COM4PPL: Comparison for People: context-assisted name matching comparison

Ariel Salgado
asalgado@df.uba.ar

Jose Pablo Baraybar Do Carmo
jpbaraybar@icrc.org

Inés Caridi
ines@df.uba.ar

October 2019

1 User Manual and Developer's specifications

COM4PPL provides Name and Non-name matching tools to compare databases to search cases that may correspond to the same individual. Different available datasets include both ICRC and external sources. An additional program written in R language pre-processes datasets, while COM4PPL, also in R, detects name and non-name matches.

COM4PPL allows to perform several steps of the matching process:

- It allows merging several databases into one, as the User may consider multiple databases altogether to compare with others.
- COM4PPL can detect possible matches among databases, taking contextual and individual information like nationality, gender, or age, using custom-made dictionaries and flexible rules for non-string variables. Defining compatible cases based on the non-string variables reduces the sets to compare strings (names, address). This fact speeds up the subsequent string comparison.
- COM4PPL calculates similarity scores between names, addresses, and telephones, using various algorithms metrics to detect the best matches. The user can select among different matching algorithms to capture the most relevant matches between bases or consider many of them simultaneously, considering the median of multiple scores.
- The Program can be used both with new and previously used bases to seek matches. The comparison results take the form of a ranking showing

the most similar compatible cases between the two datasets. The ranking result is sorted based on criteria that can mix the scoring result of the available string algorithms selected by the User to implement.

To realize these tasks, COM4PPL includes the following four modules that can work assembled or independently:

- **Preparing:** This module works on an original dataset that has not an assigned ID for individuals. The script adds an ID for each individual for the dataset to be ready to preprocess.
- **PreProcessing:** This module works on the original dataset, preprocessing it to standardize the variables to be taken into account later.
- **PreMatching:** This module allows the union of selected bases to generate the adequate two inputs (*base A* and *base B*) needed for the desired comparison. Also, the script creates some variables defined explicitly for the matching comparison.
- **Matching:** This module calculates the similarity scores between the two bases, based on the selected algorithms. It also prepares the files for expert assessment.

  The **PostMatching:** module will be included in the Phase 2.0 of the Program. It mixes the actions regarding the processing of new data with results evaluated by experts (for example, further iterations with experts).

A **Demo** of the Program is prepared in the script *Demo.R* and detailed in section 2.

Directories of the Comparison Program: The Comparison Program contains three directories, named Dictionaries, MatchingProgram, and rawBases. The Dictionaries include different possibilities in which the nationalities, genders and locations may be written (in further stages, other dictionaries could be included). These are in the file Dictionaries/dic_countries_iso2.csv. This file has the following columns:

- Country.of.origin: The name of the country as written in the original dataset.
- Processed.Africa: The name of the country in standardized form: all in capital letters, and without accent ticks. If the country has a previously written entry its standardized name should be used.
- ISO2: The ISO 2 of the country.

- ISO3: The ISO 3 of the country.
- NOM_COURT: The complete name of the country.

The matching program folder contains the necessary scripts for the comparison, as will be explained below. The rawBases folder contains a folder for each dataset, named after its code. Each folder contains the dataset with the name COD_input.csv, with COD being the dataset code. Also, the output of the Pre-Processing scripts from the matching program is located inside this folder, with the name COD.csv.

Some useful definitions: A remember*compatible case* consists of a pair of individuals who could be the same person regarding other variables than names, variables which are *compatible* based on certain criterion to be defined by the User. Thus, each pair of compatible cases is taken into consideration only once. The information of the individuals is organized as follows: on the left side the individual who correspond with the dataset selected as List A (previous order dataset), and on the right side that who belongs to the dataset selected as List B (posterior order dataset). An example of the use of the scripts can be seen in the *Demo* of the Program.

1.1 Preparation stage

The Preparation action is necessary only if a database has not a defined IDs for the individuals. The script *add_id_to_base.R* defines an ID for each individual in a correlative way following the rule: a code for the database (which is selected by the User) followed by a symbol "-" and followed by a number beginning with zeros, and then, 1 (for example *LAB-0001*) for the first case, and so on.

1.1.1 Inputs

The **input** required by the script *add_id_to_base.R* is the following file:

1. *pathAddIdToBase.csv*. In this file the User should write the desired code for the database and the name of the input file to add IDs.

1.1.2 Outputs

The **output** of the script *add_id_to_base.R* is the database with the Case.ID column added following the code chosen by the User. The file called *COD_input.csv* is added in the directory Rawbases/COD (as the rest of the databases). In case this file already exist, the file will be called *COD_input_new.csv*.

