How Creating a Pipeline for Automatically Analyzing and Sharing Paradata Facilitated the Ability to Make Data-Driven Adjustments to Improve Data Collections

Elise Alstad, Erdal Kilicdogan, Sara Grimstad, Katharina Rossbach, Gezim Seferi, Marta Krawczynska, Statistics Norway

1. Abstract

Paradata has proven key to managing dynamic data collections, as we can use the paradata to identify and measure nonresponse and measurement errors. However, paradata contains large amounts of unstructured data, and it can be resource-intensive to extract and structure the data. We therefore found that we needed a robust and efficient process for extracting, cleaning, storing, and analyzing paradata. Using Python with Google Cloud Storage services, we created a data pipeline that synchronizes paradata from Blaise daily, parses it, and stores it in the cloud. By having up-to-date paradata, we can gain insightful information about the data collection process as it progresses. For instance, using Audit Trail data with sample data, we assess nonresponse bias for different demographic groups. Likewise, by using Dial History data, we analyze the outcome of each call and initiate measures based on their effectiveness. To make information about the data collection available to all stakeholders, irrespective of coding experience, results from the paradata analysis are shared to an internal web page that is updated daily. By creating an automatic pipeline that allows colleagues to evaluate the data collection process, we have made it easier to make data-driven decisions that adjust for bias and measurement errors. Because we use Python, which is an open-source programming language, aspects of our pipeline can be implemented by others without any cost. Similarly, we hope that sharing the journey of how we created the pipeline and the benefits we saw from it can be useful for other Blaise users.

2. Introduction

Data collection demands significant resources in terms of both cost and time, and it is therefore important to understand how to improve data collection processes. Paradata can be used to understand respondent behavior and enhance survey questionnaires to gain higher survey quality, as well as to monitor and identify areas for improvement to effectively manage data collections dynamically (Kreuter, Couper, & Lyberg, 2010). At Statistics Norway, we have used paradata from Dial History and Audit Trail, and we will refer to these data sources when we use the term paradata. These data sources include all call records, actions that the interviewer and respondents perform in the survey, and the timestamps for these actions. Post-survey analyses of paradata will help us understand how to improve the data collection process for next time. But by using real-time, or nearly real-time, paradata, we can help stakeholders to make changes during data collection in ongoing surveys (Schouten, Peytchev, & Wagner, 2017). While there are numerous uses of paradata (see Hunt, 2016; Cheung et al., 2016; Kreuter, Couper, & Lyberg, 2010) at Statistics Norway, we have focused on developing the use of paradata in two main areas:

- Improving survey questionnaires by understanding respondent behavior in surveys.
- Dynamic management of surveys by monitoring and adjusting for nonresponse bias.

When we started using paradata, the initial step was to export paradata from Blaise, and here we noticed challenges concerning the sheer volume of data and database access. Querying Audit Trail data from large surveys demanded a substantial share of the server capacity, resulting in slower loading time for interviewers. Secondly, we faced challenges related to database access because colleagues outside of the Blaise developer team required access to the paradata. However, as the database with the schemas for Audit Trail also included other schemas, it was unideal that too many individuals would be given access to the database. Subsequently, only a limited number of colleagues were given access and could export Audit Trail and Dial History data. Moreover, the individuals who could export paradata had to be cautious about the amount of data they exported while also aiming to export the data during periods when few interviewers were active to avoid slow loading speed for interviewers. Thus, we realized that we needed to develop an alternative solution that would facilitate easier access to paradata.

The team tasked with utilizing paradata had several objectives, one of which was to provide colleagues in the department with an up-to-date overview of the data collection. We started by developing Python programs that would generate data collection reports using call history data from Dial History. However, the program reports were not widely used, mainly due to some colleagues perceiving Python as a barrier due to their unfamiliarity with the programming language. Additionally, considering the sensitivity of paradata, we wished to provide stakeholders with reports without giving them access to the data itself, which was required to run the program reports. To achieve this, we recognized the need to create a way to share reports without requiring users to run the program or access the data. Thus, we set out to create a web page with aggregated results over the data collection.

Furthermore, as some colleagues wished to analyze the data themselves, we saw that starting from scratch with each analysis would be inefficient. We therefore developed small Python program modules that could assist users with analyzing the paradata. To ensure accessibility and to encourage more reuse of code, all team members work within one GitHub repository. By developing programs that should work on all instruments and require little coding expertise to run, we have created a versatile solution that can accommodate the needs of colleagues who are involved in different parts of the data collection process. By establishing this pipeline, project managers would receive better support in dynamically managing data collection, and survey methodologists and Blaise developers could more easily access and analyze paradata, facilitating data-driven decisions to improve surveys.

One of Statistics Norway’s digitalization strategies includes implementing Cloud Services and making the shift toward using open-source programming languages. Thus, when creating this pipeline, we decided to use Google Cloud Services and develop the programs in Python. However, aspects of this pipeline can be implemented without the use of Cloud Services. The first part of the article illustrates an overview of the pipeline structure. Next, we share some experiences of how paradata has been used by project managers and survey methodologists to dynamically manage data collection and to improve survey questionnaires. Finally, we summarize the positive effects we have seen from implementing the pipeline and discuss our plans for enhancing the pipeline.

