

# Projet IF36

Baptiste Toussaint

2024-05-05

## Analyse : GitHub Public Repository Metadata

Projet de *Data visualization* de l'unité d'enseignement IF36 de l'[Université de Technologie de Troyes \(UTT\)](#).

Nom du groupe : ***La cité de la f eur.***

Membres : [Baptiste Toussaint](#), [XU Shilun](#), [Louis Duhal Berruer](#)

Langue : Français

---

### I. Introduction

#### 1. Description des données.

## GitHub

Figure 1: Logo de la mascotte de GitHub



Figure 2: Logo de GitHub

##### 1.1. Présentation de GitHub

Logo de GitHub et de sa mascotte : Octocat

*GitHub* est un service web à destination des développeurs permettant d'héberger, partager, et versionner du code. Initialement conçu comme outil complémentaire à *Git* pour le contrôle de version, *GitHub* est aujourd'hui une véritable plateforme de développement utilisée par plus de 100 millions d'utilisateurs à travers le monde.

*GitHub* voit le jour en 2008 et s'impose rapidement comme l'outil d'hébergement en ligne privilégié des développeurs, en particulier dans la communauté des projets open source et libres de droit.

La plateforme propose une offre gratuite performante permettant à n'importe quelle équipe de créer et faire vivre leurs projets en ligne. Elle propose aussi des offres payantes pour les entreprises de plus grande taille.

Ainsi, il est possible de retrouver sur GitHub le code de grandes compagnies comme [Google](#) ou [Microsoft](#).

*GitHub*, les utilisateurs créent des **dépôts** (en anglais *repositories* ou *repo* pour la version courte) qui accueillent l'ensemble des fichiers d'un programme, un logiciel, etc.

Le 8 novembre dernier, l'entreprise *GitHub* déclarait alors près de 420 millions de projets présents sur la plateforme, dont 284 millions publics.

**1.2. Les métadonnées de l'API GitHub** GitHub met à disposition une [API](#) permettant aux utilisateurs d'automatiser une série de tâches en lien avec le service. Cette API peut notamment être utilisée pour récupérer des informations relatives à la télémétrie des dépôts : nom du dépôt, nombre de contributeur, dates des contributions, etc.

L'utilisateur [Peter Elmers](#) a développé un [script](#) qui utilise cette API pour extraire les informations (*scraping*) d'environ trois millions de *repo* publics.

Il est important de noter que le fichier JSON de Peter Elmers ne contient que des repos publiques (visibles de tous) possédants au minimum 5 étoiles (l'équivalent des *like* sur la plateforme *github*).

Le résultat de ce script est un fichier JSON hébergé sur la plateforme [Kaggle](#).

La dernière mise à jour du jeu de donnée date du 25 février 2024. Les données sont donc actuelles et toujours pertinentes à analyser.

De plus, le jeu de données est partagé sous license : “Attribution 4.0 International (CC BY 4.0)”.

Pour faciliter la manipulation des données, nous avons décidé de ne conserver que les 200000 premières entrées de ce fichier JSON pour le moment. Cela correspond aux 200000 repos possédant le plus d'*étoiles* sur la plateforme (les repos les plus *likés* si l'on peut se permettre cette analogie).

Nous pourrons sans problème ajouter des données durant notre analyse si nous en ressentons le besoin.

Nous n'excluons pas non plus de recourir à l'API de GitHub pour augmenter les données pour améliorer l'analyse.

Afin de résoudre le problème des fichiers json trop lourd, nous avons décidé d'exécuter le code python suivant dans un notebook de sur Kaggle pour exporter les données par sections en format de CSV:

```
import pandas as pd

# read json file
json_file_path = '/kaggle/input/github-repository-metadata-with-5-stars/repo_metadata.json'
df = pd.read_json(json_file_path)

# every chunk row
chunk_size = 10000

# calculate rows
total_rows = df.shape[0]

# calculate the number of files
num_files = (total_rows + chunk_size - 1) // chunk_size

# split and output files
for i in range(num_files):
    start_idx = i * chunk_size
    end_idx = min((i + 1) * chunk_size, total_rows)
```

```

# get current data
chunk_df = df.iloc[start_idx:end_idx]

# generate file path
output_file_path = f'/kaggle/working/data_{i + 1}.csv'

# output data to csv
chunk_df.to_csv(output_file_path, index=False)

print(f'File {i + 1} written to {output_file_path}')

```

Nous avons donc dix fichiers nommés `data_X.csv` numérotés de 1 à 10. Les dix premiers fichiers sont rangés dans le dossier : `githubStar1-10`, les dix suivants sont rangés dans le dossier : `githubStar11-20`.

Nous avons ensuite regroupé les données de ces vingt fichiers dans un unique fichier nommé `data_1_200000.csv` pour en faciliter la manipulation.

Ces fichiers se trouvent dans le dossier `./data/` de ce dépôt.

**1.3. Présentation des données** Le jeu de données contient donc environ trois millions de dépôts publics, tous ayant plus de cinq étoiles (nous reviendrons plus loin sur la signification de ces étoiles).

Chaque objet du fichier JSON représente un dépôt décrit par les variables suivantes (description mise en ligne par Peter Elmer [ici](#)).

- `owner`, propriétaire ou créateur du dépôt, identifié par son nom d'utilisateur sur la plateforme GitHub. [type : `string`] ;
- `name`, nom du dépôt. [type : `string`] ;
- `stars`, nombre d'étoiles, de *like* du dépôt. [type : `int`] ;
- `forks`, nombre de *fork* du projet. [type : `int`] ;
- `watchers`, nombre d'utilisateurs surveillant le projet. [type : `int`] ;
- `isFork`, précise si le dépôt est un *fork* d'un autre dépôt. [type:`bool`] ;
- `isArchived`, précise si le dépôt est archivé ou non. [type :`bool`] ;
- `languages`, une structure de type `list` regroupant les langages de programmation utilisés dans le projet. Chaque élément de la liste est composé du nom du langage et de la taille en octet, consacré à ce langage dans le dépôt. [type : `list`] ;
- `languageCount`, nombre de langage utilisés. [type : `int`] ;
- `topics`, une structure de type `list` regroupant les *topics* associés au dépôt. Pour chaque *topic* on retrouve le nom et le nombre d'étoiles associées pour ce *topic* sur l'ensemble de la plateforme. Les *topic* permettent aux créateurs des dépôts d'identifier les objectifs ou catégories de leurs projets. Cela ressemble au système des *hashtag* popularisé par *Twitter*. [type : `list`] ;
- `topicCount`, nombre de *topic* associés au dépôt. [type : `int`] ;
- `diskUsageKb`, taille du dépôt en kB. [type : `int`] ;
- `pullRequests`, nombre de *pull request*. [type : `int`] ;
- `issues`, nombre d'*issues*. [type : `int`] ;
- `description`, description du *repo*. [type : `string`] ;
- `primaryLanguage`, nom du langage de programmation principalement dans le projet. [type : `string`] ;

- **createdAt**, date de création du dépôt. La date est au format : “AAAA-MM-JJTHH:MM:SSZ”, par exemple : “2015-03-14T22:35:11Z”. Attention cependant, il faudra préciser quel fuseau horaire est utilisé par l’API pour ne pas fausser n’analyse. [type : **string**] ;
- **pushedAt**, dernière date de *push* du projet, soit la dernière date de mise à jour du dépôt par un contributeur. Le format est identique à l’attribut **createdAt**. [type : **string**] ;
- **defaultBranchCommitCount**, nombre de *commit* sur la *branche principale*. Nous rentrerons dans l’explication complète du système de *commit* dans l’analyse des données. À ce stade on peut approximer le nombre de *commit* comme le nombre de version du projet. [type : **int**] ;
- **license**, licence utilisée par le projet. Permet de connaître les droits donnés aux utilisateurs.
- **assignableUserCount**, nombre d’utilisateurs ayant des droits d’accès sur le projet. [type : **int**] ;
- **codeOfConduct**, si le projet possède un code de bonne conduite pour ses utilisateurs (règles de communauté), mentionne son nom. [type : **string**] ;
- **forkingAllowed**, indique s’il est possible de *fork* le projet : s’il est possible d’en faire une copie. Indique vrai ou faux : **TRUE** ou **FALSE**. [type : **bool**] ;
- **nameWithOwner**, concaténation du nom du dépôt avec le nom du créateur. [type : **string**] ;
- **parent**, indique le nom du dépôt parent si ce dépôt est un *fork*. [type : **string**].

**1.4. Pourquoi étudier ces données ?** Nous sommes trois étudiants de l’UTT avec un parcours tourné vers l’informatique, les nouvelles technologies et la programmation.

GitHub est une plateforme que nous utilisons personnellement (en plus de l’utiliser pour ce projet) de manières différentes : les langages de programmation utilisées dans la branche ISI de l’UTT sont différents des langages utilisés en branche GI par exemple.

L’avantage de ce jeu de données est de permettre de présenter les langages de programmations utilisés dans les projets les plus “populaires” de la plateforme et pourquoi pas de comparer nos résultats avec nos langages préférés.

Ce travail d’analyse des tendances est déjà fait par GitHub chaque années à travers ces articles [Octoverse report](#).

---

## 2. Plan d’analyse

**2.1 Objectifs de l’analyses** Nous possédons donc les entrées relatives aux 200000 dépôts les plus *likés* de GitHub (ceux ayant obtenu les plus d’étoiles : *stars* ).

Dans un premier temps, nous nous demanderons comment se répartissent les *stars* de notre population.

On peut s’attendre à ce qu’une petite partie de projet possèdent un nombre important d’étoile, et ce nombre décroît rapidement, comme une exponentielle décroissante.

Si l’on s’en tient au principe de [Pareto](#) (principe du 80/20), on peut penser que 20% des dépôts sur GitHub concentrent 80% du nombre total d’étoiles attribués par les utilisateurs. Cette hypothèse n’est pas déraisonnable, puisque que ce principe est applicable dans beaucoup de domaines.

Bien que le principe de [Pareto](#) ne soit pas une règle parfaite, c’est en général une première approche naïve utile pour appréhender des phénomènes.

Si l’on considère GitHub comme un réseau social, et que l’on considère que les *stars* d’un projet mesurent sa “popularité”, nous chercherons dans la suite de l’étude à trouver ce qui peut expliquer la popularité d’un projet sur *GitHub*.

Nous allons donc lier la caractéristique du nombre d’étoile avec les autres paramètres de notre jeu de données.

**2.2 Mesure de la popularité** Nous disposons de la date de création des dépôts (notre jeu de données contient des dépôts créés entre 2009 et 2023), nous pourrons chercher s'il existe une corrélation entre la date de création et la popularité du projet. En effet on peut penser que les dépôts les plus anciens sont les dépôts les plus appréciées.

Avec le nombre de *fork* (ou clonage en français) par dépôt nous pourront essayer de voir si les projets les plus aimés sont aussi les projets les plus repris par les utilisateurs.

Quand un utilisateur *fork* un dépôt, il en crée une copie personnelle sur laquelle il est libre d'ajouter les modifications qu'il souhaite.

Le nombre de clonage peut être un indicateur supplémentaire de popularité et on peut chercher à savoir si cet indicateur est corrélé au nombre d'étoile d'un projet.

Le paramètre *watchers* est une donnée supplémentaire d'analyse de "popularité". Quand on utiliseur *watchs* (que l'on peut traduire par surveiller ou suivre, en français) un projet, il est averti quand ce dernier subit une modification.

Là encore, on pourra chercher à vérifier l'existence d'une corrélation entre le nombre d'étoile, le nombre de clone, et le nombre de suivi des dépôts.

**2.3 Mesure des contributions** GitHub est énormément utilisé par les projets collaboratifs pour permettre aux développeurs du monde entier de contribuer à des projets.

Selon leurs politiques de gouvernance, les propriétaires de dépôts peuvent permettre à certains utilisateurs autorisés ou à n'importe quel contributeur de proposer des modifications (les *pullRequests*) ou signaler des bugs (les *issues*).

On dispose donc de ces deux indicateurs qui permettent de quantifier l'interactivité d'un projet : plus ces deux valeurs sont grandes, plus on peut considérer le projet comme actif.

Nous ne disposons cependant pas du nombre de contributeurs par dépôt. L'attribut : `assignableUserCount` semble uniquement représenter le nombre de contributeurs ayant des droits d'accès spécifiques.

Nous pourrons par exemple utiliser l'API pour augmenter notre jeu de données et trouver pour chaque dépôt le nombre de contributeur réel.

**2.4 Étude des langages de programmation utilisés** Nous pourrons ensuite analyser les types de langage utilisés.

Nous chercherons à faire une cartographie des langages les plus utilisés et nous pourrons peut-être lier cette analyse avec la popularité des dépôts : existe-t-il des langages plus populaires que les autres ?

On peut supposer par exemple que les dépôts très populaires ne contiennent pas de *Pascal* ou d'*Ada*...

**2.5 Prise en compte des *topics*** Toute notre analyse jusqu'ici repose sur une hypothèse : les dépôts GitHub ne contiennent que des projets de programmation ou de code.

Cette hypothèse est fausse ! En effet, avec sa démocratisation, GitHub a connu une diversification des usages.

Aujourd'hui en tant que véritable réseau social pour développeurs et geek en tous genres, certains utilisent la plateforme pour créer des portfolios en ligne, des pages vitrines pour un site ou un service, etc.

Par exemple, le dépôt : [papers-we-love](#) est une immense compilation de papiers scientifiques relatifs à l'informatique.

Il est donc très difficile de comparer ce genre de dépôt avec des projets informatiques.

L'attribut `topics` du dataset peut nous aider à classifier les dépôts selon leurs objectifs et catégories identifiées : projets, vitrines, listes, cours, etc.

Il sera possible d'effectuer une analyse croisée entre les langages et les *topics* pour observer quels *topics* sont associés à quels langages.

Le créateur du dataset, Peter Elmer, propose sur la page kaggle une représentation de type *word clouds* montrant les *topics* les plus représentés par langage de programmation.

Nous pourrons essayer de reproduire ce résultat, d'autant qu'il n'a pas partagé le code correspondant.

## II. Exploration

### 1. Étape intermédiaire manipulations : charger le dataset

```
# Chagre les librairies utiles
library(tidyverse)

## -- Attaching core tidyverse packages ----- tidyverse 2.0.0 --
## v dplyr     1.1.4     v readr     2.1.5
## vforcats   1.0.0     v stringr   1.5.1
## v ggplot2   3.5.0     v tibble    3.2.1
## v lubridate 1.9.3     v tidyr    1.3.1
## v purrr    1.0.2
## -- Conflicts ----- tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()   masks stats::lag()
## i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to become errors
library(ggrepel)
```

Dans le dossier .\data\githubStar1-10\ on dispose de 10 fichiers .csv qui possèdent 10000 éléments chacun soit 100000 éléments.

