UNIVERSITÉ DE LORRAINE

Institut des sciences du Digital Management & Cognition

INSTITUT UNIVERSITAIRE DU NUMÉRIQUE

Masse de données et fouille de données

-

Etude du Covid-19 aux Etats-Unis

-

CHEVRIER Jean-Christophe

LUC Tristan

NOIROT Quentin

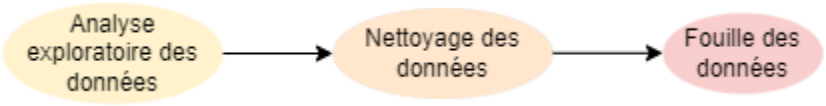
Sommaire

1. Contexte

Dans le cadre de notre formation de Master 2 MIAGE SID, et pour le module de masse de données et de fouille de données, nous avions reçu la mission d'étudier un jeu de données.

Ce jeu de données comportait des **données quotidiennes portant sur l'épidémie de Covid-19 aux Etats-Unis**.

Afin de réaliser une étude approfondie de ce jeu données, nous avons réalisé une courte **pipeline BI**, qui était celle-ci :



Il s'agissait donc d'enchainer ces 3 étapes afin d'arriver au bout à une étude complète.

Nous avons réalisé cette étude avec le langage informatique **Python** dans des scripts sur **Jupyter Notebook**. Pour résumer au mieux notre travail : nous avons réalisé un script pour chaque étape de la pipeline citée précédemment, soit 3 scripts.

Nous avons fait en sorte de bien documenté notre travail, les scripts sont commentés en Markdown, et accessibles via le répertoire GitHub public de notre projet : https://github.com/LordOfRaptor/Study-USA-Covid-19-Data. Les scripts étant dans son répertoire « src/ ».

Ce rapport, dans son plan, suit la pipeline BI que nous avons exécuté, et d'ailleurs, son plan est globalement calqué sur les parties et sous-parties des scripts de notre étude. Nous vous invitons donc à les consulter en parallèle de votre lecture.

La finalité de notre pipeline BI était d'avoir en aval de cette dernière, des modèles de prédiction cohérents de l'évolution des morts de l'épidémie aux Etats-Unis. C 'est la problématique que nous avons choisi : **pouvoir prédire l'évolution des morts de l'épidémie** donc.

2. Analyse exploratoire des données

Nous avons donc commencé notre étude en faisant une analyse exploratoire des données du jeu de données.

D'ailleurs voici le script de l'étape sur GitHub : https://github.com/LordOfRaptor/Study-USA-Covid-19-Data/blob/main/src/script_exploratory_data_analysis.ipynb.

Ce que nous entendons par analyse exploratoire des données, c'est en fait de regarder la **structure** des données, leur **volumétrie**, et surtout leurs problèmes : les **données manquantes et incohérentes**.

Cette étape avait en fait essentiellement pour but de **« diagnostiquer » les problèmes du jeu de données**, qui seraient à colmater dans l'étape suivante de nettoyage.

2.1. Volumétrie des données

En commençant à travailler sur les données, nous avons pu voir leur volumétrie :

```
dataFrame.shape
```

```
(800437, 7)
```

Le jeu de données de base, non nettoyé donc, comporte 7 colonnes, et 800 437 lignes / observations.

L'essentiel à voir ici, est que la volumétrie des données est donc réductible à cet **ordre de grandeur : 100 000, soit 5 zéros** donc.

Cela nous a servi d'indicateur sur la complexité algorithmique qui serait à l'œuvre au moment de faire des opérations lourdes sur les données.

On peut ajouter que dans le monde actuel de la Big Data, des masses de données donc, cela reste une volumétrie correcte, comparée aux échelles existantes de données actuellement.

2.2. Structure des données

Ensuite, c'est bien de connaitre la taille des données mais l'essentiel pour nous était surtout de comprendre sa structure, à savoir ses colonnes, les dépendances entre celles-ci, etc.

```
list(dataFrame.columns)
```

```
['Unnamed: 0', 'date', 'county', 'state', 'fips', 'cases', 'deaths']
```

Ce jeu de données **associe à une date et à un comté d'un état (colonnes "date", "county", "fips" et "state"), un nombre de cas et un nombre de morts du Covid-19 aux Etats-Unis (colonnes "cases", et "deaths")**.

Les comtés des états aux Etats-Unis ont des fips, il s'agit de sorte de code postal : https://fr.wikipedia.org/wiki/Federal_Information_Processing_Standard.

