MWSUG 2024 - Paper PH99

From Paper To Pedigrees: A Beginners Approach with SAS®

Vijay Susmitha Bommini, Marshfield Clinic Health System, Marshfield, WI

ABSTRACT

Hospital paper records data differ significantly from Electronic Health Records (EHR) data in terms of their format, structure, maintenance, and accessibility. Paper records are typically unstructured, and manually entered which often leads to errors in the entry of information and makes it difficult to obtain complete and accurate data. This paper explores how SAS® software was effectively used to transform these unstructured hospital paper records into a structured format as part of the "Utility of Family Data Extracted from Hospital Records" project at Marshfield Clinic Research Institute. This involved a series of steps, from abstracting the paper records data into REDCap© electronic form followed by comprehensive data conversion, cleaning, and transformation steps utilizing SAS. As a beginner in SAS, I utilized the SAS® Enterprise Guide, focusing on the SQL Procedure, DATA steps, and conditional logic to manage data extraction, cleaning, and transformation. The goal was to create actionable and reliable datasets from hospital paper records information compliant with the **Family Mapping Algorithm** (FMA) (Huang, et al., 2021), which establishes familial relationships using demographic and contact information. This paper is for beginners and aims to provide practical insights and offer a pathway into how SAS was used for transforming valuable paper record information into structured data suitable for analysis that can be further used as needed for various purposes.

INTRODUCTION

Family histories serve as a valuable resource for identifying and managing disease risks in genetic research. Traditional methods of obtaining information on family histories, however, are cumbersome. Previous studies revealed the use of Electronic Health Records (EHR) data (Huang, et al., 2018) to generate E–pedigrees using the **Family Mapping Algorithm** (FMA) (Huang, et al., 2021). While EHRs offer standardized data collected through pre-defined formats, they often lack historical depth in time periods prior to EHR implementation. To help fill such historical gaps, the project **"Utility of Family Data Extracted from Hospital Records"** conducted as a part of the Summer Research Internship Program (SRIP) at Marshfield Clinic Research Institute (MCRI) focused on utilization of family data from Marshfield Clinic's hospital paper records. Although paper records contain a wealth of longitudinal, family health data that predated the EHR, they present significant challenges for utilization. The purpose of this project is to create actionable family history datasets from paper records for use in the FMA.

This paper discusses the use of **SAS** to transform unstructured hospital paper records into structured, actionable datasets. The paper also provides detailed explanation of how the SQL Procedure, DATA steps, macros and other tools in SAS were used for data extraction, cleaning, and transformation. Practical examples and scenarios with code will be provided to illustrate the process of how key challenges were addressed in Hospital paper records information transformation, making it an accessible guide for SAS beginners.

AUTOMATIC EXTRACTION OF FILES FROM REDCAP USING MACROS

In the process of extracting paper records from **REDCap**, **SAS macros** were used to automate the extraction. This approach is particularly valuable when working with large datasets, as it minimizes manual effort and ensures consistency in data extraction, making it a powerful tool in large-scale healthcare data projects. The export macro used in this project is available, but it is not a built-in function of SAS. For a comprehensive explanation and use of the macro, please refer to my mentor's recent paper published at SESUG (Delgoffe & Roush, 2022).

The macro illustrated below is designed to extract records from different files, specifying the required file name, list of fields as needed, and output dataset name. This also ensures data confidentiality as it uses a unique token ID specific to each individual:

```
%RedCapAPIExportReportLimited(External=0
                 ,mytoken=/XXXXXXXXXXXXXXXXXXX/
                 ,BaseDir=%bquote(&folderpath.\DATA)
                 ,FieldList=%bquote(record_id, document_date, birth_record,
hospital_id, mhn, first_name, middle_initial, last_name, maiden_name, dob)
                 ,CSVName=HUME Patient Information All
                 ,SASDataName=ALL_PATIENT_RAWDATA
                 ,FormatDataName=%str()
                 )
;
```

THE 'PROC SQL' PROCEDURE AND DATA STEPS FOR DATA PRE-PROCESSING

In this project, we extensively utilized a combination of PROC SQL and DATA tools for various pre-processing steps.

