Master in Statistics for Data Science
Academic year 2020/2021

Final Master Project:

# Development of an automatic tool for periodic surveillance of actuarial and demographic indicators

******

Daniel Alonso

Tutors:

María Luz Durbán Reguera

Bernardo D'Auria

Madrid, Spain, 2021

# Development of an automatic tool for periodic surveillance of actuarial and demographic indicators

Daniel Alonso, María Luz Durbán Reguera, Bernardo D'Auria

July-August 2021

# Contents

# Summary

As a result of the COVID-19 pandemic and its large impact across Spain, the monitoring of demographic measures as a direct result of deaths related to such pandemic and future similarly deadly events has become increasingly important. It is intended with this project to develop a tool to easily monitor a selection of demographic measures. All related to collective deaths of individuals as a result of relevant worldwide events like the one mentioned previously, or perhaps other smaller scale events which could increase mortality in a likely fashion.

The tool consists of an open-source shiny dashboard (developed in R) where such measures are displayed in different visualizations across time. The dashboard is containerized using Docker, in order to make possible to run and use in in any operating system. The data is acquired from the INE and Eurostat APIs. It is then treated by several Python scripts and stored in a GitHub repository where it can be updated by anyone running the application.

# Introduction

The COVID-19 pandemic has led to a widespread and noticeable temporary increase in mortality, as well as to a reduction in life expectancy throughout Spain. This arises a need to monitor these demographic measures more closely and in real time.

This project consists in developing an application which allows for active, real-time monitoring of mortality and life expectancy of the population of Spain, through an open-source shiny web-based dashboard.

The application itself consists of the following functionality:

- Visualizing several mortality metrics:
  - Excess mortality
  - Cumulative mortality rate
  - Cumulative relative mortality rate
  - Mortality improvement factor
- Visualizing life expectancy and constructing life tables
- Visualizing a map of Spain with the previous metrics per autonomous community (CCAA)

All metrics are calculated weekly with data obtained from INE and Eurostat, starting from the year 2010.

# Objectives

The main objectives of the project are the following:

- To provide an open-source, simple-to-use, web-based, OS-agnostic tool to compute and visualize common mortality-related metrics and life expectancy in time series plots, and static or interactive maps.
  - A big focus of the project was to make this application as simple-to-use as possible, while maintaining the management of system requirements extremely low. The purpose of such

  approach is to allow virtually anyone that desires to monitor any of the metrics to do so by running a simple shell command. We want any user or institution with access to a computer and the internet to be able to monitor these metrics, or any such metrics which could be possibly added in the future without much hassle.

  - An important prerequisite was to make the proyect open-source. This allows anyone to freely use and improve the project by adding extra components like other useful mortality-related measures. This makes the project fully auditable and free to use, modify and extend. The project is licensed using the GNU General Public License V3.

- To provide the user with the ability to meaningfully customize the plot parameters.
  - The application itself should have several controls that allow the user to customize visualizations. Some of them are: what years to visualize, what metrics to visualize, what range of weeks to visualize (between 1 and 52), which CCAAs or age groups to aggregate, whether to make the plots interactive (using *plotly*) or static (using *ggplot2*), and so on. Different options will be available for different plots.
- To p3rovide the user the ability to download the plots and the data (with or without filtering).
  - The application shall include a download button with several options for all plots displayed within the application. The user can select the size of the plot from a list of predefined sizes, or specify the image resolution and the format from a list of available ones. We intend to make it so that any plot generated within the app can be downloaded.
- Allow the user to update and upload the data to the corresponding GitHub repository hosting the data, from within the application.
  - The data is constantly changing, and as soon as the data sources have new data available, the application will tell the user there is new data, and the user will be able to update the data with the click of a button. The updated data will be fetched, ran through the pipeline, and uploaded to the data repository for the project on GitHub. This ensures that the data is always updated and anyone is able to do so for everyone else from anywhere in the world.
- Have data updated in real-time from the official Spanish sources (INE) and Eurostat (also provided by INE).
  - Every time the app runs, it'll check if there's available data and display the most recent week for which there is data available, this way we allow anyone running the application to see if there is new data whenever the app is ran so they can update it if they desire to do so.

# Motivation

As we saw during the COVID-19 pandemic, the most widely publicized measures shown to the public in order to explain the status of the pandemic and the country as a whole were always related to incidence of the virus, death counts, recovery counts, amount of patients at ICU, hospitalized patients by COVID-19 vs total hospitalized patients, etc. However, **Death counts do not tell the whole story**, a death count merely tells us that an amount of people died, it does not tell us how much that amount of deaths affect the population. Also, **all these measures are static**, therefore, we cannot see the effect they cause on the population in the long term, which is where measures like the *cumulative relative mortality rate* or *life expectancy* come in, these show the impact of deaths and their long lasting effect relative to the population over time.

As an example, 100 people dying over the course of a week within a municipality of 10,000 inhabitants represents 1% of the population; that same amount of deaths would represent less than 0.01% of the population within a municipality of 1 million inhabitants. Therefore, more robust measures to determine the impact of the deaths caused by a pandemic are needed to really gauge the effect of the pandemic as a whole.

The application allows monitoring some of these measures in real time. Whenever it is desired to fetch new data (if available), the data can be fetched with a click. The measures can be instantly computed with a click and visualized to observe the evolution of the population at any time, by age group, sex and by autonomous community (CCAA) of Spain. Showcasing particular usefulness within events that cause large amounts of deaths (like a pandemic, war, or heat wave) or that reduce the general populations life expectancy (as the application can also visualize the evolution of life expectancy over time).

## Inspiration

The Mortality monitor reports published on a weekly basis by the Institute and Faculty of Actuaries of the UK was the strongest influece. The application essentially offers the same metrics and more. The intention is to improve this already quite good and useful report, make it interactive and customizable, and localize it to Spain. Currently, neither the INE (National Statistics Institute of Spain) or any other Spanish institution offer these metrics along with the rest of the features the app offers (visualizing maps, life expectancy, etc) on a per-week basis.

The COVID-19 vaccine surveillance report: Week 27, published by the PHE (Public Health England) also partly exemplifies a similar idea in its “direct impact on infection and mortality” section.

## Advantages of an open source application over weekly reports in PDF format

There are several advantages an application could potentially have over static, weekly PDF reports like the one previously linked in the Inspiration section:

- The user has control over the visualizations, as there are controls to manipulate the parameters of the visualizations and interact with them.
- The user can download the customized visualizations in their desired resolution or format.
- The user can choose what measures of those available to show.
- As the application is open source, developers could expand the application and add extra features and contribute directly to the project’s development.
- The measures will be as up-to-date as the data source is. As these reports take time to construct and analyze, they will take longer to be released, so the user can simply open the application and update the database whenever they desire to do so.