1. *path_AddID.csv*, where the User writes the desired code for the database.

1.2 PreProcessing stage

The Preprocessing action includes the script *preproc.R* used to preprocess a selected database. This script standardizes variables like names and nationalities and allows applying modifications to some variables of interest chosen by the User.

1.2.1 Inputs

The **inputs** of the script are the following:

1. *cod_to_proc.csv*: This file includes the specific codes of the bases to be preprocessed.
2. *info_proc.csv*: This file has information regarding the variables that the User wants to preprocess and appear in the processed file. It contains the following columns:
   - *Cod*: the code of the base to which the variable corresponds.
   - *var_original*: the name that identifies the variable in the original database.
   - *var_new*: the new name that will be associated with the variable after preprocessing.
   - *var_type*: the type of variable, necessary to know how the variable should be preprocessed. It can be: ID, age, age2 (year of birth), nat (nationality), name, role, address, sex and date.
   - *is.proc*: If the variable must be preprocessed or not. (i.e. if a variable named *var_new_PROC* containing the processed variable must be created or not). It can be 1 (yes) or 0 (no).
   - *to.join*: for each variable, it is possible to set if it must be included or not when the union between bases is made (in the following stage). It can be 1 (yes) or 0 (no). In case *is.proc* = 1, the preprocessed variable will also be included.
3. *Dic*: Dictionary of countries. It is assumed that it will be found in the path *'../Dictionaries/dic_countries_iso2.csv'*
4. *Dic_loc*: Dictionary of localities. It is assumed that it will be found in the path *'../Dictionaries/dic_locations.csv'*
5. *parameters.csv*: File with additional parameters. Currently it only has a cutoff date used to calculate the age using the year of birth.
6. *select_bases.csv*: File with additional information about some restrictions regarding the subset of data to build from a specific database. For example, it allows specifying whether only African countries should be included in the subset, or whether only individuals of a certain role should be chosen (i.e. sought persons), or only those cases with at least one name. It

has one line for each specified dataset. If a dataset is not included, there will be not any restrictions over it.

7. *COD_input.csv*: This file has the information about the database to be preprocessed. The script searches for file with the COD_input.csv form in the location
'../rawBases/COD/COD_input.csv', where COD is the "base code" (for example AGA, UNI, SN_TR and so on).

8. *helpersProc.R*: File with the functions needed for preprocessing.

1.2.2 Outputs

The script *preproc.R* generates as **output** a file with the preprocessed selected database, in the corresponding folder inside the folder *rawBases*. For example, for the Agadez database, whose base code is AGA, it generates a file in *'../rawBases/AGA/AGA.csv'*. This file only contains the variables that were indicated by the User in *info_proc.csv* to be preprocessed, along with the *_PROC* variables that were indicated there.

Moreover, the script writes in the R terminal a message for the User of the type: *COD preprocessed! with R rows and C counts* with *COD* being the code of the database which was preprocessed, R the number of rows of the database, and C the total number of columns of the database in the preprocessed database (the original columns which were preprocessed plus the new columns added).

The variable *age_2* indicates that age must be calculated since the year of birth. The age is calculated since the first day of the year of birth until the cutoff date indicated in *parameters.csv*. The age is defined as the integer part of the difference in days between these dates over 365.

Suppose the corresponding variable type has more than one column (like two possible ages or nationalities). In that case, the cases will be compatible if at least a pair of non-empty values (non-NA) meets the criteria. For example, if person A has the possible ages 31 and 52, and person B has 32 and 20, with a range of 6 years, A and B are compatible (because the pair of values 31 and 32 meets the criterion). If A has possible ages 31 and NA (only has one age in the data), they are still compatible, but if A has possible ages NA and 52, A and B will not be compatible.

The preproc.R script employs three dictionaries: one for nationalities (in *Dictionaries/dic_countries_iso2.csv*), one for genders (in *Dictionaries/dic_gender.csv*), and one for locations (in *Dictionaries/dic_locations.csv*). The processing assigns a case of the dictionary when there is an exact match between a record of the the dictionary and the database, made in lower case. For example, if a person has the nationality Cote d'Ivoire and the dictionary has the entry cote d'ivoire, the processed nationality will have IVORY COAST in the processed nationality. So, if another person has COSTA DE MARFIL in its nationality, there must be another entry in the dictionary with the form: COSTA DE MARFIL, IVORY COAST. The columns of the dictionary files are:

- Country.of.origin, Processed for *dic_countries_iso2.csv*
- Input,Gender for *dic_gender.csv*
- location,*GZ_name* for *dic_locations.csv*

In each case, the first column is compared with the database, and the second one is looked for processed names. While a new base is preprocessed, it's important to check the result and look for new cases to be added in the dictionaries.