Je propose un petit script pour fusionner ces fichiers en un seul fichier commun pour simplifier la manipulation :

```
# A n'exécuter qu'une fois !

# Notre dossier data contient 2 sous dossiers:
#   - 1-10 qui contient les fichiers .csv de 1 à 10
#   - 11-20 qui contient les fichiers .csv de 11 à 20
# On fait tourner une routine pour fusionner les 20 fichiers dans un seul fichier de données.

# Fichiers du dossier de 1 à 10
list_of_csv_file1 <- list.files("./data/githubStar1-10/", full.names = TRUE)
# Fichiers du dossier de 11 à 20
list_of_csv_file2 <- list.files("./data/githubStar11-20/", full.names = TRUE)
list_of_csv_file <- c(list_of_csv_file1, list_of_csv_file2)

df <- readr::read_csv(list_of_csv_file[1], show_col_types = FALSE)
for (file in list_of_csv_file[-1]) {
  df <- rbind(df, readr::read_csv(file, show_col_types = FALSE))
}

readr::write_csv(df, "./data/data_1_200000.csv")
```

Pour vérifier :

```
# Si besoin de charger le dataset
df <- read_csv("./data/data_1_200000.csv")
```

```

## Rows: 200000 Columns: 25
## -- Column specification -----
## Delimiter: ","
## chr   (9): owner, name, languages, topics, description, primaryLanguage, lic...
## dbl   (10): stars, forks, watchers, languageCount, topicCount, diskUsageKb, p...
## lgl   (4): isFork, isArchived, forkingAllowed, parent
## dttm  (2): createdAt, pushedAt
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.

```

## 2. Traitement des doublons

Pour commencer nous devons nettoyer le dataset en identifiant les possibles doublons.

```
dim(df)[1] - dim(unique(df))[1] # Nombre de doublons
```

```
## [1] 5804
```

Nous avons à priori 5804 doublons.

```
df_duplicated <- df[duplicated(df),] # stock les doublons dans un objet
head(df_duplicated) # Montre les premières lignes des doublons
```

```

## # A tibble: 6 x 25
##   owner      name  stars forks watchers isFork isArchived languages languageCount
##   <chr>     <chr> <dbl> <dbl>    <dbl> <lgl>   <lgl>   <chr>           <dbl>
## 1 FFmpeg    FFmp~ 37824 11218    1379 FALSE  FALSE   [{}'name'~          15
## 2 labstack   echo  26331 2180     523 FALSE  FALSE   [{}'name'~          3
## 3 alibaba    canal 26312 7356    1198 FALSE  FALSE   [{}'name'~          11
## 4 floating~ floa~ 26311 1523     254 FALSE  FALSE   [{}'name'~          6
## 5 portainer  port~ 26277 2223     471 FALSE  FALSE   [{}'name'~          11
## 6 lukasz-m~ awes~ 26258 2638     988 FALSE  FALSE   []                0
## # i 16 more variables: topics <chr>, topicCount <dbl>, diskUsageKb <dbl>,
## # pullRequests <dbl>, issues <dbl>, description <chr>, primaryLanguage <chr>,
## # createdAt <dttm>, pushedAt <dttm>, defaultBranchCommitCount <dbl>,
## # license <chr>, assignableUserCount <dbl>, codeOfConduct <chr>,
## # forkingAllowed <lgl>, nameWithOwner <chr>, parent <lgl>
df <- unique(df) # retire les doublons au dataset df
n <- dim(df)[1]
m <- dim(df)[2]
```

## 3. Plan de l'analyse

### 4. Analyse du nombre d'étoiles : *stars*

**4.1 Notre interprétation du nombre d'étoile d'un dépôt sur GitHub** Le fichier `data_1_200000.csv` contenu dans l'objet `df` contient les 200000 dépôts possédant le plus d'étoiles (*stars*) sur GitHub au 25 février 2024.

Sur GitHub, les étoiles jouent un rôle similaire à celui des *likes* sur les réseaux-sociaux classiques. Ainsi, on peut partir du postulat que nous disposons d'informations relatives aux *repo* les plus “populaires” de GitHub.

Cette les raisons qui peuvent pousser un utilisateur à *liker* un dépôt sont nombreuses : conserver le projet dans son historique personnel pour y revenir plus tard, signifier que l'on apprécie ce dépôt, etc.

Nous considérerons ici le nombre d'étoile comme un indicateur de "popularité" sans s'attarder sur une définition formelle de la "popularité".

**4.2 Comment se répartissent le nombre d'étoile sur la plateforme ?** La première question que nous pouvons nous poser concerne la répartition du nombre d'étoiles sur GitHub. Si l'on considère le nombre d'étoile comme une variable quantitative aléatoire discrète, comment se répartissent les valeurs de cette variable ? On peut s'attendre à avoir un très petit nombre de projets ayant un grand nombre d'étoiles et la majorité des projets autour d'une dizaine.

Pour rappel, notre jeu de données comporte un biais certain : le créateur à initialement regroupé les dépôts **publics** (donc visibles de tous, ce qui écarte la majorité des dépôts GitHub qui sont privés) ayant 5 étoiles ou plus. On peut supposer que l'écrasante majorité des dépôts du site ont moins de 5 étoiles ( c'est par exemple le cas de l'ensemble des projets présents sur nos GitHub respectifs, étant des projets personnels ou académiques).

Nous avons ensuite choisis de ne conserver que les 200000 premiers dépôts, soit les dépôts ayant le plus d'étoiles.

Nous travaillons donc sur les 200000 projets ayant le plus d'étoiles sur GitHub.

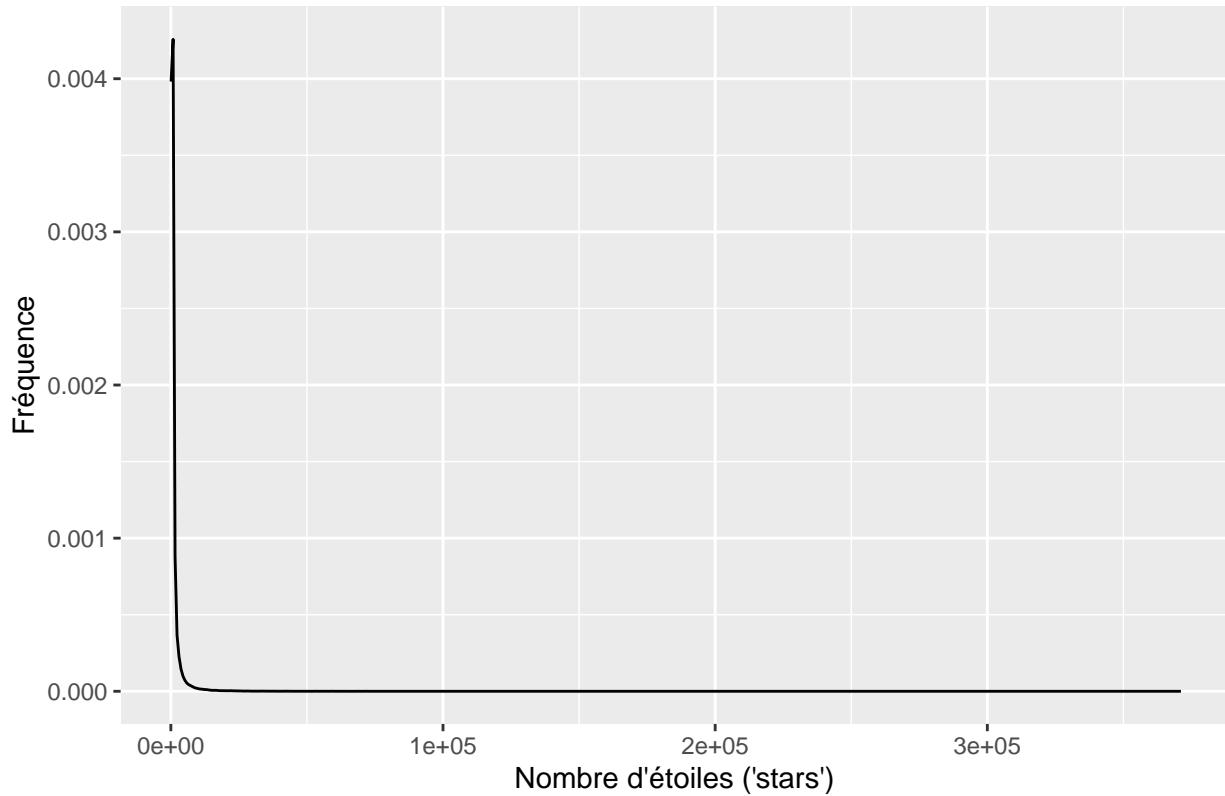
Pour simplifier le travail, je propose de trier le dataset par en fonction du nombre d'étoile dans l'ordre décroissant

```
df <- df %>% arrange(desc(stars))
```

Pour répondre à cette question, on peut utiliser un **Density Plot** pour visualiser la distribution de cette variable.

```
ggplot(data = df) +
  geom_density(aes(x = stars)) +
  labs(x = "Nombre d'étoiles ('stars')",
       y = "Fréquence",
       title = "Répartition du nombre d'étoiles des dépôts")
```

## Répartition du nombre d'étoiles des dépôts



En tant que tel, ce premier graphique n'est pas interprétable. On comprend seulement que certains dépôts ont plus de 300000 étoiles (on peut lire  $3e+05$  sur l'échelle des abscisses) là où la majorité sont proches de  $0e+00$  et ont donc un nombre d'étoiles dans l'ordre de la centaine ou du millier.

De plus, les fréquences indiquées sont difficilement manipulables par le cerveau humain : on préférera afficher le nombre de dépôt (nombre d'entrée dans le dataset) que la fréquence.

On peut identifier les valeurs minimum et maximum d'étoiles de notre dataset :

```
sprintf("Nombre maximum d'étoiles : %d", max(df$stars))
```

```
## [1] "Nombre maximum d'étoiles : 371122"
```

```
sprintf("Nombre minimum d'étoiles : %d", min(df$stars))
```

```
## [1] "Nombre minimum d'étoiles : 97"
```

Le projet ayant le moins d'étoile en possède 97 (la sélection décrite plus haut masque la très grande majorité des dépôts du jeu de donnée initial qui ont donc moins de 97 étoiles). Le projet ayant le plus d'étoile en possède 371122.

Je propose à ce stade de retirer les dépôts ayant plus de 10000 étoiles :

```
df[df$stars > 10000,] %>% nrow()
```

```
## [1] 3089
```

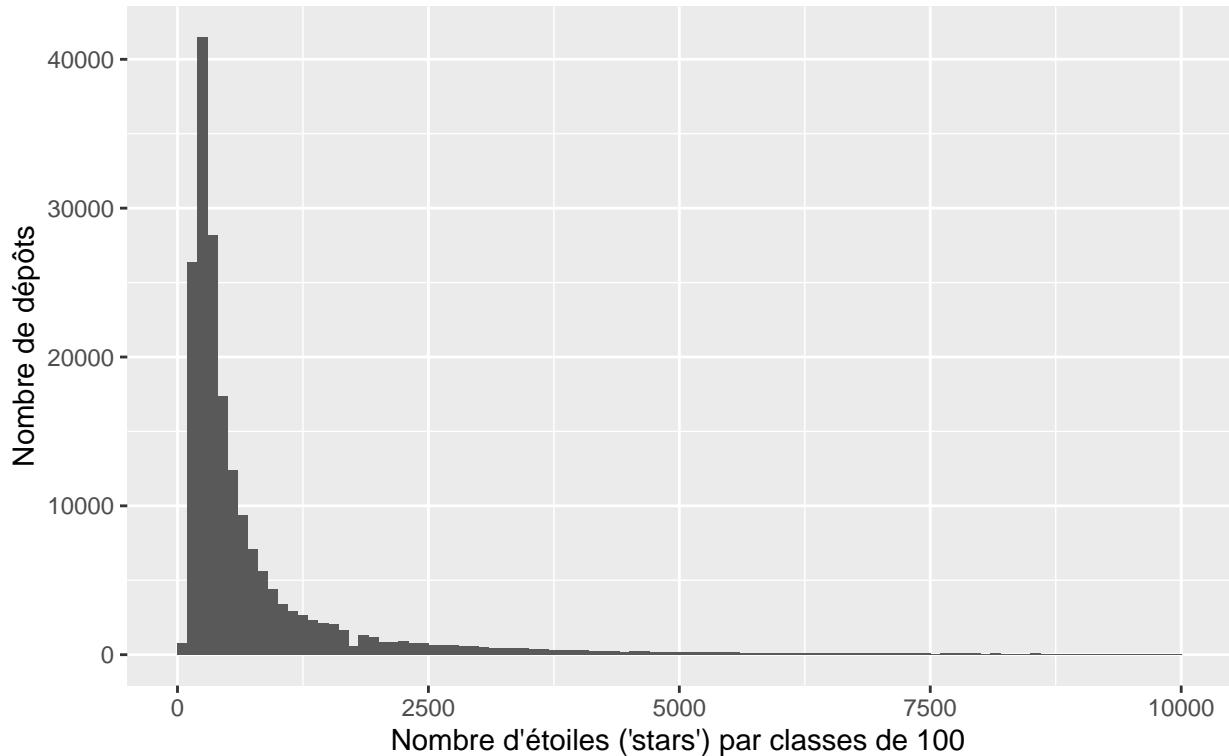
ce qui représente 3089 dépôts sur nos 194196.

Je propose donc de séparer l'analyse en deux : - pour les projets ayant plus de 10000 étoiles ; - pour les projets ayant 10000 étoiles ou moins.

