DATA MINING

Master Meci - Parcours Data - Options PISE et CCESE

Claude Grasland, Professeur de Géographie, Université de Paris (Diderot)

2021-02-09

Table des Matières

Présentation

À propos de ce document

Ce document est la première version du cours de Data Mining dispensé aux étudiants de deuxième année de l'option Data du master MECI

Il est basé sur R version 4.0.3 (2020-10-10).

Ce document est régulièrement corrigé et mis à jour. La version de référence est disponible en ligne à l'adresse :

- `https://ClaudeGrasland.github.io/DataMining1`.

Pour toute suggestion ou correction, il est possible de me contacter par mail

Prérequis

Le seul prérequis pour suivre ce document est d'avoir installé R et RStudio sur votre ordinateur. Il s'agit de deux logiciels libres, gratuits, téléchargeables en ligne et fonctionnant sous PC, Mac et Linux.

Pour installer R, il suffit de se rendre sur une des pages suivantes [1] :

- Installer R sous Windows
- Installer R sous Mac

Pour installer RStudio, rendez-vous sur la page suivante et téléchargez la version adaptée à votre système :

- `https://www.rstudio.com/products/rstudio/download/#download`

Remerciements

Ce document a bénéficié de la relecture et des suggestions ... des étudiants qui en ont été les cobayes des premières versions.

Ce document est généré par l'excellente extension bookdown de Yihui Xie et il s'est servi du template proposé par Julien Barnier pour introduire des exercices interactifs dans son cours de tidyverse.

Licence

Figure 1: Licence Creative Commons

[1] Sous Linux, utilisez votre gestionnaire de packages habituel.

Chapitre 1

Collecter des données à l'aide d'une API

```
library(knitr)
library(httr)
library(jsonlite)
```

```
Attachement du package : 'jsonlite'
```

```
The following object is masked from 'package:purrr':

    flatten
```

```
library(insee)
library(dplyr)
library(lubridate)
```

```
Attachement du package : 'lubridate'
```

```
The following objects are masked from 'package:base':

    date, intersect, setdiff, union
```

1.1 Qu'est-ce qu'une API ?

1.1.1 Définitions

On peut partir de la définition suivante

> En informatique, API est l'acronyme d'*Application Programming Interface*, que l'on traduit en français par **interface de programmation applicative** ou **interface de programmation d'application**. L'API peut être résumée à une solution informatique qui permet à des applications de communiquer entre elles et de s'échanger mutuellement des services ou des données. Il s'agit en réalité d'un ensemble de fonctions qui facilitent, via un langage de programmation, l'accès aux services d'une application. (Source : Journal du Net)

1.1.2 Domaine d'application

Une API peut remplir des fonctions très diverses :

> Dans le domaine d'internet, l'API permet aux développeurs de pouvoir utiliser un programme sans avoir à se soucier du fonctionnement complexe d'une application. Les API peuvent par exemple être utilisées pour déclencher des campagnes publicitaires d'e-mailing de façon automatique sans avoir à passer par la compréhension d'une telle application (c'est le cas avec l'API AdWords de Google, par exemple). On les retrouve aujourd'hui dans de nombreux logiciels, en particulier dans les systèmes d'exploitation, les serveurs d'applications, dans le monde du graphisme (OpenGL), dans les applications SaaS (Office 365, G Suite, Salesforce…), les bases de données, l'open data, etc.(Source : Journal du Net)

1.1.3 Système client-serveur

D'une manière générale, les API supposent un échange d'informations entre un *client* et un *serveur*.

> Ces échanges d'informations suivent un *protocole* c'est-à-dire un ensemble de règles. Il existe deux grands protocoles de communication sur lesquels s'adossent les API : Simple Object Access Protocol (SOAP) et Representational State Transfer (REST). Le second s'est désormais largement imposé face au premier car il est plus flexible. Il a donné naissance aux API dites REST ou RESTful (Source : Journal du Net)

1.2 Comment utiliser une API dans R ?

Le métier de data analyst implique presque nécessairement l'emploi d'API. Les langages de programmation R ou Python ont donc l'un comme l'autre mis au point des packages pour faciliter l'envoi de requêtes sur des serveurs dotés d'API. A titre d'introduction, nous allons reprendre (et traduire en français) quelques extraits d'un billet proposé par un étudiant en doctorat de biostatistiques à l'université de Californie San Diego.

- Pascual C., 2020, Getting Started with APIs in R

1.2.1 Pourquoi utiliser des API ?

> «API» est un terme général désignant le lieu où un programme informatique interagit avec un autre ou avec lui-même. Dans ce didacticiel, nous travaillerons spécifiquement avec des API Web, où deux ordinateurs différents - un client et un serveur - interagiront l'un avec l'autre pour demander et fournir des données, respectivement.
>
> Les API offrent aux scientifiques des données un moyen raffiné de demander des données propres et organisées à partir d'un site Web. Lorsqu'un site Web comme Facebook met en place une API, il met essentiellement en place un ordinateur qui attend les demandes de données.
>
> Une fois que cet ordinateur reçoit une demande de données, il effectuera son propre traitement des données et les enverra à l'ordinateur qui l'a demandé. De notre point de vue en tant que demandeur, nous devrons écrire du code dans R qui crée la demande et indique à l'ordinateur exécutant l'API ce dont nous avons besoin. Cet ordinateur lira ensuite notre code, traitera la requête et renverra des données bien formatées qui peuvent être facilement analysées par les bibliothèques R existantes.
>
> Pourquoi est-ce précieux? Comparez l'approche API au scraping Web pur. Lorsqu'un programmeur gratte une page Web, il reçoit les données dans un morceau de HTML désordonné. Bien qu'il existe certainement des bibliothèques qui facilitent l'analyse du texte HTML, ce sont toutes des étapes de nettoyage qui doivent être prises avant même de mettre la main sur les données que nous voulons!
>
> Souvent, nous pouvons immédiatement utiliser les données que nous obtenons d'une API, ce qui nous fait gagner du temps et de la frustration.

