

# Projet IF36

Baptiste Toussaint

2024-06-12

## 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

---

## 1. Introduction

### 1.1 Description des données.

#### 1.1.1 Présentation de GitHub

# GitHub

Figure 1: Logo de la mascotte de GitHub



Figure 2: Logo 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](#).

Sur *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.

### 1.1.2 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 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.

### 1.1.3 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**].

#### 1.1.4 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](#).

---

## 1.2 Plan d’analyse

### 1.2.1 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 réseaux 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.

### 1.2.2 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 un utilisateur *watches* (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.

### 1.2.3 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.

### 1.2.4 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*...

### 1.2.5 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.

## 1.3 Préparation des données

### 1.3.1 Charger le jeu de données

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

# thème pour les graphiques
custom_theme <- theme(
    panel.background = element_rect(fill = "#D9E8F1",
                                    colour = "#6D9EC1",
                                    size = 1, linetype = "solid")
)

## Warning: The `size` argument of `element_rect()` is deprecated as of ggplot2 3.4.0.
## i Please use the `linewidth` argument instead.
## This warning is displayed once every 8 hours.
## Call `lifecycle::last_lifecycle_warnings()` to see where this warning was
## generated.
```

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 :

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.

### 1.3.2 Triater les 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_duplicates <- df[duplicated(df),] # stock les doublons dans un objet
head(df_duplicates) # 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' ~
## 2 labstack   echo  26331 2180      523 FALSE   FALSE   [ {'name' ~
## 3 alibaba    canal 26312 7356     1198 FALSE   FALSE   [ {'name' ~
## 4 floating~ floa~ 26311 1523      254 FALSE   FALSE   [ {'name' ~
## 5 portainer  port~ 26277 2223     471 FALSE   FALSE   [ {'name' ~
## 6 lukasz-m~ awes~ 26258 2638     988 FALSE   FALSE   []
## # 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]

```

## 2. Première étude : mesurer la popularité des dépôts

## 2.1 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.

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".

## 2.2 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 choisi 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.

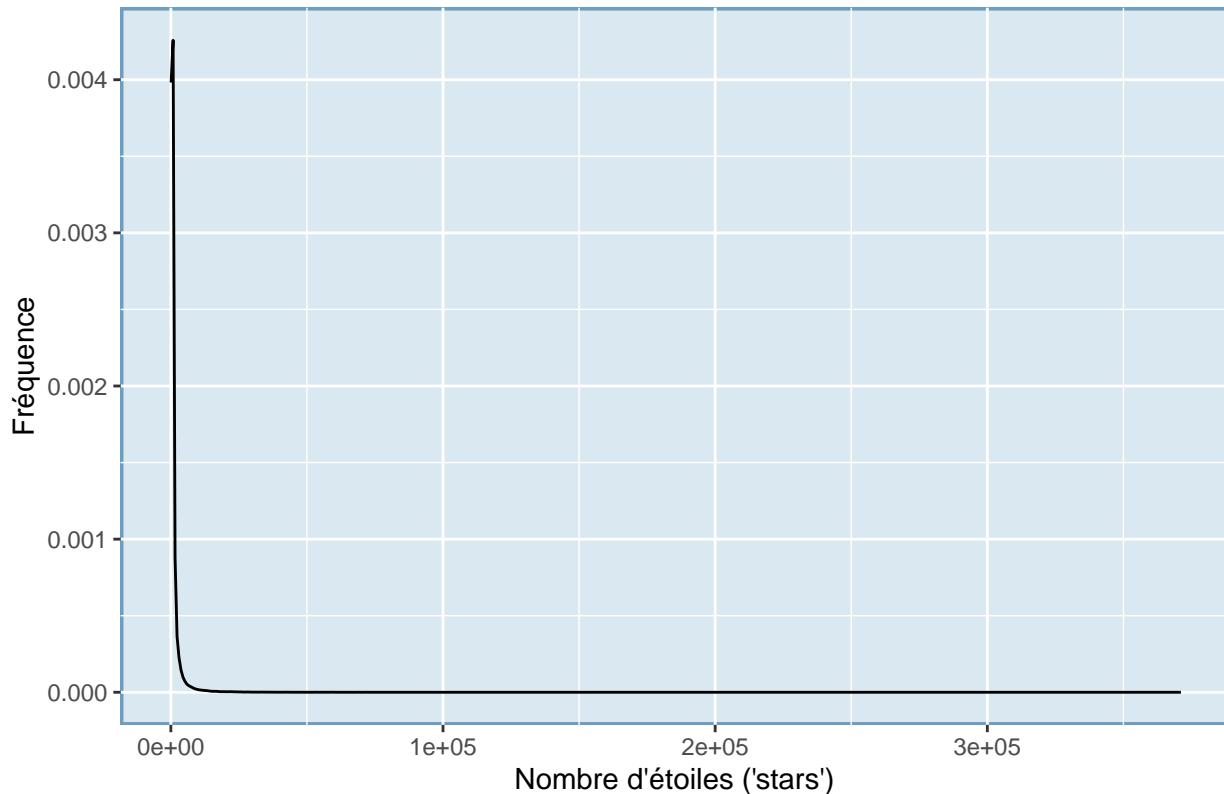
Pour simplifier le travail, je propose de trier le dataset en fonction du nombre d'étoile dans l'ordre décroissant

```
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") +
  custom_theme
```

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.

On peut afficher les données en découplant en 4 sous-jeu :

- les dépôts entre 97 et 1000 étoiles
- les dépôts entre 1000 et 10000 étoiles
- les dépôts à plus de 10000 étoiles
- les 1000 dépôts ayant le plus détoiles

grâce à notre DashBoard joint à ce rapport nous pouvons jouer sur les intervalles

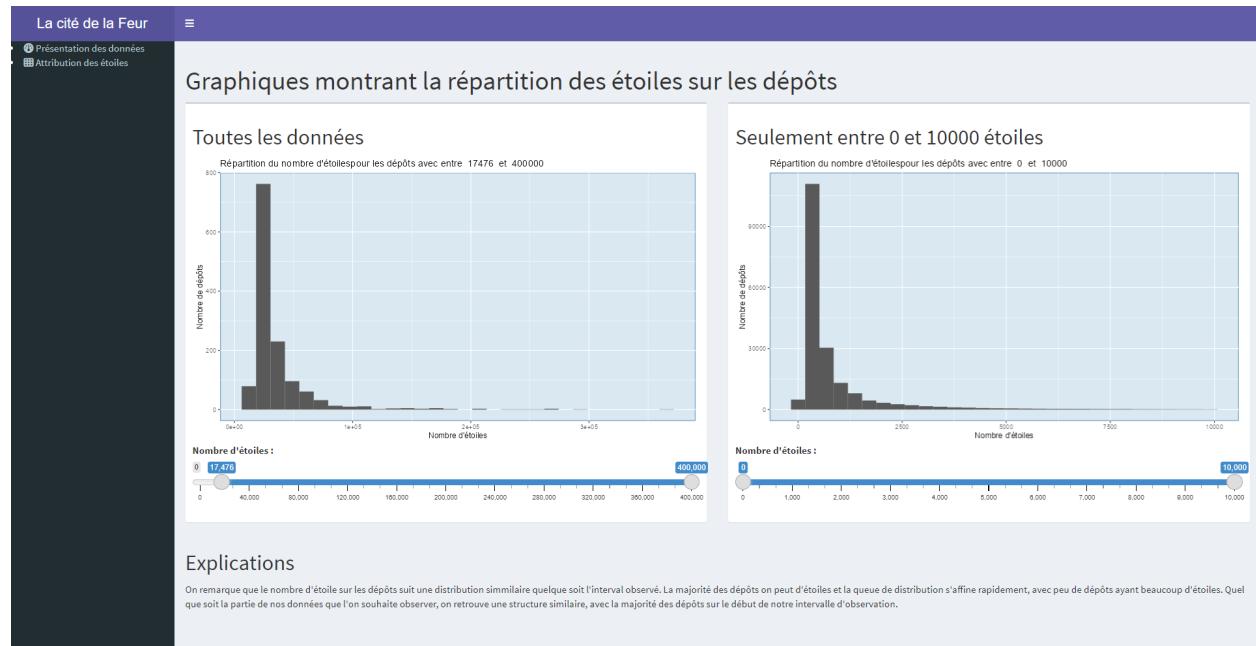
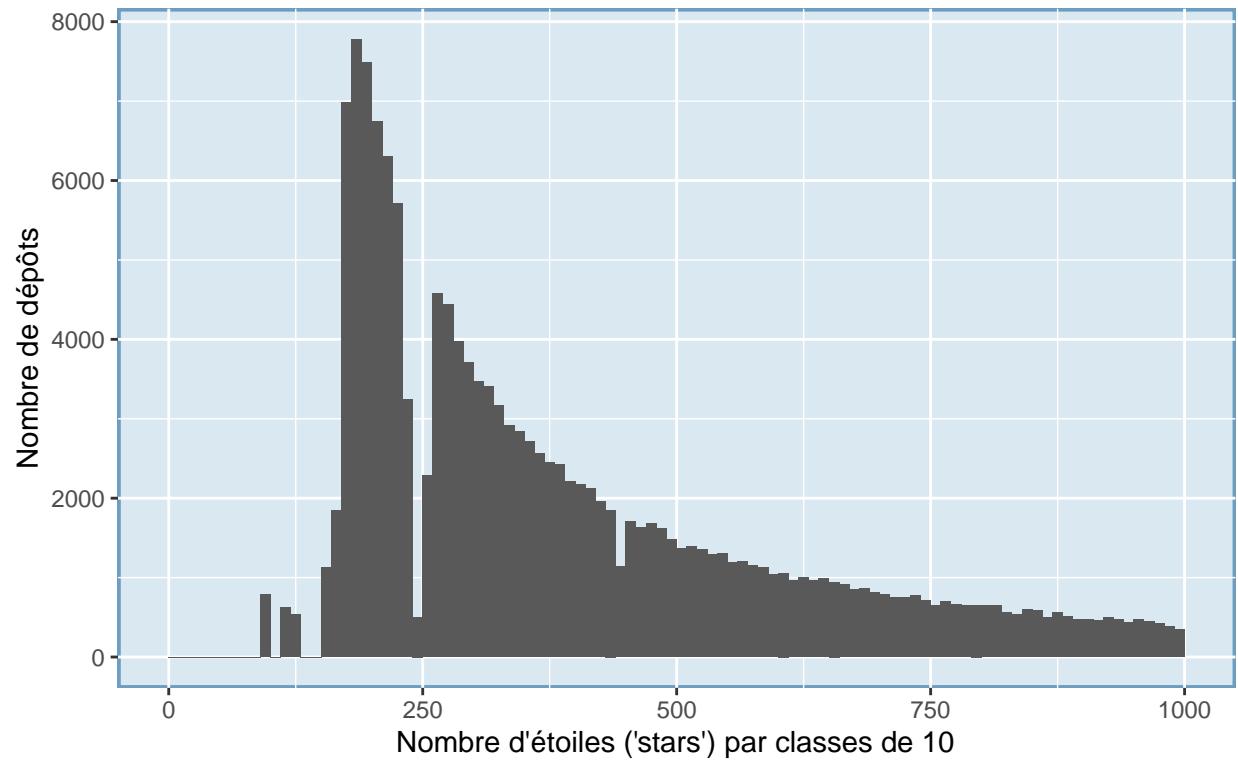
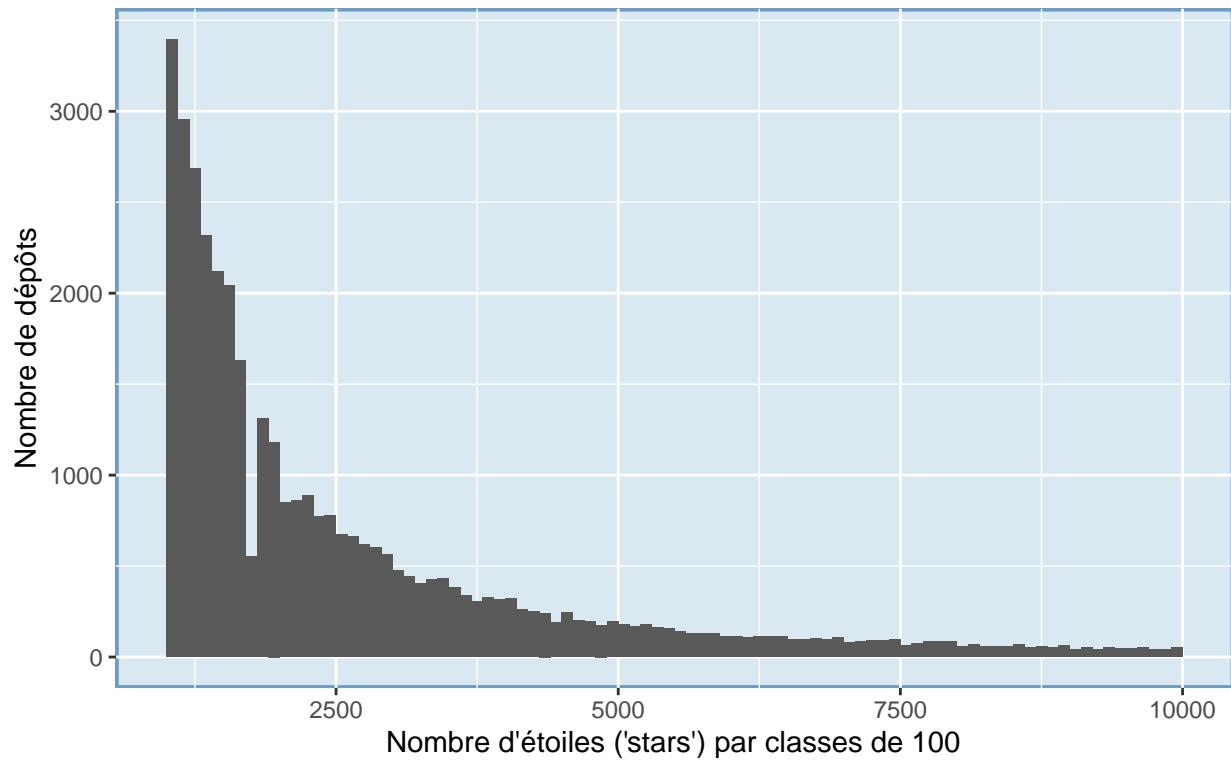


Figure 3: Capture d'écran du dashboard

Répartition du nombre d'étoiles  
pour les dépôts avec 1000 étoiles ou moins

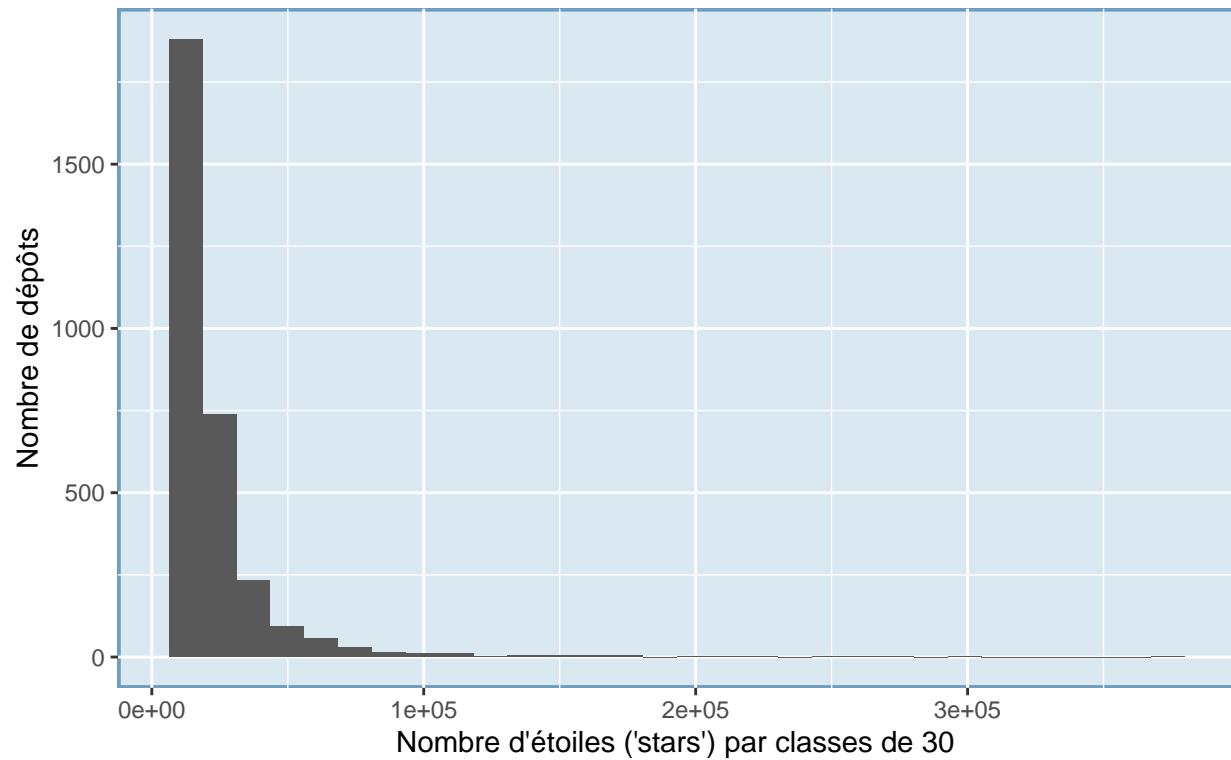


Répartition du nombre d'étoiles  
pour les dépôts entre 1000 et 10000 étoiles ou moins