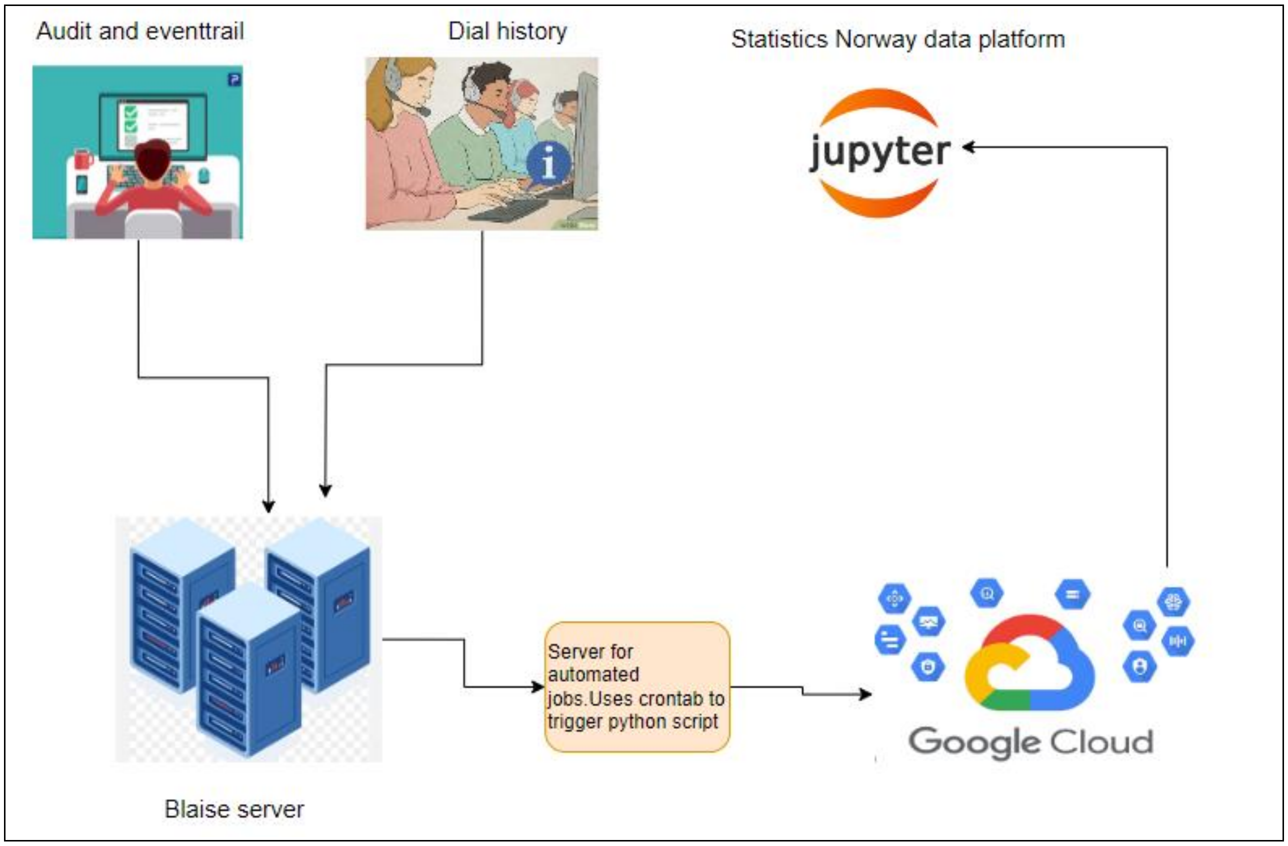
3. Paradata Pipeline

In this section, we will explain how the pipeline is set up. The program code we have used for the pipeline is included in the Appendix. The initial step of the pipeline involves exporting data from our on-premises Blaise server. Next, data is exported to a storage system in the Google Cloud Platform (GCP). Then, the following steps of the pipeline are completed using solutions from Statistics Norway’s new cloud-based data platform (DAPLA). In short, the bullet points below and Figure 1 summarize the steps of our pipeline:

- Exporting data from Blaise to GCP
- Transferring data from the source bucket to the production bucket in GCP
- Developing reports and programs in JupyterLab
- Sharing daily results to the internal web page

Dial History data is parsed in Step 1, and Audit Trail data is parsed in Step 2.

Figure 1. Pipeline from Blaise to GCP


Audit and eventtrail
Dial history
Statistics Norway data platform
jupyter
Blaise server
Server for automated jobs.Uses crontab to trigger python script
Google Cloud

Before explaining the steps in the pipeline, we will first outline the storage structure we use in GCP.

3.1 Using the Google Cloud Platform Structure

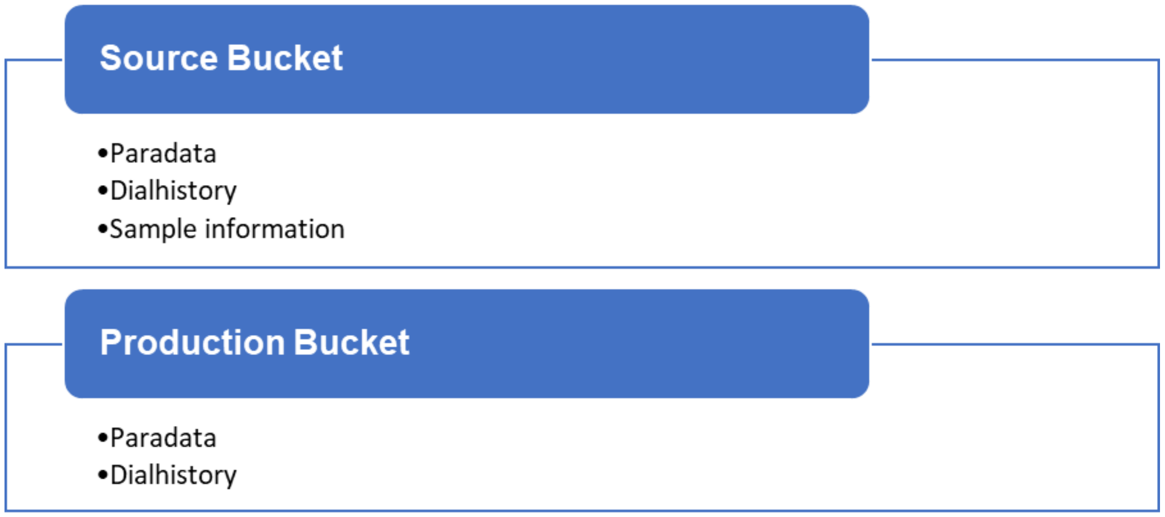
In the GCP, a "bucket" is a unit for containing data that serves the same purpose as a folder. All data in Google Cloud must be stored and organized within these buckets. Essentially, a bucket functions as a structured folder, facilitating the organization and management of different data files. Moreover, we have two buckets for different data structures: unmanipulated and manipulated data. By using different buckets for unmanipulated and manipulated data, we can keep backups of the unmanipulated data where only a few individuals have access. Since the unmanipulated data are the basis for all our analysis and manipulation, having this data in a separate bucket ensures that we have a reliable reference point.

3.1.1 Source Bucket

The source bucket contains unmanipulated paradata from Blaise and select categorical sample data variables, such as age group, region, education level, and gender. Only two people have permission to read, write, or delete files in this bucket.

3.1.2 Production Bucket

In the production bucket, we store manipulated data from the source bucket and data with the select sample variables. Only colleagues who need to use these data are given access to read, write, or delete data in this bucket.



3.2 Exporting Data from Blaise to Google Cloud Platform

The paradata files are exported from the Blaise databases and then stored as CSV files on an on-premises server. To export the paradata from Blaise, we use Code 1 (see the Appendix) for Audit Trail data and Code 2 (see the Appendix) for Dial History. We use Crontab to automatically schedule the export of the paradata each night. We then use GCP's tool "transfer job" to automatically transfer the data from our on-premises server to the source bucket in GCP. Once this step of the pipeline is completed, Audit Trail data and Dial History data are stored as CSV files in the source bucket.

3.3 Transferring Data from the Source Bucket to the Production Bucket in GCP

The next step in the pipeline is to transfer data in GCP from the source bucket and store it as Parquet files in the production bucket. We have utilized a solution created by Statistics Norway's data platform team that uses Cloud Run in GCP to automate this step. The solution allows a team to write a script that is triggered when a new file is added to the source bucket. For the Audit Trail data, we have developed a script (see Code 3 in the Appendix) that reads the data from the source bucket, parses the data, and then stores the parsed data in the production bucket. Similarly, for Dial History data, we have developed a script that reads the CSV file from the source bucket and stores it as a Parquet file in the production bucket.

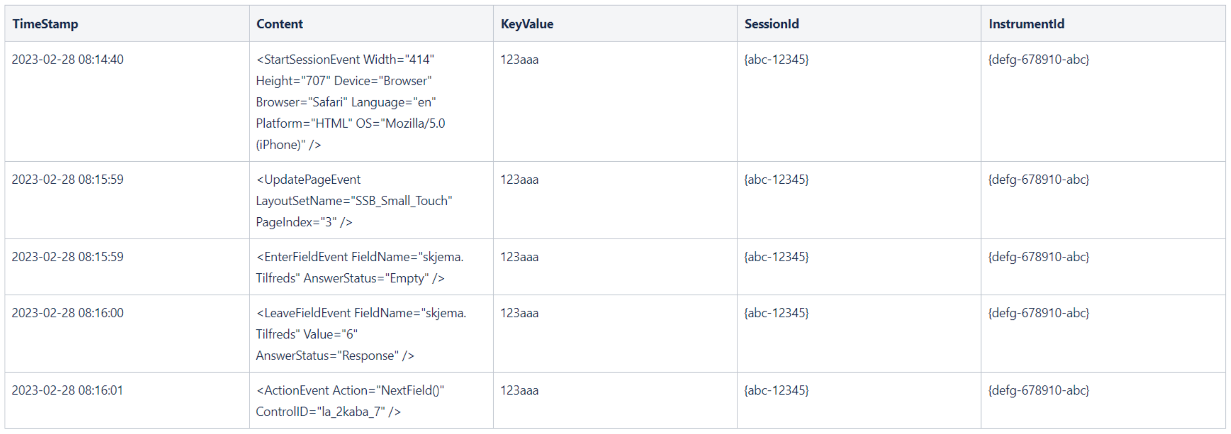
3.3.1 Parsing Audit Trail Data

As the Audit Trail data from Blaise is inadequate for effective analysis, we have parsed the paradata to a structure more suitable for analysis using a script we refer to as the paradata parser (see Code 3 in the Appendix). In the unmanipulated Audit Trail–level data, shown in Table 1, we can see that the "Content" column contains information about the action. The content of the "Content" column is structured as XML-formatted data.