1.3 PreMatching stage

This part is responsible for joining different databases in the same file to prepare them for comparison. There are various comparison schemes possible between the two lists: **List A** and **List B**. For this, we will join all the bases corresponding to the same *instance or instances*. To be clear, we consider an *instance* a grouping of different databases. It typically corresponds to the same travel instance (for example, departure points), although the user may have other reasons for grouping the database. Members of the same instance are not matched among them but with members of some other cases. We call a *scheme* (also *scenario* later) of comparison to the election of which variable is evaluated to match between members of different instances. For example, a scheme could be *name* when we search for members with similar names. Other could be *Father* when we seek for members with similar father's name.

The script *Join_bases.R* is also responsible for generating the intermediate variables that will be used later in the matching comparisons. These variables are calculated with a specified function in *helpersJoin.R*. So, if a Developer wants to modify them, they must change the code of this function.

1.3.1 Inputs

Some examples of each input file can be seen at the Annex 2, in the Demo of the Program. This script joins groups of databases to standardize their fields (since all databases don't have the same fields nor name the variables in the same way).

The script joins a group of databases in the same List (A or B). To select the group, each database has a specific *instance*, a value indicated in the column instance of the file *info_bases.csv* that could be modified according to the group of databases that the User wants to join. Databases with the same instance value will be joined together in the same List. The instances to be joined (can be one or more than one) also need to be specified on the line *instances=c(1,2)* of the script *Join_bases.R*.

The required **inputs** for the script *Join_bases.R* are the following:

1. *info_bases.csv*: This file has the information about which instance corresponds to each database and the order levels of the different bases (the order levels are not used in this Phase of the Program, but will be used

in further stages). It is essential to notice that these two columns are independent of each other.

2. *info_proc.csv*: It was previously explained.

1.3.2 Outputs

A file is generated for each considered instance in the path *instances /instX/base_instX.csv*, where X indicates the corresponding instance (for example, instance 1 and file /inst1/base_inst1.csv). This file contains the summarized cases of the different databases belonging to the same instance X and all the columns in them. If the name of a column is repeated between different databases, the columns are merged. If a column is absent in some databases, the missing values are completed with the string *NA* (i.e., not available).

1.3.3 Scenarios:

The scenario columns to be used in the subsequent matching are added in this stage. The possible scenarios currently supported (and their codes) are:

- *Name of the individual* (*A_A*): All the Name columns of the individuals, concatenated in one string;
- *Father's name of the individual* (*F_F*): All father's name columns, concatenated in one string;
- *Mother's name of the individual* (*M_M*): All mother's name columns, concatenated in one string.
- *Address or locations of the individual* (*AD_AD*): All address columns, concatenated in one string.
- *Phones* (*Ph_Ph*): All phone number columns, concatenated in one string.

1.4 Matching stage

The matching is done using the files created in the prematching stage. This step includes the script *genSubset.R* and in a further version of the program the script *addMicScore.R*. The former script calculates the similarity scores using R, while the latter adds, eventually, the similarity scores provided by Microsoft Fuzzy String Matching. The script is described below:

1.4.1 *genSubset.R*

This script starts generating a list of compatible cases from Base B for each individual case in Base A. This is done by considering the selected compatibility rules for different variables. There are two types of rules: one requires that two categorical variables (nationality or gender) are the same or not. For this rule, we consider compatible those cases belonging to the same category.

The second rule requires that the difference of a numerical variable is within a determined range (such as age). For the former, we consider compatible those cases belonging to the same category. We consider compatible cases that differ in no more than a specific value (for example, six years).

It is necessary to specify which variables should be considered and what kind of variable is each one, in the file *variableFields.csv*. In the file *compatible_data.csv*, it is specified how to compare each type of variable to define the global compatible rule.

After detecting the compatible cases for each individual of Base A, the script calculates the similarity scores for each compatible pair based on the different scenarios selected. For this purpose, the file *select_schemes.csv* indicates which scenarios should be considered, with all the possible scenarios being in the file *info_scheme.csv*.