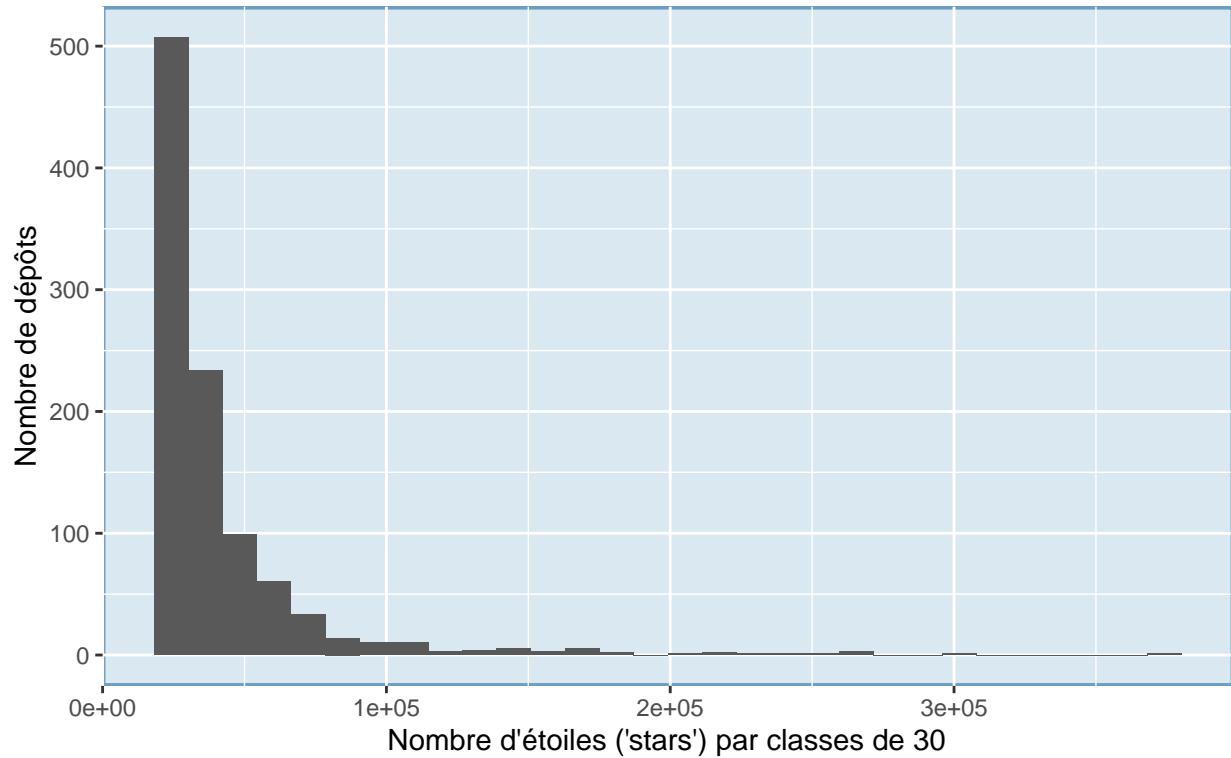
```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```

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



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

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



Quel que soit la partie de nos données que l'on souhaite observer, on retrouve une structure similaire, avec la majorité des dépôts sur le début de notre intervalle d'observation.

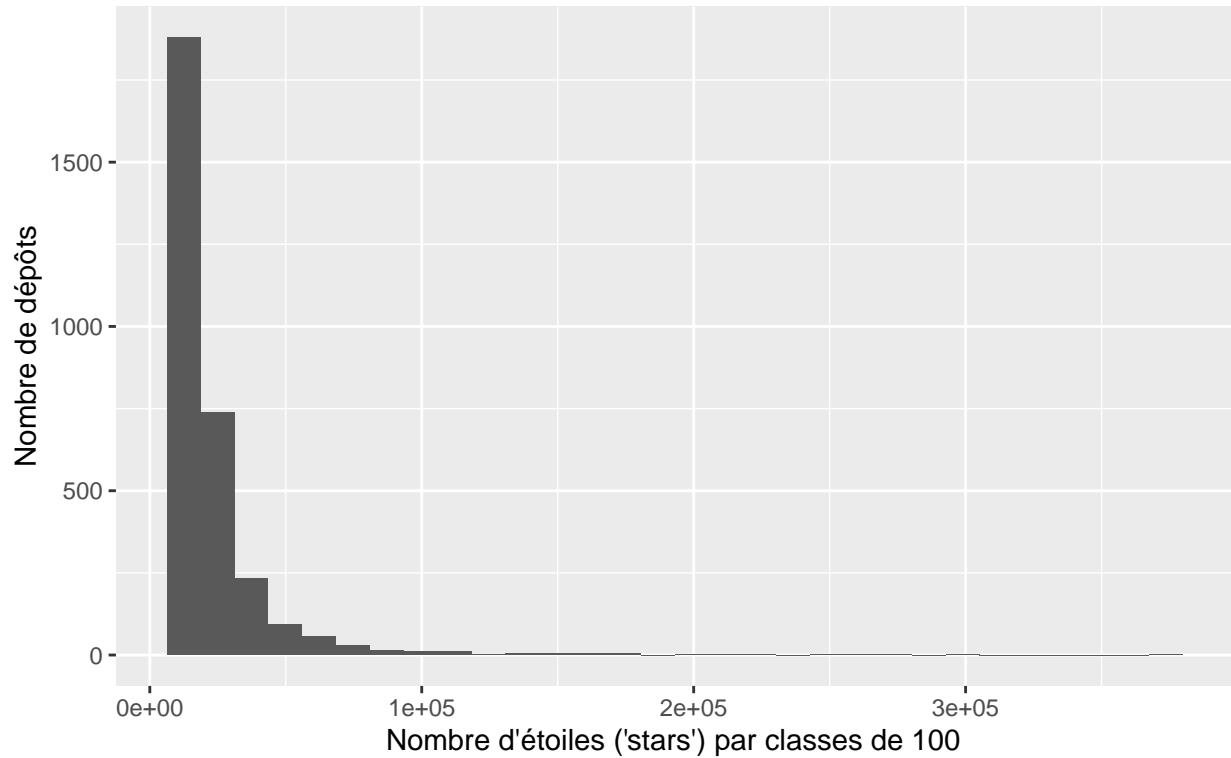
On comprend avec ce second graphique que la majorité des dépôts ont moins de 2500 étoiles et quelque uns en ont plus de 5000.

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

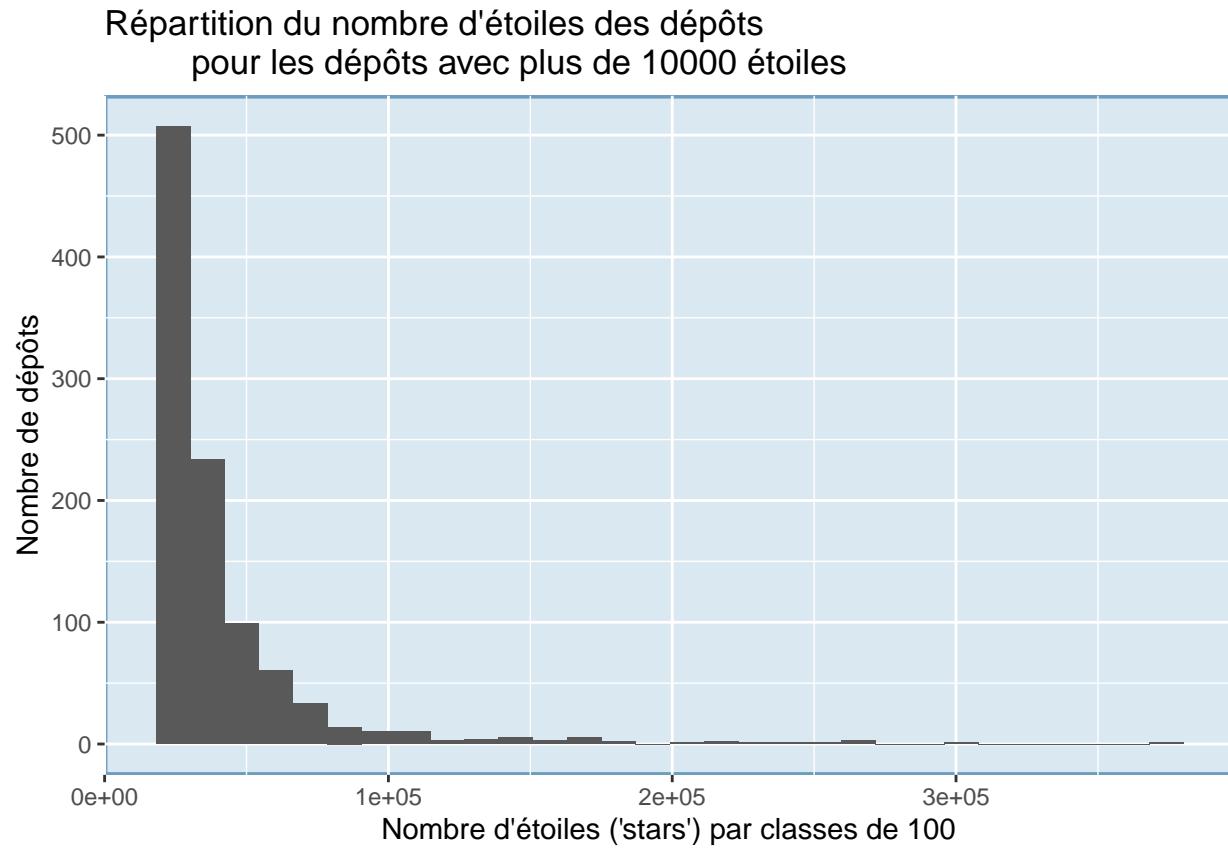


Bien que l'échelle ne soit plus la même (on atteint  $3 \times 10^5$  pour notre dépôt à 371122 étoiles), on retrouve une distribution très similaire.

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") +
  custom_theme

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



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

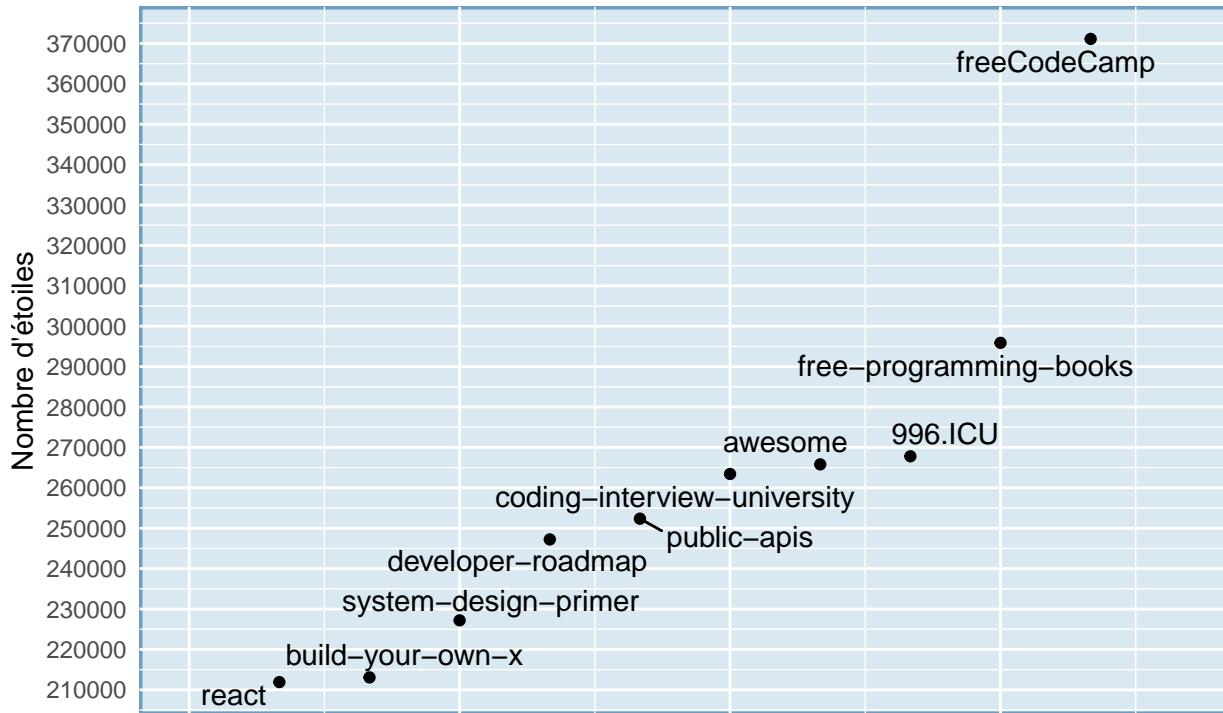
→

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 quelque 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()) +
  custom_theme
```

