Paper DH01

Data Anonymisation and Risk Assessment Automation – PHUSE WG Deliverable

Lukasz Kniola, Biogen, Maidenhead, UK

ABSTRACT

PHUSE Data Transparency WG has recently released a new white paper – "Data Anonymisation and Risk Assessment Automation". It describes and organizes the process of anonymising data and discusses ideas how to automate it, focusing on individual tasks, assumptions, and potential pitfalls. Alongside the process map it offers coding examples in different languages (Python, R, SAS®, Julia).

This paper introduces the new PHUSE deliverable and summarizes its contents. It also looks at how different programming languages can be used to process the data in the context of anonymisation. Finally, it explores the high-level similarities and differences between programming languages used in the white paper.

DISCLAIMER

The scope of this paper is to present the opinions and suggestions of the author. The interpretations of standards and procedures contained in this paper are those of the author and are not necessarily correct. Any views, thoughts, and opinions stated within this paper are those of the author, and they do not represent the positions of their employer or any other organization or group they may be affiliated with.

BRINGING THE DOCUMENT TO LIFE

The PHUSE white paper "Data Anonymisation and Risk Assessment Automation" was released in July 2020 and is freely available on the PHUSE website (https://www.phuse.eu/white-papers). The original idea behind this project was to create a process map for anonymising datasets which provides as much detail as possible on individual tasks, considerations, pitfalls as well as highlight the potential for automation on a task, step and even cross-step level. The ambition was to create a comprehensive document, reviewed by the community and accepted as the base for the dataset anonymisation process.

Under the supervision of the PHUSE Data Transparency Working Group and its lead Jean-Marc Ferran, a group was formed to work on the project, led by Lukasz Kniola (author of this paper).The team included de-identification SMEs from across the industry, who kindly contributed their time and expertise.

The first task was to list the main steps required to transform the original (source) datasets to anonymised ones. Each step was then divided into individual smaller items – tasks, rules to be applied, etc. The result was a list of 10 major steps and around 60 individual tasks/rules between them. Each had to be discussed and reviewed to provide as much information and context as possible based on the team's collective hands-on experience of implementing many of the described solutions as part of their jobs and projects. All information was collected in a huge spreadsheet of dozens of tabs, each dedicated to a small part of the process. This format did not facilitate easy consumption of the material, nor did it have the right "flow" to it. It needed to be converted to a continuous document. Through editing and revisions, some sections were merged or moved. Many examples and illustrations were also added.

The final draft had gone through three rounds of reviews – within the Working Group, within the PHUSE community, and finally a public one. The comments were reviewed and carefully implemented before the final version of the white paper was released on the PHUSE website.

OVERVIEW

In its final form, the document covers a great deal of material regarding the anonymisation of datasets. It starts with pre-processing of the datasets and setting assumptions and conditions. It discusses finding identifiers in the data, applying various rules to modify those identifiers. It looks at different aspects of risk and how to assess it. It also talks about putting it all together and adding documentation in the form of an anonymisation report.

At each step, it was important to describe the tasks involved as well as include ideas and examples which may not be immediately obvious, especially to those new to this topic. An important aspect of the document was highlighting things to look out for, like pitfalls and common mistakes.

Finally, there are thoughts on how to automate parts of the process, how to take advantage of the tools already available and to make use of the fact that there is quite a bit of repetition from one project to another.

For better readability, the discussions were based around SDTM format. This simplifies things when it comes to naming conventions, and dataset structures. However, these ideas can be easily extrapolated onto other formats – ADaM or legacy datasets.

Finally, what the white paper does not cover is concepts specific to documents anonymisation. It was a deliberate decision, as this would add another layer of complication.

CODE EXAMPLES

To offer suggestions on how parts of the process can be automated it was necessary to go one step beyond the description of individual tasks. One option was to provide block algorithms to describe the process flow. The final decision, however, was to provide actual code examples.

It became clear that choosing one programming language over another would not be appropriate. In the spirit of openness and to avoid focusing on one tool over others, code examples were prepared in Python, R, SAS, and Julia (the last one only to limited extent). The code examples cover individual rules and transformations as well as more complex tasks.