The script calculates for each pair of possible cases the similarity scores based on the algorithms indicated in the file *info_score.csv*. The algorithms that the User wants to calculate must be indicated with *yes* in the column *calculation* of the file *info_score.csv*. After this step, the median of a subset of the algorithm scores (those specified with 1 in the column *median* of *info_score.csv*) is also computed. The median value is used to define if the compatible pair of cases will be written or not as a candidate pair for expert assessment. Those candidate cases to be assessed are the ones obtaining a median value greater than or equal to the specified threshold value in the column *threshold* of the file *info_scheme.csv* for the corresponding scenario.

After putting this methodology into practice for each selected scenario, the compatible pairs considered to be candidates for expert assessment based on the selected scenarios are written in the output file *candidateList.csv* alongside their median scores and detailed scores for each implemented algorithm.

1.4.2 Inputs

The script *genSubset.R* needs the following **inputs**:

1. *helpersSubset.R*: It is a file with functions to preselect compatible cases based on the rules considering the non-name variables (age, gender, nationality) and the functions for calculating the matching similarity scores.

2. *base_instA.csv*: This file contains the database generated by the script *Join_bases.R*, which will be considered as containing the elements on the left side (i.e. for each case of base A, the plausible cases in base B will be searched for comparison to detect compatible cases).

3. *base_instB.csv*: Identical to the previous one, but the base contains the elements on the right side of the file for expert assessment.

4. *compatible_data.csv*:This file indicates how the compatibility rule to define compatible cases is determined based on the criterion applied to each variable type (one for each row). The file has the following columns:

- variable: name of the variable
- type: type of the variable, within the following options: i) *Case.ID*, only for this type of variables; ii) *cod*, only for this type of variables; iii) *categorical* for variables for which the compatibility rule implies the use of a variable of the categorical type, like nationality or sex; iv) *range* when the variable requires a range to define the compatibility rule.
- consider: in case the variable is considered to define the compatibility rule (consider=yes) or not (consider=no).
- parameter: in case the variable considered to define the compatibility rule is of the categorical type, a path to the file with equivalences between categories can be added (for example Mali as equivalent to Mauritania). This file must show two columns with the names NatFROM and NatTO. In case the variable is of the range type, the range to be used must be indicated in the column parameter (for variables like age, for example).
- na_action: the chosen option defines what action will be taken with NA values to define the compatible cases. There are three possible options: i) all: when the cases with empty (NA) value in such variable are considered to be compatible with any other case (including NAs); ii) none: when the NA cases are not compatible with any other case (nor NA cases); iii) na: in case the User wants an NA case to be compatible only with other NA cases.

5. *variableFields.csv*: A file indicating which variables should be used for each type of variable (for example, Age_1_PROC and Age_2_PROC can both be used for the type of variable age). If more than one column is specified, they should be separated by one empty space.

6. *info_score.csv*: This file contains different simmilarity algorithms and indicates which one should be calculated (*yes* in column *calculation*) or not (*no* in column *calculation*), or included in the median (*1* in column *median*) or not included (*0* in column *median*). In this version of the Program, this information is general for all the considered scenarios. In further versions, a particular selection for each considered scenario will be allowed.

7. *info_scheme.csv*: This file contains the IDs of different scenarios and indicates which columns of each List (A and B) they compare. Also, it contains the thresholds for each median of each scenario to be surpassed by compatible cases to be accepted.

8. *select_schemes.csv*: A file containing the IDs of the scenarios to be considered. One per line. The order is important (the first one is the priority scenario for sorting cases for assessment).

1.4.3 Outputs

The script generates a file named *candidateList.csv* as the **output**, where all the information of each pair of possible individuals to be assessed is detailed in each row: i.e. all the variables of the element of Base A, and all the information on the element of the Base B (information that was present in both provided input files).

Note for Developers: This and all the file names of the outputs can be changed by modifying the script.

1.5 PostMatching stage

This section of the process will allows us to incorporate previous assessments from the expert in the new results. It consists of several scripts that take into account the result of the matching, and further indications from the expert. These scripts will be included in the Phase 2.0 of the program.

2 Demo of the Matching name and non-name Program

This is an example of how to use the scripts of the Matching Program.

At first, open R and let's move there:

```
setwd('MatchingProgram/')
```

1- First of all, you will need at least two bases in the */rawBases* folder. We'll be using base 1 (BA1) and base 2 (BA2) example bases (in */rawBases/BA1/BA1_input.csv* and */rawBases/BA2/BA2_input.csv* respectively).

2- These two bases need to be preprocessed at first. For this purpose, we will use the script *preproc.R*. But previously, we have to specify two files: *cod_to_proc.csv* and *info_proc.csv*.