En termes de dépendance, on voit bien que pour un comté, on trouve toujours le même état, et également comté et fips sont des données qui font doublon, elles sont équivalentes.

2.3. Analyse exploratoire introductive

Par la suite, nous avons enchainer sur les étapes de diagnostic des problèmes dans les données. Afin de faire cela, nous avons commencé par une analyse exploratoire introductive, de recherche des valeurs manquantes au sens de cellules vides. Pour faire cela, nous avons utilisé deux fonctions, dont nous devons expliciter le fonctionnement.

La fonction len() appliquée à une colonne retourne le nombre de cellules total de cette dernière, elle n'omet pas les cellules vide. La fonction count() quant à elle renvoie sur une colonne le nombre de cellules non vides, soit le nombre de cellules qui contienne une valeur. La différence de ces fonctions retourne le nombre de données manquantes (données manquantes au sens cellules vides ici, on le rappelle) .

```
len(dataFrame) - dataFrame.count()
```

```
id            0
date        138
county      130
state       134
fips       7694
cases       145
deaths    16876
```

Ainsi, avec cette première analyse introductive, on voyait déjà que toutes les colonnes, hormis la colonne d'identifiant, comportaient des données manquantes. Ce qui n'est pas une bonne nouvelle.

Ensuite, nous avons enchainé sur des analyses exploratoires individuelles de chaque colonne, dont celle de l'identifiant. Etant donné que cette colonne après analyse a été diagnostiquée sans problème, et étant donné qu'elle pas spécialement intéressante dans le cadre du rapport, nous n'évoquerons pas son analyse ici.

2.5. Analyse exploratoire de la colonne « date »

Pour chaque colonne, nous avons effectué des analyses pour déterminer l'existence ou non de données manquantes et incohérentes. Ca a été à chaque fois le même protocole, hormis pour la partie données incohérentes ou à il fallait analysé en fonction à chaque fois de ce que représente la colonne.

La colonne « date » comportait des données manquantes, 138 pour être exact.

```
dateIsNa = pandas.isna(dataFrame["date"])
dataFrame[dateIsNa]["date"]
```

```
18243     NaN
25438     NaN
26345     NaN
28217     NaN
29487     NaN
         ...
757800    NaN
768876    NaN
775081    NaN
785062    NaN
786772    NaN
Name: date, Length: 138, dtype: object
```

```
dateIsNa.sum()
```

```
138
```

Pour l'étude la cohérence de ses valeurs, comme pour les autres colonnes, nous avons utilisé des expressions régulières. Et nous nous sommes rendus compte d'un certain problème : les dates étaient enregistrés sous deux formats différents dans la colonne.

```
dataFrame[dataFrame['date'].str.contains('^[0-9]{4}-[0-9]{2}-[0-9]{2}$', regex = True, na = False)]['date']
```

```
0         2020-01-21
1         2020-01-22
2         2020-01-23
3         2020-01-24
4         2020-01-24
             ...
800432    2020-12-05
800433    2020-12-05
800434    2020-12-05
800435    2020-12-05
800436    2020-12-05
Name: date, Length: 799042, dtype: object
```

```
dataFrame[dataFrame['date'].str.contains('^[0-9]{4}-[0-9]{2}-[0-9]{2}$', regex = True, na = False) == False]['date']
```

```
1008      2020.03.10
2242      2020.03.14
2836      2020.03.16
3774      2020.03.18
4514      2020.03.19
             ...
795819    2020.12.04
796123    2020.12.04
796776    2020.12.04
796794    2020.12.04
799827    2020.12.05
Name: date, Length: 1395, dtype: object
```

```
(dataFrame['date'].count()
- len(dataFrame[dataFrame['date'].str.contains('^[0-9]{4}-[0-9]{2}-[0-9]{2}$', regex = True, na = False)]['date'])
- len(dataFrame[dataFrame['date'].str.contains('^[0-9]{4}\.[0-9]{2}\.[0-9]{2}$', regex = True, na = False)]['date']))
```

```
0
```

Il y avait ces deux formats : « YYYY-MM-DD » et « YYYY.MM.DD ».

La colonne « date » comportait des donc des données manquantes et des données incohérentes.