Je propose aussi d'utiliser un histogramme : ce type de graphique est plus adapté dans notre cas car il permet de définir des "classes" pour ranger nos dépôts.

```
# Histogramme avec les dépôts stars <= 10000
ggplot(data = df[df$stars <= 10000,]) +
  # Ajoute la géométrie histogramme
  geom_histogram(aes(x = stars),
                 # breaks sert à définir les séparations des barres
                 breaks = seq(0,10000,100)) +
  labs(x = "Nombre d'étoiles ('stars') par classes de 100",
       y = "Nombre de dépôts",
       title = "Répartition du nombre d'étoiles des dépôts
pour les dépôts avec 10000 étoiles ou moins")
```

Répartition du nombre d'étoiles des dépôts  
pour les dépôts avec 10000 étoiles ou moins



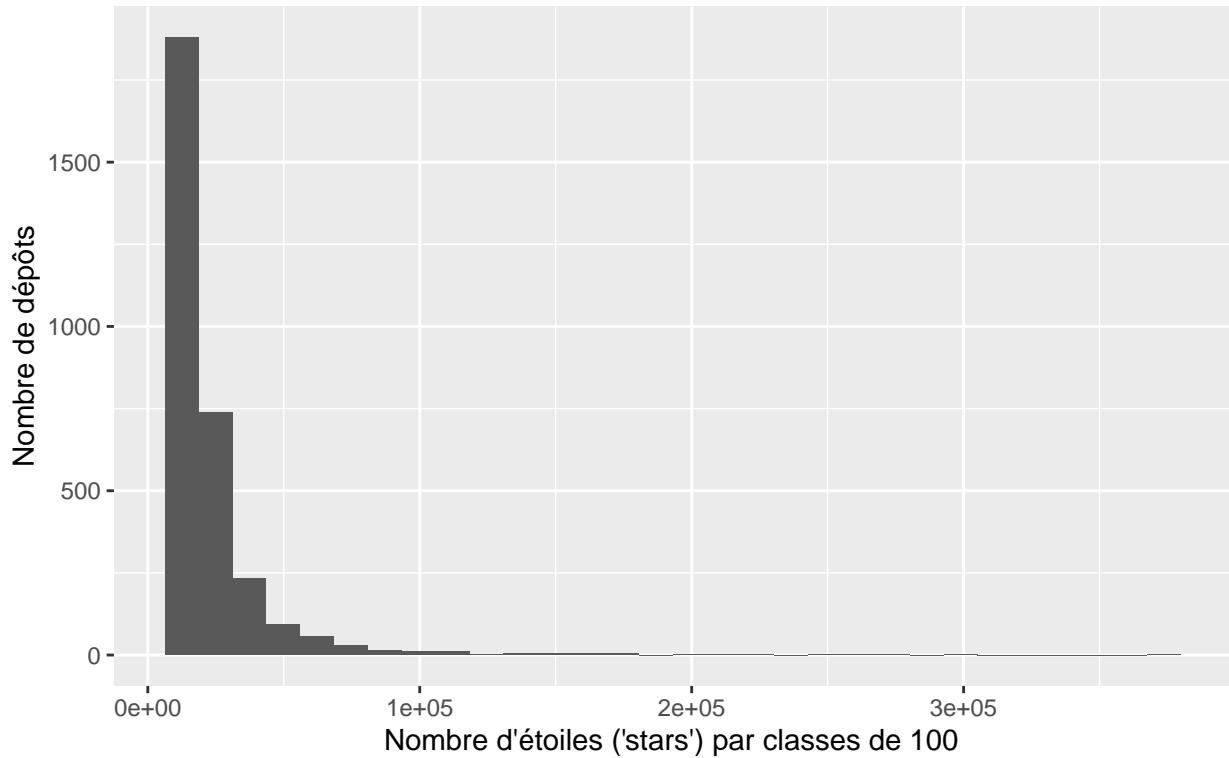
On comprend avec ce second graphique que la majorité des dépôts ont moins de 2500 étoiles et quelque uns en ont plus de 5000.

On peut ensuite afficher uniquement les dépôts ayant plus de 10000 étoiles.

```
ggplot(data = df[df$stars >= 10000,]) +
  # Ajoute la géométrie histogramme
  geom_histogram(aes(x = stars)) +
  labs(x = "Nombre d'étoiles ('stars') par classes de 100",
       y = "Nombre de dépôts",
       title = "Répartition du nombre d'étoiles des dépôts
pour les dépôts avec plus de 10000 étoiles")

## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```

## Répartition du nombre d'étoiles des dépôts pour les dépôts avec plus de 10000 étoiles



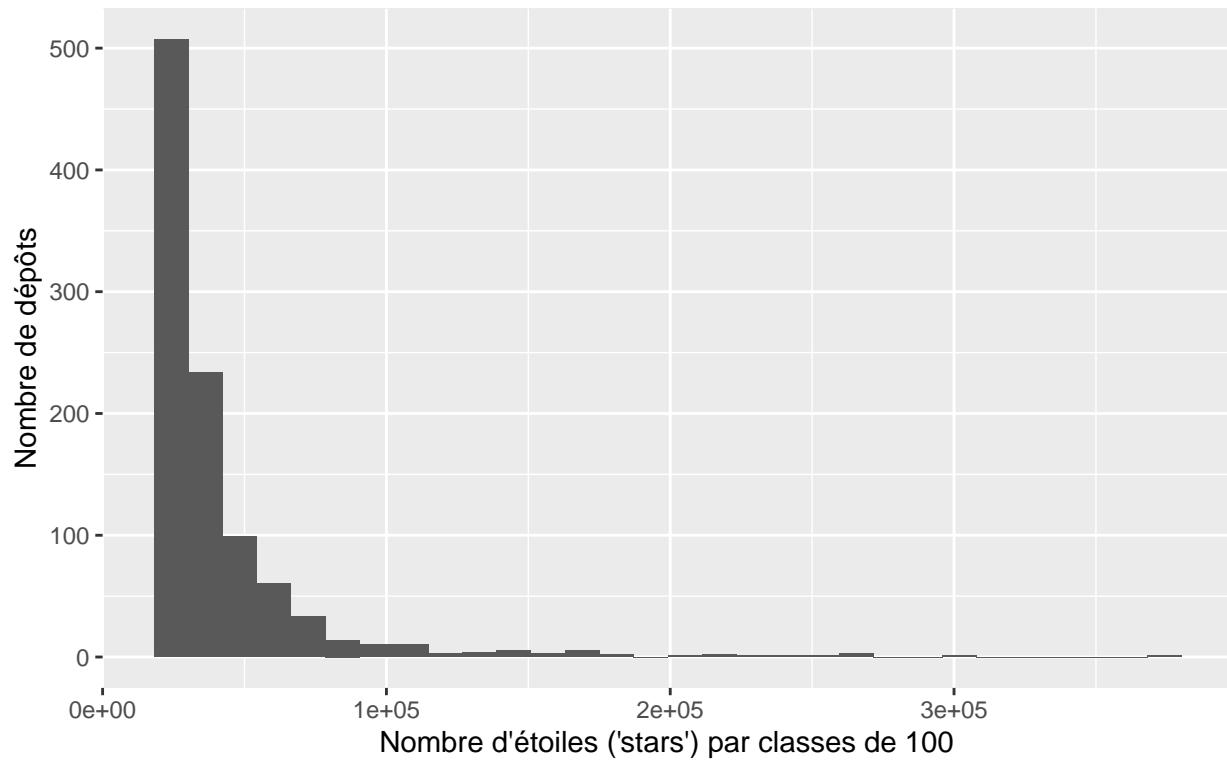
Bien que l'échelle ne soit plus la même (on atteint  $3 \times 10^5$  pour notre dépôt à 371122 étoiles), on retrouve une distribution très similaire.

Si l'on se restreint avec les 1000 dépôts avec le plus d'étoiles on a :

```
# le dataset df est déjà trié par nombre d'étoiles décroissante
ggplot(data = df[1:1000,]) +
  # Ajoute la géométrie histogramme
  geom_histogram(aes(x = stars)) +
  labs(x = "Nombre d'étoiles ('stars') par classes de 100",
       y = "Nombre de dépôts",
       title = "Répartition du nombre d'étoiles des dépôts pour les dépôts avec plus de 10000 étoiles")
```

## `stat\_bin()` using `bins = 30` . Pick better value with `binwidth` .

## Répartition du nombre d'étoiles des dépôts pour les dépôts avec plus de 10000 étoiles



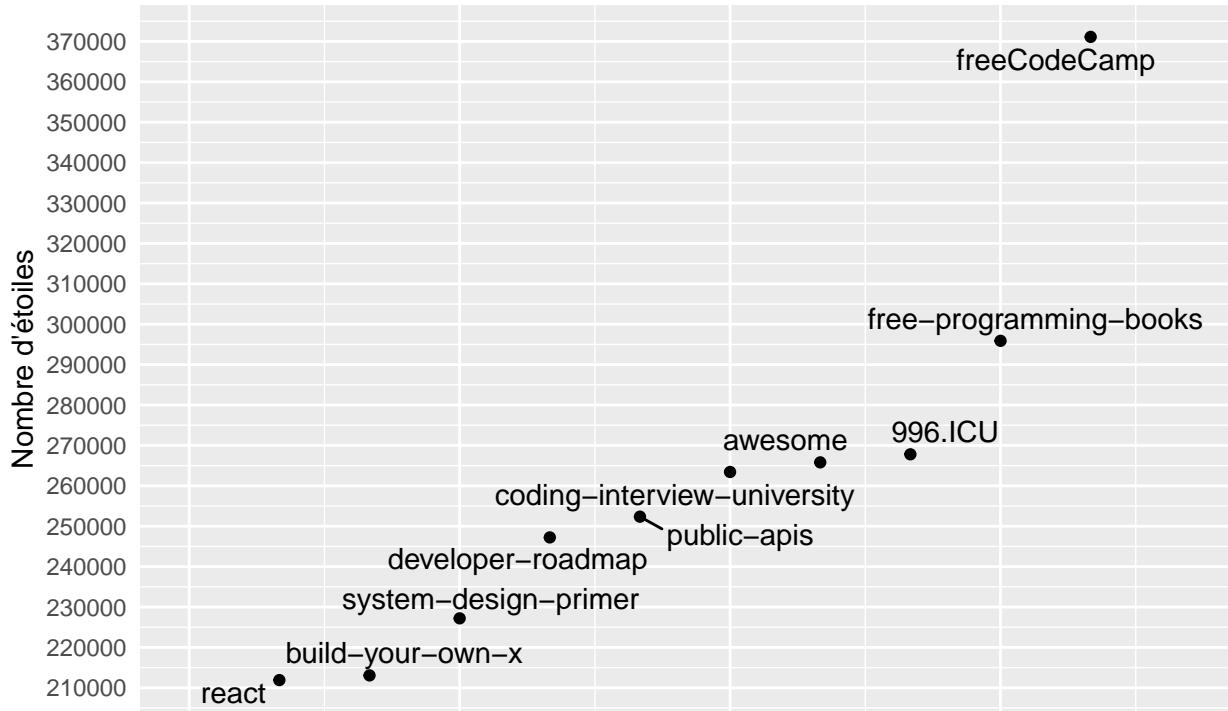
Il semble que l'on retrouve encore une distribution avec une forme similaire.

On peut conclure que notre hypothèse de base était la bonne : la majorité des dépôts ont peu d'étoiles ("peu d'étoile" relativement à notre échantillon, soit entre 97 et 1000) et quelques uns arrivent à monter plus haut, voir pour certains atteindre des centaines de milliers.

Pour appuyer cette remarque on peut regarder les 10 dépôts ayant le plus d'étoiles.

```
library("ggrepel")
ggplot(arrange(df[1:10],stars),
       aes(x = 1:10, y = stars)) +
  geom_point() +
  geom_text_repel(aes(label = name)) +
  # on ajoute une graduation pour ne pas couper le label du 10 élément
  xlim(0,11) +
  labs(title = "Nombre d'étoile des 10 projets en ayant le plus
        sur GitHub",
       x = " ",
       y = "Nombre d'étoiles") +
  # améliorer l'échelle affichée pour les ordonnées
  scale_y_continuous(breaks = seq(200000,400000,10000)) +
  # retire l'échelle en abscisse
  theme(axis.text.x = element_blank(),
        axis.ticks = element_blank())
```

## Nombre d'étoiles des 10 projets en ayant le plus sur GitHub



On remarque immédiatement qu'il y a plus de 100000 étoiles de différence entre le dépôt `freeCodeCamp` qui en possède 371122 et le dépôt `react` qui en possède 211912.

```
sprintf("Nombre d'étoiles de %s : %d", df[1,]$name, df[1,]$stars)
```

```
## [1] "Nombre d'étoiles de freeCodeCamp : 371122"
```

```
sprintf("Nombre d'étoiles de %s : %d", df[10,]$name, df[10,]$stars)
```

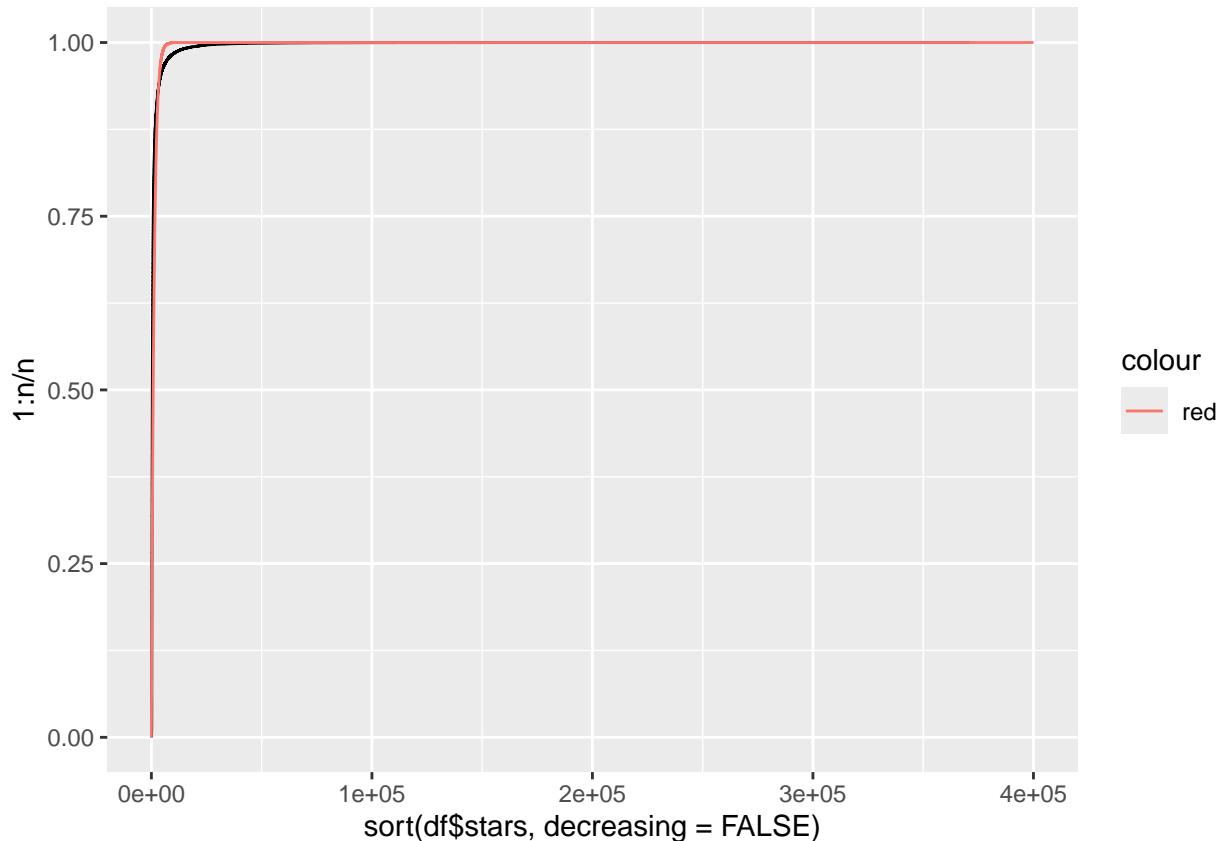
```
## [1] "Nombre d'étoiles de react : 211912"
```

L'augmentation du nombre d'étoile n'est absolument pas linéaire et grimper dans le classement de *popularité* est de plus en plus difficile à mesure que l'on progresse.

Si l'on considère le nombre d'étoile comme une variable aléatoire, on pourrait essayer d'estimer la loi de cette dernière.

En première approche on peut tester l'hypothèse selon laquelle cette variable aléatoire suit une loi exponentielle (le plus probable au vu des histogrammes tracés précédemment).