Source : Traduction française d'un billet de Pascual C., 2020

1.2.2 Installer les packages jsonlite et httr

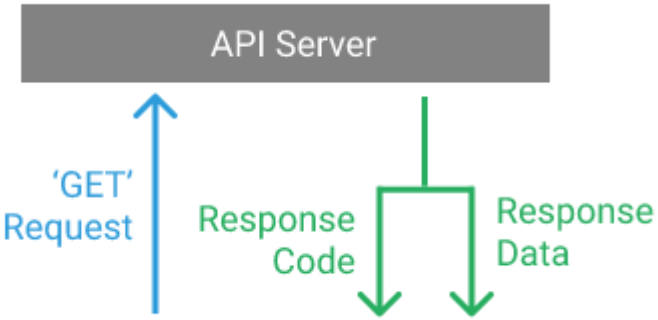
Pour travailler avec des API dans R, nous devons intégrer certaines bibliothèques (*library*). Ces bibliothèques prennent toutes les complexités d'une requête d'API et les enveloppent dans des fonctions que nous pouvons utiliser dans des lignes de code uniques. Les bibliothèques R que nous utiliserons sont `httr` et `jsonlite`. Elles remplissent des rôles différents dans notre introduction des API, mais les deux sont essentiels.Si vous ne disposez pas de ces bibliothèques dans votre console R ou RStudio, vous devez d'abord les télécharger.

```
library(httr)
library(jsonlite)
```

1.2.3 Structure d'une requête

Une requête adressé à une API va suivre le schéma suivant :

```
knitr::include_graphics("img/API_GET.png",)
```



Il existe plusieurs types de requêtes que l'on peut adresser à un serveur API. Pour nos besoins, nous allons simplement demander des données, ce qui correspond à une demande **GET**. Les autres types de requêtes sont POST et PUT, mais nous n'avons pas à nous en préoccuper dans l'immédiat

Afin de créer une requête GET, nous devons utiliser la fonction *GET()* de la bibliothèque `httr`. La fonction GET() nécessite une URL, qui spécifie l'adresse du serveur auquel la demande doit être envoyée. A titre d'exemple, C. Pascual propose de travailler avec l'**API Open Notify**, qui donne accès à des données sur divers projets de la NASA. À l'aide de l'API Open Notify, nous pouvons notamment en savoir plus sur l'emplacement de la Station spatiale internationale et sur le nombre de personnes actuellement dans l'espace.

Notre programme télécharge les données disponibles à l'adresse du serveur et les stocke dans un objet auquel on peut donner le nom que l'on souhaite, par exemple *toto*

```
toto <- GET("http://api.open-notify.org/astros.json")
toto
```

```
Response [http://api.open-notify.org/astros.json]
  Date: 2021-02-09 13:24
  Status: 200
  Content-Type: application/json
  Size: 356 B
```

Lorsqu'on affiche la réponse, on obtient ici quatre informations :

- **Date** : le moment exact du téléchargement, très utile pour suivre les mises à jour
- **Status** : le code informatique de résultat de la requête. La valeur *200* indique un succès alors que les autres valeurs signaleront un problème.
- **Content-Type** : le type d'information recueillie. Ici, une application au format json

- **Size** : la taille du fichier résultant du transfert.