2.6. Analyse exploratoire de la colonne « county »

Pour la colonne « county », nous avons trouvé deux formes de données manquantes : des données manquantes sous forme de cellules vides (130 lignes), et des données manquantes renseignées explicitement avec le mot clé « Unknown » (6870 lignes).

```
countyIsNa = pandas.isna(dataFrame["county"])
dataFrame[countyIsNa]["county"]
```

```
7877      NaN
9410      NaN
16113     NaN
36581     NaN
40683     NaN
          ...
776385    NaN
783350    NaN
790449    NaN
794963    NaN
800247    NaN
Name: county, Length: 130, dtype: object
```

```
dataFrame[dataFrame["county"]=="Unknown"]["county"]
```

```
418       Unknown
450       Unknown
485       Unknown
522       Unknown
569       Unknown
           ...
799602    Unknown
799803    Unknown
800090    Unknown
800107    Unknown
800281    Unknown
Name: county, Length: 6870, dtype: object
```

En revanche, nous n'avons pas trouvé de données incohérentes, l'ensemble des comtés matchaient avec une expression régulière cohérente avec la manière dont pourrait être nommé un comté, pas de chiffre, de moins devant les noms, etc.

```
dataFrame[dataFrame['county'].str.contains('^[a-zA-Z- \.\'ñ]+$', regex = True, na = False)]['county']
```

```
0          Snohomish
1          Snohomish
2          Snohomish
3               Cook
4          Snohomish
             ...
800432    Sweetwater
800433         Teton
800434         Uinta
800435      Washakie
800436        Weston
Name: county, Length: 800307, dtype: object
```

```
len(dataFrame[dataFrame['county'].str.contains('^[a-zA-Z]+[a-zA-Z- \.\'ñ]+$', regex = True, na = False)]['county'])
```

```
800307
```

```
(dataFrame['county'].count()
  - len(dataFrame[dataFrame['county'].str.contains('^[a-zA-Z- \.\'ñ]+$', regex = True, na = False)]['county']))
```

```
0
```

La colonne « county » comportait donc des valeurs manquantes de deux types, mais ne contenaient pas de valeurs incohérentes.

2.7. Analyse exploratoire de la colonne « state »

La colonne « state » comportait des valeurs manquantes, d'un seul type : des cellules vides (134 lignes pour être exact).

```
stateIsNa = pandas.isna(dataFrame["state"])
dataFrame[stateIsNa]["state"]
```

```
12916     NaN
15191     NaN
19325     NaN
22837     NaN
40051     NaN
          ...
786480    NaN
786545    NaN
787167    NaN
788075    NaN
791545    NaN
```

```
stateIsNa.sum()
```

```
134
```

De même que pour la colonne « county », la colonne « state » ne comportait pas de valeurs incohérentes, ses valeurs matchant avec une expression là aussi cohérente pour le nom qui peut être attendu pour un état.

```
dataFrame[dataFrame['state'].str.contains('^[a-zA-Z- ]+$', regex = True, na = False)]['state']
```

```
0         Washington
1         Washington
2         Washington
3           Illinois
4         Washington
             ...
800432       Wyoming
800433       Wyoming
800434       Wyoming
800435       Wyoming
800436       Wyoming
Name: state, Length: 800303, dtype: object
```

```
(dataFrame['state'].count()
     - len(dataFrame[dataFrame['state'].str.contains('^[a-zA-Z- ]+$', regex = True, na = False)]['state']))
```

```
0
```

La colonne « state » comportait donc uniquement des valeurs manquantes.

2.8. Analyse exploratoire de la colonne « fips »

Par la suite, nous avons enchaîné sur les colonnes avec des valeurs numériques, les colonnes quantitatives : « fips », « cases », et « deaths ». Pour la colonne « fips », nous avons trouvé déjà des valeurs manquantes (7694).

```
fipsIsNa.sum()
```

```
7694
```

Ensuite, l'ensemble des valeurs de la colonne fips se reconnaissait dans l'expression régulière des nombres décimaux négatifs, il y avait donc deux problèmes, un fips d'un comté ne devrait pas pouvoir avoir des décimales, ici .0, et ne devrait pas pouvoir être négatif.

```
dataFrame[dataFrame['fips'].apply(str).str.contains('^-[0-9]+.[0-9]+$', regex = True, na = False)]['fips']
```

```
439       -6097.0
1590     -51059.0
1650      -6095.0
2105     -26115.0
2874     -17089.0
            ...
799381   -40093.0
799496   -42089.0
799503   -42103.0
799586   -72133.0
799862   -48101.0
Name: fips, Length: 1405, dtype: float64
```

```
dataFrame[dataFrame['fips'].apply(str).str.contains('^-?[0-9]+.[0-9]+$', regex = True, na = False)]['fips']
```