## 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"
```

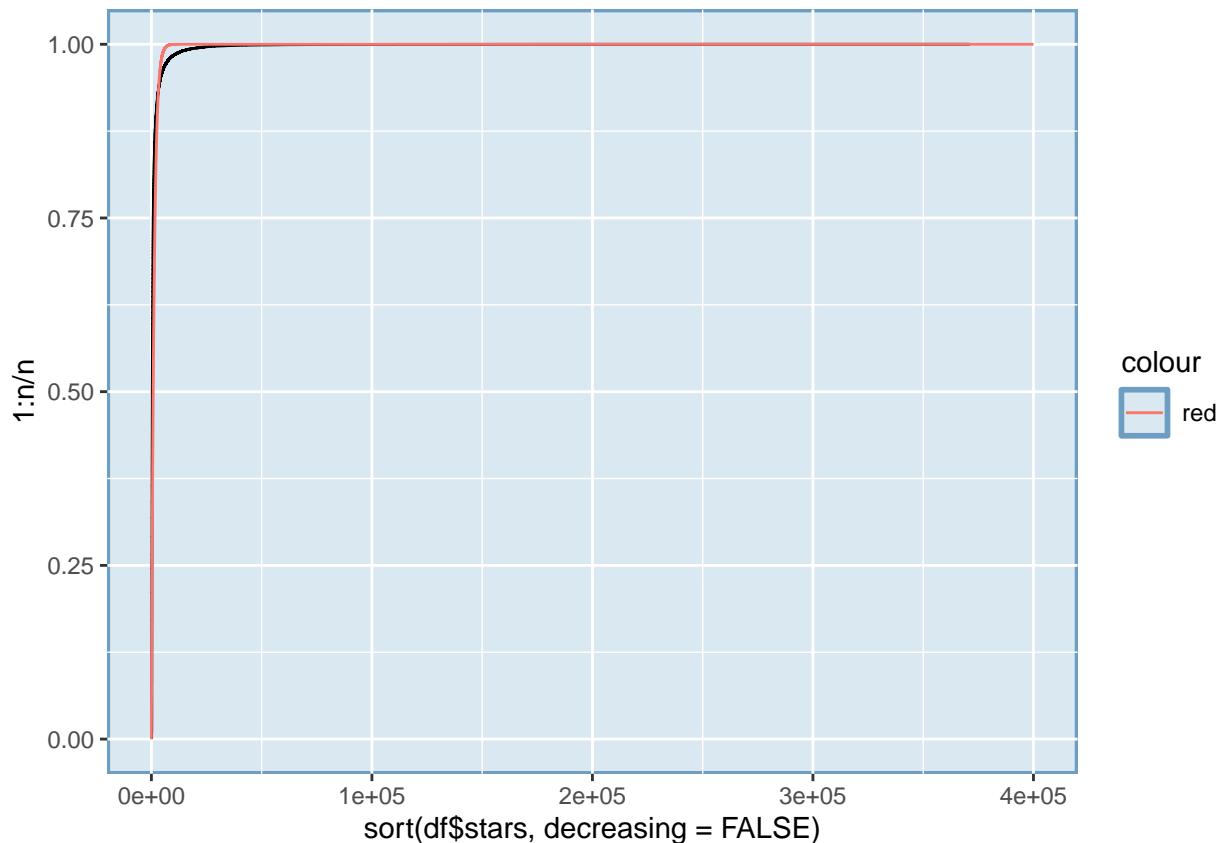
L'augmentation du nombre d'étoile n'est absolument pas linéaire et grimper dans le classement de *popularité* est de plus en plus difficile à mesure que l'on progresse.

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).

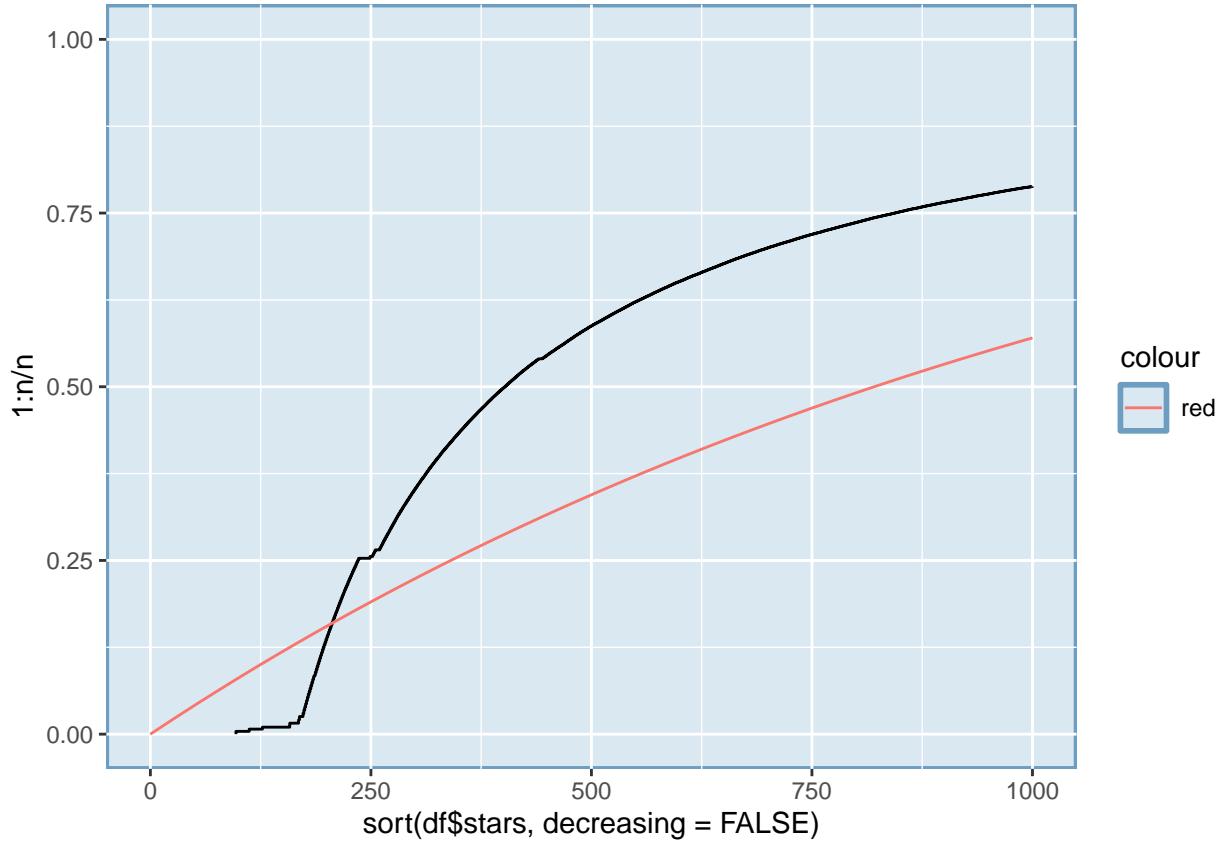
```
# crée une séquence de 1 à 400000
s <- seq(0,400000,1)
# mesure
lambda_mm <- 1/mean(df$stars)
ggplot() +
  # fonction de répartition mesurée
  geom_line(aes(x = sort(df$stars, decreasing = FALSE), y = 1:n/n)) +
  # loi exponentielle avec un paramètre estimé par méthode des moments
  # pour ce jeu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

```
geom_line(aes(x = s, y = pexp(s, rate = lambda_mm), color = 'red')) +
custom_theme
```



```
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) +
  custom_theme
```

```
## 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](#).

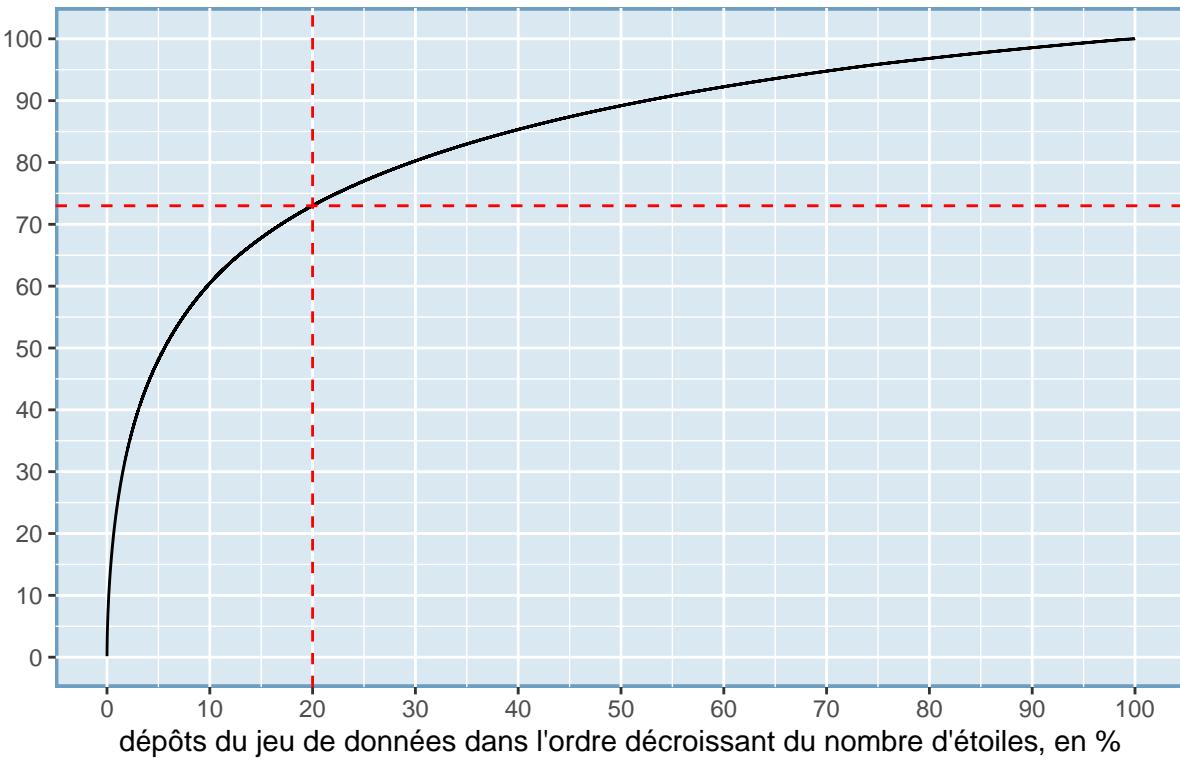
Selon ce principe, 20% des causes produisent 80% des effets. Dans notre cas, cela signifierait que 20% des dépôts de notre jeu de données représenteraient 80% du nombre total d'étoiles.

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") +
  custom_theme
```

Diagramme de Pareto du nombre d'étoiles en fonction des dépôts



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

### 2.3 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 utilise directement une échelle logarithmique pour éviter d'avoir un effet de "tassemement" des dépôts à cause des différences extrêmes dans les ordres de grandeur.

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.

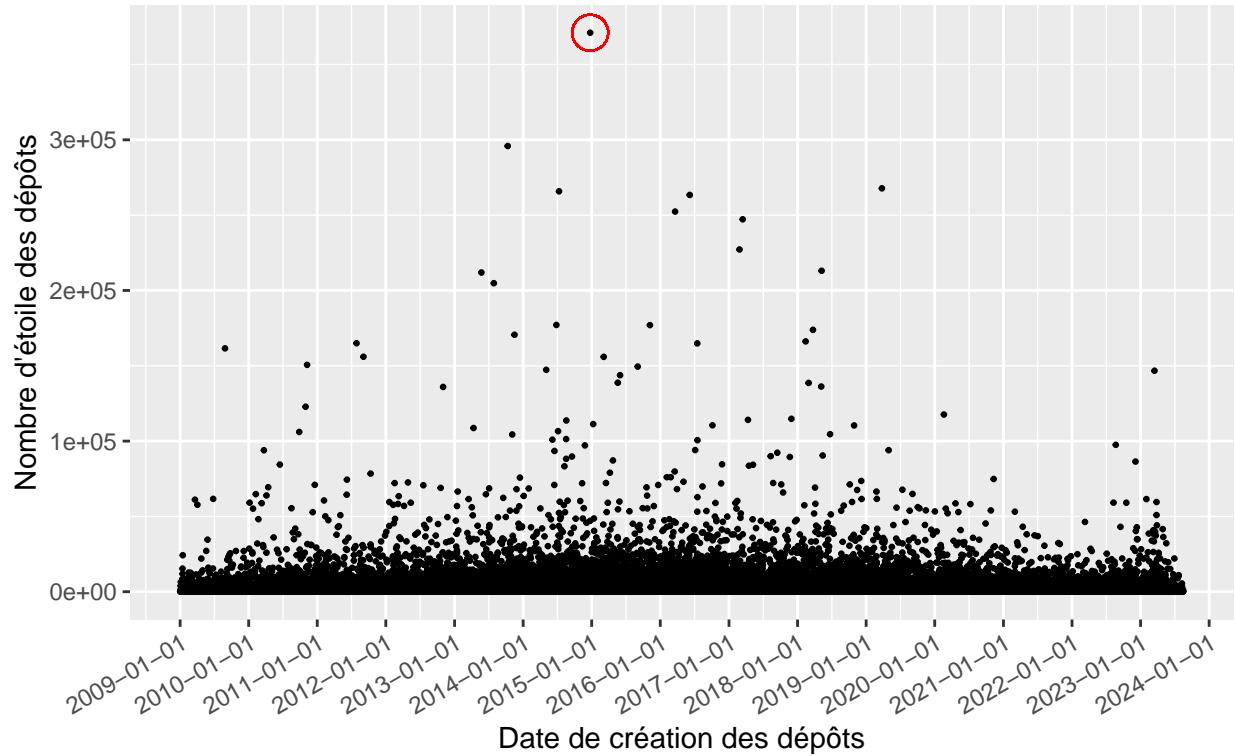
On retrouve notre dépôt `freeCodeCamp` qui possède le plus d'étoile en haut au centre :

```
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 centaines ( $10^2$ ) et des dépôts avec des centaines de milliers ( $10^5$ ).

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).

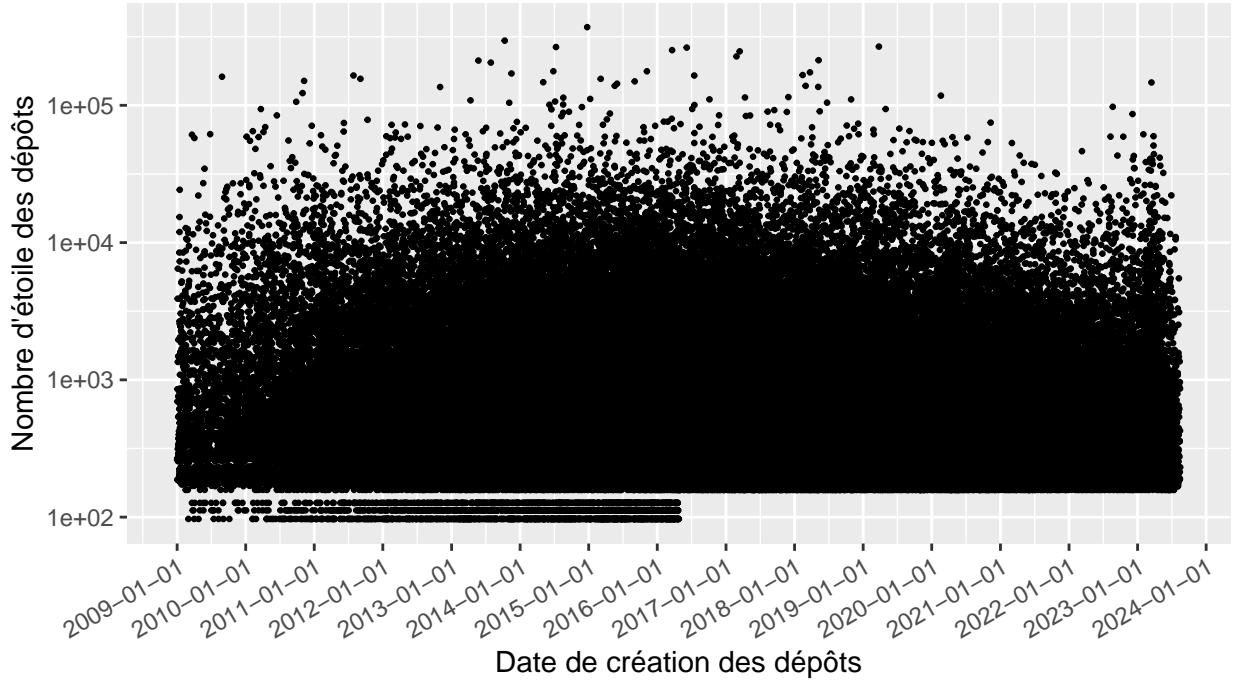
Si l'on utilise une échelle logarithmique pour le nombre d'étoile on s'en rend mieux compte que les dépôts "populaires"

```
# 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$creationDate))
separation_annees[length(separation_annees)] <- as.character(as.Date(separation_annees[length(separation_annees)] + 1))
# nbr = 0
mean_stars_by_year <- c()
for (i in 2:length(separation_annees)) {
  # nbr = nbr + dim(df[df$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()
```

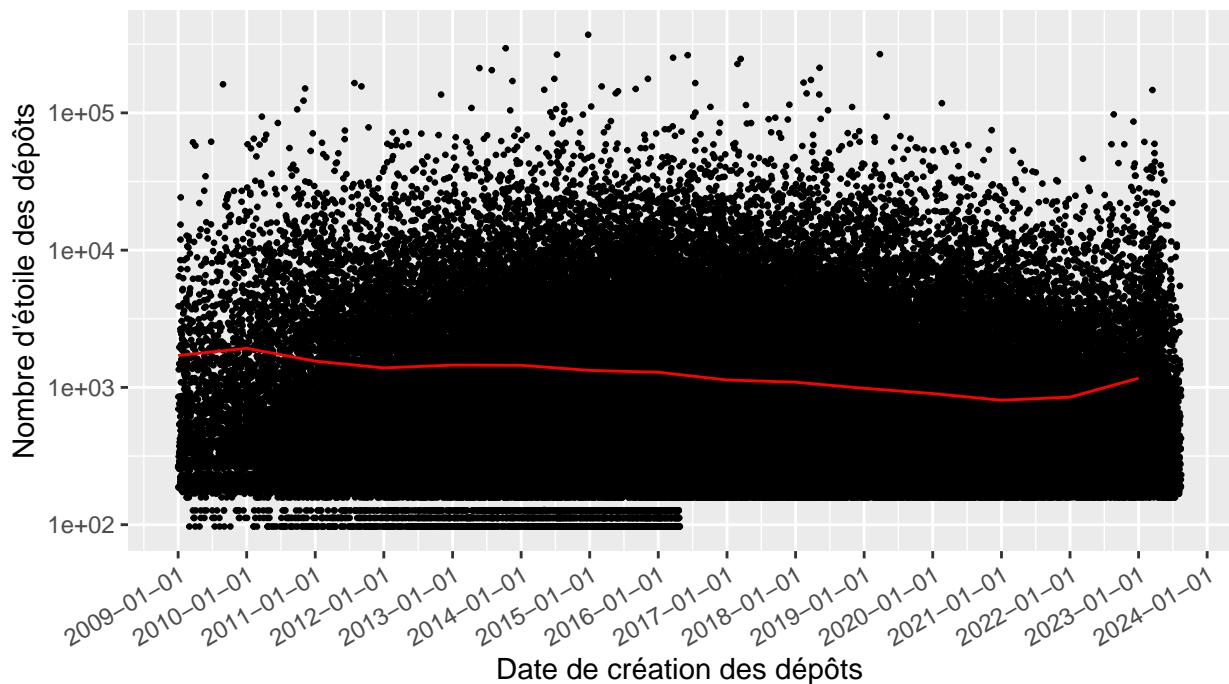
```

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.