UNLOCKING DATA TRANSFORMATION WITH PROC SQL

PROC SQL allowed us to perform Exploratory Data Analysis (EDA), understanding the structure of different files, and also identify underlying patterns in individual files. Using this approach, we performed efficient joins of multiple tables, filter records using specific conditions as per the requirement (Ex – WHERE, case when) along with tasks that require queries and aggregations.

For Merging Datasets

The below PROC SQL step is used to join two individual files/datasets (in this case, joining adult patient records with baby patient records) and create a new dataset:

```
PROC SQL;
CREATE TABLE PATIENT_RAWDATA as
      SELECT
           a.record_id as record_id
             ,a.first_name as first_name
             ,a.last_name as last_name
             ,a.middle_initial as middle_initial
             ,a.unique_id as unique_id
             ,a.dob_year as dob_year
             ,a.dod_year as dod_year
             ,a.mhn as mhn
      from updated_mchs_patients_1 AS a
      UNION ALL
      SELECT
           b.record_id as record_id
             ,b.baby_firstname as first_name
             ,b.baby_lastname as last_name
             ,b.baby_middleinitial as middle_initial
             ,b.unique_id as unique_id
             ,b.bdob_year as dob_year
```

```
            ,b.bdod_year as dod_year
            ,b.baby_mhn as mhn
      from BABY_PATIENT_RAWDATA_ID AS b;
quit;
```

The **SELECT** statement is utilized to extract the necessary columns from two datasets. In this case, the **UNION ALL** operation is applied, which combines all rows from both **SELECT** statements, including duplicates. This method ensures comprehensive data merging without excluding any duplicate records. Additionally, aliases are used to reference the datasets, simplifying the code and improving readability. The use of these SQL features in SAS effectively handles large datasets while maintaining clear and concise code.

Smart Data Merging Based on Conditions

Another effective use of PROC SQL step we employed is combination of conditional transformation and merging of datasets based on specific criteria:

```
PROC SQL;
     CREATE TABLE merged_contacts_EC1 as
     SELECT a.record_id,
             case when b.ec_phonesame_1___1 = 1 or ec_phonesame_1___2 = 1
then a.phone_number else ' ' end as phone_number,
            b.*
      from test_ac_updated as a
      left join EC1_FILLDATA_UPDATED as b
      on a.record_id = b.record_id;
      quit;
```

In this example, we integrated multiple SQL techniques, including **CREATE TABLE**, **SELECT statement**, **conditional transformation**, **aliasing**, and **LEFT JOIN**, to efficiently manipulate and transform data.

The **conditional transformation** is achieved through the **CASE WHEN** statement, which conditionally modifies the 'phone_number' field. Specifically, if either 'ec_phonesame_1___1' or 'ec_phonesame_1___2' equals 1 in dataset b, the values of 'phone_number' from dataset a are retained. This step ensures the proper transformation of fields based on predetermined criteria.

The **LEFT JOIN** is utilized to combine all rows from 'dataset a' with matching records from 'dataset b', based on the 'record_id' field. This operation ensures that all records from 'dataset a' are included, along with corresponding data from 'dataset b'. If there is no match in 'dataset b', the result will include null values for those fields. This technique is particularly useful when combining datasets from multiple sources, ensuring no loss of critical data from the primary dataset.

This process enables the creation of new fields while allowing conditional transformations within the dataset.