Thoughts and impressions on using different programming languages in relation to dataset anonymisation are shared later in this paper.

THE LENGTH OF THE PROCESS

The document is divided into seven sections.

1. GATHER ORIGINAL DATA

This chapter considers identifying the right source data, specifications, and formats.

2. PRE-PROCESS DATA

The focus is placed on non-standard datasets to bring them close to SDTM-like structure. Recommendations to ensure consistent subject ID, unifying codes and decode, dealing with formats and information (for SAS datasets) can simplify subsequent steps.

3. FIND POTENTIAL DIRECT AND QUASI-IDENTIFIERS

There are numerous suggestions on how identifier variables and values can be found in datasets. It is advisable to take advantage of the PHUSE De-identification Standard for SDTM [1]. There are code examples specifically how to apply the standard to datasets and match the metadata with the classification and rules specified in the standard.

Beyond that, the text offers thoughts on how to find identifier variables based on the variable names as well as their contents, and what type of information to search for.

4. DEFINE DE-IDENTIFICATION RULES, BUILD SPECIFICATIONS

All common rules which may be applied to different types of identifiers are described in this section. Starting with simple KEEP, DROP, through recoding and hashing ID variables, offsetting dates (OFFSET), to pooling age values (e.g. AGE_BANDS), countries (COUNTRY_POOL), and low frequency values (LOW_FREQ_POOL).

Each rule includes detailed description and context. Potential pitfalls are highlighted. Each rule also includes code examples in the 3 common languages.

5. CALCULATE RESIDUAL RISK OF RE-IDENTIFICATION

Risk of re-identification is calculated to confirm if the applied transformations are sufficient for the data to be shared and reused safely, and participants' privacy is respected and secure. This is achieved by comparing the calculated residual risk of re-identification to a pre-specified threshold.

This is an integral part of the document which introduces the terminology, assumptions, and formulas. It covers all relevant aspects of re-identification risk calculation: risk of attempt, risk of success, k-anonymity and uniqueness, reference populations, as well as thresholds to check the results against.

An example runs through this chapter to better illustrate the concepts and help follow the logic.

Well described code examples provide ready-to-use scripts to calculate relevant risk statistics on a dataset. Apart from SAS, Python and R, used for all other examples, this is also offered in Julia.

6. APPLY DE-IDENTIFICATION RULES TO DATASET

When all direct and quasi-identifiers have been found, and the risk on original data calculated, the next step is to prepare dataset specifications by assigning anonymisation rules to each variable across datasets. Based on the classification of all identifiers and non-identifiers each variable should be assigned a corresponding rule.

Although individual rules are described in section 4, there are multiple layers of applying those rules and finding the optimal ruleset.

I. When each variable has a rule assigned, these can be programmatically applied one by one. Clear definition of rules allows for creating a program which assigns them based on the specification, without any manual intervention.

II. Many identifiers will have more than one potential rule to test (e.g. AGE can have the following rules: KEEP, DROP, AGE_BANDS(5-year), AGE_BANDS(10-year), etc.). By specifying those options it's possible to create code which will generate all possible rulesets.

III. With programs to apply rules to dataset (per level I) and to generate sets of rules (per level II), it makes sense to automate the process of iterating over the rulesets to transform the dataset and calculate risk for all scenarios.

The white paper includes code examples for those tasks which can help start building tools and macros to automate the process and make it more efficient.

7. ANONYMISATION REPORT

The last section shares thoughts on creating documentation around the anonymisation process. It offers high level information on what should be included although stops short of suggesting the exact contents as this will be specific to the deliverable and requirements.

PROCESS IN A NUTSHELL

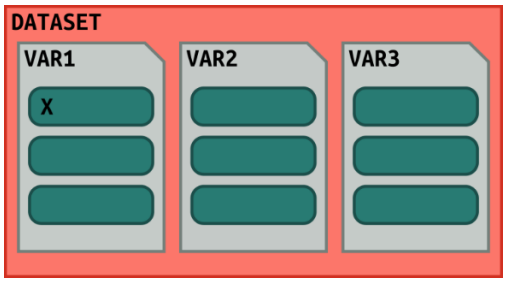
In its most condensed form, the proposed anonymisation process looks as follows (numbers in brackets are paragraphs in the white paper).