On pourrait également en savoir plus en tapant la commande *str()* qui nous indique que le résultat est une liste comportant 10 branches et de nombreuses sous-branches :

```
str(toto)
```

```
List of 10
 $ url        : chr "http://api.open-notify.org/astros.json"
 $ status_code: int 200
 $ headers    :List of 6
  ..$ server                      : chr "nginx/1.10.3"
  ..$ date                        : chr "Tue, 09 Feb 2021 13:24:15 GMT"
  ..$ content-type                : chr "application/json"
  ..$ content-length              : chr "356"
  ..$ connection                  : chr "keep-alive"
  ..$ access-control-allow-origin: chr "*"
  ..- attr(*, "class")= chr [1:2] "insensitive" "list"
 $ all_headers:List of 1
  ..$ :List of 3
  .. ..$ status : int 200
  .. ..$ version: chr "HTTP/1.1"
  .. ..$ headers:List of 6
  .. .. ..$ server                      : chr "nginx/1.10.3"
  .. .. ..$ date                        : chr "Tue, 09 Feb 2021 13:24:15 GMT"
  .. .. ..$ content-type                : chr "application/json"
  .. .. ..$ content-length              : chr "356"
  .. .. ..$ connection                  : chr "keep-alive"
  .. .. ..$ access-control-allow-origin: chr "*"
  .. .. ..- attr(*, "class")= chr [1:2] "insensitive" "list"
 $ cookies    :'data.frame':    0 obs. of  7 variables:
  ..$ domain    : logi(0)
  ..$ flag      : logi(0)
  ..$ path      : logi(0)
  ..$ secure    : logi(0)
  ..$ expiration: 'POSIXct' num(0)
  ..$ name      : logi(0)
  ..$ value     : logi(0)
 $ content    : raw [1:356] 7b 22 6d 65 ...
 $ date       : POSIXct[1:1], format: "2021-02-09 13:24:15"
 $ times      : Named num [1:6] 0 0.00212 0.16031 0.16047 0.32035 ...
  ..- attr(*, "names")= chr [1:6] "redirect" "namelookup" "connect" "pretransfer" ...
 $ request    :List of 7
  ..$ method    : chr "GET"
  ..$ url       : chr "http://api.open-notify.org/astros.json"
  ..$ headers   : Named chr "application/json, text/xml, application/xml, */*"
  .. ..- attr(*, "names")= chr "Accept"
  ..$ fields    : NULL
  ..$ options   :List of 2
  .. ..$ useragent: chr "libcurl/7.29.0 r-curl/4.3 httr/1.4.2"
  .. ..$ httpget  : logi TRUE
  ..$ auth_token: NULL
  ..$ output    : list()
  .. ..- attr(*, "class")= chr [1:2] "write_memory" "write_function"
  ..- attr(*, "class")= chr "request"
 $ handle     :Class 'curl_handle' <externalptr>
 - attr(*, "class")= chr "response"
```

La branche qui nous intéresse le plus est *content* puisque c'est celle qui contient les données.

1.2.4 Extraction des données

Les données contenues dans la réponse ont été stockées au format *JSON (JavaScript Object Notation)* qui est devenu un standard pour les échanges de données. Sans entrer dans le détail de ce langage, on retiendra qu'il va falloir convertir les données JSON dans un format de tableau lisible par R ce qui se fait ici en deux étapes.

Tout d'abord extraire le champ content et le convertir en mode caractère :

```
# conversion du contenu de toto en mode character
toto2<-rawToChar(toto$content)
toto2
```

```
[1] "{\"message\": \"success\", \"number\": 7, \"people\": [{\"craft\": \"ISS\", \"name\": \"Sergey Ryzhik
```

```
str(toto2)
```

```
 chr "{\"message\": \"success\", \"number\": 7, \"people\": [{\"craft\": \"ISS\", \"name\": \"Sergey Ryzhik
```

Puis convertir ces données de type JSON en données utilisables par R à l'aide de la fonction *fromJson()* du package `jsonlite()`

```
toto3 <- fromJSON(toto2)
str(toto3)
```

```
List of 3
 $ message: chr "success"
 $ number : int 7
 $ people :'data.frame':    7 obs. of  2 variables:
  ..$ craft: chr [1:7] "ISS" "ISS" "ISS" "ISS" ...
  ..$ name : chr [1:7] "Sergey Ryzhikov" "Kate Rubins" "Sergey Kud-Sverchkov" "Mike Hopkins" ...
```

On obtient finalement une liste de trois éléments dont le dernier est un *data.frame* décrivant les astronautes présents dans la station spatiale internationale au moment de l'execution du programme.

```
toto4<-toto3$people
str(toto4)
```

```
'data.frame':   7 obs. of  2 variables:
 $ craft: chr  "ISS" "ISS" "ISS" "ISS" ...
 $ name : chr  "Sergey Ryzhikov" "Kate Rubins" "Sergey Kud-Sverchkov" "Mike Hopkins" ...
```

```
kable(toto4,caption = "Passagers de l'ISS en temps réel")
```

1.2.5 API et mise à jour en temps réel

Sur le site web du billet proposé par C. Pascual en février 2020, on trouve une autre liste ne comportant que 6 passagers et avec des noms totalement différents :

En effet, l'API renvoie les résultats au moment de l'execution de la fonction *GET()* ce qui correspond à février 2020 pour le billet de blog. Or, les astronautes sont remplacés au plus tous les six mois ce qui explique que tous les noms soient différents un an après.

Table 1.1: Passagers de l'ISS en temps réel

craft	name
ISS	Sergey Ryzhikov
ISS	Kate Rubins
ISS	Sergey Kud-Sverchkov
ISS	Mike Hopkins
ISS	Victor Glover
ISS	Shannon Walker
ISS	Soichi Noguchi

Table 1.2: Passagers de l'ISS en février 2020

craft	name
ISS	Christina Koch
ISS	Alexander Skvortsov
ISS	Luca Parmitano
ISS	Andrew Morgan
ISS	Oleg Skripochka
ISS	Jessica Meir

NB : Cet exemple permet de mettre en évidence une fonction centrale des API qui est la mise à jour en temps réel des données !

1.2.6 API et requête paramétrique

L'exemple précédent consistait à télécharger la totalité d'un tableau et ne demandait donc pas de paramètres particuliers. Mais il peut aussi arriver (par exemple si une base de données est très volumineuse) que l'on précise à l'aide de paramètres ce que l'on veut précisément télécharger.

A titre d'exemple, C. Pascual propose d'utiliser une autre API de la NASA intitulée ISS Pass Time qui permet de savoir à quel moment la station ISS passera au dessus d'un certain point du globe.