->

On ajoute aussi au graphique 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

```

```

# 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

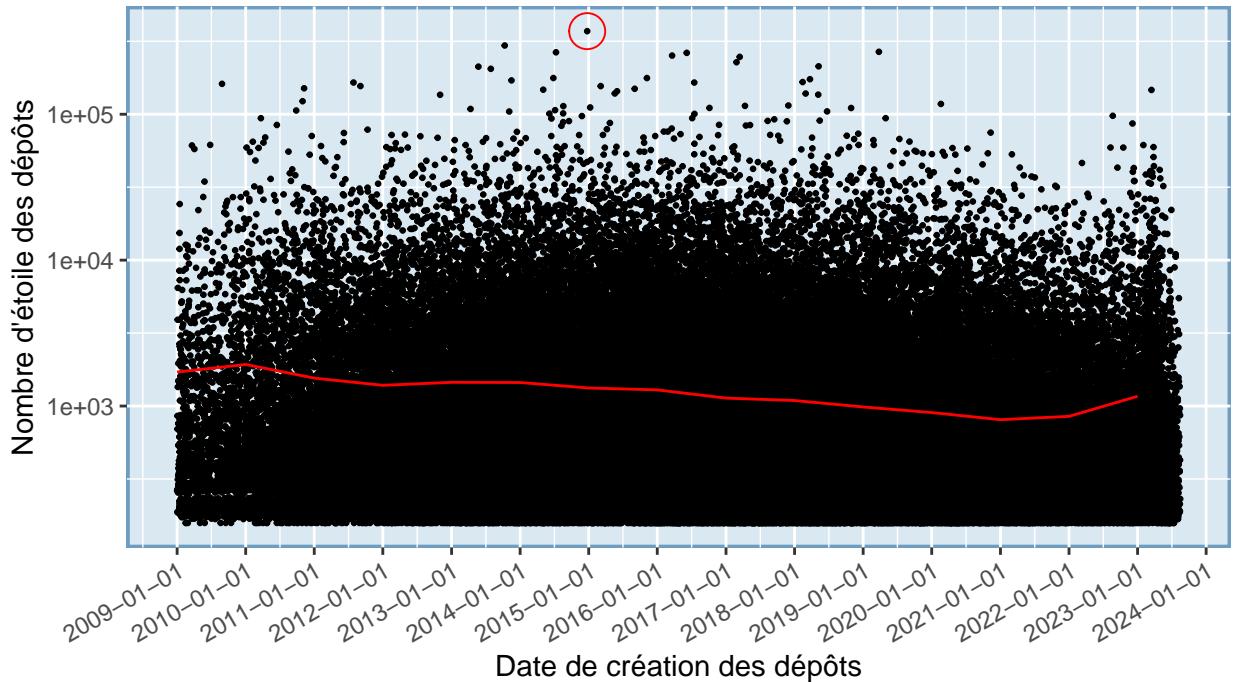
  mean_stars_by_year <- c(
    mean_stars_by_year,
    mean(df[df$creationDate < separation_annees[i] &
            df$creationDate >= separation_annees[i-1],]$stars)
  )
}

datebreaks <- seq(as.Date("2009-01-01"), as.Date("2024-01-01"), by = "1 year")
ggplot() +
  geom_point(aes(x = df[df$stars > 130,]$creationDate,
                 y = df[df$stars > 130,]$stars), size = 0.5) +
  geom_point(aes(x = df[df$name == 'freeCodeCamp',]$creationDate,
                 y = df[df$name == 'freeCodeCamp',]$stars),
             shape = 1,
             size = 6,
             color = 'red') +
  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") +
  custom_theme

```

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

Le nombre d'étoile est représenté par une échelle logarithmique



```
# Le lecteur averti aura remarqué que dans cette représentation, nous avons
# retiré les dépôts ayant moins de 130 étoiles.
# ces derniers perturbent la représentation graphique en ajoutant une ligne
# continue en bas du nuage du point pour les dates de 2009 à 2015
# ce qui perturbe la représentation graphique.
# Les retirer n'enlève en rien les conclusion faite sur ce graphique.
# et participe uniquement au confort de lecture.
```

L'échelle logarithmique rendant plus difficile l'interprétation du graphique, nous avons entouré le dépôt `freeCodeCamp` en rouge. Ce dépôt qui possède le plus d'étoile a été créé en 2015 et l'échelle logarithmique le rapproche du reste des dépôts dans le représentation graphique.

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 populaires créés dans les premières années de GitHub (2009-2010), comme des dépôt populaires très récents (2023).

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.

## 2.4 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.

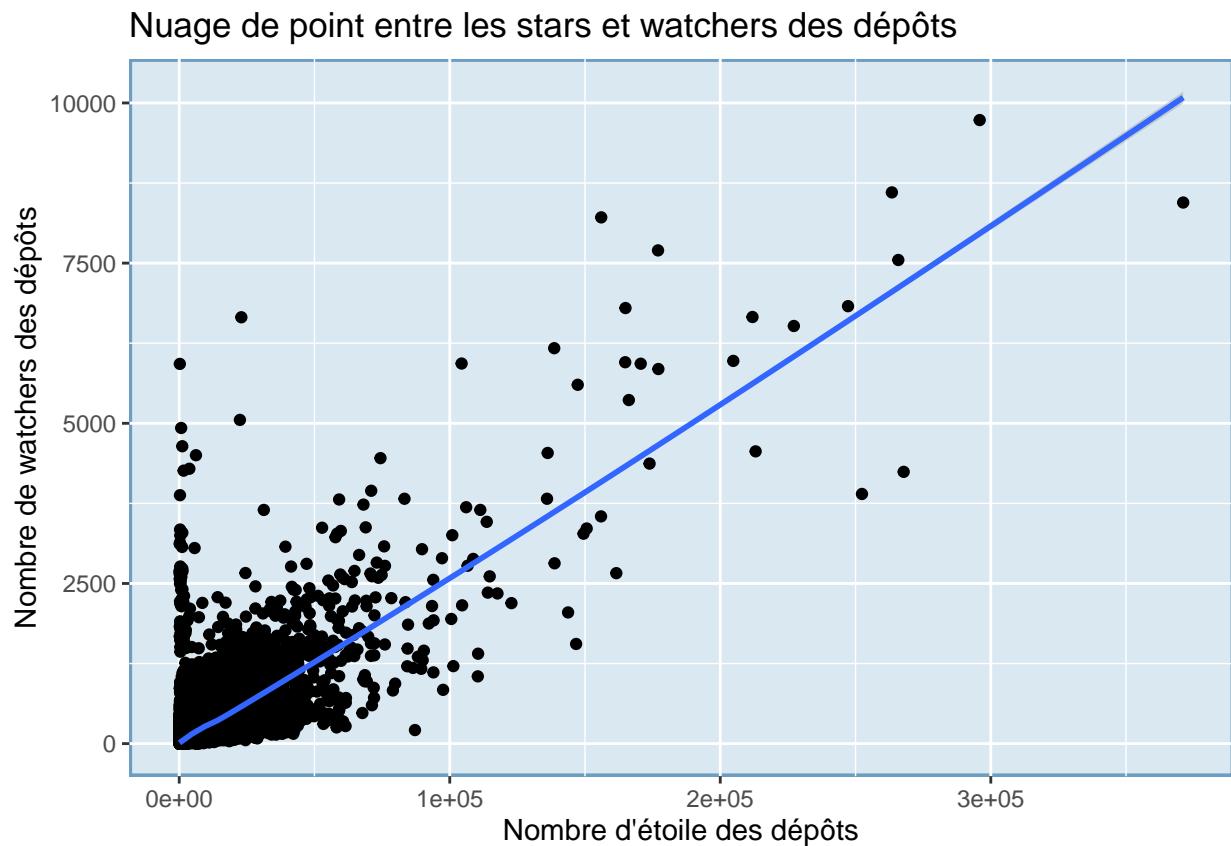
Si l'on reprend la métaphore d'un réseau social : un utilisateur peu *liker* un dépôt qu'il apprécie (ajoute une étoile *stars*) mais il peut également *follow* le dépôt pour être mis au courant des évolutions.

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](#).

```
ggplot(df, aes(x = stars, y = watchers)) +
  geom_point() +
  geom_smooth() +
  labs(x = "Nombre d'étoile des dépôts",
       y = "Nombre de watchers des dépôts",
       title = "Nuage de point entre les stars et watchers des dépôts") +
  custom_theme

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



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**)

Nous avons réalisé ce même graphique en utilisant Tableau :

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

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

Nuage de points entre les stars et watchers des dépôts

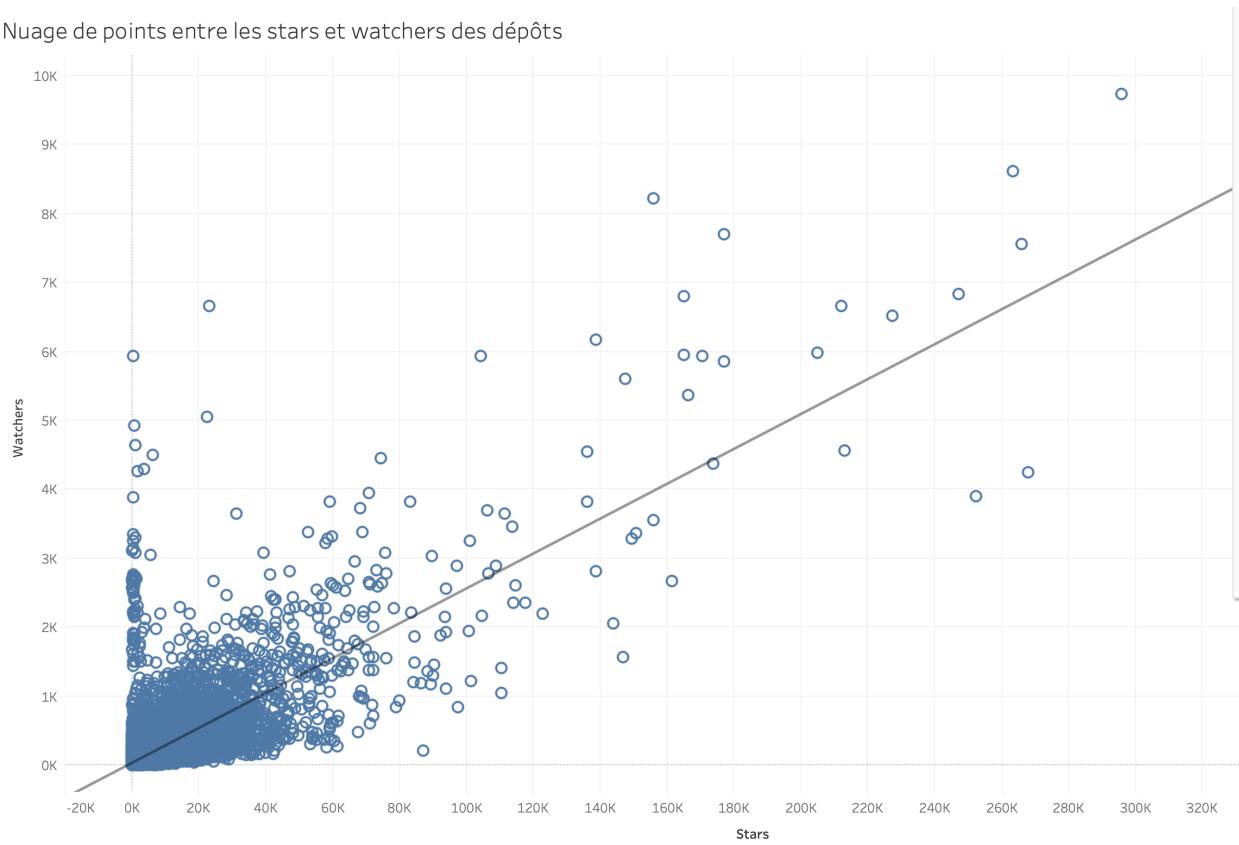


Figure 4: Capture d'écran de Tableau

```

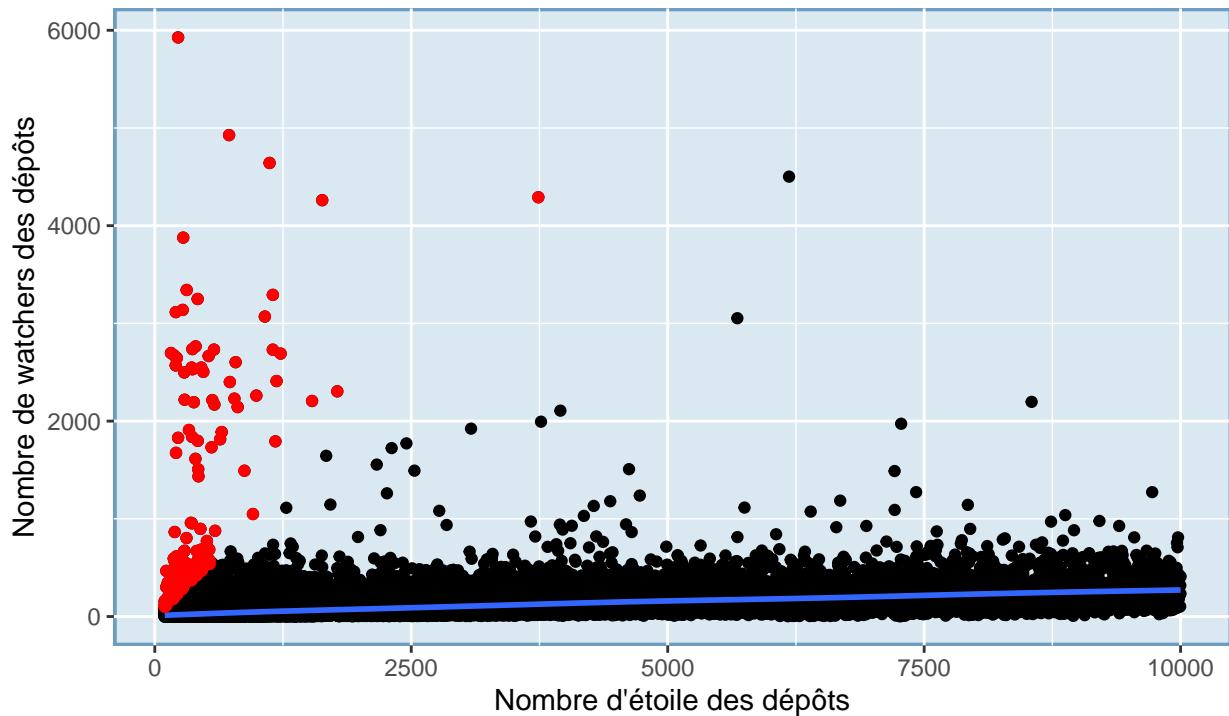
geom_smooth(aes(x = df[df$stars <= 10000,]$stars,
                y = df[df$stars <= 10000,]$watchers)) +
geom_point(aes(x = df[df$stars <= df$watchers,]$stars,
                y = df[df$stars <= df$watchers,]$watchers),
                color = 'red') +
labs(x = "Nombre d'étoile des dépôts",
     y = "Nombre de watchers des dépôts",
     title = "Nuage de point entre les stars et watchers des dépôts
pour les dépôts de moins de 10000 étoiles.",
     subtitle = "En rouge les dépôts avec moins d'étoiles que de watchers") +
custom_theme

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

```

### NUAGE DE POINT ENTRE LES STARS ET WATCHERS DES DÉPÔTS POUR LES DÉPÔTS DE MOINS DE 10000 ÉTOILES.

EN ROUGE LES DÉPÔTS AVEC MOINS D'ÉTOILES QUE DE WATCHERS



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.

Nous avons réalisé ce même graphique en utilisant Tableau :

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). Nous les avons affiché en rouge sur le précédent graphique.