- Prepare input datasets (2.1)
- Set Thresholds (5.8)
- Assess risk of re-identification attempt – deliberate, inadvertent, breach (5.1)
- Find identifiers in the data, assign classification (3.1)
- Create BASE DATASET by extracting quasi-identifier values for all subjects (5.3)
- Create RULE DATASET by assigning possible rules to each quasi-identifier in BASE DATASET (6.3)
- For each scenario in RULE DATASET:
- Create TRANSFORMED DATASET by applying de-identification rules to BASE DATASET (6.2)
- Calculate risk of successful re-identification (5.5)
- Combine risk of success and risk of attempt to find overall risk of re-identification (5.6)
- Check k-anonymity condition (5.7)
- Store results
- Select optimal (final) rule set based on risk calculations and request requirements (6.5)
- Apply final de-identification rules to all datasets (6.7) and convert to desired formats (6.8)
- Compile anonymised package (6.9)
- Create anonymisation report and add to the package (7)

AUTOMATION

The goal of the white paper was not only to lay out the dataset anonymisation process but to identify opportunities for automation. The code examples in the document are intended as suggestions. As with most programming problems, there is usually more than one way, even within each language, to achieve something. The choice will depend on whether we want to focus on performance, flexibility, readability, as well as how it fits with the other layers of the tool we want to build.

In the following section, we will look at transforming values in a dataset (in other words – applying de-identification rules to variables). To illustrate the following problems, we will use a generic DATASET shown below. The set of rules to be applied to each variable is stored in RULESET.



1. Creating a rule

First, let's create an example rule. This will be a deliberately simple macro or function, which only takes a value as a parameter and returns a transformed value. In this case a numeric rounded to the nearest 10.

In real life rules can be more complex, take multiple parameters, or even require some pre-processing.

$X' = f(X)$

Datastep-oriented (SAS)

```
%macro rule1(var);
   &var = round(&var,10);
%mend;
```

Functional (Python, R, Julia)

```
function rule1(var):
  return round(var,10)
```

Note that the “Functional” examples in all the tables of this section use a generic pseudo code. Each language uses a different syntax, but the implementation is equivalent between all three of them.

E.g.
```
Python:     def roundup10(x):
R:          roundup10 <- function(x) {}
Julia :     function roundup10(x)
```

2. Applying a rule to a variable

The rule we just created can be applied to a single value, but to apply it to a variable, we need to add some code around it. We can create another macro/function which will be applied to:

- **a)** the variable holding the value,
  or
- **b)** the dataset holding the variable holding the value.

(a)	**(b)**
`in DATASET:` `  VAR' = f(VAR, RULE)`	`DATASET' = f(DATASET, VAR, RULE)`

Datastep-oriented (SAS)
 	
`%macro apply_rule(var, rule);`
`  %&rule(&var);`
`%mend;`
 	
`data DATASET;`
`  set DATASET;`
`  %apply_rule(VAR1, RULE1);`
`run;`
 	
Resolves to:
 	
`data DATASET;`
`  set DATASET;`
`  VAR1 = round(VAR1,10);`
`run;`
	Datastep-oriented (SAS)
	
	`%macro apply_rule(dataset, var, rule);`
	`  data &dataset;`
	`  set &dataset;`
	`  %&rule(&var);`
	`  run;`
	`%mend;`
	
	`%apply_rule(DATASET, VAR1, RULE1);`
	
	Resolves to:
	
	`data DATASET;`
	`  set DATASET;`
	`  VAR1 = round(VAR1,10);`
	`run;`

Functional (Python, R, Julia)
 	
`function apply_rule(var, rule):`
`  return var.map(rule)`
 	
`DATASET.VAR1 =`
`  apply_rule(DATASET.VAR1, RULE1)`
 	
Resolves to:
 	