To extract and structure the information in the "Content" column, we use the Python XML package xml.etree.ElementTree. The paradata parser script processes the XML content in the "Content" column of the input dataframe, extracting attributes and tags, and creates a new dataframe with the columns "event", "KeyValue", "TimeStamp", "SessionId", and "InstrumentId", in addition to columns from the attributes as shown in Tables 2 and 3. The parser iterates through each row, creating a new column for each new attribute and adding the attribute's value to the corresponding row. Each tag represents an event type and is included in the "event" column. The paradata parser organizes the data into the structure shown in

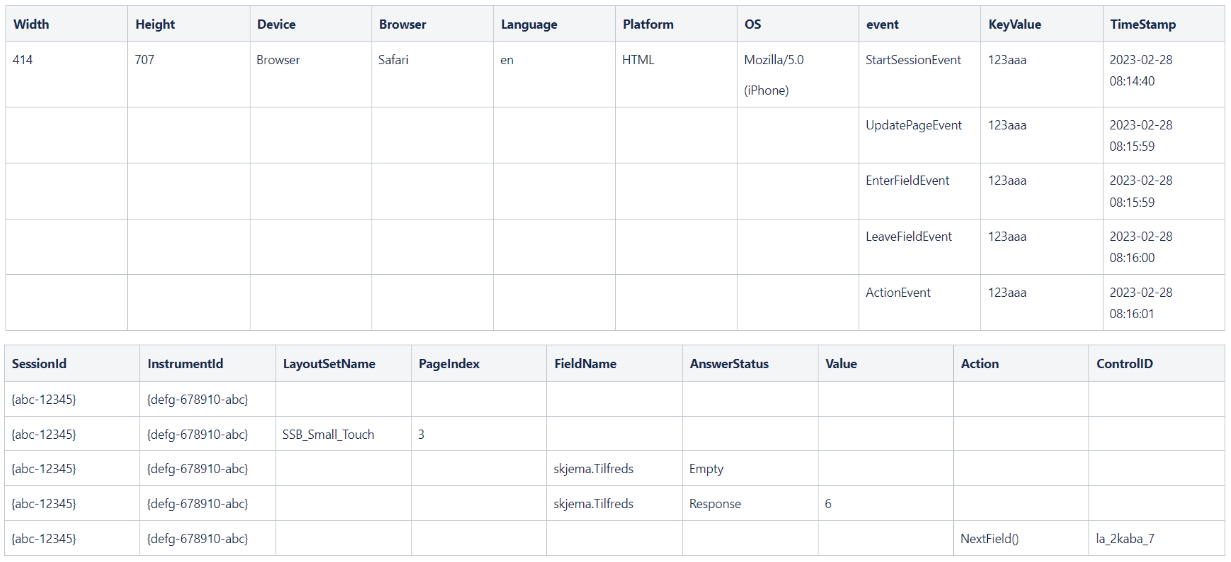
Tables 2 and 3. To enhance readability, the data is split into two separate tables. Also, the column "OS" would contain more information, but for viewing purposes, the text has been shortened.

Table 1. Audit Trail Data from Blaise



TimeStamp	Content	KeyValue	SessionId	InstrumentId
2023-02-28 08:14:40	<StartSessionEvent Width="414" Height="707" Device="Browser" Browser="Safari" Language="en" Platform="HTML" OS="Mozilla/5.0 (iPhone)" />	123aaa	{abc-12345}	{defg-678910-abc}
2023-02-28 08:15:59	<UpdatePageEvent LayoutSetName="SSB_Small_Touch" PageIndex="3" />	123aaa	{abc-12345}	{defg-678910-abc}
2023-02-28 08:15:59	<EnterFieldEvent FieldName="skjema.Tilfreds" AnswerStatus="Empty" />	123aaa	{abc-12345}	{defg-678910-abc}
2023-02-28 08:16:00	<LeaveFieldEvent FieldName="skjema.Tilfreds" Value="6" AnswerStatus="Response" />	123aaa	{abc-12345}	{defg-678910-abc}
2023-02-28 08:16:01	<ActionEvent Action="NextField()" ControlID="la_2kaba_7" />	123aaa	{abc-12345}	{defg-678910-abc}

Tables 2 and 3. Parsed Audit Trail Data



Width	Height	Device	Browser	Language	Platform	OS	event	KeyValue	TimeStamp
414	707	Browser	Safari	en	HTML	Mozilla/5.0 (iPhone)	StartSessionEvent	123aaa	2023-02-28 08:14:40
							UpdatePageEvent	123aaa	2023-02-28 08:15:59
							EnterFieldEvent	123aaa	2023-02-28 08:15:59
							LeaveFieldEvent	123aaa	2023-02-28 08:16:00
							ActionEvent	123aaa	2023-02-28 08:16:01

SessionId	InstrumentId	LayoutSetName	PageIndex	FieldName	AnswerStatus	Value	Action	ControlID
{abc-12345}	{defg-678910-abc}							
{abc-12345}	{defg-678910-abc}	SSB_Small_Touch	3					
{abc-12345}	{defg-678910-abc}			skjema.Tilfreds	Empty			
{abc-12345}	{defg-678910-abc}			skjema.Tilfreds	Response	6		
{abc-12345}	{defg-678910-abc}						NextField()	la_2kaba_7

An advantage of the paradata parser, in contrast to programs that are dependent on regular expressions or fixed string/column position to extract information from the "Content" column, is that the paradata parser automatically creates new columns as it iterates over the data and encounters new attributes. This flexibility ensures that the program can process data with different Audit Trail–level settings. The paradata parser program has worked with data that has "Page", "Field", and "Keyboard" as the Audit Trail–level setting. By utilizing the paradata parser, the pipeline can process all our surveys, each potentially having different Audit Trail–level configurations, thus ensuring flexibility.

Briefly summarized, each night, data is extracted from Blaise using Crontab and then stored in the GCP source bucket. Subsequently, the paradata parser script (see Code 3 in the Appendix) and the solution offered by Statistic Norway's data platform team are employed to parse and transfer the data to the production bucket. As a result, we have updated data that is ready to be analyzed each day.

3.4 Developing Reports and Programs in JupyterLab

We access and analyze the paradata in a JupyterLab environment within Statistics Norway's cloud-based data platform. JupyterLab's interactive computing environment allows us to combine code, visualizations, and explanatory text in a single Jupyter Notebook. The development of new program reports and our web page is done in the JupyterLab environment in Statistics Norway's cloud-based data platform.

As we focus on developing scripts that contain standardized code that work with all surveys, all team members work within one GitHub repository and aim to develop analyses that can be run on various surveys. Moreover, we use Poetry, a tool for package management in Python, to ensure that every team member works with the same set of libraries. By using Statistics Norway's JupyterLab environment and using GitHub for version control, team members find it easy to collaborate on developing code.

When we initially started to analyze paradata, we often found ourselves repeatedly writing and running code for the same or similar tasks when analyzing the data. Therefore, our team decided to focus on code modularization. Code modularization involves creating modules, which are self-contained units of code that promote reusability and organization. Thus, we aim to identify repeated tasks and write this into modules. The infrastructure of our GitHub repository is set up to be flexible for the user, with modules stored within one folder so they can be imported and used when the user is working in the Jupyter Notebook. Thus, code does not need to be copied and pasted between users or Jupyter Notebooks. For instance, two modules are frequently used when analyzing paradata: data query and paradata manipulation.

