-8
2 # CaseStudyRQuebec2017
3 # Authors : David Beauchemin & Samuel Cabral Cruz
4
5 ##### Question 3 #####
6 # We visualize the impact of a changes of distance and the weight
    starting from the YUL airport.
7 curve(shippingCost("YUL","YQB",x)$price,0.01,50,ylim=c(0,200),
8       main="Shipping Price Variation with Weight",xlab="weight (Kg)"
9       ,
10      ylab="price (CND $)",lwd = 2)
11 curve(shippingCost("YUL","YVR",x)$price,0.01,50,xlab="weight (Kg)",
12       ylab="price (CND $)",add=TRUE, col = "red", lwd = 2)
13 curve(shippingCost("YUL","YYZ",x)$price,0.01,50,xlab="weight (Kg)",
14       ylab="price (CND $)",add=TRUE, col = "blue", lwd = 2)
15 curve(shippingCost("YUL","YYC",x)$price,0.01,50,xlab="weight (Kg)",
16       ylab="price (CND $)",add=TRUE, col = "purple", lwd = 2)
17 text(x=c(25,25,25,25),y=c(50,90,140,175),c("YUL-YYZ","YUL-YQB","YUL-
    YVR","YUL-YYC"),adj = 0.5,cex = 0.75,font = 2,col = c("blue","
    black","red","purple"))

```

Code Source A.5 – CaseStudyDevQ4.R

```

1 # coding: utf-8
2 # CaseStudyRQuebec2017
3 # Authors : David Beauchemin & Samuel Cabral Cruz
4
5 ##### Question 4 #####
6 # Import data of the competition
7 compData <- read.csv(paste(path,"/Reference/benchmark.csv",sep=""))
8 View(compData)
9 colnames(compData) <- c("weight","distance","price")
10 summary(compData)
11
12 # Weight visualisation
13 hist(compData$weight, freq = TRUE, main = "Repartition according to
    the weight",
14      xlab = "weight (Kg)", col = "cadetblue",breaks = 50)
15 weightCDF <- ecdf(compData$weight)

```

```

16 curve(weightCDF(x),0,15,ylim = c(0,1),lwd = 2,
17       xlab = "weight (Kg)",
18       ylab = "Cumulative Distribution Function")
19
20 # Distance visualisation
21 hist(compData$distance, freq = TRUE, main = "Repartition according
    to the distance",
22       xlab = "distance (Km)", col = "cadetblue",breaks = 50)
23 distanceCDF <- ecdf(compData$distance)
24 curve(distanceCDF(x),0,2500,ylim = c(0,1),lwd = 2,
25       xlab = "distance (Km)",
26       ylab = "Cumulative Distribution Function")
27
28 # Price according to weight
29 plot(compData$weight,compData$price,main = "Price according to the
    weight",
30       xlab = "weight (Kg)", ylab = "Price (CAD $)")
31
32 # Price according to distance
33 plot(compData$distance,compData$price,main = "Price according to
    the distance",
34       xlab = "distance (Km)", ylab = "Price (CAD $)")
35
36 # Price according to weight and distance
37 # install.packages("rgl")
38 library(rgl)
39 plot3d(compData$weight,compData$distance,compData$price)
40
41 # Chi's Square Test of Independency between the two variables
42 weightsBinded <- as.numeric(cut(compData$weight,25))
43 distancesBinded <- as.numeric(cut(compData$distance,25))
44 contingencyTable <- table(weightsBinded,distancesBinded)
45 chisq.test(contingencyTable)
46 contingencyTable <- rbind(contingencyTable[1:4,],colSums(
    contingencyTable[5:18,]))
47 (contingencyTable <- cbind(contingencyTable[,1:14],rowSums(
    contingencyTable[,15:24])))
48 independencyTest <- chisq.test(contingencyTable)
49 head(independencyTest$expected)
50 head(independencyTest$observed)
51 head(independencyTest$stdres)
52 independencyTest
53 cor.test(compData$weight,compData$distance,method = "pearson")
54
55 # Linear model
56 # we assume the same profit margin to simplify the situation
57 # We keep an intercept since we have a fixed cost
58 profitMargin <- 1.12
59 avgTaxRate <- sum(table(airportsCanada$province)*as.numeric(paste(
    taxRates$taxRate)))/length(airportsCanada$province)
60 compModel <- lm(price/(profitMargin*avgTaxRate) ~ distance + weight
    , compData)
61 summary(compModel)
62
63 # We plot the model
64 par(mfrow=c(2,2))
65 plot(compModel)

```

Code Source A.6 – CaseStudyDevQ5.R