`DATASET.VAR1 =`
`  DATASET.VAR1.map(round(x,10))`
	Functional (Python, R, Julia)
	
	`function apply_rule(dataset, var, rule):`
	`  return dataset.var.map(rule)`
	
	`DATASET =`
	`  apply_rule(DATASET, VAR1, RULE1)`
	
	Resolves to:
	
	`DATASET = DATASET.VAR1.map(round(x,10))`

At this stage, although the portions of the static code and the dynamic code (highlighted; coming from function) are different, the instructions to which the code resolves to are the same for **(a)** and **(b)**.
While of little significance now, the choice will affect the second level described in the next section, especially in SAS.

3. Applying all rules to a dataset

The next step is to build functionality which will apply relevant transformations/rules to each variable. In principle, this is achieved by using the ruleset info merged with the metadata of the dataset (structure, variable list, etc.), to then update the contents of dataset itself.
The implementation of that problem will be very different in datastep-oriented vs functional languages. Since we need to use metadata, data, and ruleset simultaneously, the functional languages have a huge advantage. This is because functions are “first class citizens” in those languages. This means that they can be used and passed to and returned from other function as parameters, like any other variable and object type.

<table>
<tr><th>Following from (a)</th><th>Following from (b)</th></tr>
<tr><td>
Datastep-oriented (SAS)

in DATASET:
for each VAR, RULE in RULESET:
VAR' = f(VAR, RULE)

Wrapper data step:
<pre>
data _null_;
  set RULESET end=last;
  if _N_ eq 1 then do;
    call execute('data '||dataset||';');
    call execute(' set '||dataset||';');
  end;
  call execute(
   ' %apply_rule('||var||','||rule||');'
  );
  if last then call execute('run;');
run;
</pre>
Resolves to:
<pre>
data DATASET;
  set DATASET;
  %apply_rule(VAR1, RULE1);
  %apply_rule(VAR2, RULE2);
  %apply_rule(VAR3, RULE3);
run;
</pre>
</td><td>
Datastep-oriented (SAS)

for each VAR, RULE in RULESET:
DATASET' = f(DATASET, VAR, RULE)

Wrapper data step:
<pre>
data _null_;
   set RULESET;
   call execute(
     ' %apply_rule('||dataset||','||
        var||','||rule||');'
   );
run;
</pre>
Resolves to:
<pre>
data DATASET;
  set DATASET;
  %apply_rule(VAR1, RULE1);
run;
data DATASET;
  set DATASET;
  %apply_rule(VAR2, RULE2);
run;
data DATASET;
  set DATASET;
  %apply_rule(VAR3, RULE3);
run;
</pre>
</td></tr>
<tr><td>
Functional (Python, R, Julia)

for each VAR in DATASET:
VAR' = f(VAR, RULE)

<pre>
for var in dataset:
  rule = find_rule(var)
  dataset.var =
     dataset.apply_rule(var, rule)
</pre>
Resolves to:
<pre>
VAR1 = DATASET.apply_rule(VAR1,RULE1)
VAR2 = DATASET.apply_rule(VAR2,RULE2)
VAR3 = DATASET.apply_rule(VAR3,RULE3)
</pre>
</td><td>
Functional (Python, R, Julia)

for each VAR in DATASET:
DATASET' = f(DATASET, VAR, RULE)

<pre>
for var in dataset:
  rule = find_rule(var)
  apply_rule(dataset, var, rule)
</pre>
Resolves to:
<pre>
DATASET = apply_rule(DATASET,VAR1,RULE1)
DATASET = apply_rule(DATASET,VAR2,RULE2)
DATASET = apply_rule(DATASET,VAR3,RULE3)
</pre>
</td></tr>
</table>

In SAS, this can be achieved by using macros and macro variables or by having a "wrapper" data step generate code which will be subsequently submitted (e.g. using `call execute();` statements).That generated code will be different depending on the choice we made earlier. If the subject was a variable/column – as per **(a)** – then the result will be a single data step with multiple calls to apply rules to variables. Everything happens within one dataset. For **(b)** the result will be a list of data steps, each only changing one variable at a time.