	id	date	county	state	fips	cases	deaths
0	0	2020-01-21	Snohomish	Washington	53061.0	1.0	0.0
1	1	2020-01-22	Snohomish	Washington	53061.0	1.0	0.0
2	2	2020-01-23	Snohomish	Washington	53061.0	1.0	0.0
3	3	2020-01-24	Cook	Illinois	17031.0	1.0	0.0
4	4	2020-01-24	Snohomish	Washington	53061.0	1.0	0.0
...	...	...	...	...	...	...	...
800432	800432	2020-12-05	Sweetwater	Wyoming	56037.0	2098.0	10.0
800433	800433	2020-12-05	Teton	Wyoming	56039.0	1739.0	2.0
800434	800434	2020-12-05	Uinta	Wyoming	56041.0	1187.0	5.0
800435	800435	2020-12-05	Washakie	Wyoming	56043.0	519.0	8.0
800436	800436	2020-12-05	Weston	Wyoming	56045.0	419.0	2.0

792743 rows × 7 columns

```
(dataFrame['fips'].count()
  - len(dataFrame[dataFrame['fips'].apply(str).str.contains('^[0-9]+.[0-9]+$', regex = True, na = False)]['fips'])
  - len(dataFrame[dataFrame['fips'].apply(str).str.contains('^-[0-9]+.[0-9]+$', regex = True, na = False)]['fips']))
```

```
0
```

La colonne « fips » contenait donc des valeurs manquantes et des valeurs incohérentes.

2.9. Analyse exploratoire de la colonne « cases »

Pour la colonne « cases », du nombre de cas quotidien, nous avons trouvé exactement les mêmes problèmes que pour la colonne précédente. Des données manquantes sous forme de cellules vides (145), et des données incohérentes : les valeurs contenant des décimales (toutes de type également .0), et étant parfois négatives.

```
casesIsNa.sum()
```

```
145
```

```
dataFrame[dataFrame['cases'].apply(str).str.contains('^-[0-9]+.[0-9]+$', regex = True, na = False)]['cases']
```

```
2614        -1.0
2689        -1.0
2971        -3.0
4454       -42.0
6106        -6.0
           ...
798207   -1911.0
798244    -690.0
798891    -295.0
799369    -626.0
800170   -2622.0
Name: cases, Length: 1295, dtype: float64
```

```
dataFrame[dataFrame['cases'].apply(str).str.contains('^-?[0-9]+.[0-9]+$', regex = True, na = False)]['cases']
```

```
0           1.0
1           1.0
2           1.0
3           1.0
4           1.0
          ...
800432    2098.0
800433    1739.0
800434    1187.0
800435     519.0
800436     419.0
Name: cases, Length: 800292, dtype: float64
```

```
(dataFrame['cases'].count()
  - len(dataFrame[dataFrame['cases'].apply(str).str.contains('^[0-9]+.[0-9]+$', regex = True, na = False)]['cases'])
  - len(dataFrame[dataFrame['cases'].apply(str).str.contains('^-[0-9]+.[0-9]+$', regex = True, na = False)]['cases']))
```

```
0
```

La colonne « cases » contenait donc des valeurs manquantes et des valeurs incohérentes.

2.10. Analyse exploratoire de la colonne « deaths »

Et pour la colonne « deaths », toujours les mêmes problèmes que pour les deux colonnes précédentes, des valeurs manquantes sous formes de cellules vides (16876), et des valeurs incohérentes : les valeurs contenant des décimales (toujours toutes de type .0), et étant parfois négatives.

2.11. Vérification de l'unicité du couple (date, comté)

Une fois que nous avions terminé les analyses exploratoires individuelles des colonnes, nous avons vérifié l'**unicité du couple (date, comté) dans** les données. Pour expliciter cela, tout le jeu de données tient sur le principe que chaque ligne / observation donne le nombre quotidien de cas et de morts pour le couple (date, comté). Et donc, si le couple (date, comté) n'aurait pas vérifié le principe d'unicité, alors cela aurait été un gros problème, car cela aurait signifié que le **cœur sémantique du jeu de données** est erroné.

Pour vérifier le couple (date, comté), nous avons utilisé le couple de colonnes (« date », « fips ») pour éviter le fait que plusieurs comtés puissent avoir le même nom, le fips lui étant unique.