```

1 # coding: utf-8
2 # CaseStudyRQuebec2017
3 # Authors : David Beauchemin & Samuel Cabral Cruz
4
5 ##### Question 5 #####
6 # install.packages("actuar")
7 library("actuar")
8
9 distName <- c("Normal", "Gamma", "LogNormal", "Weibull", "Pareto", "
    InvGaussian")
10 empCDF <- ecdf(compData$weight)
11 empPDF <- function(x, delta=0.01)
12 {
13   (empCDF(x+delta/2)-empCDF(x-delta/2))/delta
14 }
15
16 # We built a general function for all kind of distribution.
17 distFit <- function(dist,...)
18 {
19   dist = tolower(dist)
20   args = list(...) # Using of kargs since we are not sure the
                       # number of parameters
21   if(dist == "normal")
22   {
23     law = "norm"
24     nbparam = 2
25   }
26   else if(dist == "gamma")
27   {
28     law = "gamma"
29     nbparam = 2
30   }
31   else if(dist == "lognormal")
32   {
33     law = "lnorm"
34     nbparam = 2
35   }
36   else if(dist == "weibull")
37   {
38     law = "weibull"
39     nbparam = 2
40   }
41   else if(dist == "pareto")
42   {
43     law = "pareto"
44     nbparam = 2
45   }
46   else if(dist == "invgaussian")
47   {
48     law = "invgauss"
49     nbparam = 2
50   }
51   else
52   {
53     message <- "The only distribution available are:

```

```

54     Nnormal, Gamma, LogNormal, Weibull, Pareto and InvGaussian.
55     (The case is ignored)"
56     stop(message)
57 }
58 if(nbparam != length(args))
59 {
60     message <- paste("There is a mismatch between the number of
        arguments passed to the function and the number of
        arguments needed to the distribution.",
61                      "The",dist,"distribution is taking",nbparam,"
        parameters and",length(args),"parameters
        were given.")
62     stop(message)
63 }
64
65 # Treament
66 param <- optim(par = args, function(par) -sum(do.call(eval(parse(
        text = paste("d",law,sep=""))),c(list(compData$weight),par,
        log = TRUE))))
67 # Deviance value of the fitting
68 devValue <- sum((empPDF(x <- seq(0,30,0.1))-do.call(eval(parse(
        text = paste("d",law,sep=""))),c(list(x),param$par)))*2)
69
70 # Return List
71 distFitList <- list()
72 distFitList$param <- param$par
73 distFitList$errorValue <- param$value
74 distFitList$devValue <- devValue
75 distFitList
76 }
77
78
79 (resultDistFitting <- sapply(distName,function(x) unlist(distFit(x
    ,1,1))))
80
81 law <- c("norm","gamma","lnorm","weibull","pareto","invgauss")
82 col <- c("red", "yellow", "purple", "green", "cyan", "blue")
83 x <- seq(0,30,0.1)
84
85 # Visulization of the fitting distribution
86 par(mfrow = c(1,2),font = 2)
87 plot(function(x) empCDF(x), xlim = c(0,15), main = "", xlab = "
    weight (Kg)", ylab = "CDF(x)")
88 invisible(sapply(1:length(law),function(i) curve(do.call(eval(parse
    (text = paste("p",law[i],sep=""))),c(list(x), as.vector(
    resultDistFitting[c(1:2),i]))), add = TRUE, lwd = 3, col = col[
    i]))))
89 hist(compData$weight, xlim = c(0,15), main = "", xlab = "weight (Kg
    )", breaks = 300,freq = FALSE)
90 invisible(sapply(1:length(law),function(i) curve(do.call(eval(parse
    (text = paste("d",law[i],sep=""))),c(list(x), as.vector(
    resultDistFitting[c(1:2),i]))), add = TRUE, lwd = 3, col = col[
    i]))))
91 legend(x="right",distName, inset = 0.1, col = col, pch = 20, pt.cex
    = 2, cex = 1, ncol = 1, bty = "n", text.width = 2, title = "
    Distribution")
92 mtext("Ajustement sur distribution empirique", side = 3, line = -2,

```



```

    outer = TRUE)
93
94 # We thus choose the LogNormal distribution which possesses the
    smallest deviance and the best fit.
95 distChoice <- "LogNormal"
96 (paramAdjust <- resultDistFitting[c(1:2),match(distChoice,distName)
    ])
97
98 # It is also possible to do the equivalent with fitdistr of the
    library MASS,
99 # but with less option for the distribution.
100 library("MASS")
101 (fit.normal <- fitdistr(compData$weight,"normal"))
102 (fit.gamma <- fitdistr(compData$weight,"gamma"))
103 (fit.lognormal <- fitdistr(compData$weight,"lognormal"))
104 (fit.weibull <- fitdistr(compData$weight,"weibull"))

```

Code Source A.7 – CaseStudyDevQ6.R