2.a- This Demo provides you with an *info_proc* file. Open it, and look at its content. You will see in each line something like this:

```
Cod,var_new,var_original,var_type,is.proc,to.join
BA1,Case.ID,ID,ID,0,1
```

First there is the code of the base, then the new name of the column, then the original name, then the type, and then how the script should use this information. For further details, check the User's Manual.

2.b- Now you have to specify which databases must the script *preproc.R* preprocess, through *cod_to_proc.csv*. You can manually create that file, or create it directly with this command:

```
write(c('BA1','BA2'),'cod_to_proc.csv')
```

Note that the file has two lines, one for each code of the base to preprocess.

2.c- Now you can preprocess both databases:

```
source('preproc.R')
```

In the screen, the message will appear:

```
[1] "BA1 preprocessed!"
[1] "with ... rows and ... columns"
[1] "BA2 preprocessed!"
[1] "with ... rows and ... columns"
```

3- Now you have to use the script *Join_bases.R* to prepare two instances for matching.

3.a- Let's start checking the file *info_bases.csv*. For our purposes, we have these lines:

```
cod,instance,orden
BA1,0,1.1
BA2,0,1.2
```

Change them so that they look like this

```
cod,instance,orden
BA1,1,1.1
BA2,2,1.2
```

Also, look for other lines having a nonzero value for instance, and change them to 0.

3.b- Now open the *Join_bases.R* script and check the line 3. It should say

```
instances = c(1,2)
```

in order for the instances 1 and 2 to be considered for joining. Now run the script:

```
source('Join_bases.R')
```

4- Now we can run the matching between these two databases, which now are in the folders *instances/inst1* and *instances/inst2*

4.a- First of all, copy the two new files to the main folder

```
file.copy(from='instances/inst1/base_inst1.csv',to='base_instA.csv',overwrite = T)
file.copy(from='instances/inst2/base_inst2.csv',to='base_instB.csv',overwrite = T)
```

4.b- Now, open the script *genSubset.R*, and check the lines 3 and 4. They should be:

```
base1 = read.csv('base_instA.csv',stringsAsFactors=F)
base2 = read.csv('base_instB.csv',stringsAsFactors=F)
```

You can also check the file *compatible_data.csv*. Rules for compatibility between cases are written inside. You can change the age range for example, or exclude some of the rules. Here, we'll be using:

```
variable,type,parameter,use
Case.ID,Case.ID,,ignore
cod,cod,,ignore
nationality,categorical,"natEquivalences.csv",include
sex,categorical,,include
age,range,6,include
```

Inside the file *natEquivalences.csv* we are considering as equivalent nationalities MALI and MAURITANIA.

4.c- Let's check now the files *info_score.csv* and *info_scheme.csv*. In *info_score.csv* you will find what similarity algorithms scores will be calculated, and which one should be considered to calculate the median for each scenario. In *info_scheme.csv* you will find the four different possible scenarios and the threshold below which the pair of compatible cases will be discarded.

4.d- Now you have to specify which scenarios you want to compare. You can change the file *select_scheme.csv* manually or from here.

```
write(c('A_A','F_F'),'select_scheme.csv')
```

and run the script:

```
source('genSubset.R')
```

With the output *candidateList.csv*, we have completed the comparison of both databases. You can read the result through:

```
candidates=read.csv("candidateList.csv",stringsAsFactors = F)
View(candidates)
```

5- In case you receive a new database without the individual's ID codes, it is necessary to apply the *preparing* stage first in order to add the IDs to this database. Lets suppose the database is the file *new_database.csv*. It is necessary that the name of the input file (the new database in *.csv* format) is written in the file *pathAddIdtoBase.csv* (in the first line the name of the input, in the second one the new code for this database), like:

```
New_database.csv
NEW
```

Run the script to add IDs:

```
source("add_id_to_base.R")
```

Then, the NEW directory is created, with the dataset *NEW_input.csv* and the IDs added. In case the directory already exists, the file will be called *NEW_input_new.csv* in the same directory.

