Analyzing location-specific error patterns in train data

Final Report

Bachelor Project 2018

Henk Grent

Mark Haakman

Frenk van Mil

Sander Waij

Coach: M. de Weerdt, TU Delft

Client: I. Kalsbeek, Nederlandse Spoorwegen

Project coordinator: O. Visser, TU Delft

TU Delft

Abstract

Through the years, companies have been exploring the field of data science. The Nederlandse Spoorwegen (NS) is not an exception to this. Modern trains are equipped with sensors that measure a variety of conditions within the train. This data is being stored in their data warehouse. This data has been proven useful for detection and response times to problems, which warrants two high-level goals of the NS: punctuality and reliability. However, even with the available data, visualization and detection of location-specific problems are not yet implemented. Location-specific problems are problems that are not caused by the train, but by the infrastructure or human fault at that specific location. At the moment, most patterns in error codes are only backed up by suspicions, since these error codes are not stored in a way they are easily readable. Therefore, it is hard to find connections between multiple error codes. This document describes the created system that supports the analysis of location-specific error code patterns. With the system, the NS will be able to improve their two high-level goals and ultimately improve customer satisfaction.

For the system, a framework was made, which allows the NS to further develop and extend on data analyses. Furthermore, an extensive UI was created, allowing users to investigate found error code patterns and trace back problems to their origin. With the system, the NS is able to verify and create new hypotheses on possible problematic locations. In this document, the problem in elaborated on, multiple solutions are given of which one is chosen and thoroughly motivated, the solutions are elaborated on and, finally, some recommendations for future expansion are given.

Foreword

This project has been executed as part of a bachelor end project (BEP) for the TU-Delft by a team of four computer science students whose names are disclosed on the cover. Guidance from the TU-Delft's side has been given by M. de Weerdt. He primarily gave advice about the process of the BEP and jumped in with suggestions to improve this process. Guidance from the NS' side has been given by I. Kalsbeek, she provided us with the initial problem statement, as formulated by the NS, and helped us through the difficulties a big company brings. We thank her for that. Additional credit to two of our end-users, T. van Haperen and M. Schulte, for participating in demos and voicing their findings.

Table of contents

Introduction

Every day more than a million people travel by train in the Netherlands. For all these passengers it is of uttermost importance that they arrive at their location on time. It is not uncommon for trains to have delays, whereas there are many causes that could lead to an uphold. Presumably, some of these upholds can be prevented if data gathered by trains could indicate on which track sections infrastructural problems are situated. Currently, there are systems that could show error codes in trains and surveillance on this data helps in picking out trains at the right moment for repairs. However, there are currently no systems that indicate where a problem is caused. Especially problems caused by a location, e.g. the rails, are not detected. For this project, the aim is to find these problems and try to trace back their exact location. Examples for these location-specific problems could be skewed rails, maimed power signals, but also the education of train operators.

For an enterprise, data analysis provides insight which supports decision making that ultimately leads to the achievement of higher (business) goals, when done in a meaningful way. In this case, the data of the trains, owned by the Dutch Railways (NS), is subject to analysis. By analyzing such data the NS hopes to gain meaningful insight into patterns that occur in diagnostic error codes that are location-specific. For example, a certain code occurring abnormally often on Rotterdam Central Station. A diagnostic error code is a code, e.g. ATB105, that is thrown when sensors detect deviant behavior in the train system(s) they belong to. The indication that such patterns may be location-specific allows analysts to address these location-specific problems.

A long-term goal for the maintenance and development department of the NS is to use data generated by trains to improve punctuality and reliability of the trains while balancing the costs of train maintenance. Punctuality and reliability will lead to better customer satisfaction, which includes the ordinary traveler. Being able to detect location-specific observations helps with addressing infrastructure issues, which will both prevent future breakdowns/delays and reduce costs due to unnecessarily sending trains through maintenance to fix problems that were not related to the train. Other applications, such as seeing what mistakes are often made for a certain location, follow and add value in a similar manner.

A system was created for giving insight into location-specific patterns in train data. This is an entirely new system within the analytics environment of the NS using big-data analysis in order to find location-specific patterns. It also is one of the first systems within the analytics environment that attempts to make a more centralized environment for long-term, user-friendly data analysis.

The system is designed to be a manageable, scalable and high-performance system for running varying types of analyses, data filters, and post processors with a running database connection which is open for end-user interaction and customization. The design of the system ought to make it easier to implement new types of analyses, filters and post-processors and make it easier to visualize the results of the analyses. It also provides an attractive user interface (UI) which allows the user to run analyses, customize analysis settings/parameters and a separate UI for data visualization (Power BI).

This great potential comes with several challenges to be overcome. The relatively recent transition of the NS to a more (digital) data-driven organization is one of these challenges. This transition applies to the train maintenance part of the NS in particular, where they are (e.g.) progressing from periodic maintenance towards condition-based maintenance. Another challenge is the sheer volume of data the analyses have to run on. Additionally, challenges were found in the fact that it is a new system.

In broad terms, this report consists out of three main sections: the problem description, solutions to this problem and a discussion. The problem is elaborated upon in the problem description, in order to better understand the problem, its challenges and the context in which a solution had to be made. In the solutions section, different solutions to the problem are given. This includes the high-level design that is to address the problem and more specific solutions that are placed within the high-level design. Finally, the previous sections are discussed and an evaluation of the process as a whole is given. From this, a conclusion is drawn and some future recommendations are given.

Problem Description

To understand the problem of the NS, the problem description is elaborated on. The problem description is divided in the problem statement and the challenges. The problem statement analyzes the problem that needs to be solved by the system and describes the current situation. The challenges section describes the challenges that come with solving the problem statement. The long term vision of the maintenance and development department of the NS can be found in Appendix M: Research report, chapter 'Vision'.

Problem statement

A long-term goal for the maintenance and development department of the NS is to use data generated by trains to improve punctuality and reliability of the trains while balancing the costs of train maintenance. Equipping trains with sensors and collecting (real-time) data has proven to be useful for the NS, in order to be able to detect and respond to problems faster. A faster response time helps with improving punctuality and reliability.

The NS has to make the trade-off between premature maintenance sessions and failures due to a lack of maintenance check-ups. Premature maintenance sessions cost money and put more stress on the service stations. Train failures result in extra costs and are detrimental to the punctuality, reliability, and image of the NS. Premature maintenance thus should be done as few times as possible, whereas failures which could be prevented with a maintenance session are to be minimized as well. These are two contradicting goals for which achieving the optimal balance is wanted.

However, not all failures are due to the trains themselves. The track could be a huge factor in failures as well. It could now be the case that a train appears to be causing troubles, although the track it was riding on initially caused it. This may result in unnecessary maintenance check-ups and delayed repairs to the tracks. Therefore, it is a good idea to check for problems on the track. This can be done by checking if there are multiple trains that have the same problems at this track section. This can be spotted by having multiple trains sending the same error codes. In this report, these recurring error codes will be called location-specific error code patterns. Being able to find such trends would contribute to achieving the long-term goal of improving punctuality and reliability.

At the moment, there is no system implemented that allows for the detection of location-specific error code patterns, whilst there is data that could possibly indicate the existence of location-specific problems. As a result of the lack of this system, it is hard to detect which problems structurally occur or suddenly arise on given railway sections. Patterns in problems are being found, although they are now mostly based on human notice. Moreover, the data sent by trains is currently not used for long-term location-specific analyses, which implies that there is still a lot of room for patterns to be found. This project will be one of the first steps toward location-specific analyses.

Objective

The objective is to create a system that can be integrated within the current software environment of the NS, in order to gain insight into location-specific error code trends to prevent future breakdowns and faults. The target demographic consists of two end-users: the Reliability Engineer and the Fleet Analyst.

Current System

The current system is defined in this subsection. First, the end users will be introduced in more detail. Secondly, an overview of the existing software will be given. Thirdly, the available data will be assessed. Finally, the available tools for this project will be expanded on.

End users

There are two kinds of end users which are key for indicating for what kind of users the application will be developed for. In choosing solutions related to the problem their needs need to be taken into account. As their knowledge of specialization is important for understanding the technical significance behind the data.

The function of the Reliability Engineer (RE) is to monitor the reliability of an assigned train type. His task is to ensure the reliability and safety of trains, within a given budget. He generally functions on more long-term analyses. The Fleet Analyst (FA) on the other hand, looks at the problems that are going on right now, determines where and in which train or area those problems occur and delegates this problem to the task force. More extensive descriptions can be found in Appendix M: research report, chapter ‘End users’.

RTMO / RTMA

Real-Time Monitoring Operations (RTMO) is an integrated environment with a visual interface used for by the operations side of the NS. This environment is outsourced to be developed by another company. RTMO is primarily used by the Fleet Analyst to see what problems are going on at the moment and to connect those problems to actions for the OCCR taskforce. RTMO uses data until the last maintenance session, which is, on average, every three months.

Real-Time monitoring Analytics (RTMA) is a concept used in the data analytics environment. This environment has no universally platform/API, nor does it have a universally way of displaying data. RTMA is primarily used by the Reliability Engineer and other technical analysts. RTMA contains all available data fetched from sensors in trains and is not time-bound.

Available tools

Big data processing

The NS has an integrated big data processing environment. During the project, this environment can be used without having to consider the implementation of the underlying cluster too much. The following tools

are available: Apache Hadoop, Hive, Mahout, Pig, Zookeeper, and Spark. These tools can be accessed through Jupyter.

Visualization

For visualization NS primarily uses Microsoft Power BI. As quoted from the front page of the Power BI site[1]: "Power BI is a suite of business analytics tools that deliver insights throughout your organization. Connect to hundreds of data sources, simplify data prep, and drive ad hoc analysis. Produce beautiful reports, then publish them for your organization to consume on the web and across mobile devices." This tool is used by data analysts to display their data. Other data visualization techniques for the RTMA environment are incoherent, commonly various Python libraries or interfaces such as Excel are used.

Challenges

From the problem statement and the current system, a number of challenges stand out. Creating an application is influenced by the given challenges. Explaining what these challenges mean for the project is essential for understanding the context in which choices are made and why, consequently, the certain solutions are chosen. This chapter outlines which challenges affect the project the most in the sense of time and/or complexity.

Unknown domain

This is one of the first project within the NS that creates a system within RTMA that attempts to make a more centralized environment for long-term, user-friendly data analysis. This means that an application has to be created for an unknown domain. Currently, the NS is advancing in the field of data science. The relatively recent transition to a more (digital) data-driven organization has some on-going effects, which means that the data present is still not perfect, that knowledge about what types of analyses are useful for location-specific analysis is in the process of being figured out, and that validating whether the chosen system delivers meaningful results is seemingly difficult.

There are only hypotheses made about the existence of location-specific error code patterns. It could be that there is no such thing. The challenge that comes with this is that the end user has to be convinced the application can actually point him to real-world problems which are worth his time. Without being able to prove a found pattern is valuable by ourselves, close communication is needed between the developers and experts in order to distill a concept of how the system could add value to the organization. Feature selection has to be done by asking the experts for the meaning of certain data/features, with respect to the underlying technicalities that determine the assumptions that can be made for a given feature or aggregation of features. However, it is challenging to get an idea of which supposed facts can be used to base value propositions on because it is an unknown domain. This makes it not at all self-evident which analyses should be used and whether those analyses are meaningful.

[1] *https://powerbi.microsoft.com/en-us/*

There is no validation data present, to check if a found error pattern from past data really was paired with a problem. This means algorithms which need this verification data cannot be used. Furthermore, the results of other algorithms have to be closely looked at. It could be that these results contain lots of false positives, which is hard to check without verification data.

As the search for location-specific error patterns is still in the experimental phase, it is important to make the system easily extendable. The challenge for the system lies in the expandability of the program. Therefore, the system must be built as a framework for the user, in order to allow more future analyses to be built with the reuse of components. Furthermore, class contracts must be built, allowing the user to expand the system with new analysis, while still following consistent coding agreements.

Existing environment

The application has to be integrated within current software and hardware systems in use by the NS, as stated in design goal 3: Manageability. Which systems are exactly available can be found in the Current Systems section.

Given that the system has to be integrable within the NS, additional research has to be done in order to figure out what system is already implemented within the existing development environment and how the project can be developed within this existing environment's architecture. The NS has strict protocols regarding which software/libraries they use and who gets which permissions to do what. They have to do this as software often costs a fair amount of money when used for commercial purposes and changing versions of libraries may give unexpected errors if not done in a controlled manner. This brings along two challenges: restricted freedom in designing the architectural design of the system and limited access to permissions required to use software/libraries.

The NS working environment lacks some utilities which make programming easier. There is no access to Git on the cluster, which means no access to version control. This becomes a challenge when a team of four developers is working on the same code. Also, there is no Integrated Development Environment available for use, which means, for example, no static code checking.

Big data

The application has to run on large, rapidly growing, data sets which are filled with data that is not perfect yet. Not every train sends real-time diagnostic data yet, but as time passes, more trains will be equipped with this functionality. Every new train will have this functionality enabled. This means the already large dataset will grow even faster in the future. Even with the NS having its own powerful Hadoop cluster to perform calculations on, more complex analyses have to be designed in a way that does not use all the available memory and computational resources.

Big data algorithms generally function differently from more classical algorithms by having different needs in terms of underlying hardware. This has its direct implications for the types of algorithms that could be

used effectively and on the paradigm, the system's design is based on. In the current environment of the NS, the only viable option for data analysis is Hadoop in combination with PySpark.

Due to the lack of centralized knowledge on our part about trains and how they operate, it is difficult to interpret the data they transmit. As there is no manual that contains the location and structure of data, it is also a challenge to grasp which data is actually available and where it can be found. Data is stored in separate tables within the data warehouse, without necessarily keeping in mind how these could be combined in order to achieve more valuable information. As such collection of combined information may leave some columns sparsely filled. The fact that some data can be incomplete, ranging from NULL values to entire missing data points, requires even more knowledge about the data. Information about the current system is decentralized, so a lot of asking around and collaboration is needed.

Besides the challenge of incomplete data, there has to be dealt with excessive data. A mechanic triggers a lot of error codes on purpose during maintenance and these error codes are also sent to the database. There is no clear indication of the train throwing the error code was under maintenance at the time, so the database contains false data which needs to be filtered out. The challenge that comes with this is finding good parameters to filter maintenance data out. Another example of excessive data is that some problems cause multiple error codes to be thrown out. Because of this, it is hard to determine whether multiple error codes were caused by one problem or by multiple problems.

Solutions

The final product is an aggregation of multiple solutions. In this section, these solutions are explained. First of all, the design is extensively explained in the 'Design' section. Afterward, the chosen solutions for the implementation are given. These two sections together will give a detailed overview of the final product as a whole. The reasoning behind choosing the used frameworks and algorithms can be found in Appendix M: Research Report, in chapter 'Solutions'.

Design

This section starts off by shortly elaborating on the design goals that should be warranted by the system's design in order to achieve a successful project and then explains the design of the system that was made in terms of high-level architecture and more specific implementation details.

Design goals

The following design goals are a broad overview of what should be warranted by the design of the system, in order to best address the problem as defined in the problem statement. These design goals were defined at the start of the project in the *research report* (Appendix M) and have been key throughout the execution of the project.

1. **End-user satisfaction:** The final product should satisfy the long-term goal of the NS and should aid the end-users in helping them achieve this long-term goal. Customer collaboration should be valued in order to achieve this design goal.
2. **A quick insight into data:** Error trends should be visible in one eyesight by summarizing the data, as it should help the end-users find trends. Further, more detailed, information regarding each trend should also be easily accessible.
3. **Manageability:** The application will be made using techniques easily integrable by the NS, additionally the code-base should be able to be maintained well. Independent layers of analysis, as described in the vision, would warrant this design goal. The design of the system must also be easy to extend.
4. **Performance:** The application should not overload the server. Users should be able to obtain the results of their queries in a reasonable time, with respect to the complexity/size of the query.
5. **Scalability:** Collecting data has proven to be useful for the NS as they are trying to innovate in the direction of data analytics[2]. Plans for increasing the amount of collected data have been made. In order to accommodate for this increase in collected data, the application should be scalable.

[2] https://www.newcraftgroup.com/nl/nl-24-hour-data-challenge_students/

System Design

This section will specify the final architecture of the system. The system design from the research phase is implemented, with some extra additions. The use cases and requirements from the research phase can be found in Appendix M: Research report. The overall design will be explained in three sections: high-level design, individual class design, and the data flow diagram. For the high-level and individual class design, the core elements and their interaction will be explained to the extent of being able to implement a similar system as was made during this project. The data flow diagram, as can be seen in figure 2, will primarily focus on the interaction of the core components and the visualization of this interaction, from the perspective of a user.

High-level design

The system is designed to be a manageable, scalable, and high-performance system for running varying types of analyses, data filters, and post-processors. This is done with a running database connection which is open for end-user interaction and customization. The design of the system ought to make it easier to implement new types of analyses, filters, and post processors and make it easier to visualize the results of the analyses. It also provides an attractive user interface (UI) which allows the user to run analyses, customize analysis settings/parameters and a separate UI for data visualization (Power BI). The system comes with some pre-implemented analyses, filters, and post-processors. The high-level component diagram can be found in figure 1 and will be elaborated upon in this section.

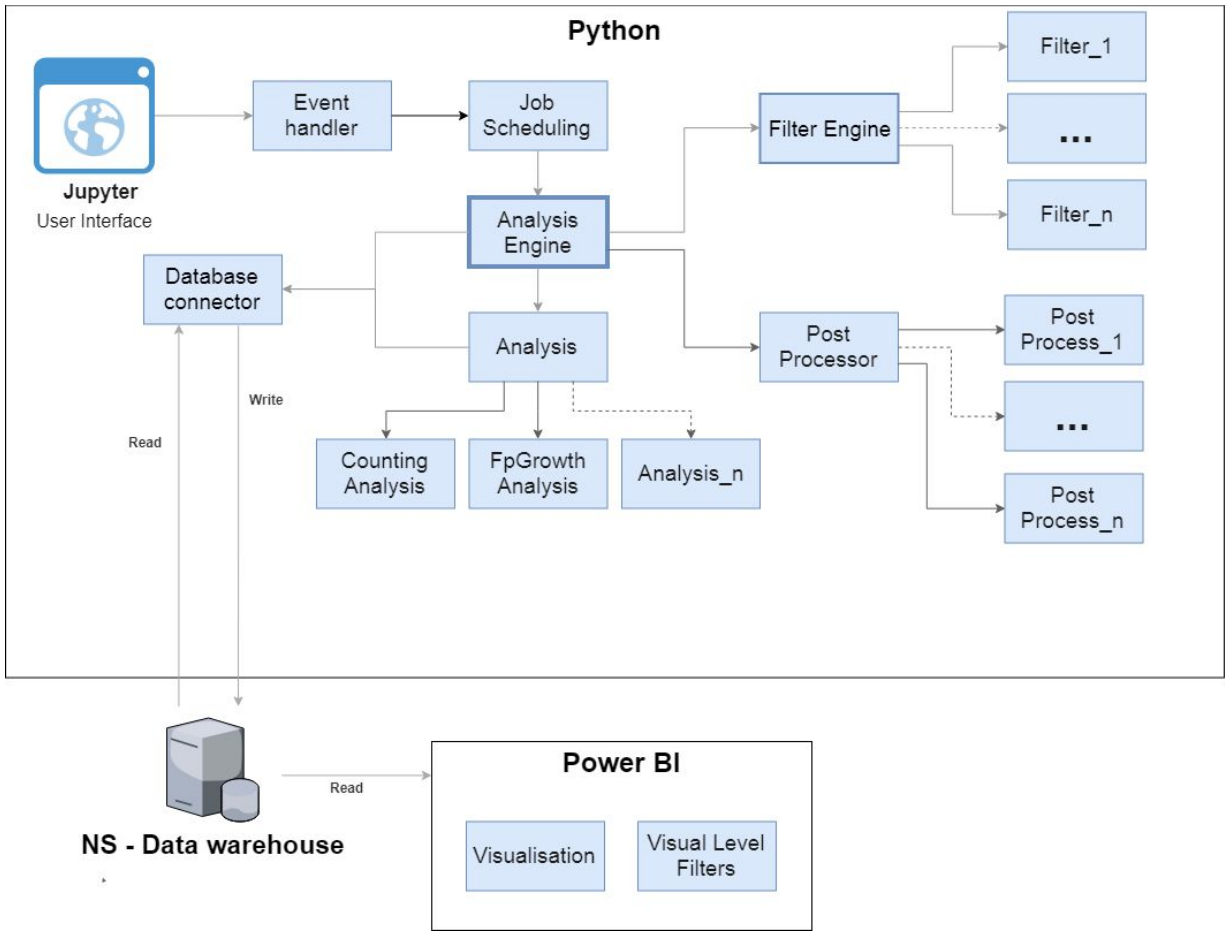

Figure 1: High-level component diagram.

To understand the architecture of the system, the following components are essential:
Analysis Engine: The Analysis Engine is the core driving the analysis of the data. It makes sure the data from the database is fetched by the responsible class and calls the filters module as specified by the user, calls the various types of analyses over this data, makes sure the analysis results are written to the database and ultimately calls the post-processing module, whose results are also written to the database. The Analysis Engine is what makes the individual components for data analysis work together.

Filter Engine: The Filter Engine is used by the Analysis Engine in order to filter the input data that is used for the analyses. The Filter Engine reads the settings for the filters and instantiates the filters using a decorator pattern. A filter may, for example, filter all diagnostic report codes that were thrown outside a certain interval or limit the selection to a certain subset of train series.

Analysis: Analyses explore and transform the (filtered) input data in order to discover error-code patterns for specific locations. There are currently two types of analyses: Counting Analysis and FP-Growth Analysis. For the former: every diagnostic code on each track section will be counted. This gives insight into track sections where a lot of error codes are thrown. This information is enriched with paired train delays and relative occurrences of the error with respect to the number of trains on the track section. For the latter, all diagnostic codes on each track are mined for patterns following the FP-Growth Frequent Pattern Mining algorithm by Han, Pei & Yin (2000). The patterns resulting from this type of analysis consist out of two or more diagnostic codes, which reveals something about errors which co-occur on a track section.

Post Processor: The Post Processor runs after all analyses have finished. The Post Processor is meant to process the found patterns. A typical use case is to post-process the trends for statistics, e.g. 'how many trends were on each track section?' At the moment the Post Processor is only used for collaborative filtering.

Jupyter UI: Jupyter[3], a browser-based notebook interface, is used to display the UI that allows the end-user to directly interact with the Analysis Engine. The user can specify settings for the filters that are to be applied and the user can create analysis jobs that will execute an analysis at a given time with the given settings. Through Jupyter the user can fill the NS data warehouse with data resulting from his analysis jobs. The Jupyter UI can be seen in Appendix F.

Power BI**:** Power BI[4] is used as the tool to visualize the data that was generated by running the analyses, by means of interactive reports. These reports are relatively easy to make and allow for a dynamic, attractive insight into the data. As for now, Power BI desktop does not allow for interaction with the Python code base and is solely used as a separate interface for visualization.

Class diagram

In this section, some technical system engineering details, that relate to the structure of the individual classes, will be covered. Mostly the interactions/designs that are essential to the understanding of the design of the system are explained. The classes follow the same structure as the high-level design above. The individual classes and their position in the system can be seen in Appendix D: UML Class diagram.
Some classes have an additional explanation in the section 'Chosen solutions'.

Writers: Writers are used for Create, Read, Update and Delete (CRUD) operations related to managing the filter settings and managing analysis jobs. Each writer inherits the parent Writer class, which includes the basic CRUD functionality, defines the FILE_PATH, which indicates where the Writer should read/write to and may be extended with extra functions needed for its task. There are three writers: TrendBlacklistWriter, SettingsWriter, and JobWriter.

Event Handler: The Event Handler functions as the middle layer between the functions present in the

[3] https://jupyter.org/
[4] *https://powerbi.microsoft.com/en-us/*

Jupyter UI and the back-end. Its primary function is to handle updates from the back-end to the front-end and to handle and validate user-input towards the back-end. Because this middle layer exists, it will be relatively easy to plug other UIs into it.

Analysis Queue: The Analysis Queue is a priority queue that receives analysis jobs from the JobListener and calls the Analysis Engine with the appropriate jobs one by one.

Database Connector: The Database Connector is the place where, as the name suggests, the connection with the database is handled. Tables get read out from here and loaded into workable data frames. The data frame also gets written back to a table here.

Counting Analysis: This analysis type goes over every track section and for each track section counts the number of times each error code type occurred.

FP-Growth Analysis: This analysis type goes over every track section and checks for each track section how many unique trains have thrown the same type of error code. When this is above a certain percentage, this error code type is shown.

Occurrences Recommender: The recommender system for occurrences is a form of post-processing. The recommender makes use of Collaborative Filtering, in order to recommend the user which track section or trend might become relevant in the future. The Occurrence Recommender is instantiated and called by the Post Processor.

Application flow

This subsection describes the application and data flow when a user uses the program. UML visualizing the full application flow including user interaction can be found in Appendix E: Application flow.

When the Jupyter UI is started a number of events happen and some objects are created in the background. Their interaction is visualized by the UI Data flow diagram in Appendix E. What the Jupyter UI looks like can be seen in Appendix E. Upon launch, the first object that is created is a JobListener. The Job Listener has an Analyses Queue object and the Analysis Queue object has an Analysis Engine. The Job Listener will check for newly created jobs and pass them to the Analysis Queue which will execute them one by one by calling the Analysis engine. Using the *start analysis* button, the user can create an analysis job. The then stored settings will be attached to the job. This job will, in turn, be passed to the Analysis Queue again.

The Analysis Engine first applies every filter specified by the user. This is done by calling its FilterEngine. The FilterEngine reads the settings for the filters and instantiates the filters using a decorator pattern. When the filters are applied to the input data, the Analysis Engine starts running the different analyses one by one. When an Analysis is finished, it returns a Spark DataFrame with found trends. These trends are converted to the format used in the database and thereafter written to the database. The database now contains the result of the analysis and the application terminates. This process is visualized in figure 2.

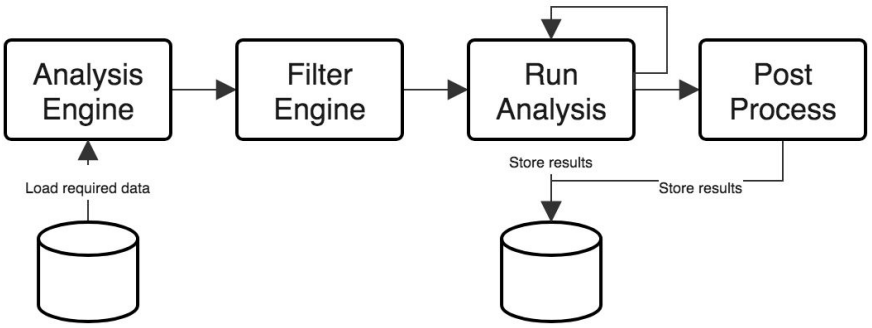

Figure 2: Data flow of the back-end Analysis Engine

After the results are stored in the database they can be fetched in Power BI and loaded in the there defined structure. This structure and the current state of the Power BI interface can be seen in Appendix G. Power BI allows flexible views and visualizations to be applied to the structured data.