L'exemple choisi par C.Pascual est la recherche des trois prochaines dates de passage de l'ISS au dessus de New York dont les coordonnées de latitude et de longitude sont 40.7 et -74.0 :

```
titi <- GET("http://api.open-notify.org/iss-pass.json",
            query = list(lat = 40.7, lon = -74, n=3))
titi2 <- fromJSON(rawToChar(titi$content))
titi3 <- titi2$response
titi3
```

```
  duration   risetime
1      629 1612881083
2      637 1612886885
3      569 1612892775
```

Le résultat paraît à première vue assez déconcertant. Mais la lecture de la documentation de l'API indique que les deux variables du tableau correspondent respectivement :

- *duration* : nombre de secondes pendant lesquelles la station sera à la verticale du point avec un angle de + ou - 10 degrés.
- *risetime* : moment de passage exprimé en Unix Time c'est-à-dire en nombre de secondes écoulées depuis le 1er Janvier 1970 UTC.

Si l'on veut se ramener à une date précise, il faut donc convertir ce temps à l'aide d'une fonction R. Le plus simple est pour cela d'utiliser la fonction *as_datetime()* du package `lubridate`.

```
library(lubridate)
titi3$risetime<-as_datetime(titi3$risetime)
kable(titi3)
```

duration	risetime
629	2021-02-09 14:31:23
637	2021-02-09 16:08:05
569	2021-02-09 17:46:15

1.3 API ou data packages ?

L'utilisation d'API à l'aide des fonctions de base `httpr` et `jsonlite` constitue à moyen terme une étape indispensable de la formation d'un data analyste. Mais heureusement elle n'est pas toujours indispensable pour le débutant car plusieurs packages R (ou Python) ont été développées par des programmeurs pour faciliter l'usage des API.

Ces packages executent en pratique les commandes de l'API, mais sans que l'utilisateur ait besoin d'avoir aucune connaissance sur la syntaxe de la fonction *GET()* qui a collecté les données ni des transformations effectuées sur les résultats pour transformer les données JSON en *data.frame* ou *tibble*. La connaissance de ces packages spécialisées offre donc une grosse économie de temps … s'ils ont été bien conçus.

On va prendre comme exemple le package `insee` mis au point récemment pour faciliter l'accès aux données de cette organisation. La documentation du package est accessible par le lien ci-dessous

https://www.data.gouv.fr/fr/reuses/insee-package-r/

Cette page renvoie vers une "vignette" c'est-à-dire une suite de programmes exemples.

https://inseefr.github.io/R-Insee-Data/

1.3.1 Installation et chargement du package

On commence par installer le package `insee` ce qui peut prendre quelques minutes mais sera fait une seule fois (sauf mise à jour).

```
# install.packages("insee")
```

On peut ensuite lancer le package pour l'utiliser avec *library()* et on ajoute le package *tidyverse* que l'INSEE semble privilégier pour l'exploitation des données :

```
library(insee)
library(tidyverse,warn.conflicts = F)
```

1.3.2 Chargement de la liste des tableaux

On commence par télécharger le catalogue des tableaux de données disponibles,à l'aide de la commande *get_dataset_list()*

```
catalogue = get_dataset_list()
kable(head(catalogue))
```

id	Name.fr
BALANCE-PAIEMENTS	Balance des paiements
CHOMAGE-TRIM-NATIONAL	Chômage, taux de chômage par sexe et âge (sens BIT)
CLIMAT-AFFAIRES	Indicateurs synthétiques du climat des affaires
CNA-2010-CONSO-MEN	Consommation des ménages - Résultats par produit, fonction et durabilité
CNA-2010-CONSO-SI	Dépenses de consommation finale par secteur institutionnel - Résultats par opération e
CNA-2010-CPEB	Comptes de production et d'exploitation par branche

Chaque tableau comporte un très grand nombre de séries chronologiques parmi lesquelles il faut opérer un choix afin d'extraire exactement ce que l'on veut.