As data from each day is stored as separate files in the production bucket, we have created a program (see Code 5 in the Appendix) that queries all the data for a specific survey. The program allows colleagues to query all the data for the specific survey anddata in between two dates, before a specific date, or after a specific date.

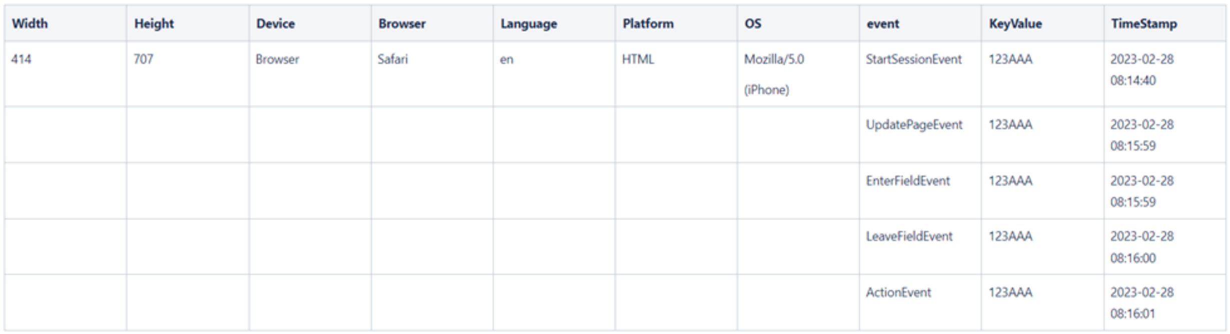
Moreover, the data obtained from the paradata parser may sometimes be insufficient for direct analysis, so we made a module (see Code 4 in the Appendix) to manipulate the paradata, creating a more suitable structure for analysis. The module performs various operations, such as sorting observations by SessionId and TimeStamp, filling values of PageIndex and FieldName, and creating a new column labeled "diff_time" to calculate time frames between consecutive observations within each session. The Tables 3 and 4 show how the paradata looks after applying the module. For viewing purposes, the dataset is split into two tables.

As we aim to continually enhance manipulation processes and conduct quality checks on our programs, we wished to minimize manipulation of the paradata in Step 2 of our pipeline and to instead perform data manipulation in the JupyterLab environment in Step 3. This approach ensures uniformity in the structure of data within the production bucket while we continually develop our code for processing and analyzing paradata.

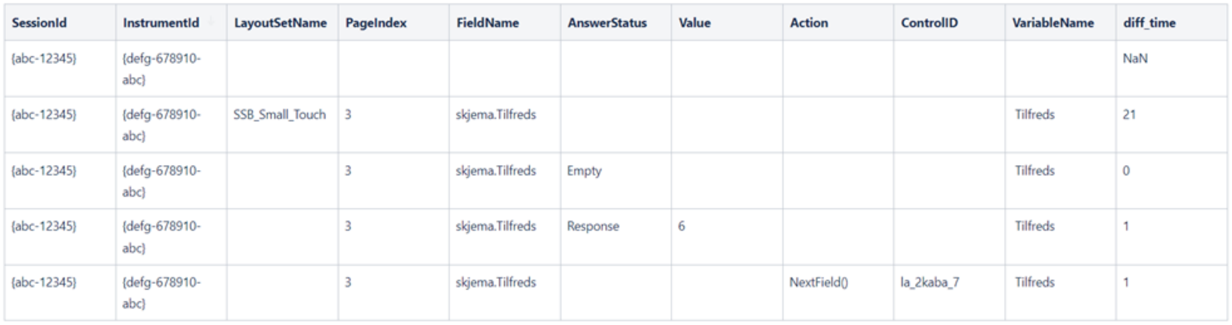
3.5 Sharing Daily Results to Internal Web Page

We wished to make it easier for project managers and field staff to see an updated overview of the data collection. Moreover, it was important that seeing daily results should not require any coding skills or access to the actual data. Thus, we decided to create a web page where we could share reports with key metrics for the data collection.

Tables 3 and 4. Manipulated Paradata



Width	Height	Device	Browser	Language	Platform	OS	event	KeyValue	TimeStamp
414	707	Browser	Safari	en	HTML	Mozilla/5.0 (iPhone)	StartSessionEvent	123AAA	2023-02-28 08:14:40
							UpdatePageEvent	123AAA	2023-02-28 08:15:59
							EnterFieldEvent	123AAA	2023-02-28 08:15:59
							LeaveFieldEvent	123AAA	2023-02-28 08:16:00
							ActionEvent	123AAA	2023-02-28 08:16:01



SessionId	InstrumentId	LayoutSetName	PageIndex	FieldName	AnswerStatus	Value	Action	ControlID	VariableName	diff_time
{abc-12345}	{defg-678910-abc}									NaN
{abc-12345}	{defg-678910-abc}	SSB_Small_Touch	3	skjema.Tilfreds					Tilfreds	21
{abc-12345}	{defg-678910-abc}		3	skjema.Tilfreds	Empty				Tilfreds	0
{abc-12345}	{defg-678910-abc}		3	skjema.Tilfreds	Response	6			Tilfreds	1
{abc-12345}	{defg-678910-abc}		3	skjema.Tilfreds			NextField()	la_2kaba_7	Tilfreds	1

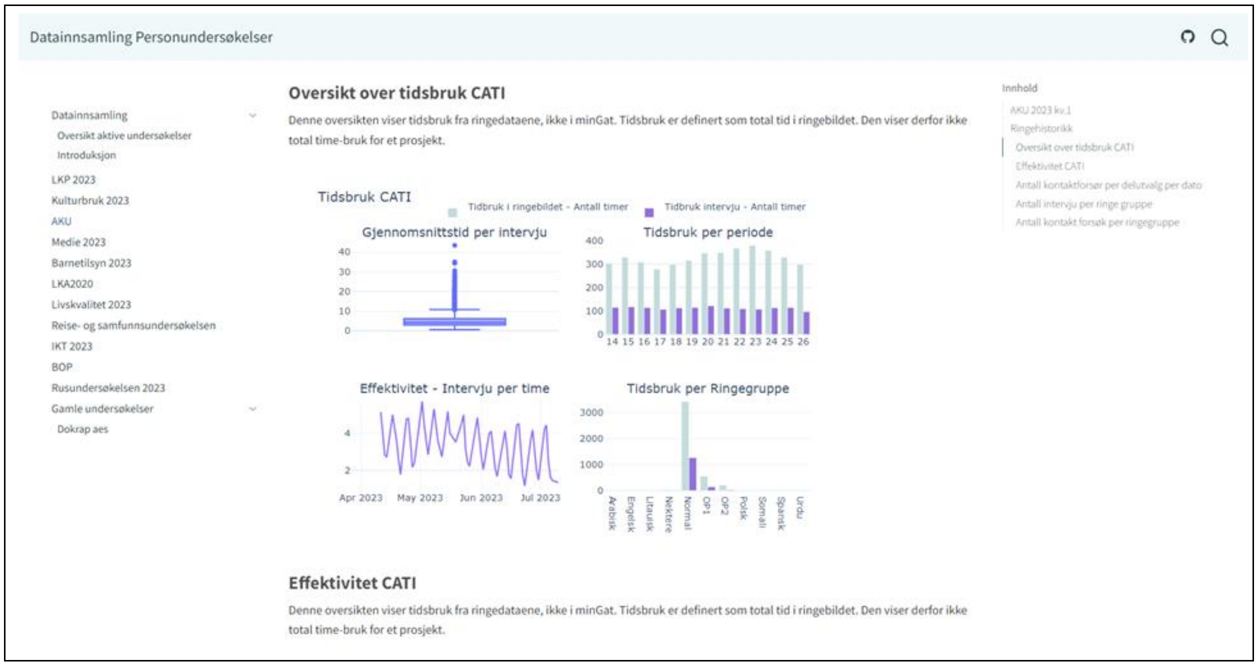
We developed the internal web page by using Quarto[1] and host it with GitHub Pages.[2] Quarto is an open-source solution that combines markdown text and executable Python or R code. When rendering a Quarto file, code is executed, and the output of the code and the markdown text is rendered to HTML files. The collection of HTML files is combined into a complete web page that is hosted with GitHub Pages. As the web page is static, we update it every day by rendering and publishing the complete GitHub repository using Quarto's render and publish commands. By using Quarto, we can utilize the Python reports we create in Step 3 in the JupyterLab environment and then transfer the code to Quarto files for our web page.