The Power BI interface that was delivered alongside this project visualized the Counting Analysis, FP-Growth analysis, and Occurrences Recommender results. This visualization is highly interactive, allowing different perspectives of the data, and can be changed by applying visual level filters which filter the results based on certain conditions. Related figures can be found in Appendix G: Power BI.

The Counting Analysis result, as visualized in figure 19, displays the most important features of the patterns that were found for each track section from a top-level view. From this top-level view, two more detailed visualizations can be accessed: 'Track Section' and 'Error Codes', figure 20 and 21 respectively. The 'Track Section' view shows all the patterns that were found for the selected track section, shows which combined codes (FP-Growth patterns) often occurred and gives predictions for which patterns are likely to occur again (Occurrence Recommender). The 'Error Codes' view, in some ways similar to the 'Track Section' view, shows all the diagnostic error codes that were part of a (counting) pattern shown at the top-level view. This more detailed view is essential for giving the top-level patterns credibility for both the end-users at the NS and those who are going to have to act upon the supposed pattern that is causing a problem on a specific location. The FP-Growth Analysis result is visualized in a very similar manner as the Counting Analysis, this visualization can be seen in figure 22. It has one more detailed view which shows the diagnostic error codes that were part of the (FP-Growth) pattern that was selected. This view can be seen in figure 23.

Manageability

The system provides a framework in which several components can easily be added. The main points of extension are the Analysis classes, the Filter Engine filters, and Post Processor processes. This section will describe briefly how this extension can be done. Further details can be found in the Systems engineering guide in Appendix L.

To add a new analysis, a new class inheriting the Analysis class must be created in the analyses.py module. For this class, three required methods must be specified: execute(), convert_to_db_tables() and is_finished(). The Analysis Engine will immediately detect new analyses and instantiate them. The system

does not allow priority execution, as each analysis is expected to be fully independent of other analyses.

To add a new filter, a new class in the filters.py module which extends the FilterDecorator class has to be created. The __init__ and filter method from FilterDecorator have to be overwritten. In the __init__ method, the filter attributes can be set, the filter method gets an input Data Frame, removes rows from it by using a PySpark SQL filter[5], and returns the filtered Data Frame. One also has to instantiate the filter in the FilterEngine class. To make the filters configurable, the filter must be registered in the filters.json. Furthermore, in the FilterEngine class __init__ function, the filter must be configured as desired. All settings for filters are read in JSON format.

To add a new post-process, a new class inheriting the PostProcess class in the postprocessing.py module. For this class, two required methods must be specified: execute( ) and database_names( ). All post processes are instantiated by the Post Processor. Just like the analyses, all post processes are automatically instantiated after the coding guidelines have been followed. Post-processes support multiple results, as all results are returned as tuples of undefined sizes.

Chosen Solutions

The following section explains what the chosen solutions are and why they were chosen. Additionally, the problems that were found during the development of these solutions and the reasoning behind these solutions will be explained. The solutions that will be explained are FP-Growth, Filters, Collaborative filtering, the consideration of computing in Power BI and in Pyspark, the user interface and finally the database normalization.

FP-Growth

Single error codes might give different information than the combination of error codes. These combination patterns of error codes can be found using the FP-Growth algorithm. The purpose of this algorithm was to unveil some of the underlying patterns between multiple error codes which are statistically relevant (Mabroukeh & Ezeife, 2010). This should provide the end-users with a quick insight as to which combinations occur within the individual codes. This provides insight as to how codes causally relate to each other, which is a kind of analysis providing more analytical depth.

FP-Growth is one of the available Frequent Pattern Mining (FPM) algorithms (also see Appendix M for a comparison of FPM algorithms). For this algorithm, different lists of error codes are analyzed for frequent combinations of elements within lists. An example of this algorithm would be with groceries, where each basket of groceries forms a list. In figure 3, possible lists with pseudo-Python code and their respective outcome are given.

[5] http://spark.apache.org/docs/2.1.0/api/python/pyspark.sql.html

FP-Growth grocery code example

```
basket1 = ["apple", "banana"]
basket2 = ["apple", "kiwi", "apple", "banana"]  # This will become a set!
basket3 = ["orange"]
basket4 = ["apple", "banana", "kiwi"]
basket5 = ["razer"]

# Call the FP-Growth algorithm with a minimum support of 40%.
result = fp_growth([basket1, basket2, basket3, basket4, basket5], minSupport=0.4)
print(result)

>>>>
Frequent items sets:
["apple"] = 3
["banana"] = 3
["apple", "banana"] = 3
["apple", "banana", "kiwi"] = 2
```

Figure 3: FP-Growth grocery code example

In the above example, there are four different sets as the outcome. Note that the baskets at first were represented as lists but are converted to sets (even as the output) in the FP-Growth algorithm, i.e. duplicates are removed and the order is not important. The order of items, that is diagnostic error codes, could not be kept when using the PySpark FP-growth implementation. This property was not considered as important initially, as this sequentiality of items could be checked by looking at the activation time of the corresponding diagnostic error codes. Additionally, codes that have some causal relationship to each other will often appear on the same track section within a similar time frame. As a result, these sequences will often be able to be detected without initially preserving the order.

This example could be transformed to work with trains if each basket is replaced with a unique ride of a train. The content of the basket is replaced with the error codes thrown during the single ride. The baskets could also possibly have been replaced with unique trains, but this would make the algorithm rather train-specific, which is not intended by the scope, and would decrease the chances of finding location-specific patterns. Whenever all rides of a single train would form a set, there would be less possible patterns given the fact that duplicates are removed in sets.

To make the algorithm location-specific, the algorithm runs for each track section. This means that for each run of the algorithm, the frequent patterns are only on that specific track section. Besides making the analysis location-specific, running it for each track section also has the benefit that the memory usage is lowered. Several groupings are needed to make the sets, as described above, in order to run the algorithm. Whenever these grouping would be performed on all data, it would require massive amounts of

memory (RAM) to perform those operations. Consider hundreds of thousands of lines that need to be grouped multiple times This is now reduced to $\frac{1}{749}$ of all records for each grouping (if no filters are applied), as there are 749 distinct track sections as of the current state of the database table.
The example above was a typical example of memory issues during the development of the FP-Growth pattern mining algorithm. Because combinations are created, the memory usage can grow rapidly.

Specifying rides

As mentioned earlier, sets are formed from rides of a train. However, individual rides are not indicated in the database. Each train gets a train number, which can be used to identify individual rides. However, train numbers are not unique and may be used again on a new day.
Therefore, to uniquely identify rides on a day, a combination of the rolling stock (the individual train), the train number, and the date is needed. Still, some rides are excluded from this, as trains moving from a station toward a maintenance facility or service station usually do not have a train number. For these rides, some information is lost, as the same trains on the same day, on the same track section are identified as a single ride. The impact should be minimized for this, if assumed there is only a small chance for a train to make this ride multiple times on the same day. Only the same train with a return ride on the same day is collected as one single ride.
In the current implementation, this may be considered as a flaw in the algorithm, while it can also be considered a flaw in the data. The identification of single rides is already lost after the train number is removed. Therefore, if these individual rides should be identified as distinct rides, then a new identification method for these trains should be implemented.

The problem as described above was a typical example of missing and incomplete data, as well as inconsistent data. With the solution described above, the system is still able to distinguish different rides, while not losing too much information.

Minimum support problem

As was earlier mentioned, a minimum support is given to the FP-Growth algorithm. This minimum support specifies how many rides should support a pattern (i.e. have the combination of error codes in the ride) to be included for a new iteration. In such a new iteration, new combinations are made between the error codes. For example, sets [A] and [B] in a new iteration form a new set [A, B] if both [A] and [B] satisfy the minimum support. Assume the algorithm has a minimum support set to 0.05, i.e. 5% of the rides should support the pattern, then at least one in twenty rides on a single track section should include the pattern. Imagine a track section where less than or equal to twenty distinct train rides were. In this case, every error code generated by a train is considered a pattern, as $\frac{1}{19}$ is already more than 5%.
To prevent such a scenario, we introduced a variable specifying the minimum amount of supporting rides, which is by default set to two. The minimum support is incremented such that, whenever the minimum support is too low to require a support of two trains (in the case of default), it will still require two rides. For example, if the support is 0.05 and only has fifteen train rides based on the input of the FP-Growth algorithm on a single track section, then the minimum support will be adjusted to $2/15 = 0.13\overline{3}$. Similarly, if there are only two train rides, then the minimum support will be adjusted to $2/2 = 1.$ Whenever there are no

train rides or only a single train ride, the FP-Growth will not be executed. The above mentioned solution is executed before each call to the FP-Growth algorithm for each track section.
This problem was a typical case of inconsistent behavior on different track sections, where one track sections could be far less crowded than others.

The challenge

The FP-Growth solution was one of the solutions to our challenges. First of all, it was important to figure out what exactly would be a valid solution for location-specific problems. With the FP-Growth, combinations of different error codes can be found, where the order of the errors is irrelevant. By leaving out the order in the errors, some information is lost. However, as was thought from the start, this information is not required to find error patterns. Especially for rare occurrences of error patterns, it would be hard to find the error patterns in such a large data set, if the order was also included. With FP-Growth, the challenge of the unknown domain of location-specific errors was encountered for both the assumptions and the validation. During the development, most outcomes would have to be verified through assumptions, verification with knowledge of experts, and logical reasoning. From the start, it was hard to make assumptions about the data, as there were both unforeseen hiccups (e.g. inconsistency) in the data, as well as missing knowledge about trains.

Filters

In this section, some of the more advanced filters will be discussed. For each filter, the name of the filter is mentioned and a description for the filter is given. Overall, filters provide the general purpose of denoising the data for the end-users or the types of analysis, in order to make the data easier and more meaningful to analyze. With respect to denoising the data for the types of analysis, this is motivated with the 'garbage in, garbage out' (Kim, et al., 2016, O'hurley, et al., 2014) principle; if the data is not filtered properly, the result cannot be expected to be always right .

ServiceTrackFilter

All trains generate data during a ride but also during maintenance or repairs. These diagnostic codes come from the testing done by the engineers. The data generated by these trains should be excluded (to the extent possible) from all analyses, as trends found in the data would then indicate trends during maintenance. All errors generated during maintenance would then also be location-specific, because trains do not move to different track-sections during maintenance. For this section, two different types of maintenance locations are used: maintenance facilities and service stations (Appendix A: Glossary). Both mean the same for the outcome of the filter, as both should be excluded, but actually have different meanings in practice.

From the NS, some proposals for the filters were introduced. These proposals provided some of the technical knowledge that was needed to implement an effective filter. Below a list of their proposals is explained:

(1) A train is assumed to be in service [not to be confused with a service track] when it has a train number (may also be referred to as 'treinnummer').
(2) Whenever a train has no train number but the gap between their previous ride and next ride is less than 30 minutes, it is assumed to be in (near-)service and may, therefore, not be in maintenance. This time would be required for a train to startup and it should thus not be possible for it to have a quick maintenance run. All codes generated within this timespan are assumed to be dependent on either the previous ride or the startup for the next ride.
(3) The speed of the train is higher than 30 km/h. The reason for this is that some rides do not have an existing train number, e.g. the ride from Station Maastricht toward the maintenance facility of Maastricht. The ride toward the maintenance facility may also introduce location-specific problems and should, therefore, also be included.

In all other cases, the train is assumed to be in maintenance. Although these proposals filter out most errors generated during maintenance, they still do not filter out all maintenance errors and also filter out more than desired. Each of the three proposed filter options has some problems.

(1) The data is not always consistent and can not always be trusted. Some trains in maintenance still have a train number. Rules state that a train that is stranded on a track section should be removed within three hours. Therefore, we introduced an additional condition that any code generated should be generated on a track section, where at the time of being generated the train should be within a three-hour range from its next track section or its previous track section. This means that if a train departs from A toward B (i.e. track section A-B) and generates an error on location a location we call *x*, which is somewhere between the timetable points A and B. The time before *x* (i.e. the difference in time of the train being in position A and the train being in position *x*) and the time after *x* (i.e. the difference in time of the train being in position *x* and the train being in position B) summed should be less than three hours for the data to not be filtered out. The figure below illustrates the scenario described.

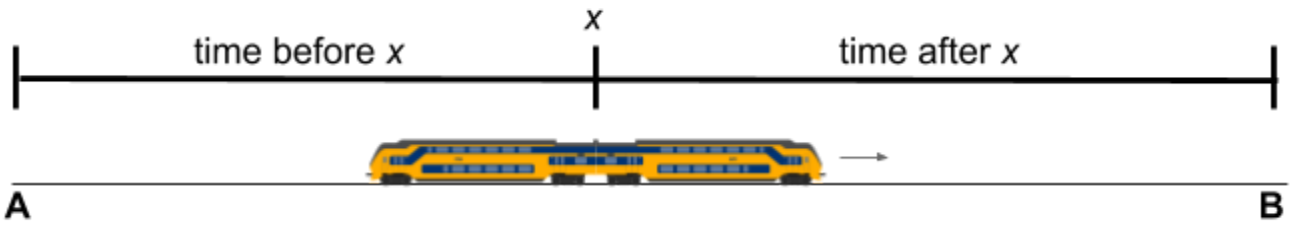

Figure 4: a train riding from A to B, which fires an error code on position x. The time before x and time after x summed should be less than three hours.

(2) The first proposal only stated that the gap between two rides should be 30 minutes. However, the time to start up a train takes 30 minutes but this does not include the time required for a train to shut down. This means that if a train was to stop and shutdown and start up again for a new ride, it would take more than the specified 30-minute gap. Therefore, the 30-minute gap was split into a

gap of 10 minutes after a ride and 30 minutes before a ride, which would be needed for a train operator to execute his full checklist. This means that the 30-minute gap is still included, but as described earlier, a 40-minute gap might be possible in some scenarios.
This condition may still be insufficient, as train operators may do it faster or slower than specified. It may, therefore, still be better to include a schedule for all train maintenance.

(3) When trains are only included if they drive faster than 30 km/h it is assumed that on all track sections between stations and maintenance facilities the driving speed is above 30 km/h. This may not be true for all track sections. Also, many trains that do not have real-time (RTM) data system have invalid GPS coordinates and their speed is unknown. These trains are still filtered out if they do not comply with the other two conditions. From a visit to a maintenance facility, indications were given that trains are defrosted while driving 5 km/h, while they normally drive a maximum speed of 40 km/h on the train yard. This shows that there may still be an improvement on the 30 km/h, as it now only includes the trains driving between 30-40 km/h (if the inaccuracy of GPS is not taken into account). According to Sathyamoorthy, Shafiii, Amin & Ali (2015), the error rate can be optimized to maximum 0.254 km/h with a change of satellite geometry included. This would mean that if the GPS speed in the NS database is corrected using the right method, the speed threshold of 30km/h could be lowered to somewhere around 5.254 km/h. Although, it is recommended to keep the threshold higher, or dependent on the number of available satellites at the moment of the measurement. The maximum error rate of Sathyamoorthy et al. (2015) was created with the availability of six satellites, while the trains mostly use similar numbers of satellites but are not guaranteed to.

Besides the conditions, it was still possible to find some cases of trains that did not comply with the above conditions but did not seem to be in maintenance. To test this all records were filtered using the negation of the above conditions (i.e. all records assumed to be fired during maintenance). The data that was left was filtered based on the name of the track section they were on. All track sections that could possibly be a service track were removed. First, all track sections with the exact name of all service stations were removed. This assumption was in some cases a very strong assumption. For example, all track sections with the name The Hague, Amsterdam, Rotterdam, and Utrecht were removed. Also, trains were filtered out if they did not seem to drive, i.e. the timetable point the train is coming from is equal to the timetable point it is driving to. Even after most records on track sections were removed, where even too many were removed, there were still trains that seemed to be where they should not be (see table 1 below). The trains there show some possible scenarios. (1) the data is corrupted; the trains are driving, not where the data says they are, or they should have a train number. (2) the trains are really not where they should be. Although the first scenario is far more likely, it is still interesting for the NS to further investigate this situation to verify the integrity of the data and also verify that the trains were not really ‘missing’.

Table 1: possible 'missing' trains. These trains do not have a train number but are not on a service station. These records may possibly indicate that some records are corrupted.

Property	Train #1	Train #2	Train #3	Train #4	Train #5	Train #6
rolling_stock_number	8707	9402	9555	9575	9575	9575
date_time	2017-08-06 14:46:54	2018-01-18 20:32:46	2018-01-24 13:32:02	2017-09-10 15:58:41	2017-09-10 14:52:29	2017-09-10 15:48:00
date_time_deact	2017-08-06 14:48:18	2018-01-18 20:33:04	2018-01-26 20:40:52	2017-09-10 16:14:42	2017-09-10 15:25:50	2017-09-10 15:49:36
track_section	Dtzo-Dtz	Odw-Gdg	Utg-Cas	Hlo-Utg	Hlo-Utg	Hlo-Utg
location_name_start	Delft Zuid Overloopwissel	Oudewater Wachtspoor	Uitgeest	Heiloo	Heiloo	Heiloo
gebiedtype_start	STATION	OVERLOOP	STATION	HALTE	HALTE	HALTE
location_name_eind	Delft Zuid	Gouda Goverwelle	Castricum	Uitgeest	Uitgeest	Uitgeest
gebiedtype_eind	STATION	STATION	HALTE	STATION	STATION	STATION
record_id	2500255	2746540	2760184	2681448	2681446	2681447

DuplicateFilter

Whenever a train fires an error, this is due to some conditional constraints in, for example, a thermometer. There are cases where a train fires the same error code multiple times within a few seconds. This may be due to many reasons. Most likely the error is not sent for a short while due to inconsistency and inaccuracy in a gauge. The problem is that, if no filter is applied, a train with a single problem sending a single error code for this problem seems less severe than a train, which has the same problem but sends five of the same error codes. To reduce this problem, we introduced a DuplicateFilter. The name implies that the records that are filtered out are duplicates, this is not per se the case. The filter merely decreases the chance of double counting for a single problem, as in most of these cases the cause of the problem is the

same but multiple codes were fired. For this filter, we recommend a five-second interval between codes. All codes between this interval are removed. From the NS an indication for the filter of 60 seconds was given but seemed to filter out more than desired. Also, as stated earlier, the codes are not duplicate and, therefore, removing too many records means that you ignore the fact that they were present.
Merging the records was not an option, as Hive uses an immutable data structure. Furthermore, changing the database table would mean that other users of the table would be affected by the changes.
An advantage of merging the records would be that problems with the same cause are seen as a single record. Ultimately this would be implemented for this system and all similar analysis scripts. However, this would come with the major drawback, where wrongly merged records would lose information. Imagine two records that are wrongly merged into one record. These will now be seen as one single record with a longer timespan. The information of these records being two distinct occurrences is lost. Also, analyses, where these presumed duplicates are expected to be distinct records, cannot be executed freely anymore, as the data suggests there are fewer records left.
Another option would be to duplicate the table with merged records but this would be a bad decision memory-wise. Especially with the future expansion of RTM trains, the number of records will increase significantly. Duplicating the database table would double this increase.

It is, however, important to notice that the order in which the filters are applied affects the outcome of the analysis. Whenever the filter is applied first, too many records might be removed. Take, for example, the case where the filter threshold is set to 60 seconds. Below the scenario is visualized. In the first scenario, the duplicates are removed first, which leads to all records to be removed. With the duplicates removed after all other filters, one record is left. This is because the record is not duplicate anymore, as the records it would be duplicate with are removed.

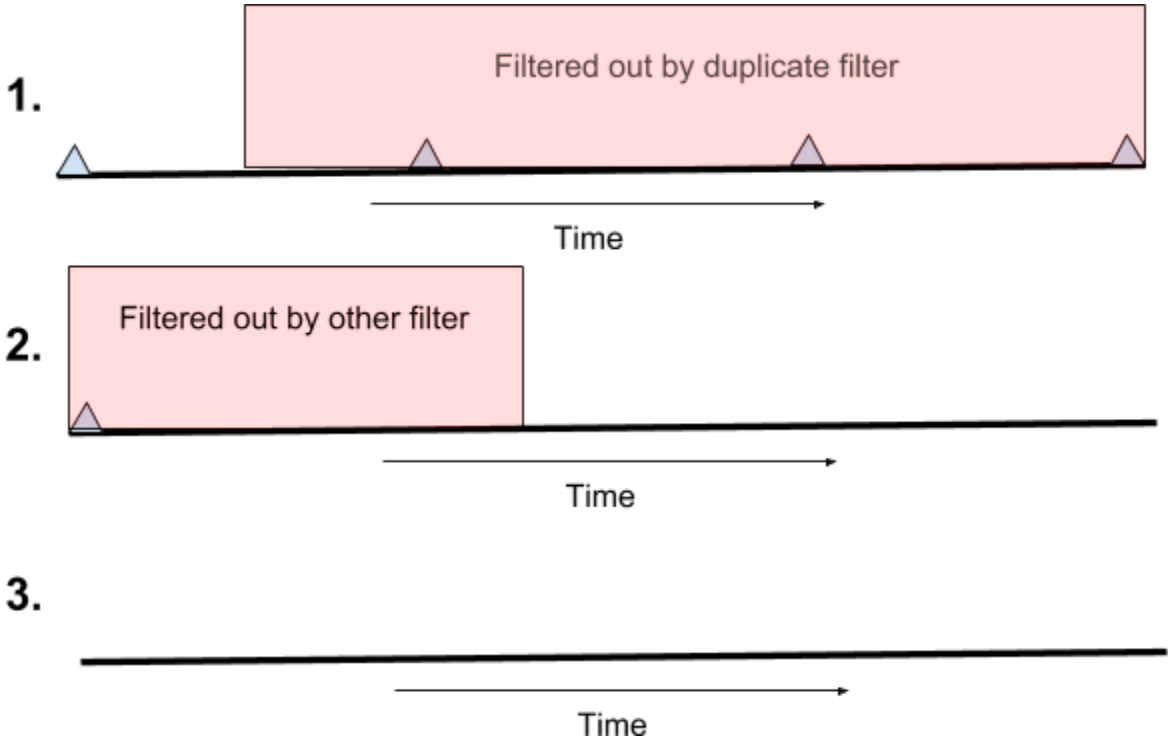

Figure 5: duplicates removed before all other filters are applied leads to no records being left.

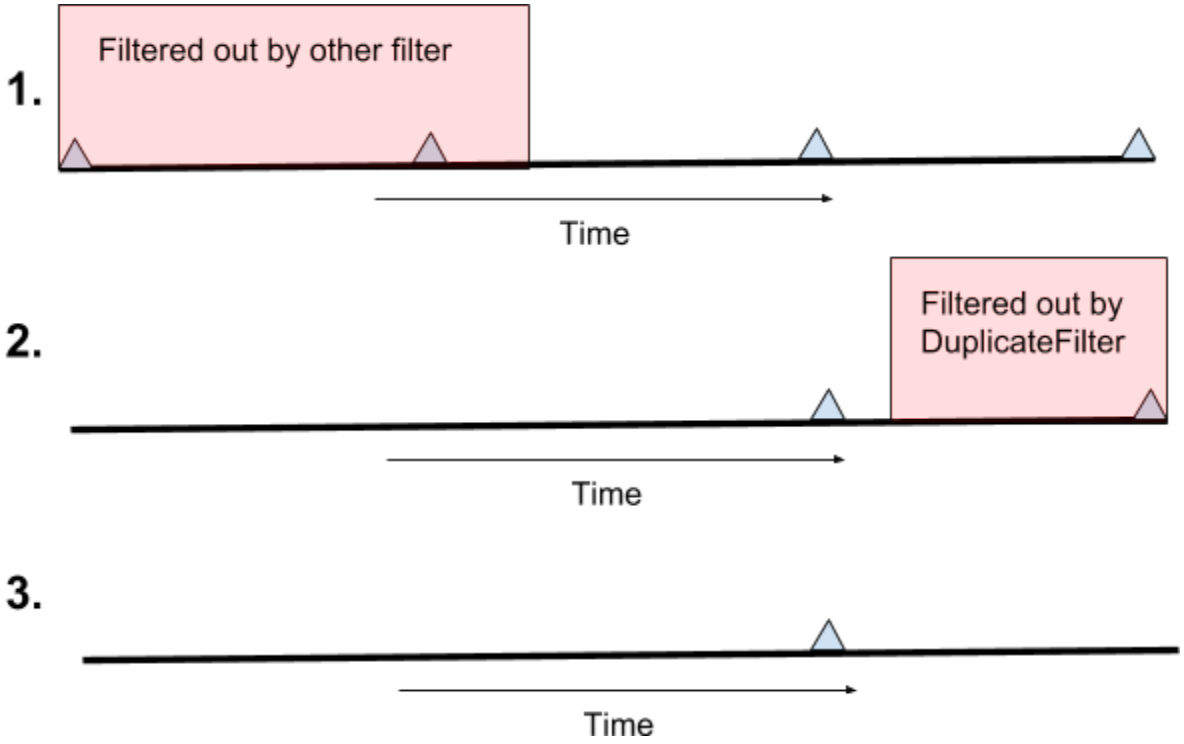

Figure 6: applying all filters except the duplicate filter, leads to one record being left. This is the desired behavior.

Filter influence

A difference in filter order can also benefit or hinder performance. For performance's sake, it is best to have the filter that filters out the most records execute first. Because of this, figuring out which filter filters the most records is essential. For each of the filters, the percentage of records left, the absolute amount of records left and the absolute amount of filtered out records of the total 636,011 records is calculated in the tables below.

In the first table, the individual filters are tested. The DuplicateFilter with the interval set to 60 seconds will maintain 67.5% of the records. As stated in the previous section about the DuplicateFilter, these 60 seconds proved to filter more out than desired. The interval set to 5 seconds will maintain 99.4% of the records. We believe that this interval is better since not too many records will be filtered out now. Furthermore, the FP-growth analysis is not affected by having too many of the same records, since it uses sets. The CountTrend analysis is however affected. But since it is better to keep a few too many records in the data and therefore give some track sections a bit more attention than necessary than it is to keep important records out of the data. Leaving out records out of the data can lead to missing an important pattern. We decided that a lower interval for the Duplicatefilter, like 5 seconds, may be better. However, since this interval is easily changeable and the difference is more easily visible with the interval of 60 seconds, the tables below will feature the DuplicateFilter with the 60 seconds interval.

The simplest ServiceTrackFilter, which only filters out the records with 'treinnummer' equal to null, maintains 44.7% of the records. This is quite a rough filter and will not be used as a final filter.
By adding the requirement of the gps_speed being faster than 30 km/h, the filter maintains 47.0% of the records. By decreasing this minimum speed to more than 5 km/h, the percentage of records left gradually increases up to 48.3%. However, with a speed of more than 0 km/h, this percentage makes an interesting bump to 82.4%. The reason for this bump is inaccuracy in the data is, mostly, the GPS sensor. A lot of GPS locations of the trains sway a bit due to this inaccuracy. Because of this, the gps_speed is often also a bit more than 0 and will thus not be filtered out in this last instance.
The program will use the filter configured at a 30 km/h. This could maybe be improved by lowering this speed, but with this data, it is uncertain whether the records that are removed are actually the correct records to remove.

When taking into account the amount of time needed to perform maintenance, only 49.8% of the total records remains when using a minimum gap of 30 minutes in between two rides for maintenance to be able to happen. When using 30 minutes before each ride and a minimum of 10 minutes after each ride, the percentage of records maintained is upped to 55.5%.