Toujours, pour vérifier le couple, nous avons aussi nettoyer les deux colonnes « date » et « fips » en amont, avant la vérification, pour éviter que les données problématiques ne biaisent la vérification.

```
cleanedDataFrame['date-fips'] = cleanedDataFrame['date'].str.cat("---" + cleanedDataFrame['fips'].apply(str))
```

```
cleanedDataFrame['date-fips']
```

```
0         2020-01-21---53061.0
1         2020-01-22---53061.0
2         2020-01-23---53061.0
3         2020-01-24---17031.0
4         2020-01-24---53061.0
                 ...
800432    2020-12-05---56037.0
800433    2020-12-05---56039.0
800434    2020-12-05---56041.0
800435    2020-12-05---56043.0
800436    2020-12-05---56045.0
Name: date-fips, Length: 775346, dtype: object
```

```
len(cleanedDataFrame['date-fips'].unique())
```

```
775346
```

```
(len(cleanedDataFrame)
 - len(cleanedDataFrame['date-fips'].unique()))
```

```
0
```

L'unicité du coupe a bien pu être vérifiée, comme vous le voyez sur la capture précédente.

3. Nettoyage des données

Une fois que nous avions identifié tous les problèmes des données, avec l'analyse exploratoire des données précédente. Nous avons ensuite entamé les démarches nécessaires, pour nettoyer les données de tous les problèmes de données manquantes et incohérentes identifiés, l'étape de nettoyage des données donc.

Voici le script en lien sur le répertoire GitHub : https://github.com/LordOfRaptor/Study-USA-Covid-19 Data/blob/main/src/script_data_cleaning.ipynb.

3.1. Suppression des lignes avec des valeurs manquantes

La première chose que nous avons fait, a été de supprimer toutes les lignes / observations du jeu de données, pour lesquelles au moins une colonne comportait une donnée manquante (cellules vides ou valeurs « Unknown »).

```
cleanedDataFrame = cleanedDataFrame.dropna()
```

```
len(cleanedDataFrame)
```

```
775346
```

```
cleanedDataFrame.count()
```

```
id        775346
date      775346
county    775346
state     775346
fips      775346
cases     775346
deaths    775346
dtype: int64
```

```
cleanedDataFrame.count() - len(cleanedDataFrame)
```

```
id        0
date      0
county    0
state     0
fips      0
cases     0
deaths    0
dtype: int64
```

```
cleanedDataFrame = cleanedDataFrame.drop(cleanedDataFrame[cleanedDataFrame["county"]=="Unknown"].index)
```

```
len(cleanedDataFrame)
```

```
775346
```

```
cleanedDataFrame[cleanedDataFrame["county"]=="Unknown"]
```

id	date	county	state	fips	cases	deaths

```
len(cleanedDataFrame[cleanedDataFrame["county"]=="Unknown"]["county"])
```

```
0
```

3.2. Conversion au même format pour les dates

Ensuite, nous avons convertit toutes les dates au même format, celui qui était de base le plus utilisé dans la colonne, soit celui-ci : « YYYY-MM-DD ». Nous avons réalisé ça à l'aide du **remplacement par expression régulière**.

```
cleanedDataFrame['date'] = cleanedDataFrame['date'].replace(to_replace = "^([0-9]{4})\.([0-9]{2})\.([0-9]{2})$", value = "\\1-\
```

```
cleanedDataFrame[cleanedDataFrame['date'].str.contains('^[0-9]{4}\.[0-9]{2}\.[0-9]{2}$', regex = True, na = False)]
```

id	date	county	state	fips	cases	deaths

```
len(cleanedDataFrame[cleanedDataFrame['date'].str.contains('^[0-9]{4}\.[0-9]{2}\.[0-9]{2}$', regex = True, na = False)]['date'])
```

```
0
```