5.a- Then, lets suppose we can compare this new dataset to the others (*MR_TR*, *SN_TR*, *LAB* and *DEB*). First it is necessary to preprocess it (and all the datasets, if there are not already preprocessed, following the same steps). To do that, it is necessary to follow the step 2.b- specifying that you want to preprocess this new dataset, which code is "NEW". Then this is the code to replace in *cod_to_proc.csv*. Then we write manually:

```
NEW
MR_TR
SN_TR
LAB
```

in this file, in order to preprocess these two files, or through:

```
write(c('NEW','BA1','BA2'),'cod_to_proc.csv')
```

5.b- Besides, the corresponding information in the file *info_proc.csv* is needed to be included for this NEW dataset (following step 2.a-) Then, let us preprocess these four datasets:

```
source("preproc.R")
```

5.c- Then, it is necessary to join the new base as the baseA and *MR_TR* *SN_TR* and *LAB* to baseB, following the step 3.a-. In *info_bases.csv*, we need to have these lines:

```
cod,instance,orden
BA1,1,1.8
LAB,1,2.8
... (other bases)
NEW,1,1.3
```

Now, following the step 3.b-, the *Join_bases.R* script should say

```
instances = c(1,2)
```

Now run the script

```
source('Join_bases.R')
```

Then it is necessary to update the files, following step 4

```
file.copy(from='instances/inst1/base_inst1.csv',to='base_instA.csv',overwrite = T)
file.copy(from='instances/inst2/base_inst2.csv',to='base_instB.csv',overwrite = T)
```

5.d- Finally, we are ready to run the comparison between NEW base (Base A) against the bases *MR_TR*, *SN_TR* and *LAB* (Base B). As the *NEW* base has no information about nationalities, it is necessary not to consider this variable for the compatibility rule. Just put the option *consider=no* in the file *Compatible_data.csv*

```
source("genSubset.R")
```

You can observe the results in R:

```
candidates=read.csv("candidateList.csv",stringsAsFactors = F)
View(candidates)
```

or open the output file in another program of Excel type.

5.e- You can check the change in the output file *CandidateList.csv* when you modify the threshold of the Scenario *A_A* and the input file *info_scheme.csv* (for example 0,7):

```
scheme listA.column listB.column threshold
A_A      Name_A        Name_A        0,7
```

You can run the script once again:

```
source("genSubset.R")
```

and observe the new results in R:

```
candidates=read.csv("candidateList.csv",stringsAsFactors = F)
View(candidates)
```

3 Comparison Program Glossary

Information files The files in this section correspond to files that contain information specifying aspects of the information processing.

- `info_proc.csv`: Contains the information about which variables should be processed for each dataset. It has six columns: *cod*,*var_new*,*var_original*,*is.proc*, *type* and *to.join*.
- `info_bases.csv`: Contains the information about which instance corresponds to each dataset, alongside it's order level.
- `info_score.csv`: Contains the different possible measures to be used in the comparison between names, and also the subset of measures to be used in the calculation of the median.
- `info_scheme.csv`: Contains the columns associated to each scheme of compatibility, and also the threshold for each scheme.
- `select_bases.csv`: Contains information for the preprocessing of each dataset. For example, if only certain cases (like African or missing persons) should be considered.
- `compatible_data.csv`: Contains the different types of variables used to search for compatibility between cases, and which parameters and NA considerations should be used with them.
- `variableFields.csv`: Contains the columns associated to each type of variable used to search for compatible cases.
- `parameters.csv`: Contains parameters needed for the matching. Currently it only contains a cutoff date.

Select files These files indicated which bases or scenarios should be used in different places.

- `cod_to_proc.csv`: Indicates which datasets should be preprocessed by `preproc.R`.
- `select_schemes.csv`: Indicates which scenarios should be calculated by `genSubset.R`.

Helpers files These files contain helper functions used by R scripts

- `helpersSubset.R`: Associated with genSubset.R
- `helpersScore.R`: Associated with genSubset.R
- `helpersProc.R`: Associated with preproc.R
- `helpersJoin.R`: Associated with Join_bases.R

Script files These files correspond to the different tasks expected to be made by the matching program.

- `preproc.R`: This script preprocesses the datasets, standardizing them.
- `Join_bases.R`: This script mixes and structures the bases to make them able to be accessed by `genSubset.R`
- `genSubset.R`: This script detects compatible cases and calculates the scores between them.

Input and output files These files function as outputs or inputs of the scripts.

- `instances/instX/base_instX.csv`: These files correspond to the outputs of `Join_bases.R`. Each X corresponds to a different instance.
- `base_instA.csv`,`base_instB.csv`: These files are the bases used as inputs for `genSubset.R`.
- `candidateList.csv`: This file is the output of `genSubset.R`. It contains the cases that surpassed the rules and the score threshold.
- `rawBases/COD/COD_input.csv`: It's the original dataset corresponding to the code COD. The column names which are going to be preprocessed must correspond with the ones listed in `info_proc.csv`
- `rawBases/COD/COD.csv`: It's the preprocessed dataset corresponding to code COD. The column names correspond to the ones in `rawBases/COD/COD_input.csv` through `info_proc.csv`.