When using a filter for corrupted data, i.e. when the date_time_deact is null, 98.7% of the records remain. When only allowing RTM trains in this filter, 99.5% remains. This filter shows that there is more than 1.3% of the data that can not be trusted, which for this filter it is merely based on a single column. If more columns were filtered for their validity, more incorrect rows will likely be found. The reason for these corrupted records is probably caused, during their generation, by the train operators. Whenever these error codes

are generated, the train operator might shut down the train instantaneously, which does not allow the train to add a deactivation time to the error code.

Table 2: Individual filter influence on data. The amount of records filtered out by each of the filters is indicated in this table, both relative and absolute.

Filter influence on the data					
ID	**Name filter**	**Description of records left**	**Records left**	**Percentage left**	**Difference (filtered out)**
1.1	Duplicate filter. Values in seconds.	'dt_lastdiagcode' == null | 'dt_lastdiagcode' > 60	429,529	67.5%	206,482
1.2		'dt_lastdiagcode' == null | 'dt_lastdiagcode' > 5	618,637	99.4%	17,374
2	Simplest service track filter	'treinnummer' != null	284,595	44.7%	351,416
2.1.1	Service track filter with speed. Values in km/h.	'treinnummer' != null | 'gps_speed' > 30	299,236	47.0%	336,775
2.1.2		'treinnummer' != null | 'gps_speed' > 25	300,119	47.2%	335,892
2.1.3		'treinnummer' != null | 'gps_speed' > 20	301,208	47.4%	334,803
2.1.4		'treinnummer' != null | 'gps_speed' > 15	302,570	47.6%	333,441
2.1.5		'treinnummer' != null | 'gps_speed' > 10	304,364	47.9%	331,647
2.1.6		'treinnummer' != null | 'gps_speed' > 5	307,075	48.3%	328,936
2.1.7		'treinnummer' != null | 'gps_speed' > 0	523,811	82.4%	112,200
2.2.1	Service track filter with gap '30 min.' Values in seconds.	'treinnummer' != null | ('tijd_nadrp') + ('tijd_voordrp') <= 1800)	316,562	49.8%	319,449
2.2.2	Service track filter with 10 minutes after ride and 30 minutes before ride. Values in seconds.	'treinnummer' != null | ('tijd_nadrp') <= 600 | ('tijd_voordrp') <= 1800	353,217	55.5%	282,794
3	Corrupted filter	'date_time_deact' != null	627,875	98.7%	8,136
	Corrupted filter of which RTM	'date_time_deact' != null & 'rolling_stock_number' is RTM	632,993	99.5%	3,018
X	1 + 2.1.1 + 2.2.2 + 3	See 1, 2.1, 2.2 and 3	211,435	33.2%	424,576

Since the DuplicateFilter is a vital part of the filter system, having insight into the percentage deleted of each filter in combination with the DuplicateFilter is important. Therefore, all the previously mentioned filters can be seen in the table below, where the number of records left, percentage left and amount of records filtered out by the combination of the DuplicateFilter and the filter itself is written out.

The simplest ServiceTrackFilter in combination with the DuplicateFilter only preserves 27.9% of the records. As stated previously, this filter is quite rough.
When adding the minimum speed of more than 30 km/h, a lot more records are preserved: 29.4%. This gradually increases to 30.3% at more than 5 km/h. Once again, the minimum speed of more than 0 gives a huge spike due to inaccuracies.
The filter for corrupted data in combination with the DuplicateFilter filters just a bit more than the DuplicateFilter on its own, 66.3% and 67.5% respectively.

Table 3: Filter influence on data in combination with the DuplicateFilter. The amount of records filtered out by each of the filters in combination with the DuplicateFilter is indicated in this table, both relative and absolute.

Filter influence with the duplicate filter also applied (order independent).					
ID	**Name filter**	**Description of records left**	**Records left**	**Percentage left**	**Difference (filtered out)**
2	Simplest service track filter	'treinnummer' != null	177,277	27.9%	458,734
2.1.1	Service track filter with speed. Values in km/h.	'treinnummer' != null | 'gps_speed' > 30	187,077	29.4%	448,934
2.1.2		'treinnummer' != null | 'gps_speed' > 25	187,637	29.5%	448,374
2.1.3		'treinnummer' != null | 'gps_speed' > 20	188,305	29.6%	447,706
2.1.4		'treinnummer' != null | 'gps_speed' > 15	189,186	29,7%	446,825
2.1.5		'treinnummer' != null | 'gps_speed' > 10	190,454	29.9%	445,557
2.1.6		'treinnummer' != null | 'gps_speed' > 5	192,409	30,3%	443,602
2.1.7		'treinnummer' != null | 'gps_speed' > 0	337,856	53.1%	298,155
2.2.1	(see below)				
2.2.2	(see below)				
3	Corrupted filter	'date_time_deact' != null	421,566	66.3%	214,445
X	(see below)				

For the ServiceTrackFilter with the time before and after the maintenance sessions, the order of applying the filters matter. This is explained in the DuplicateFilter section and Figures 5 and 6.
When applying the DuplicateFilter first, the filter that keeps the records with 'treinnummer' null and having a less than a 30-minute gap between two trips maintains 31.9% of the records. While accounting for 10 minutes before and 30 minutes after a trip, this percentage rises to 36.4%.
When applying the DuplicateFilter after, these numbers are 33.4% and 37.9% respectively.

When using the most important filters, i.e. DuplicateFilter, ServiceTrackFilter with a minimum speed of more than 30 km/h, the ServiceTrackFilter with 10 minutes after a trip and 30 minutes before a trip and the filter for corrupted data, where the DuplicateFilter is applied first, the percentage of records maintained is 33.2%. When applying the DuplicateFilter after, this number is 37.7%. This is also the setup that is used in the program. This setup is used, since it does not filter out many useful records, but still filters out 37.7% of the records. When using a DuplicateFilter with a threshold of 5 seconds, this percentage is 48.5%.

Table 4: Filter influence on data in combination with the DuplicateFilter applied first. The amount of records filtered out by each of the filters in combination with the DuplicateFilter (applied first) is indicated in this table, both relative and absolute.

Filter influence with the duplicate filter applied first (order dependent).					
ID	**Name filter**	**Description of records left**	**Records left**	**Percentage left**	**Difference (filtered out)**
2.2.1	Service track filter with gap '30 min.' Values in seconds.	'treinnummer' != null | ('tijd_nadrp') + ('tijd_voordrp') <= 1800)	202,659	31.9%	433,352
2.2.2	Service track filter with 10 minutes after ride and 30 minutes before ride. Values in seconds.	'treinnummer' != null | ('tijd_nadrp') <= 600 | ('tijd_voordrp') <= 1800	231,628	36.4%	404,383
X	1 + 2.1.1 + 2.2.2 + 3	See 1, 2.1.1, 2.2.2 and 3	211,435	33.2%	424,576

Table 5: Filter influence on data in combination with the DuplicateFilter applied last. The amount of records filtered out by each of the filters in combination with the DuplicateFilter (applied last) is indicated in this table, both relative and absolute.

Filter influence with the duplicate filter applied last (order dependent).					
ID	**Name filter**	**Description of records left**	**Records left**	**Percentage left**	**Difference (filtered out)**
2.2.1	Service track filter with gap '30 min.' Values in seconds.	'treinnummer' != null | ('tijd_nadrp') + ('tijd_voordrp') <= 1800)	212,401	33.4%	423,610
2.2.2	Service track filter with 5 minutes after ride and 30 minutes before ride. Values in seconds.	'treinnummer' != null | ('tijd_nadrp') <= 600 | ('tijd_voordrp') <= 1800	241,176	37.9%	394,835
X	1 + 2.1.1 + 2.2.2 + 3	See 1, 2.1.1, 2.2.2 and 3	240,082	37.7%	395,929

The data can be used for further discussions on the filters, which should be held by the NS as they have the related engineering knowledge required. As can be seen from the results, the number of records left after applying all essential filters is close to one-third of all records. From this, the NS could discuss if this is a good result or not. Possibly too much is filtered out, while on the other hand a lot of data might be irrelevant or even useless. This data should not be used for verifying the integrity of the filters. What can be concluded is that with the filters, that are now assumed to be working and relevant, two-thirds of all records are filtered out.

Collaborative filtering

To get a certain sense of how important found patterns are, a system was introduced to recommend the user which track section or trend to look at. This recommendation system used the technology called 'Collaborative Filtering' or CF in short. The importance of a pattern suggestion is relevant for providing the end-user with a quick insight into what data is relevant to look at. From a system-engineering point of view, CF also tests the flexibility of the system and shows that it is indeed capable of extension, as it uses a different way of analyzing than the Counting- and FP-Growth analysis. The two types of analysis use the analysis engine, whereas CF uses a different post-processing module.

Normally, CF is applied for recommender systems for human recommender systems, like the Amazon webshop or the Netflix movie recommendation system. In these cases, the recommendation system is based on the rating of users for the products or movies. Take, for example, the case of Netflix, where multiple users watch movies and have rated the movies they have watched. Each of these ratings can then be used to determine which movies they may also like, based on the ratings of others. These prediction rating for these users will be based on the similarity of ratings between other users that already have watched the movie that will be predicted. A small example of how these ratings may look like in a matrix representation is given below.

Table 6: an example of User-Item Collaborative Filtering as can be used for a Netflix movie recommendation system. Each of the values represents a user rating for a movie.

		Movies				
		Terminator	The Godfather	Fight Club	Gladiator	Saving Private Ryan
Users	John	5	2	1	4	5
	Susie	2	5			3
	Mona	2		2	4	2
	Ann		2			
	Tyrone	5	1		5	

In this example, a User-Item CF filtering is represented as a matrix where each filled-in value for a user U*x* and movie M*y* is a rating given by the user for that movie. All blanks in the matrix need to be filled in using the CF algorithm in order to give recommendations for the users. A high-prediction value for a movie means that the user is likely to enjoy watching the movie. For example, a rating for user Tyrone and movie Saving Private Ryan can be predicted by finding the similar users.
To verify the algorithm, all data is split into a training set and a test set. The training set is used for finding the right latent factors, while the test set is used to verify if the latent factors work as expected. For the test set, some ratings are removed and predicted again. The difference between the prediction and the actual rating is the error rate. The predictions are made in such a way that the error rate, or in our case the Root Mean Squared Error (RMSE) is minimized for the test set. The RMSE is defined as $\sqrt{\sum \frac{(prediction - actual)^2}{N}}$, where N is the number of predicted ratings in the test set.

The system used in our application works differently than the recommender systems used for the earlier mentioned examples, like Amazon and Netflix. According to Leskovec, Rajaraman & Ullman (2011, p. 309-310), Collaborative Filtering is used for 1. product recommendations (e.g. Amazon), 2. Movie recommendations (e.g. Netflix), or 3. News articles (e.g. identify articles of interest based on earlier read articles).

To transform this system to work with the track sections and trends, we had to find a rating system in our application. This application uses the frequencies of trends as a replacement of the ratings. These frequencies are defined as the number of rides contributing to the individual trend. This means that a single train unit with two rides contributing to the trend adds two points to the rating (or frequency). The above example in table 6 can be transformed to the track section to trend recommendation system by substituting the users with track sections and the movies with the distinct trends. The same example, but transformed, can be found in table 7.

Table 7: an example of Track section-Pattern Collaborative Filtering for a pattern recommendation system. Each value represents the frequency of train rides for a trend on a track section.

		Error code pattern				
		["HSP020","HSP022"]	REM234	KLM638	DVR090	["LSP673","LSP682"]
Track sections	Nvp-Rvbr	5	2	1	4	5
	Dvnk-Vkbr	2	5			3
	Gw-Hrmk	2		2	4	2
	Brdl-Hr		2			
	Dta-Rsw	5	1		5	

In the above example, again the same ratings are used but this time they are the rating formed from the actual frequency of trains on a track section for a single trend. The blank fields can be filled in based on the frequencies found on track sections. With this is assumed that track sections may act similar to each other and, therefore, might lead to the same behavioral patterns for their trends. The blank field for track section Dta-Rsw and error code pattern ["LSP673","LSP682"] filled in will mean that the value given in that cell represents the expected number of train rides on the track section for that trend.

Overestimation

Also important to note for the algorithm is that more crowded track sections, like stations, are more likely to be recommended as important tracks to look at. These track sections are likely to be recommended as their predicted frequencies are higher than other less crowded track sections. This behavior could lead to some 'important' track sections not to be recommended as important while their trend is in fact important. However, it also has desired behavior, as more crowded track sections are also the track sections with the highest priority of being fixed when a location-specific problem is located. Location-specific problems in crowded track sections affect more trains than less crowded track sections and, therefore, their economic impact could be higher.

To still minimize this effect a little but also to minimize overestimation, the training data is filtered for outliers. For now, training data entries are considered an outlier if the frequency has a value higher than three standard deviations from the mean. It is calculated as: $data.filter(|f - \bar{f}| < 3 * stddev)$, where f is the frequency, $\bar{f}$ is the mean of all frequencies, and $stddev$ is the standard deviation of all frequencies. The filter method keeps all data that suffices the given formula, i.e. all values above three times the standard deviation is removed.

Ensemble method

To make some predictions more accurate overall, a simple ensemble method using the average of all predictions is applied. Although the test sets showed a much better RMSE, still the strong assumption is made that this actually improves the predictions. For the ensemble method to work "A necessary and sufficient condition for an ensemble of classifiers [a single prediction] to be more accurate than any of its individual members is if the classifiers are accurate and diverse" (Dietterich, 2000; Hansen & Salamon, 1990). However, for the assumption, the former (diverse) can be easily be done, as the average is sometimes higher but also lower than the CF prediction. Which means that both predictions make different errors on the same new data point (Dietterich, 2000). Whereas for the accuracy, the classifier must perform better than a random guess (Dietterich, 2000). Still, there is much room for improvement, as the average classifier is dependent on the CF classifier.

Discussion

The use of CF for recommending trends for track sections is very experimental. As mentioned earlier by Leskovec et al. (2011, p. 309-310), the applications of CF is based on user behavior, preference and item and user similarity. In our application, the system is merely based on track section behavior and track similarity. The track sections do not have preferences if assumed that a preference defined as "a greater liking for one alternative over another or others"[6] in this case means an aware decision of choosing a trend over another trend.
Due to the fact that there are no users involved in the recommendation system, it is assumed that the human factor is not key to the CF algorithm. This assumption should be safe, as there are still ratings generated for the track sections. However, if many of the trends are only on one track section and not on any others, the matrix might become too sparse. Although the algorithm is optimized for sparse matrices, its performance can be badly influenced when the matrix does not have enough entries for it to generate predictions. This problem becomes bigger when there are also not enough trends found. In these cases, it might be better to not recommend to the user at all, as the prediction rating would be overfitting to the few samples given. In the current implementation of the program, if there are not enough training data, no recommendations are given. Also, if there RMSE is too high to give an accurate prediction, it will also be skipped. Too high is as default defined with an RMSE of 20.0 but a lower value might be desirable.

To further verify the validity of the algorithm, verification data is required. This can be generated by using the application and saving the found error patterns, that were identified as location-specific problems. Verification could also be implemented from the same post-processing system, as the CF algorithm, which also emphasizes the challenge that was put in the post-processing system and the required expandability of the system.

The post-processing and thereby the CF was introduced after the initial system was implemented. To verify our system design for its expandability, the post-processing was introduced. This showed that the system

[6] https://en.oxforddictionaries.com/definition/preference

could be expanded in a clean and easy fashion, while it was also consistent with other components of the system. More on this will be elaborated in the ‘Evaluation’ chapter on the system design.

Counting analysis

The Counting Analysis makes use of simple counting of error codes and finally relatively compares the occurrences of the error codes with different track sections. The simplicity of the algorithm may also be seen as the strength of the algorithm. The speed performance is high, while it is still able to find many possible error patterns for different locations. Its simplicity also makes it a robust and reliable algorithm, which is good for the developers when exploring the unknown domain as a benchmark. The simplicity, which in terms comes with traceability and trustworthiness, makes it easier for the end-user to trust the results that 'the new analysis system' provides.

To make sure more crowded track sections are not always seen as more likely to have an error pattern, the error codes are also relatively compared with the number of trains passing the track section. The data of the passing trains is gathered from a different data source than the diagnostic error codes. To be able to give found patterns more prioritization, the delay paired with the error pattern is calculated. This data also comes from a different data source. The different data sources used different type formats, column names, and sometimes even different definitions, which emphasizes the challenge of data inconsistency. With the implementation of the Counting Analysis, the system proofs that it is able to use multiple data sources in analyses. Furthermore, both this counting analysis and the above-mentioned FP-Growth analysis have the same signature and parent class. This shows the extensibility of the framework built into the system.

Power BI and PySpark computation

During the early execution phase of the scope, a decision has been made regarding the storage of the patterns found by the counting analysis in the database and how this method of storing these patterns would impact the performance of the application. Some of the attributes of count trends rely on summarized data collected from individual diagnostic error rows. One example of such a property is the amount of individual diagnostic rows that are related to a railway section for a given time frame.

A static time frame for each trend was chosen, such that the number of occurrences would not change by adjusting the time frame. The main motivation for such a static time frame was the performance of Power BI, as opposed to PySpark; PySpark is able to collect the summarized attributes for count patterns faster than Power BI is. Also, with the plans to increase the number of diagnostic error rows reported by the train sensors of the NS Power BI could easily turn into a bottleneck in terms of performance, perhaps to the point of it being hard to use. Additionally, PySpark can more easily be used to apply complex filters to create the conditions for which the summarized attributes should be collected, this would make this a more flexible option. For example, wanting to compare consecutive winters can be more easily configured from Pyspark’s side than Power BI’s side.

Half-way the project, it was discovered that the performance of Power BI was fast enough to deliver summarized attributes of the Counting Analysis from corresponding diagnose rows in less than a minute,

which is acceptable. This discovery was discussed within the team. After some discussion, the decision was made to stick to our initial choice of a static time frame for each found pattern, but to also include the possibility to use Power BI to compute summarized attributes with a dynamic time frame.

This decision was made for several reasons, some similar to the motivation of the initial choice. The flexibility in filter application was as strong of an argument as before. The performance of Power BI was still taken into consideration, as performance guarantees are hard to predict as the NS may want to increase their rate of data collection. The system is created such that it combines PySpark attributes with Power BI attributes, where the removal of the user interface does not impact the PySpark attributes. With the implementation of both, we think to have made a choice that can most likely support future demand and provide useful information.

Additional motivation came from the impact changes would have to our system. The system was built in such a way, that it best supports static attribute collection and changing mid-way would likely cause a lot of difficulties, more than it is worth. The same results, provided by the dynamic attributes of Power BI, can be obtained by, for example, defining the time frame beforehand. As such, the scope was not adjusted based on this discovery.

Centralized location of interaction

Initially, the design included one central user interface (UI) through which the end-users could change settings, initialize new analyses and apply filters: Power BI. Having a centralized environment is important for effectively spreading knowledge throughout the company and providing a central framework from which new types of analysis could be started, which prevents duplicate effort and saves on costs.

Early on the discovery was made that Power BI does not allow for a connection between its interface and any other language, that is unless Power BI Embedded[7] is used. However, given its pricing of approximately 5000 euro each month and available, cheaper solutions the Power BI Embedded solution is a hard business case to defend. Moreover, in the current environment of the NS, direct connections with the cluster and Power BI would require a connection between two different,strongly separated environments.

It is not possible to create a user interface in Power BI to interact with the Hadoop cluster, because the cluster is a closed system. This means a user interface has to be created in a different environment, which still has to be understandable for the end users of the program. This adds to the challenge of creating an integrated system, divided over two applications.

The UI was changed to have two separate interfaces: Power BI and a Jupyter Notebook. Power BI was still used for data visualization, as it does an excellent job in doing just that, but Jupyter Notebook was used to display a UI through which the end-users could schedule their analyses and specify the settings to use for those analyses. In order to make integrating any UI with the back-end, the Event Handler has been made. The Event Handler is further elaborated upon in the *Class Diagram* section above. This mitigates the

[7] https://azure.microsoft.com/nl-nl/pricing/details/power-bi-embedded/

problem of having to resort to two separate locations of interaction.

Database Analyses Normalization

Removing redundancy in the data, that was stored for the analyses, was a challenge when designing the database structure. Counting Analyses can have an overlap in time and as such they share occurrences of diagnostic error codes. The idea was to reduce the possible redundancy due to overlap. As a solution to this problem, linking tables were created where each pattern would store links. The links would have in one column the 'trend_id', identifying the pattern, and in the other column the reference to the occurrence's corresponding row: 'record_id'.

This worked well at the start of the project when the multi-user and multi-analysis context was not fully thought through. After allowing multiple users to store multiple analyses an extra condition constraint was added, being that trend_ids could have multiple record_ids and record_ids could have multiple trend_ids. This is a many-to-many relationship, which is not supported by Power BI. This would not be a problem in a more classical SQL oriented visualization tool, Power BI is DAX oriented. The solution was to merge the linking tables with the tables storing the patterns on 'trend_id' with an inner join. This caused redundancy which initially was hoped to be prevented. However, this can not be avoided when using Power BI. The initial way in which the data was stored does not include this redundancy. It is within Power BI that the redundancy is introduced, which allows this redundancy to be removed when using the data in another environment than Power BI.

The impact of this introduced redundancy is moderate. For each analysis there is approximately an average of 350 patterns, each pattern has on average seven corresponding occurrences, see Figure 7 for reference. This means the amount of diagnose rows that have to be stored increases by a factor six, resulting in an average 2100 additional rows per analysis. This is fairly limited, in term of data storage redundancy. The most significant part of the impact comes from Power BI having to do these joins. This currently takes a really long-time, for smaller joins it adds about 50 minutes of loading time, for larger join the time becomes significantly longer to the point of being unreasonable. This problem may be solved by doing the join within Pyspark, which removes the most significant part of the problem.

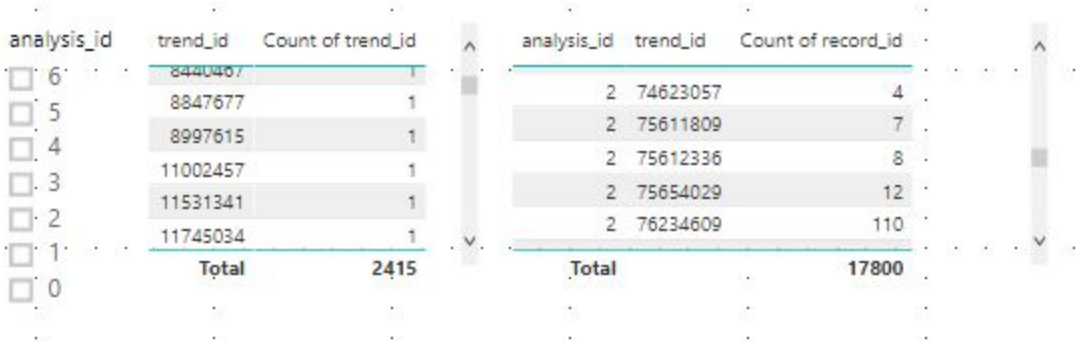

Figure 7: Amount of analyses, corresponding patterns and corresponding occurrences.

Discussion

In this section, the original problem description and chosen solutions which address the problem will be discussed. The methodology influencing the process of the project, which influences the former two, and the ethical evaluation, placing the project as a whole in a larger social-economic picture, are also discussed.

Problem approach

This section explains how the original problem statement was changed in order to better address the underlying problem and how this shift impacted the way the underlying problem was and will be able to be addressed.

Problem statement comparison

The original problem project description provided by the client (Appendix C) differs from the problem description at the start of this report. Whereas the higher-level goal of the project remains the same, that is improving reliability and punctuality of trains. The description of what the NS said they needed and what was proposed they needed differs. This shift is characterized by moving from a short-term analysis tool to a long-term analysis tool and a system for making the analysis environment more centralized. The problem description of this project also moves away from trying to give warnings or directly trying to observe the root of incidents, as this required engineering expertise specifically related to the kind of mechanical engineering and context the NS uses. This knowledge was not present within the team, nor did it seem like something that should be included within the problem statement, as making a system that would support new analysis/patterns would allow those with the expertise to move towards a more extensive system with more potential that would be able to reveal some of the roots of the problems and building a new system relates more to the team's area of expertise. As such, these choices were made in changing the problem statements.

Validating the system

Validating the usefulness of the analyses, and in terms validating the premise that the suggested system would indeed be able to improve reliability and punctuality, is an important part for evaluating whether the system may be worth expanding upon. The created application is able to find some interesting error patterns. Due to the lack of validation data, the knowledge of experts was needed on validating found error patterns. Ultimately, the error patterns should be verified with real examples of location-specific problems and through practical examinations on the track sections themselves. During the final demo with the end-users, multiple error code patterns were discovered and acknowledged as relevant and meaningful. A

subset of interesting error code patterns can be found in table 8. Some of the found error patterns are explained, in order to specify their possible importance.

Table 8: Subset of interesting error code patterns found during demo with experts.

	Analysis Type	Track Section	Diagnostic Codes	Error description
1	Fp-Growth	[redacted]	[HSP020, HSP022]	Power failure intervention.
2	Counting	[redacted]	DIA031	Two operating cabins are being connected.
3	Counting	[redacted]	ATB073	Change of cabins was not sufficiently carried out
4	Counting	[redacted]	ATB034, ATB035, ATB032	Signal in the rail is malformed
5	Fp-Growth	[redacted]	[REM110, REM111]	Train operators are doing their break test, which is actually good behavior.

The first error pattern is significant because the train probably went into a slip, which is very bad for the wheels. The error can be explained by the fact that this error is located at a bridge, which has its own reverse current system. If this system fails, it can result in a power failure intervention. The error was unknown for this location but with the information the program gave, it could immediately be identified as a problem. The reaction to this error is that the infrastructure of this point has to be improved.

The second error pattern is important because it could cause delays. The train operator did something wrong while connecting cabins. The reaction to this error is that this could be prevented by adapting the training the train operator gets. Trainers in the area of the error could be notified to give more attention to connecting cabins. Interesting to this pattern was that the pattern was only found for a certain track section (station) and not on other track sections. This means that the training should be adapted especially in that area.

The fourth error pattern is interesting, as ATB032 was earlier expected to be location-specific by the NS. Now, more of a confirmation is added to this hypothesis. ATB034 and ATB035 are very similar diagnostic error codes and are also considered location-specific. All three are due to the infrastructure and can be fixed by ProRail.

The fifth error pattern is also interesting, as it does not indicate something wrong, in contrary, it shows good behavior. Train operators are doing break tests on specific track sections, which is a standard procedure which should be carried out. It is known that this procedure is not always carried out. It can now be confirmed that this location does it more consistently than other locations. This information could, for example, be used to compliment the train operators on this track section for their behavior.

The most effective way of validating the system was to use the knowledge of the experts to determine whether the provided patterns would indeed lead to useful insights, other methods of validation were limited.
The primary reasons for limited methods of validation were a lack of traceable data to validate patterns with. The system is a new concept within the RTMA environment, which removes the positive of benchmarking. Also, a lack of time did not allow further extension on creating validation data. The lack of traceable data means that it was not possible, within the current implementation of the system, to trace any measure of severity of a pattern, be it travel delay or customer satisfaction, back to patterns that were found. This was not possible as the system never focused on finding the cause of certain patterns, which is explained in the section above. If there was time for further validation, then it would be possible to keep track of whether location-specific problems would actually decrease.