Si l'on observe les noms de ces dépôts :

```

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

```

Nuage de points entre les stars et watchers des dépôts pour les dépôts de moins de 10,000 étoiles

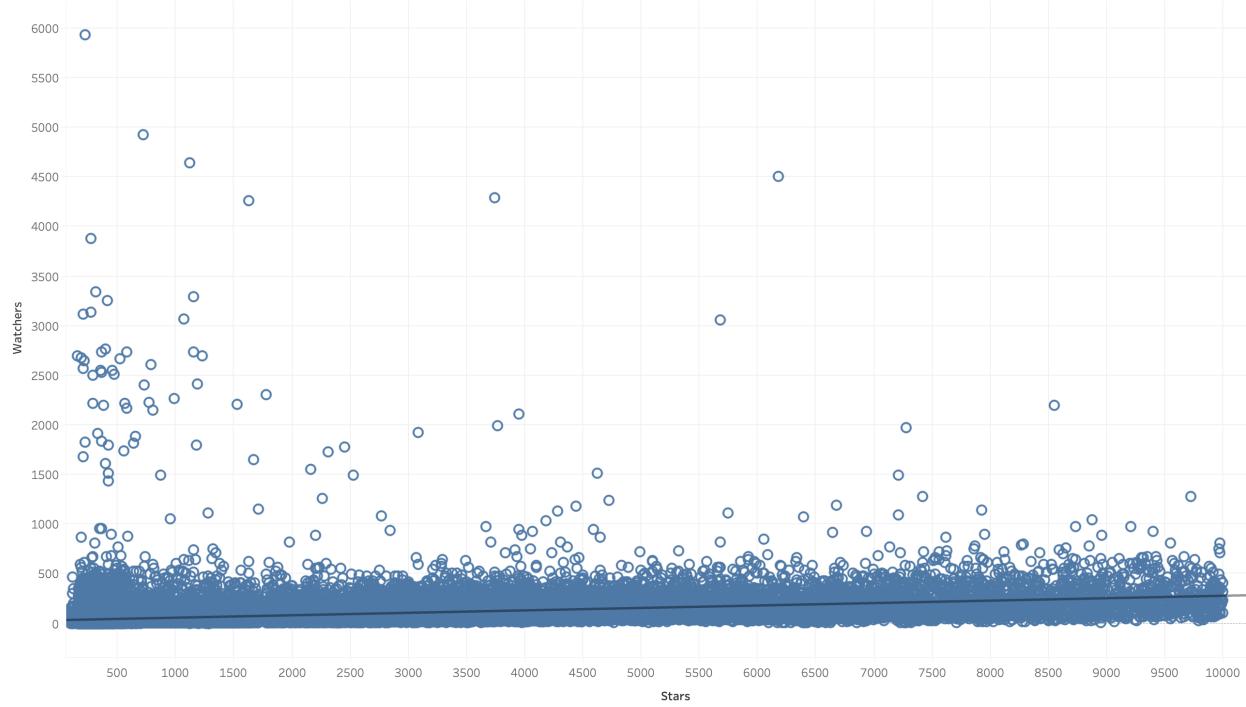


Figure 5: Capture d'écran de Tableau

```
## [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 remarque aussi grâce au courbe ajustées au nuages de points que l'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.

```
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 a d'étoile, plus il a tendance à avoir été suivi et donc avoir plus de **watchers**. Certains dépôts font exception à cette règle.

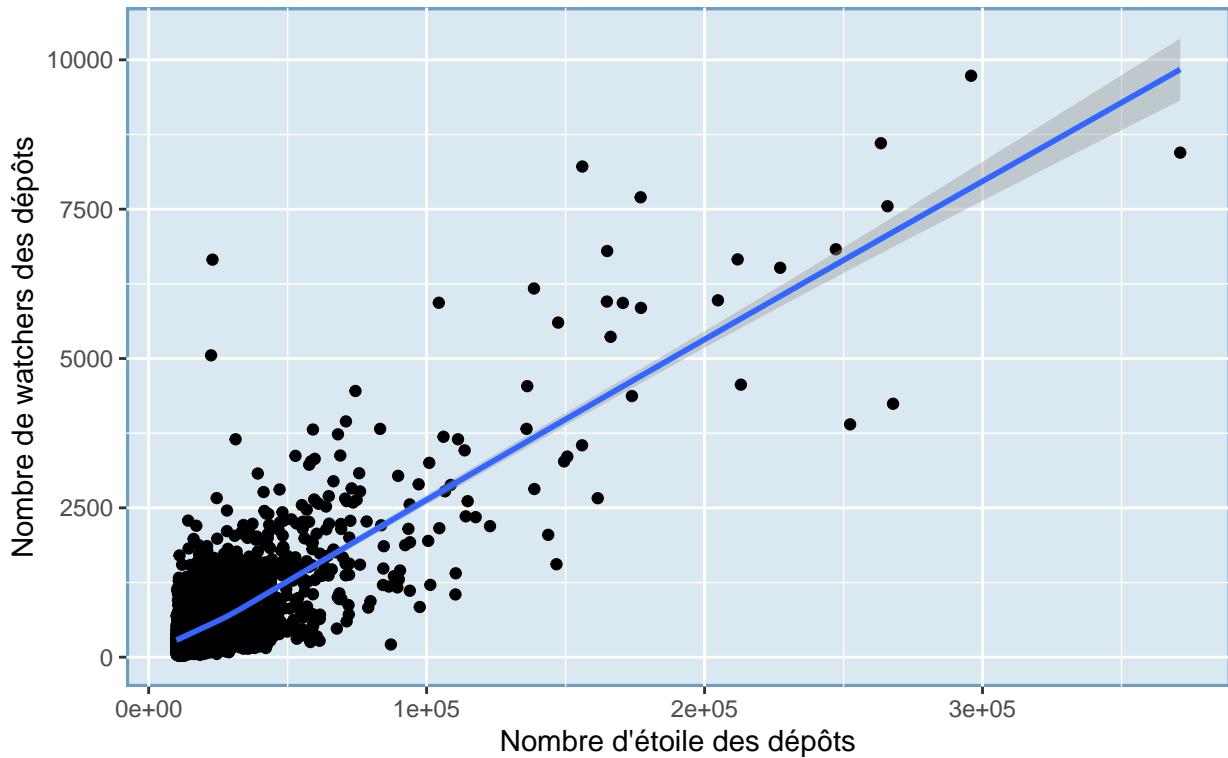
On pourrait pousser l'analyse statistique pour définir la qualité d'adéquation d'un modèle linéaire.

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

```
ggplot(df[df$stars > 10000], aes(x = stars, y = watchers)) +
  geom_point() +
  geom_smooth() +
  labs(x = "Nombre d'étoile des dépôts",
       y = "Nombre de watchers des dépôts",
       title = "Nuage de point entre les stars et watchers des dépôts
pour les dépôts de moins de 10000 étoiles.") +
  custom_theme

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

**Nuage de point entre les stars et watchers des dépôts pour les dépôts de moins de 10000 étoiles.**



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.

## 2.5 Existe-t-il une 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.

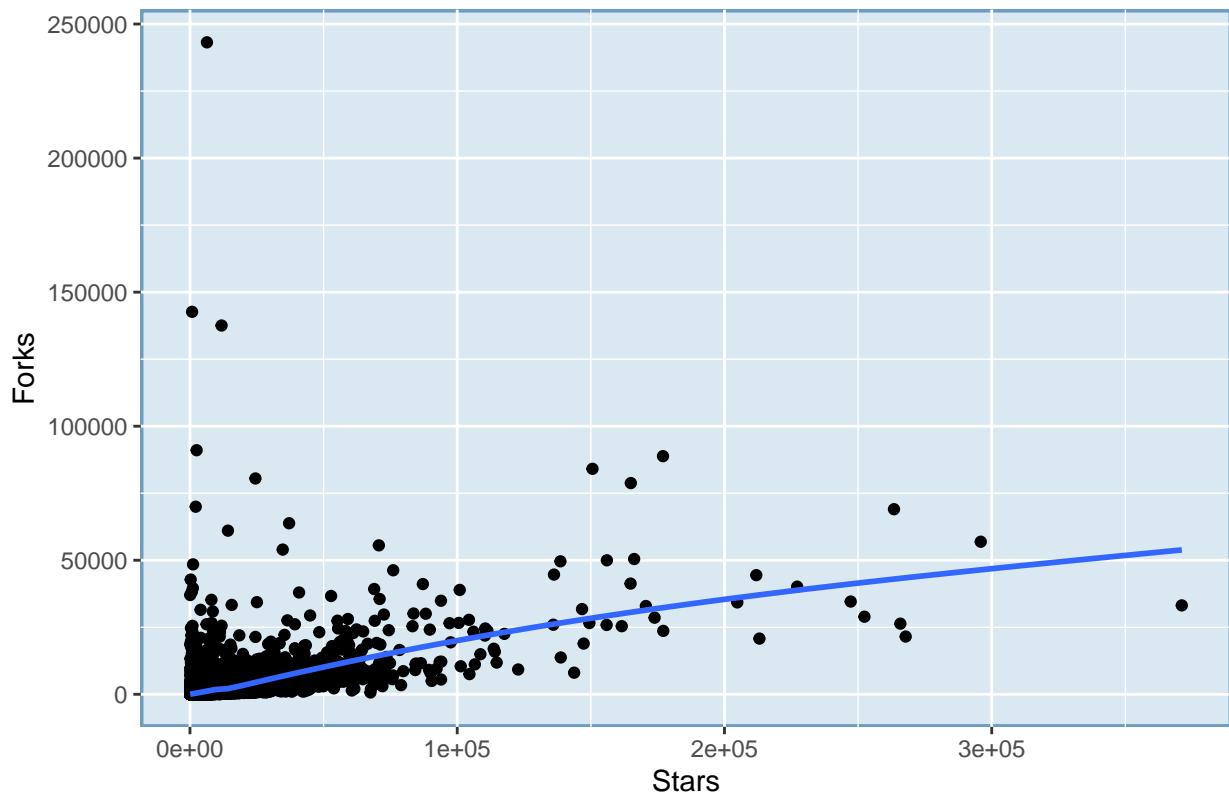
Coefficient de corrélation de rang de Spearman : 0,6531888, indiquant qu'il existe un certain degré de relation monotone entre **stars** et **forks**, mais il n'est pas nécessaire que la relation soit linéaire. Ce coefficient est légèrement supérieur au coefficient de corrélation de Pearson, ce qui indique 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") +
  custom_theme

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

## Relation entre Stars et Forks



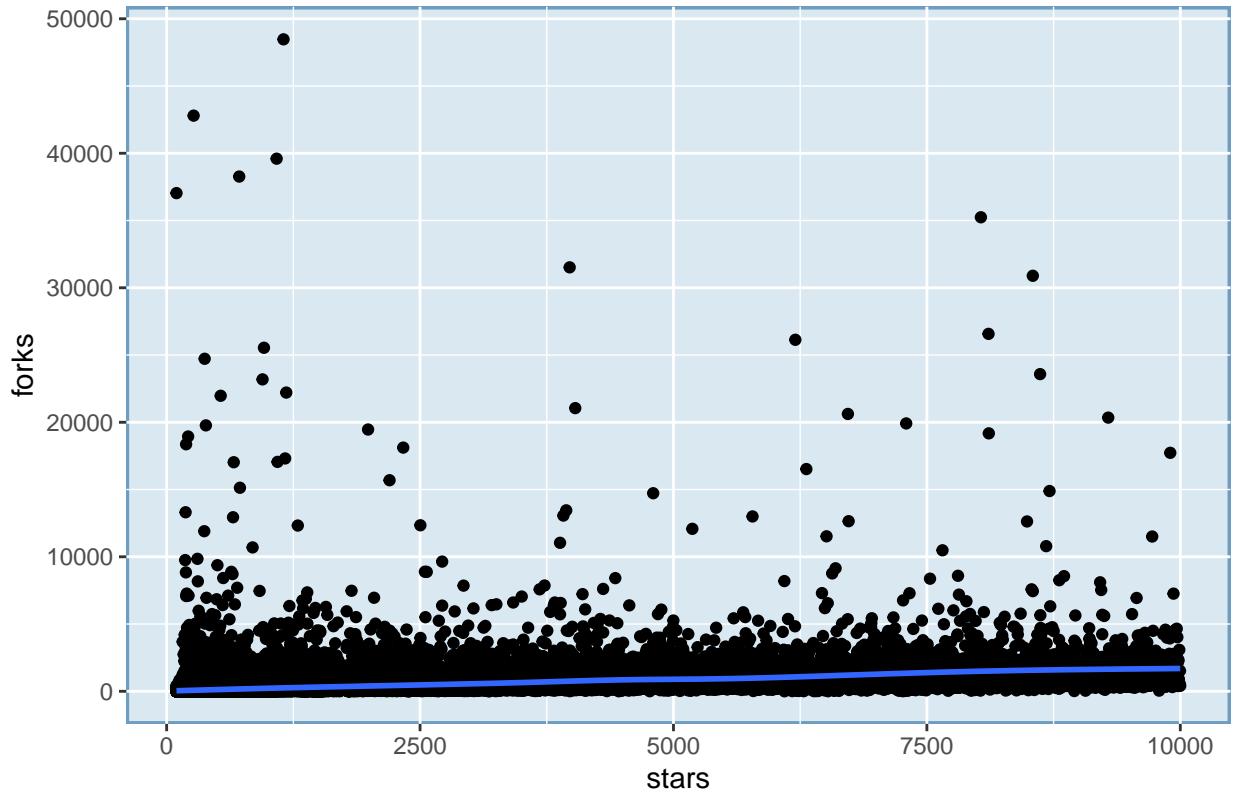
Nous avons constaté que certains dépôts ont un nombre de **stars** extrêmement faible, mais un nombre de **forks** extrêmement élevé. Il s'agit évidemment de valeurs aberrantes.

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") +
  custom_theme

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

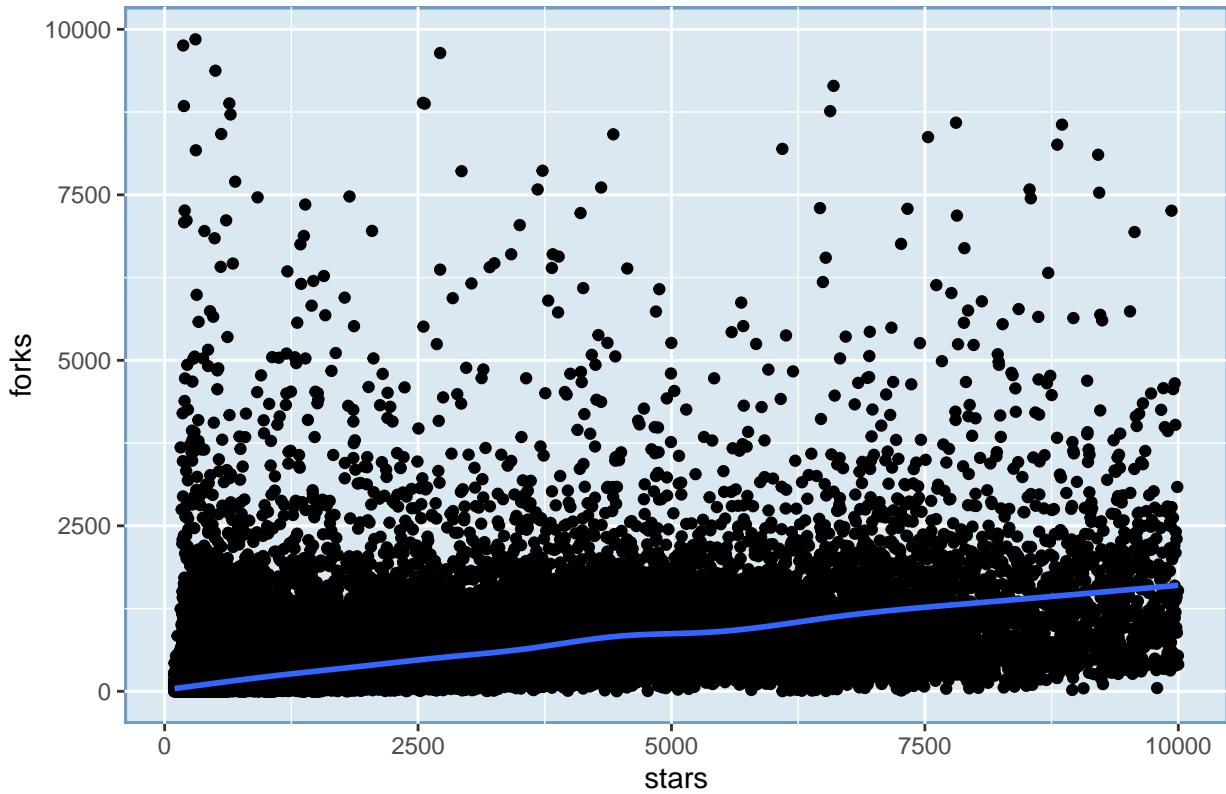
## Relation entre Stars et Forks



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") +  
  custom_theme  
  
## `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.

### 3. Deuxième étude :Étude des langages de programmation utilisés: *language*

#### 3.1 Notre interprétation du langages de programmation

Au cœur de l'écosystème numérique, les langages de programmation jouent un rôle crucial dans le développement et l'évolution des logiciels. Comprendre l'évolution de l'utilisation des langages de programmation est essentiel pour saisir les dynamiques de l'industrie informatique, ainsi que pour guider nos propres choix.

Il existe deux attributs dans notre jeu de données qui sont directement liés aux langages de programmation: `languages` et `primaryLanguage`.

Il existe un autre attribut lié au langage de programmation : `createdAt`. Lorsqu'il est observé avec `primaryLanguage`, il peut refléter le premier langage sélectionné lors de la création du projet. Lorsqu'il y a suffisamment d'échantillons, les langages de programmation populaires de chaque période peuvent être vus.

Car d'autres attributs, tels que `stars`, `forks` et `watchers`, peuvent refléter dans une certaine mesure la popularité de la langue.

#### 3.2 Popularité du langage de programmation

##### 3.2.1 Analyse par un histogramme

Puisque certains référentiels n'indiquent pas le langage de programmation, nous ne considérerons pas ces référentiels:

```
df_filtered <- df[df$languageCount > 0 , ]
```

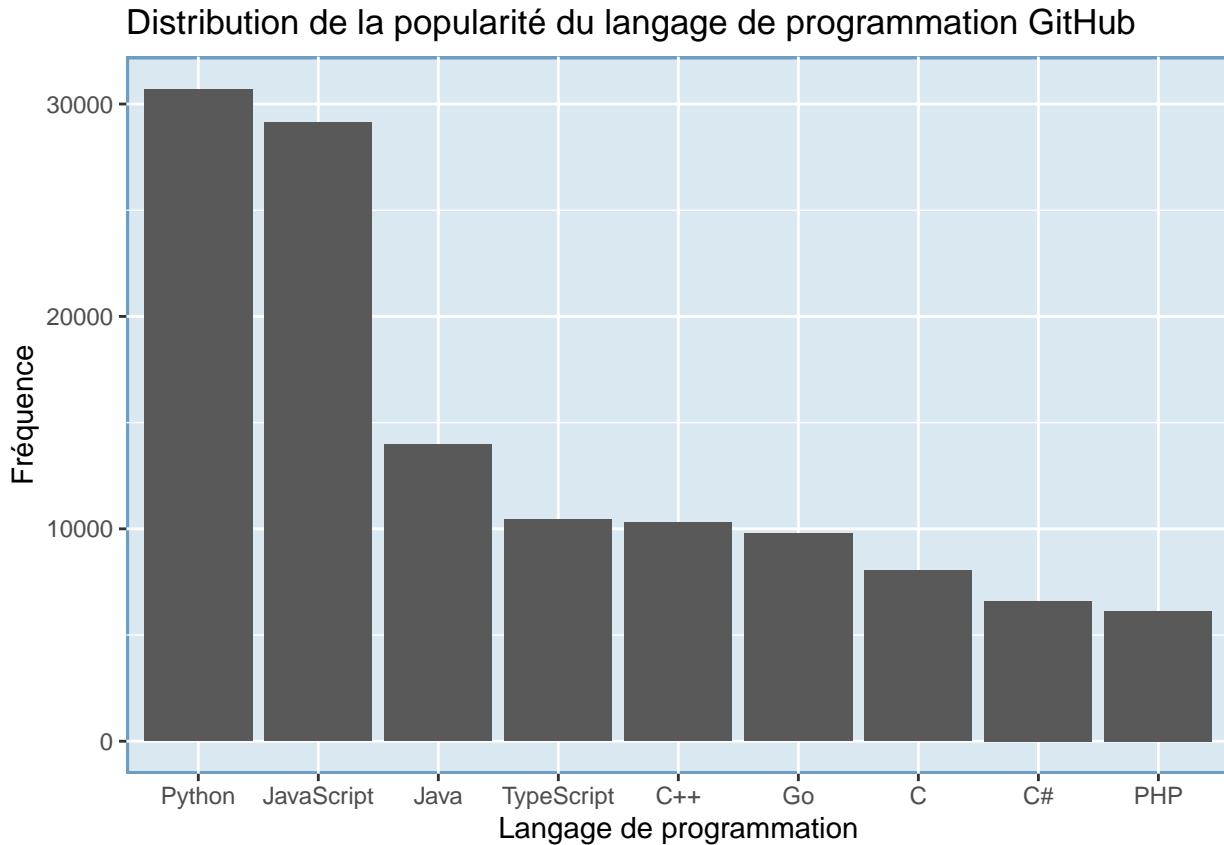
Nous pouvons simplement compter la fréquence d'apparition des langages de programmation principaux parmi les 20 000 premiers dépôts pour explorer la popularité des langages de programmation.

Nous pouvons utiliser un diagramme à barres pour découvrir la distribution des langages de programmation principaux dans les dépôts GitHub. Nous traçons la fréquence d'utilisation des principaux langages sur l'axe des y et les différents langages sur l'axe des x. Chaque barre représente un langage de programmation, et la hauteur de la barre correspond à sa fréquence d'utilisation.

Ce choix de graphique est pertinent car il permet de visualiser de manière claire et directe les différences de popularité entre les langages. Le diagramme à barres facilite la comparaison visuelle des fréquences, ce qui est essentiel pour identifier rapidement les langages les plus et les moins utilisés. De plus, sa simplicité rend les résultats facilement compréhensibles pour un large public.

```
language_distribution <- df_filtered %>%  
  group_by(primaryLanguage) %>%  
  summarise(count = n()) %>%  
  arrange(desc(count))  
  
ggplot(language_distribution[language_distribution$count>5000,], aes(x =  
  reorder(primaryLanguage, -count), y = count)) +  
  geom_bar(stat = "identity") +  
  
  labs(title = "Distribution de la popularité du langage de programmation GitHub",  
       x = "Langage de programmation",
```

```
y = "Fréquence") +
custom_theme
```



On constate que le principal langage de programmation qui apparaît le plus fréquemment dans ces 20 000 dépôts GitHub est Python et JavaScript, suivis dans l'ordre par Java, TypeScript et C++ etc. On peut grossièrement le diviser en quatre niveaux selon la fréquence d'utilisation: Python et JavaScript sont les plus courants, suivis de Java. TypeScript, C++, Go et C troisième et les autres.

### 3.2.2 Alternative : représentation avec un nuage de mots

Une autre manière de représenter ces résultats est d'utiliser un graphique en nuage de mots : *cloud word*.

```
# charge les librairies utiles pour le graphique en nuage de mots
library(wordcloud)
```

```
## Le chargement a nécessité le package : RColorBrewer
library(RColorBrewer)
library(stringr)
library(ggwordcloud)
```

```
# Pour tracer ce graphique, j'utilise la variable nommée languages de notre
# dataset qui est une chaîne de caractère en format json qu'il faut traiter au
# préalable avec une opération de regex.
# Appliquée à tous les dépôts
text <- df$languages
pattern <- "'name': '[^']+'" # Mon patern regex utilisé
matching <- str_extract_all(text, pattern)
```

```

langages <- mapply(matching,
                    FUN = substr,
                    start = 10,
                    stop = 1000)

# Convertir en une unique liste qui liste la fréquence d'appartition du langage
# dans les dépôts
langagesVec <- unlist(langages)
word_freqs <- table(langagesVec) %>% as.data.frame()
colnames(word_freqs) <- c("word", "freq")

set.seed(1234) # for reproducibility
# windows()
wordcloud(
  words = word_freqs$word,
  freq = word_freqs$freq,
  min.freq = 1,
  max.words=200,
  random.order=FALSE,
  color=brewer.pal(8, "Dark2"),
  random.color = TRUE,
  fixed.asp = TRUE
)

```



Ce type de visualisation permet de rendre compte d'une impression, d'un ordre de grandeur: certains langages sont bien plus représentés que d'autres. Cette représentation est très jolie et agréable à regarder et

conviendrait parfaitement à une utilisation à visée marketing ou de communication. Par exemple, GitHub pourrait parfaitement utiliser ce type de visuel sans ses billets de blog annuels sur les grandes tendances du site.

Cependant, cette représentation n'est pas plus utile dans le cadre d'une analyse plus précise car elle ne permet pas de comparer les langages entre eux (difficile de dicerner qui de Python ou Shell est le plus grand dans le nuage de mots).

Cette représentation ne traitant pas sur les mêmes données (`languages` et non `primaryLanguage`), on remarque que de nouveaux langages apparaissent comme très utilisés : `Shell`, `HTML`, `CSS`, `Makefile`, `Dockerfile`.

Ces langages sont souvent utilisés en support : - `HTML` et `CSS` pour gérer des interfaces graphiques. - `Shell` est le langage d'interpréteur de commande des systèmes d'exploitations Unix et type Unix comme Linux. - `Makefile` est utilisé pour configurer les étapes de compilation notamment pour le langage C et C++

Ces langages sont donc souvent présents dans les dépôts mais ne constituent jamais le cœur des projets et ne sont donc pas représentés à travers l'attribut `primaryLanguage`.

### 3.2.3 Nuage de points pour comparer la popularité des langages

Après avoir examiné la fréquence d'utilisation des principaux langages de programmation dans les dépôts GitHub, nous souhaitons désormais explorer plus en profondeur la popularité de ces langages en fonction de différents indicateurs. Pour ce faire, nous allons analyser la distribution de popularité des langages de programmation en utilisant un nuage de points.

Nous allons traquer trois mesures clés de la popularité des dépôts :

- Le nombre total de `stars`, qui reflète l'appréciation générale de la communauté.
- Le nombre total de `forks`, qui indique l'intérêt des développeurs pour collaborer ou créer des variantes du projet.
- Le nombre total de `watchers`, qui montre le niveau d'intérêt et de suivi des développeurs pour les mises à jour du projet.

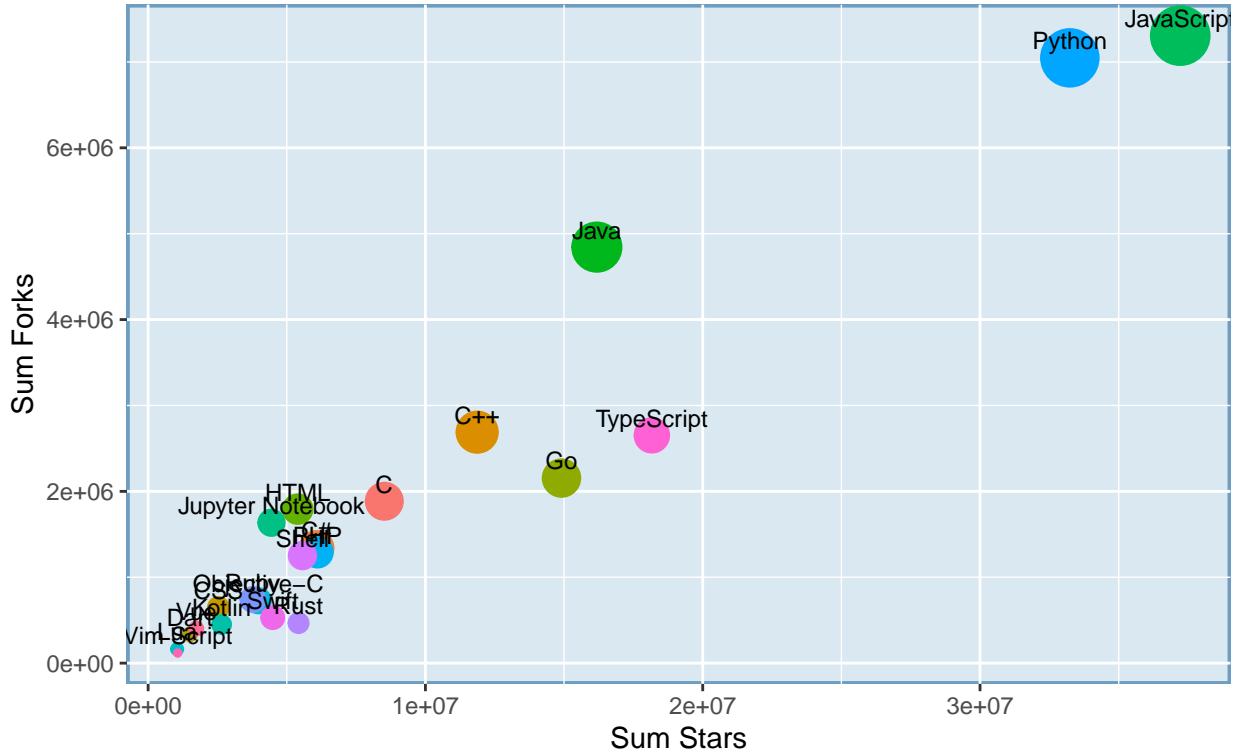
Le nuage de points est un outil efficace pour cette analyse, car il permet de visualiser simultanément ces trois dimensions. L'axe des x représentera le nombre total de `stars`, l'axe des y le nombre total de `forks`, et la taille des points reflètera le nombre total de `watchers`. Cette représentation graphique nous aidera à identifier les tendances et les relations entre ces indicateurs de popularité pour chaque langage de programmation.

*#On regroupe les données en fonction des attributs du langage de programmation et calcule  
# la somme des `stars`, des `forks` et des `watchers` pour chaque langage:*

```
language_pop <- df_filtered %>%
  group_by(primaryLanguage) %>%
  summarise(sum_stars = sum(stars),
            sum_forks = sum(forks),
            sum_watchers = sum(watchers))

ggplot(data = language_pop[language_pop$sum_stars > 1000000,], aes(x = sum_stars,
y = sum_forks, size = sum_watchers)) +
  geom_point(aes(color = primaryLanguage), show.legend = FALSE) +
  geom_text(aes(label = primaryLanguage), size = 3, vjust = -0.5) +
  scale_size_continuous(name = "Watchers", range = c(1, 10)) +
  labs(x = "Sum Stars", y = "Sum Forks", title = "Distribution de la popularité du
langage de programmation GitHub") +
  custom_theme
```

## Distribution de la popularité du langage de programmation GitHub



Nous pouvons diviser tous les langages de programmation en quatre niveaux selon le schéma :

- $T_0$ : Python, JavaScript
- $T_1$ : Java
- $T_2$ : C++, TypeScript, Go, C(peut-être)
- $T_3$ : Les autres langues

La classification en niveaux confirme et enrichit les résultats obtenus avec le diagramme à barres :

- *Consistance des Tendances*: Le diagramme à barres montrait déjà que Python et JavaScript sont les langages les plus fréquemment utilisés. La conclusion obtenue à partir du nuage de points est exactement la même que la conclusion que nous avons obtenue à partir de l'histogramme auparavant. Le nuage de points a confirmé leur suprématie en termes d'étoiles, de forks, et de watchers.
- *Détection des Outliers*: On a observé que, bien que Python soit utilisé plus fréquemment que JavaScript dans les dépôts GitHub, il obtient moins de stars, forks et watchers par rapport à JavaScript. Cela pourrait être dû à la nature plus visible et collaborative des projets JavaScript, ainsi qu'à l'accent mis par la communauté sur les contributions open source. Python, malgré sa fréquence d'utilisation élevée, peut inclure de nombreux projets spécialisés ou utilitaires qui ne génèrent pas autant d'engagement visible.
- *Distribution Équilibrée*: Les niveaux  $T_2$  et  $T_3$  montrent une distribution plus équilibrée sans dominance excessive, ce qui indique une adoption répartie selon les cas d'usage spécifiques des langages.

### 3.3 Tendances dans les langages de programmation grand public

Après avoir vu les résultats ci-dessus, nous ne pouvons nous empêcher de nous demander quand Python et JavaScript sont-ils devenus si populaires? Nous devons découvrir les tendances de ces langages de programmation.

Mais avant cela, nous devons traiter spécialement notre ensemble de données

```
# Convertir createdAt au format de date et extraire l'année
df_filtered <- df_filtered %>%
  mutate(year = lubridate::year(as.Date(createdAt)))