### Why is monitoring mortality/life expectancy metrics important?

As it is with any demographic measure, knowing general metrics about population is important to many companies, institutions and individuals, to list a few:

- The Government and the Ministry of Health, the main decisionmakers in terms of public health related issues. These two institutions can use these metrics to construct policies or regulations to protect the general health of the population, prioritize or purchase particular medication or medical utensils if needed, etc.
- Insurance companies, those to which customers, companies, governments and other institutions transfer risk in order to protect themselves financially from health-related liabilities or death. Insurance companies directly use life tables to measure risk when providing life insurance to customers, and in order to remain profitable and offer a risk-assessed quality service to their customers these metrics modulate insurance premiums and aid in the process of decisionmaking when it comes to offering a policy to what the company could deem high or low risk customers.
- Ordinary people. Perhaps mere curiosity or desire to be informed, being able to know in a timely manner whether the population is at risk or healthy helps people make choices regarding how to take care of themselves or how to take care of their loved ones.

## Metrics computed by the application

The application's visualizations are especially focused on 3 specific mortality metrics inspired by the regular rolling report by UK's Institute and Faculty of Actuaries:

- Cumulative mortality rate
- Cumulative relative mortality rate
- Mortality improvement factor

Given the host of other possible metrics to be included, which are also related to the ones previously mentioned, we decided to include the following two additional metrics as a way to extend the capabilities of the application:

- Excess mortality
- Life expectancy

## Excess mortality

The term *excess mortality* ($EM$) refers to a measurement which corresponds to the difference between the average deaths which have ocurred during the $n$ years prior to the current $t$ time. Typically, the window of years to compute the average deaths is often $n = 5$. For the application, a window of $n = 5$ years is used to compute the measurement.

If excess mortality is very high and above zero, then there have been more deaths than the previous $n$ year average, if the number is smaller than zero, then there have been less deaths than the previous $n$ year average.

Excess mortality is used to assess how many more or less deaths than the previous years' average have ocurred. This way it's possible to determine if there is an increase in deaths that exceed what has been considered "normal".

During the COVID-19 pandemic the measurement was particularly useful as mortality spiked beyond the expectations of pretty much any country where the pandemic became widespread. Spain is no exception, and during the time where the deaths were at their peak, the excess mortality of that period of time was a highly publicized measure to illustrate how many deaths above what usually occurs happened.

We can see an example of the most generalized $EM$ plot the application can produce:

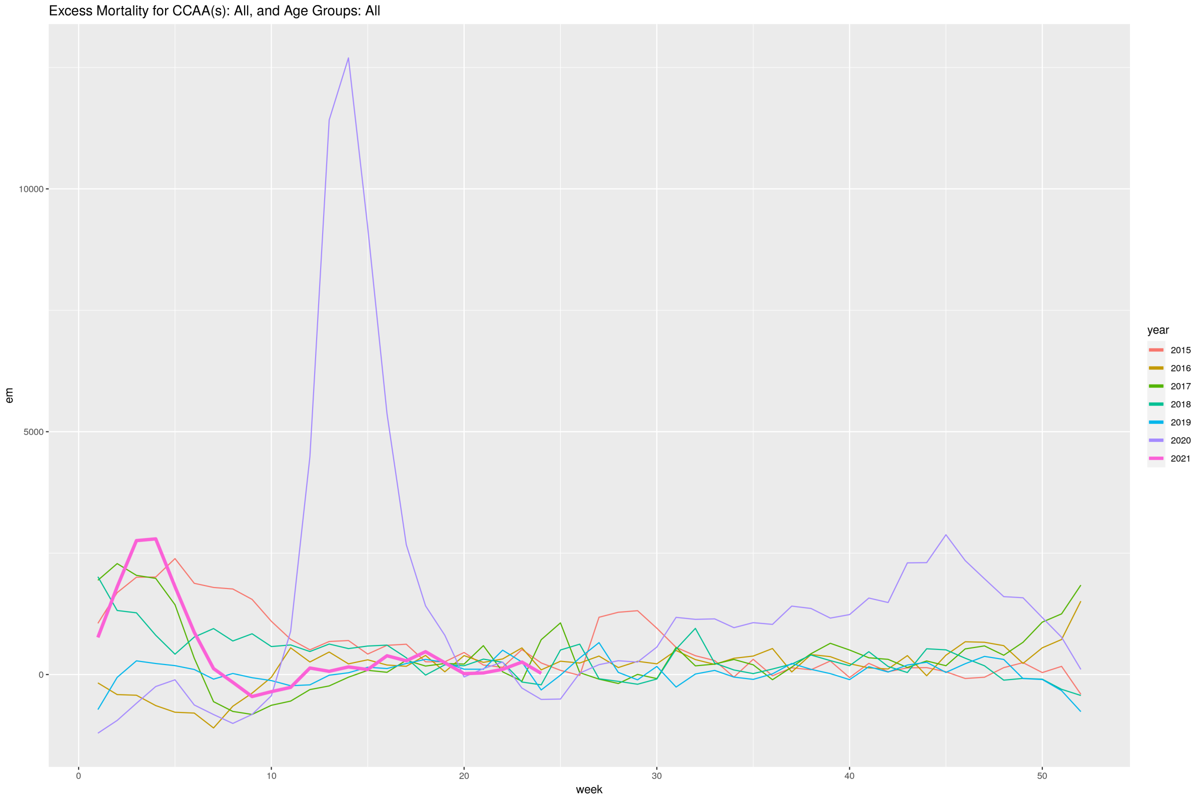

Figure 1: Excess mortality, from 2015 up to week 24 of 2021

In Figure 1, we can clearly notice a pattern where 2020 severely exceeded the mortality expected value of the previous 5 years. Particularly during the 1st wave of COVID-19 (around march, april and may), and towards the end of the year (around october, november and december).

As a result of the COVID-19 pandemic, future measures calculated after 2020 must exclude 2020 and

2021 from the calculation as the death counts are anomalous compared to previous and subsequent years prior the COVID-19 pandemic.

For the years between 2021 and 2027 inclusive, the application uses the same 5-year moving average used for 2020, meaning the average deaths corresponding to week $w$ for $2015 \leq y \leq 2019$. After 2027 or prior to 2021 the average death counts used are computed from the 5 year window prior to the year(s) selected. For example, the average deaths for $w = 1$, 2019 will correspond to the average deaths during $w = 1$ for the years $2014 \leq y \leq 2018$, and the average deaths for $w = 1$, 2022 will correspond to the same average deaths observed on $w = 1$ for the years $2015 \leq y \leq 2019$.