Requirements

The following section is divided into three sections: ‘Design goals’, ‘System Design’ and ‘Scope’. The ‘Design goals’ section is an evaluation of the application as a whole that verifies whether the application conforms to the previously set design goals. The ‘System Design’ section evaluates the design of the system, in particular the expansion possibilities. The ‘Scope’ section is a brief evaluation of the more specific requirements of the system. As such, the project is validated at various levels of abstraction.

Design goals

The original research report formulated five design goals, which gave a broad overview of the characteristics our final product should have. The design goals were: end-user satisfaction, a quick insight into data, manageability, performance, and scalability. These design goals give a good idea of what the product was/is to become and will thus give a good benchmark for a measure of project success. In this section, we will give our view on the design goals, if we think they were met, and how we think the design goals were met.

End-user satisfaction

In order to warrant these design goals, we held multiple demos with the client and two possible end-users. With the demos, the client could be informed of the progression but also could be used to test the relevance of the program. In the last and second last demo, actual results obtained from real data could be shown. These results could be validated with the expertise of the end-users. These demos showed that the program was able to find location-specific error patterns. From the enthusiastic reactions, we could measure the relevance of the found patterns. From these reactions, we could conclude that for them the program is built to their satisfaction. Moreover, as the error patterns proposed by the system were approved to their belief, the system seemed to operate to their expectations.

We also held one demo with a possible end-user who was not involved with the project from the beginning of the process. He was unaware of the project and could, with the demo, give his expectations from such a program. From this demo, different viewing points were given and it could be verified if the program fitted with his view. From his reactions, at that time, we could conclude that we were on track with his expectations.

Quick insight into data

To break down the problem into multiple levels, it could first be compared with simply giving a glance at the data in the database. This is clearly not a 'quick insight' and takes an employee of the NS much time to make a sensible statement. Also, making new individual scripts (outside this application) is not faster, as these scripts require reproduced parts of the framework. By implementing components of the framework provided by our application, it is easier to create new analyses, filters, and scripts. Components that could be (re)used are, amongst others, instantiation, integration, and inheritance of other classes. Also, using the framework, each script follows the same conventions, which improves the consistency amongst all filters, analyses, and post-processes. Hence, the framework has a huge advantage over individual scripts.

Besides the scripts giving an insight in the data by finding the patterns, the Power BI integration even expands the insight. Power BI shows multiple types of patterns (e.g. Statistical patterns and FP-Growth patterns) in the same environment. Furthermore, the Power BI interface gives an exact location on a map to each error code or pattern found (where for the patterns the average of all records is used). With this map and the signature (e.g. error codes A with B) of the pattern found and the location on the map, the user can easily pinpoint the location and even in some cases in one glance pinpoint the actual cause.

Manageability

Manageability also was one of our five design goals. Throughout the development of the project, we made the project with this in mind.

To further support the manageability, several documents for the user and system engineer were made. For the user, there is a user guide. The user guide can be used for installation instructions and the first steps into the program. For the system engineer, two documents were created: the Systems engineering guide and the UML.
The former can be used for the explanation of components within the program, how these components work, and how they can be expanded.
The user guide can be used to inform the user about the first steps to undertake within the program. The guide also explains the functioning of the program, which allows the user to also understand the behavior of the application. The user guide is not added to this document, to prevent the leakage of classified information. The Systems engineering guide is added as appendix L.

Performance and scalability

For the program, performance is an important aspect. This is closely related to scalability, since having enough performance allows for the scalability to be high as well. According to the NS, the minimal desired analysis timespan should cover a day, week, month, half a year, and possibly a year or longer. With the current implementation of the system, the system can make analyses of a timespan covering at least one and a half year. This means that the goals of a year analysis (or longer) were reached.

Regarding speed performance, the system is able to process analyses with a timespan of a year within an hour during quiet hours of the cluster. Through a scheduled system, the NS can schedule the analyses during these quiet hours. The speed performance is highly dependent on the server. Whenever there are more scripts running, the performance can significantly be reduced. Therefore, no strong guarantees on the speed performance can be given. However, using the scheduled system and reasonable speeds during an average usage capacity on the server, the performance of the program can be considered sufficient. Also, in the recommendations below, a recommendation is made, in order to reach higher speed performances in the future. Below, a table describes some of the measurements of different runs. These values can be used to get an indication of the speed of the system. However, it should be noted that the performance of the program is highly dependent on the cluster and, therefore, different results can be expected, as mentioned previously.

Table 9: Different measurements of runs with the application, where the days are varying. The execution time (in minutes and seconds) varies on the number of records, the content analyzed, and the performance of the cluster.

Run	Days	Execution time (MM:SS)
1	365	47:26
2	31	20:12
3	7	9:02

System Design

Analysis and PostProcessor

In this section, both the analyses and post-process classes will be discussed for their system design. All classes involved in both the processes will be elaborated on how they should be extended. These two processes (and the filter process) will, after the delivery, most likely be subject to changes. In this section, first, the different classes in each of the processes are explained and secondly how each of the components can be expanded is explained.

Both the analyses and post-process components were designed with future extension in mind. Also, both are very similar to each other. The difference between both is that all the analyses must be finished before

the post-processing can be started and, furthermore, the results of the analyses is used for the post-processes. For the analyses, there are two main components: the AnalysisEngine and the Analysis components. The former can be seen as the master, while the latter can be seen as the slave. The AnalysisEngine is responsible for initiating all Analysis and making sure they are all executed. Also, whenever an analysis is divided into sub-analyses, the engine makes sure all sub-analyses are temporarily written and eventually are all written together to a final state in the database. An analysis with such a sub-analyses system is the FP-Growth analysis, where each track section is a sub-analysis.
For an analysis to be initiated and called, it must follow the contract of an Analysis class. This contract defines several methods that should be extended with functionality specified for the analysis. Similarly, all post-processes must follow the contract defined in the Analysis class. See the Systems engineering guide Appendix L for more information on the contract methods. Each class in the same module or a module imported into the analysis_engine module is automatically instantiated and executed. A developer does not have to do more than to focus on the analysis itself. For the post-processes, the same system is applied. Whenever a developer created a new class that extends the PostProcess class, it is immediately picked up by the PostProcessor, instantiated, and executed.

Scope

The degree to which the scope was satisfied will be evaluated here. The scope was translated into a MoSCoW list during the research phase. Most of the items in the Must, Should and Could section were able to be implemented. Exceptions on which scope items were changed are formulated in the scope changes (Appendix B). In hindsight, the initial scope was a bit thin with respect to the amount of time its implementation ended up taking and was, therefore, extended with more features. The size of the scope and the schedule is further discussed under Methodology in the schedule section below.

SIG feedback

The feedback the system got from SIG regarding code quality was very positive. The code scored 4.2 stars, which means the code is above average maintainable. The tips they gave were to decrease the unit size, add more tests, and keep up the good work until the end of the project. The full feedback can be read (in Dutch) in Appendix H: SIG feedback.
The feedback was addressed by maintaining the code quality at the time by keeping thorough peer reviewing each others code and, as they suggested, the unit size was decreased by splitting methods. Also, more tests were added. By making the unit sizes smaller, testing was made easier. Furthermore, it makes the code more readable, which is essential when the NS takes over the responsibility of the code. As mentioned earlier in the challenges, there was no option for a coverage tool within the NS environment. An exact indication of line or branch coverage is thus unknown.

Methodology

During the project, Scrum was the software development methodology of choice. Two characteristics of Agile Development[8] that were particularly emphasized on were customer collaboration and responding to change. Customer collaboration was needed in order to make sure the types of analysis that were selected for implementation were actually useful for the end-users. This relates directly to the design goal of end-user satisfaction. It was also needed to involve the end-users and those who are ultimately going to maintain the system in the process, making it easier to hand over the system. Responding to change was needed because the project was developed within an unknown domain which leaves any design or set of choices prone to known unknowns and unknown unknowns.

Communication

During the project, we met at the offices of the NS for, on average, three times each week, this was our main means of communication between the team members and other parties. Secondary tools of communication were Skype, electronic mails, phone calls, Whatsapp and GitHub. Documents related to the project were stored on a shared drive and backlogs were maintained on Trello.

Working together at a central location, that is the offices of the NS, has proven to be valuable for communication. It allowed us to align our ideas and goals for the project better, brought us closer to the people that were needed to make sense of the unknown domain and it offered the opportunity for personally interacting with each other, which as a whole makes the project a lot more enjoyable. We discovered that working at a central location three times a week made us work significantly faster, even to the point that it makes the approximate 2.5 hours travel time each day easily worth it. It is worth noting that travel time in the train is not time that is strictly lost, as exchanging ideas or writing some code is still possible to a certain extent.

Communication with the coach and the client was done in a very similar manner as communication within the team. The coach, fortunately, spends time at the NS once a week, which made us able to meet him there for a meeting. Some additional communication regarding scope additions and minor questions were done by email. The client's representative was often present at the NS, this made it possible to be in direct, quick contact with her. Same goes for most NS related staff that are mentioned in this report. This made customer collaboration a lot easier. More formal documents were sent by email. The feedback on those documents was discussed at the office. Small questions or updates about the data were communicated by approaching one another at the office. Small talk and other interesting ideas were shared with during lunch breaks and lunch walks in Utrecht.

[8] http://agilemanifesto.org/

Division of labor

Sprints of, usually, one week were used, for which at the start backlog items were formulated. No formal sprint-reviews were had, due to being in close contact with each other. When formulating the backlog items and adding them to Trello, the scope and initial planning, as formulated in the research report, were frequently taken into consideration. This made for a better sense of control of the scope and schedule. Within the backlog items of each sprint, tickets were labeled with a priority and basic dependencies of tickets were taken into account, such that they would unlikely block each other. When formulating the tickets we tried to subdivide the tasks for a week into corresponding sub-tickets, such that the features that have to be implemented would be sufficiently specific.

After creating the high-level design of the system the responsibility for some key components was assigned to each team member. This was based on which member favored what components. These responsibilities were most significant during the first sprint, however, they defined the general area of expertise each member would come to develop. These areas of expertise had some overlap, such that pair programming could be applied when needed. What these areas of expertise were, is shown in the key contributions in Appendix I: Info sheet. The contributions of each team member to each component can be found in Appendix K: Individual contributions.

In distributing the labor for each sprint, we mostly guessed the amount of work that was needed for a ticket and assigned members with the required knowledge to these tickets, such that everybody would have an equal amount of work. When tickets took longer than expected they were either postponed or the help of another team member was asked. Everybody put in a significant amount of effort and worked equal amounts as a team.

Regarding work-effectivity; in the NS environment, there was no option for Git integration. Therefore, files had to be uploaded manually to GitHub. This way of working was not ideal, as it put a lot of additional work and effort into the project. However, it was the only way of having version control in the environment of the NS. There are plans for the NS to convert the environment, where the new environment does allow Git integration. This was, however, too late for this project.

This approach of distributing knowledge and labor worked well and would not be subject to change had the exact same project to be done again. The key elements for the success of this approach were the partially overlapping areas of expertise and the usage of a Scrum board on Trello with sufficiently specific tickets. Git integration is a factor that could be improved upon, which would have a significant impact.

Schedule

In about week 6 of the project, most requirements were implemented or, less often, discarded. These requirements include the MoSCoW list and initial scope, as described in Appendix M. With three weeks remaining the decision was made to formulate new elements to add to our scope, these are described in Appendix B. The initial planning had some room reserved to explore these additions because the project

functioned in an unknown domain. The general schedule of the project was thus well planned and maintained. Reaching intermediate deadlines for deliverables was always possible without the loss of hair.

The Scrum methodology, with the two aforementioned characteristics emphasized, was fit for the purpose of this project. It directly corresponded with a suitable approach for dealing with an unknown domain while having a focus on end-user satisfaction. Communication, division of labor, and maintaining the schedule were three aspects that went well within the unique setting of the project. As such, the Scrum methodology fit the project well and its implementation and execution also went particularly well.

Ethical Evaluation

The system will have a real-world application and relevance, as such the ethical ramifications should be considered. To estimate whether it would be the right thing to, the degree of achievement of the goals of the NS has to be compared to the potential disadvantages such a system brings. The goal of the maintenance and development department of the NS is to achieve higher punctuality and reliability while balancing the costs. These are not considered unethical by themselves, as they are in the interest of the public good by being traveler oriented or by trying to reduce the government spending needed for the NS. The ethical considerations below primarily consider epistemic topics, confirmation bias and obscuring the source, and a social-economic topic, which is elaborated upon in the section job replacement.

Confirmation Bias

When the end-users, or any group of users, use the system they may try to validate their suspicions by means of the data the system provides. This resembles behavior related to *confirmation bias*, which is described as 'the tendency to search for, interpret, favor, and recall information in a way that confirms one's preexisting beliefs or hypotheses' (Plous, 1993). The system may well be used in a similar fashion; in order to validate the suspicions they previously had, rather than exploring the provided data in order to attain new insights. In fact, during some of the demo's they gave trends credit because they recognized it from their suspicion, rather than recognizing the trend from the data. However, they also gave trends credit because the data lead them to believe so.

If confirmation bias turns out to be a significant effect for this system, it may very well be detrimental for the NS. For example, if users only address the trends they previously believed in, this may lead to some regions in the Netherlands being under-maintained. Similarly, users may also be more inclined to look for trends in the areas that areas that are relevant for them, as they travel through these areas, this may also lead to under-maintained regions. This is typical evidence of our judgments being affected by self-interest (Sezer, Gino & Bazerman, 2015). These under-maintained regions can be subject to more train delays, which would have a negative effect on the general quality of living.

A way to mitigate this effect would be to gain the users' trust with respect to the indicated trends that do not yet have extra support because the users know them beforehand. This means that the indicate trends

really do have to have a ground in reality. As to 'the user only looking for trends that are relevant to them'; all regions are relevant to them, as it is directly related to how well they perform their job.

Job replacement

If the system performs well, it will be easier to find problems and it will be faster to fix them. As an example: currently, whenever there is a problem, someone has to analyze the problem and conclude what the possible impact of the supposed problem can be. This has to be done without clearly presented data that can help in concluding the possible severity. If the program is able to help the end-user to pinpoint problems with more precision than before, it is possible that the investigation process beforehand is shortened. This would mean that there are fewer man-hours needed for tracking problems and, therefore, fewer jobs might be needed. This could lead to possibly fewer jobs being available for the, mostly, lower educated workers.

How technology may influence future employment has been researched by Frey C. B. and Osborne M. A (Frey, C. B., & Osborne, M. A., 2017). They describe a similar phenomenon as discussed in the alinea above. In the conclusions they imply that "as technology races ahead, low-skill workers will reallocate to tasks that are non-susceptible to computerisation – i.e., tasks requiring creative and social intelligence". Such a shift occur within the NS, because of this system and systems alike.

Obscuring the source

While the new system could bring great potential for solving problems faster, it also poses a new problem. Whenever some error pattern is recognized as a potential problem, in many cases someone on the field should go to the location to fix the problem. It could be the case that the engineer could not find any problems, irrelevant if there was a problem or not. In such a case, it is simpler for the people involved to blame the system, due to the automation of the system. This fits with the view of Ramaswamy & Joshi (2009), as "automation simplifies unethical behavior by obscuring its source, e.g. people blaming automation for mistakes, delays, inefficiency, and other weaknesses (*It's not me; it's the dumb computer*)."

Also, if the person was sent, this person is likely from ProRail or any other third party organization. If the program was, in fact, wrong, then the question is also raised of who is liable and accountable for the damage caused. The program and findings were issued by the NS. However, there were no problems found. It would then be the question if the program was wrong or the engineer at the specific location. NS could insist that their system was not wrong, while ProRail could insist that there was no problem and they were thus falsely sent to the location. Would it come to a judicial issue for the accountability, then the focus should be on civil law, "whereas, in criminal law, the accountability for bad AI behavior is typically imposed on individuals who voluntarily commit a wrong prohibited by law, in the field of civil law we should further distinguish between contracts and torts." (Pagallo, 2017). For such cases, it will be hard to find a judicial outcome, as it is still unknown if there was actually no problem or if the problem was simply missed. Presumably, there are already contracts made concerning similar issues. However, these scenarios are not totally preventable and should, therefore, not be ignored. Hence, the users should also keep in mind that falsely identifying could bring unethical consequences.

From the confirmation bias, we found that there is a possibility of unethical acting, whenever a problem is ignored on specific locations in the Netherlands, possibly without the user being aware of it. In the obscurity of the source, it was found that through falsely identifying problems, also unethical issues could be imposed. This creates a scenario, where the user could end up in a decision between three options: ignore the pattern, falsely identify the pattern as a problem, or truthfully identify the pattern as a problem. However, ignoring a pattern is not unethical by definition; it is unethical if certain regions would (unknowingly and) repeatedly be ignored. Both ignoring patterns and falsely identifying patterns as problems are unethical, whenever the program was wrong or the user. If the program is right in the first place this should already be prevented. If patterns are rightfully identified, then the decision might even have an ethical consequence. Solving infrastructural problems could lead to fewer delays, which could on its turn improve the living conditions of the passengers. Hence, the user should not be scared to use the program unethically and should rather try to cohere ethics with the standard workflow.

Handing over the system

As the system that was made is an entirely new system and two of the design goals were *manageability* and *scalability*, it is important to hand over the system to existing engineer within the NS environment. This way the framework the system offers can be used optimally and be extended upon. I. Kalsbeek and N. Oosterhof, from the maintenance and development department, who will take over the system, were informed. They were both well introduced to the system, by means of extensive documentation, personal meetings, and keeping them informed throughout the project. The documentation includes well-commented code and the *RTMA - Systems engineering guide*. I. Kalsbeek, being the coach, was naturally well informed throughout the project. We met with Nick as well on several occasions. The end-users, T. van Haperen and M. Schulte, were also informed about who will be continuing development at the end of this part of the project.

After the two engineers have decided the code is fit for real use, they will hand it over to the DNA department within the NS, which is specialized in data analytics. This department will further maintain and extend the code. They will eventually be responsible for the system to be updated and maintained.

Future approach

During the project, some decisions were made that could have been done differently or features that could have been improved on. Some of these will be outlined below, whereas these future suggestions are feedback on the process and can be used for new projects in the future.

One of the smaller problems we encountered was about the naming of columns in databases. Some of the database tables generated using the analysis system are not always consistent in the naming of columns. These columns could have been named differently if these names were agreed upon on beforehand. At the start, it was agreed upon, which information was required for the UI to be loaded but it was not aligned

which names the columns should get. The different names did not arise problems, as aliases can be used for different namings. However, it would have made life easier.

Also, during the project, different tools were needed for which sometimes permissions were required within the NS environment. An example is the coverage.py module, used for coverage testing within Python. This module was not available within the permissions that were granted to our accounts. After some effort to gain these permissions, eventually, no definite answer was given. In future projects, it would, therefore, be a lesson to start earlier with gathering the tools that are required. During this project, the coverage.py tool could only be executed on our own machines, outside the NS environment.

Furthermore, regarding the division of components in the system, some modules might still have been moved. The analyses components are now still in the main folder, while they should have been moved to a new package for analyses. Due to time constraints, moving these modules was postponed and eventually never done. It could, therefore, still be a recommendation to move the modules regarding analyses to a new package.

Reflection

The BEP project is an interesting and informative process. Different to previous projects, this project gives full responsibility to the students. The usage of version control systems (like Git) were not checked, which gave freedom on one hand but the urge to check yourself on the other hand. The project also allows students to learn from the experience of bigger companies, which is something that should be learned from practice.

Regarding SIG, the feedback was sufficient. However, it would have been better if the code integration tool BetterCodeHub could also have been used for this project. During this project, a new language had to be learned. Therefore, to know the guidelines of SIG, the tool BetterCodeHub would have helped us to follow their quality guidelines. In this case, the guidelines for Java were used, whereas it seemed that the guidelines for Python were (almost) similar.

All in all, we do believe we made the right product for the NS and for the TU Delft. It was a challenging project and we have seen results from the system, that seemed to be valid location-specific problems. Especially due to its accuracy, it was even possible to pinpoint problems to their location within meters. This accuracy will help save time for repairs and in that way save on costs.

Conclusion

The system that was designed is the first manageable, scalable, and high-performance system for running varying types of analyses, data filters, and post processors with a running database connection, which is open for end-user interaction and customization. The choice to implement such a system appears to promise potential for the NS to extend the system in a more complete tool, which can be used by the

varying groups of end-users as a centralized location of knowledge.

The requirements, that were to fulfill the design and promise of the system, were generally well implemented. Even some extra features were added to the scope. The quality of the code that was delivered scored very well, also in terms of maintainability, which is relevant to a system that should be able to be maintained well.

The Scrum methodology was used throughout the project, which influenced the process and project success. Communication between team members and the client was strong, which enabled better alignment of thoughts through the development of the system. The division of labor and schedule of the project was managed well. The team as a whole put in significant effort in bringing the software development cycle successfully to a close. The quality of the process of this project was experienced as a highlight by the team.

With the newly implemented system, it is important that the user uses the program with care. From the ethical evaluation, it was found that ignoring patterns (unintentionally) in the same region could lead to underrepresented areas in the Netherlands. Furthermore, jobs might be replaced with the program, if the search process for problems is reduced compared to current track investigations. At last, faulty decisions should not be obscured by the source; the user might make a wrong decision and blame the program.

At the moment the system is not suited to be used by the end-users directly, because the system should be integrated into one application with UI and analyses in the same environment. With the current NS environment, this was not possible. This problem will be solved in the future, when the environment of the NS has changed and this system is taken over by the NS.

From the outcomes of the system and the reactions of experts on the presumably location-specific problems, it can be concluded that the system is able to find location-specific problems. These problems are based on patterns found using statistics and patterns of code combinations found using an FP-Growth algorithm.

The system that was built is the first manageable, scalable, and high-performance system which will be the stepping stone towards a more effective analytics environment. This analytics environment will in turn help the NS improve its reliability and punctuality, thus successfully addressing the original problem.

Recommendations

Each project has goals that should be reached, while there are also ideas and suggestions that could be met in the future after the project has finished. The future ideas and suggestions, for this project, are outlined in this section. Most of these ideas were not implemented, as there were other features with a higher priority. In this section are only feature suggestions that could be of direct importance for the NS or this project. In Appendix B, more feature suggestions are given, which are more for brainstorming purposes.

General direction

In broad terms, three things need to happen in order to make the system directly usable by the end-users. For one, the data currently used in the analysis is not perfect. When the integrity and the amount of available data are improved, end-users will be able to discover more meaningful patterns. Secondly, support for multiple train types ought to be added. Currently, only the VIRM series is supported, which limits the completeness of the overview the system is supposed to provide to the end-users. Finally, the system as it is should be more integrable with the current tools the end-users use, which it is currently not. No concrete recommendation is written for this, as the specific knowledge needed, for integrating the analytics system with the other tools the end-users use, is not present within the project team. Nor should this knowledge be present. The final point can be addressed throughout the process, as sketched out in the 'Handing over the product' section.

Uniqueness analysis_id

Every distinct run in the system gets a new analysis_id. The problem with the current assignment of this analysis_id is that it is not always guaranteed to be unique. At the end of the project, it was found that there is a time-interval of around six seconds in which the uniqueness of an analysis_id cannot be guaranteed. This should be prevented, as these runs would corrupt each other. Afterward, we should have made this analysis_id user dependent (e.g. analysis_id “user-1” instead of “1”). Another option could have been to make the database table, in which the different runs are stored and checked for their uniqueness, transactional to ensure each transaction to be unique.

Email reporting

In the current system, there is a small class that was added with the intention to add an emailing system to the application. This system could possibly be used to notify the user of their analysis being finished. This could become handy whenever a bigger analysis was started which could run for more than an hour. Also, the emailing system could be used to summarize the findings of the program, which could make it easier to share results with colleagues.
The class that was introduced for this mailing system is currently only the framework for it. It does not have any mail protocols built-in. Moreover, to enable messaging over the servers of the NS environment, permissions must be granted to the application.

Availability of Data

Currently, the application bases its assumptions for patterns merely on errors sent by trains. In the future, it might be desirable to also include sensor data with the error, to learn more about conditions of sensors in combination with the errors and locations. It could be the case, where certain sensors seem to be more sensitive on specific track sections but also the influence of the weather on the sensors could be integrated with the analyses. This could enable the application to provide assumptions dependent on the weather. An example could be: “on sunny days, track section X has more problems with sensor Y, which

leads to error code Z".
Also, a correction with the weather could be applied, whereas some sensors might be overreacting when, for example, the temperature outside is higher than a certain threshold. These measurements by the seconds could then be corrected with the temperature of the specific timestamp.

Database problems

Many database tables seem to be incomplete or corrupted to some extent. The table containing diagnose data, for example, has duplicate records. However, this is not the main problem; there are records with an equal record_id (the identifier) but an unequal row. This means that there are two, probably, distinct records with the same record_id. Another possibility could be that two of the same records somehow got different values due to, for example, lost details during the transaction.

Some of the database tables were introduced with all columns with the same type. Each of the columns in these tables is a String type. The problem with this solution is that any analysis done on these databases have to either introduce some computation overhead to convert the data types, or the tables are copied and converted to their respective types. Both solutions have a major drawback, the former introduces extra computational cost, while the latter introduces memory waste. If the database tables would have been converted from the start, there would only have been a single computational overhead for the initial conversion and no further computational costs for any analysis done afterward.

Many tables in the databases are not partitioned. This means that there are possibilities for improvement in speed and efficiency of use in resources. For example, the table used in our application default.trends_diagnose_enriched could possibly be partitioned on columns track_sections or date_time (best performances are expected for the column track_sections for our application). Also, the tables generated by our application are currently not partitioned and may be partitioned on the analysis_id, which is the unique identifier for different runs with the application. Better partitioning could help the application run significantly faster.

Finally, the join needed to reduce the loading times within Power BI, as described in the section 'Database Analyses Normalization', should be moved to a PySpark operation as a part of the standard analyses flow.

Multiple train types

Currently, the system is only designed for the VIRM train type but should still be easy to extend to different train types. Different trains also have different behavior, so if the system should run with more trains, it might be desirable to have a post-process that focuses on finding, for example, statistics about the presence of train types within error patterns. For example, it could be that a certain pattern is mostly concerning the SLT train type, while the VIRM is not affected by it.
Whenever the system is expanded with more train types, the programmer should also be careful with the meaning of error codes. It could be the case that a certain train type gives the same error but for different sensor thresholds. While the ERTMS (the European Rail Traffic Management System) is also not fully

implemented in all trains, there might even be a possibility for trains to give a similar error code with a different definition.

Analyze specified intervals in consecutive years

It would be useful to be able to analyze specified intervals in consecutive years. Seasonal error patterns can get clear that way. It can be for example that the infrastructure at some places causes more errors in the winter when it is cold or more errors in the fall when leaves start covering the tracks. With this knowledge, you can track down the location where these season errors happen and try to prevent them next season.