1.3.3 Examen des séries présentes dans un tableau

Une fois que l'on a choisi un tableau, on peut examiner plus en détail les différentes séries qui y sont présentes à l'aide de la commande *get_idbank_list()*. On va par exemple examiner le contenu de la base de données "DECES-MORTALITE" :

```
var<-get_idbank_list("DECES-MORTALITE")
str(var)
```

```
FALSE tibble [1,905 x 39] (S3: tbl_df/tbl/data.frame)
FALSE  $ nomflow          : chr [1:1905] "DECES-MORTALITE" "DECES-MORTALITE" "DECES-MORTALITE" "DECES-MORTA
FALSE  $ idbank             : chr [1:1905] "000436398" "001641606" "000869058" "001780755" ...
FALSE  $ cleFlow          : chr [1:1905] "M.TAUX_MORTALITE.TAUX.TXMORINF.FM.0.SO.SO.BRUT" "M.TAUX_MORTALIT
FALSE  $ FREQ                : chr [1:1905] "M" "M" "A" "A" ...
FALSE  $ INDICATEUR        : chr [1:1905] "TAUX_MORTALITE" "TAUX_MORTALITE" "DECES_DOMICILIES" "DECES_DOMIC
FALSE  $ NATURE              : chr [1:1905] "TAUX" "TAUX" "VALEUR_ABSOLUE" "VALEUR_ABSOLUE" ...
FALSE  $ DEMOGRAPHIE         : chr [1:1905] "TXMORINF" "TXMORINF" "DECES-DOM" "DECES-DOM" ...
FALSE  $ REF_AREA            : chr [1:1905] "FM" "FR-D976" "AU" "F_H_IDF" ...
FALSE  $ SEXE                : chr [1:1905] "0" "0" "SO" "SO" ...
FALSE  $ AGE                 : chr [1:1905] "SO" "SO" "SO" "SO" ...
FALSE  $ UNIT_MEASURE        : chr [1:1905] "SO" "SO" "NOMBRE" "NOMBRE" ...
FALSE  $ CORRECTION          : chr [1:1905] "BRUT" "BRUT" "BRUT" "BRUT" ...
FALSE  $ FREQ_label_fr       : chr [1:1905] "Mensuelle" "Mensuelle" "Annuelle" "Annuelle" ...
FALSE  $ FREQ_label_en       : chr [1:1905] "Monthly" "Monthly" "Annual" "Annual" ...
FALSE  $ INDICATEUR_label_fr  : chr [1:1905] "Taux de mortalité" "Taux de mortalité" "Décès domiciliés" "Déc
FALSE  $ INDICATEUR_label_en  : chr [1:1905] "Mortality rate" "Mortality rate" "Deaths domiciled" "Deaths do
FALSE  $ NATURE_label_fr     : chr [1:1905] "Taux" "Taux" "Valeur absolue" "Valeur absolue" ...
FALSE  $ NATURE_label_en     : chr [1:1905] "Rate" "Rate" "Absolute value" "Absolute value" ...
FALSE  $ DEMOGRAPHIE_label_fr : chr [1:1905] "Taux de mortalité infantile" "Taux de mortalité infantile" "Dé
FALSE  $ DEMOGRAPHIE_label_en : chr [1:1905] "Infant mortality rate" "Infant mortality rate" "Deaths of all a
FALSE  $ REF_AREA_label_fr   : chr [1:1905] "France métropolitaine" "France hors Mayotte" "Territoires d'out
FALSE  $ REF_AREA_label_en   : chr [1:1905] "Metropolitan France" "France excluding Mayotte" "French oversea
FALSE  $ SEXE_label_fr       : chr [1:1905] "Ensemble" "Ensemble" "Sans objet" "Sans objet" ...
FALSE  $ SEXE_label_en        : chr [1:1905] "All" "All" "Not applicable" "Not applicable" ...
FALSE  $ AGE_label_fr      : chr [1:1905] "Sans objet" "Sans objet" "Sans objet" "Sans objet" ...
FALSE  $ AGE_label_en      : chr [1:1905] "Not applicable" "Not applicable" "Not applicable" "Not applicable
FALSE  $ UNIT_MEASURE_label_fr: chr [1:1905] "sans objet" "sans objet" "nombre" "nombre" ...
FALSE  $ UNIT_MEASURE_label_en: chr [1:1905] "not applicable" "not applicable" "number" "number" ...
FALSE  $ CORRECTION_label_fr  : chr [1:1905] "Non corrigé" "Non corrigé" "Non corrigé" "Non corrigé" ...
FALSE  $ CORRECTION_label_en  : chr [1:1905] "Uncorrected" "Uncorrected" "Uncorrected" "Uncorrected" ...
FALSE  $ dim1                : chr [1:1905] "M" "M" "A" "A" ...
FALSE  $ dim2           : chr [1:1905] "TAUX_MORTALITE" "TAUX_MORTALITE" "DECES_DOMICILIES" "DECES_DOMICIL
FALSE  $ dim3              : chr [1:1905] "TAUX" "TAUX" "VALEUR_ABSOLUE" "VALEUR_ABSOLUE" ...
FALSE  $ dim4               : chr [1:1905] "TXMORINF" "TXMORINF" "DECES-DOM" "DECES-DOM" ...
FALSE  $ dim5                : chr [1:1905] "FM" "FR-D976" "AU" "F_H_IDF" ...
FALSE  $ dim6                : chr [1:1905] "0" "0" "SO" "SO" ...
FALSE  $ dim7                : chr [1:1905] "SO" "SO" "SO" "SO" ...
```

```
FALSE  $ dim8                  : chr [1:1905] "SO" "SO" "NOMBRE" "NOMBRE" ...
FALSE  $ dim9                  : chr [1:1905] "BRUT" "BRUT" "BRUT" "BRUT" ...
```

Le résultat est un tibble comportant 1905 lignes et 39 colonnes. Il correspond en pratique aux 1905 séries chronologiques que l'on peut extraire de la base de données. Chaque série dispose d'un code unique contenu dans la variable *idbank*.