Placeholders in Action: Filling Data Gaps with PROC SQL

Another way we explored in using PROC SQL is effective utilization of placeholders in cases where the datasets being combined are of **no same structure** with different variables:

```
PROC SQL;
CREATE TABLE AC_EC1 as
      SELECT
            a.redcap_repeat_instrument as type
```

```
                ,a.redcap_repeat_instance as rr_instance
                ,a.record_id as record_id
                ,a.sex_at_birth as gender_code
                ,a.first_name as f_name
                ,a.middle_initial as m_name
                ,a.last_name as l_name
                ,a.document_date as doc_date
                ,a.dob as dob
                ,a.dod as dod
                ,a.contactinfo_date as contactinfo_date
                ,'' as ec_date  /* Using place holder for column ec_date as it
is not included in the dataset test_ac_updated. */
                ,a.phone_number as phone_number
                ,a.unique_id as combined_id
        from test_ac_updated AS a
        UNION ALL
        SELECT
             b.redcap_repeat_instrument as type
               ,b.redcap_repeat_instance as rr_instance
               ,b.record_id as record_id
               ,'' as gender_code   /* Using place holder for column
gender_code as it is not included in the dataset EC1_PHONE_FILLED. */
               ,b.ec_firstname_1 as f_name
               ,b.ec_middleinitial_1 as m_name
               ,b.ec_lastname_1 as l_name
               ,'' as doc_date   /* Using place holder for column doc_date as
it is not included in the dataset EC1_PHONE_FILLED. */
               ,'' as dob     /* Using place holder for column dob as it is not
included in the dataset EC1_PHONE_FILLED. */
               ,b.emergencycontact_date as ec_date
               ,b.ec_phonenumber_1 as phone_number
               ,b.unique_id as combined_id
        from EC1_PHONE_FILLED AS b
order by record_id, combined_id;
quit;
```

The above **PROC SQL** approach allows the inclusion of **placeholders,** ensure that the combined dataset maintains a consistent structure and format, even when some variables are missing from one of the datasets. This method guarantees that the resulting dataset includes all variables from both source datasets, regardless of their presence in individual datasets. By doing so, we can maintain data integrity and ensure uniformity in the structure, making it easier for subsequent data transformations and analysis.

Aligning Datatypes: Merging Datasets with Different Datatype Variables

We also effectively combined dataset by UNION ALL with variables of different datatypes (CHARACTER/NUMERIC) that are combined as one to ensure consistency in the resulting dataset:

```
PROC SQL;
CREATE TABLE ADULT_BM_ADDRESS as
      SELECT
           record_id as record_id
             ,ci_year as ci_year
             ,'' as Mdoc_year
             ,'' as Mdob_year
             ,street_1 as street_1
             ,street_2 as street_2
             ,city as city
             ,state_abbr as state
             ,zip_5 as zip
             ,unique_id as unique_id
      from ADULT_TEST_ADDRESS_CIYEAR AS a
      UNION ALL
      SELECT
           b.record_id as record_id
             ,'' as ci_year
             ,b.Mdocument_year as Mdoc_year
             ,b.Mdob_year as Mdob_year
             ,b.mom_street as street_1
             ,'' as street_2
         ,b.mom_city as city
             ,b.mom_state as state
             ,INPUT(b.mom_zip, BEST32.) as zip
             ,b.unique_id as unique_id
      from BABYMOM_TEST_ADDRESS_ID2 AS b
order by record_id, unique_id;
quit;
```

To achieve data type conversions, we employed the **PUT** (numeric to character) and **INPUT** (character to numeric) functions in the **PROC SQL** step. In the example provided above, the **INPUT** function was applied to the variable 'mom_zip' from dataset b to convert it from character to numeric format using **BEST32.** formatting, ensuring that the combined dataset treated 'zip' as numeric. Similarly, the **PUT** function can convert numeric variable type into character format, ensuring that both datatypes can be handled seamlessly within the dataset.

Null No More: Efficient Filtering with COALESCE in WHERE Clauses

In data pre-processing, we are often bound to eliminate rows/records that are incomplete/null across multiple variables to reduce the missingness of data in the final dataset.

Here we utilized COALESCE with WHERE in PROC SQL to eliminate rows/records that has no value across all the specified variables:

```
PROC SQL;

CREATE TABLE NAMES_SAMPLE AS

SELECT *

FROM NAMES_AECBMD3

WHERE COALESCE(first_name, last_name, middle_initial) not in ("","."") /*
excludes records where all the mentioned variables are missing/null or has a
period(.) */

      or COALESCE(dob_year, dod_year) not in(.); /* excludes records where
all the mentioned variables are missing.  */

quit;
```

The 'Coalesce' looks for non-missing values from left to right order of the specified list of variables. Along with this, the code also addresses the problem of handling records with variables of different data types (Numeric/Character).

SELECT *: Selects all columns from the source dataset.