References

1. Bannerman, P. L. (2008). Defining project success: a multilevel framework. In: P*MI Research Conference 2008.* Retrieved from https://www.pmi.org/learning/library/defining-project-success-multilevel-framework-7096
2. Dietterich, T. G. (2000) Ensemble Methods in Machine Learning. In: *Multiple Classifier Systems,* 1-15. doi: 10.1007/3-540-45014-9_1
3. Frey, C. B., & Osborne, M. A. (2017). The future of employment: How susceptible are jobs to computerisation? Technological Forecasting and Social Change, 114, 254-280. doi:10.1016/j.techfore.2016.08.019
4. Han, J., Pei, J. & Yin, Y. (2000). Mining Frequent Patterns without Candidate Generation. In: *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, 1-12. doi: 10.1145/335191.335372
5. Hansen, L. & Salamon, P. (1990). Neural network ensembles. In: *IEEE Trains. Pattern Analysis and Machine Intell., 12*(10), 993-1001. doi: 10.1109/34.58871
6. Kim, Y., Huang, J., & Emery, S. (2016). Garbage in, Garbage Out: Data Collection, Quality Assessment and Reporting Standards for Social Media Data Use in Health Research, Infodemiology and Digital Disease Detection. *Journal of Medical Internet Research, 18*(2). doi:10.2196/jmir.473
7. Leskovec, J., Rajaraman, A. & Ullman J. D. (2011). In: *Mining of Massive Datasets.* Retrieved from http://infolab.stanford.edu/~ullman/mmds/book.pdf
8. Mabroukeh, N. R., Ezeife, C. I. (2010). A taxonomy of sequential pattern mining algorithms. *ACM Computing Surveys, 43*(1), 1-41. doi:10.1145/1824795.1824798
9. O'hurley, G., Sjöstedt, E., Rahman, A., Li, B., Kampf, C., Pontén, F., . . . Lindskog, C. (2014). Garbage in, garbage out: A critical evaluation of strategies used for validation of immunohistochemical biomarkers. *Molecular Oncology, 8*(4), 783-798. doi:10.1016/j.molonc.2014.03.008
10. Pagallo, U. (2017). From Automation to Autonomous Systems: A Legal Phenomenology with Problems of Accountability. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence*. 17-23. doi: 10.24963/ijcai.2017/3
11. Pinto, J. K. & Slevin, D. P. (1988). Project success: definitions and measurement techniques. In: *Project Management Journal, 19*(1), 67–72. Retrieved from https://www.pmi.org/learning/library/project-success-definitions-measurement-techniques-5460
12. Plous, S. (1993). McGraw-Hill series in social psychology. In: *The psychology of judgment and decision making.* New York, NY, England: Mcgraw-Hill Book Company
13. Ramaswamy, S., Joshi, H. (2009). Automations and Ethics. In: *Springer Handbook of Automation*. 809-833. doi: 10.1007/978-3-540-78831-7_47
14. Sathyamoorthy, D., Shafii, S., Amin, Z. F. M., Jusoh, A. & Ali, S. Z. (2015). Evaluation of the accuracy of global positioning system (GPS) speed measurement via GPS simulation. In: *Defence S and T Technical Bulletin, 8*(2), 121-128. doi: 10.3141/1818-03
15. Sezer, O., Gino, F. & Bazerman, M. H. (2015). Ethical blind spots: explaining unintentional unethical behavior. In: *Current Opinion in Psychology, 6*, 77-81. doi: 10.1016/j.copsyc.2015.03.030

Appendices

Appendix A: Glossary

Diagnostic error code: A diagnostic error code is a code, e.g. ATB105, that is thrown when sensors detect deviant behavior in the train system(s) they belong to.

Maintenance facility: A kind of maintenance location for trains, where trains go for longer types of maintenance. There are four maintenance locations in the Netherlands.

OCCR: Operationeel Controle Centrum Rail (English: Operational Control Centrum Rail) is a control centrum, functioning at an operational level, for preventing problems and addressing them if they occur.

RTM: Acronym for 'Real Time Monitoring', trains with RTM have a live connection to the database and thus provide the most up-to-date data.

RTMA: Real-time management analytics system for the application of data analytics in the NS.

RTMO: Real-time management operation system used by the OCCR.

Service station: A kind of maintenance location for trains, where trains go for shorter, often less severe, types of maintenance. These locations are common.

Timetable point (Dutch: dienstregelpunt)**:** A timetable point is a primary area that forms a continuous limited part of the railway network. The timetable point fulfills the function of setting up and recording the timetable. Examples of timetable points are stations and bridges.

Track section (Dutch: baanvak)**:** The area between two timetable points on a railroad track. Different track sections may have different lengths.

VIRM: Acronym for 'Verlengd InterRegio Materieel', a train series that is relatively new which supports an RTM connection.

Appendix B: Scope change

In about week 6 of the project most of the elements described in the *MoSCoW* method and scope were implemented or, less often, discarded. With three weeks remaining the decision was made to formulate new elements to add to our scope. The scope additions were chosen such that they show the adaptability of the system and allow easier validation of indicated trends by means of indicating the delay in a given trend. The latter was chosen as a complementary feature to the currently existing types of analysis; seeing whether the facts that are pointed out are significant is important, in terms of the proof of concept and the *quick insight into data* aspect. Adaptability would be shown by showing the ease with which new features can be added and the ease with which different sources of data can be integrated.

Scope Additions

Implemented

These scope additions are actually implemented.

Display delay next to the trends

The delay paired with a trend is one measure of the severity of a trend. The delay meant here is the amount of time (minutes) a train arrived later at a given location. The end-users can use this in order to validate the severity of trends better. This addition was chosen, because, as explained above, it helps validating the trends found by the implemented analyses.

Remove trends

The trends which are indicated as significant by the system should be able to be removed from the analysis by the user. Trends may be insignificant, because the trends have either been addressed or because they are not of any significance in terms of travel-delay or other measures of significance. This feature should help the users stay focused on the information that is significant.

Post processing by Collaborative Filtering

The Collaborative Filtering algorithm is used for finding similarities between the trends found on track sections and the algorithm tries to guess for some trends what their frequency might become in the future for a track section.

Not implemented

Given we could spend a limited amount of time on this project and it created an entirely new system, there were a number of ideas that were thought of but were never implemented. These ideas will be described here, such that they can be picked up and implemented later.

Interface for adding any type of time-series data

Since the diagnostic error codes indicating deviant behavior are thrown over time they inherently have a time component to them, as such these diagnostic codes can be enriched with any other time-series data. For example, the thrown codes can be enriched with the weather data. This could allow the user to see whether certain codes are thrown more often during colder weather. Traveler satisfaction per day is another example.

Highlights for track sections

In order to gain easier insight as to what stands out for a certain track section, significant observations can be added to track sections as a kind of highlight. Such a highlight may be that the track-section participated in a strong FP-Growth pattern or that the weather was often below freezing point when a certain diagnostic code was thrown.

Currently, the post-processor of the system summarizes the results of all analyses. In this post-processor, the highlights may be determined by looking over the meta-characteristics. This looking over the meta-characteristic could be done by specifying a new type of post-processor.

N-gram analysis

A type of analysis that could be added to the analyses would be the N-Gram analysis. This type of analysis would allow the users to gain insight into contiguous patterns that occur. For example, if ATB105-REM012 often occur after each other, then this pattern would be picked up by this type of analysis.

Regarding implementation, this could be another analysis class which is another child of the Analysis class. For tokenizing the sequences of diagnostic error codes into n-gram tuples PySpark's tokenizer can be used. PySpark provides an out-of-the-box n-gram tokenizer in the standard MLib library[9]. After the sequences have been tokenized per track section, one could check, with a frequency algorithm of preference, which n-gram tokens frequently occur for a track-section. After writing a selection of n-gram tokens to the database this analysis can be added to Power BI interface, which would provide additional insight for the end-users. Considering the possible memory constraints, it is recommended to tokenize the sequences of diagnostic error codes per track section and storing the result after every track section, similar to the FP-Growth algorithm.

Neural networks

A deep neural network could be used on top of certain analysis types to learn correlations between error codes and deviant behavior. Deviant behavior surrounding trains could, for example, be traveler complaints or the number of minutes a train is delayed. These correlations between error codes and deviant behavior can be used to indicate the relevance of patterns with respect to a measure of deviant behavior. For example, using the 2-gram model, if during train rides the tuple ATB105-REM012 have a high correlation

[9] https://spark.apache.org/docs/2.2.0/ml-features.html#n-gram

with the number of minutes a train is delayed, then the neural network may indicate this. As such, patterns could have a certain degree of autonomous validation with respect to the relevance of an indicated pattern.

A specific use-case for such a neural network could be to provide a multi-layer perceptron (MLP) with enriched n-gram tuples, in order to predict how relevant the tuple and its features were for different measures of deviant behavior. Figure 8 below visualizes what the design of such a network may look like. For the input n-gram tuples enriched with the activation time and severity class feature are used. These features are currently present in the diagnose table. This may look like: [(ATB105, B, 2018-05-03 18:30:15), (TRP098, B, 2018-05-03 18:32:15), (REM098, B, 2018-05-03 18:32:25)], i.e. a list of tuples with (diagnostic report code, severity, activation time). Any amount of features could be chosen to enrich the tuples. For the output various measures of deviant behavior can be used as the predicting target. For example, with the target being the number of minutes of delay that was caused on a track section the neural network would learn to associate the input tuples with the delay that occurred.

Additionally, on top of these measures another classifier could be trained in order to indicate a (meta) measure of the significance of a pattern, which could be presented to the end-user. This classifier would have to be trained by end-users, as they can determine whether certain measures of deviant behavior are significant or not when combined. This would mean the classifier is not autonomously trained. Regarding a possible implementation, Pyspark provides various classification and regression utilities[10] which can be used in order to create the neural network.

[10] https://spark.apache.org/docs/2.2.0/ml-classification-regression.html#classification

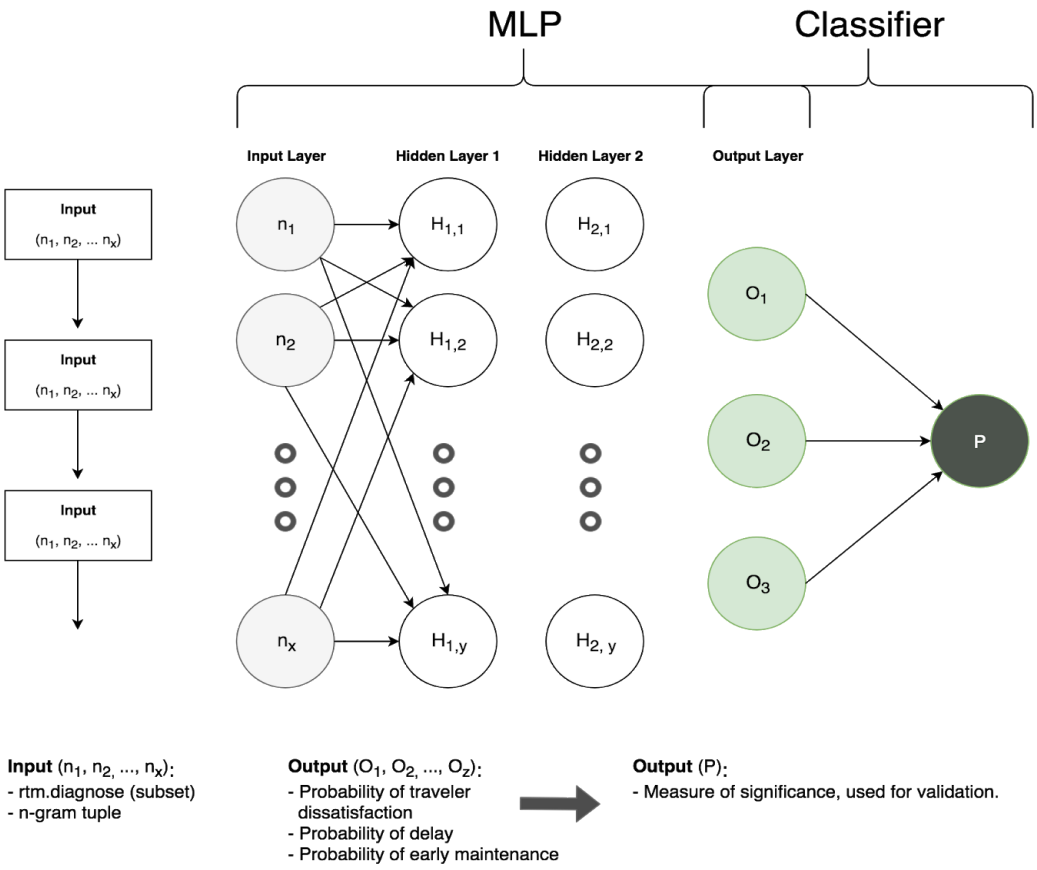

Figure 8: Neural network architecture

Errors in specified intervals for consecutive years

Trends may be specific for a certain moment of the year, consecutive winters or holidays may deliver interesting results when compared. It is known that the weather could have influence on the performances of trains. During hot days, different errors might be sent than snowy or rainy days. Being able to verify and identify these problems is wanted, as it may be possible to identify the problems and take precautionary actions.

Discarded/Changed from Scope

The user shall be able to receive an automatically scheduled analysis report of a user-specified time interval.

Changed: Users can schedule analyses, but they will not receive reports based on it. The reports have been omitted, as they do not add enough value to the product. Especially considering similar reports can be obtained by opening the scheduled analysis in Power BI.

Receive an automatically scheduled analysis report of the last 24 hours that will be generated daily
Similar motivation as the item above.

Export to CSV
Power BI supports this feature.

Display an error indicating no trends were found:
Displaying errors indicating whether no trends were found, no connection with the database was possible or the features in the data are changed were partially done by Power BI and when it is not it would not be of any use to display to the user.

Appendix C: Original project description

This is a copy of the original project description as provided by the client through BepSys.

Problem definition

One of the important factors for customers satisfaction of the Dutch Railways (NS) is the reliability and punctuality of trains. Malfunctioning of the train affects train punctuality and can be due to failure of the train itself or due to infrastructure that affects train hardware. For instance, a tilted track can lead to non-closing doors at the station under some conditions, which means that the train cannot depart. Another example is a disrupted ATB signal, which leads to immediate braking of the train. NS intensively uses the tracks, some trains travel around 1500 km a day, in total NS has approximately 650 trains (around 3000 coaches). Therefore, NS is often first to notice anomalies or inconsistencies in the tracks.

For NS it is very important to have insights in these kind of location-specific incidents. When something is detected, actions can be taken to prevent future incidents. This will improve train punctuality, but also will reduce unnecessary work for mechanics, since some faults are actually location specific instead of train specific; if a mechanic is unaware of this interaction with the infrastructure, fault finding in the maintenance shop may be impossible.

What we have - Current situation

Nowadays, location-specific incidents are only noticed by aware employees, who are noticing reoccurring events. However, data of hardware incidents together with their location and time is continuously available in a database. This data can be used to support assumptions and to pinpoint incidents in real time. Also, incident calls by train drivers together with the measured delays and their time and location are available. However, since trains generate a lot of data, smart visualisation and analyses are necessary to notice these kinds of anomalies.

What we need - The project

Concrete challenges for the software product:

- Dashboard: Visualize collected data to detect (new) patterns, e.g. based on train type or time window
- Automatic detection of “patterns”, including a warning system
- Adjust patterns or add new patterns to the automatic detection system
- Real-time connection with database
- Monitor the monitor: warning system if data collection fails

The dashboard should support the following types of users:

- the reliability engineers who need to take action on incidents 24 hours/day

- analysts who aim to identify root causes of incidents and support the reliability engineers

What we offer

- Freedom and responsibility from day one
- A monthly internship compensation of approximately 500,- per person (25,- per working day)
- Our Utrecht office, for approximately 3 days a week, which is inside Utrecht Central station

Other relevant information

- Some group members should speak Dutch
- TU Coach: Mathijs de Weerdt
- Client: Inge Kalsbeek

Nederlandse Spoorwegen (NS)

Every day, NS transports more than 1.1 million train passengers. We are their connection between home and work, between family and friends and all of the other people in their lives. The train is still the only mode of transport that can take passengers to the heart of the city centre unhindered, in a sustainable and safe manner. The train has been a unique way to get from place to place for more than 175 years, and that is a source of pride for more than 30,000 NS employees. On average, NS has performed much better over the past few years, but not on every route and not for every passenger. So to us, 'average' is not good enough. Our ambition for the next few years is to turn our best performance from the recent past into the new standard. In order to achieve that goal, we must improve our performance on the main rail network, including HSL and the international train service that runs on that network.

Appendix D: UML Class diagram

Below the UML is given as separate components in the complete framework. For a full UML diagram, see the external file *UML.html*.