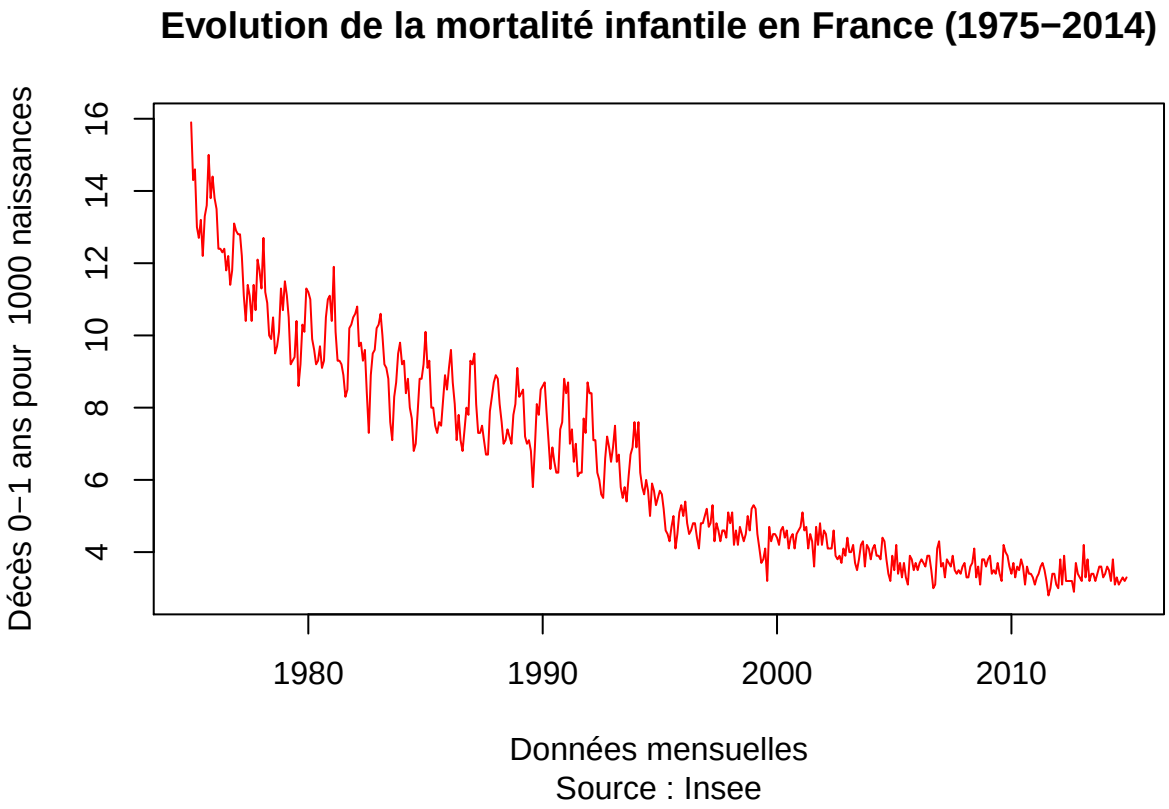
1.3.4 Extraction d'une série à l'aide de son identifiant

Une première solution pour extraire une série consiste à parcourir le tableau des variables jusqu'à repérer la ligne qui nous intéresse puis à noter son idbank et à extraire la série correspondante à l'aide de la fonction **get_insee_idbank()**. Par exemple, la première ligne du tableau des variables dont le code est "000436398" va renvoyer un tableau du taux brut de mortalité infantile en France métropolitaine de Janvier 1975 à Décembre 2014. On peut en faire rapidement un graphique avec la fonction *plot()* de R-Base

```
don<-get_insee_idbank("000436398")
```

```
FALSE  |                                                                      |
```

```
don<-don[order(don$DATE),1:3]
plot(don$DATE,don$OBS_VALUE,
     type ="l",
     col="red",
     ylab = "Décès 0-1 ans pour  1000 naissances",
     xlab = "Données mensuelles",
     main = "Evolution de la mortalité infantile en France (1975-2014)",
     sub = "Source : Insee")
```



On remarque que la courbe a des oscillations saisonnières beaucoup moins fortes après 1995 ce qui est sans doute lié à un changement dans le mode de collecte des données plutôt qu'à la réalité.

On note aussi que les données s'arrêtent en 2014 ce qui est bizarre puisque l'API devrait nous donner les chiffres les plus récents. en fait les données plus récentes sont disponibles mais elles font partie d'une autre série de données.

1.3.5 Extraction d'un ensemble de séries d'un même tableau

Supposons que l'on veuille extraire trois courbes décrivant l'espérance de vie des hommes en France métropolitaine, à 20, 40 et 60 ans. Nous lançons alors une requête pour ne retenir dans le tableau des variables que les lignes qui nous intéressent.

```
sel  =
  get_idbank_list("DECES-MORTALITE") %>%
  filter(SEXE == "1") %>%
  filter(FREQ == "A") %>% #données annuelles
  filter(REF_AREA == "FM") %>% #France métropolitaine
  filter(DEMOGRAPHIE %in% c("ESPV-20","ESPV-40","ESPV-60")) # Espérance de vie

kable(head(sel))
```

nomflow	idbank	cleFlow
DECES-MORTALITE	001686948	A.ESPERANCE_VIE.VALEUR_ABSOLUE.ESPV-20.FM.1.SO.ANNEES.BRUT
DECES-MORTALITE	001686949	A.ESPERANCE_VIE.VALEUR_ABSOLUE.ESPV-40.FM.1.SO.ANNEES.BRUT
DECES-MORTALITE	001686950	A.ESPERANCE_VIE.VALEUR_ABSOLUE.ESPV-60.FM.1.SO.ANNEES.BRUT
DECES-MORTALITE	010536470	A.ESPERANCE_VIE.VALEUR_ABSOLUE.ESPV-20.FM.1.SO.ANNEES.BRUT
DECES-MORTALITE	010536474	A.ESPERANCE_VIE.VALEUR_ABSOLUE.ESPV-40.FM.1.SO.ANNEES.BRUT
DECES-MORTALITE	010536478	A.ESPERANCE_VIE.VALEUR_ABSOLUE.ESPV-60.FM.1.SO.ANNEES.BRUT

On découvre que le programme renvoie **6 lignes au lieu de 3**. Pourquoi ? Parce que l'INSEE stocke différemment des séries anciennes et des séries récentes. Il faut donc effectuer une requête sur les 4 codes à la fois pour avoir la série la plus longue.