WHERE statement with COALESCE and NOT IN:

Here, the variables (first_name, last_name, middle_initial) belong to CHARACTER datatype and the variables (dob_year and dod_year) are of NUMERIC datatype. This condition uses 'COALESCE' to check columns in the mentioned order for empty value or a period in case of CHARACTER variable type and missing numeric value in case of NUMERIC variable type and eliminate those records.

This combination of handling different data types and using the COALESCE function allows the code to effectively filter out records that contain no meaningful data in any of the specified fields, ensuring that only useful records are retained for analysis. This approach is particularly useful when dealing with missing data for required fields in large datasets.

Quantifying the data with PROC SQL

PROC SQL is also used for generating descriptive counts based on various conditions applied to the dataset:

```
PROC SQL;

select 'Number of Records in the Emergency contacts file' as Description

      ,count(distinct ec_study_id) format=comma9. as Count

      , . format=comma9. as DENOMINATOR

      ,. format=percent. as Percent

      from EC_draft

union all

select '-- excluding records where relationship is missing' as Description

      ,count(distinct ec_study_id) format=comma9. as Count

      ,(select count(distinct ec_study_id) from EC_draft )
```

```
      ,count(distinct ec_study_id) / (select count(distinct ec_study_id) from
EC_draft ) format=percent. as Percent
      from EC_draft_updated
union all
select '-- excluding records where relationship defined is not biological' as
Description
      ,count(distinct ec_study_id) format=comma9. as Count
      ,(select count(distinct ec_study_id) from EC_draft_updated)
      ,count(distinct ec_study_id) / (select count(distinct ec_study_id) from
EC_draft_updated ) format=percent. as Percent
      from EC_draft_updated3;
quit;
```

This PROC SQL code is used to count and summarize records from different stages of cleaning the "Emergency Contacts" data. The first SELECT statement counts all unique ec_study_id values from the initial dataset. The subsequent statements refine the counts by excluding records where key information is missing or irrelevant. The UNION ALL combines the results from each condition, and the code also calculates percentages to show the proportion of remaining records compared to the original dataset.

Description	Count	Percent
Number of Records in the Emergency contacts file	103,847	.
-- excluding records where relationship is missing	103,321	99%
-- excluding records where relationship defined is not biological	65,471	63%

Output 1. Output for Descriptive Statistics

This approach quantifies how much data is affected by each exclusion criteria, which is essential for understanding the completeness and relevance of the dataset for further analysis.

In this way we can leverage PROC SQL by integrating different functions and conditional statements to perform various data cleaning and pre-processing operations. The final dataset thus obtained is more reliable, complete, and cleaned.

DATA STEP MAGIC: CRAFTING DATASETS FROM RAW TO REFINED

Besides, PROC SQL another package we used are numerous DATA steps which played a significant role for various data cleaning, complex transformations, and creation of several different sub-datasets for effective analysis.

Converting to SAS date Using DATA

In the below DATA step, we used INPUT function to convert the variables document_date, dob, dod into SAS date format for the program to read the values in the respective fields.

In addition, we also extracted just the year as distinct columns from respective variables which are in yyyy-mm-dd form originally:

```
DATA updated_mchs_patients;
   SET ADULT_PATIENT_INFO;
   document_date2 = input(document_date, yymmdd10.);
   dob2 = input(dob, yymmdd10.);
```

```
   dod2 = input(dod, yymmdd10.);


   document_year = year(document_date);
   dob_year = year(dob2);
   dod_year = year(dod2);
RUN;
```

Optimizing Address: Using Conditional Logic and Update Fields

In SAS, the DATA step also provides a way to implement conditional logics to update the values of variables using specific conditions. This is especially useful in case of transformation tasks involving multiple variables:

```
DATA ADULT_TEST_ADDRESS_COL;
     SET ADULT_TEST_ADDRESS;
       if street_1 in(" ",'NI') and care_of_location not in('NI'," ") then do
street_1 = care_of_location;
END;
RUN;
```

The data step above creates a new dataset named ADULT_TEST_ADDRESS_COL from SET data. The if statement specifies two conditions:

- Checks the variable street_1 if it is empty or having a value 'NI' (No Information)
- Then moves to check AND condition if the variable care_of_location is neither empty nor 'NI'.