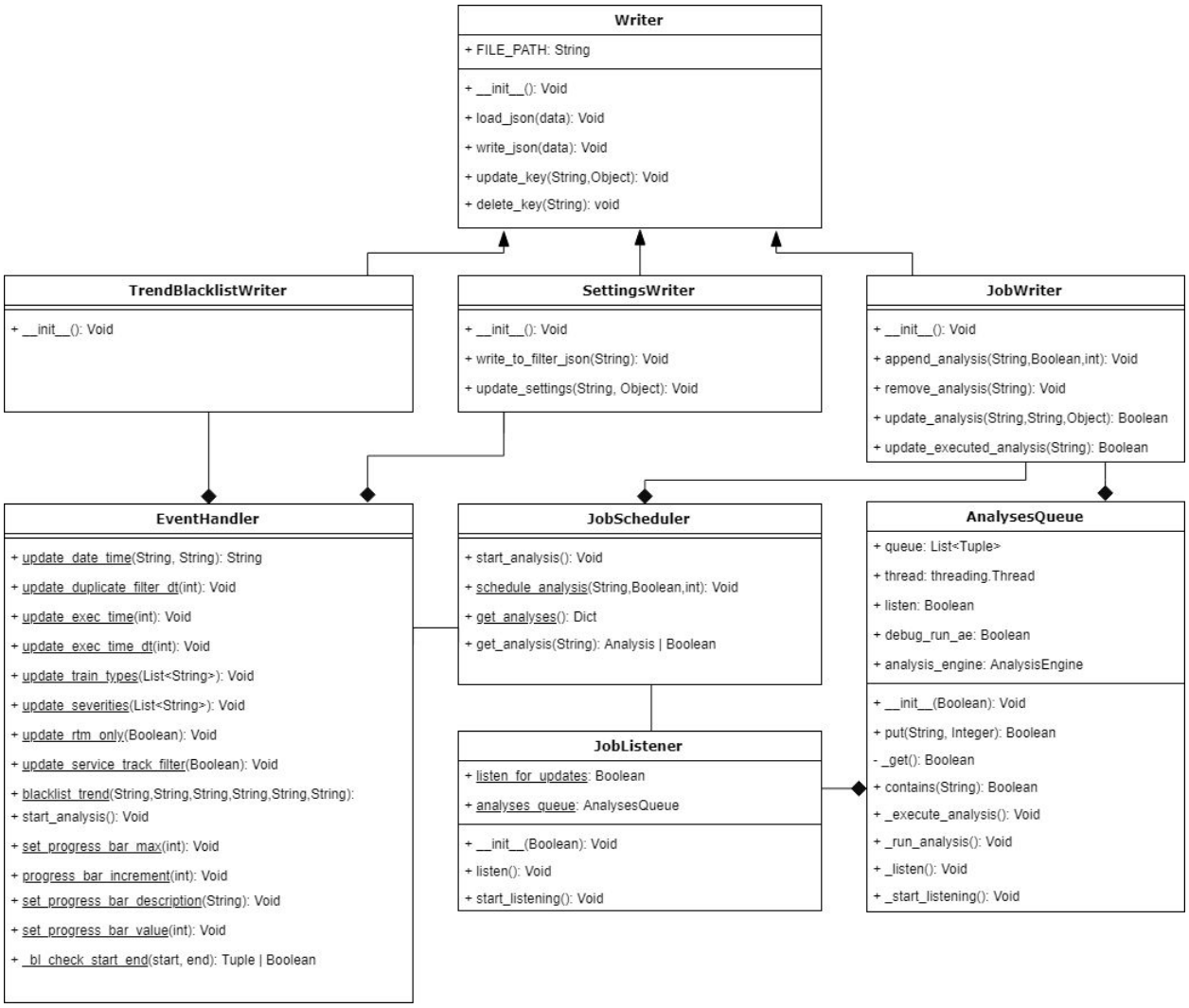

Figure 9: UML class diagram of the EventHandler and Writers

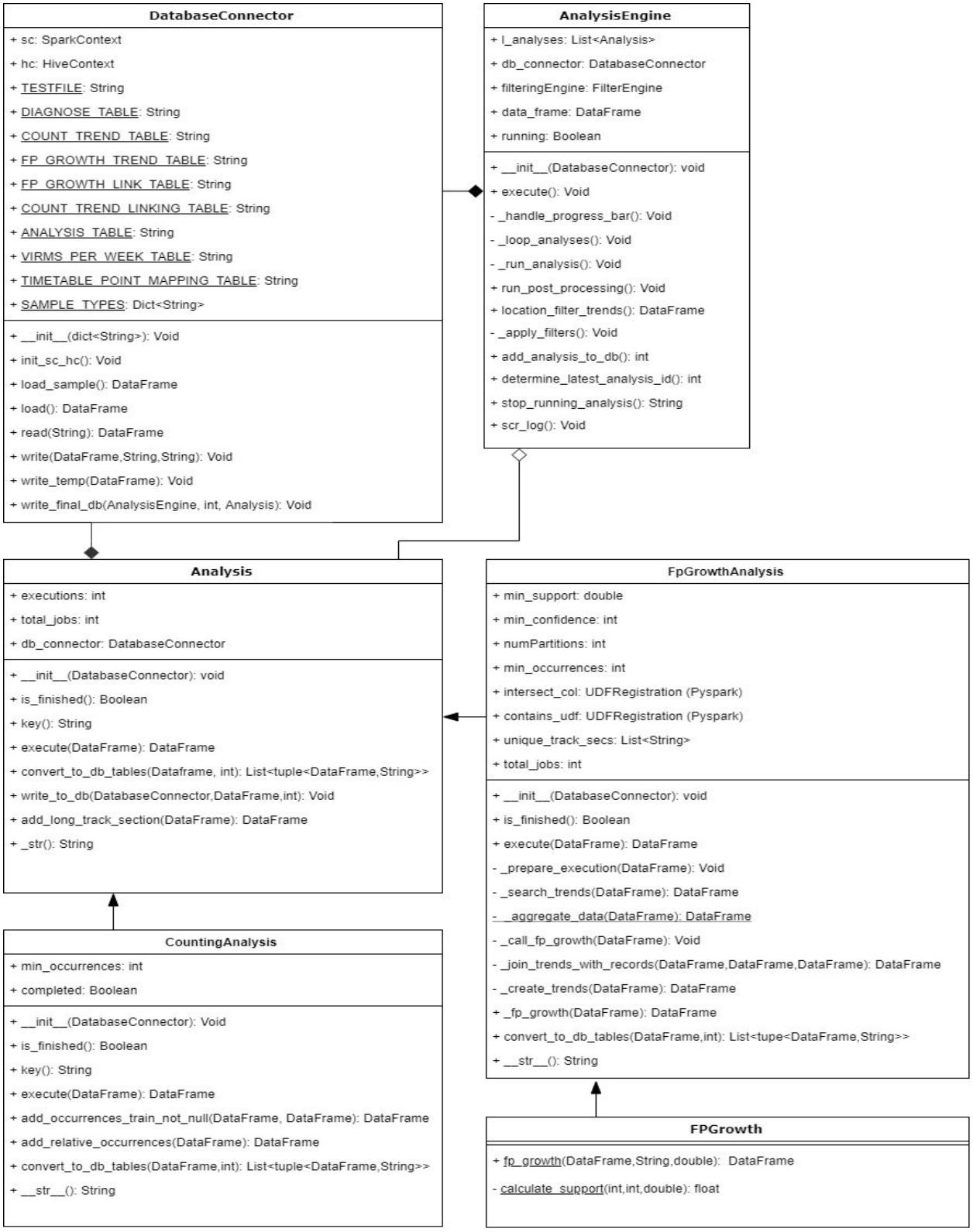

Figure 10: UML class diagram of the Analyses

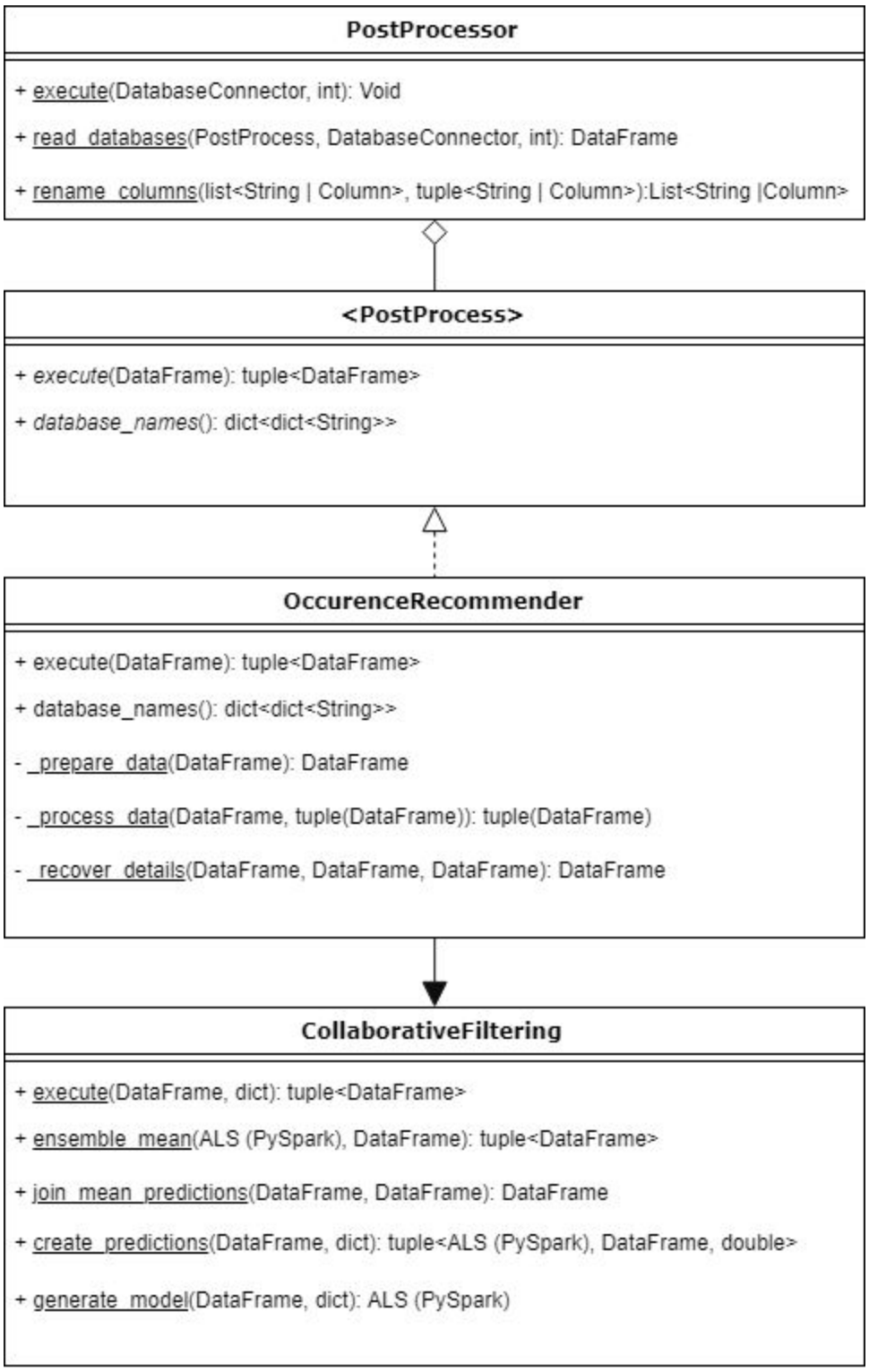

Figure 11: UML class diagram of the Post Processor

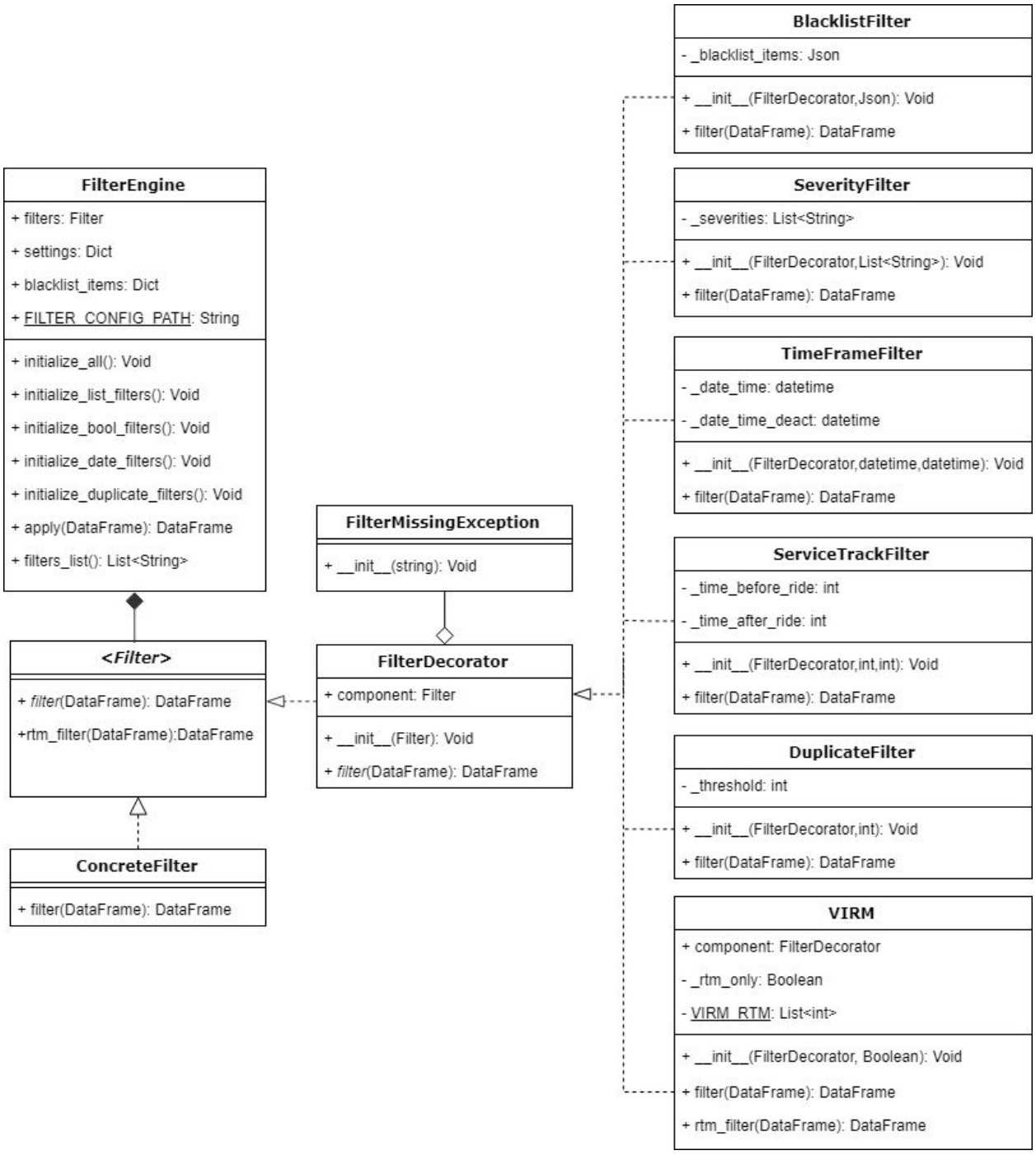

Figure 12: UML class diagram of the Filters

Appendix E: Jupyter UI application flow diagram

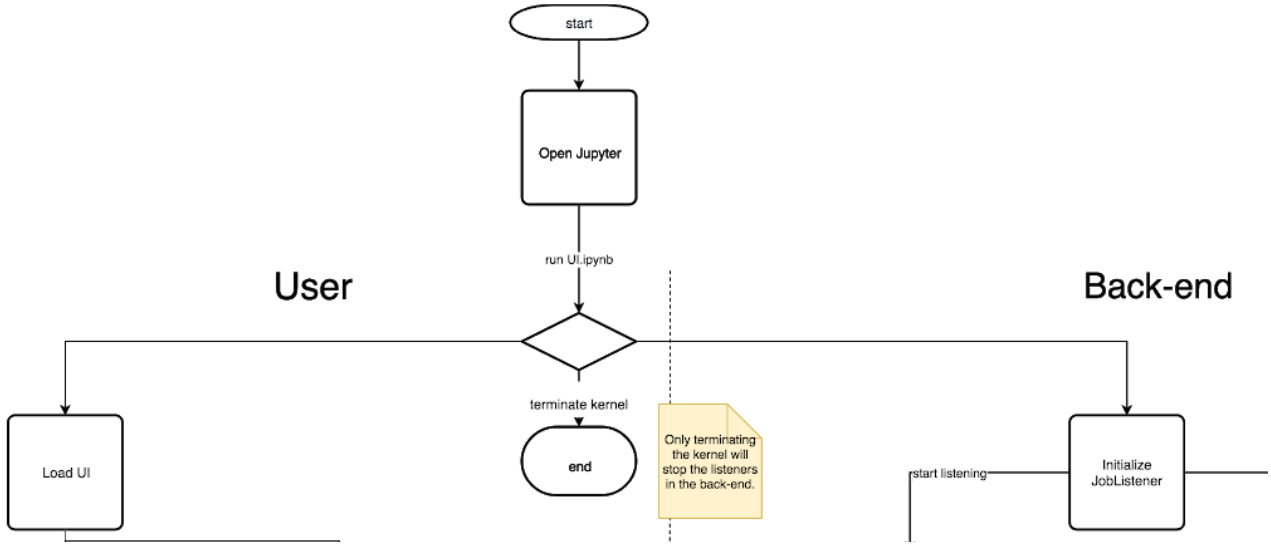

Figure 13: Application flow upon launching the Jupyter UI.

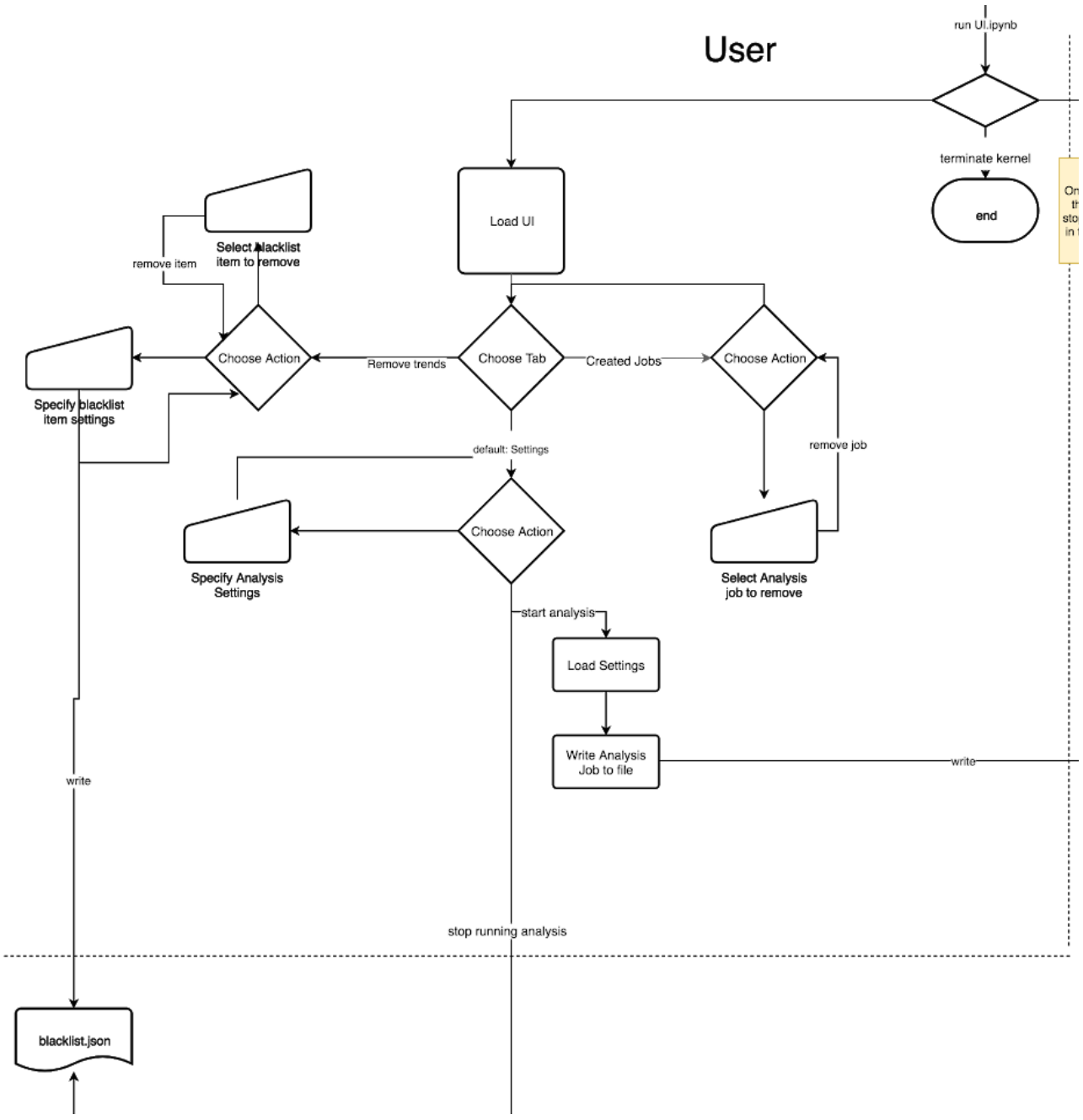

Figure 14: Front-end application flow and user-interaction

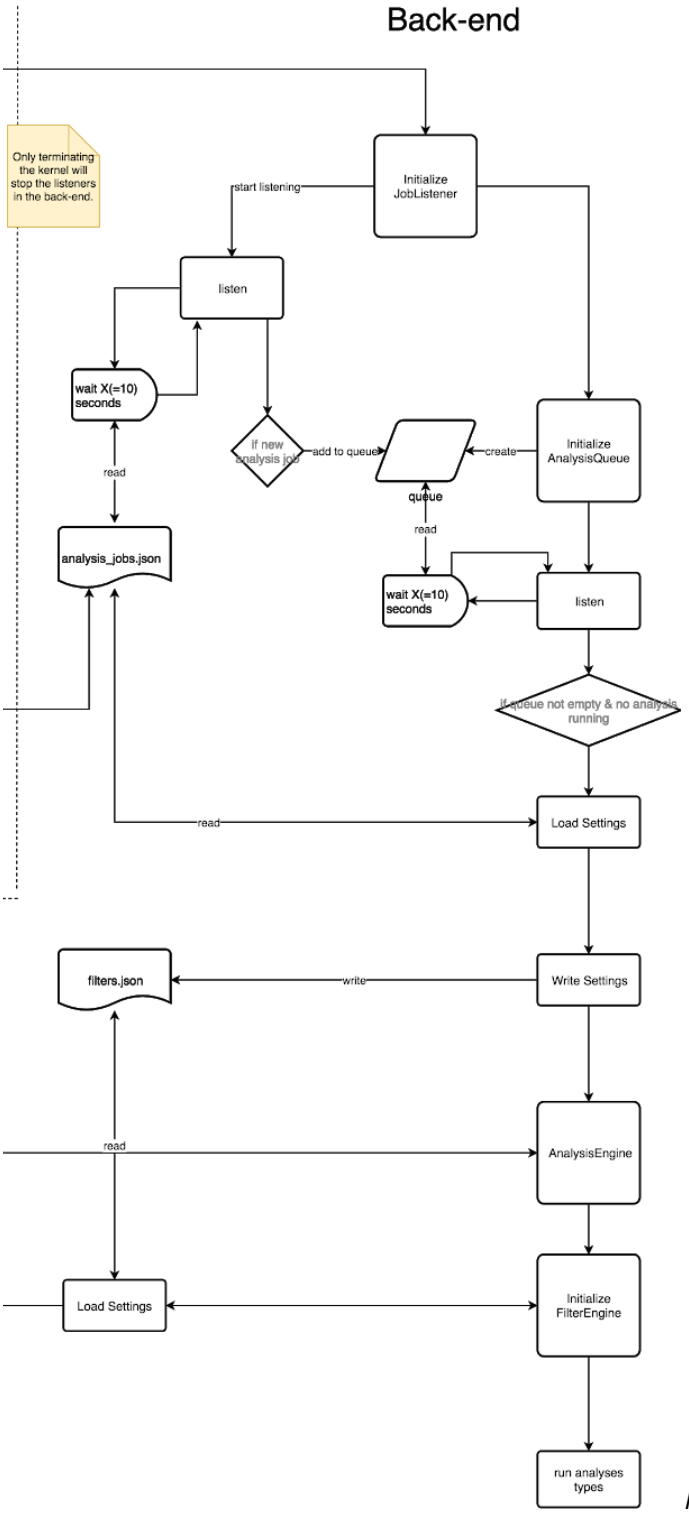

Figure 15: Back-end application flow

Appendix F: Jupyter user interface

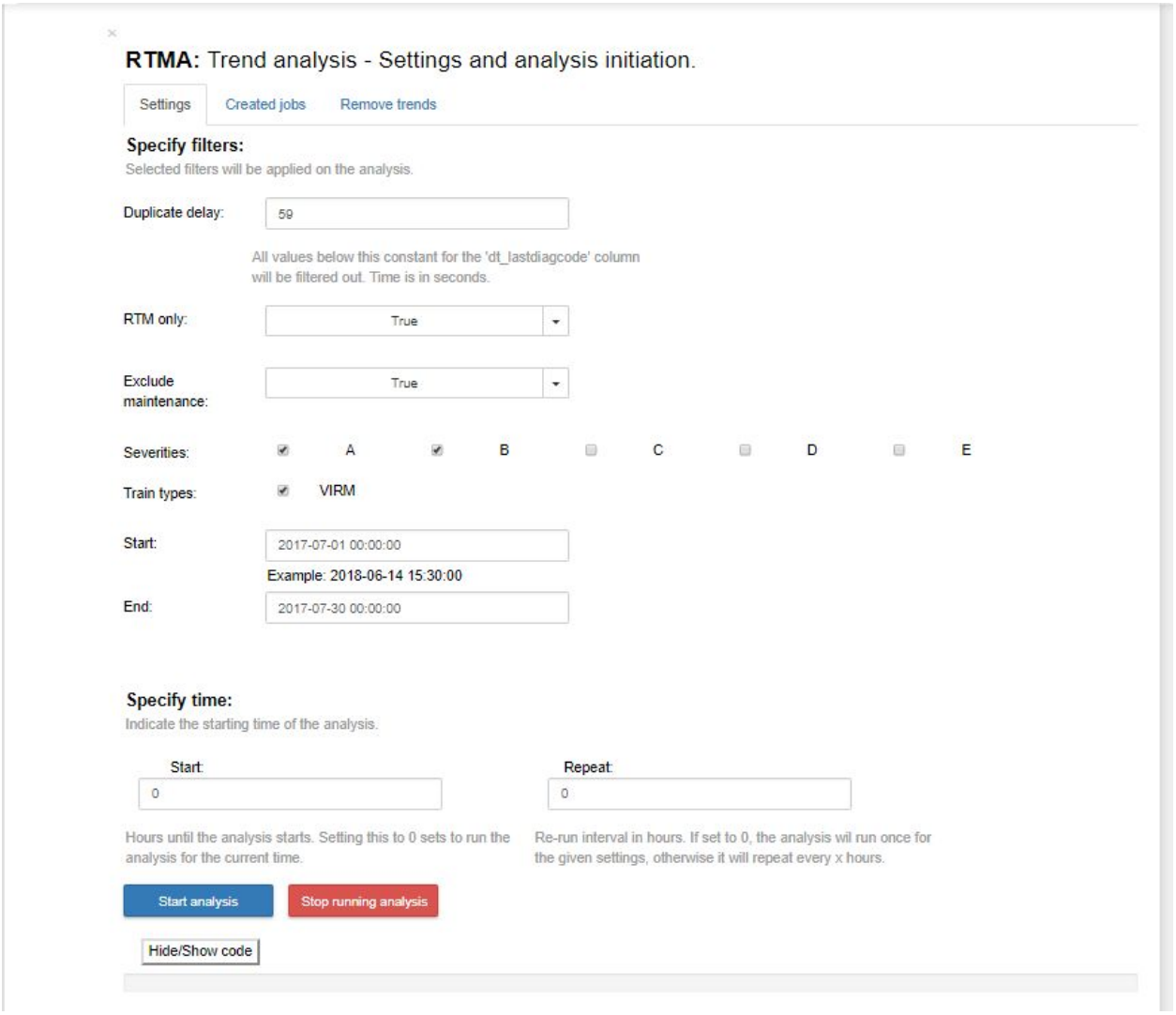

Figure 16: Jupyter UI starting screen.

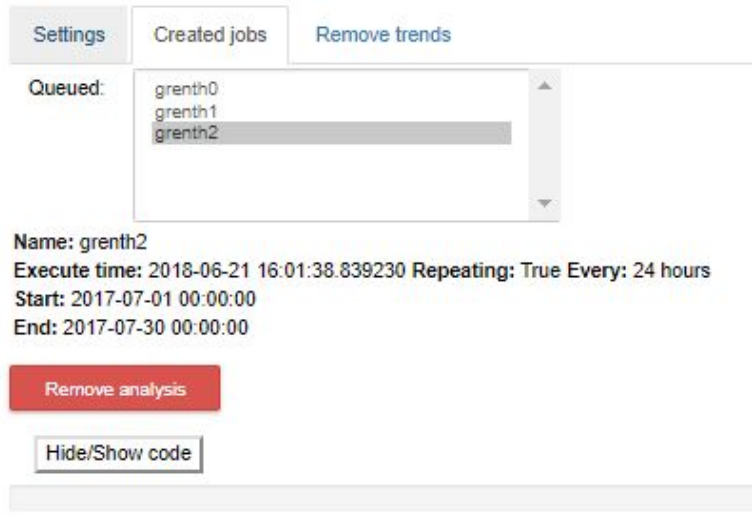

Figure 17: Jupyter UI 'Created jobs' tab.

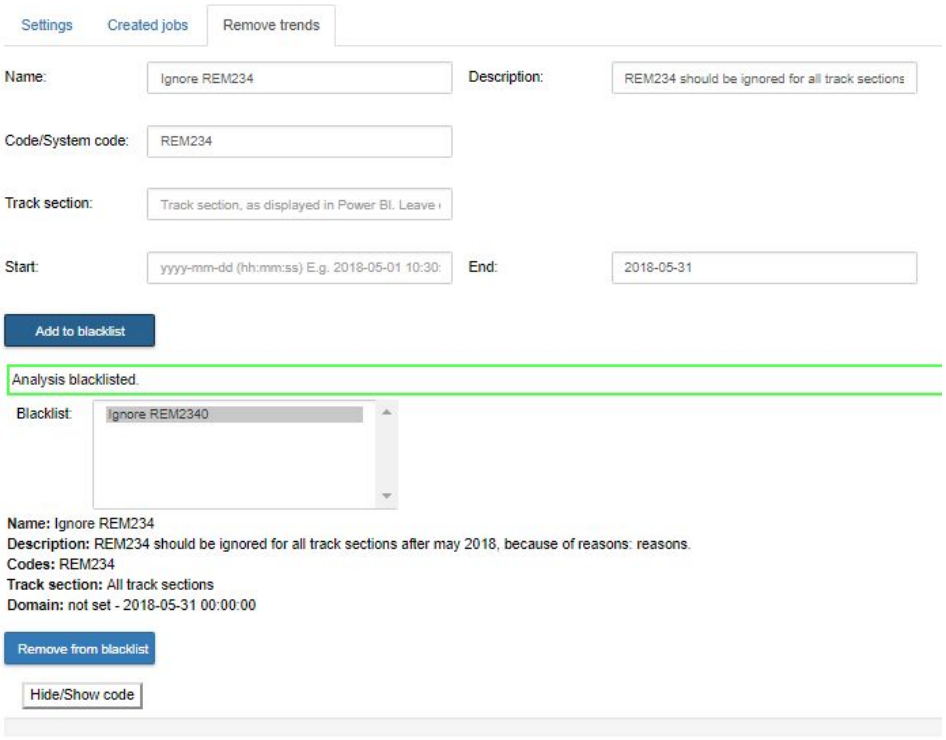

Figure 18: Jupyter UI 'Remove trends' tab.