The $EM$ for week $w$ of year $y$ is computed as follows:

$$EM_{w,y} = \frac{D_{w,y}}{\overline{D_{w,y-5,y}}}$$

Where:

- $D_{w,y}$: death count for week $w$ of year $y$
- $\overline{D_{w,y-5,y-1}}$: average death count for week $w$ of years $y - 5$, $y - 4$, ..., $y - 1$.

Exact definition:

$$EM_{w,y} = \frac{D_{w,y}}{\frac{1}{n}\sum_{k=1}^{n} D_{w,y-k}}$$

Where:

- $D_{w,y}$: death count for week $w$ of year $y$
- $n$: years to look back to, default is $n = 5$
- $D_{w,y-k}$: death count for week $w$ of year $y - k$, where $k$ starts at $k = 1$ and ends at $n$ with $k \in \mathbb{N}$.

## Cumulative mortality rate

The *cumulative mortality rate* ($CMR$) represents the ratio between the sum of all deaths during the first week of a year up to a user-defined week of the same year, and the population at that same user-defined week and year, where weeks must be $1 \leq w \leq 52$.

This ratio can be used standalone to gauge the amount of deaths in a time period divided by the population alive at that time period, but, it is most commonly used in this project as a component of the more informational *cumulative relative mortality rate* or *mortality improvement factor*.

The ratio is computed, depending on user selection, for years stretching as far back as 2010 and as updated as the current year. It can only be computed for any specified range between 1 and week $w$.

The $CMR$ for year $y$ and up to week $s$ is computed as follows:

$$CMR_{w,y} = \frac{\sum_{w=1}^{s} D_{w,y}}{P_{w,y}}$$

Where:

- $D_{w,y}$: death count for week $w$ of year $y$.
- $P_{w,y}$: population at week $w$ of year $y$
- $s$: upper bound of week for which the death count is computed, with $1 \leq s \leq 52$ and $s \in \mathbb{N}$.

## Cumulative relative mortality rate

The *cumulative relative mortality rate* ($CRMR$) corresponds to the ratio of the difference between the cumulative mortality rate ($CMR$) at a specified week and the average $CMR$ between years $n-k$ and $n-1$ inclusive, and the average $CMR$ in that same range of years for their last week (the 52nd week).

The $CRMR$ allows us to show the medium to long-term effect of catastrophic or mortality-rising events on the population based on the $CMR$.

The effect of long-lasting events like the COVID-19 pandemic clearly show how mortality can part away from the mean significantly and how the cumulative effect of such mortality can rise to unprecedented levels for relatively long periods of time.

This particular metric can be affected significantly by events which we might perceive as not-so-dramatic when they occur (perhaps like a severe heat wave), but it lets us historically see how much worse or better (in terms of mortality) a given time period can be.

Here is an example plot of the $CRMR$ we can generate for all age groups, sexes and CCAA aggregated:

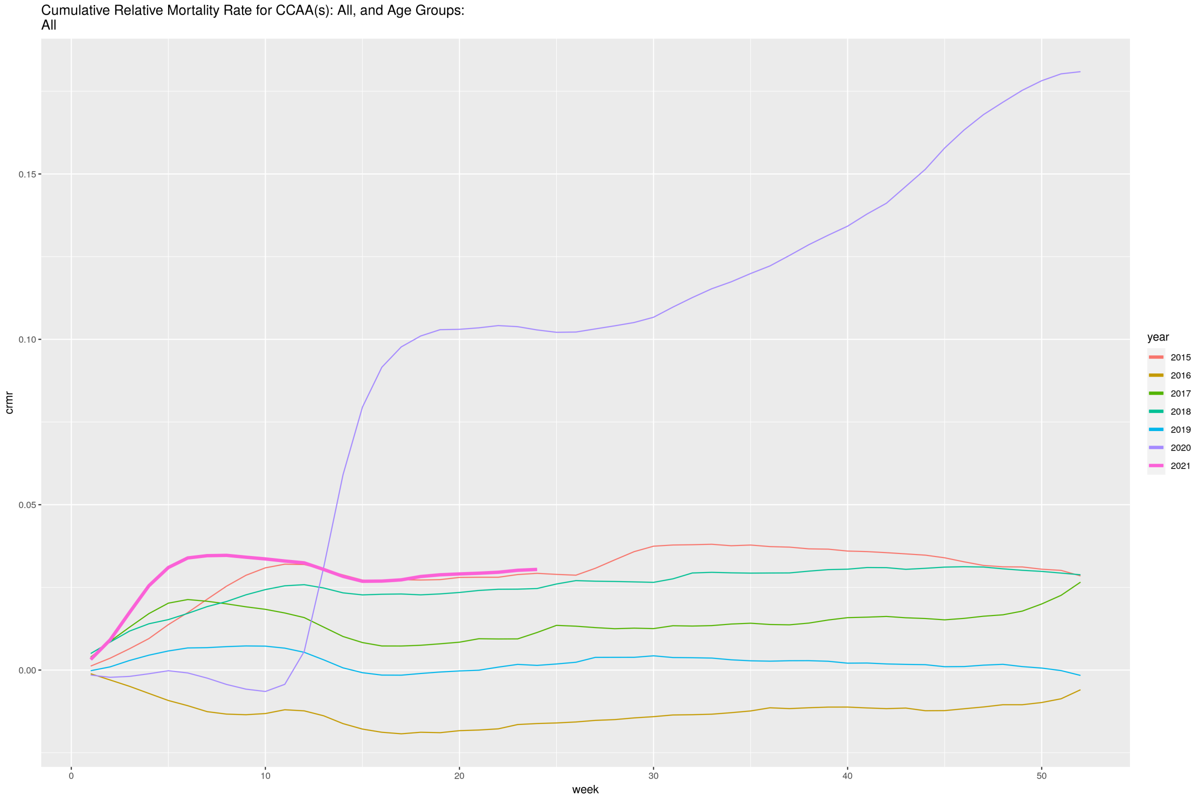

Figure 2: $CRMR$, from 2015 up to week 24 of 2021

As we can notice in Figure 2, 2020 severely exceeded the previous years' $CRMR$ throughout the year right after the pandemic started, and the trend was sustained throughout the whole year. The dramatic increase towards the first wave of the pandemic completely skews the y-axis scale of the plot. We can also notice how the start of 2021 was also out of the ordinary, eventually returning to levels seen around 2015 (a year previously considered particularly bad for mortality in Spain), as the pandemic has continued to extend, but with a significantly smaller $CRMR$ than that of 2020 given the widespread vaccinations and more experienced treatment of the virus.

In our particular case, the metric has been computed using the average $CMR$ for years between 2010 and 2019. 2020 is explicitly excluded, as the effect of the COVID-19 pandemic would skew our perspective on 2021 mortality and further years.