If both the above conditions are met, then the new dataset has an updated value of the variable street_1 with the value of care_of_location.

Optimizing Address: Handling different datatypes

In the above scenario we dealt with variables that are of same data type. But it is quite common to encounter this requirement with different data type variables.

In this case, we use PUT and INPUT functions (as mentioned already in PROC SQL step – Aligning Datatypes: Merging datasets with different datatype variables) to efficiently update the variables as required:

```
DATA EC1_ADDRESS_UPDATED;
SET merged_address_EC1_updated;
if ec_street_1 in(" ",'NI') and street_1 not in('NI'," ") then do ec_street_1
= street_1;
end;
if ec_street_1 in(" ",'NI') and street_2 not in('NI'," ") then do ec_street_1
= street_2;
end;
if ec_city_1 in(" ",'NI') and city not in('NI'," ") then do ec_city_1 = city;
end;
if ec_state_1 in(" ",'NI') and state_abbr not in('NI'," ") then do ec_state_1
= state_abbr;
end;
```

```
if ec_zip_1 in(" ",'NI') and zip_5 not in('NI'," ") then do ec_zip_1 =
INPUT(zip_5, BEST32.);
END;
RUN;
```

If and else in DATA

The below example illustrates to calculate AGE as of individuals date of death year(dod_year) or if the individual is alive or dod_year is missing:

```
DATA updated_mchs_patients_1;
SET updated_mchs_patients;
current_year = year("01JAN2000"d); /*use date literal to cap the value*/


if dod_year NE . then age = dod_year - dob_year;
else age = current_year - dob_year;
RUN;
```

Firstly, we define the current year that can be used for calculations. Then we used if and else statements to specify the conditions to be performed.

Conditional Statement:

- if there is a value in dod_year, uses the same for calculating age.
- if the value of dod_year is null then uses defined current year to calculate age as date literal mentioned.

What if AGE is known?

What if age of the individual is known instead of the date of birth and date of death years with the date of documentation :

```
DATA BABY_MOM_RAWDATA_FILTERED_2;
SET BABY_MOM_RAWDATA_FILTERED_1;
current_year = year(mdocument_year);
if mdob_year NE '.' then mdob_year = mdob_year;
else mdob_year = mdocument_year - mom_age;
format mdob_year 4.;
RUN;
```

In this case we defined the current_year as year of documentation (mdocument_year).

The code processes the BABY_MOM_RAWDATA_FILTERED_1 dataset to update the mdob_year (mother's date of birth) in the new dataset BABY_MOM_RAWDATA_FILTERED_2.

Condiitonal Statement:

- if mdob_year is not missing, it keeps its value.
- if missing, it estimates mdob_year by subtracting the mother's age (mom_age) from the document year(mdocument_year).
- it also extracts year only from mdocument_year and formats the result in a four digit year format(4.).

This process ensures that the mother's date of birth (mdob_year) is populated either from existing data or by estimation using her age and the document year.

Data Management Using CATX() & KEEP()

When dealing with patient records, creating a unique identifier for each individual is extremely useful. This helps in easy identification of data specific to individuals as the data might be coming from across different sources or departments.

In this context, we used CATX function in data step to create a new unique_id combining multiple pieces of information across different columns:

```
DATA ADULT_TEST_ADDRESS_ID;
     SET ADULT_TEST_ADDRESS1;
       if redcap_repeat_instrument = 'contact_information' then
redcap_repeat_instrument = 'c';

      unique_id = catx('.',record_id,redcap_repeat_instrument);
run;
```

In the above code, we efficiently combined the columns record_id, redcap_repeat_instrument, redcap_repeat_instance with the specified delimiter '.' This ensures that the output column 'unique_id' has a consistent format with components separated by '.'.

record_id	record_source	repeat_instance	unique_id
7	baby		7.b
7	baby_mom		7.bmi
7	baby_dad		7.bdi
8	adult/child		8.c
8	emergency contact 1	1	8.ec1.1
8	emergency contact 2	1	8.ec2.1
8	emergency contact 2	2	8.ec2.2
8	emergency contact 3	1	8.ec3.3

Figure 1. CATX() function use case output

We also employed various DATA steps using KEEP functions to create new datasets.