Option **(a)** is computationally more efficient, as it only needs one run through the dataset. However, any dependencies need to be considered carefully and applying the same rule to multiple variables may result in unexpected and unwanted interactions, artifacts, and problems.

Option **(b)** will remove the problem of interactions, as each variable transformation will happen in a separate data step. However, this option is computationally much more expensive as the same dataset needs to be processed as many times as there are variables.

In the functional languages, although the application of **(a)** and **(b)** is slightly different, they are computationally equivalent. The choice is effectively down to preference and tools used.

4. Testing multiple scenarios (rulesets)

Once the code in step 2 (regardless of language and option) is converted to a parametrized macro/function, it can then be used in this final stage. Given multiple rulesets, each needs to be applied and tested. The results of risk calculation are stored in a dataset/container, and choosing the optimal scenario can be either manual, or based on pre-programmed rules and assumptions.

The way to apply it programmatically will be very similar to step 2. We can iterate over multiple rulesets, running the macro/function for each of them.

```
for each RULESET:
    DATASET' = apply_all_rules(DATASET)
    calculate_risk(DATASET') → RISKS
```

The code for calculating risk on a dataset is provided in the white paper in all four languages.

PROGRAMMING LANGUAGES USED IN THE WHITE PAPER – AUTHOR'S VIEW

DISCLAIMER

The white paper offers code examples in four programming languages – SAS, Python, R, and Julia.

My relationship with each of those is different. I have been programming in SAS since 2000. I am fluent in SAS and capable of solving most problems I may need to. I am a Python enthusiast and have been learning different aspects of it over the last few years. It is still a trial and error kind of programming and StackOverflow is never far away. R is a language I am aware of and hear about rather a lot. I may be able to write simple functions but find the syntax hard to read and follow. I have come across Julia while working on the white paper. I have only scratched the surface of what Julia is capable of, but I am curious enough to want to learn more of it and about it. I believe it has a lot of promise in the data science landscape.

DIFFERENCES

Availability

SAS is available through licensing with pricing aimed at institutions rather than individuals. In simple terms this means that anyone who hasn't got access to a commercial license (through their employer or organization) will not find it easy to try and learn it or test features which are not covered by the license they happen to use.

Python, R, and Julia are all open source. The "base" versions can be installed and used for free and each has an extensive community support and multitude of packages which help solve different problems. Those are almost always also free to use. Python is a general-purpose language which can be used for many different purposes. From web design and API backend, to desktop apps and even microcontrollers. The most popular modules focused on data processing are Numpy and Pandas. R is a free software environment for statistical computing and graphics. Data manipulation capabilities are one of its core functionalities. Julia is the youngest of the four. Developed by MIT, it is also data processing orientated and is focused on readability and performance.

Logic

SAS uses a completely different paradigm than the other three languages. All logic is largely bound to a data step or a proc step. Any information needed to manipulate a variable must be brought into the dataset beforehand, in the form of more variables, mainly by joining or merging datasets. This can be extended by hash tables and macro variables, but these have challenges of their own. The former are not very intuitive, the latter are quite limited (e.g. there are no macro arrays, only singular macro variables which need their values explicitly assigned).

Python can be used as either object-oriented or functional. Data and objects can be stored on environment, class, or method/function level and can be mixed as necessary. E.g. a dictionary/hash table can be dynamically created, truncated, or extended before being applied to a data frame without the need to merge the two explicitly.

R and Julia are functional languages. Similarly to Python, they can combine information from different types of objects.

Python, R, and Julia treat functions as so called "first class citizens". This means that functions are treated like any other object and can be passed to and returned from another function as an argument. This makes the languages very flexible and opens possibilities which are not easy to emulate in SAS.