```
# crée une séquence de 1 à 400000
s <- seq(0,400000,1)
# mesure
lambda_mm <- 1/mean(df$stars)
ggplot() +
  # fonction de répartition mesurée
  geom_line(aes(x = sort(df$stars, decreasing = FALSE), y = 1:n/n)) +
  # loi exponentielle avec un paramètre estimé par méthode des moments
  # pour ce jeu de données
  geom_line(aes(x = s, y = pexp(s, rate = lambda_mm), color = 'red'))
```



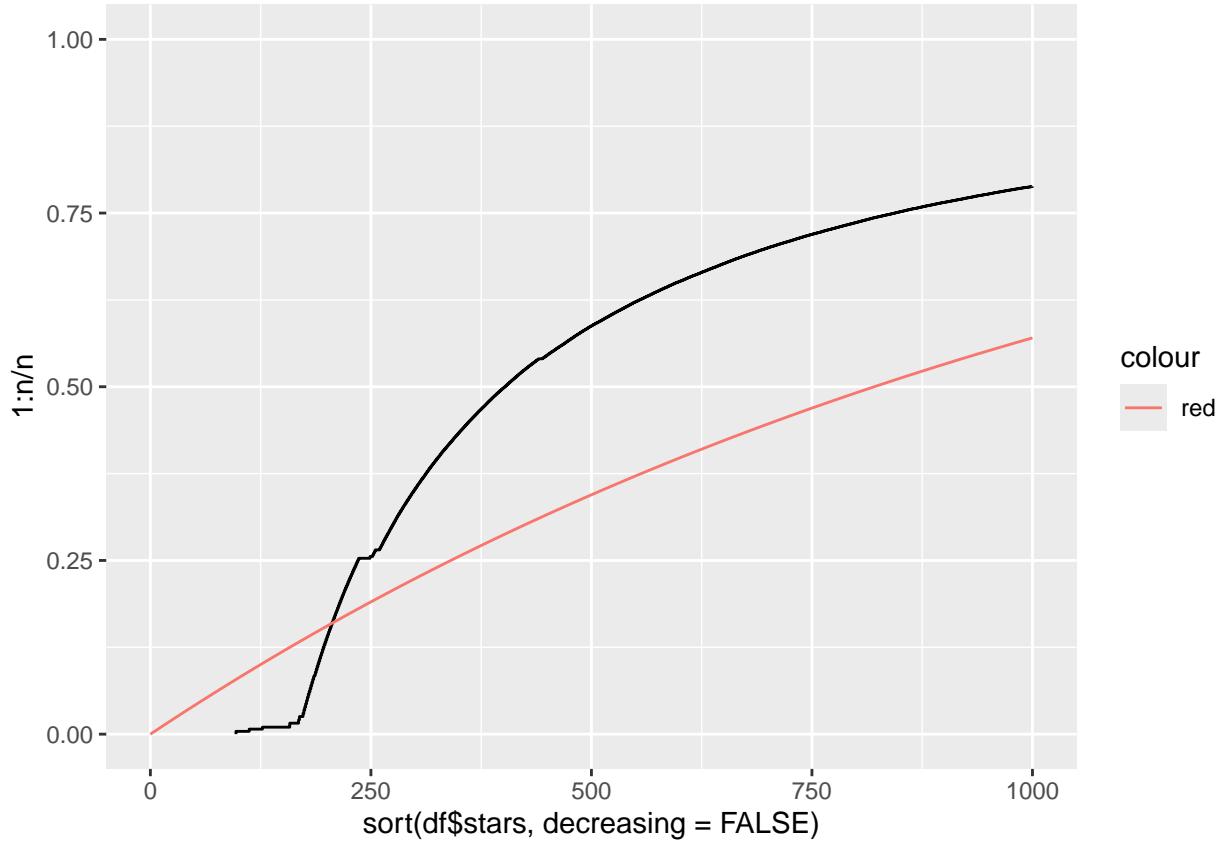
```

ggplot() +
  # fonction de répartition mesurée
  geom_line(aes(x = sort(df$stars, decreasing = FALSE), y = 1:n/n)) +
  # loi exponentielle avec un paramètre estimé par méthode des moments
  # pour ce jeu de données
  geom_line(aes(x = s, y = pexp(s, rate = lambda_mm), color = 'red')) +
  xlim(0,1000)

## Warning: Removed 41124 rows containing missing values or values outside the scale range
## (`geom_line()`).

## Warning: Removed 399000 rows containing missing values or values outside the scale range
## (`geom_line()`).

```



explication relative à l'estimation de paramètre de lois par [méthode des moments](#)

Graphiquement l'approximation par une loi exponentielle n'est pas très bonne. Il faudrait pousser l'analyse statistique plus loin pour conclure, ce qui n'est pas l'objet de cette analyse.

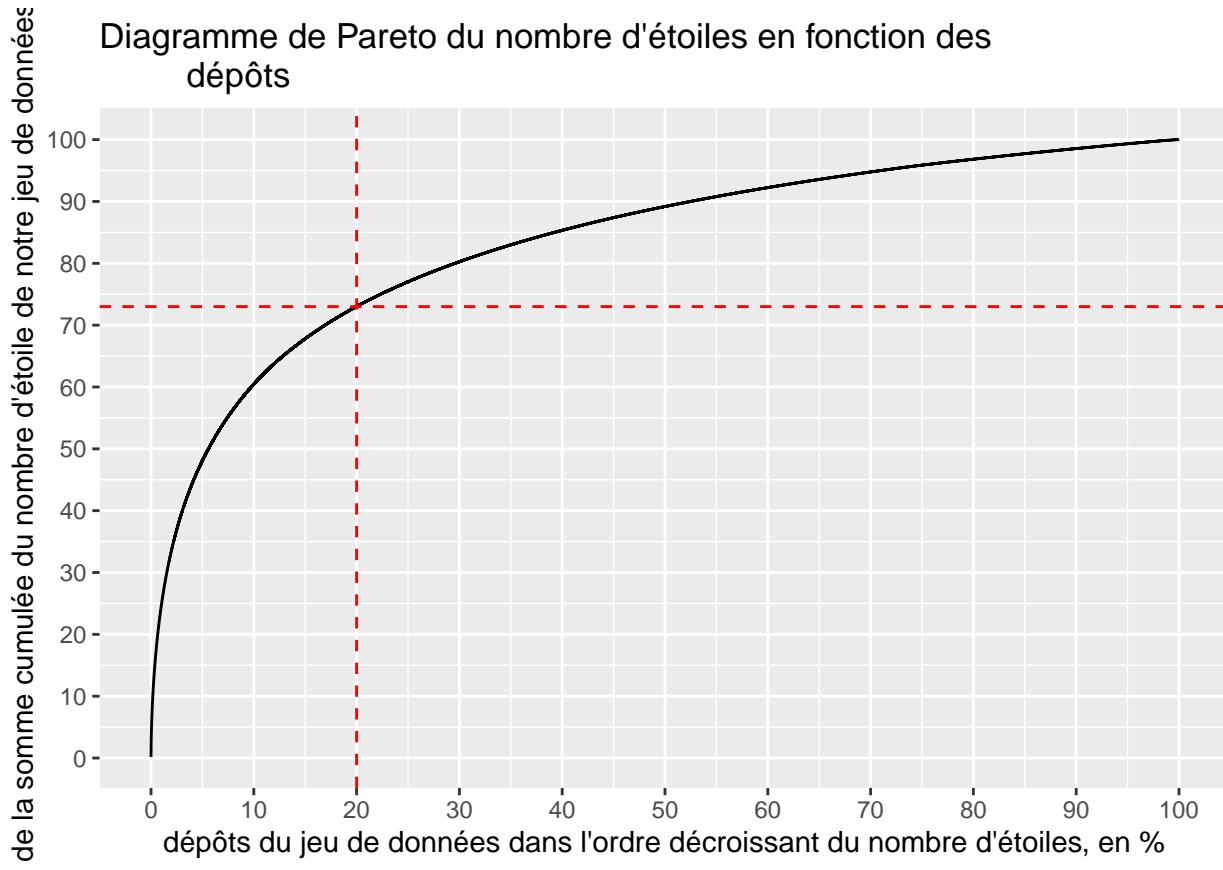
Dans l'introduction de ce rapport, nous avions émis l'hypothèse selon laquelle le nombre d'étoile des dépôt se répartissait selon le principe de [Pareto](#).

Selon ce principe, 20% des causes produisent 80% des effets. Dans notre cas, cela signifierait que 20% des dépôts de notre jeu de données représenteraient 80% du nombre total d'étoiles.

On peut construire un diagramme de Pareto :

```
p <- ggplot() +
  geom_line(aes(x = (1:n)*100/n, y = cumsum(df$stars)*100/sum(df$stars))) +
  labs(title = "Diagramme de Pareto du nombre d'étoiles en fonction des
  dépôts",
       x = "dépôts du jeu de données dans l'ordre décroissant du nombre d'étoiles, en %",
       y = "% de la somme cumulée du nombre d'étoiles de notre jeu de données") +
  scale_y_continuous(breaks = seq(0,100,10)) +
  scale_x_continuous(breaks = seq(0,100,10))

p +
  geom_vline(xintercept = 20,
             linetype = "dashed",
             color = "red") +
  geom_hline(yintercept = 73,
             linetype = "dashed",
             color = "red")
```



Ce diagramme peut se lire ainsi : “20% des dépôts (lecture sur l’axe des abscisses) représentent environ 73% (lecture sur l’axe des ordonnées) du nombre total d’étoiles de notre jeu de données”.

Comme sur ce graphique les dépôts sont classés par ordre décroissant d’étoiles, on peut dire que les 20% des dépôts ayant le plus d’étoiles ont à eux seul environ 73% des étoiles totales attribuées sur GitHub (avec le biais de sélection dont on a parlé plus haut).

Le principe de Pareto s’applique donc dans une certaine mesure ici.

Si l’on considère le nombre d’étoile d’un dépôt comme un marqueur de sa “popularité”, on peut ensuite se demander comment expliquer cette popularité.

On dispose de plusieurs informations concernant les dépôts qui peuvent nous renseigner :

- les langages utilisés
- les *topics* (les tags associés au dépôt)
- la date de création

**4.3 La date création d'un dépôt influence-t-elle sa popularité ?** On peut par exemple penser que plus un dépôt est ancien, plus son nombre d’étoile est important. C'est un raisonnement plutôt naturel : ce qui est plus ancien a eu le temps de se faire connaître et donc de gagner en popularité.

Notre dataset dispose d'un attribut `createdAt`.

```
str(df$createdAt)
```

```
## POSIXct[1:194196], format: "2014-12-24 17:49:19" "2013-10-11 06:50:37" "2019-03-26 07:31:14" ...
```

Les informations sont stockées dans un format `POSIXct` qui l'un des deux formats utilisés pour stocker des dates.

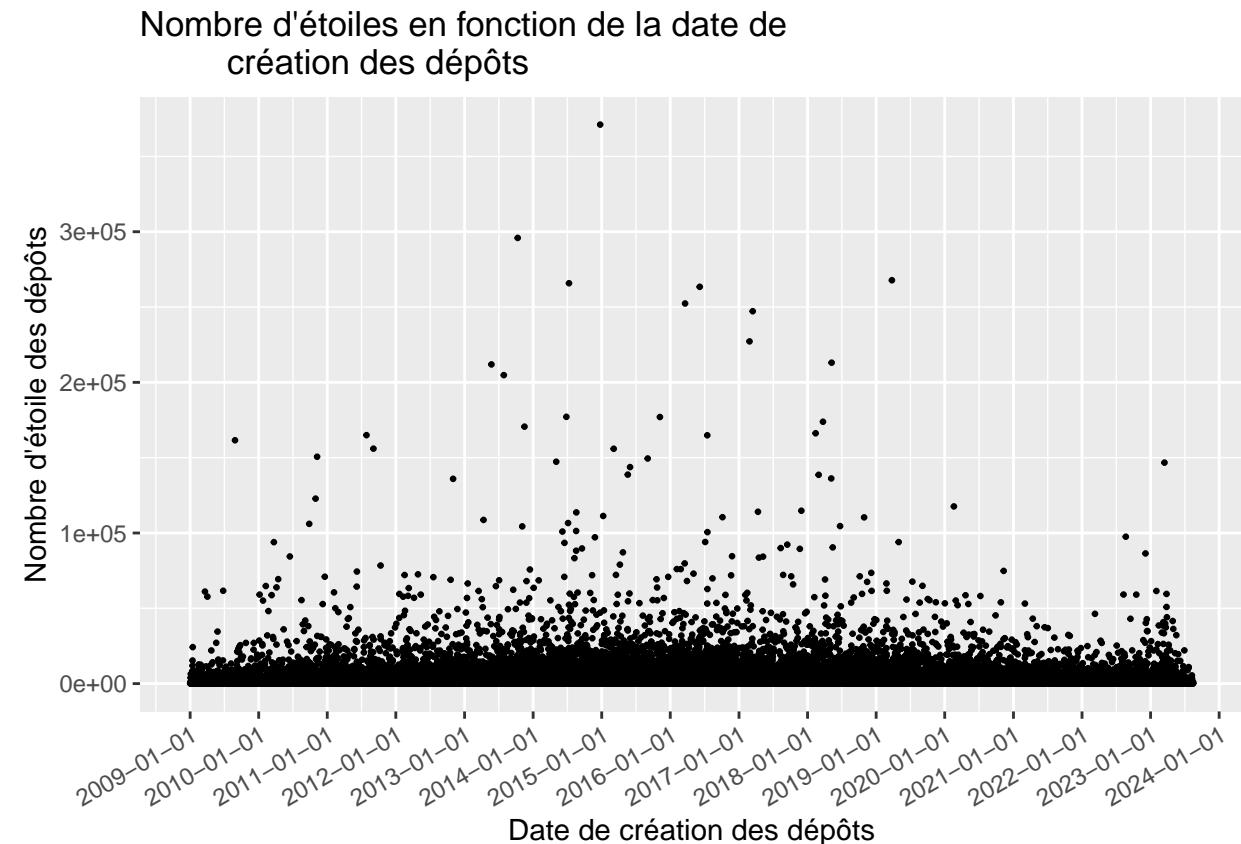
Nous n'avons pas besoin de conserver une précision sur l'heure de création des dépôts, on peut donc commencer par ne regarder que la date :

```
# On crée une nouvelle colonne au dataset en convertissant la date dans
# un format plus simple : aaaa-mm-jj
df <- df %>% mutate(creationDate = as.Date(createdAt))
```

On va donc utiliser un `scatterplot` pour observer une possible corrélation entre la date de création et le nombre d'étoiles d'un dépôt.