The $CMR$ for the last week of those years is averaged as it includes the entire year, given that the CMR is computed for a range of weeks starting at week 1 and ending at week $s$, for this metric we compute the denominator for week $s = 52$ and the numerator for a specified week range.

The $CRMR$ for week $w$ of year $y$ has been computed as follows:

$$CRMR_{w,y} = \frac{CMR_{w,y} - \frac{1}{10}\sum_{y=2010}^{2019} CMR_{w,y}}{\frac{1}{10}\sum_{y=2010}^{2019} CMR_{w=52,y}}$$

Where:

- $CMR_{w,y}$ is the *cumulative mortality rate* at week $w$ of year $y$

## Cumulative mortality improvement factor

The *cumulative mortality improvement factor* ($CMIF$) at a specific week $w$ of a year $y$ is an actuarial measure defined as the ratio of the difference between the $CMR$ at week $w$ for the previous year $y-1$ and the $CMR$ at week $w$ for the current year $y$, and the $CMR$ at week 52 for the previous year $y-1$.

The $CMIF$ is used to assess how much mortality is improving within specific age ranges, or the population as a whole. By "improvement factor" we mean that it could either get "better" or get "worse", the "better" the $CMIF$ gets, the lower the mortality is, the "worse" it gets, the opposite, meaning mortality will be higher.

This measure is an excellent one when it comes to determining a population's longevity. If the improvement factor is high, then the older the population can become, in some ways, very similarly to life expectancy.

The metric is particularly useful for older age groups, which during the COVID-19 pandemic saw a significant increase in mortality as a result of it being a virus especially dangerous to older age groups and it also being very easily spread throughout populations. In Spain in particular, retirement homes are very commonly used to house retired elderly people. The virus was unfortunately spread among some retirement homes causing very high mortality. These events directly affect the improvement factor negatively. Improvement factors are especially useful in insurance, where they're often used to compute projections for life insurance policies.

As a way to exemplify, we can also show a plot for the $CMIF$, as plotted through the application:

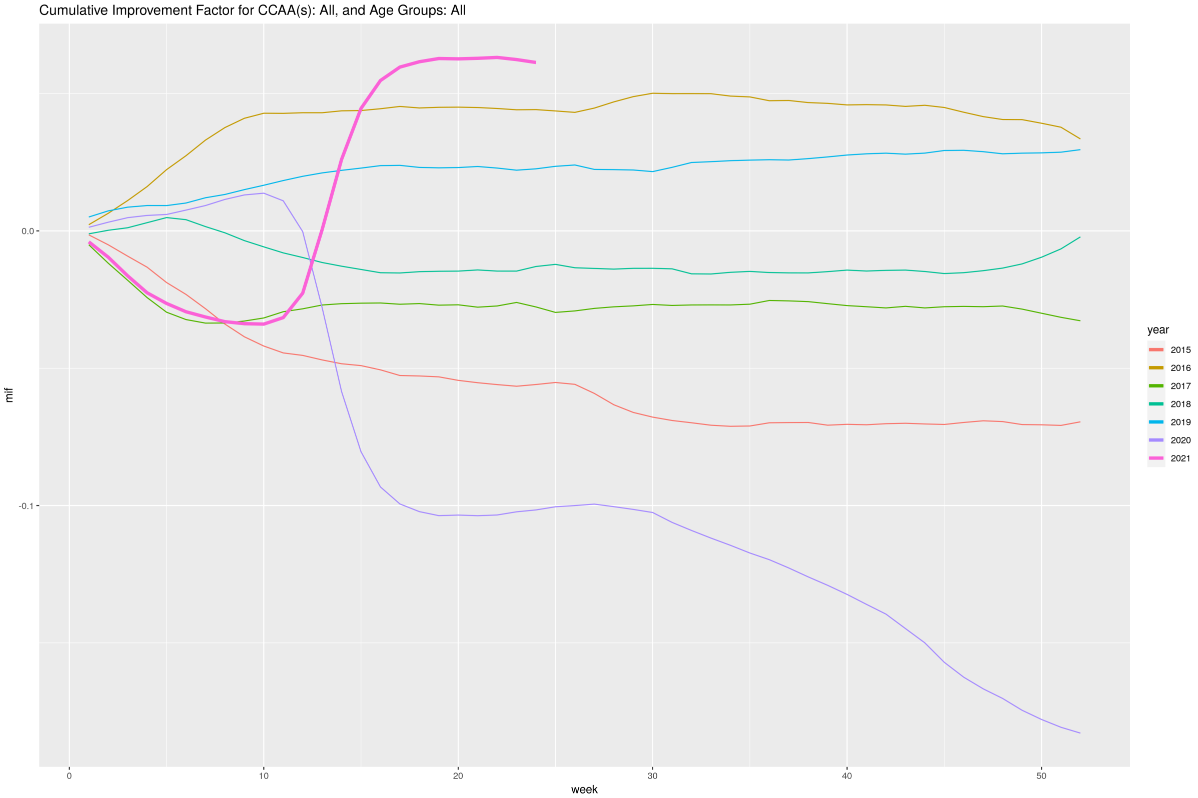

Figure 3: $CMIF$, from 2015 up to week 24 of 2021

As for the $CMIF$ visualized in Figure 3, it is clear how significant the decline of the $CMIF$ is for 2020, and how noticeable of an improvement the widespread vaccination has been during 2021.

Comparatively we can say that rises in $CRMR$ correspond with declines in $CMIF$.

For the application in particular, the $CMIF$ can be computed for years above 2011, as the data only stretches as far back as 2010 and the measure looks back 1 year.

The $CMIF$ for week $w$ of year $y$ is computed as follows:

$$CMIF_{w,y} = \frac{CMR_{w,y-1} - CMR_{w,y}}{CMR_{w=52,y-1}}$$

Where:

- $CMR_{w,y}$ is the *cumulative mortality rate* at week $w$ of year $y$

## Life expectancy

*Life expectancy* ($LE$) is a measure that estimates the time an organism is expected to live. It is computed for a specific age group $A$ using a *life table*, which is a table commonly used in actuarial science, demography and life insurance policy risk modelling that shows the probability that a person of a certain age or age group will die before reaching their next birthday or entering the next age group. The last column of it commonly corresponds with the *life expectancy* for that age or age group based on that probability of death.

$LE$ at birth would correspond to the $LE$ of the youngest age group in the table, assumming we don't have the *infant mortality rate* ($IMR$).