Native Data Files

SAS uses the native Dataset, which is the basis for data processing and all programming philosophy revolves around the Dataset. It is also a self-contained file. It consists of variables which can be either character or numeric. Formats can be applied to variables to show decodes of values. R and Julia's comparable construct is a Dataframe. Although Python doesn't have a native format equivalent to the above, a popular and widely used module Pandas introduces a

DataFrame object and methods for processing and transforming it, similar to the other languages. Dataframe is an object in the programming environment, but not a file format.
In SAS, the majority of processing is done on and to a Dataset, within a data step or a proc step. The open source languages have a wide range of objects and variable types (string, float, integer, list, dictionary, etc.) available outside of the Dataframe context.

Working with Other Files
SAS is at its best when dealing with native datasets. Bringing other formats into the environment or exporting to them can be unnecessarily challenging. When importing CSV files, SAS insists on being useful by "guessing" which variable should be assigned what type and format (e.g. dates). While this may work in some scenarios, there is no easy way of "switching it off" when it doesn't. Parsing other formats, which are becoming increasingly popular is either not natively possible or available via licensing additional modules, which may not be possible in many settings. Parsers for those text-based formats need to be programmed from scratch through multitude of concatenating, substringing, counting quotation marks, etc.
Another popular format - Sqlite file can only be accessed via ODBC. This is both an archaic method and not available is base SAS installation.
In contrast, for all three open source languages dealing with external files is either native or achieved by using widely available modules. Not only CSV, but other very popular formats like JSON, YAML, XML or even SVG, etc. can be easily accessed and generated without the need to create new parsing tools. This makes reading configuration files or even data files much easier and flexible.

OVERALL
How do the findings affect the implementation of tools, functions, and macros in the context of anonymising datasets and specifically regarding the code examples provided in the white paper?
All actions and tasks described in the paper can be achieved in all languages. However, how we get from A to B can differ greatly in some instances. As with most programming problems, there is often more than one way, even within each language, to achieve something. The choice of the language will most likely be driven by what tools are available and allowed in the organization. The choice of the algorithm will depend on whether the focus is on performance, flexibility, readability. Not surprisingly, it is important to understand how the application of one stage affects the subsequent ones. What seems a better option in isolation may be least optimal as part of a bigger tool.

CONCLUSIONS AND NEXT STEPS
Anonymising datasets is a complex task and involves a thorough understanding of the process. Having a comprehensive reference which provides as much detail as possible on individual tasks, considerations, and pitfalls helps to build the process, take advantage of the community's expertise, and allows to identify tasks which can be successfully automated
The team working on the white paper believes that it is the comprehensive document, reviewed by the community and accepted as the base for the dataset anonymisation process, which the team had set out to be.
Any programming language with the capability to work with data structures will allow implementation of concepts and processes described in the white paper. Often, the decision of which to choose will depend on the environment of the user and their knowledge of the tools. Just like with human languages, it is always worth being familiar with more than one.

REFERENCES
PHUSE White Paper: *Data Anonymisation and Risk Assessment Automation*
Version 1.0, 01-Jul-2020
(http://phusewiki.org/docs/WorkingGroups/Deliverables/Data%20Anonymisation%20%26%20Risk%20Assessment%20Automation%20-%20Data%20Transparency%20.pdf)

Kniola, L (2016). *DH09 Calculating the Risk of Re-Identification of Patient-Level Data Using Quantitative Approach.* PhUSE Annual Conference, 2016.

Kniola, L (2019). *PP24 Data Anonymisation & Risk Assessment – Process Map and Automation Efforts.* PhUSE Annual Conference, 2019.

Institute of Medicine. (2015). *Sharing Clinical Trial Data Maximizing Benefits, Minimizing Risk.* Washington, DC: The National Academies Press.

Pharmaceutical Users Software Exchange (PhUSE). De-identification standards for CDISC SDTM 3.2. (https://www.phuse.eu/documents/working-groups/data-transparency/de-identification/PHUSE-de-identification-standard-for-sdtm-32-version-101-21300.xls)

The R Project for Statistical Computing (https://www.r-project.org/)

Welcome to Python.org (https://www.python.org/)

The Julia Language (https://julialang.org/)

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Lukasz Kniola
Biogen Idec Ltd
Innovation House
70 Norden Road
Maidenhead, Berkshire SL6 4AY
UK
Email: lukasz.kniola@biogen.com