Figure 2 shows a screenshot of our web page with key metrics for ongoing surveys. Each survey instrument is represented on the left, and the user can click on the page to view the instrument they wish to see. The figure shows four different figures of CATI time use for the instrument. On the right, we can see an index of the possible reports available for the instrument.

Quarto and GitHub Pages were chosen due to their ease of use, thereby reducing the need for IT support during setup. Furthermore, GitHub Pages guarantees restricted access solely to authenticated employees of Statistics Norway via their GitHub accounts. While we aim to build a more comprehensive solution in the future, such as an interactive dashboard application, the current solution accomplishes our main objective: sharing results and an overview of the data collection process to colleagues without the need for coding or access to the data.

[1] https://quarto.org/
[2] https://docs.github.com/en/enterprise-cloud@latest/pages/getting-started-with-github-pages/about-github-pages

Figure 2. Screenshot of Our Internal Web Page



Oversikt over tidsbruk CATI

Denne oversikten viser tidsbruk fra ringedataene, ikke i minGat. Tidsbruk er definert som total tid i ringebildet. Den viser derfor ikke total time-bruk for et prosjekt.

Effektivitet CATI

Denne oversikten viser tidsbruk fra ringedataene, ikke i minGat. Tidsbruk er definert som total tid i ringebildet. Den viser derfor ikke total time-bruk for et prosjekt.

4. How The Pipeline Is Used

Creating the pipeline allows colleagues to see daily updates of the data collection. Additionally, because it is easier to access structured paradata, colleagues can complete their own ad hoc analysis or use predefined programs to run reports. The section below includes examples of how colleagues use paradata to evaluate survey questionnaires and dynamically manage data collection.

4.1 Survey Project Managers—Dynamic Management of Surveys

An important responsibility for survey project managers is to continually monitor and manage data collection, for example, by adapting to the amount of interviewer resources available. Having daily updated paradata and sample information can be very beneficial for survey managers. As we have had several survey project managers involved in the development of the pipeline, they have effectively started analyzing paradata and developing reports for the web page.

We have used Dial History data to create key metrics that are used to monitor the data collection process. Dial History data contains information about the most important indicators used to supervise the data collection, such as the number of interviews conducted, response rate among respondents who have been contacted at least once, the number of dropouts, and the average interview time. The average interview time is often estimated during testing, but because it may vary from the actual interview time, it is beneficial to check the average interview time early in the data collection process. Since interviewers inform respondents about the estimated duration of the interview when asking them to participate in the survey, it is important that the estimated interview time is close to the actual interview time.

We used paradata in the Living Conditions Survey 2023 to assess nonresponse bias during the data collection period, which helped us initiate measures to try to reduce that bias. When comparing the distribution in the gross sample with the distribution in the net sample, we can say something about nonresponse bias. By using paradata, combined with grouped sample data for gender, age, and education level, we made a visualization showing nonresponse bias for these characteristics. People aged 25 to 44 and people with lower education levels were underrepresented in the net sample. We therefore decided to offer incentives for people from these two groups to increase response rates. Without paradata combined with grouped sample data being updated on the web page daily, we would not have been able to monitor the nonresponse bias continually and initiate the exact measures we did, in addition to measuring the effect in retrospect.

In the same survey, we decided to reduce the maximum number of possible calls to respondents because we saw from paradata that the last calls yielded very few interviews compared with the large share of nonresponses. Paradata enabled us to demonstrate the result of each call in an accessible manner, and we could adjust the daybatch settings accordingly. Explicitly, we reduced the maximum number of calls from 15 to 12 in the Blaise CATI dashboard as the 13th, 14th and 15th call proved to be unfruitful in yielding interviews. This insight was particularly important, as we had limited interviewer resources that demanded efficient time allocation.

4.2 Survey Project Managers—Evaluating Survey Questionnaires

We have also used paradata to understand the flow of the Travel and Shopping Survey. Norwegians' travel and shopping habits abroad were previously included as a minor part of a larger questionnaire conducted through telephone interviews. Later, the survey transitioned to a dedicated web-based platform, resulting in significant restructuring and rephrasing of questions. In connection with this, we used the paradata foundation to test a hypothesis about underreporting.

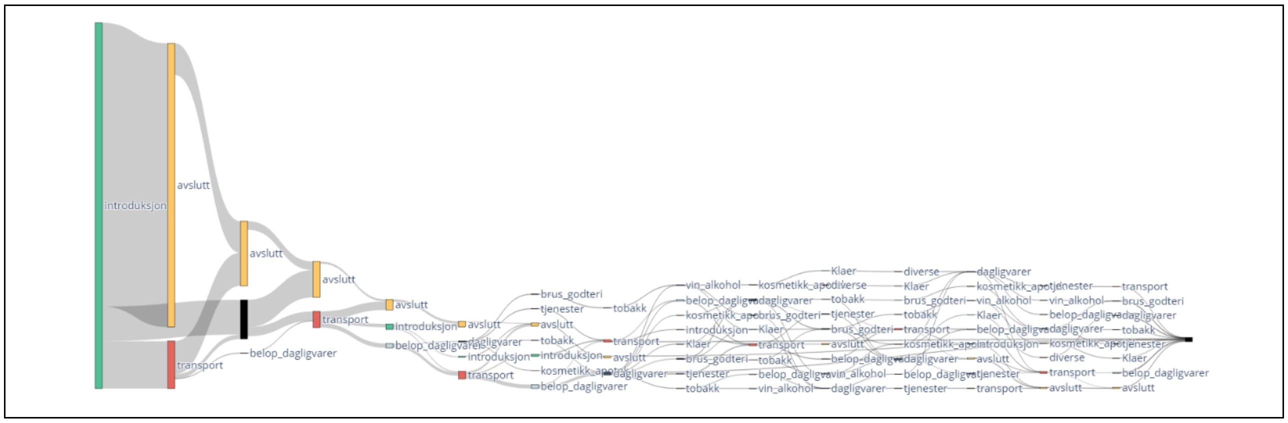
The mode shift and restructuring of questions led to the hypothesis of underreporting for the number of trips taken abroad, as the survey's burden on respondents increases in line with the number of trips. Respondents are first asked how many trips abroad they have taken in the last month, and then they are asked follow-up questions for each trip. The questions are often identical and repeated for each trip. Response burden is often associated with long completion time, poorly formulated questions, and nearly identical repetitive questions (Sharp & Frankel, 1983). Considering this, it is conceivable that respondents may understand the logic of the Blaise questionnaire and reduce the reported number of trips to avoid survey burden and the number of repetitive questions.

In summary, we utilized paradata to test the hypothesis of underreporting by examining respondents' navigation and user journey through the questionnaire. After unpacking, rows were grouped by respondents and timestamp and were ranked in chronological order. By doing so, it became possible to track the user journey of respondents during questionnaire completion. However, due to the volume of information, it was crucial to find a visualization method that would present the user journey of all respondents in a straightforward manner.