```
# on crée les graduations en abscisse pour faciliter la lecture
datebreaks <- seq(as.Date("2009-01-01"), as.Date("2024-01-01"), by = "1 year")

ggplot(df, aes(x = creationDate, y = stars)) +
  geom_point(size = 0.5) +
  scale_x_date(breaks = datebreaks) +
  # pour afficher les dates en biais
  theme(axis.text.x = element_text(angle = 30, hjust = 1)) +
  labs(title = "Nombre d'étoiles en fonction de la date de
        création des dépôts",
      x = "Date de création des dépôts",
      y = "Nombre d'étoile des dépôts")
```



Sur ce graphique, les points qui se détachent de la base de points noirs (coordonnée 0e + 00 sur l'axe des ordonnées) sont ceux qui ont atteint un nombre important d'étoiles.

On retrouve notre dépôt `freeCodeCamp` qui possède le plus d'étoile en haut au centre :

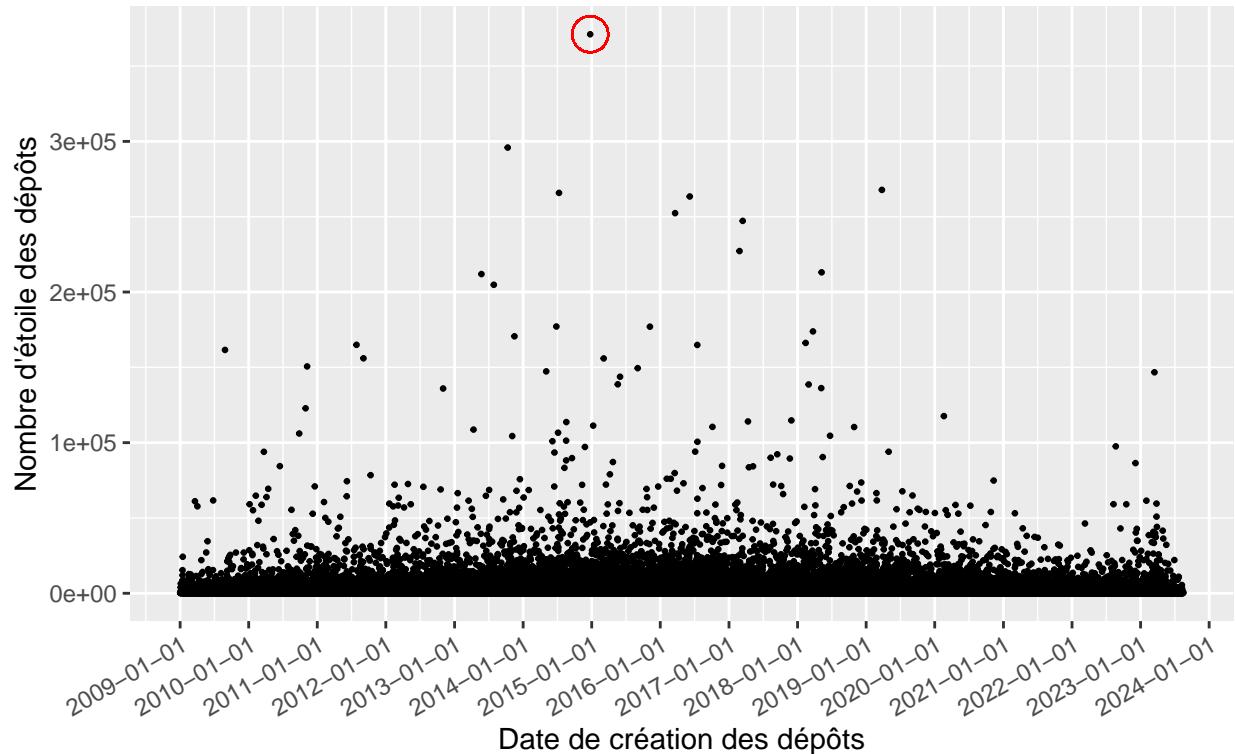
```

ggplot(df, aes(x = creationDate, y = stars)) +
  geom_point(size = 0.5) +
  geom_point(aes(x = df[df$name == 'freeCodeCamp', ]$creationDate,
                 y = df[df$name == 'freeCodeCamp', ]$stars),
             shape = 1,
             size = 6,
             color = 'red') +
  scale_x_date(breaks = datebreaks) +
  # pour afficher les dates en biais
  theme(axis.text.x = element_text(angle = 30, hjust = 1)) +
  labs(title = "Nombre d'étoiles en fonction de la date de
        création des dépôts",
       x = "Date de création des dépôts",
       y = "Nombre d'étoile des dépôts")

```

## Warning in geom\_point(aes(x = df[df\$name == "freeCodeCamp", ]\$creationDate, : All aesthetics have le  
## i Did you mean to use `annotate()``?

**Nombre d'étoiles en fonction de la date de  
création des dépôts**



Attention, encore une fois à l'échelle utilisée dans ce graphique : comme nous avons une très forte disparité entre le nombre d'étoile des dépôts, nous affichons en même temps des dépôts avec un nombre d'étoile dans les centaine ( $10^2$ ) et des dépôts avec des centaines de milliers d'étoiles ( $10^5$ ).

Ce graphique nous montre très clairement que la date de création du dépôt n'influence en rien la popularité de ces derniers : on trouve des dépôts aujourd'hui populaire créés dans les premières années de GitHub (2009-2010), comme des dépôt populaire très récents (2023).

Si l'on utilise une échelle logarithmique pour le nombre d'étoile on s'en rend mieux compte que les dépôts

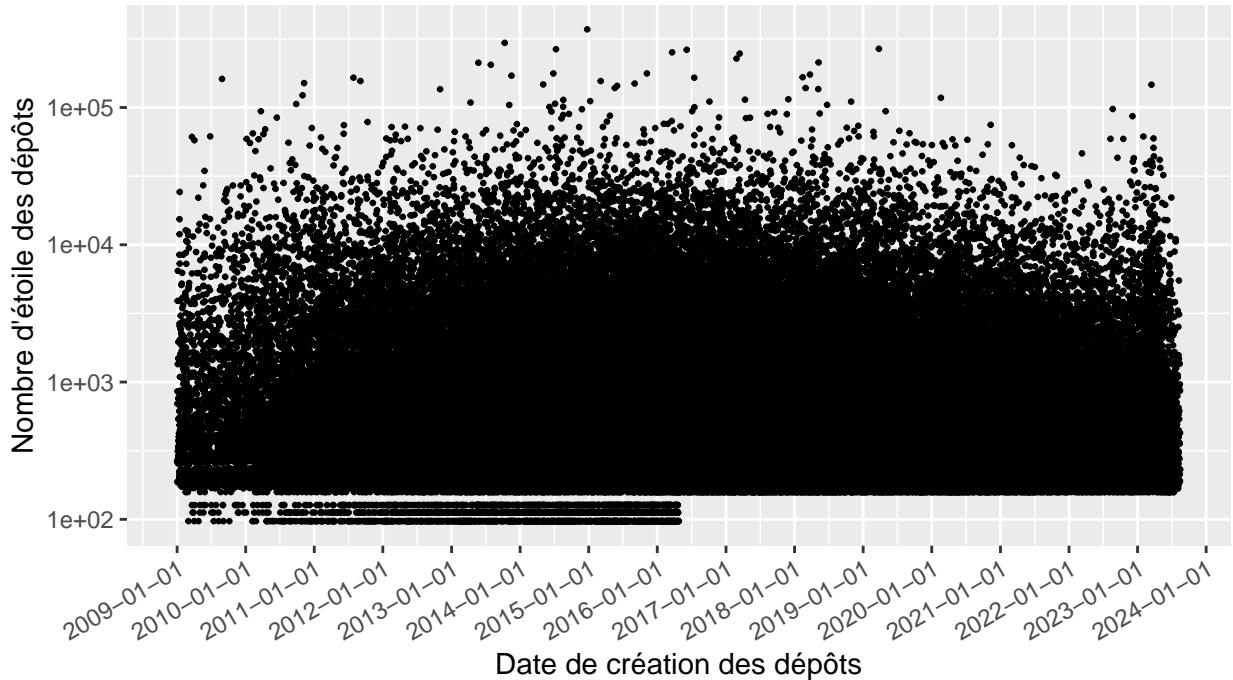
“populaires

```
# on crée les graduations en abscisse pour faciliter la lecture
datebreaks <- seq(as.Date("2009-01-01"), as.Date("2024-01-01"), by = "1 year")

ggplot(df, aes(x = creationDate, y = stars)) +
  geom_point(size = 0.5) +
  scale_y_log10() +
  scale_x_date(breaks = datebreaks) +
  # pour afficher les dates en biais
  theme(axis.text.x = element_text(angle = 30, hjust = 1)) +
  labs(title = "Nombre d'étoiles en fonction de la date de
        création des dépôts",
       x = "Date de création des dépôts",
       y = "Nombre d'étoile des dépôts",
       subtitle = "Le nombre d'étoile est représenté par une échelle
                  logarithmique")
```

## Nombre d'étoiles en fonction de la date de création des dépôts

Le nombre d'étoile est représenté par une échelle  
logarithmique



On peut se convaincre en affichant la moyenne des étoiles des dépôts pour chaque année création (la moyenne pour les dépôts créés entre 2009 et 2010, pour ceux créés entre 2010 et 2011, etc.).

```
# On calcul la moyenne du nombre d'étoiles des dépôt en fonction de leurs
# # date de création en divisant en années
separation_annees <- paste(2009:2023, "-01-01", sep="")
separation_annees[length(separation_annees)+1] <- as.character(max(df$creationDate))
separation_annees[length(separation_annees)] <- as.Date(separation_annees[length(separation_annees)])
# nbr = 0
```

```

mean_stars_by_year <- c()
for (i in 2:length(separation_annees)) {
  # nbr = nbr + dim(df[df$creationDate < separation_annees[i] &
  #                     df$creationDate >= separation_annees[i-1],])[1]
  # Pour vérifier le nombre d'éléments comptés

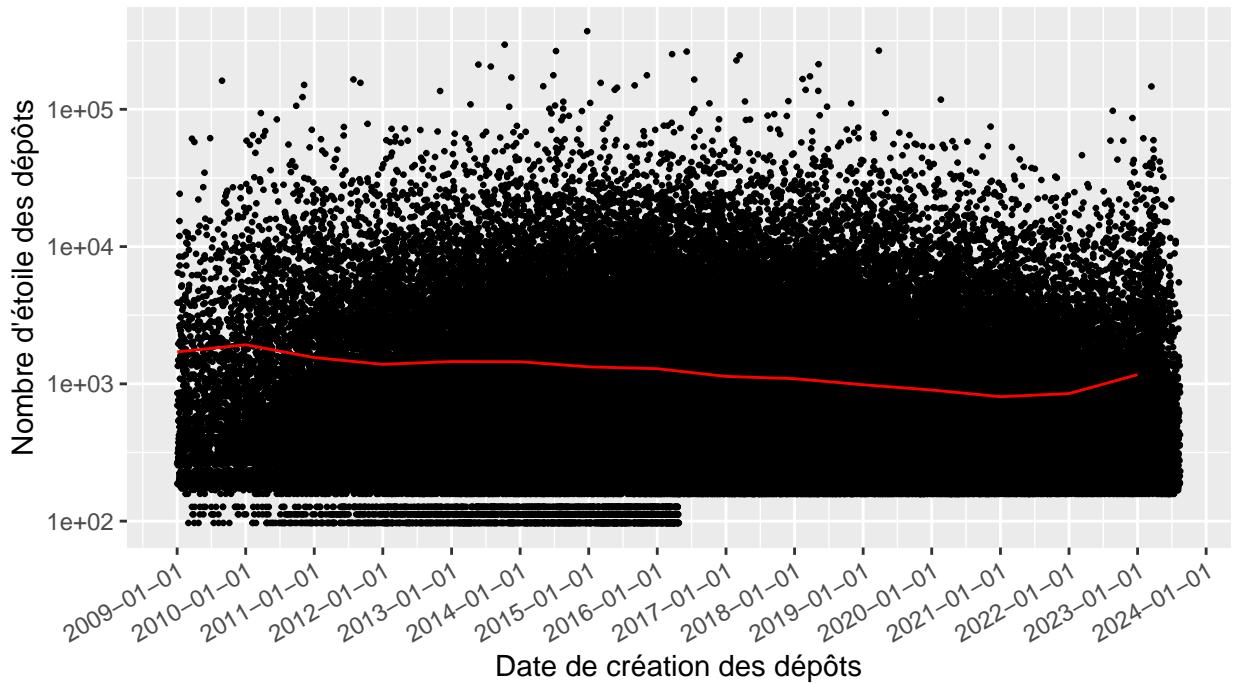
  mean_stars_by_year <- c(
    mean_stars_by_year,
    mean(df[df$creationDate < separation_annees[i] &
            df$creationDate >= separation_annees[i-1],]$stars)
  )
}