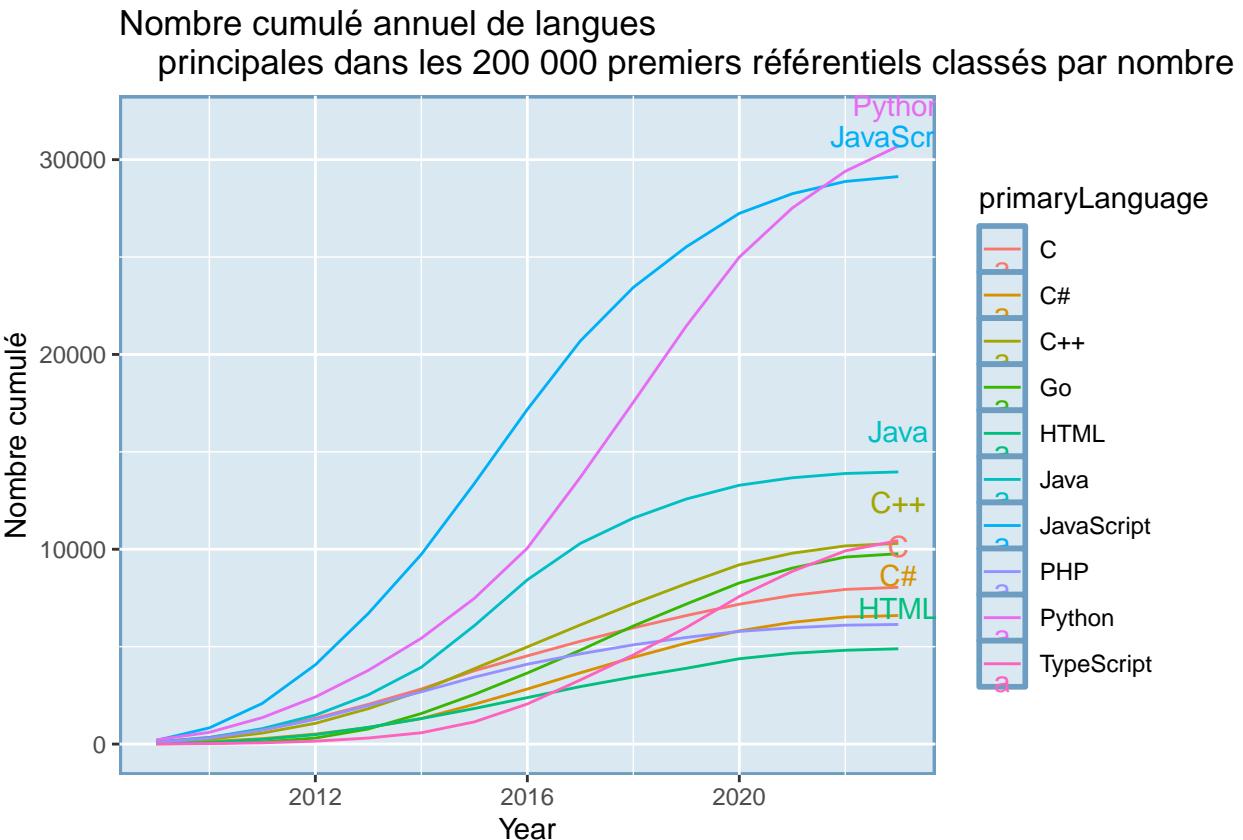
# Regroupés par année et langage de programmation, comptez le nombre d'entrepôts dans chaque langue par
language_count <- df_filtered %>%
  group_by(year, primaryLanguage) %>%
  summarise(count = n()) %>%
  ungroup()

## `summarise()` has grouped output by 'year'. You can override using the
## `.groups` argument.

# Calculer le nombre cumulé de chaque langue par an
language_count <- language_count %>%
  group_by(primaryLanguage) %>%
  mutate(cumulative_count = cumsum(count)) %>%
  ungroup()

# Filtrer les dix principales langues
top_languages <- language_count %>%
  group_by(primaryLanguage) %>%
  summarise(total_count = sum(count)) %>%
  top_n(10, total_count) %>%
  pull(primaryLanguage)

# Dessiner un graphique linéaire
ggplot(language_count %>% filter(primaryLanguage %in% top_languages), aes(x = year,
y = cumulative_count, color = primaryLanguage)) +
  geom_line() +
  geom_text(data = language_count %>% filter(primaryLanguage %in% top_languages) %>%
    group_by(primaryLanguage) %>% filter(year == max(year)), aes(label = primaryLanguage),
    vjust = -0.5, nudge_y = 1000, check_overlap = TRUE) +
  labs(x = "Year", y = "Nombre cumulé", title = "Nombre cumulé annuel de langues
principales dans les 200 000 premiers référentiels classés par nombre d'étoiles") +
  custom_theme
```



Sur la figure, nous pouvons voir que python et JavaScript étaient les premiers langages de nombreux entrepôts au début, mais leur utilisation a considérablement augmenté vers 2016 et est finalement devenue le taux d'utilisation de ces deux langages que nous voyons maintenant.

### 3.4 Analyse des langages de programmation et des métriques des dépôts

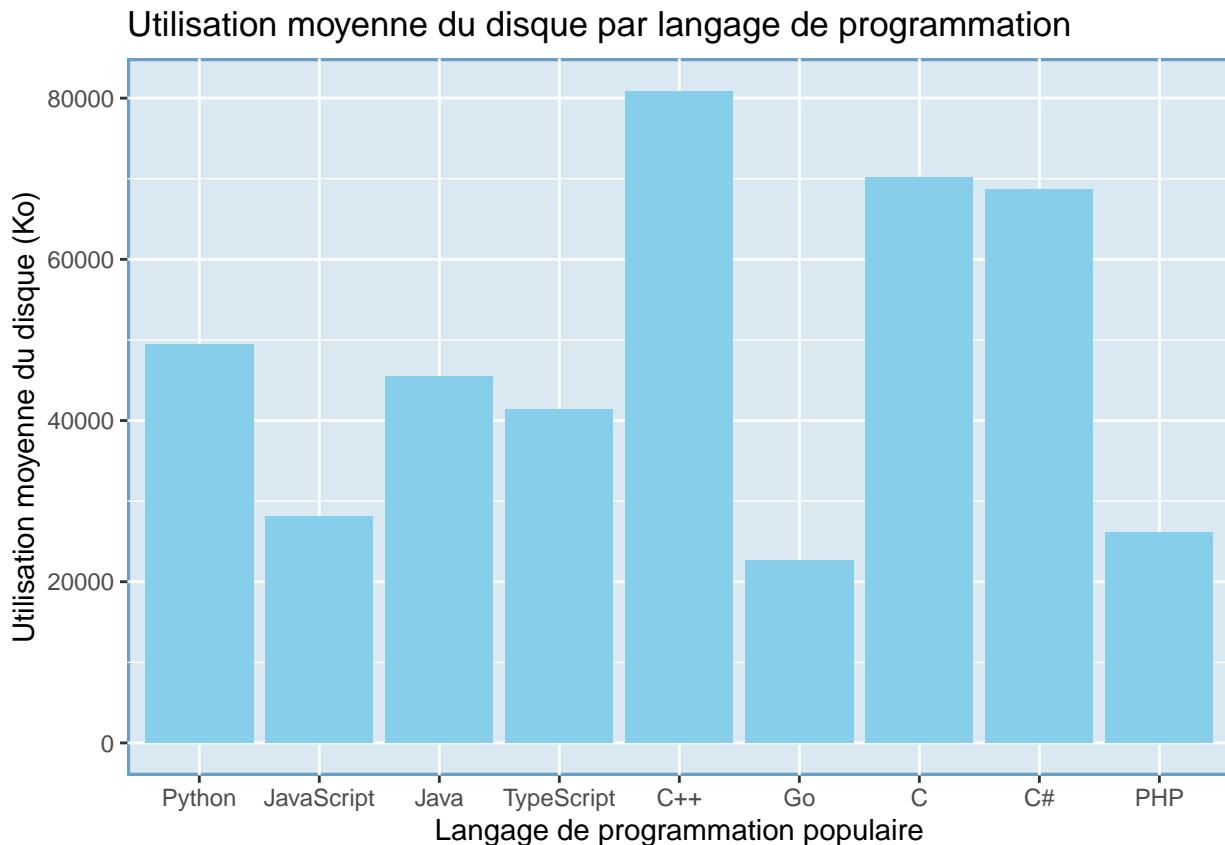
La popularité d'un langage peut être mesurée de diverses manières, telles que le nombre de dépôts utilisant ce langage, le nombre de contributeurs actifs, ou encore le nombre de téléchargements de bibliothèques associées, il est tout aussi important de comprendre comment cette popularité se traduit concrètement dans les caractéristiques et les métriques des dépôts GitHub.

Dans cette étude, nous nous penchons sur la relation entre la popularité des langages de programmation et les métriques des dépôts GitHub. Plus précisément, nous cherchons à comprendre si les langages les plus populaires sont associés à des dépôts de taille et de complexité plus importantes, ou si d'autres facteurs entrent en jeu. Pour ce faire, nous analysons plusieurs métriques clés, notamment l'utilisation moyenne du disque (`diskUsageKb`), le nombre moyen de `pullRequests`, et le nombre moyen d'`issues`, en fonction des langages de programmation utilisés dans les dépôts GitHub. Cette analyse nous permettra de mieux comprendre comment la popularité des langages de programmation se manifeste dans les caractéristiques des dépôts.

#### 3.4.1 Comparaison de l'utilisation du disque par langage de programmation

```
# Calculer l'utilisation moyenne du disque par langage de programmation
utilisation_disque <- df_filtered %>%
  group_by(primaryLanguage) %>%
  summarise(utilisation_moyenne_disque = mean(diskUsageKb, na.rm = TRUE), count = n()) %>%
  arrange(desc(count))
```

```
# Créer un graphique à barres
ggplot(utilisation_disque[utilisation_disque$count > 5000], aes(x =
  reorder(primaryLanguage,-count) , y = utilisation_moyenne_disque)) +
  geom_bar(stat = "identity", fill = "skyblue") +
  labs(title = "Utilisation moyenne du disque par langage de programmation",
       x = "Langage de programmation populaire",
       y = "Utilisation moyenne du disque (Ko)") +
  custom_theme
```



Nous avons constaté que parmi les langages de programmation les plus populaires, C, C++ et C#, langages efficaces mais complexes, utilisent un espace disque moyen plus élevé.

Nous avons les spéculations suivantes sur ce résultat:

- *Nature des projets:* Ces langages sont souvent choisis pour des projets complexes et de grande envergure, tels que les systèmes d'exploitation, les moteurs de jeux et les logiciels d'entreprise, qui nécessitent l'utilisation de nombreuses bibliothèques et de ressources supplémentaires.
- *Exigences de performances et d'efficacité:* Les langages de bas niveau comme C et C++ sont privilégiés pour leur efficacité et leur contrôle direct sur le matériel, ce qui peut entraîner l'utilisation de ressources supplémentaires pour l'optimisation du code et la gestion de la mémoire.
- *Complexité des projets:* La gestion de la mémoire, la manipulation des pointeurs et l'intégration avec des bibliothèques externes peuvent rendre les projets dans ces langages plus complexes, ce qui se traduit par une augmentation de la taille des fichiers et donc de l'utilisation du disque.
- *Priorités de performance:* Dans certains cas, les performances peuvent être prioritaires par rapport à l'optimisation de l'utilisation de l'espace disque, ce qui peut entraîner une augmentation de la taille du binaire final pour garantir une exécution rapide et efficace.

En conclusion, l'observation que les langages de programmation comme C, C++ et C# ont des taux

d'utilisation de disque plus élevés met en lumière la complexité et les exigences spécifiques des projets développés dans ces langages, ainsi que les compromis nécessaires entre performances, efficacité et utilisation des ressources. De plus, de nombreux outils et bibliothèques populaires utilisés dans des langages plus haut niveau comme Python sont eux-mêmes écrits en C, C++ ou C#, ce qui signifie que même les projets développés dans des langages plus accessibles peuvent dépendre de ces langages de bas niveau pour leur fonctionnement interne.

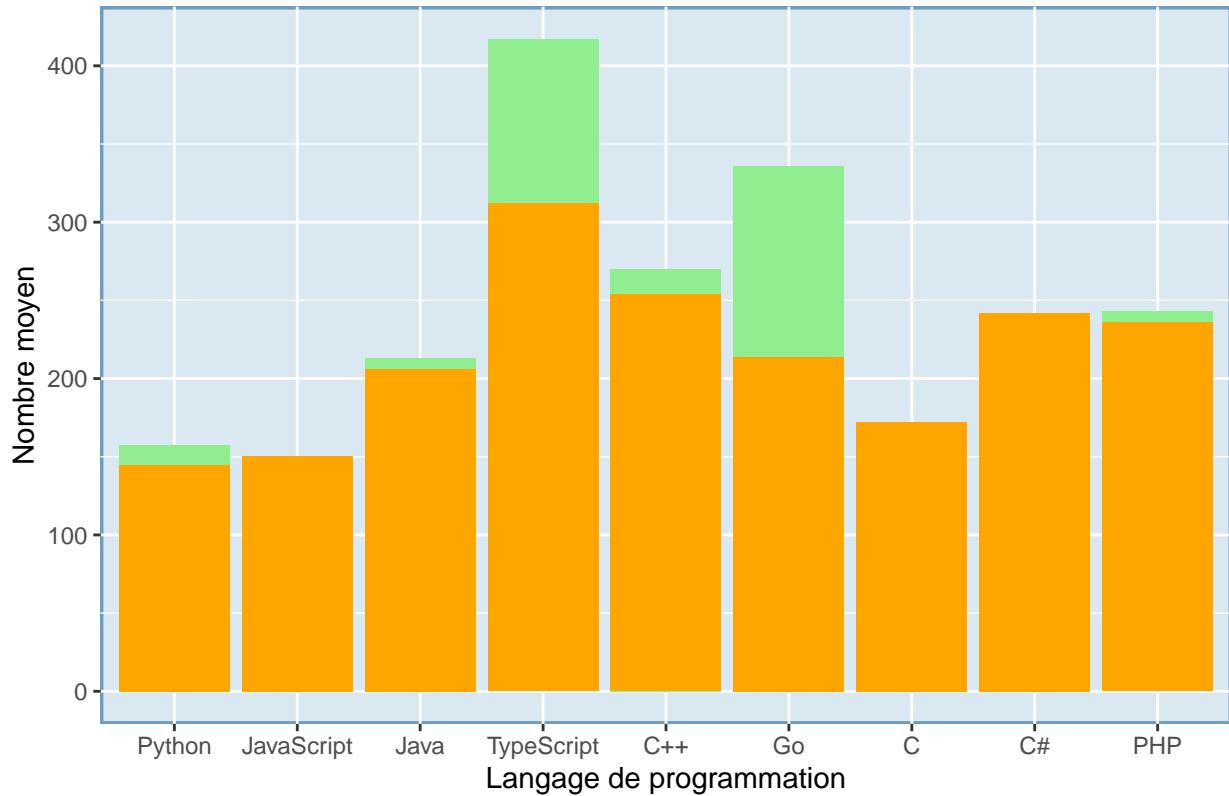
### 3.4.2 Analyse des pull requests et des issues par langage de programmation

Les pull requests et les issues sont des indicateurs cruciaux de l'activité et de la collaboration au sein d'un projet logiciel. Les pull requests représentent les propositions de modification du code source, tandis que les issues correspondent aux problèmes ou aux demandes d'amélioration signalés par les utilisateurs ou les contributeurs. Ces deux métriques sont essentielles pour évaluer la santé et la dynamique d'un projet, ainsi que pour mesurer l'engagement de la communauté de développeurs.