The purpose of this measure is to track how long we should expect our population to live. As an upper limit on how old we should expect the population to become. It is widely used in insurance to calculate insurance premiums on life insurace policies, and to determine if it's even profitable to offer such policies to an individual. A very old individual would be surcharged for a life insurance policy or be rejected altogether when attempting to acquire the policy given their high risk of death and very low life expectancy.

It also gives us a perspective on perhaps how developed a country is, given that most developed countries a have relatively high life expectancy for their population.

For the purposes of this application, it is useful to track the effect mortality has on how long we should expect the Spanish population to live. In particular, mortality as a result of the COVID-19 pandemic has affected life expectancy significantly, however, perhaps not in a permanent way, as mortality eventually goes back to the mean, and with it, so does life expectancy.

As we need to compute $LE$ for specific weeks/years and display them as a time series, we must then calculate a *life table* per week.

The process is largely inspired from the following methodology explained on the MEASURE evaluation website. However, it was adapted to fit weekly measures by taking data one year prior to the specified week. Calculations after the step 2 described below are nearly identical to those displayed on the previously linked article.

To compute $LE$ we must first compute a life table as follows, note that each step also corresponds to a column of the life table in the same order:

**1.** Compute the age-specific death rate ${}_nm_x$:

$$ {}_nm_x = \frac{\sum\ {}_{x,x+n}D_{w,y}}{{}_{x,x+n}P_{w,y}} \tag{1} $$

Where:

- $\sum\ {}_{x,x+n}D_{w,y}$ corresponds to the sum of deaths between week $w$ of the year $y$ and week $w$ of the year $y-1$ for individuals aged between $x$ and $x+n$.

- ${}_{x,x+n}P_{w,y}$ corresponds to the population at week $w$ of the year $y$ for individuals aged between $x$ and $x+n$.

**2.** Compute the proportion of individuals alive at the start of an age interval which are no longer alive at the end of that age interval (they die at some point during the interval), ${}_nq_x$:

a. For non open-ended age groups:

$$ {}_nq_x = 1 - \frac{1}{e^{n*{}_nm_x}} \tag{2.1} $$

b. For open-ended age groups (ex. 90+ year olds):

$$ {}_nq_x = 1 \tag{2.2} $$

Where:

- $n$ is the length of the age interval. INE and Eurostat data is grouped in age interval lengths of $n = 5$.
- $e$ is the exponential function.

**3.** Use the previously calculated measure ${}_nq_x$ to compute $l_x$:

a. For the first value we set $l_0 = 100,000$

b. For the subsequent values, we calculate the next values using the previous one as a sequence:

$$ l_{x+n} = l_x * (1 - {}_nq_x) \tag{3} $$

Where:

- $x$ corresponds to the numbering of the age group, (ex. for *less than 5 year olds* which is our first age group, we have $l_0$, then $l_1$, and so on)

- $x+n$ corresponds to the next age group after $x$

**4.** Compute the number of deaths ocurred within each age interval ${}_nd_x$:

$$ {}_n d_x = l_x * {}_n q_x \tag{4} $$

Where:

- The measure is calculated for individuals aged between $x$ and $x + n$

**5.** Calculate the person-years of life of all individuals in each age interval $x$ to $x + n$, ${}_n L_x$:

$$ {}_n L_x = \frac{{}_n d_x}{{}_n m_x} \tag{5} $$

**6.** We calculate the cumulative person-years of life after age $x$, $T_x$. Therefore, for the first age group in the table, the value of $T_x$ will correspond to the sum of all ${}_n L_x$, the entire 5th column. For the second age group, it'll be the sum of all ${}_n L_x$ with the exception of the ${}_n L_x$ value of the first age group, and so on.

So:

a. For age groups different to the last one:

$$ T_x = \sum_{k=g}^{19} {}_n L_x \tag{6.1} $$

b. For the last age group:

$$ T_x = {}_n L_x \tag{6.2} $$

Where:

- $g$ is the number of the age group for which we want to compute the measure, so for group 1, $g = 1$, and so on.

- The sum goes up to 19, as there are 19 age groups in our computation.

**7.** We compute the *life expectancy* ($LE$) of each age group as follows:

$$ e_x = \frac{T_x}{l_x} \tag{7} $$

# Methodology

## Data querying and updating

The data querying process goes over two APIs, the Eurostat API and the INE API.

### Eurostat API querying

In order to query the Eurostat API, we need to execute several queries using the `query_eurostat` function defined in the functions.py script.

The queries are executed through the query.py script upon user request.

In order to query the Eurostat API:

1. We have to use the Eurostat Query Builder.
2. Pass in the database ID, in our case it's **demo_r_mwk2_05**, we select the parameters and then query the API. The parameters can be concatenated.

The previously mentioned `query_eurostat` function does this, thereby simplifying the process of querying by just passing the parameters to the function as desired.

**Death dataset update pipeline** As the Eurostat API returns a JSON, we must convert this response object into a workable pandas dataframe and eventually a CSV file, for easier storage and updating.

The pipeline followed from when the data is obtained until it's updated the data hosted in the repository is the following, with the exception of the data manipulation pipeline, an intermediate step:

1. Open the update_database.log log file, in which diagnostic messages related to the query are printed temporarily until the query is over.
2. Create an empty list where obtained raw datasets from the API will be added to.
3. Run a nested loop where we iterate over each age group (19 age groups) and sex (3 sexes, male, females and total) and the Eurostat API is queried for each combination of sex and age group. The queried data is queried as far back as the data is provisional (Week 1, 2020 at the time of writing).
4. Each obtained dataset is passed through the `generate_death_df` function defined in the functions.py script, which runs the steps described in the pipeline for the death dataset section. Thereby generating a partial dataset for the current *age+sex* looping combination.
5. The previously generated partial dataset is added to the `death_datasets` list created previously.
6. The datasets within the `death_datasets` list are all concatenated using the `pandas.concat` function.
7. The updated dataset is passed as a parameter of the `perform_update_datasets` function defined in the functions.py script, which in turn checks the currently available dataset within the repository file tree, imports it, updates all the provisional data and appends the newly obtained data if available.
8. The acquired/updated dataset is exported as a CSV file.

### INE API querying

To query the INE API, a single query is executed using the `query_INE_pop` function defined in the functions.py script.

The function executes a single query with data starting at the date defined by the *start* parameter of the function, whose default value is *20100101*, corresponding to the 1st of January, 2010. This single query returns a JSON response with all the data.

In order to query from INE we have to use their data request (in Spanish) methods. In particular, the ones described in the "*Obtención de datos Sistema Tempus3*" section. In our case, we must create a URL like: "https://servicios.ine.es/wstempus/js/ES/DATOS_TABLA/%7Bdf_id%7D?date=%7Bstart%7D:{end}", where *df_id* is the ID of the database (**9681** for us), and *start* being where the time series will start and *end* where it will end. If *end* is left empty, then it'll fetch the most recent data.