ggplot() +
  geom_point(aes(x = df$creationDate, y = df$stars), size = 0.5) +
  geom_line(aes(x = as.Date(separation_annees[1:(length(separation_annees)-1)]),
                y = mean_stars_by_year), color = "red") +
  scale_y_log10() +
  scale_x_date(breaks = datebreaks) +
  # pour afficher les dates en biais
  theme(axis.text.x = element_text(angle = 30, hjust = 1)) +
  labs(title = "Nombre d'étoiles en fonction de la date de
        création des dépôts",
       x = "Date de création des dépôts",
       y = "Nombre d'étoile des dépôts",
       subtitle = "Le nombre d'étoile est représenté par une échelle
                  logarithmique")

```

## Nombre d'étoiles en fonction de la date de création des dépôts

Le nombre d'étoile est représenté par une échelle logarithmique



On voit bien que la moyenne reste relativement stable. On note une légère hausse en 2022 qui peut traduire une hausse de popularité de la plateforme.

Le 25 janvier 2023, GitHub annonçait avoir atteint les [100 millions d'utilisateurs](#).

On peut donc voir dans notre graphique un début d'une augmentation d'utilisateurs actifs sur la plateforme, sans pouvoir le confirmer.

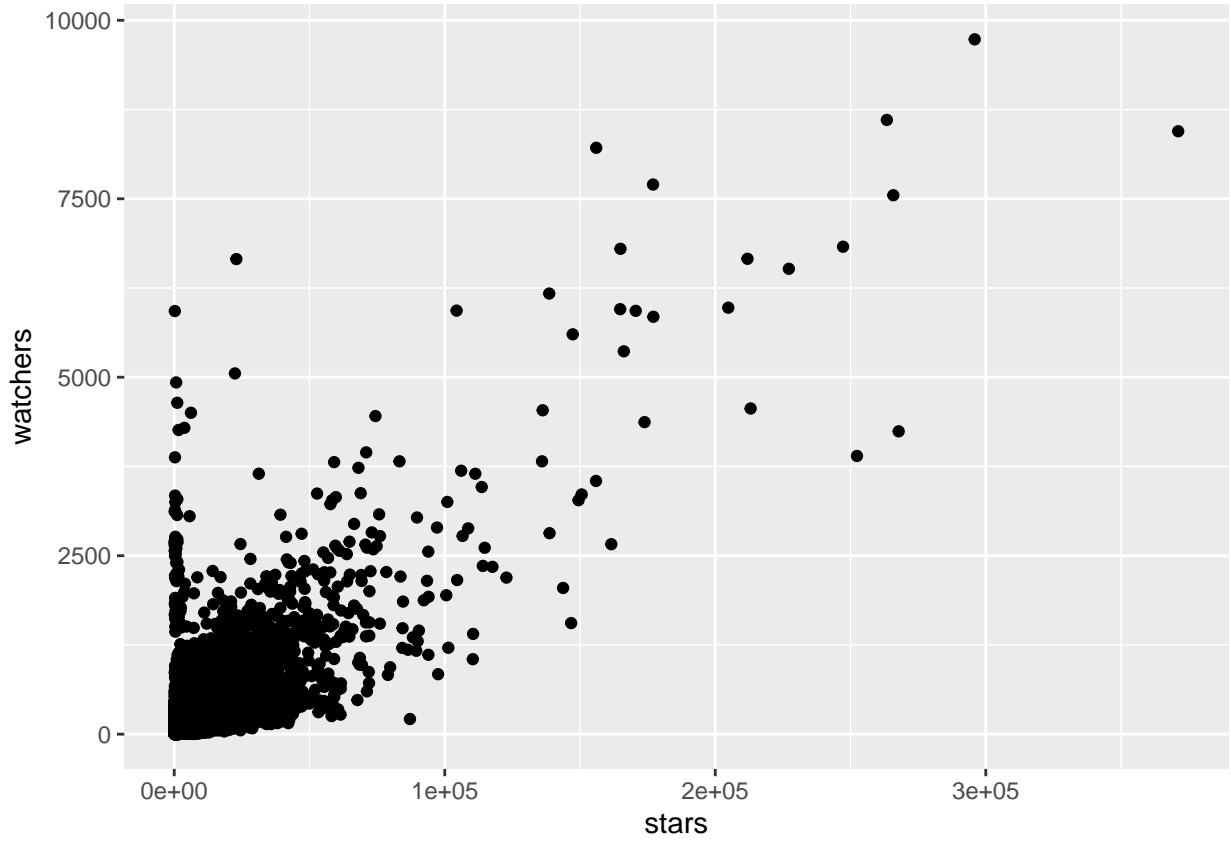
**4.4 Existe-t-il un lien entre le nombre d'étoile et le nombre de de watchers ?** Comme évoqué dans la présentation du jeu de données, nous disposons d'autres indicateurs de “popularité”. L'un deux, nommé **watchers** dans le dataset, indique combien de personne “suivent” un dépôt.

Si l'on reprend la métaphore d'un réseau social : un utilisateur peu *liker* un dépôt qu'il apprécie (ajoute une étoile *stars*) mais il peut également *follow* le dépôt pour être mis au courant des évolutions.

On peut supposer qu'un dépôt ayant beaucoup d'étoiles aura aussi beaucoup de *watchers* qui souhaitent être informé des évolutions.

On va donc confirmer ou non cette hypothèse à l'aide d'un [scatterplot](#).

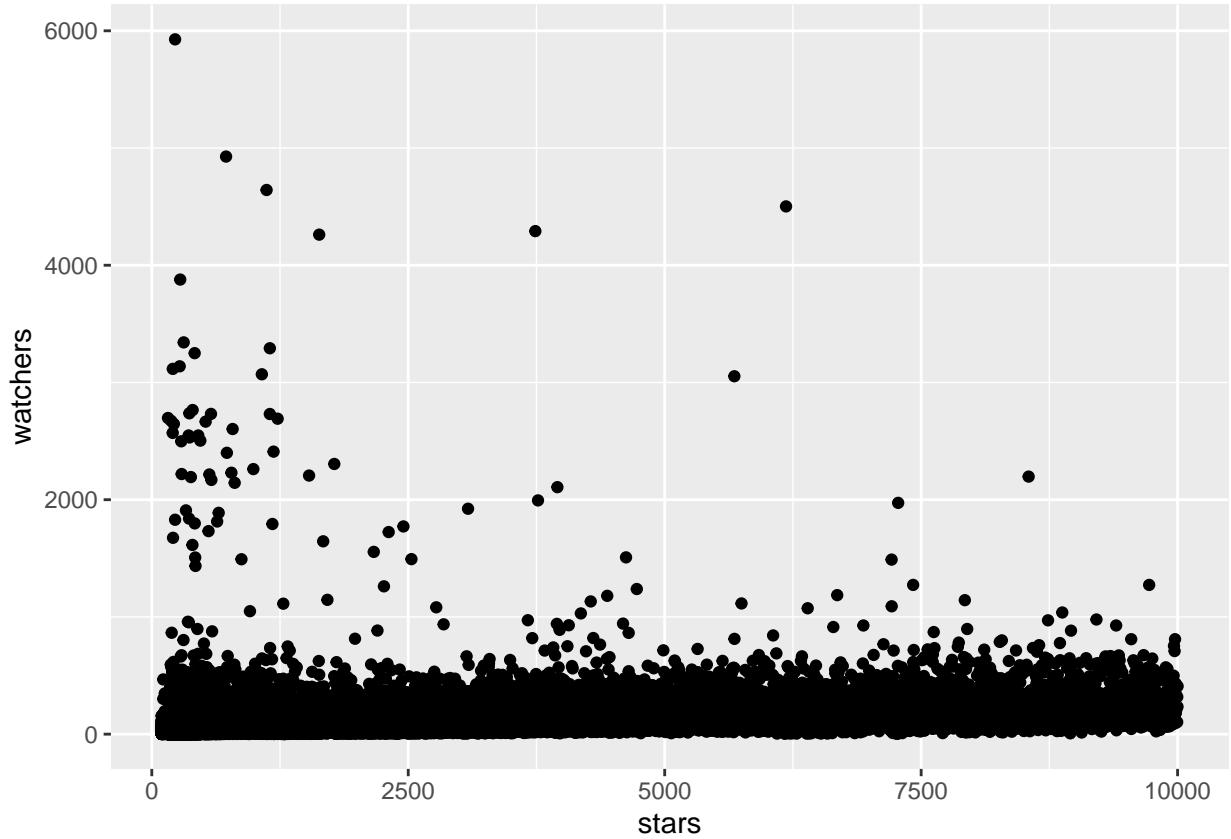
```
scplt_stars_watchers <- ggplot(df, aes(x = stars, y = watchers)) +  
  geom_point()  
scplt_stars_watchers
```



Tel quel le graphique n'est pas exploitable à cause de la très grande disparité dans les valeurs de `stars` qui vient "tasser" la très grande majorité des points dans le coin inférieur gauche ("faibles" valeurs de `stars` et faibles valeurs de `watchers`)

On peut commencer par tracer le même graphique en retirant tous les dépôts possédant moins de 10000 étoiles.

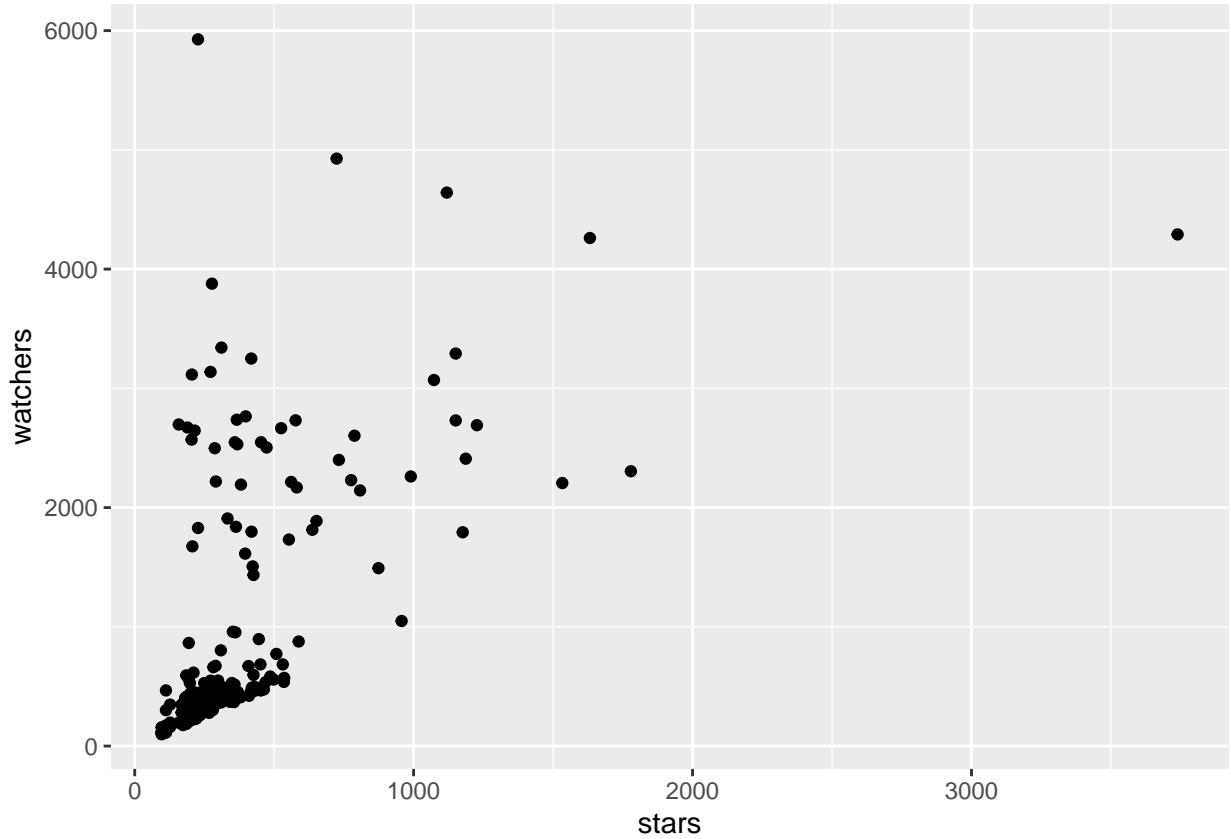
```
scplt_stars_watchers2 <- ggplot(df[df$stars <= 10000,],
                                    aes(x = stars, y = watchers)) +
  geom_point()
scplt_stars_watchers2
```



Contrairement à notre hypothèse de base, il semble que le nombre de `watchers` n'évolue que très peu quelque soit le nombre d'étoile attribué au dépôt.

On observe quelque dépôts qui sortent du lot (valeurs abérantes) qui possèdent plus de `watchers` que de `stars` (ce qui va à l'encontre de la très large majorité des dépôts).

```
scplt_stars_watchers3 <- ggplot(df[df$stars <= df$watchers,],
                                    aes(x = stars, y = watchers)) +
  geom_point()
scplt_stars_watchers3
```



Si l'on observe les noms de ces dépôts :

```
# Liste complète
# df[df$stars <= df$watchers,]$nameWithOwner
head(df[df$stars <= df$watchers,]$nameWithOwner)