```
# Calculer le nombre moyen de pull requests et d'issues par langage de programmation
pr_issues <- df_filtered %>%
  group_by(primaryLanguage) %>%
  summarise(nombre_moyen_pull_requests = mean(pullRequests, na.rm = TRUE),
            nombre_moyen_issues = mean(issues, na.rm = TRUE), count = n()) %>%
  arrange(desc(count))

# Créer un graphique à barres groupées
ggplot(pr_issues[pr_issues$count>5000,], aes(x = reorder(primaryLanguage,-count))) +
  geom_bar(aes(y = nombre_moyen_pull_requests), fill = "lightgreen", position = "dodge", stat = "identity") +
  geom_bar(aes(y = nombre_moyen_issues), fill = "orange", position = "dodge", stat = "identity") +
  labs(title = "Nombre moyen de pull requests et d'issues par langage de programmation",
       x = "Langage de programmation",
       y = "Nombre moyen") +
  scale_fill_manual(values = c("lightgreen", "orange"), name = "Métrique",
                    labels = c("Pull Requests", "Issues")) +
  custom_theme
```

Nombre moyen de pull requests et d'issues par langage de programmation



La constatation que TypeScript et Go reçoivent le plus grand nombre de pull requests, tandis que TypeScript et C++ ont le plus grand nombre d'issues, souligne les tendances distinctes dans ces langages et leurs communautés respectives. TypeScript, un langage en croissance rapide utilisé principalement dans le développement web, attire un grand nombre de contributions de la part des développeurs, ce qui se traduit par un nombre élevé de pull requests. Cependant, cela peut également conduire à davantage de questions techniques et de problèmes, comme indiqué par le nombre élevé d'issues. D'autre part, bien que C++ soit utilisé dans des domaines variés tels que les jeux et la programmation système, ce qui peut générer un grand nombre d'issues, il reçoit moins de contributions sous forme de pull requests. Cette dichotomie entre les langages reflète les différences dans leurs écosystèmes, leurs applications et l'engagement de leurs communautés respectives dans le développement et la résolution de problèmes.

## 4. Troisième étude : Étude des contributeurs

### 4.1 Analyse des contributeurs

Une autre mesure importante pour évaluer la vitalité d'un dépôt est le nombre de contributeurs. Plus un projet a de contributeurs, plus il est probable qu'il soit actif et qu'il bénéficie d'un développement continu. Nous utilisons un dataset de 1200 lignes, augmenté en utilisant un script et l'API de GitHub. En réalité nous n'ajoutons que le nombre de contributeurs. Cette limite est due aux contraintes d'utilisation de l'API GitHub, notamment le nombre de requêtes par seconde.

### 4.2 Relation entre le nombre de contributeurs et le nombre d'étoiles

```
d1 <- read_csv("../data/githubContrib/data_1.csv")
```

```
## Rows: 300 Columns: 26
```

```

## -- Column specification -----
## Delimiter: ","
## chr (9): owner, name, languages, topics, description, primaryLanguage, lic...
## dbl (11): 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.
d2 <- read_csv("../data/githubContrib/data_2.csv")

## Rows: 300 Columns: 26
## -- Column specification -----
## Delimiter: ","
## chr (9): owner, name, languages, topics, description, primaryLanguage, lic...
## dbl (11): 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.
d3 <- read_csv("../data/githubContrib/data_3.csv")

## Rows: 300 Columns: 26
## -- Column specification -----
## Delimiter: ","
## chr (9): owner, name, languages, topics, description, primaryLanguage, lic...
## dbl (11): 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.
d4 <- read_csv("../data/githubContrib/data_4.csv")

## Rows: 300 Columns: 26
## -- Column specification -----
## Delimiter: ","
## chr (9): owner, name, languages, topics, description, primaryLanguage, lic...
## dbl (11): 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.
df <- bind_rows(d1, d2, d3, d4) %>% filter(!is.na(contributors))

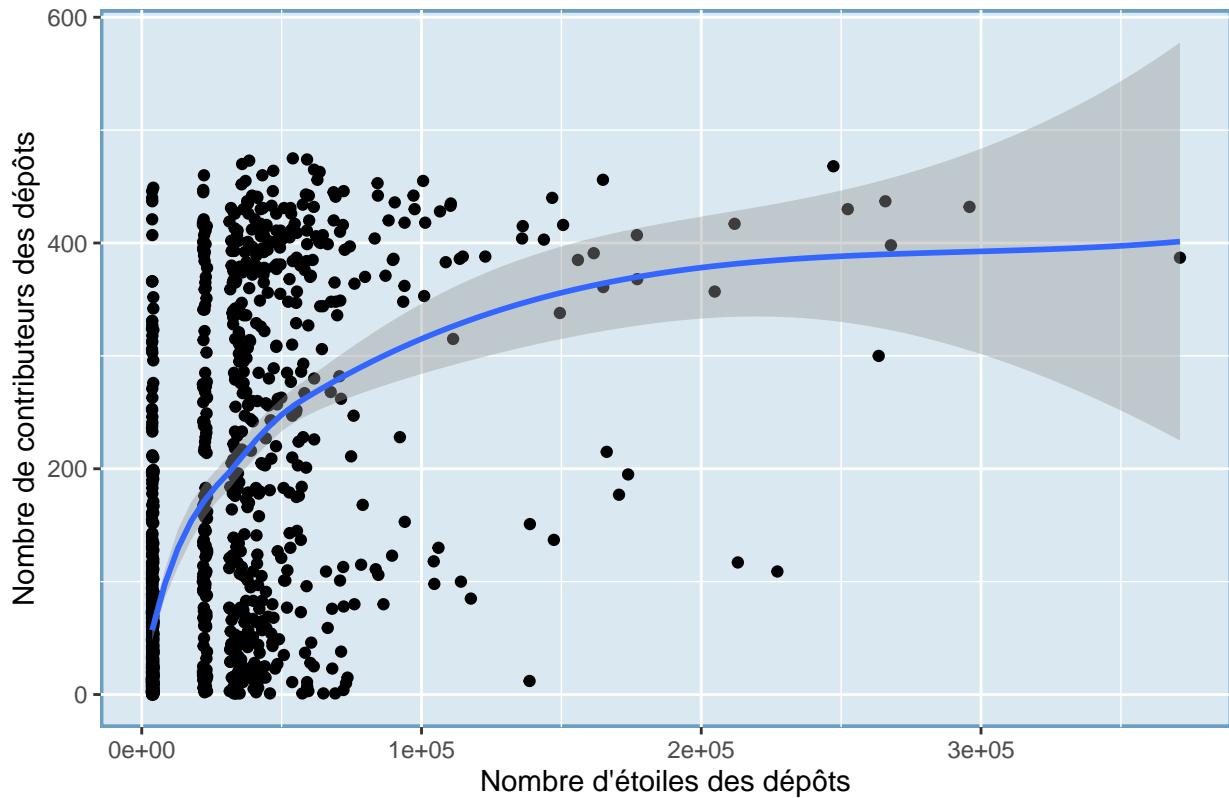
ggplot(df, aes(x = stars, y = contributors)) +
  geom_point() +
  geom_smooth(method = "loess") +
  labs(x = "Nombre d'étoiles des dépôts",
       y = "Nombre de contributeurs des dépôts",
       title = "Relation entre les étoiles et les contributeurs des dépôts") +

```

```
custom_theme
```

```
## `geom_smooth()` using formula = 'y ~ x'
```

### Relation entre les étoiles et les contributeurs des dépôts

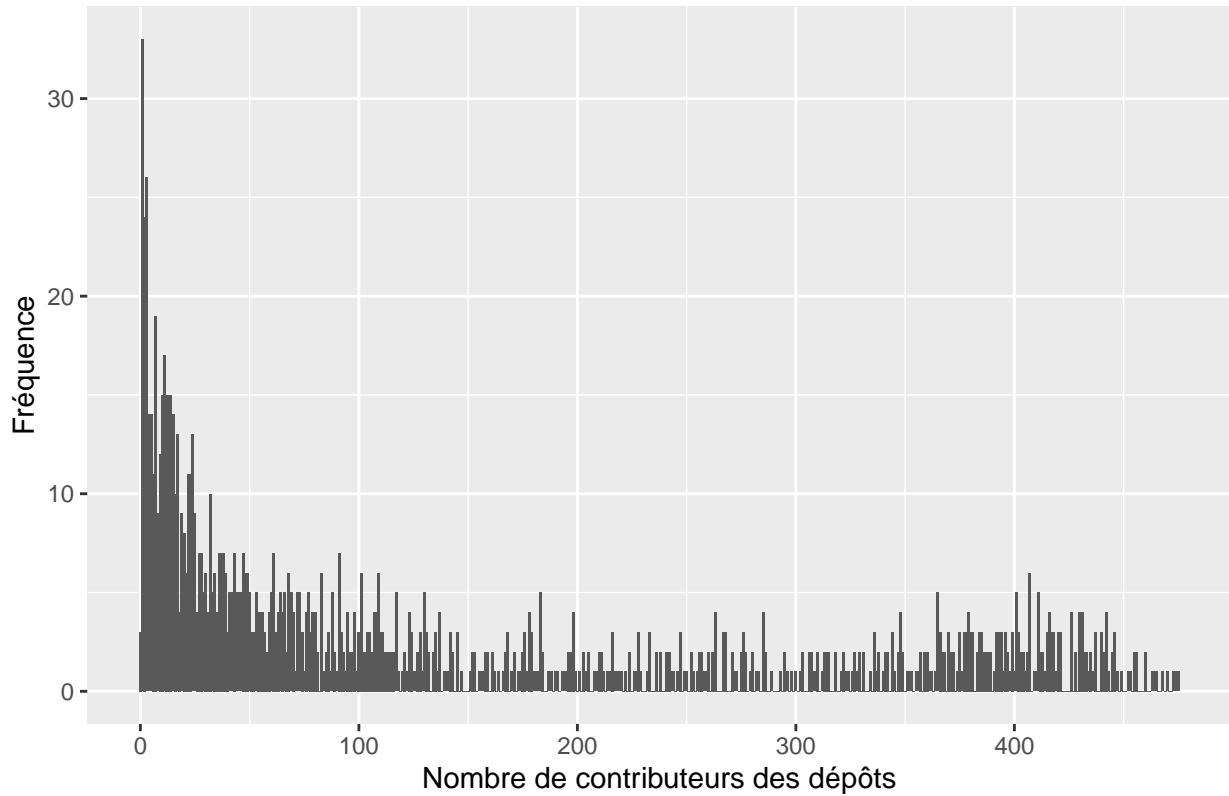


Ce graphique présente la relation entre le nombre d'étoiles et le nombre de contributeurs des dépôts. Nous observons une tendance générale montrant que les dépôts avec un plus grand nombre d'étoiles ont tendance à avoir également plus de contributeurs. Cependant l'augmentation n'est pas linéaire, et les dépôts n'ayant pas encore eu une grande popularité n'ont pas beaucoup de contributeurs. Le nombre de contributeurs semble même se stabiliser. Il semblerait donc, que ce sont les contributeurs qui amènent à une plus grande popularité.

### 4.3 Distribution du nombre de contributeurs des dépôts

```
ggplot(df, aes(x = contributors)) +  
  geom_histogram(binwidth = 1) +  
  labs(x = "Nombre de contributeurs des dépôts",  
       y = "Fréquence",  
       title = "Distribution du nombre de contributeurs des dépôts")
```

## Distribution du nombre de contributeurs des dépôts

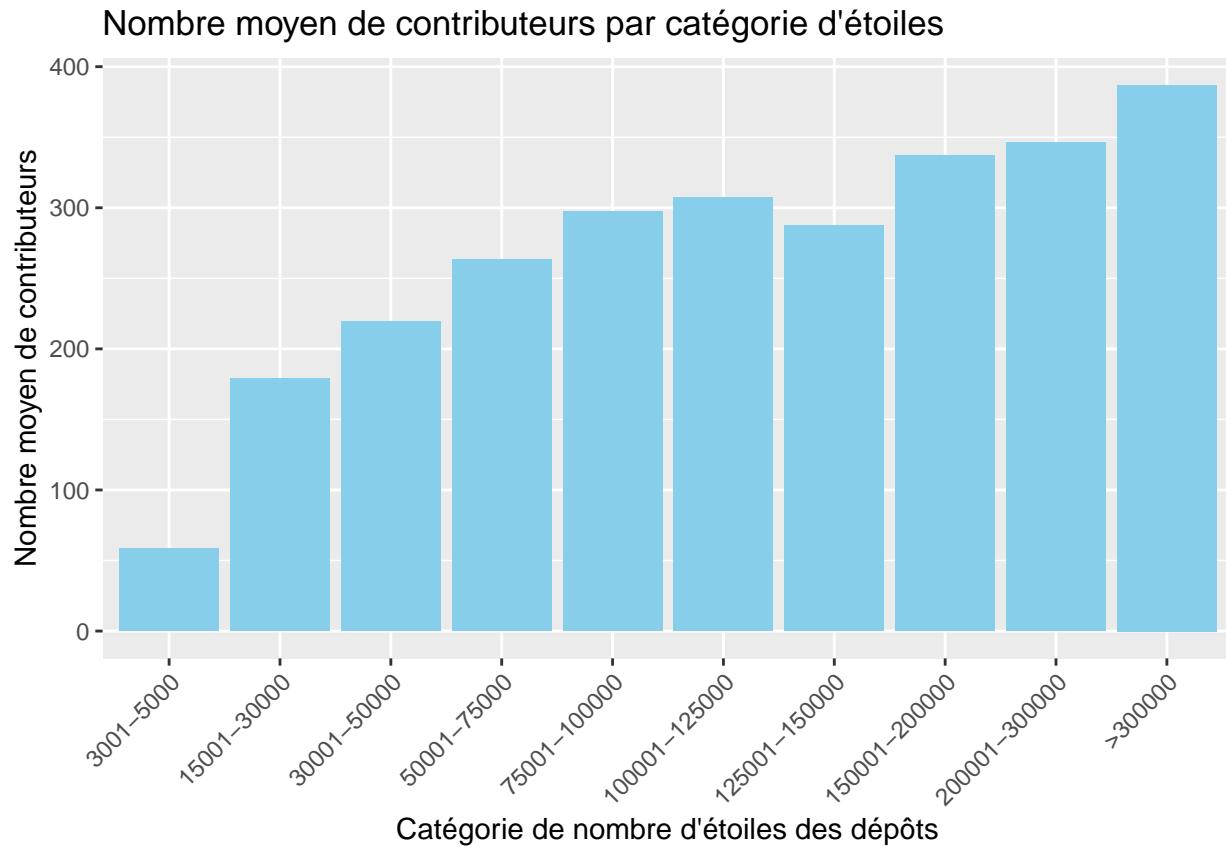


Ce graphique montre la distribution du nombre de contributeurs des dépôts. La plupart des dépôts ont un nombre relativement faible de contributeurs, tandis que quelques-uns ont un nombre élevé de contributeurs.

### 4.4 Nombre moyen de contributeurs par catégorie d'étoiles

```
df_summary <- df %>%
  mutate(stars_category = cut(stars, breaks = c(0, 100, 500, 1000, 2000, 3000, 5000, 10000, 15000, 30000),
                               labels = c("0-100", "101-500", "501-1000", "1001-2000", "2001-3000", "3001-5000")),
  group_by(stars_category) %>%
  summarize(mean_contributors = mean(contributors))

ggplot(df_summary, aes(x = stars_category, y = mean_contributors)) +
  geom_bar(stat = "identity", fill = "skyblue") +
  labs(x = "Catégorie de nombre d'étoiles des dépôts",
       y = "Nombre moyen de contributeurs",
       title = "Nombre moyen de contributeurs par catégorie d'étoiles") +
  theme(axis.text.x = element_text(angle = 45, hjust = 1))
```



Ce graphique montre le nombre moyen de contributeurs par catégorie de nombre d'étoiles des dépôts. Il nous permet de voir comment le nombre moyen de contributeurs varie en fonction du nombre d'étoiles reçues par un dépôt. Les dépôts les plus populaires ont plus de contributeurs mais ce n'est pas proportionnel au nombre d'étoiles. Il semblerait qu'il existe un seuil de popularité à partir duquel le nombre de contributeurs augmente.

## Annexes - Répartition des contributions

### Rapport

Baptiste Toussaint : 1. Introduction 1.1 Description des données 1.1.1 Présentation de GitHub 1.1.2 Les métadonnées de l'API GitHub 1.1.3 Présentation des données 1.1.4 Pourquoi étudier ces données ? 1.2 Plan d'analyse 1.2.1 Objectifs de l'analyses 1.2.2 Mesure de la popularité 1.2.3 Mesure des contributions 1.2.4 Étude des langages de programmation utilisés 1.3 Préparation des données 1.3.1 Charger le jeu de données 1.3.2 Triater les doublons 2. Première étude : mesurer la popularité des dépôts 2.1 Notre interprétation du nombre d'étoile d'un dépôt sur GitHub 2.2 Comment se répartissent le nombre d'étoile sur la plateforme ? 2.3 La date création d'un dépôt influence-t-elle sa popularité ? 2.4 Existe-t-il un lien entre le nombre d'étoile et le nombre de de watchers ? 2.5 Existe-t-il un relation entre le nombre d'étoile et le nombre de forks ? 3.2.2 Alternative : représentation avec un nuage de mots

Shilun XU : 3. Deuxième étude : Étude des langages de programmation utilisés: language 3.1 Notre interprétation du langages de programmation 3.2 Popularité du langage de programmation 3.2.1 Analyse par un histogramme 3.2.3 Nuage de points pour comparer la popularité des langages 3.3 Tendances dans les langages de programmation grand public 3.4 Analyse des langages de programmation et des métriques des dépôts 3.4.1 Comparaison de l'utilisation du disque par langage de programmation 3.4.2 Analyse des pull requests et des issues par langage de programmation

Louis Duhal Berruer: 4. Troisième étude : Étude des contributeurs 4.1 Analyse des contributeurs 4.2 Relation

entre le nombre de contributeurs et le nombre d'étoiles 4.3 Distribution du nombre de contributeurs des dépôts 4.4 Nombre moyen de contributeurs par catégorie d'étoiles

## **Dashboard R Shiny**

Shilun XU: onglet “Répartition des langues” Baptiste Toussaint: le reste

## **Dashboard Tableau**

Shilun XU