1.3.6 Recupération et nettoyage des données

On récupère les données puis on procède à un petit nettoyage du tableau pour ne conserver que les colonnes utiles.

```
don = get_insee_idbank(sel$idbank)
```

```
FALSE  |                                                  |
```

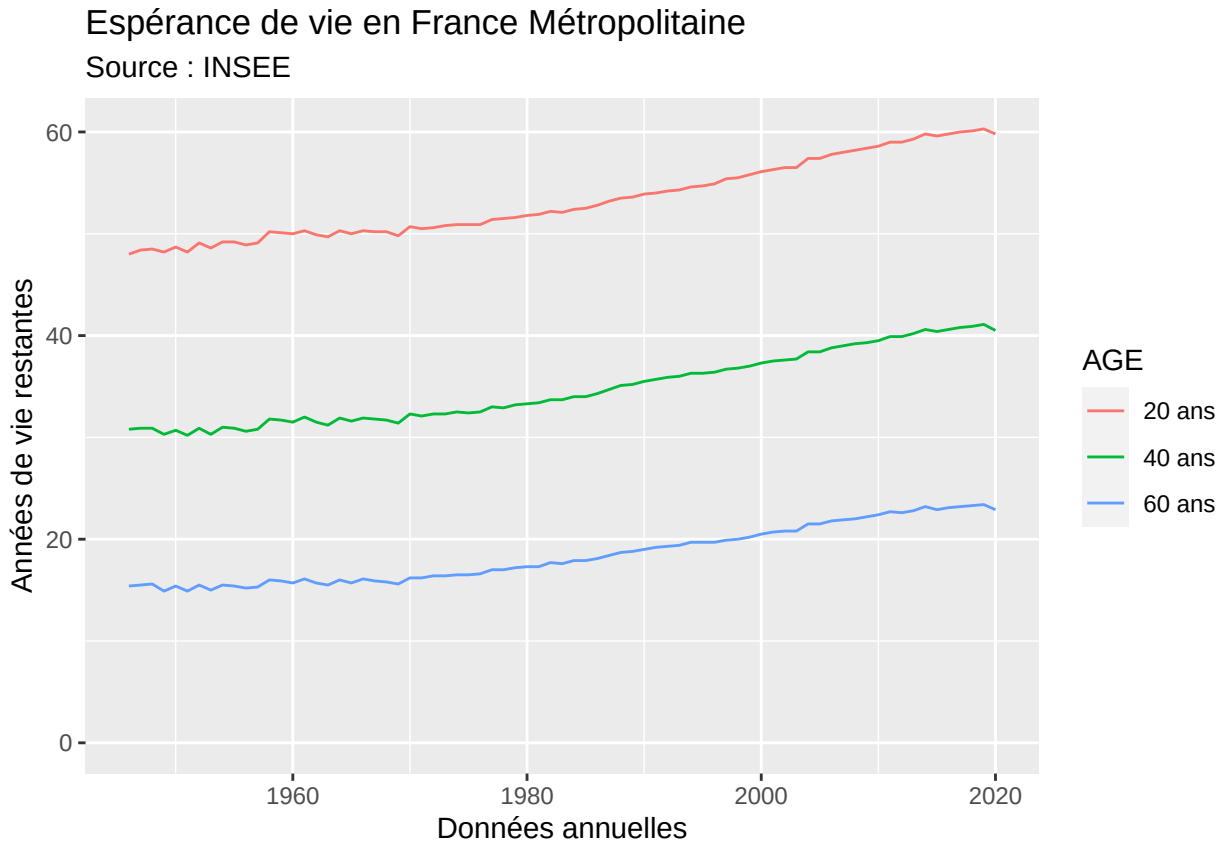
```
don2<-don %>% select(ANNEE = DATE, ESPVIE= OBS_VALUE, AGE = TITLE_FR) %>%
              mutate(AGE = as.factor(AGE)) %>%
              arrange(AGE, ANNEE)
levels(don2$AGE) <- c("20 ans", "40 ans","60 ans")
kable(head(don2))
```

ANNEE	ESPVIE	AGE
1946-01-01	48.0	20 ans
1947-01-01	48.4	20 ans
1948-01-01	48.5	20 ans
1949-01-01	48.2	20 ans
1950-01-01	48.7	20 ans
1951-01-01	48.2	20 ans

1.3.7 Construction d'un graphique

On peut maintenant construire notre graphique à l'aide par exemple de `ggplot2` :

```
p<-ggplot(don2) +
   aes(x=ANNEE,y=ESPVIE, color = AGE) +
    geom_line() +
    ggtitle(label= "Espérance de vie en France Métropolitaine",
            subtitle = "Source : INSEE")+
    scale_x_date("Données annuelles") +
    scale_y_continuous("Années de vie restantes",limits = c(0,NA))
p
```



1.3.8 Discussion

Comme on peut le voir, l'utilisation d'un package simplifie l'usage des API mais ne dispense pas d'un apprentissage souvent long pour comprendre toutes les finesses du package (et parfois ses bugs ...). Dans le cas du package INSEE, l'utilisation s'avère assez lourde mais permet d'accéder à un nombre considérable de données !