Appendix G: Power BI user interface

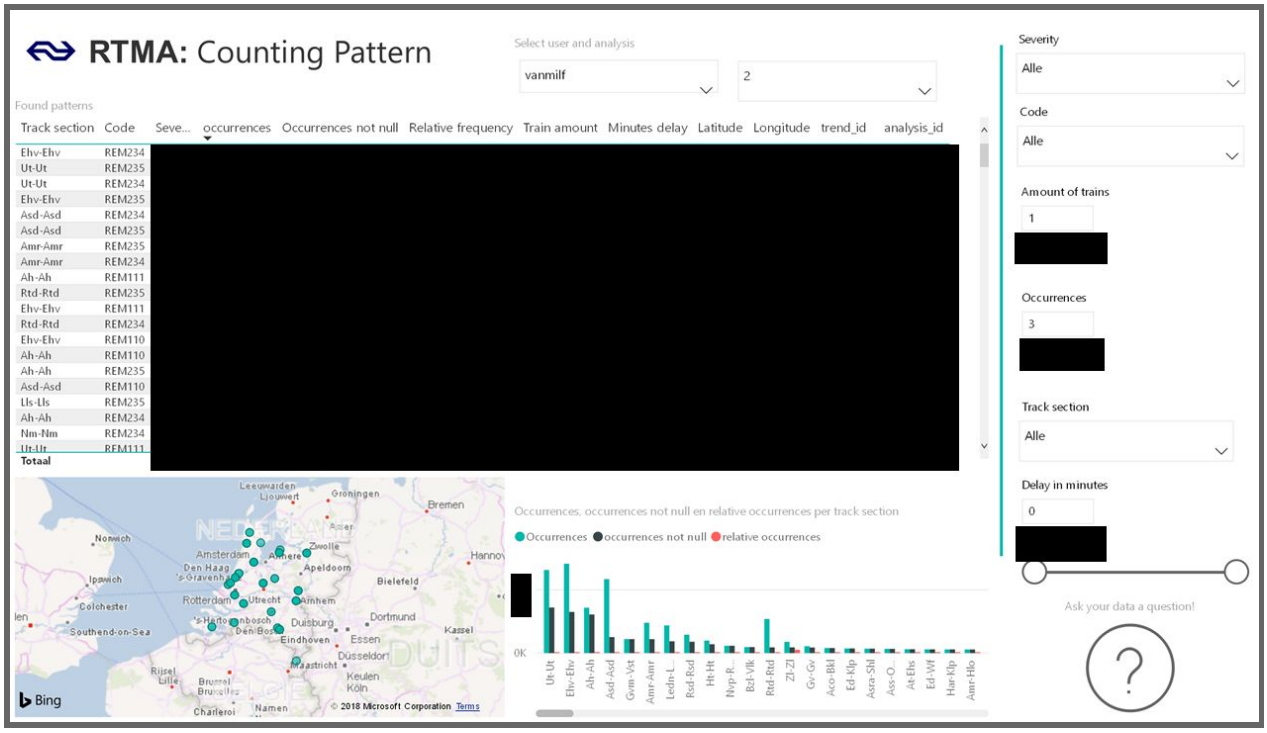

Figure 19: Top-level Counting Analysis visualization

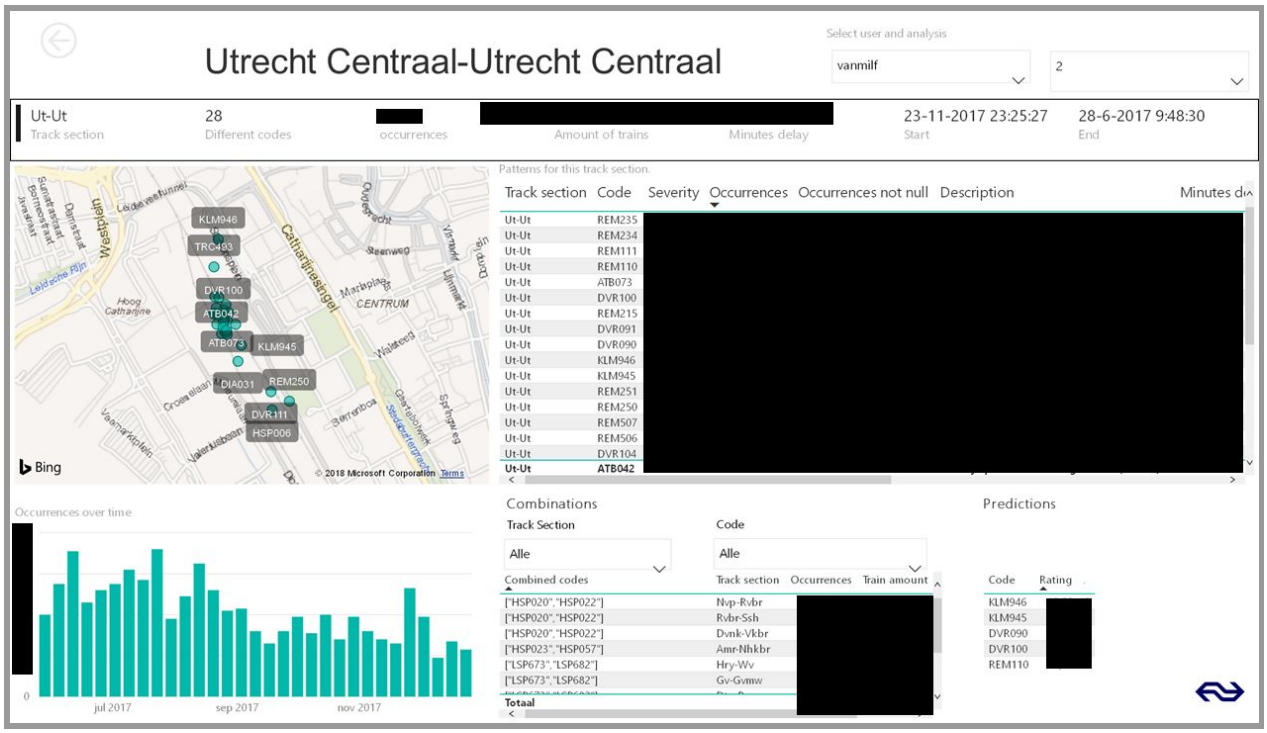

Figure 20: Detailed Track Section point of view

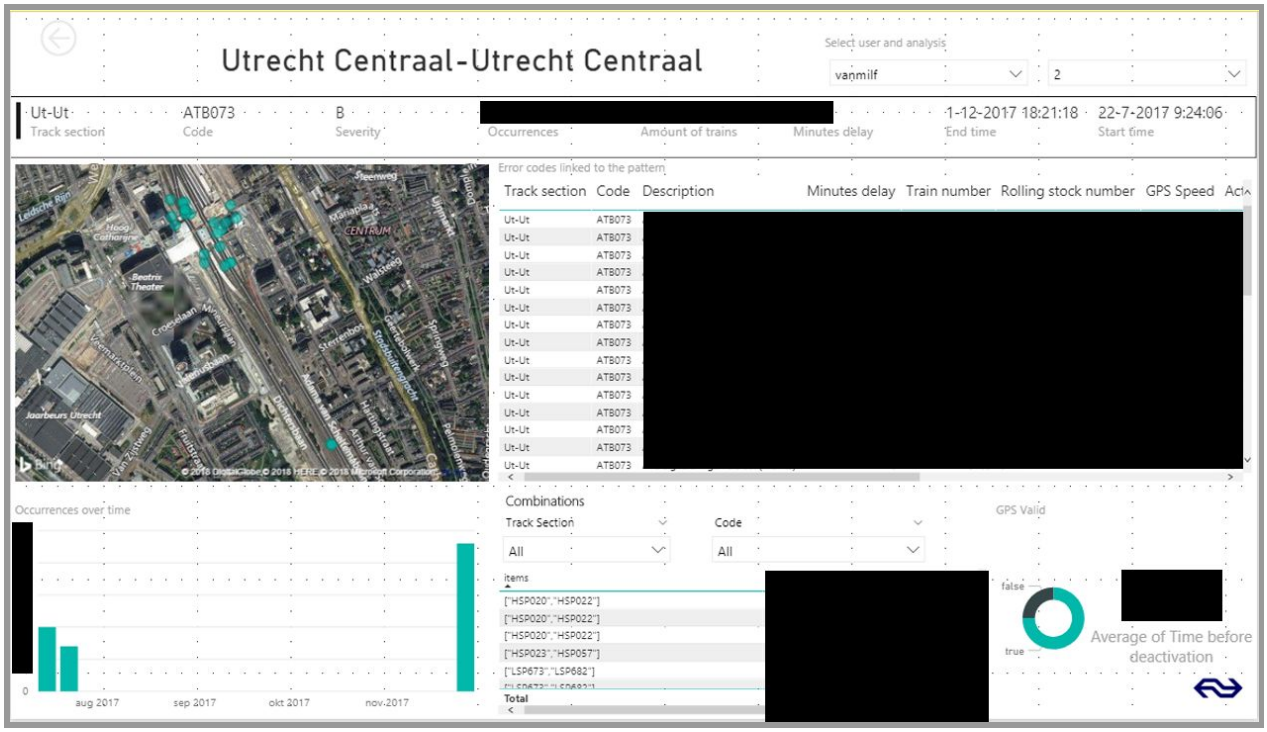

Figure 21: Detailed Error Code point of view

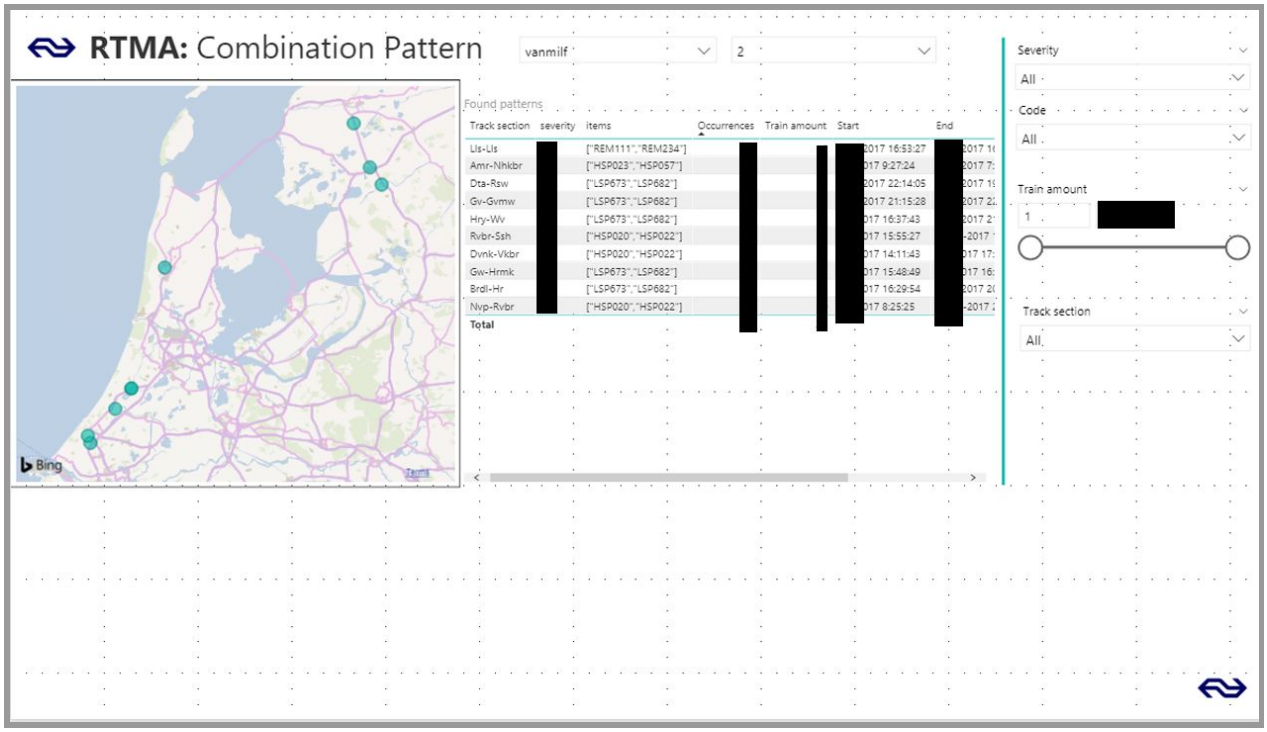

Figure 22: Top-level FP-Growth Analysis visualization

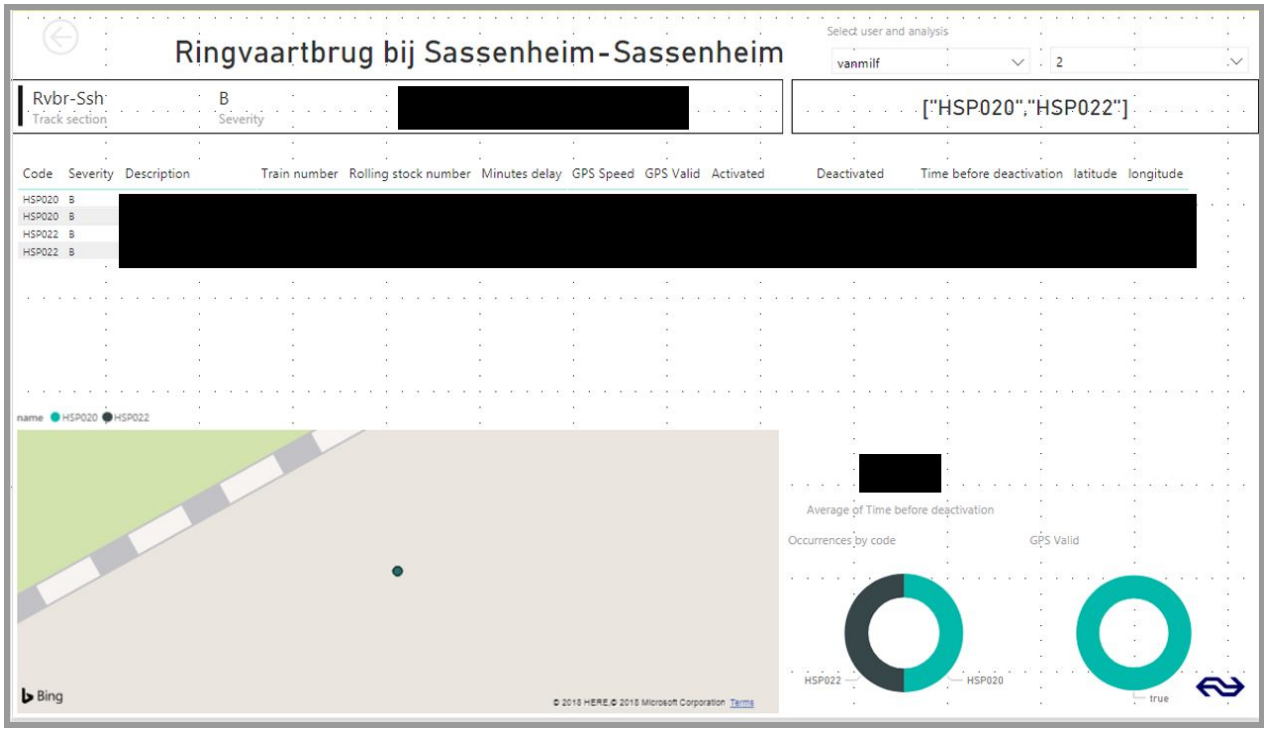

Figure 23: Detailed Code Combinations point of view

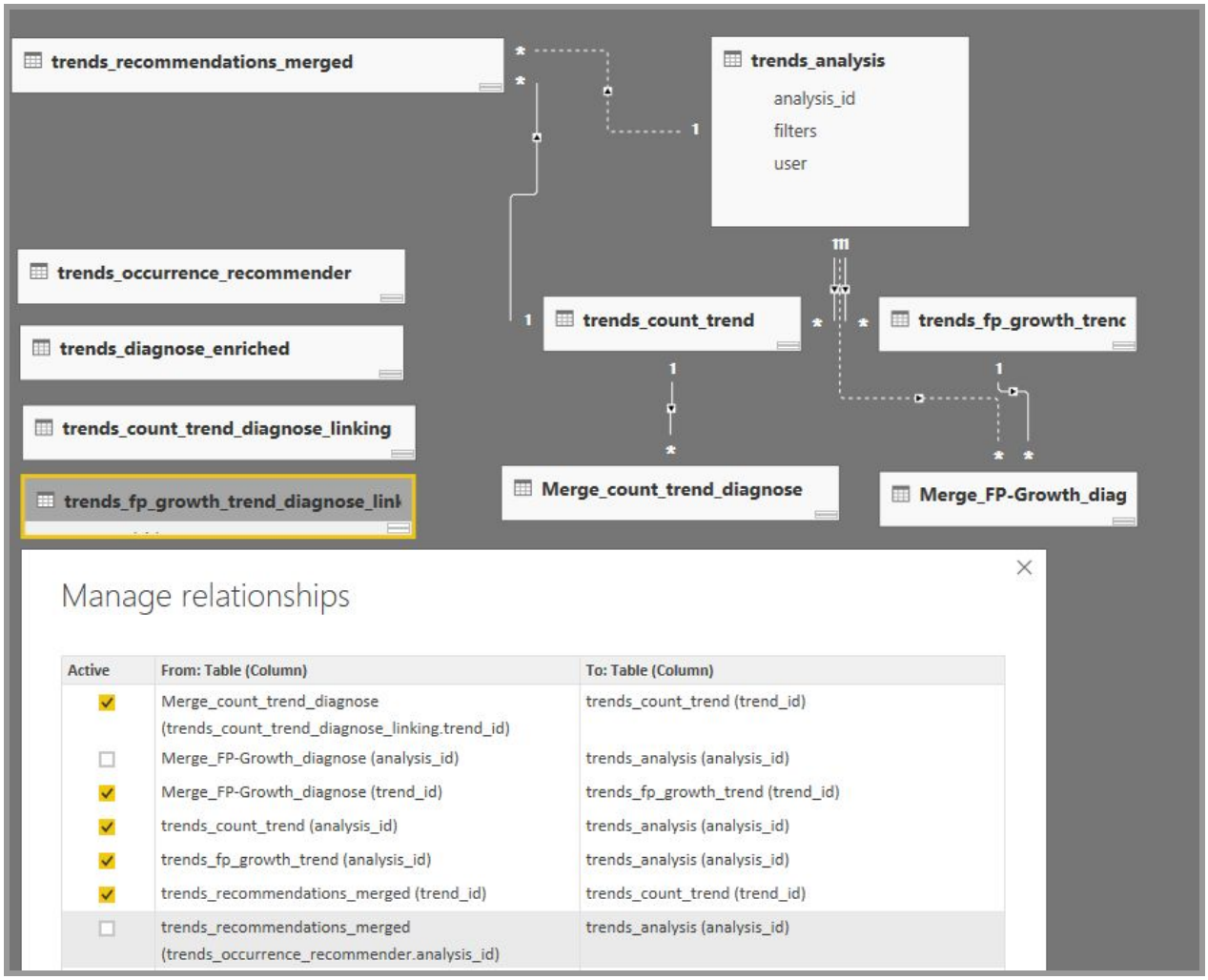

Figure 24: Database tables and relationships in Power BI.

Appendix H: SIG Feedback

[First feedback, 12-06-2018]

“De code van het systeem scoort 4.2 sterren op ons onderhoudbaarheidsmodel, wat betekent dat de code bovengemiddeld onderhoudbaar is. De hoogste score is niet behaald door een lagere score voor Unit Size.

Op dit moment is de score dusdanig hoog dat we geen concrete aanbevelingen voor verdere verbetering hebben, hulde! Wel is het zaak om ervoor te zorgen dat jullie dit niveau tijdens het vervolg van het project vast weten te houden, en al helemaal op het moment dat de deadline in zicht komt.

De aanwezigheid van testcode is in ieder geval veelbelovend. De hoeveelheid tests blijft nog wel wat achter bij de hoeveelheid productiecode, hopelijk lukt het nog om dat tijdens het vervolg van het project te laten stijgen.

Over het algemeen scoort de code dus bovengemiddeld, hopelijk lukt het om dit niveau te behouden tijdens de rest van de ontwikkelfase.”

Appendix I: Info sheet

Info Sheet - Analyzing location-specific patterns in train data

Client organisation: Nederlandse Spoorwegen (NS) **Final presentation:** 03-07-2018 (16:30 - 17:30)

Description

The main challenge of this project was to give insight into location-specific problems of trains. Location-specific problems are problems that are not caused by the train, but by the infrastructure or human fault at that specific location. When such insight is provided, the NS is able to tackle the problems and thereby improving punctuality and reliability of their trains. During the research phase, we learned which frameworks, programs, and algorithms could assist us in finding location-specific error code patterns. The used techniques were PySpark, Power BI, Jupyter and FP-Growth Frequent Pattern Mining, which can all be used within the existing big data environment of the NS.

This insight is given by creating an application which analyses error code messages from trains and tries to find patterns in the error codes on specific locations. With the system, the NS is able to verify and create new hypotheses on possible problematic locations. Challenges that come with this are the unknown domain of trains, the lacking of validation data, using the existing software environment of the NS, and dealing with large amounts of data. The output of analyses is verified by close collaboration with field experts.

The Scrum methodology was used throughout the project. Communication between team members and the client was strong, which enabled better alignment of thoughts through the development of the system. The system has been handed over to the client and will be further developed and put into use. Given recommendations, such as implementing multiple train types and partitioning the results in the database, will be evaluated and implemented if seen necessary by the client.

Members of the project team

Henk Grent - Key contributions:

- Implementation of the Blacklist filter.
- Front-end and middle layer for interacting with the back-end, creating analysis jobs and storing settings.
- Designing database structure suitable for analysis specific querying and Power BI visualization.

Mark Haakman - Key contributions:

- Preprocessing input data by gathering data from different sources and removing irrelevant records.
- Creating database structure to store large amounts of data in an efficient and understandable way.
- The simple but powerful analysis which counts error codes per location and their paired delay.

Frenk van Mil - Key contributions:

- Post-processing of analyses results with a prediction model using Collaborative Filtering.
- Implementation of the FP-Growth Frequent Pattern Mining algorithm on the data.
- The core implementation of the analysis and filter engines.

Sander Waij - Key contributions:

- Implementation of the FP-Growth Frequent Pattern Mining algorithm on the data.
- Handle the reads and writes to the database in a coherent manner.

- Optimizing memory usage throughout the project.

Client: I. Kalsbeek, Maintenance Development Engineer, Nederlandse Spoorwegen
Coach: M. de Weerdt, Algorithmics Group, EEMCS, TU Delft
Contact Person: F. van Mil, superdeboeren@hotmail.com

The final report for this project can be found at: http://repository.tudelft.nl

Appendix J: Executive summary

Analyzing location-specific patterns in train data

Executive Summary

A system was created for giving insight into location-specific error patterns in train data. Location-specific problems are problems that are not caused by the train, but by the infrastructure or human fault at that specific location. This is an entirely new system within the analytics environment of the NS using big-data analysis in order to find location-specific patterns. It also is the first system within the analytics environment that attempts to make a more centralized environment for long-term, user-friendly data analysis.

The system is designed to be a manageable, scalable and high-performance system for running varying types of analyses, data filters, and post processors with a running database connection, which is open for end-user interaction and customization. The design of the system ought to make it easier to implement new types of analyses, filters, and post-processors and make it easier to visualize the results of the analyses. It also provides an attractive user interface (UI), which allows the user to run analyses, customize analysis settings/parameters and a separate UI for data visualization (Power BI). The system comes with some implemented analyses, filters, and post-processors.

Objective

The objective is to create an application that can be integrated within the current software environment of the NS, in order to gain insight into location-specific error code trends to prevent future breakdowns and faults. The target demographic consists of two end-users: the Reliability Engineer and the Fleet Analyst.

Relevance and added value

A long-term goal for the maintenance and development department of the NS is to use data generated by trains to improve punctuality and reliability of the trains while balancing the costs of train maintenance. Punctuality and reliability will lead to better customer satisfaction, which includes the ordinary traveler. Being able to detect location-specific observations helps with addressing infrastructure issues, which will both prevent future breakdowns/delays and reduce costs due to unnecessarily sending trains through maintenance to fix problems that were not related to the train. Other applications, such as getting an insight of which mistakes are often made for a certain location by human follow and add value in a similar manner.

Technologies

For big data processing Hive and Pyspark were run on the NS' cluster. Data was also stored in the HIVE environment. For the system engineering part, Python was used to implement the designed architecture. The visualization was done using Power BI desktop.

Follow up

Considerable time was taken to hand over the system to two employees of the maintenance and development department, they will decide what to do with the system based on the in the final report mentioned recommendations. Ultimately they will hand it over to the DNA production environment. Extensive documentation for a system-engineer and end-users has been added. The DNA will further extend the program and make it into one integrated system.

Executed by:

Henk Grent, TU Delft, h.a.grent@student.tudelft.nl
Mark Haakman, TU Delft, m.p.a.haakman@student.tudelft.nl
Frenk van Mil, TU Delft, f.c.j.vanmil@student.tudelft.nl
Sander Waij, TU Delft, s.waij@student.tudelft.nl

Client:

Inge Kalsbeek, NS Maintenance Development, inge.kalsbeek@ns.nl

Coach: Mathijs de Weerdt, TU-Delft, M.M.deWeerdt@tudelft.nl

Appendix K: Individual contributions

Table 10: Significant individual contributions made to components (marked with "x")

Component	Henk Grent	Mark Haakman	Frenk van Mil	Sander Waij
Filters			x	x
Post processing			x	
Jupyter UI	x			
UI event handling	x			
Job scheduling	x			
FP-growth analysis			x	x
Counting analysis		x		
Analysis Engine		x	x	x
Database connector		x	x	x
Power BI	x			
Generating input data from raw data		x		

Appendix L: Systems engineering guide

Introduction

This document serves as a guide for the system engineer which will maintain this application to give him insight into how the application works.

1. Application flow

How to start the application in the front-end is explained in the *User Guide* document. When the user starts an analysis, an *AnalysisEngine* is created. The *AnalysisEngine* first applies every filter specified by the user. This is done by calling its *FilterEngine*. The *FilterEngine* reads the settings for the filters and instantiates the filters using a decorator pattern. When the filters are applied on the input data, the *AnalysisEngine* starts running the different analyses one by one. When an *Analysis* is finished, it returns a Spark DataFrame with found trends. These trends are converted to the format used in the database and thereafter written to the database. The database now contains the result of the analysis and the application terminates.

2. Input data for the application

The main input to the application is the table with diagnose data from trains. This table is enriched in the Jupyter notebook *Create enriched diagnose table.ipynb*. From the 'materieelbewegingen' table, the diagnose table is enriched with data about where the train was when it throwed the diagnostic code. The most important column added here is the *track_section* column. The diagnose table is further enriched with the amount of delay in minutes paired with each diagnose entry. The delay data is fetched from an csv handed to us. How each diagnose entry is linked to a delay (with zero of more minutes), can be found in the notebook file.

Another, smaller, input to the application is the table with amount of trains on each track section in one week, with the normal timetable. This table is generated in the *Trains per track section - avg of 10 weeks.ipynb* notebook file. This data is used to calculate the relative occurences of some error trends.

3. Analyses

There are currently two types of analysis, which can be found in the analyses.py module. This chapter will covers each analysis and also explains how to add more analyses.

3.1.1 CountingAnalysis

In the counting analysis, every diagnostic code on each track section will be counted. This gives insight into track sections where a lot of error codes are thrown. To be able to give the list of found trends some prioritization, multiple informative column are also added.

- train_amount: The amount of distinct rolling_stock_numbers on this track section which has thrown this diagnostic code.
- occurrences: The amount this diagnostic code is thrown in total on this track section.
- rel_occurrences: Occurrences amount divided by the total amount of trains on this track section.
- delay_minutes: The delay in minutes paired with this trend.

3.1.2 FpGrowthAnalysis (Pattern Mining)

In the FpGrowthAnalysis, all diagnostic codes on each track are mined for patterns following the FP-Growth Frequent Pattern Mining (FPM) algorithm[11]. For the implementation of the algorithm, the PySpark implementation of FPM is used[12] [13] of the Spark version 2.2.0.

3.2 Adding more analyses

To add a new analysis, only a new class must be created in the *analyses.py* module. For this class three required methods must be specified for optimal behavior. The *AnalysisEngine* will immediately detect new analyses and instantiate them. The system does not allow priority execution, as each analysis is expected to be fully independent of other analyses.
All analyses are executed after all filters are applied (see 4. Filters) and before the post process is executed (see 5. Post processing).

All new analyses should at least follow the contract of the below given code snippet.

Analysis - Code Snippet #1 *analyses.py*

```
class NewAnalysis(Analysis):

    @property
    def is_finished(self):
        return finished  # boolean indicating the analysis has finished.

    def execute(self, data_frame):
        return result  #resulting data frame.

    def convert_to_db_tables(self, data_frame, analysis_id):
        return tables #A list with tuples containing the location and DataFrame of the data
          to write to the database.
```

[11] Han, J., Pei, J. & Yin, Y. (2000). Mining Frequent Patterns without Candidate Generation. *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, 1-12. doi: 10.1145/335191.335372
[12] http://spark.apache.org/docs/2.2.0/api/python/_modules/pyspark/ml/fpm.html
[13] https://spark.apache.org/docs/2.3.0/ml-frequent-pattern-mining.html

Optionally, one of the other methods of Analysis can be overridden. These methods are designed to be general for different analyses but different behavior might be desired.

4. Filters

Currently, the application has the following filters, which can be viewed in the filters.py module:

- DuplicateFilter: Filters duplicates (i.e. excessive errors that are created for the same problem) out of the data.
- ServiceTrackFilter: Filters out the error codes it expects to be triggered during maintenance.
- TimeFrameFilter: Filters out all data that comes before the given start date and data that comes after the given end date.
- SeverityFilter: Filters out diagnose codes for all the severities not specified by the user.
- BlacklistFilter: Filters out all the data belonging to a blacklisted trends specified by the user.

Adding more filters

To add a new filter, create a new class in the *filters.py* module which extends the *FilterDecorator* class. You have to overwrite the *__init__* and *filter* method from *FilterDecorator*. In the *__init__* method, the filter attributes can be set, the *filter* method gets an input Data Frame, removes rows from it by using a filter and returns the filtered Data Frame. You also have to instantiate the filter in the *filter.filters.FilterEngine* class.

To make the filters configurable, the filter must be registered in the *filter.filters.json*. Furthermore, in the *filter.filters.FilterEngine* class *__init__* function the filter must be configured as desired. All settings for filters are read in JSON format.

All new filters should at least follow the contract of the below given code snippet.

Filter - Code Snippet #1 *filters.py*

```
class NewFilter(FilterDecorator):

    def filter(self, data_frame):
        return filtered_data_frame  # data frame with filtered data.
```

5. Post processing

Post processing is the process after all analyses have finished. The post process is meant to process the found trends. A possible example could be to post process the trends for statistics, e.g. ‘how many trends were on each track section?’ Another possible post process, implemented in the program, is the Collaborative Filtering algorithm (abbr. CF). The CF algorithm is used for finding similarities between the trends found on track sections and the algorithm tries to guess for some trends what their frequency might become in the future for a track section.
The above example and all future post processes are instantiated by the *PostProcessor*. Just like the

analyses, all post processes are automatically instantiated after the coding guidelines have been followed (see below). Each new post process is a child of the *PostProcess* class in the *postprocessing.py* module. Post processes support multiple results, as all results are returned as tuples of undefined sizes.

All new post processes should at least follow the contract of the below given code snippet.

PostProcess - Code Snippet #1 *postprocessing.py*

```
class NewPostProcess(PostProcess):

    def execute(self, data_frame):
        return tuple(data_frame) # tuple of the resulting [0,n] data frames.

    @property
    def database_names(self):
        return ["path.to.database"] # List of two paths; the read and the write database.
```

6. User Interface in Jupyter

The UI.ipynb file contains the User Interface of the application and should be started by the user to run the application. This notebook file works by adding a Jupyter event listener to every button, menu and text field. When a button is pressed, the event listener will be triggered and executes a specified function. Most of the events will be passed to the *EventHandler* class, which handles the event by for example starting an analysis or changing the settings.json with the new settings.

Data flow

When the UI is started a number of events happen and some objects are created in the background. Their interaction is visualized by the *UI Data flow diagram* in *Appendix A*. Upon launch, the first object that is created is a *JobListener.* The JobListener has an *AnalysesQueue* object and the AnalysisQueue object has an *AnalysisEngine.* The JobListener will periodically read the *analysis_jobs*.json file and checks whether new jobs were added. If a new job was added, then the listener will add it to the queue of the AnalysesQueue which will one by one execute the jobs that are in its queue. The queue is a type of priority queue, although the priority flag is currently not used. Upon the execution of a job, first the *settings* attribute of the job is loaded to the *filters.json* file and then the AnalysisEngine's *execute* method is called, which will load the *filters.json*'s content into its filter engine and finally run the job in the HIVE/Spark environment. Upon the initiation of the AnalysisEngine a Spark and HIVE context are created, which is why the UI takes some additional time to load before being ready for execution.

Start analysis

Using the *start analysis* button the user can start an analysis. When this button is pressed, a job will be created. This job is an item in the *analysis_jobs.json* file, with a number of attributes. The settings, as specified by the user through the UI, will be loaded from the *settings.json* file and are passed as an

attribute of such a job. The execution time of the analysis (*exec_time*), given as a valid date time string, and the amount of hours until the next execution of the given job (*exec_time_dt*) are other attributes. If *exec_time_dt* is set to 0, then the job will only execute once and it will be executed and finished directly.