```

1 # coding: utf-8
2 # CaseStudyRQuebec2017
3 # Authors : David Beauchemin & Samuel Cabral Cruz
4
5 ##### Question 6 #####
6 theurl <- getURL(paste("file://",path,"/Statement/
    CaseStudyStatement.html",sep=""),.opts = list(ssl.verifypeer =
    FALSE))
7 tables <- readHTMLTable(theurl)
8 lambdaTable <- as.data.frame(tables$"NULL")
9 colnames(lambdaTable) <- c("Month","Avg3yrs")
10 lambdaTable
11
12 # The possible routes are filtered from the starting point 'YUL'
13 # and a distribution is created according to the destination index.
14 simAirportsDests <- as.character(paste(routesCanada[routesCanada$
    sourceAirport == "YUL","destinationAirport"]))
15 simCombinedIndex <- combinedIndex[names(combinedIndex) %in%
    simAirportsDests]
16 airportsDensity <- simCombinedIndex/sum(simCombinedIndex)
17
18 # Function for the simulation of the shipment prices.
19 simulShipmentPrice <- function(Arrival,Weight)
20 {
21   ownPrice <- ifelse(is(testSim <- try(shippingCost("YUL",Arrival,
    Weight)$price,silent = TRUE),"try-error"),NA,testSim)
22   distance <- airportsDist("YUL",Arrival)$value
23   nd <- as.data.frame(cbind(distance,Weight))
24   colnames(nd) <- c("distance","weight")
25   compPrice <- predict(compModel,newdata = nd)
26   customerChoice <- ifelse(is.na(ownPrice),0,ifelse(ownPrice <
    compPrice,1,0))
27   rbind(Arrival,distance,Weight,ownPrice,compPrice,customerChoice)
28 }
29
30 # Function for the simulation of the shipment parameters, weights
    and destinations.

```

```

31 simulShipment <- function(simNbShipments)
32 {
33   # Weights are then generated for each of the packages.
34   simWeights <- eval(parse(text = paste("r",law[match(distChoice,
35     distName)],sep = " "))(simNbShipments,paramAdjust[1],
36     paramAdjust[2])
37   # We finally generate a destination for each package (the
38     departure will always be from 'YUL').
39   simArrivals <- sample(size = simNbShipments,names(airportsDensity
40     ),prob = airportsDensity,replace = TRUE)
41   sapply(seq(1,simNbShipments),function(x) simulShipmentPrice(
42     simArrivals[x],simWeights[x]))
43 }
44
45 # Function for overall simulation
46 simulOverall <-function()
47 {
48   # We generate n observations of the Poisson distribution with
49     param = sum (lambda).
50   # We know from probability notion that the sum of independent
51     Poisson distribution follows
52   # a Poisson distribution with param = sum (lambda).
53   simNbShipments <- rpois(1,lambda = sum(as.numeric(paste(
54     lambdaTable$Avg3yrs))))
55   # We simulate each shipment
56   simulShipment(simNbShipments)
57 }
58
59 nsim <- 1
60 simulResults <- replicate(nsim, simulOverall(),simplify = FALSE)
61 (marketShareSales <- sapply(1:nsim,function(x) sum(as.numeric(
62   simulResults[[x]][6,]))/length(simulResults[[x]][6,])))
63 (ownRevenus <- sum(as.numeric(simulResults[[1]][4,])*as.numeric(
64   simulResults[[1]][6,]),na.rm = TRUE))
65 (compRevenus <- sum(as.numeric(simulResults[[1]][5,])*(1-as.numeric(
66   simulResults[[1]][6,]),na.rm = TRUE))
67 (marketShareRevenus <- ownRevenus/(ownRevenus+compRevenus))
68
69 arrivalSales <- as.character(simulResults[[1]][1,simulResults
70   [[1]][6,]==1])
71 distanceSales <- as.numeric(simulResults[[1]][2,simulResults
72   [[1]][6,]==1])
73 weightSales <- as.numeric(simulResults[[1]][3,simulResults
74   [[1]][6,]==1])
75
76 arrivalComp <- as.character(simulResults[[1]][1,simulResults
77   [[1]][6,]==0])
78 distanceComp <- as.numeric(simulResults[[1]][2,simulResults
79   [[1]][6,]==0])
80 weightComp <- as.numeric(simulResults[[1]][3,simulResults
81   [[1]][6,]==0])
82
83 # Representation of the result
84 table(arrivalSales)
85 mean(distanceSales)
86 table(arrivalComp)
87 mean(distanceComp)

```

```

71 par(mfrow = c(1,1))
72 hist(weightSales, freq = FALSE, breaks = 100, xlim = c(0,15), main =
    "Sales vs Theoretical Weights Distribution", xlab = "weight (Kg
    )")
73 curve(do.call(eval(parse(text = paste("d",law[match(distChoice,
    distName)], sep = " "))),c(list(x),as.vector(paramAdjust))),add =
    TRUE, lwd = 2)
74 abline(v = v <- exp(paramAdjust[1]+paramAdjust[2]**2/2), lwd = 2)
75 text(v+0.75,0.3,as.character(round(v,2)))
76 abline(v = v <- mean(weightSales), col = "red", lwd = 2)
77 text(v - 0.75,0.3,round(v,2), col = "red")

```