3.3. Suppression des "-" dans les colonnes quantitatives

Par la suite, nous nous sommes occupés des signes « - » dans les colonnes quantitatives : « fips », « cases », « deaths », nous les avons supprimés, là aussi via le remplacement par expression régulière.

```
cleanedDataFrame['fips'] = cleanedDataFrame['fips'].apply(str).replace(to_replace = "^-(.+)$", value = "\\1", regex=True)
```

```
cleanedDataFrame[cleanedDataFrame['fips'].apply(str).str.contains('^-.+$', regex = True, na = False)]['fips']
```

```
Series([], Name: fips, dtype: object)
```

```
len(cleanedDataFrame[cleanedDataFrame['fips'].apply(str).str.contains('^-.+$', regex = True, na = False)]['fips'])
```

```
0
```

```
cleanedDataFrame['cases'] = cleanedDataFrame['cases'].apply(str).replace(to_replace = "^-(.+)$", value = "\\1", regex=True)
```

```
cleanedDataFrame[cleanedDataFrame['cases'].apply(str).str.contains('^-.+$', regex = True, na = False)]['cases']
```

```
Series([], Name: cases, dtype: object)
```

```
len(cleanedDataFrame[cleanedDataFrame['cases'].apply(str).str.contains('^-.+$', regex = True, na = False)]['cases'])
```

```
0
```

```
cleanedDataFrame['deaths'] = cleanedDataFrame['deaths'].apply(str).replace(to_replace = "^-(.+)$", value = "\\1", regex=True)
```

```
cleanedDataFrame[cleanedDataFrame['deaths'].apply(str).str.contains('^-.+$', regex = True, na = False)]['deaths']
```

```
Series([], Name: deaths, dtype: object)
```

```
len(cleanedDataFrame[cleanedDataFrame['deaths'].apply(str).str.contains('^-.+$', regex = True, na = False)]['deaths'])
```

```
0
```

3.4. Transformation des colonnes quantitatives en entiers

Et enfin, nous avons retiré les décimales dans les valeurs des colonnes quantitatives, en transformant en entier ces dernières, cela via de simples **opérations de cast**.

```
cleanedDataFrame["fips"] = pandas.to_numeric(cleanedDataFrame["fips"], downcast='integer')
```

```
cleanedDataFrame["fips"]
```

```
0         53061
1         53061
2         53061
3         17031
4         53061
          ...
800432    56037
800433    56039
800434    56041
800435    56043
800436    56045
Name: fips, Length: 775346, dtype: int32
```

```
cleanedDataFrame["cases"] = pandas.to_numeric(cleanedDataFrame["cases"], downcast='integer')
```

```
cleanedDataFrame["cases"]
```

```
0            1
1            1
2            1
3            1
4            1
           ...
800432    2098
800433    1739
800434    1187
800435     519
800436     419
Name: cases, Length: 775346, dtype: int32
```

```
cleanedDataFrame["deaths"] = pandas.to_numeric(cleanedDataFrame["deaths"], downcast='integer')
```

```
cleanedDataFrame["deaths"]
```

```
0          0
1          0
2          0
3          0
4          0
          ..
800432    10
800433     2
800434     5
800435     8
800436     2
Name: deaths, Length: 775346, dtype: int16
```

Une fois toutes ces opérations de transformation faites, nous avons exporté le jeu de données nettoyé, et nous repris à la dernière étape qui suit.

4. Fouille des données

Une fois les données nettoyées, nous pouvions enchainé sur l'étape finale de notre pipeline BI, soit la fouille de données. Et entres autres, nous avons utilisé différentes méthodes de fouille de données pour mettre au point des modèles de prédiction de l'évolution du nombre de morts.

En effet, nous avons choisi de concevoir deux types de modèles, un **premier type de modèle par régression,** et un **second type de modèle par classification**. Vous le verrez par la suite dans cette partie.

Le script de cette étape est celui-ci : https://github.com/LordOfRaptor/Study-USA-Covid-19-Data/blob/main/src/script_data_mining.ipynb.

4.1. Complétion des données

Après importé les données nettoyées, nous avons complété les données. Entre autres, nous avons ajouté le fips de l'état qui n'est rien d'autres que les deux premiers chiffres du fips du comté.

```
completedDataFrame = dataFrame.copy()
completedDataFrame['state_fips'] = (completedDataFrame['fips'] / 1000).apply(int)
completedDataFrame = completedDataFrame.rename(columns = {'fips': 'county_fips'})
completedDataFrame = completedDataFrame[['id', 'date', 'county', 'county_fips', 'state', 'state_fips', 'cases' , 'deaths']]
```

```
completedDataFrame
```

	id	date	county	county_fips	state	state_fips	cases	deaths
0	0	2020-01-21	Snohomish	53061	Washington	53	1	0
1	1	2020-01-22	Snohomish	53061	Washington	53	1	0
2	2	2020-01-23	Snohomish	53061	Washington	53	1	0

4.2. Sélection des données

Ensuite, nous sélectionné les colonnes qui servirait pour les modèles, en d'autres termes, nous avons retenu que les colonnes qui nous semblées les plus pertinentes.