## [1] "Azure/azure-powershell"
## [2] "microsoft/code-with-engineering-playbook"
## [3] "aspnet/Announcements"
## [4] "uber/okbuck"
## [5] "Azure/azureml-examples"
## [6] "Azure/azure-sdk-for-node"
```

On remarque qu'il s'agit surtout de dépôt publics créés par des grandes entreprises comme Microsoft (les produits Azure entre autre), Uber, Netflix, Shopify, etc.

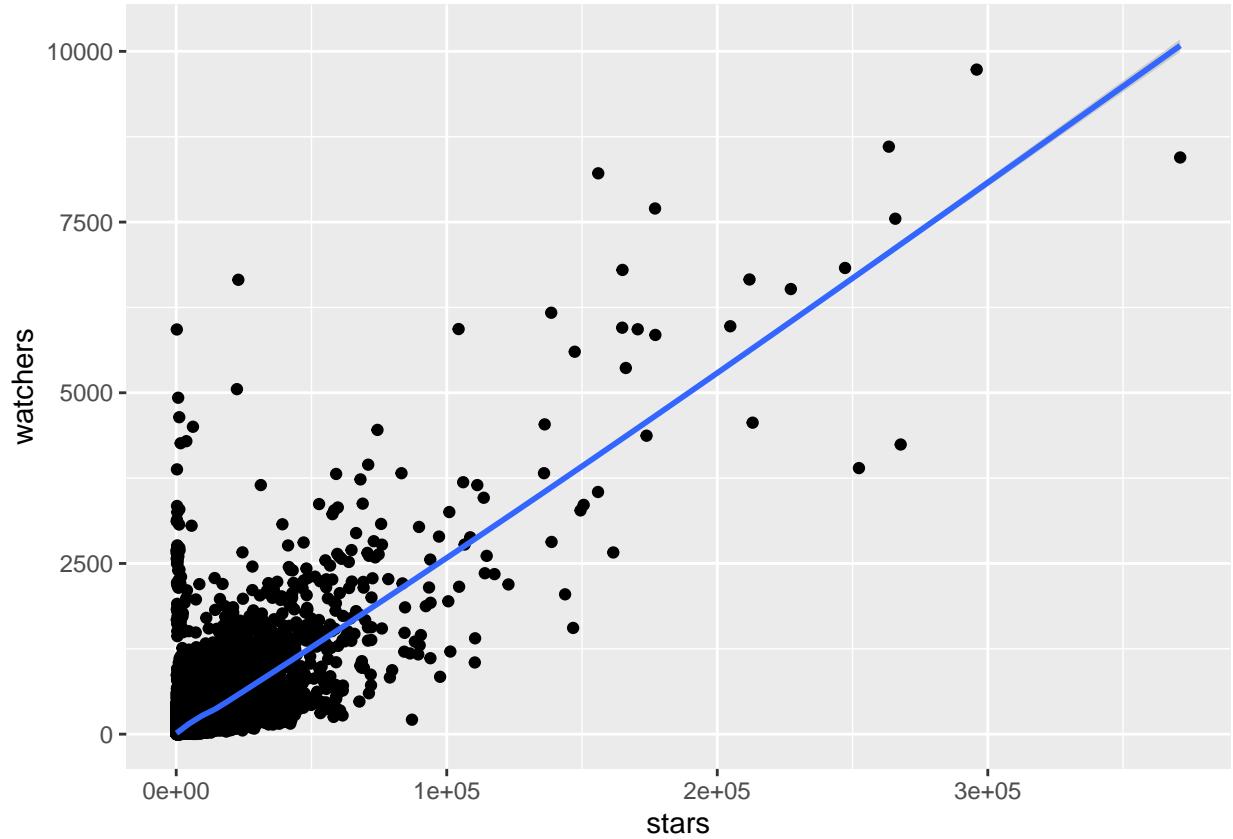
On peut donc supposer que ces dépôt abritent le code source de programmes ré-utilisés par de nombreux développeurs (tel que des API ou des SDK : software development kit).

Les entreprises ou développeurs utilisant ces programmes doivent donc souhaiter être tenu à jour des évolutions (ce sont des **watchers**) sans pour autant *liker* le dépôt (sans prendre le temps de le faire).

On peut donc supposer qu'il existe une relation linéaire entre les deux variables du type :  $watchers = a \times stars + b$ . Avec  $a$  proche de 0.

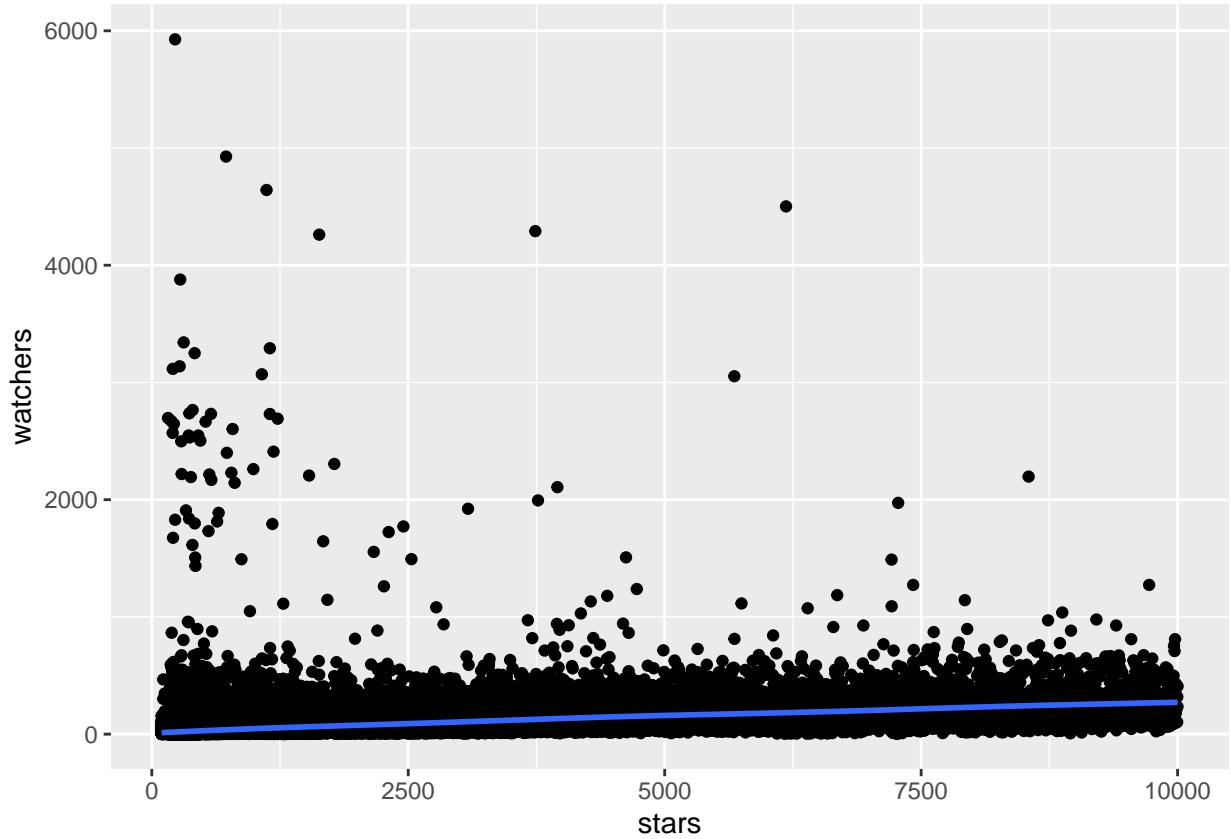
```
scplt_stars_watchers + geom_smooth()

## `geom_smooth()` using method = 'gam' and formula = 'y ~ s(x, bs = "cs")'
```



```
scplt_stars_watchers2 + geom_smooth()
```

```
## `geom_smooth()` using method = 'gam' and formula = 'y ~ s(x, bs = "cs")'
```



```
lm(df$watchers~df$stars)$coefficients
```

```
## (Intercept)      df$stars
## 16.10759764  0.02587358
```

Si on ajuste une droite de régression linéaire on obtient une droite du type :  $watchers = 0.02587 \times stars + 16.11$ .

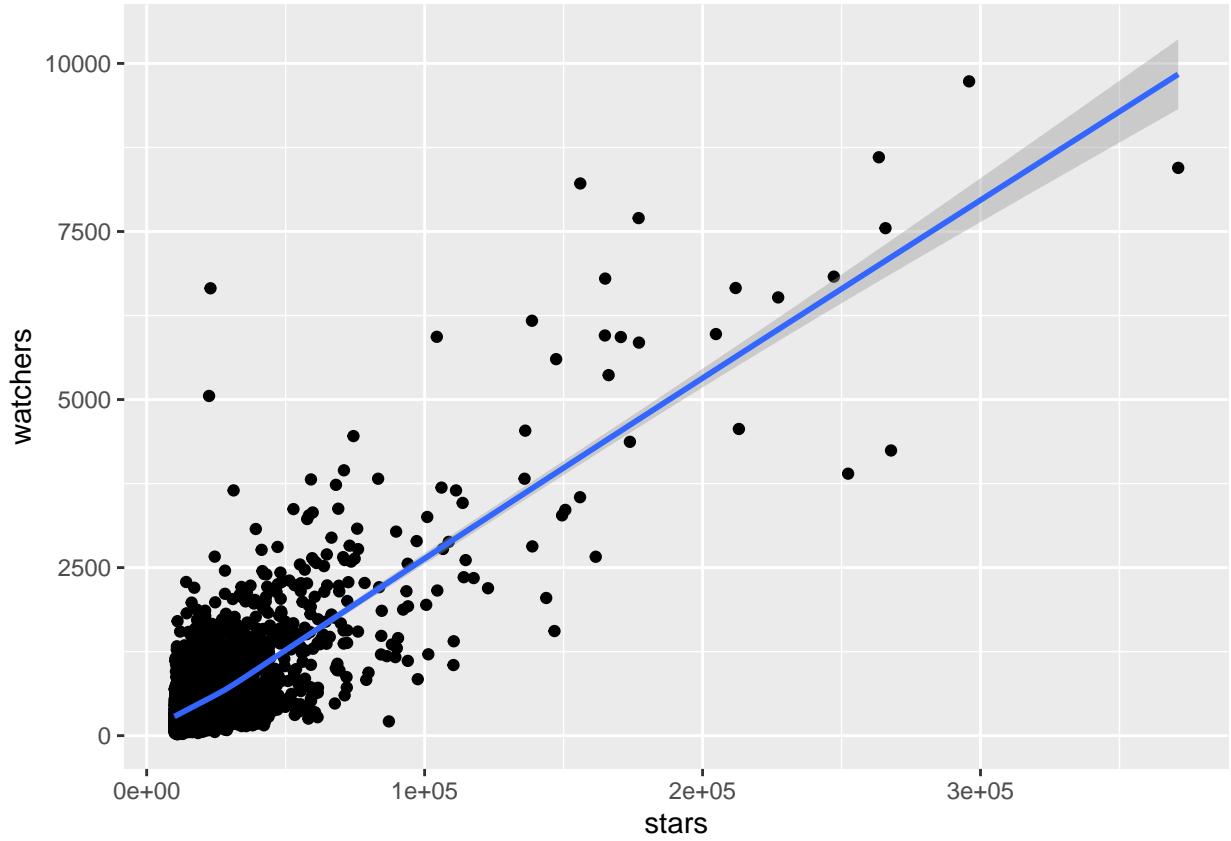
Cette hypothèse de linéarité entre les deux variables doit cependant être traité avec prudence au vu du nuage. On peut observer une tendance générale : plus un dépôt à d'étoile, plus il a tendance à avoir été suivi et donc avoir plus de **watchers**. Certains dépôts font exception à cette règle.

On pourrait pousser l'analyse statistique pour définir la qualité d'adéquation d'un modèle linéaire.

Si l'on affiche que les dépôts ayant plus de 10000 étoiles on a :

```
scplt_stars_watchers3 <- ggplot(df[df$stars > 10000,],
                                    aes(x = stars, y = watchers)) +
  geom_point()
scplt_stars_watchers3 + geom_smooth()

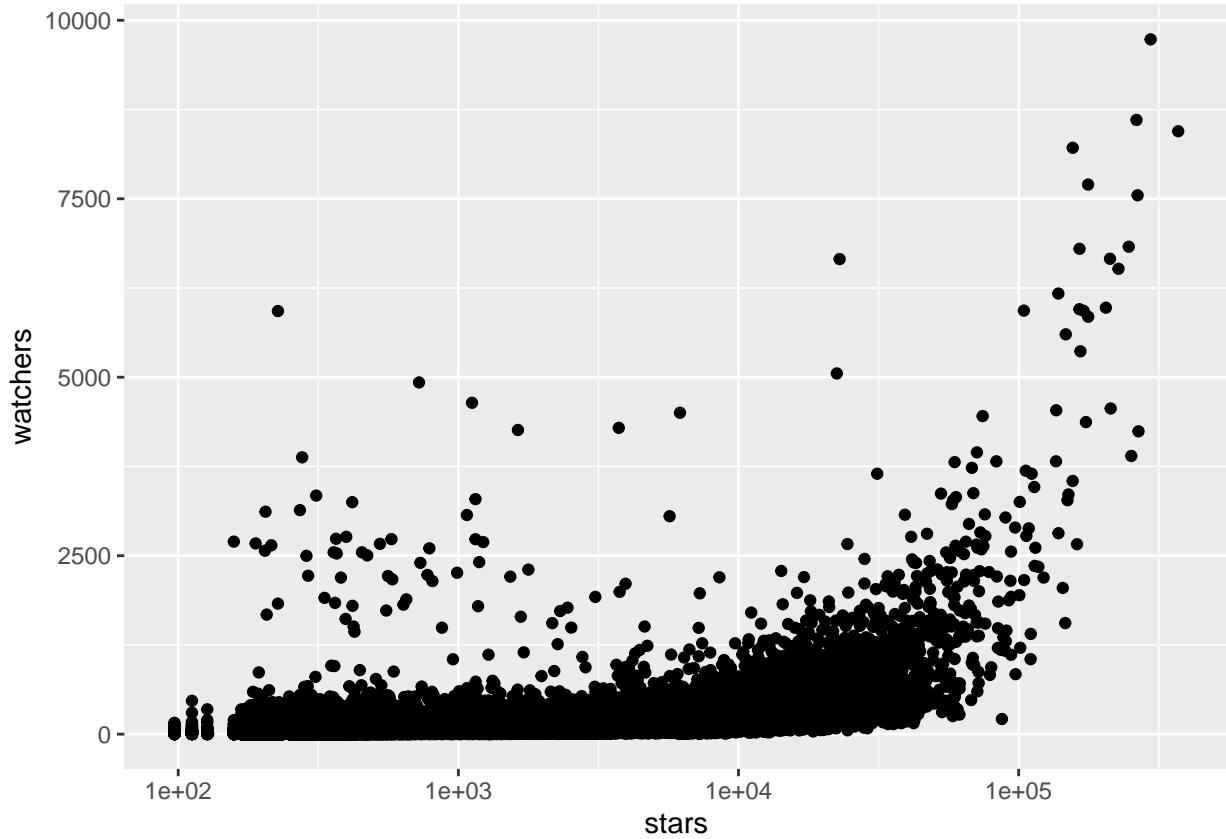
## `geom_smooth()` using method = 'gam' and formula = 'y ~ s(x, bs = "cs")'
```



On observe un comportement très similaire (certaines valeurs extrêmes vont venir *tasser* les plus petites dans le coin inférieur gauche) et on retrouve une relation linéaire.

On peut observer le nuage de point complet en utilisant une échelle logarithmique.

```
scplt_stars_watchers + scale_x_log10()
```



Attention dans ce cas : le graphique affiche une tendance non linéaire (exponentielle) pour les grandes valeurs de `stars`. Cependant cet effet est principalement dû à l'échelle logarithmique.

La forme prise par le nuage de point est indique bel et bien une relation linéaire quand seul l'axe des abscisses est en échelle logarithmique.

On peut donc en conclure qu'il existe bien une relation linéaire entre les deux grandeurs, même si on observe une grande variabilité dans les données avec quelques exceptions dont le comportement diffère.

**4.5 Existe-t-il un relation entre le nombre d'étoile et le nombre de forks ?** Pour cet jeu de données, un autre attribut important est celui des `forks`, qui reflète dans une certaine mesure le degré de participation des utilisateurs dans un dépôt GitHub. Autrement dit, nous voulons comprendre la corrélation entre la popularité d'un projet et la participation.

Il convient de mentionner qu'en ce qui concerne le degré de participation, il peut y avoir des personnes qui participent directement à la contribution au code, ou il peut y avoir des personnes qui se contentent de *fork* un dépôt dans le but d'apprendre.

Pour deux attributs dans un jeu de données, calculer leur coefficient de corrélation est un bon moyen de trouver la relation entre eux. Ici, nous choisissons le coefficient de corrélation de Pearson et le coefficient de corrélation de rang de Spearman pour vérifier et découvrir la relation entre eux.

Les deux coefficients de corrélation prennent des valeurs comprises entre 1 et -1, 1 indiquant une corrélation élevée et 0 indiquant aucune corrélation. La différence est que le coefficient de corrélation de Pearson est plus sensible aux relations linéaires et que le coefficient de corrélation de rang de Spearman est plus sensible aux relations monotones entre les variables.