Considering the hypothesis, we used a Sankey diagram, as it provides a clear visual representation of the transition between survey questions, revealing bottlenecks where the respondent either drops off, stalls, or goes back to a previous question. Lastly, if the user journey has multiple paths or options, the Sankey diagram can demonstrate how these paths diverge and converge.

Figure 3 shows a visualization of all recorded user journeys through the survey. As seen in the figure, many respondents move from the introduction to the transport questions, but upon closer examination of the diagram, it is evident that the introduction repeats later in the journey. This provided us with an indication that respondents navigate backward in the questionnaire. After closer inspection, we observed that respondents with more trips often adjusted the reported number of trips to avoid repetitive questions about each trip. This led to adjustments in survey questions and the introduction of a cap on the number of trips the respondent must report.

Figure 3. Survey Flow in the Travel and Shopping Survey—Sankey Diagram



Paradata proves to be well suited for analyses aimed at gaining insight into the user journey or flow of respondents in a questionnaire. Often, such an analysis can serve as an initial exploration to identify potential weaknesses in the questionnaire. Furthermore, such visualization provides a focus on where to allocate time and effort for both further analysis of reported data and coding of the Blaise questionnaire itself.

4.3 Survey Methodologists—Recruitment and Questionnaire Monitoring

To design new questionnaires or improve existing questionnaires, it is important for survey methodologists to assess how well the questions and questionnaire flow work for respondents to assure high data quality. Various qualitative methods, such as expert reviews, explorative and cognitive testing, and focus groups, are used to assess problems with the questions and improve them. Problems with the questions might lead to a large respondent burden, for example, due to sensitive questions, confusing question formulations, or no suitable answer options, which can lead to inadequate data quality. Quantitative data, such as paradata, can help to determine if problems detected in qualitative testing persist in the data collection. Similarly, paradata, combined with sample data, can give useful insights into whether a problem occurs for a specific group in the population.

Paradata have been available in Statistics Norway for a while, but with higher accessibility through the pipeline, survey methodologists can more easily use paradata. Paradata can be used for pilot surveys and user testing during the data collection process and after the data collection process is completed. Especially for pilot surveys, the use of paradata during the data collection process can be handy for recruiting specific people to focus groups. For instance, we can recruit those who took a long time to answer the survey, those who are registered with several sessions, or those who received several error messages when answering the questionnaire.

For evaluating the quality of questions or questionnaire flow of a survey, various paradata indicators can be used. It is important to keep in mind that analyzing paradata is time-consuming, and it is therefore advisable to map out the questions we want to analyze using paradata and choose which paradata indicators should be used for the analysis. For instance, indicators we often use to assess respondent burden and data quality are:

- Error messages
- Response time
- Previous page

Error messages can help the respondent avoid obvious mistakes or unanswered questions. Yet, it can be problematic if the error message is not comprehensible to the respondent or if the respondent receives many error messages while answering the survey. If many respondents receive error messages, this can lead to dropout or irritation and might affect the responses for the remaining questions. It may also be an indication of a poorly formulated question.

Analyzing response time has been studied extensively and a review of existing literature has been done by Vehovar and Čehovin (2023). A very short response time can indicate that a respondent does not read the question formulation thoroughly or not at all. On the other hand, a very short completion time could also mean that the respondent is just a particularly quick reader. A very long response time might indicate that the question is difficult to understand or answer. What determines if a response time is too long or too short depends on the type of question. Hence, researchers have to consider the question at hand to determine if a response time is too short or too long.

If many respondents revisit the previous page, it might indicate that there is a problem with the questionnaire flow, for instance, if there is a follow-up question on a new page but respondents require repetition of important information and must revisit the previous page. It might also indicate that the respondents perceive questions as too similar and therefore go backward in the questionnaire to find the difference between the questions.

In sum, paradata can help us to assess data quality and reduce response burden. Although paradata cannot replace qualitative work such as user testing, paradata serve as an additional help for survey methodologists to improve survey questionnaires.

5. Summary

In this article, we have outlined the process of exporting, cleaning, storing, and sharing paradata in our pipeline. The primary objectives for the team responsible for utilizing paradata was to continually provide stakeholders with up-to-date reports of the data collection and to make paradata more easily accessible for colleagues. We achieved these objectives by building this data pipeline. Within the pipeline, we have focused on code modularization and building program reports that can be run on several surveys. In addition to using program reports, colleagues can conduct their own analyses and develop program reports tailored to their specific needs. Our data collection department is composed of colleagues with varying levels of programming proficiency, and everyone has a critical role in data collection procedures. To ensure that everyone has the opportunity to view the outcomes of data collection and participate in enhancing its efficiency, irrespective of their programming proficiency, we share an overview of the data collection process on an internal web page. The examples of how our survey project managers and survey methodologists use paradata highlight the foundational role of our pipeline in shaping a range of crucial decisions. The pipeline facilities effective use of paradata to inform decisions about data collection processes and survey questionnaire development, thereby enhancing several aspects of the data collection.

6. The Way Forward

Since our pipeline project is in its early stages, we wish to improve certain aspects. Firstly, we wish to include more reports on our web page and build more supporting modules and programs for ad hoc analyses. To understand the needs of our colleagues and update them on our progress, we have regular sprint reviews where we show program reports we have developed and new additions to the webpage. These sprint reviews are opportunities for colleagues to suggest improvements and solutions that can help support them in their roles. Secondly, the steps in our pipeline are mostly automated, except for updating the web page. Thus, one of our priorities is to develop a solution that will automatically update our web page. Furthermore, we wish to create a dashboard that will give the user more flexibility with interactivity.

Our pipeline is a work in progress, and we expect to update and improve aspects of the pipeline as we discover bugs or find improvements. Thus, we suggest that interested readers contact us to see the updated versions of the code shown in this article.

7. References

Cheung, G., Piskorowski, A., Wood, L., & Peng, H. (2016). *Using survey paradata*. 17th International Blaise Users Group Conference, The Hague, Amsterdam, the Netherlands.

Hunt, J. (2016). *Using Audit Trail data to move from a black box to a transparent data collection process*. 17th International Blaise Users Group Conference, The Hague, Amsterdam, the Netherlands.

Kreuter, F., Couper, M., & Lyberg, L. (2010). *The use of paradata to monitor and manage survey data collection*. *American Statistical Association, Proceedings of the oint statistical meetings,* (pp. 282–296). Alexandria, VA, United States.