Comme « county_fips » et « county » faisaient doublons et qu'on avait besoin plus de colonnes quantitatives pour les modèles, nous avons retenu que « fips » parmi ces deux colonnes. Et nous en avons fait de même pour colonnes « state_fips » et « state ». ET nous également retiré « id » qui ne nous semblait pas être utile pour nos modèles.

```
reducedDataFrame = completedDataFrame.copy()
reducedDataFrame = reducedDataFrame.drop(columns = ['id', 'county', 'state'])
```

```
reducedDataFrame
```

	date	county_fips	state_fips	cases	deaths
0	2020-01-21	53061	53	1	0
1	2020-01-22	53061	53	1	0
2	2020-01-23	53061	53	1	0

4.3. Première normalisation des données

Par la suite, nous avons effectué une première normalisation. En effet, nous avons transformé la colonne « date » en entier. Et cela, afin de n'avoir plus que des colonnes de type entier.

```
normalizedDataFrame['date'] = (normalizedDataFrame['date'].astype(numpy.int64) / 100000000000).apply(int)
normalizedDataFrame
```

	date	county_fips	state_fips	cases	deaths
0	15795648	53061	53	1	0
1	15796512	53061	53	1	0
2	15797376	53061	53	1	0

4.4. Mélange des données

Une fois ces opérations de base faites, il nous restait une dernière opération basique à faire, à savoir mélanger les données, pour éviter que l'ordre de base des données ne puissent permettre certains biais dans la conception de nos modèles.

```
normalizedDataFrame = shuffle(normalizedDataFrame)
normalizedDataFrame
```

	date	county_fips	state_fips	cases	deaths
17330	15853536	54077	54	1	0
302403	15940800	29019	29	547	2
485589	15991776	28139	28	563	16

4.5. Conception d'un modèle de prédiction du nombre de morts par régression

Une fois les opérations de base terminées, nous avons enchainé sur notre premier type de modèle. Celui par régression. Comme notre modèle tendait à prédire le nombre de morts, nous avions en variables explicatives du modèle : « date », « county_fips », « state_fips » et « cases », et nous avions en unique variable résultat : « deaths ». Ce qui est logique.

4.5.1. Division en deux ensembles : données des prédictions, et du modèle

Tout d'abord, nous avons commencé en divisant les données en d'un côté des données pour faire des tests de prédiction manuels (les 100 premières lignes), et d'un autre côté des données pour le modèle (toutes les autres lignes).

```
deathsModelDataFrame = normalizedDataFrame.copy()
deathsModelDataFrame
```

```
deathsModelForPredictionDataFrame = deathsModelDataFrame.iloc[:100,]
deathsModelForPredictionDataFrame
```

	date	county_fips	state_fips	cases	deaths
17330	15853536	54077	54	1	0
302403	15940800	29019	29	547	2
485589	15991776	28139	28	563	16

```
deathsModelForPredictionDataFrame = deathsModelDataFrame.iloc[:100,]
deathsModelForPredictionDataFrame
```

	date	county_fips	state_fips	cases	deaths
17330	15853536	54077	54	1	0
302403	15940800	29019	29	547	2
485589	15991776	28139	28	563	16

4.5.2. Division en deux ensembles : variables explicatives, et variable résultat

Ensuite nous avons séparé ces deux nouveaux ensembles encore une fois, mais cette fois-ci en ensembles de variables explicatives, et en ensembles de la variable résultat.

```
deathsModelForPredictionEVDataFrame = deathsModelForPredictionDataFrame.iloc[:,:4]
deathsModelForPredictionEVDataFrame
```

	date	county_fips	state_fips	cases
17330	15853536	54077	54	1
302403	15940800	29019	29	547
485589	15991776	28139	28	563

```
deathsModelForPredictionRVDataFrame = deathsModelForPredictionDataFrame.iloc[:,4:]
deathsModelForPredictionRVDataFrame
```

	deaths
17330	0
302403	2
485589	16

```
deathsModelEVDataFrame = deathsModelDataFrame.iloc[:,:4]
deathsModelEVDataFrame
```

	date	county_fips	state_fips	cases
557468	16011648	28157	28	305
64186	15870816	13257	13	30
350889	15954624	17117	17	101

```
deathsModelRVDataFrame = deathsModelDataFrame.iloc[:,4:]
deathsModelRVDataFrame
```

	deaths
557468	18
64186	1
350889	3

4.5.3. Division en deux ensembles : données des entrainements, et des tests

On sépare ensuite une dernière fois, mais cette fois-ci que les ensembles des données du modèle et cela pour faire les ensembles d'entrainement (75% des données) et de test (25% des données).

```
deathsModelEVTrainDataFrame, deathsModelEVTestDataFrame, deathsModelRVTrainDataFrame, deathsModelRVTestDataFrame = train_test_sp
```