KEEP is used to extract only specific required variables from the source dataset:

```
DATA BABY_MOM_RAWDATA_FILTERED;
SET BABY_MOM_RAWDATA1 (KEEP =record_id document_date mom_firstname
mom_middleinitial mom_lastname mom_dob mom_age unique_id);
run;
```

Creating TimeFrames (From and Thru Years)

Creating timeframes by generating new variables that represent specific time periods based on existing year variables is useful for analysis over time. The FMA algorithm requires all the input data validated across non-overlapping timeperiods.

Hence, we have tailored FROM_YEAR and THRU_YEAR using existing year variables information and assigning the most relevant based on the source data by utilizing DATA step:

DATA DEMO_AECBMD3;
SET DEMO_AECBMD2;
if contactinfo_year NE '' then
from_year =contactinfo_year;
 	
else if ec_year NE '' then
from_year = ec_year;
 	
else if doc_year NE '' then
from_year = doc_year;
 	
else if birth_year NE '' then
from_year = birth_year;
 	
else if birth_year < 1925 then
from_year = 1925;
 	
else from_year = 1900;
 	
RUN;
a. condition1: we are creating a from year using contactinfo_year if not null.
b. condition2: we are stating to use ec_year as from_year if contactinfo_year is null.
c. condition3: else if we state to use doc_year as from_year if both contactinfo_year and ec_year are null.
d. condition4: We say if all the above 3 conditions doesn't work (contactinfo_year, doc_year and ec_year are null)set default value of from_year to 1900.
	DATA DEMO_AECBMD4;
	SET DEMO_AECBMD3;
	if deceased_year NE '' then
	thru_year = deceased_year;
	
	else if contactinfo_year < 1925 then thru_year = 2000;
	
	else if ec_year < 1925 then
	thru_year = 2000;
	
	else if doc_year < 1925 then
	thru_year = 2000;
	
	else thru_year = year(today());
	
	RUN;
	a. condition1: we are creating a thru year using deceased_year if not null,
	b. condition2: we are stating to use thru year as default to 2000 if contactinfo_year&ec_year&doc_year < 1925,
	c. condition3: We say if all the above conditions doesn't work (deceased_year is null and contactinfo_year&ec_year&doc_year > 1925)set default value of thru_year to current year.

Cleaning Data Using COMPRESS AND UPCASE with DATA

COMPRESS and **UPCASE** functions are used to standardize data fields such as names and phone numbers, essential for ensuring consistency in data analysis.

COMPRESS is employed to strip unwanted characters, like spaces, punctuation, or special symbols, from fields such as phone numbers (‘kda’ – keep only digits & alphabets) or names (‘ka’ – keep only alphabets):

```
DATA cleaned_phones;
SET demo_draft_updated;
```

```
cleaned_primary_phone_id = compress(PRIMARY_PHONE_ID, , 'kda');
cleaned_primary_phone_id = upcase(cleaned_primary_phone_id);
run;
```

UPCASE transformed all letters to uppercase for uniformity. For instance, in names, the combination of these functions helped remove unnecessary characters and ensure that all names were uniformly structured and standardized.

```
DATA upcase_names;
SET names_draft;
last_name = compress(upcase(last_name), , "ka");
first_name = compress(upcase(first_name), , "ka");
middle_name = compress(upcase(middle_name), , "ka");
run;
```

This approach ensures cleaner datasets and facilitates the accurate identification of duplicates or related records, which is crucial in our efforts to standardize and transform unstructured hospital data into actionable information.

De-identifying Data with RETAIN()

In this particular step, we utilized **RETAIN()** to maintain and assign unique identifiers for each sensitive variable such as names, phone numbers, and addresses to ensure consistency and privacy during the de-identification process:

```
DATA last_names_id;
SET sorted_last_names;
by last_name;

RETAIN last_name_ID 0;

if first.last_name then do;
   if last_name = " " THEN
   last_name_ID = -1;
   ELSE last_name_ID +1;
end;
run;
```

After cleaning the data, when we declare a variable with RETAIN, it helps to keep its value from one iteration to the next instead of starting anew each time. In simple words, it will carry over its value.