Matching similarity measures The range value of these similarity measures is $[0, 1]$. The highest similarity corresponds to a value of 1, and the lowest to a value of 0.

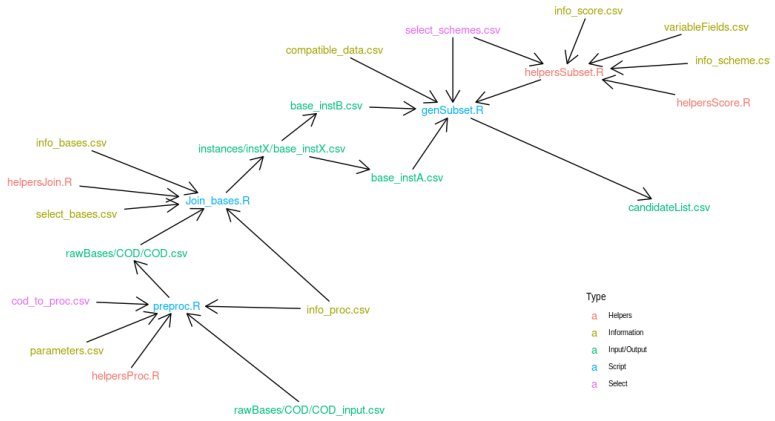

Figure 1: Files and scripts of the Comparison Program. The network shows which input needs each script, and the flow of the matching program.

3.0.1 A commentary on the scoring functions installing

The functions used for scoring the similarity among cases come from two sources: R-package stringdist and Python-package fuzzywuzzy. As the whole code is implemented on R, the connection to python can be cumbersome. Particularly, special attention should be put on the installation of Python-packages python-Levenshtein, fuzzywuzzy and difflib. Their installation should be done under the same source of python which R is using. For further questions and issues on this particular topic, we refer the reader to the fuzzywuzzyR repository, available on github. Many common issues have already been addressed there.

Stringdist Functions A detailed explanation of these functions can be found in [**?**], the reference paper of the stringdist package [**?**].

- **Levenshtein distance** (*lv* in *stringdist* R package): This method calculates the distance between words considering insertion, deletion or changing of letters, but it does not consider transposition. The distance calculation is done by counting the number of necessary changes. The similarity (s) is defined as 1 minus the ratio between the distance (d) and the length of the longest string (l1 or l2).

  Example: string1 = John Smith, string2 = Jonh Smit We need two deletions and one insertion, making a distance of d=3. The similarity is s=1-d/(max(—l1—,—l2—) = 1- 3/10 = 0.7 (as the space counts as character).

  Note: there is more than one way to obtain a string2 from string1 through deletions, insertions, and changing of letters.

- **D-L** (osa in stringdist R package): This implements the Damerau-Levenshtein method, which includes the possibility of transposing adjacent letters or characters to the Levenshtein distance. This method considers only one transposition per letter (character).

  Example: using s1 and s2 as before, we can transpose the 'h' and the 'n' in s2, making a distance of 2 and a similarity of 0.8.

- **D_L_FULL** (dl in stringdist R package): The same as osa method, but allowing multiple transpositions on the same character. Example: as no more than one transposition is involved in the previous example, it gives the same as osa.

- **LongestCommonSubstring** (lcs in stringdist R package) This method was designed by (Needleman and Wunsch, 1970). Conceptually, it measures the number of pairing characters between both strings. The distance is the number of insertions and deletions needed to convert one string to another (so, It's like the lv distance, but without substitutions). As we have no substitutions, the similarity differs from the lv similarity. Here the maximum distance is the sum of the length of both strings sum(—l1—,—l2—), so the similarity is 1 less the ratio between the distance and the maximum Example: Using string1 and string2 as before, we need to delete one character, and insert two. For example Jonh Smit Jon Smit John Smit John Smith The similarity will be 1- d/sum(—l1—,—l2—) = 1 – 3/19 = 16/19 = 0.842

- **GramX** (qgram in stringdist R package with different q values): this method compares the strings by doing anagrams. The q determines the size of the anagram to be used. For example, the q = 1 case separates the string in its letters; the q = 2 case, in pairs of letters, and so on. The distance counts the unmatched q-grams between the strings (without matter the order of the q-grams). The similarity is obtained as usual, with the maximum of the q-gram distance being the sum of the length of both strings, less 2(q-1).