4.5.4. Seconde normalisation des données

Et enfin dernière étape avant de faire les modèles, on normalise les données.

```
scalerExplanatoryVariable = preprocessing.StandardScaler()
scalerResultVariable = preprocessing.StandardScaler()
```

```
deathsModelEVTrainDataFrame = pandas.DataFrame(scalerExplanatoryVariable.fit_transform(deathsModelEVTrainDataFrame),
                                               columns = deathsModelEVTrainDataFrame.columns)
deathsModelEVTrainDataFrame
```

	date	county_fips	state_fips	cases
0	0.283146	-0.547085	-0.541434	-0.123538
1	0.079231	0.956910	0.962976	-0.095758
2	0.582220	-0.280913	-0.279797	1.023011

```
deathsModelRVTrainDataFrame = pandas.DataFrame(scalerResultVariable.fit_transform(deathsModelRVTrainDataFrame))
deathsModelRVTrainDataFrame
```

	0
0	-0.101692
1	0.043410
2	0.633491

Etc.

4.5.5. Première configuration de modèle

On fait un premier modèle.

4.5.5.1. Création de l'architecture du modèle, et entrainement du modèle

On fait un modèle notre premier modèle avec une architecture en trois couches d'apprentissage : 32, puis 16, puis 4 neurones.

```
def createDeathsModel():
    input_layer = keras.layers.Input(shape=(4,)) # Entrée.
    h = keras.layers.Dense(32, activation="relu")(input_layer) # Entrainement.
    h = keras.layers.Dense(16, activation="relu")(h)
    h = keras.layers.Dense(4, activation="relu")(h)
    h = keras.layers.Dense(1)(h) # Sortie.
    model = keras.models.Model(inputs=input_layer, outputs=h)
    return model
```

Puis on crée notre modèle, avec un taux d'apprentissage de 1%, avec paquets de taille de 521 lignes/ observations, et devant durer sur 150 époques. Et on lance l'entrainement du modèle.

```
deathsModel = createDeathsModel()
deathsModel.compile(optimizer = keras.optimizers.SGD(learning_rate=1e-2),
                    loss = 'mean_squared_error',
                    metrics = ['accuracy'])
```

```
deathsModelHistory = deathsModel.fit(deathsModelEVTrainDataFrame, deathsModelRVTrainDataFrame,
                                     batch_size = 512,
                                     epochs = 250,
                                     validation_data = (deathsModelEVTestDataFrame, deathsModelRVTestDataFrame))
```

4.5.5.2. Test manuel de la justesse de prédiction du modèle

On effectue un premier test de vérification manuellement avec les données qu'on avait isolé du modèle et en comparant données prédites par le modèle et données réelles.

```
deathsModelPredictions = deathsModel.predict(scalerExplanatoryVariable.transform(deathsModelForPredictionEVDataFrame))
```

```
comparisonResultAndPredictionDataFrame = deathsModelForPredictionRVDataFrame.copy()
```

```
comparisonResultAndPredictionDataFrame['predicted_deaths'] = scalerResultVariable.inverse_transform(deathsModelPredictions)
```

```
print(comparisonResultAndPredictionDataFrame.to_string())
```

```
        deaths  predicted_deaths
17330        0          4.753232
302403       2         16.294720
485589      16         15.594065
691070      38         28.337814
758104       9          6.467901
256496       0         -1.343020
232359       6          2.777506
309578       0         -1.410750
589763      16         24.387894
78162       29         28.076611
517686     120         54.796371
123122       0          3.477075
184168       0         -0.139132
541487       7         10.941565
153836       0          3.987215
648339      10          9.889151
196702       1         10.655466
199972       0          2.743494
488563       5         49.090252
679465      46         56.300140
90959        1          6.145078
```

Les ordres de grandeurs semblent conservés, mais les valeurs exactes ne sont pas retrouvées.

4.5.5.3. Etude du modèle à partir de graphique

Ensuite, on regarde via des graphiques ce qu'a donné l'entrainement du modèle. On peut voir que le loss, la distance entre les valeurs attendues et celles prédites, est moyennement stable, cela rappelle notre conclusion pour le test manuel, les ordres de grandeur sont respectés, mais le modèle manque d'exactitude (loss peu stable).

Pour la suite des modèles, nous vous invitons à suivre le script et ses commentaires bien rédigés, car par limite de temps, nous n'avons pu les développer dans ce rapport.