**Population dataset update pipeline** The INE API returns a JSON file with somewhat unique formatting, therefore, in order to generate a clean dataset and update the database with the cleaned up dataset, we need to go through a few steps. The intermediate step detailing the cleaning and pre-processing pipeline is detailed in the data pre-processing pipeline section for the population dataset.

1. Gather the latest week updated in the death dataset and assign it to the `most_recent_week` object.

2. Open the update_database.log log file, and log the start of the querying for the dataset.

3. Query the INE database using `query_INE_pop`.

4. Run the data pre-processing pipeline using the function `generate_pop_df` as defined in the functions.py script. The steps for this pipeline are detailed its corresponding section.

5. The updated dataset is passed as a parameter of the `perform_update_datasets` function, which in turn checks the currently available dataset within the repository file tree, imports it, updates all the provisional data and appends the newly obtained data if available.

6. The acquired/updated dataset is exported as a CSV file.

## Data pre-processing pipeline

After the data is acquired, then a pipeline to convert the raw JSON response from the APIs must be ran in order to construct actionable dataframes to store and use for the app. In these two sections there will be a description of all the steps in the pipeline for each of the two datasets.

### Pipeline for the deaths dataset

The biggest challenge that the *deaths* dataset imposed was dealing with the indexing for the JSON response, as it had additional, and admittedly useless or empty data entries within the JSON. Also, the data obtained included a 53rd week for those years that don't fit the same exact number of weeks, this must be dealt with as well.

The function that performs all these tasks is the `generate_death_df` function defined in the functions.py script.

The pipeline steps are the following:

1. We define a `fields` dictionary to obtain all the relevant elements we want from the response JSON object. Each one of them corresponds to either the `keys` or `values` of the object, as JSONs are interpreted as *dictionaries* in python, which makes them easier to work with.
   - Here we exclude weeks marked as *W99* by Eurostat, which are basically empty values.
2. The amount of values in the raw data is obtained from the length of the `values` key in the `fields` dictionary. The amount of "region codes" is obtained from the length of the `region_codes` key in the `fields` dictionary, which are the CCAAs (autonomous communities of Spain). These two values are used to compute the amount of *weeks* obtained from the raw data, the result of this ratio is allocated in the `weeks_queried` object.
3. The `timeframes` key of the `fields` object that keeps track of the week label for each data point is cut up to the present date, as the current week always includes empty data. This cutoff is a way to keep the dataset with the same amount of data points and make it a multiple of the amount of CCAAs queried.
4. A `df` dictionary is defined with its keys being the names of the columns of the final dataset to be outputted and its values being the values which the dataframe will contain with the exception of the *year_week* key which will later on be split into *week* and *year* columns.
   - The *year_week* column contains the values for the weeks times the amount of CCAAs queried, in our case, there will be 19 CCAAs queried, given that Spain has 19 CCAAs at the time of writing.
5. The `df` dictionary is then converted into a pandas DataFrame and the *year* and *week* columns are generated from splitting the *year_week* column.
6. Data points marked as 53rd week are marked with a '0' to be removed and are then subsequently removed.
7. The *year_week* column is dropped and the resulting, cleaned up DataFrame is returned.

### Pipeline for the population dataset

The population dataset response JSON came with challenges:

- The raw data had the full names of the CCAAs, so in order to keep the Eurostat standard, as it is shorter and would occupy less space in the repo, they were replaced.
- The sex also needed to be replaced, as it was in Spanish and it didn't go in line with the deaths dataset.
- The data is gathered on a bi-yearly basis, therefore, a linear interpolation between the two data points that INE records (in January and June) had to be applied to have weekly data. The linear interpolation had to also look ahead about 6 months, given that the data recorded by INE in either June or January takes about a month to be published. The deaths data is often much more up to date, as it is already recorded on a weekly basis, which sometimes causes the deaths data to go ahead of the population data.
- The sheer size of the obtained JSON and its somewhat odd formatting compared to the previous

one obtained from Eurostat requires a large inefficient loop to process. This looping was done using *numpy* arrays, which are often much faster than *pandas* DataFrames, this introduced a few new challenges in order to keep the updating script much more efficient.

- The data came for individual ages, therefore, to match and be able to perform the computation of the metrics previously mentioned, grouping the people within the same specific age groups present in the previous dataset is needed.

All the steps are performed within the `generate_pop_df` function defined in the functions.py script. The pipeline steps are the following:

1. Define several dictionaries with reference data and replacements to align the formatting with the deaths dataset. The dictionaries defined are:
    - `df`, which is going to eventually become the final population dataset. It contains as keys the names of the columns of the final population dataset and as values empty lists.
    - `date_ref` is a dictionary that includes the week as obtained from the API, which originally labels the January data point as 26 and the June data point as 27. These are matched in the object with roughly equivalent dates, Jan. 1st for 26 and Jul. 1st for 27. The conversion performed by this object remains only if the parameter **date** is **True**.
    - `ccaa_eurostat_replace_dict` is a dictionary that will serve as a reference for replacing the CCAA full names with the shorter Eurostat standard, which consists of the country's ISO 3166-1 alpha-2 code (*ES* for Spain) and a number per NUTS 2 statistical region (ex. *ES11* = *Galicia*).
    - `sex_dict` defines how the Spanish names for the sexes are determined, so longer, full names corresponding to each sex are replaced by the Eurostat standard, M (males), F (females) and T (total).
    - `week_dict` is a dictionary with the month (1 and 7) as keys and the corresponding week of the year (1 and 26) for which the data will be matched. This object is similar to `date_ref`, however, the conversion made by `date_ref` is dependent on the parameter **date** of the function, which is not the case with `week_dict`.
2. A loop goes through the raw data points and performs several actions inside of it:
    - Data points for the country's total are omitted given that we intend to perform these aggregations from within the Shiny App. These would also occupy unnecessary space in the repository.
    - The metadata elements from each entry's header which contain sex, age and CCAA are split and passed onto the previously defined dictionaries. This replaces each entry's basic identifying information with the Eurostat standard.
    - The rest of the data is acquired, appended to lists and a date column is generated.
3. The `df` dictionary is converted to a pandas DataFrame for easier manipulation. Then an empty *age_group* column is created, which later on will replace the *age* column, as to classify individuals by age group using the Eurostat format from the deaths dataset.

4. After assigning all individuals of all ages an age group, the *age* column is dropped, and the *age_group* column is renamed to *age*, to match the column names of the deaths dataset.

5. The dataset is grouped by all identifying columns (CCAA, sex, date and age) and applied the `sum()` method on the grouping. The purpose of this is to sum individuals of specific ages as to have the sum of each age group instead of multiple instances of each age group per set of the identifying columns.