  Examples: Using string1 and string2 as before, with q=1 we have s1 J,o,h,n, ,S,m,i,t,h, s2 J,o,n,h, ,S,m,i,t So we can see that there is only one 1-gram unmatched (the last 'h'). The similarity is 1- 1/19 = 18/19 =0.947

  Using q=2 we have s1 Jo,oh,hn,n , S,Sm,mi,it,th, s2 Jo,on,nh,h , S,Sm,mi,it So we can see that the number of unmatched 2-grams is 7. The similarity is 1-7/(19-2)=10/17=0.588

  Using q=3 we have s1 Joh,ohn,hn ,n S, Sm,Smi,mit,ith, s2 Jon,onh,nh ,h S, Sm,Smi,mit So we can see that the number of unmatched 3-grams is 9. The similarity is 1-9/(19-4) = 6/15 = 0.4

Fuzzywuzzy Functions The fuzzywuzzy functions are base upon the Gestalt Pattern Matching method, which is described in the ratio function. An explanation of this method can be found in [**?**], where is compared with the Jaro-Winkler distance.

- **Ratio** The ratio function is based on the Gestalt Pattern Matching method (https://ilyankou.files.wordpress.com/2015/06/ib-extended-essay.pdf). From two strings, the method takes the longest common string they have (like in the lcs method), and removes it from the two strings, obtaining two pairs of strings (one which was before the removing string, and one after it). Recursively, it repeats the extraction of the longest common string in each pair until there are no more common characters. The number of characters left is the distance between strings. The similarity is defined as two times the ratio between the sum of lengths of matching common strings and the sum of the length of both strings.

  Example: Using string1 and string2 as before, we have s1='John Smith' and s2 =' Jonh Smit', so the longest inner string is ' Smit', so we get a 5. On the left, we have 'John' and 'Jonh'. Repeating the procedure, the new longest inner string is 'Jo', so we get $5+2=7$. There is nothing on the left for this substring, but in the right, it is hn and nh. The longest substring is n (and we get $7+1=8$), and, as h is on the left in one case and on the right in the other, there are no more characters to continue. On the right of the first string, there remains another 'h', but with nothing to be compared. The similarity is defined as two times the ratio between the sum of lengths of matching common strings and the sum of the length of both strings.The similarity then is $2*8/19 = 16/19 = 0.842$

- **partial_ratio** It is the same as the ratio method, with the difference that this method considers the result with respect to the length of the shortest initial string. So the similarity between two strings for which one includes the other is 1. Example: Using string1 and string2 as before, the distance calculation will be the same, but the similarity will be calculated over the shorted string as $8/9 = 0.889$ (number of matching characters over a length of shorter string).

- **token_sort_ratio** This method tries the different combinations of the tokens. The tokens are splittings of the initial strings, using a special character for the splitting (a space for example). It's especially useful for strings containing more than one words. It returns the best similarity score possible from the combination of tokens.

  Example: Using string1 and string2 as before it will give the same result as in ratio. If we consider the string3 = Smit Jonh, it will give the same result too, because the method tokenizes string3 as Smit,Jonh, and then calculates the ratio for 'Smit Jonh' as 'Jonh Smit'.

- **WRATIO** This method is a combination of other fuzzy methods. It takes into consideration the length of each string and scales the result considering it. First, it calculates the ratio method of the strings. Then, it compares the length of the strings. If one of the strings is more than 1.5 as long as the other, it takes the partial ratio, but scaled to a max similarity of 0.9. If one of the strings is more than 8 times it scales the result to a max of 0.6 instead. If one of the strings is more than 1.5 as long as the other, it calculates a version of token sort ratio considering *partial_ratio* (instead of *ratio*), else it calculates token sort ratio. It scales this results to a max of 0.95. Finally, it returns the maximum of all these similarities.

  Example: Using string1 and string2 as before will give the same result as in ratio. Let's consider string4 = William Jonh Smit. The length of string1 is 10 and the length of string4 is 17, so we get the similarity to 0.9. As the number of matching character works as the same as in ratio, we will have 8 matching characters, but with a similarity of $2 * 8/27 = 0.593$, and scaling it by 0.9 gives 0.533. Then, we need to calculate the partialized version of token sort ratio. As the best ordering is the one that we already have, it's the same as using partial ratio. As before, it will give 8/10, because it's limited to the length of the shortest string (string1 in this case). This result is scaled by 0.95, so we get $0.80 * 0.95 = 0.76$. As this is the maximum of the scores, this is also the result.