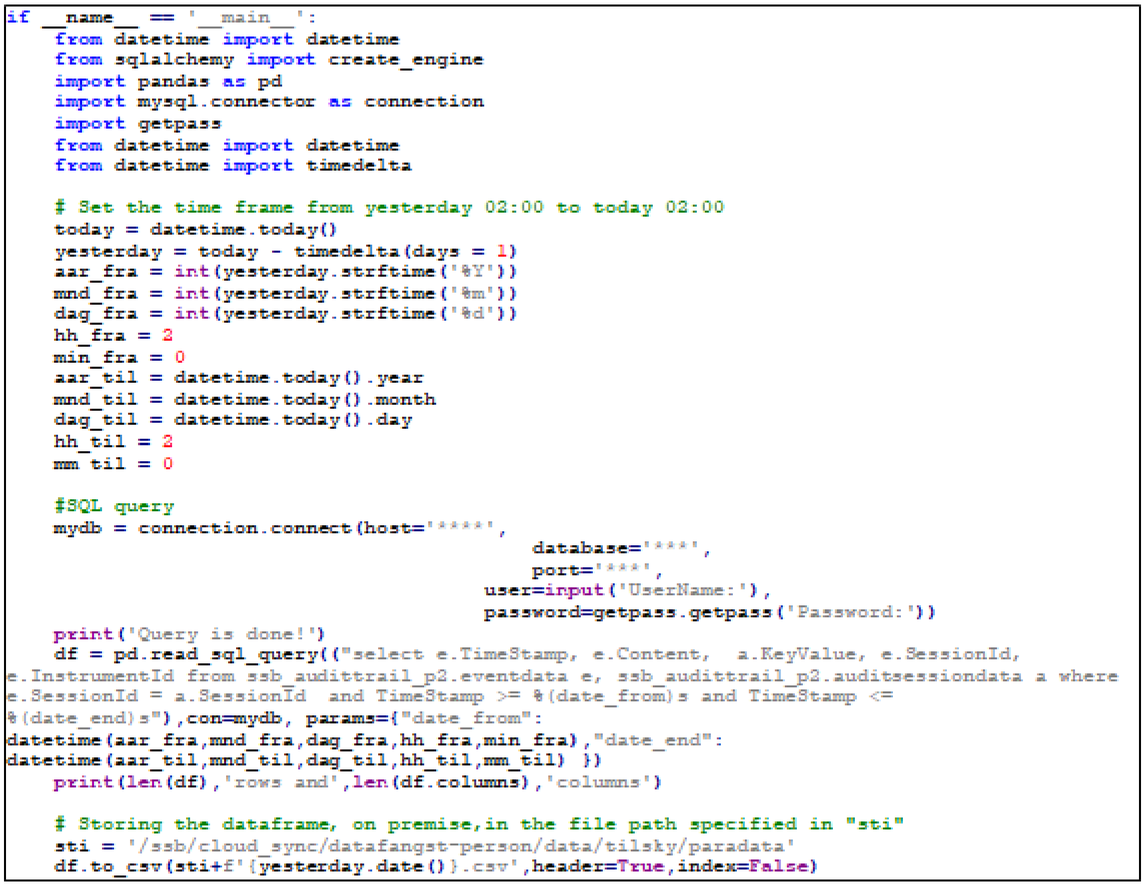
Schouten, B., Peytchev, A., & Wagner, J. (2017). *Adaptive survey design* (1st ed.). Chapman and Hall/CRC.

Sharp, L. M., & Frankel, J. (1983). Respondent burden: A test of some common assumptions. *Public Opinion Quarterly*, *47*(1), 36–53.

Vehovar, V., & Čehovin, G. (2023). *Direct paradata usage for analysis of response quality, respondent characteristics, and survey estimates: State-of-the-art review and typology of paradata* (Working Paper). Center for Social Informatics, University of Ljubljana. https://www.fdv.uni-lj.si/docs/default-source/cdi-doc/direct-paradata-usage-for-analysis-of-response-quality.pdf?sfvrsn=0%20

8. Appendix

Code 1. Exporting Audit Trail Data from Blaise



```
if __name__ == '__main__':
    from datetime import datetime
    from sqlalchemy import create_engine
    import pandas as pd
    import mysql.connector as connection
    import getpass
    from datetime import datetime
    from datetime import timedelta

    # Set the time frame from yesterday 02:00 to today 02:00
    today = datetime.today()
    yesterday = today - timedelta(days = 1)
    aar_fra = int(yesterday.strftime('%Y'))
    mnd_fra = int(yesterday.strftime('%m'))
    dag_fra = int(yesterday.strftime('%d'))
    hh_fra = 2
    min_fra = 0
    aar_til = datetime.today().year
    mnd_til = datetime.today().month
    dag_til = datetime.today().day
    hh_til = 2
    mm til = 0

    #SQL query
    mydb = connection.connect(host='*****',
                                        database='****',
                                        port='****',
                                    user=input('UserName:'),
                                    password=getpass.getpass('Password:'))
    print('Query is done!')
    df = pd.read_sql_query(("select e.TimeStamp, e.Content,  a.KeyValue, e.SessionId,
e.InstrumentId from ssb_audittrail_p2.eventdata e, ssb_audittrail_p2.auditsessiondata a where
e.SessionId = a.SessionId  and TimeStamp >= %(date_from)s and TimeStamp <=
%(date_end)s"),con=mydb, params={"date_from":
datetime(aar_fra,mnd_fra,dag_fra,hh_fra,min_fra),"date_end":
datetime(aar_til,mnd_til,dag_til,hh_til,mm_til) })
    print(len(df),'rows and',len(df.columns),'columns')

    # Storing the dataframe, on premise,in the file path specified in "sti"
    sti = '/ssb/cloud_sync/datafangst-person/data/tilsky/paradata'
    df.to_csv(sti+f'{yesterday.date()}.csv',header=True,index=False)
```

Code 2. Exporting and Preparing Dial History Data from Blaise

```
from datetime import datetime, timedelta
from sqlalchemy import create_engine
import pandas as pd
import mysql.connector as connection
import getpass
# Create connection
mydb = connection.connect(host='****', database='*****', user=username,
password=getpass.getpass('Password:')
current_time = datetime.now()
# Parse data for the current day
df = pd.read_sql_query(('select * from ****.dialhistory where StartTime >= %(date_from)s and
EndTime <= %(date_end)s")',con=mydb, params={"date_from": datetime(current_time.year, cur-
rent_time.month, current_time.day,0,0), "date_end": datetime(current_time.year,cur-
rent_time.month,
current_time.day,current_time.hour, current_time.minute) })
# Extract and expand each line
a_1  = df['AdditionalData'].str.extract(pat = '("styring.ToWhom" Status=\S*..\S*)', expand =
True)
a_2  = df['AdditionalData'].str.extract(pat = '("io_nummer" Status=\S*..\S*)', expand = True)
a_3  = df['AdditionalData'].str.extract(pat = '("innled" Status=\S*..\S*)', expand = True)
a_4  = df['AdditionalData'].str.extract(pat = '("intslutt" Status=\S*..\S*)', expand = True)
a_5  = df['AdditionalData'].str.extract(pat = '("intervjustatus" Status=\S*..\S*)', expand =
True)
a_6  = df['AdditionalData'].str.extract(pat = '("ioblokk.PeriodeNr" Status=\S*..\S*)', expand
= True)
a_7  = df['AdditionalData'].str.extract(pat = '("ioblokk.Delutvalg" Status=\S*..\S*)', expand
= True)
a_8  = df['AppointmentInfo'].str.extract(pat = '(<StartDate>\S*..\S*</StartDate>)', expand =
True)
a_9  = df['AppointmentInfo'].str.extract(pat = '(<StartTime>\S*..\S*</StartTime>)', expand =
True)
# Extract and expand new lines and give the column names
a1_value = a_1[0].str.extract(pat = '(Value=\S*)', expand = True)
a1_value['value_towhom']  = a1_value[0].str.extract('.*\'(.*)\'.*')
a2_value = a_2[0].str.extract(pat = '(Value=\S*)', expand = True)
a2_value['value_io_nummer']  = a2_value[0].str.extract('.*\"(.*)\".*')
a3_value = a_3[0].str.extract(pat = '(Value=\S*)', expand = True)
a3_value['value_innled']  = a3_value[0].str.extract('.*\"(.*)\".*')
a4_value = a_4[0].str.extract(pat = '(Value=\S*)', expand = True)
a4_value['value_intslutt']  = a4_value[0].str.extract('.*\"(.*)\".*')
a5_value = a_5[0].str.extract(pat = '(Value=\S*)', expand = True)
a5_value['value_intervjustatus']  = a5_value[0].str.extract('.*\"(.*)\".*')
a6_value = a_6[0].str.extract(pat = '(Value=\S*)', expand = True)
a6_value['value_ioblokk.PeriodeNr']  = a6_value[0].str.extract('.*\"(.*)\".*')
a7_value = a_7[0].str.extract(pat = '(Value=\S*)', expand = True)
a7_value['value_ioblokk.Delutvalg']  = a7_value[0].str.extract('.*\'(.*)\'.*')
#endtime - starttime
df['Diff_ES_min'] = ((df['EndTime'] - df['StartTime']).astype('timedelta64[s]')/60).round(2)
#Appointment_StartDate
a8_startdate = a_8[0].str.extract('.*\>(.*)\<.*')
#Appointment_StartTime
a9_starttime = a_9[0].str.extract('.*\>(.*)\<.*')
df['value_towhom'] = a1_value['value_towhom']
df['value_io_nummer'] = a2_value['value_io_nummer']
df['value_innled'] = a3_value['value_innled']
df['value_intslutt'] = a4_value['value_intslutt']
df['value_intervjustatus'] = a5_value['value_intervjustatus']
df['value_ioblokk.PeriodeNr'] = a6_value['value_ioblokk.PeriodeNr']
df['value_ioblokk.Delutvalg'] = a7_value['value_ioblokk.Delutvalg']
df['app_startdate'] = a8_startdate[0]
df['app_starttime'] = a9_starttime[0]
df["Date"] = df["StartTime"].dt.date
dial_history = df[['Id','InstrumentId','PrimaryKeyValue','CallNumber','DialNumber',
'Date','StartTime','EndTime','Status','value_intervjustatus','io_nummer','LastDialTi-
me','DialResult','app_startdate','app_starttime','value_towhom','value_io_nummer','value_inn-
led','value_intslutt','value_intervjustatus','value_ioblokk.PeriodeNr',
'value_ioblokk.Delutvalg','Diff_ES_min']]
# Saving to csv
dial_history.to_csv(path_to_save + f'{datetime.now().date()}.csv', header=True, index=False)
```