6. The *year* and *week* columns are obtained from the *date* column. And subsequently, a *marker* column is created as a way to identify entries at a later step. The *marker* column is created by concatenating the *CCAA*, *sex* and *age* columns, which are the most important columns to identify entries.

7. The unique markers are obtained from the *marker* column. These are then used to obtain the amount of unique years in the dataset, the purpose of this is to determine whether we need to interpolate one year ahead or not. This means that the last data point obtained from the INE API was a week 26 data point, which would require us to look ahead until week 26 of the following year, if the data point was a week 1 data point, we would only need to look ahead within the current year.

8. The `idx` dictionary is defined to keep track of the indexes of the *pop* and *week* columns, which are referenced repeatedly in the following loop. After defining `idx`, then an empty array `df_array` of the expected size of the resulting dataframe is defined. The array `df_array` is an array which will eventually contain the contents of the resulting dataframe, it's size is always slightly overestimated, but this won't matter, as empty rows will be dropped.

9. A long loop is initiated iterating between 0 and the length of the `umarkers` object, meaning the total amount of possible *CCAA*, *age* and *sex* combinations. The loop performs the following actions:

   - The loop goes through every unique marker, obtains an array for the elements of the DataFrame matching that marker in their *marker* column.
   - An empty row is added to the previous array.
   - The last row is essentially a clone of the previous one, but with the date modified. This is done to later on be able to interpolate and add population up to a year after the previously obtained data point. If the data point is at week 26, a full year is added, otherwise, the next one remains in the same year but with the month changed to match week 26.
   - The last element's population is estimated using the ratio of the previous two as a way to keep the population consistent with the previously observed trend.

10. A second, nested loop is initiated, this loop goes through the entries of the array defined previously and performs the interpolation betweeen each pair of points. The steps the loop takes to perform the interpolation are the following:

    - A 2-row array is obtained with the 2 points between which to interpolate. Then the starting week is obtained from these two and a new array `new_p` is created with size corresponding to the amount of weeks between the two data points previously selected. Usually, the size of this will be 26 weeks, as there are 26 weeks between week 1 and week 26 (inclusive).

- This array is filled with linearly spaced values between the initial and ending population. For periods starting in week 26, the amount of values obtained is of 25, given that week 26 is already included and we want to avoid repetition.

- The population column of the previously created array `new_p` is replaced with the interpolation. Then the values of the array which shall constitute the finalized dataframe (`df_array`) are filled with the prediction and the rest of the data.

11. After the nested loop, the last value is saved to the `last_val` variable. The ending week is saved to the `week_end` variable. The interpolation ranges for the future population prediction (the one up to one year after the last data point) is estimated using the ratio between the population at the starting week of the period in question and the population at the subsequent weeks.

12. If the last week of the last data point obtained from INE is a 1st week, then the weeks added forward span the time between week 1 and week 26 of that same year, if it's a 26th week, then the weeks added forward span the time between week 26 and week 52 of that same year.

13. The estimation is then added to the array at its corresponding position. Then the array is converted into a DataFrame, the columns are renamed with the previously defined names. The marker column is dropped. Empty values are removed (as we had previously overestimated the size of the DataFrame).

14. Only data up until the latest week in the deaths dataset is kept. Empty values are removed from the DataFrame, and the columns are finally reordered prior to being outputted in order to match the column order of the deaths dataset.

### Dataset update methodology after pre-processing

After the data is ran through the pre-processing pipeline, the `perform_update_datasets` is ran when performing an update. This function goes through several steps in order to avoid replacing the entire datasets with each update. The steps are as follows:

1. First the data is read from the repository.

2. A marker column is created to identify entries in both the current dataset within the repository tree and the fully updated dataset (the one obtained and treated by the previous two functions `generate_death_df` and `generate_pop_df` in the pre-processing steps).

3. A `to_add` object is created filtering the elements in the updated dataset that are not in the current dataset, the new elements.

4. Only elements in the current dataset are kept in the updated dataset, this way we define what is going to be updated.

5. A `to_keep` object is created filtering only elements in the current dataset that are not in the modified `updated_dataset` DataFrame, these are elements that are no longer provisional or new and will be kept in the database.

6. A `to_update` object is created with only the ielements in the current dataset that are in the modified `updated dataset`, these are values that are still considered provisional and could carry

some updates (perhaps mode deaths or births being registered in the provisional period). These are then subsequently updated.

7. All three datasets are concatenated, the marker column is dropped and the dataset is returned with a clean index.

## Data hosting

The data is hosted in both the dreth/tfm_uc3m repository and the dreth/tfm_uc3m_data repository, however, the newest data, and the one that is pulled every time the app is launched from docker (to check for changes) is contained in the dreth/tfm_uc3m_data repository.

Every time a user performs a database update through the docker container, the application will push the changes to the dreth/tfm_uc3m_data repository.

# Appendix

This project has a very large amount of code, therefore only some code chunks from the Python, R and Docker parts will be shown in this appendix. All the code shown in the appendix is from what has been written up to August 2021. Any future extra additions to the application will be in the displayed in the project's repository.

The last appendix will show the project repository tree structure, with an explanation of each file in the main repository's file tree, their purpose and what they contain.

## Appendix A: Python

The first component of the project is the data. All the data querying and pre-processing parts of the project are written in python. There's several functions that do different things. In this section, each one will be explained and linked to the specific lines of code on github that define them.

There are 3 python scripts in this project, these are explained in detail along with all the files in the repository in the Appendix D:

- **dbs_check.py**: The script imports the `check_eurostat_provisional` and `check_INE_latest` functions from the **functions.py** script. After doing so, it runs them and writes the information obtained to the log files contained in the logs folder and prints a diagnostic message.