1.4 Exercices

1.4.1 Exercice 1 : utilisation de httr et jsonlite

Déterminer la durée et la date des 10 prochains dates de passage de l'ISS au dessus de Paris (Latitude = 48.86, Longitude = 2.35)

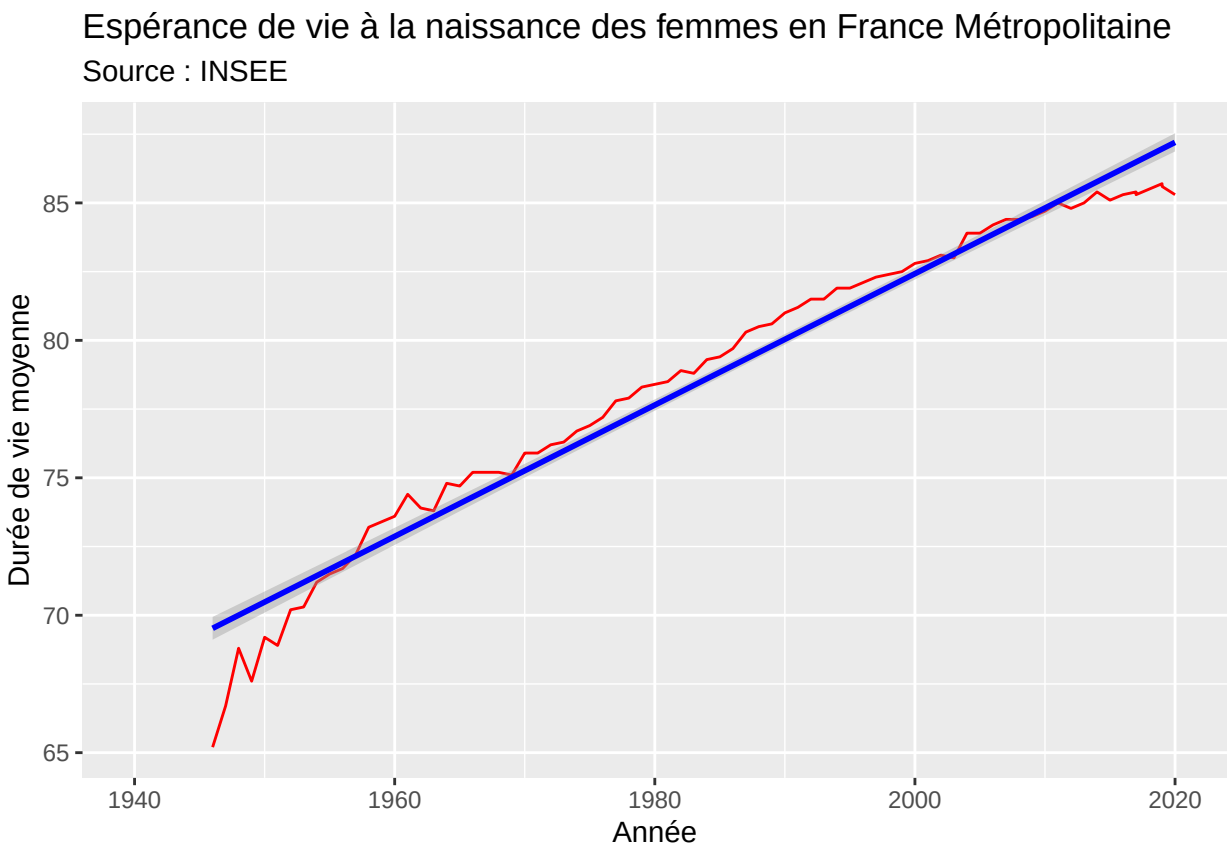
Table 1.3: Prochains passages de l'ISS au dessus de Paris

duration	risetime
657	2021-02-09 14:45:07
625	2021-02-09 16:22:00
366	2021-02-09 18:00:03
541	2021-02-10 09:08:27
651	2021-02-10 10:43:48
654	2021-02-10 12:20:39
656	2021-02-10 13:57:42
643	2021-02-10 15:34:34
478	2021-02-10 17:12:02
468	2021-02-11 08:21:56

1.4.2 Exercice 2 : utilisation du package 'insee'

Construire à l'aide du package INSEE un graphique de l'évolution mensuelle de l'espérance de vie des femmes à la naissance en France Métropolitaine de 1945 à 2020.

```
FALSE  |                                                                      |
```



1.4.3 Exercice 3 : Exploration de nouvelles API

Vous devez identifier une API intéressante, accessible soit par un package R, soit par une combinaison de comandes GET() puis montrer son utilisation à l'aide d'un exemple de création d'un tableau puis d'un graphique.

Vous présenterez le résultat sous la forme d'un document markdown d'une à deux pages maximum.