```
correlation_pearson <- cor(df$stars, df$forks, method = "pearson")
correlation_spearman <- cor(df$stars, df$forks, method = "spearman")
```

```

print(correlation_pearson)

## [1] 0.5847746

print(correlation_spearman)

## [1] 0.6531888

```

Concrètement, ces deux coefficients nous disent :

Coefficient de corrélation de Pearson : 0,5847746, indiquant qu'il existe un certain degré de corrélation linéaire entre **stars** et **forks**, mais la corrélation n'est pas très forte et est une corrélation modérée. Cela signifie que plus le nombre de **stars** d'un projet augmente, plus le nombre de **forks** augmente, et vice versa, mais l'ampleur du changement peut être relativement faible.

Coefficient de corrélation de rang de Spearman : 0,6531888, indiquant qu'il existe un certain degré de relation monotone entre **stars** et **forks**, mais il n'est pas nécessaire que la relation soit linéaire. Ce coefficient est légèrement supérieur au coefficient de corrélation de Pearson, ce qui indique que la relation entre **stars** et **forks** a tendance à être davantage une tendance monotone croissante, plutôt qu'une relation nécessairement linéaire stricte.

Considérant que le coefficient de corrélation de Pearson est très sensible aux valeurs aberrantes, nous essayons d'utiliser un **scatterplot** pour trouver ces valeurs aberrantes.

```

ggplot(df, aes(x = stars, y = forks)) +
  geom_point() +
  geom_smooth(method = "gam", se = FALSE) + #Puisque nous constatons qu'il n'existe pas nécessairement
  labs(x = "Stars", y = "Forks", title = "Relation entre Stars et Forks")

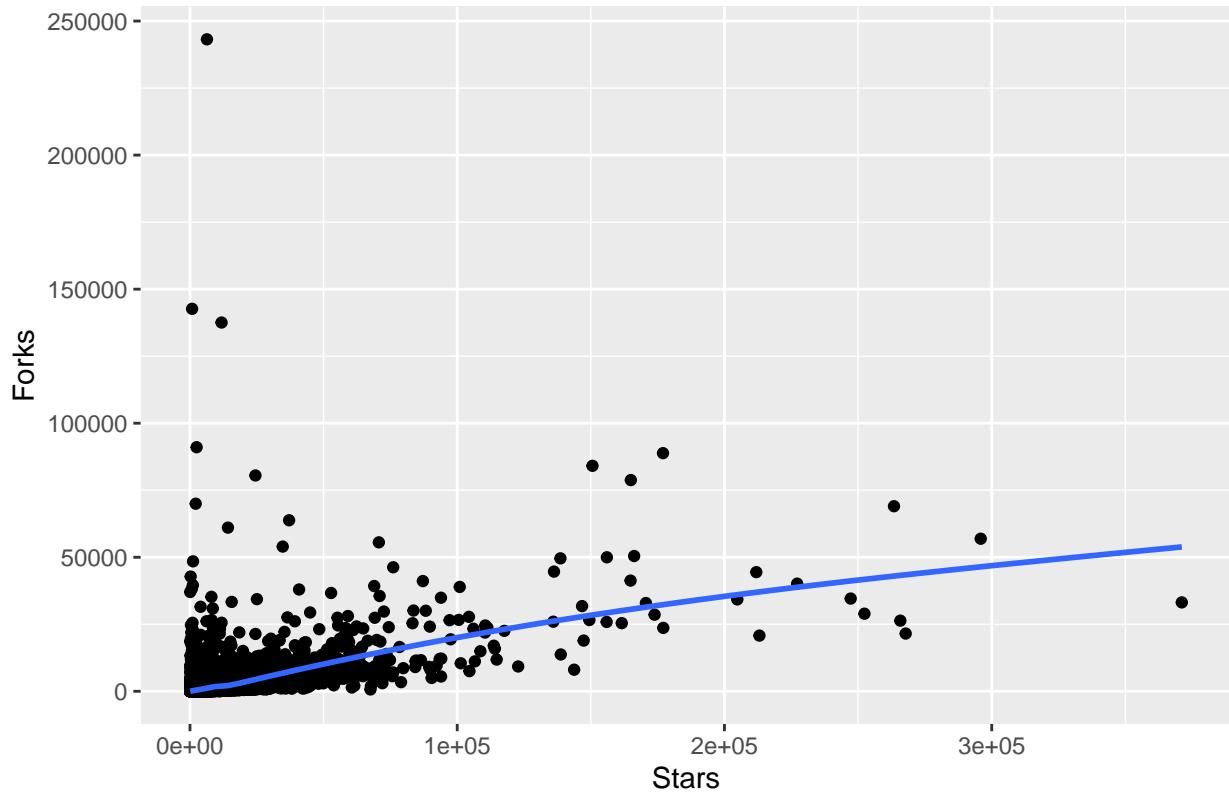
```

```

## `geom_smooth()` using formula = 'y ~ s(x, bs = "cs")'

```

## Relation entre Stars et Forks



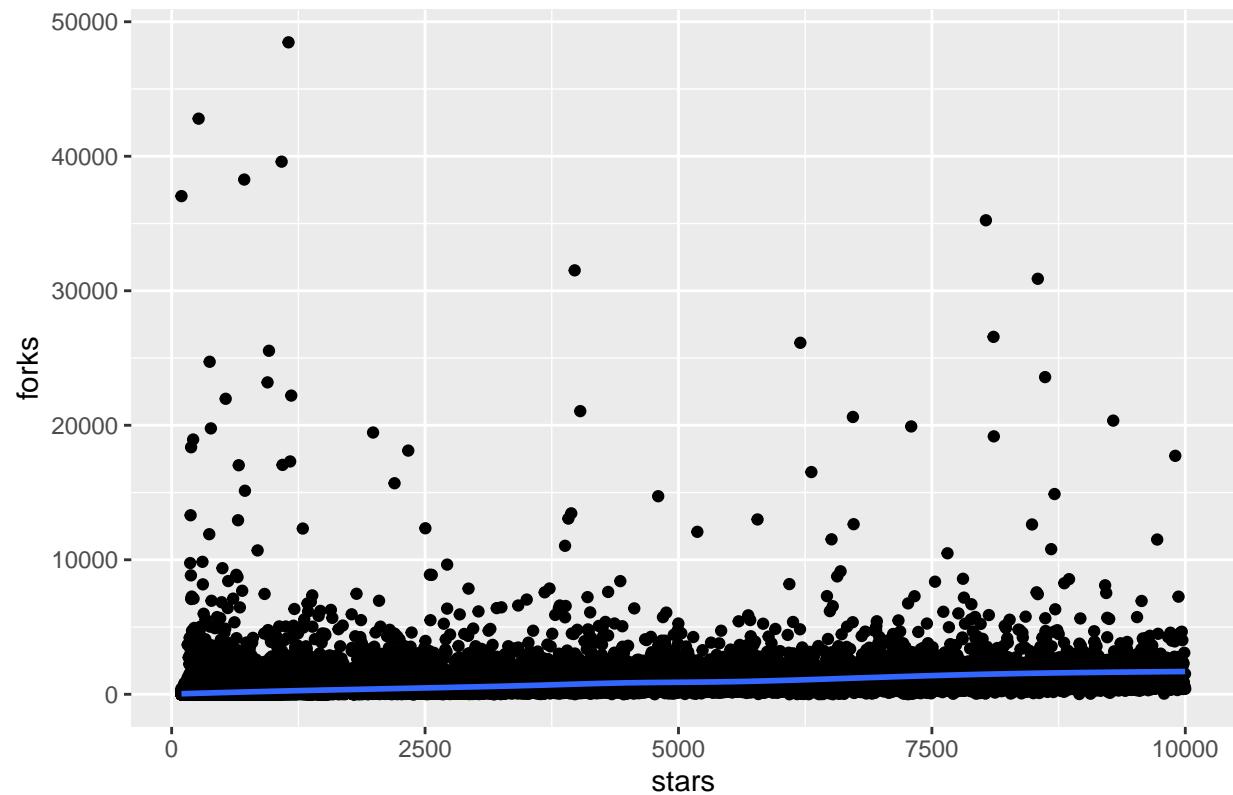
Nous avons constaté que certains dépôts ont un nombre de **stars** extrêmement faible, mais un nombre de **forks** extrêmement élevé. Il s'agit évidemment de valeurs aberrantes.

Nous avons également remarqué que le nombre de **stars** de la plupart des dépôts est inférieur à 10 000 et le nombre de **forks** est inférieur à 50 000. Afin d'étudier la relation entre le nombre d'étoiles et la fourchette, nous devons extraire la majorité des données.

```
ggplot(df[df$stars < 10000 & df$forks < 50000, ], aes(x = stars, y = forks)) +  
  geom_point() +  
  geom_smooth(method = "gam", se = FALSE) +  
  labs(title = "Relation entre Stars et Forks")
```

```
## `geom_smooth()` using formula = 'y ~ s(x, bs = "cs")'
```

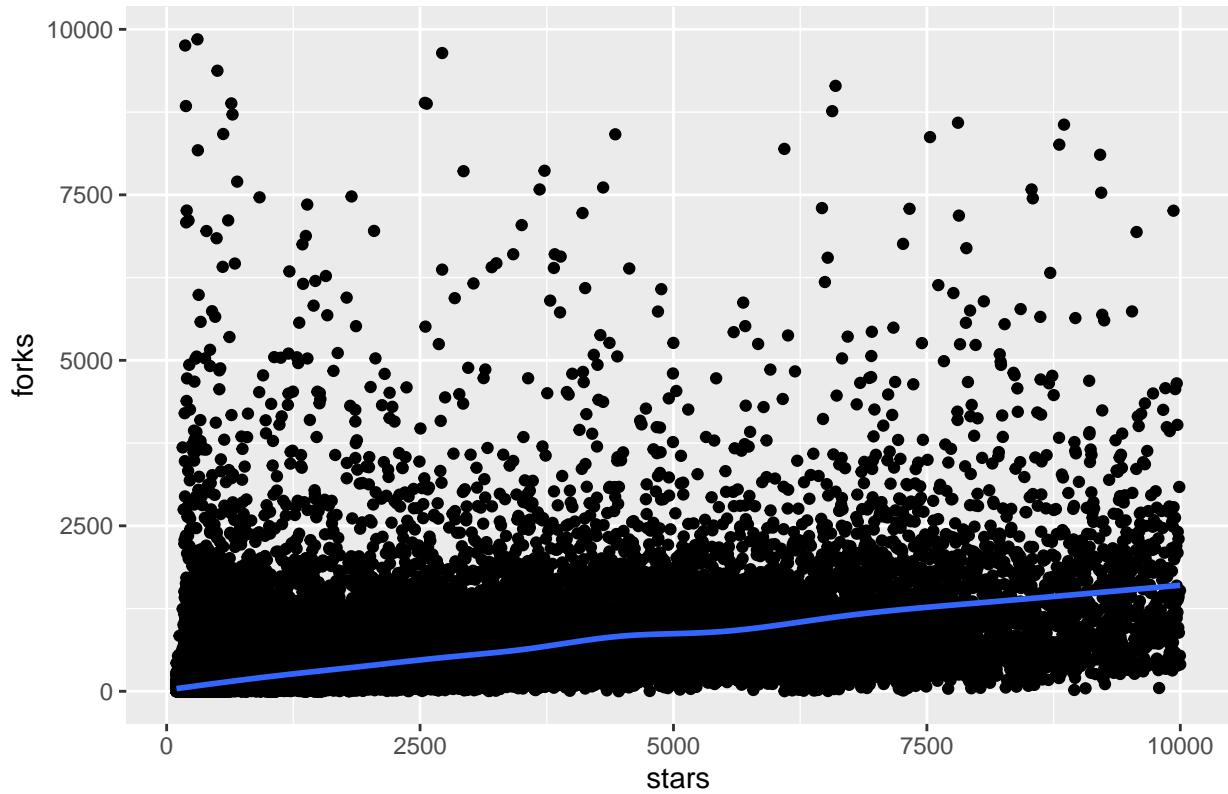
## Relation entre Stars et Forks



Affiner davantage la portée:

```
ggplot(df[df$stars < 10000 & df$forks < 10000 , ], aes(x = stars, y = forks)) +  
  geom_point() +  
  geom_smooth(method = "gam", se = FALSE) +  
  labs(title = "Relation entre Stars et Forks")  
  
## `geom_smooth()` using formula = 'y ~ s(x, bs = "cs")'
```

## Relation entre Stars et Forks



Après avoir filtré de nombreuses valeurs aberrantes, nous essayons de recalculer les deux coefficients de corrélation et de les comparer avec les résultats du calcul précédent.

```
correlation_pearson_ajuste <- cor(df[df$stars < 10000 & df$forks < 10000 , ]$stars, df[df$stars < 10000 & df$forks < 10000 , ]$forks)
correlation_spearman_ajuste <- cor(df[df$stars < 10000 & df$forks < 10000 , ]$stars, df[df$stars < 10000 & df$forks < 10000 , ]$forks, method = "spearman")
print(correlation_pearson_ajuste)

## [1] 0.5870711
print(correlation_spearman_ajuste)

## [1] 0.6363605
print(correlation_pearson)

## [1] 0.5847746
print(correlation_spearman)

## [1] 0.6531888
```

Nous pouvons constater que les deux coefficients de corrélation n'ont pas beaucoup changé.

On peut donc dire qu'après avoir éliminé de nombreuses valeurs aberrantes, le coefficient de corrélation entre **stars** et **forks** montre toujours une corrélation positive modérée, ce qui montre qu'il existe effectivement un certain degré de corrélation entre elles, mais qu'il n'y a pas de relation linéaire évidente. Cette situation peut refléter la complexité et la diversité des données, c'est-à-dire que la popularité (**stars**) et la participation (**forks**) des dépôts sont affectées par de multiples facteurs et ne peuvent être simplement décrites par un modèle linéaire.

Par conséquent, nous pouvons conclure que la relation entre **stars** et **forks** est statistiquement modérément

positive, mais n'a pas de relation linéaire évidente et peut être affectée par divers facteurs. Cette conclusion contribue à une compréhension et une analyse plus approfondies de la relation entre la popularité des dépôts et la participation.