In the above code, variable ‘last_name_id’ starts at value ‘0’ and when moving through observations in the dataset ‘sorted_last_names’ the value of the variable is updated based on the condition inside ‘do’ block.

The ‘if first.last_name’ means the code is executed only when a new last name is executed, that is if the value is the first occurrence of a last name, it simply increments the value by 1 or otherwise retains the same value as the previous one.

In summary, RETAIN is like saying, “Remember this value for the next time if you encounter.” It is particularly useful in scenarios where we have to keep track of the number of unique values you have within a specific variable across multiple rows of data in the dataset.

This mapping produces clean datasets with numeric identifiers that can be used in subsequent algorithms, maintaining data integrity while protecting sensitive information.

Raw Data

study_id	last_name	first_name	middle_name
1.c	doe	John	
1.ec1.1	doe	Jane	M.
2.c	kat	Kit	

Mapped (Algorithm Input)

study_id	last_name	first_name	middle_name
1.c	0	0	-1
1.ec1.1	0	1	0
2.c	1	2	-1

Output 2. De-identified Data

CONCLUSION

This paper demonstrates how **SAS** was utilized to transform unstructured hospital paper records into structured datasets ready for analysis. By leveraging **PROC SQL**, **DATA steps**, and other SAS features, we successfully cleaned and transformed paper records data for use in the **Family Mapping Algorithm (FMA)**. This process highlights the power of SAS in handling complex healthcare data and also acts as a guide for beginners to apply similar methods in future projects. Looking forward, leveraging SAS can be further enhanced by integrating more advanced techniques, improving data quality for modern healthcare applications, ultimately improving patient outcomes.

REFERENCES

Huang, X., Elston, R. C., Rosa, G. J., Mayer, J., Ye, Z., Kitchner, T., Brilliant, M. H., Page, D., & Hebbring, S. J. (2018). “Applying family analyses to electronic health records to facilitate genetic research.” *Bioinformatics (Oxford, England)*, *34*(4), 635–642.

Huang, X., Tatonetti, N., LaRow, K., Delgoffee, B., Mayer, J., Page, D., & Hebbring, S. J. (2021). “E-Pedigrees: a large-scale automatic family pedigree prediction application.” *Bioinformatics (Oxford, England)*, *37*(21), 3966–3968.

Delgoffe, B. E., & Roush, S. (2022). “REDCap: Your SAS Friend for EHR Manual Abstraction.” *Southeast SAS Users Group Conference*. Mobile, AL: SESUG.

CODE AND MATERIALS

To get access to SAS export macro used in this project and code for REDCap, developed by my mentor, Brooke E. Delgoffe, please use the below REDCap© survey link. This ensures that most recent versions along with documented updates and previous versions are accessible.

https://redcap.link/REDCapIsMySASFriend

ACKNOWLEDGMENTS

I would like to express sincere thanks to my mentor Brooke Ellen Delgoffe, for her support and guidance throughout my summer internship at Marshfield Clinic Health System. Her patience in helping me learn and use SAS for my project, especially as a beginner to this language has been valuable. Thank you, Brooke for helping me boost my technical skills and being an inspiring mentor.

I would also like to extend my gratitude to my PI, Dr. Scott J. Hebbring, for his guidance and leadership throughout this project. I also want to thank the Summer Research Internship Program at the Marshfield Clinic Research Institute (MCRI) for providing me with the opportunity to be a part and work on this project. To learn more about SRIP program and other research initiatives at MCRI, please visit https://www.marshfieldresearch.org/srip

RECOMMENDED READING

- *Base SAS® Procedures Guide*: SAS SQL Procedure, DATA Step, Formats and Informats

CONTACT INFORMATION

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

Name: Vijay Susmitha Bommini
Company: Marshfield Clinic Health System
Phone (Home): 906-767-2889
Email (College): vbommini@mtu.edu
Email (Personal): vijaysusmitha2000@gmail.com
College: Michigan Technological University