Here we can see the entire script as it is quite short:

```python
# %% IMPORTING FUNCTIONS TO USE
from functions import check_eurostat_provisional, check_INE_latest

# %% RUN SCRIPT AND UPDATE LOG FILES
print('\nCalculating database information...\n')

# obtaining eurostat info
eurostat_dates = check_eurostat_provisional(earliest='both')

# logging dates
with open('./logs/last_eurostat_update.log', 'w+') as f:
    latest_date = eurostat_dates[1]
    f.write(f'Last date obtainable from Eurostat DB: {latest_date[0:4]}, week: {latest_date[-2:]}')

with open('./logs/last_ine_update.log', 'w+') as f:
    latest_date = check_INE_latest()
    f.write(f'Last date obtainable from INE DB: {latest_date[0:4]}, week: {latest_date[-2:]}')

with open('./logs/earliest_eurostat_provisional.log', 'w+') as f:
    earliest_date = eurostat_dates[0]
    f.write(f'Earliest provisional date from Eurostat DB: {earliest_date[0:4]}, week: {earliest_date[-2:]}')

# done (:
print('\nDatabase information obtained, Check DB info section for more information\n')
```

- **functions.py**

- **query.py**

## Appendix B: R

## Appendix C: Docker

## Appendix D: Project repository tree structure

The project consists of two repositories and their respectives **subfolders**, **files** and **databases**:

### The main project repository (*dreth/tfm_uc3m*)

- **api**: Contains all files related to the querying, acquisition, manipulation of new data from the two aggregated data sources (INE and Eurostat), along with various log files used to keep track of some metadata from the data sources (last updated data and provisional data).
  - **logs**: This folder contains log files used in the DB info section of the application to display diagnostic info about the databases.
    - **earliest_eurostat_provisional.log**: Eurostat marks data as provisional when it can still be updated, as new deaths might occur. This file logs the earliest data entry marked as provisional.
    - **last_eurostat_update.log**: This file logs the last week for which there is available and updated data from Eurostat for Spain within the database *demo_r_mwk2_05*.
    - **last_ine_update.log**: This file logs the last week for which there is available and updated data from INE for table No. *9681*.
  - **dbs_check.py**: This script obtains a small sample of data from Eurostat and INE and checks when the earliest provisional data point is, what is the latest obtainable week of data from both Eurostat and INE and writes it in the log files contained within **logs**.
  - **functions.py**: This script contains all the functions used for fetching, manipulating and outputting the data obtained from the aggregated data sources' respective APIs (Eurostat and INE).
  - **query.py**: This script will run a pre-constructed pipeline where the data is systematically queried per age group, manipulated and where new entries are added to the *.csv* files contained within the **data** folder.

- **dashboard**: Contains all files that allow the shiny app itself to run along with shapefiles for displaying maps, shell scripts to check the databases' last update and last provisional status and the shell script to update the database from within the app.
  - **www**: This is a special shiny folder that includes files
    - **maps** Contains 2 folders which themselves contain the shapefiles for the map displayed using leaflet and the map displayed using ggplot2 in the maps section.
      - **map_shapefiles**: Contains the shapefiles used to display the map using the leaflet library.
      - **map_shapefiles_ggplot**: Contains the shapefiles used to display the map using the ggplot2 library.
    - **scripts**: This folder contains some shell scripts used when updating the database or to show diagnostic information about the databases from within the app at launch.

      - **check_dbs.sh**: This script is ran every time the app is launched. This shell script runs **dbs_check.py** from the **api** folder.
      - **update_database.sh**: This is the update database script, basically checks the date, runs the **query.py** and **dbs_check.py** script from the **api** folder, appending new data (if available) to the data currently in the cloned repository tree. Then it copies the new files into the **dreth/tfm_uc3m_data** repository, then it commits and pushes the files to that repository.
    - **dimension.js**: This javascript file is used within the application to capture the screen resolution in order to adjust the height of some plots within the app.
    - **styles.css**: This stylesheet is used to modify the style of a few elements in the app like the update database button, width and borther within tables displayed, etc.
  - **global.R**: This file does several things:
    1. Import the libraries required to run the app.
    2. Run the diagnostic database checking script **check_dbs.sh**.
    3. Read the data from the **data** folder.
    4. Assign the names of the databases from where the data was fetched.
    5. Define all the variables used in the **ui.R** script for all the fields used in the app's UI.
    6. Load the shapefiles and append the CCAA codes and IDs to their respective databases.
    7. Define all the functions used to calculate the measures the app calculates, so the *CMR*, *CRMR*, *CMIF*, *EF*, *LE* and the *death count* hidden option.
    8. Define all the functions to generate the tables used in the plotting function *plot_metric()* defined in **server.R**.
  - **server.R**: This is the server file for the shiny app, it has all the definitions for what each input or output should take and do during runtime.
  - **ui.R**: This is the UI file, it defines how the entire UI and controls for the application are.

- **data**: This folder is a subtree of the **dreth/tfm_uc3m_data** repository, therefore, it contains all the data used in the application along with logs showing diagnostic and historical information about database updates.
  - **ccaa_guide**: This folder contains files used to match CCAA names with their Eurostat denomination, as there can be multiple denominations for a single CCAA at times (like it happens with Madrid and Canarias). The files are in several different formats.
  - **logs**: This folder contains the log files used to display diagnostic information about database updates (dates).
    - **update_database.log**: This log file contains the date the database was last updated, it is written to every time the database is updated.
    - **update_history.log**: This log file contains the dates in which the database has been updated (historically).
  - **death.csv**: This plaintext file is the database file for the app containing the deaths database, corresponds to a query to the *demo_r_mwk2_05* database to Eurostat for Spain data, with all the cleaning and manipulation seen in the python scripts within the **api** folder.
  - **pop.csv**: This plaintext file is the database file for the app containing the deaths database,

corresponds to a query to the *9681* database to INE, with all the cleaning and manipulation seen in the python scripts within the **api** folder.

- **docker**: This folder contains the *Dockerfile* used to build the docker container.
  - **Dockerfile**: This file is used to build the docker container.
- **docs**: Folder containing the documentation for the application.
  - README.md: Application documentation and instructions on use.
- **paper**: Folder containing this report and all the files the report depends on.
- **COPYING**: This file contains the GNU General Public License V3, as the project is licensed under it.
- **README.md**: Readme file explaining how to run the app and some basic information about it.

### The data repository (*dreth/tfm_uc3m_data*)

- **ccaa_guide**: This folder contains files used to match CCAA names with their Eurostat denomination, as there can be multiple denominations for a single CCAA at times (like it happens with Madrid and Canarias). The files are in several different formats, like *.txt* and *.p*. The *.p* file corresponds to a binarized python object representing a python dictionary with the contents of the *.txt* file.
- **logs**: This folder contains the log files used to display diagnostic information about database updates (dates).
  - **update_database.log**: This log file contains the date the database was last updated, it is written to every time the database is updated.
  - **update_history.log**: This log file contains the dates in which the database has been updated (historically).
- **death.csv**: This plaintext file is the database file for the app containing the deaths database, corresponds to a query to the *demo_r_mwk2_05* database to Eurostat for Spain data, with all the cleaning and manipulation seen in the python scripts within the **api** folder.
- **pop.csv**: This plaintext file is the database file for the app containing the deaths database, corresponds to a query to the *9681* database to INE, with all the cleaning and manipulation seen in the python scripts within the **api** folder.
- **README.md**: Readme file explaining the source of the data and where the repository stems from, as well as linking to the main repository *dreth/tfm_uc3m*