Code 3. Parsing Paradata and Storing the Structured Paradata in a Parquet File

```
# Importing packages
import pandas as pd
import xml.etree.ElementTree as ET
import dapla as dp
import pyarrow as pa

def paradata_parser(raw_paradf):
    datalist = [] # Create an empty list that will contain all the dict
    for i, row in raw_paradf.iterrows(): # Iterate through each row
        dicInner = ET.fromstring(row['Content']).attrib # Transforms the [Conetent] columns by
creating a dict of all found attributes to a dict
        dicInner['event'] = ET.fromstring(row['Content']).tag # add event from the tag
        dicInner['KeyValue']= row['KeyValue']
        dicInner['TimeStamp']= pd.to_datetime(row['TimeStamp'])# add TimeStamp column
        dicInner['SessionId']= row['SessionId'] # add sessionID
        dicInner['InstrumentId']= row['InstrumentId']# add [KeyValue]
        datalist.append(dicInner) # append the row to a dataframe
    df = pd.DataFrame(datalist)  # create a dataframe of all rows
    return df

# Creating a schema where all variables the TimeStamp column is datetime timestamp and all
other variables are string variable
def create_schema(column_names, timestamp_column_name="TimeStamp"):
    fields = []
    for column_name in column_names:
        if column_name == timestamp_column_name:
            fields.append((column_name, pa.timestamp('ms')))
        else:
            fields.append((column_name, pa.string()))
    schema = pa.schema(fields)
    return schema

def main(file_path):
    prod_file_path = "gs://ssb-prod-datafangst-person-data-produkt/Inndata/paradata/"
    para_date = file_path[-22:-4]

    # Read the last file and creat a dataframe
    raw_paradf = dp.read_pandas(f"gs://{file_path}", file_format = "csv")

    # parsing the dataframe with raw paradata through the paradata parser.
    df = paradata_parser(raw_paradf)

    # create a schema
    column_names = df.columns.to_list()
    schema = create_schema(column_names)

    # Write the cleaned  with defined schema
    dp.write_pandas(df = df,
                    gcs_path = f"{prod_file_path}{para_date}.parquet",
                    file_format = "parquet",
                    schema=schema)

    return
```

Code 4. Manipulating Paradata for Analysis

```
def fill_para(table_df):
       # Sorting rows based on SessionId and timestamp.
        table_df = table_df.sort_values(['SessionId','TimeStamp'])

        # Reset the index
        table_df.reset_index(inplace = True, drop = True)

        # fill pageindex downwards
        cols = ['PageIndex']
        table_df.loc[:,cols] = table_df.loc[:,cols].ffill()

        # Grouping on page index and fills FieldName down and then up.
        table_df['FieldName'] = table_df.groupby('PageIndex')['FieldName'].transform(lambda v:
v.ffill())
        table_df['FieldName'] = table_df.groupby('PageIndex')['FieldName'].transform(lambda v:
v.bfill())

        # Create a new variable that contains the short question variable name
        table_df['VariableName'] = table_df['FieldName'].str.rsplit('.', n=1).str.get(-1)

        # Create a variable diff_time. Time in seconds between one observation and the next
within each session.
        table_df['diff_time'] = table_df.groupby(['SessionId'])['TimeStamp'].diff()
        table_df['diff_time'] = table_df.diff_time.dt.total_seconds()

        # Capitalise KeyValue because Blaise is not case-sensitive and Python is.
        table_df['KeyValue'] = table_df['KeyValue'].str.upper()
        return table_df
```

Code 5. Module to Query Data from the Production Bucket

```
def file_concat(InstrumentId, dager=None, start_dato=None, slutt_dato=None):
    # Importerer nødvendige pakker
    import dapla
    import sys
    from dapla import FileClient
    import pandas as pd
    import os
    import pyarrow.parquet as pq
    import warnings
    warnings.filterwarnings('ignore')
    # Får tilgang til bøtte strukturen
    fs = FileClient.get_gcs_file_system()
    alle_filer_i_bøtta = fs.glob('gs://ssb-prod-datafangst-person-data-produkt/Inndata/ringedata'+
"/*.parquet")
    # LAger tester foir ulike parametre
    alle_dager = ((dager is None) &(start_dato is None) &(slutt_dato is None))
    antall_dager = ((dager is not None)&(start_dato is None)&(slutt_dato is None))
    fra_dato = ((dager is None)&(start_dato is not None)&(slutt_dato is None))
    til_dato = ((dager is None)&(start_dato is None)&(slutt_dato is not None))
    intervall = ((dager is None)&(start_dato is not None)&(slutt_dato is not None))
    alle_filer_i_bøtta = FileClient().ls('ssb-prod-datafangst-person-data-produkt/Inndata/ringedata')
    # Sjekker indexen for ønsket datoer i alle filers liste
    start = ''
    if start_dato is not None:
        for index, element in enumerate(alle_filer_i_bøtta):
            if start_dato in element:
                start = index
        if start == '':
            sys.exit(0)

    elif start_dato is None:
        pass
    else:
        raise UnboundLocalError(f'{start_dato} finnes ikke i bøtta')
    slutt = ''
    if slutt_dato is not None:
        for index, element in enumerate(alle_filer_i_bøtta):
            if slutt_dato in element:
                slutt = index
        if slutt == '':
            sys.exit(0)
    elif slutt_dato is None:
        pass
    else:
        raise UnboundLocalError('Denne datoene finnes ikke i bøtta')

     #Velger filer i som ønsket antall dager
    if alle_dager:
        files = alle_filer_i_bøtta[1:]
    elif antall_dager:
        files = alle_filer_i_bøtta[-dager:]
    elif fra_dato and start != '':
        files = alle_filer_i_bøtta[start:]
    elif til_dato:
        files = alle_filer_i_bøtta[1:slutt+1]
    elif intervall and start!='':
        files = alle_filer_i_bøtta[start:slutt+1]
    # Lager en beholder for ønskede fielr
    out = []
    # Looper gjennom alle ønskede filer
    for file in files:
        file = 'gs://' + file
        out.append(file)
    table_df = pq.ParquetDataset(out, filesystem=fs, filters=[("InstrumentId", "==",
InstrumentId)]).read().to_pandas()
    return table_df
```