A job in *analysis_json* will look like this:


```
{
    "grenth0": {
        "exec_time": "2018-06-13 17:19:33.984094",
        "exec_time_dt": 5,
        "repeating": true,
        "settings": {
            "date_time": "2018-07-01 00:00:00",
            "date_time_deact": "2018-07-30 00:00:00",
            "duplicate_threshold_sec": 59,
            "exec_time": "2018-06-13 17:19:33.984094",
            "exec_time_dt": 5,
            "remove_duplicates": true,
            "rtm_only": true,
            "service_track_filter": true,
            "severities": [
                "A",
                "B"
            ],
            "train_types": [
                "VIRM"
            ]
        },
        "status": "Not executed."
    }
}
```


Stop Analysis

The *stop analysis* button will stop the analysis at the next possible moment. No hard interrupts are chosen, because this may cause IO errors when interacting with the database. However, hard interrupts are possible by resetting the kernel. The next possible moment is the next safe moment, which is after a full step of a type of analysis. What a step is depends on type of analysis, as such the *CountingTrend* can only be stopped after being (basically) fully finished. This is because the majority of what that type of analysis does is contained in one big aggregate function. On the other hand, *FpGrowthTrend* can be stopped after the analysis of each track section.

7. Trend visualisation in Power BI

Power BI has been chosen as the main tool for data visualisation. The connection with the database is established using the HortonWorks ODBC 64-bit connector, as described in the the manual *Introductie - Tools DTA omgeving (lite versie)* in the section *Power BI* (Hive).

The data from the production environment is used as it is stored in their corresponding tables, without the transforming or inferring data in Power BI. The used tables, and how they relate to each other, can be seen in the *relationships* tab.

Visualizing Data

In Power BI the data provided by executing analyses can generally be found in the *default.trends_x* tables. Data stored in these tables can be found in numerous ways. The following section explains some core elements which are useful for making different visualisations. An example can be found in the repository: *main.pbix*.

Record ID linking
Linking tables exist which link the trends to their corresponding, individual diagnostic error code rows in the database. This database, containing the diagnostic rows, is *default.trends_diagnose_enriched*. This can be used to add drill through functionality, which visualizes the corresponding rows. It can also be used to count the amount of corresponding rows in the top level view by counting the distinct record_id's alongside trend_id's.

Aggregate values
Power BI can also aggregate any of the diagnostic error codes' columns. The average GPS speed for all diagnostic rows belonging to a trend is an example of this. It is worth noting that this negatively impacts performance, as such a number of such aggregate values are provided in a trend's row. A*ggregate values* are also called *summarized values*.

8. Tests

To run the tests, execute 'python3 -m pytest tests/' in a terminal in the main folder of the application.

Dependencies

Below is a list dependencies the application is build on. The list may be incomplete. Note that the lists contains the versions that were used for the development of the application. The application may not be restricted to these versions. It is not recommended to use a version lower than specified in table 1, higher versions can be used if backward compatibility is guaranteed or not required. Versions indicated with an asterisk are strict requirements. Versions below the strict requirements are guaranteed to break the program.

Table 1: Application dependencies for the application. Versions specified as '*<default>*' are the versions supported as default in the NS environment.

Dependency	Version
Python	3.5

PySpark (Spark)	2.2.0*
Power BI	<default>

Appendix M: Research Report

Bachelor Project 2018

Research Report

Henk Grent

Mark Haakman

Frenk van Mil

Sander Waij

05-2018

Coach: M. de Weerdt, TU Delft

Client: I. Kalsbeek, Nederlandse Spoorwegen (NS)

TU Delft

Executive summary

An application will be created giving insight into location-specific errors. For the interface, Power BI will be used. PySpark in the Jupyter environment will be used based on a Hive database for data storage and data processing. Simple statistic trends and FP-growth Sequential Pattern Mining enable the application to find trends in the error data. This application shall be developed over a period of 10 weeks. The final application will be delivered on 22-06-2018.

Table of contents

Introduction

This research report contains the problem statement and possible approaches to solve it. One approach will be chosen, with proper motivation. Thereafter, different solutions for frameworks and algorithms will be researched. Furthermore, use cases and requirements will be created and the minimal viable product will be defined. At last, the initial release plan will be outlined.

Problem statement

Equipping trains with sensors, and collecting (real-time) data has proven to be useful for the NS, in order to be able to detect and respond to problems faster. A long term goal for the maintenance engineering department of the NS is to use data generated by trains to improve punctuality and reliability of the trains while balancing the costs of train maintenance. A faster response time helps with improving punctuality and reliability.

Employees of the maintenance and development department have to make the trade-off between premature maintenance sessions and failures due to a lack of maintenance check-ups. Premature maintenance sessions cost money and put more stress on the service stations. Train failures result in extra costs and are detrimental to the punctuality and reliability. Premature maintenance thus should be done as few times as possible, whereas failures which could be prevented with a maintenance session are to be minimized as well. These are two contradicting goals for which achieving the optimal balance is a goal.

At the moment, there is no system implemented that allows for the detection of location-specific error code trends, whilst there is data that could possibly indicate the existence of location-specific problems. As a result, it is hard to detect which problems structurally occur or suddenly arise on given railway sections. Trends are being found, although they are now mostly based on human notice. Moreover, the data sent by trains is currently not used for location-specific analysis, which implies that there is still a lot of room for trends to be found. This project will be one of the first steps toward location-specific analysis. Additionally, it is hard to know whether the errors are caused by the train itself or the tracks it is using, this results in unnecessary maintenance check-ups and delayed repairs to the tracks. It could now be the case that a train appears to be causing troubles, although the track it was riding on initially caused it. Being able to find such trends would contribute to achieving the long-term goal of improving punctuality and reliability. Additionally, being able to detect location-specific errors allows the NS to trace problems back to the tracks/infrastructure and not the trains. This would prevent unnecessary and costly maintenance sessions.

Objective

The objective is to create an application that can be integrated within the current software environment of the NS, in order to gain insight into location-specific error code trends to prevent future breakdowns and faults. The target demographic consists out of two end-users: the reliability engineer and the fleet analyst.

End users

In this chapter, the end users of the applications are defined as our contact and the function of the given end user.

Reliability Engineer

Who: T. van Haperen
Function: The function of the Reliability Engineer (RE) is to monitor the reliability of an assigned train type. His task is, among others, to ensure the reliability and safety of the trains, within a given budget. He has to make sure the performance goals are made. To achieve this, he wants to find recurring problems and solve them in order to protect his fleet or improve its performance. Not all these problems can or should be solved, which means that he has to make a decision if he wants to solve them. Some problems might not be worth the money or time to fix.

A possible reaction to a problem he finds is giving additional instructions to for example mechanical engineers or adapt the training the train operator gets. He can also communicate the errors with the railway operator ProRail with results he found, so they can fix the problem.

Fleet Analyst (Vlootanalist)

Who: M. Schulte
Function: The function of the fleet analyst (FA) is to look at problems that are going on right now, determine where and in which train or area those problems occur and delegates this problem to the task force in the OCCR. When a problem occurs due to the interface between the track and the train, this can also be communicated with the OCCR, so the train itself is not unnecessary inspected.

Current system

Nowadays, location-specific incidents are only noticed by aware employees, who are noticing recurring events. Data of hardware is available though, provided by the Real-Time Monitoring system, which will be explained in this chapter.

RTMO / RTMA

Real-Time Monitoring Operations (RTMO) is an integrated environment with a visual interface used for by the operations side of the NS. This environment was once outsourced to be developed by another company. RTMO is primarily used by the Fleet Analyst to see what problems are going on at the moment and connects those problems to actions for the OCCR taskforce. RTMO uses data from the last maintenance session, which is, on average, up to 3 months ago.

Real-Time monitoring Analytics (RTMA) is a concept used in the data analytics environment. This environment has no universally used platform/API, nor does it have a universally used way of displaying data. RTMA is primarily used by the Reliability Engineer and other technical analysts. RTMA contains all available data fetched from sensors in trains and is not time-bound.

Available data

Currently, the VIRM series of trains are equipped with up to 4500 sensors and 200 data-points are stored in the database. The computer inside the train will generate diagnostic codes based on data from these sensors. These diagnostic data can have different severities, ranging from “door opened” to “emergency break activated because of a crippled signal”. This diagnostic data is sent to the database of the NS and available for the application which will be created. This is the data which can be used to discover location-based error trends. Some attributes are not useful, because they, for example, are always equal to *NULL*. The useful columns of the RTM database table can be seen in Table 1.

Table 1: Overview of useful columns from the RTM data send by trains.

Datatable: train_maintenance_rtm.diagnose		
Column name	Description	Example
gps_speed	The speed of the train measured by the GPS sensor km/h	0.12485
gps_valid	Can the GPS data can be considered valid?	true
class	The severity of the diagnostic code	A
date_time_act	The activation time of the event triggering the diagnostic code	2016-03-23T00:59:13
date_time_deact	The deactivation time of the event triggering the diagnostic code	2016-03-23T01:03:13
diagnostic_report_code	The diagnostic code	REM234

diagnostic_report_description	Description of the code	Remkraan niet in A. in niet bed.cabine
latitude	The latitude read by the GPS sensor	52.593149
longitude	The longitude read by the GPS sensor	6.156789
rolling_stock_name	The type of train	VIRM
rolling_stock _number	The number of the train	8637

Furthermore, a database is available which contains data stating where each train should have been at a specific point in time and where it actually was. Together with the timestamp of a diagnostic error gathered from the RTM database, this data can be used to track down its timetable point ("*dienstregelpunt*") during the error. The useful columns of this database table can be seen in Table 2.

Table 2: Overview of useful columns from the database containing the movement of the trains.

Datatable: default.materieelbewegingen		
Column name	Description	Example
treinnummer	A number unique for the ride that day. *English translation: train number.*	702227
dienstregelpunt	A point which divides the track into multiple sections. *English translation: timetable point.*	Abr
uitvoeringstijd	The actual timestamp the train was at the timetable point. *English translation: execution time.*	2017-05-02T06:03:00
rijkarakteristiek	Indicates what the function of the train was. For example IC for operating as an intercity, driving people around and LM for a train without passengers in it. *English translation: driving characteristics.*	IC
materieelnummer	The number of the physical train itself. *English translation: rolling stock number.*	8715

To get more information on the situation of the location of an error or trend found, the table 'dienstregelpunten_with_gpslocation_and_fullname' gives insight of the surrounding of the GPS coordinates given. The table shows the type of the area (e.g. a bridge or station) and a respective description of the timetable point (e.g. 'bridge over the Arne'). The important information from this table can be seen in Table 3.

Table 3: Overview of useful columns from the database containing the description of the area of gps locations and the respective names of the timetable points.

Datatable: default.dienstregelpunten_with_gpslocation_and_fullname_kopie04052018		
Column name	Description	Example
dienstregelpunt	A point which divides the track into multiple sections. *English translation: timetable point.*	Abr
gebiedtype	The type of the area shortly describing the situation of the location. *English translation: area type.*	STATION
Naam	The description of the timetable point. *English translation: Name.*	Brug over de Arne

Available tools at NS

Big data processing

The following tools are currently available: Apache Hadoop, Hive, Mahout, Pig, Zookeeper, and Spark. These tools are currently being used by the NS. These tools could also be useful to use as our solution to the problem will probably depend on processing a big amount of data.

Visualization

For visualization NS primarily uses Microsoft Power BI. As quoted from the front page of the Power BI site[14]: "Power BI is a suite of business analytics tools that deliver insights throughout your organization. Connect to hundreds of data sources, simplify data prep, and drive ad hoc analysis. Produce beautiful reports, then publish them for your organization to consume on the web and across mobile devices." This tool is used by RE's and data analysts to display their data. Other data visualization techniques for the RTMA environment are incoherent, usually various Python libraries or textual interfaces such as Excel.

[14] *https://powerbi.microsoft.com/en-us/*

Vision

In this chapter, the long term vision of the maintenance and development department of the NS will be outlined. In the interest of time, we can not implement the entire vision, so not all elements described in the below can be included in this project.

In an ideal system, the train itself would publish an interface displaying what kind of problems there are and what could be done. This system gives a notification if a system within the train shows deviant behavior. The users can interact with this system by indicating what kind of behaviour is normal and what kind of behaviour should be classified as deviant, as such the system will learn. The feedback will be at different levels: operational and strategic. Such a system, using pattern recognition based on user-feedback and objective, statistical analysis will be a robust and extensive method for analyzing train data.

On an operational level engineers will get a clear indication of which part of the system of a train is malfunctioning and what steps can be taken to solve this. Engineers would also have an accessible method of indicating whether the provided plan was effective for solving this problem and whether the system that was indicated to be malfunctioning was indeed malfunctioning. Regarding raw sensor data, engineers shall haven an interface to analyze patterns in raw sensor data and specify what conditions lead to the triggering of a diagnostic error code, in order to enrich the information the system can provide at an strategic level. This would also make it easier to identify what triggers are missing, and index what sensors need to be added or what conditions need to be identified by the means of raw sensor data.

On a strategic level the interface indicates if train malfunctions are due to the infrastructure. The system gives the user enough information about where and why the malfunctions happen. That way, the user could either go to the infrastructure manager and let him fix the problem or he could adapt the training the train operator gets for that part of the infrastructure. The information displayed shall help the users, functioning at a strategic level, find trends in the data and help backing up claims with numbers. Furthermore, the train could indicate the estimated time it can still be operational before maintenance is really needed and adjust train maintenance schedule accordingly. Also, information about the most likely financial/quality of service implications of taking certain actions can be given, in order to guide users in making choices within the business case.

Regarding maintenance and expanding this integrated system, ideally computer engineers would have one centralized environment in which the code is maintained. Components could be added and changed at one level of analysis, e.g. the raw sensor layer, without having to change the implementation of the layers above. At each level of analysis, end-users would perform their analysis in a centralized environment through a uniform interface.

Below the design goals are defined, which are defined as such that they will cover most of the needs in the vision. How these design goals are implemented in our application is described in the chapter ‘Different approaches’, where different types of approaches are defined and compared.

Design goals

In this chapter, the design goals of the application are defined. These design goals are a broad overview of what should be included in the application, in order to satisfy most needs defined in the vision. For each of the design goals, a short name defining the goal and a description are given.

1. **End-user satisfaction:** The final product should satisfy the long term goal of the NS and should aid the end-users in helping them achieve this long term goal. Customer collaboration should be valued in order to achieve this design goal.
2. **A quick insight into data:** Error trends should be visible in one eyesight by summarizing the data. Further information regarding each trend should also be easily accessible.
3. **Manageability:** The application will be made using techniques easily integrable by the NS, additionally the code-base should be able to be maintained well. Independent layers of analysis, as described in the vision, would warrant this design goal.
4. **Performance:** The application should not overload the server. Users should be able to obtain the results to their queries in a reasonable time, with respect to the complexity/size of the query.
5. **Scalability:** Given collecting data has proven to be useful for the NS, plans for increasing the amount of collected data have been made. In order to accommodate for this increase in collected data the application should be scalable.

Different approaches

In this chapter, different approaches will be discussed which will solve the problem statement.

Approach 1: Offline analysis

The first approach to address the problem is providing a searchable overview of error trends, which is generated by data from past diagnostic data. An analysis is activated by the user and the analysis terminates after the results are found. This type of analysis will focus more on finding complex correlations in the already existing data and extensively being able to search the existing dataset. This allows the user to modify the fleet or talk to the infrastructure owners, with relevant data as proof.

This approach can use either the table- or the map interface, as discussed in the section ‘considerations’ below.

Advantages: No live connection with the database has to be maintained. Additionally, no listener for the data stream has to be made which makes the application more lightweight in terms of server load and computational power. Implementing this approach will allow users to inspect diagnostic errors, which is not possible right now. This way, the reliability of the train fleet can be increased in the long term. An approximation of real-time can be simulated by taking a timeframe covering the last few minutes (e.g. the last 2 minutes) of data. This near real-time approach can still find trends in the last few minutes, which comes close to the time required for a train operator to report problems in his/her train. This means that the problem of neglection of reporting problems by train operators can be partly solved through automatic reporting. Besides, some of the trends in the last few minutes might not be detected by the human eye at

first glance, which means that some already caused problems might only become visible the train operator in the coming few minutes, which means that the system could be faster than physical detection.

Disadvantages: Using this approach, problems occurring right now with the train fleet will not be detected. Therefore, this approach will not improve the train punctuality in the short term.

Approach 2: Real-time analysis

The second approach to address the problem is providing an application which monitors the active fleet and notifies the user of currently occurring error trends. The program runs continuously until the user fully closes the application. This type of analysis will focus more on indicating when boundary conditions for trends are met and notifying the end-users of the trend.
With this design the focus will be on being able to link the live data to current trends using recent data, so the problem can be fixed immediately. Currently, when there is a failure in a train, which causes the train to have unwanted behavior, then the train operator first has to report this to the helpdesk. The time between the occurrence of the failure, the report to the helpdesk and the action could be reduced by introducing a real-time detection. A problem with the current system is that some operators do not report all their problems anymore, as they occur too often or they already found a workaround to temporarily fix them. This, however, means that some problems are not detected and can therefore never be solved.

This approach can use either the table- or the map interface, as discussed in the section 'considerations' below.

Advantages: The real-time approach has the advantage of being able to see the trends at the moment of happening. This means that when a trend of errors is found, there can be a direct action on the event. This saves a phone call to the helpdesk, which could have only been done when the train operator had time (and was therefore often done late, if at all), and solves the negligence in reporting occurring problems.

Disadvantages: The disadvantages of the real-time approach is that the data on which trends can be focused on is restricted to a smaller set, than the data of an offline analysis. In the real-time implementation, a shifting time frame is used, which delimits the data that can be looked at. Because of the speed of data flowing in, this frame can be expected to be significantly smaller than the timeframe possible for offline analysis. Additionally, a listener is needed to query the database which makes for a constant load on the server and the computers running the program.
Furthermore, in order to be able to have a real-time application, it is required to also have the data available in real-time. This is currently not the case for all tables. Some tables, like the 'materieelbewegingen' are only updated once a day. Alternatively to this table, the geofencing can be used , as this is real-time, but the accuracy might be lost. Geofencing will probably improve over time, which means that in the future this problem might become irrelevant.

Feedback

During the plenary session on April 30, we retrieved feedback regarding the two approaches from the RE and the FA. Both of them voted in favor of the offline approach, which is also our preferred approach. The main reasons for choosing the offline approach were a delay in workforce response time and the lack of long-term analysis tools.

Regarding the delay in the workforce, the FA said that 'it can take hours for the workforce to be present on the location of the problem, as such getting the detection of trends by the minute is not as useful'. This real-time analysis is also not something that caters towards the wishes of the RE, as he does

the long-term analysis. Furthermore, a long-term location-specific analysis tool for the RE does not exist in a well-rounded form. As such, opting for an initially long-term focused offline analysis is the best solution for both end-users.

Conclusion

As a team, we preferred the offline option as well because the implementation of offline analysis is theoretically easier and it appears to add the most value to the NS as any such long-term analysis tool is not implemented. Also, considering the long-term goal of the NS, implementing the offline version, long-term focused solution first would help engineers with balancing maintenance costs better and help prevent train delays due to, for example, detected infrastructural problems. The latter would improve punctuality, the former would be achieved because the problems can be traced back to the infrastructure better, thus preventing unnecessary maintenance sessions. The considerations can be read in more detail in the *advantages* and *disadvantages* above.

Considerations

Table vs. map interface

The first interface consists out of a table, of which the rows represent error trends and can be clicked to reveal further information. The error trends are ranked in prevalence and severity. The second interface consists out of a map, which visualizes the locations of the error trends. These trends appear as pins on the map and can be clicked to reveal further information.

Table interface	**Map interface**
+ Can display more information without making the interface really cluttered. + Easier to implement: time can be used to create better analysis tools. + Can be ordered on certain properties.	+ Provides a clearer overview regarding where on the map trends occur + Visually aesthetic + Visual connections are easier to make. This allows for easier detection of geographically close trends.
- Geographically close trends are not immediately obvious - Number of columns before squeezing text is limited without horizontal scroll. Horizontal scroll limits the amount of data a user can see at once.	- Can get cluttered when a lot of information has to be displayed - The NS does not have a coherent system for displaying specific railway locations on a map. Until the NS has one integrated system for displaying geographical data, it may not be worth creating our own full-fletched interface.

Independent of both implementations, filters can be applied to configure the analyses. These filters are predefined filters in which the user can change the values. For example, a timeframe filter can be applied, where the user can change the start and end time.

Feedback

These two approaches were discussed with the end users of this program, which are the RE and the FA, on the plenary session of April 30. The map interface is the approach that is appreciated the most, both by the FA and the RE.

The reason for the FA to prefer the map interface over the table interface is that the map interface provides the most information in the least amount of time, while a table interface is too complicated to easily extract information from it. For this reason, a map is ideal, as it gives a good overview. A table will, however, be used as a means of providing detailed information on the error. This enables the FA to order the data such that it has the most benefit for him.

The reason for the RE to prefer the map interface over the table interface is that the RE needs to know which timetable points have problems, which are much easier visible on a map than in a table. With information on the timetable points, he can decide which problems need to be addressed.

Conclusion

As a team, we are convinced that the map interface is the optimal way of visualizing the problems. Since the way this will be implemented is just a simple map with points at the coordinates of the most important errors, this will not add too much complexity, while still providing a lot of clarity in the visualization of the locations of the errors. A map will provide the users with a quick overview of the location of the errors, and in that way easily outclasses the table interface. To get the right information on the map, users could configure their filters by setting their filters to their desired values. We still think it is better to have a table next to the map, as the map can get cluttered when there are many trends. The table also provides more information. Not all information will be provided in tables, as this would only add noise to the screen. The information that will be provided in tables will be mostly covering an overview of all trends found.

Query the data

The user shall be able to search for specific error codes and areas. These searches will filter the data, which makes the user able to focus on more specific subsets of data. The user shall be able to indicate a time-frame which he wants to analyze. Additionally, the user shall be able to analyze the data-set with more complex queries. These queries can be implemented with two different sub-approaches:

- **Query search:** The user could search in the data by using search queries. These queries will be similar to business rules used in the current RTMO system. An example query could be: "*error_code* = *ATB3103* **AND** *location* = *Amsterdam*". Here the italic components are variable components, which can be edited by the user. The bold operators (i.e. '=' and '**AND**') can be chosen from a list of standard operators. This list of operators could consist of the following values: =, ≠, <, >, ≤, ≥, AND and OR. For the error codes, there will be an extra option to search for the error type, i.e. the first three letters of an error. In the example, this would be 'ATB'. The variable components could consist of names of table columns in the database and their respective values.
- **Script search:** The user could search in the data by using script search. This scripting could be done using Python, as the client has a preference for this language. The working for this would be that the user would make a filter in python on the data. This filter could add data-points from a dataset to the query result if the specific conditions specified by the user are met. The filter can be seen in Algorithm 1 and the specification of specific conditions can be seen in Algorithm 2.

Algorithm 1: Background algorithm.

Input: the set of data D[d_1,d_2,..,d_n].
Output: the set of filtered data.
Description: background algorithm used to call the custom algorithm of the user.

```
1:  Foreach d_i ∈ D:
2:    If custom_filter(d_i) then      // The function custom_filter/1 can be changed by the user.
3:      Continue;                     // Do nothing. Keep d_i in D.
4:    Else
5:      D - {d_i};                    // Remove d_i from D.
6:    End
7:  End
8:  Return D;
```

Algorithm 2: Example custom algorithm made by a user.

Input: data point *d* from the set of data.
Output: true if the error has code ATB3101 and was sent by a VIRM, else false.
Description: boolean indicating if data point should be filtered kept or not (true=keep, false=filter out).

```
1:  If d.getName() ≠ "ATB3101" then     // Filter out all error codes that are not ATB3101.
2:    Return false;                     // 'false' means filter out.
3:  Else
4:    Return d.getTrainType() = "VIRM"; // Only keep trains with the type 'VIRM'.
5:  End
```

Both of the implementations can query the data and save these queried results. These results can be shared by the users on the database itself, or possibly by email using a subscriber system. The program could then send an analysis report to the subscribers of the query. Daily queries (i.e. queries that are executed daily) can have a fixed subscriber list and a fixed configuration of filters. These can be saved in a settings file.

Conclusion

As a team, we chose to decide the query search alternative based on the advantages and drawbacks of both implementations. The implementation of the script search requires all users to be able to program in Python or at the very least understand Python. We can, however, not guarantee that all users are able to do so. Also, the chances of getting unexpected results increase with the use of script searches. For example, a user can make an error in a script, which leads to an infinite loop. It could be the case that the user waits for hours on a result and expects the program to give a result. The program, however, never gives a result based on the fact that it is still searching for the impossible.

With query search, the program is easier to understand because of multiple reasons. First, no upfront knowledge of a language is needed apart from the working of each of the operators (see definition). Also, current applications in the NS environment also use a similar query search. The RTMO tool uses a query based search for business rules. Furthermore, the problem of infinite loops is solved by restricting the users to only search using the given operators. These operators can be translated to SQL queries. These SQL queries will then be evaluated, where unsolvable queries are detected by the SQL evaluator and will get an empty result.

The client asked for a query option for known trends, to verify that the trends can be traced back in the data. This option, however, will be a low priority for our application. Tracing back such trends can be done with external scripts if the option is not added. It is more important to have the option to query the trends in the first place.

Multi user context

As more people will want to make an analysis at the same time, there should be some solution to make this possible. In the current NS environment, each developer has its own partition in the cluster. For this application, there are two options.

First, create a multi-user context within the program, where each of the users gets assigned its own partition within the application. The application then has to prioritize analyses per person and parallelize different analyses. The advantage is that users do not require all permissions that the program is required to have, as only one central permission is needed. However, it introduces an extra complexity, which is not entirely in the interest of this project.

Second, the NS environment can schedule multiple instances of the application in the cluster. Here, each user of the application uses its own instance of the application. The advantage of this is that the NS server does the job prioritization, parallelization, and scheduling, which makes our application less complex and gives us the possibility to keep our focus on the trend finding. The drawback is that the users need the required permissions for the program, in order to do an analysis.

Conclusion

As a team, we chose the scheduling using the NS environment, where multiple instances of the application are called in the cluster, as it makes sure we can keep our focus on what the project is mostly about: finding trends. In the future, it could still be possible to add a multi-user context by changing the job scheduling of the application. Each analysis could get its own thread assigned by the application. Another problem that is avoided in this way is the danger of unauthorized personnel using the application, as we do not have the required knowledge of the permission system within the NS.

Scope

In this chapter, functionalities will be defined as ‘in scope’, or ‘out of scope’. Functionalities in scope, we definitely want to tackle in this project. Functionalities ‘out of scope’ will not be implemented in the current project.

In scope

Analyze diagnostic data of the past and indicate location-specific errors with specified settings with the push of a button:
There is a big difference between the analysis of data that is already saved in the database and data that flows in each second. By analyzing only the data that is already in the database within a timespan x to y (where x < y), then a near real-time analysis can be done by taking, for example, a timespan of y equal to the current time and x one minute into the past.

This would allow the reliability engineer to gain insight into location-specific error trends and do research on them. After this, the RE can decide to for example modify each train or get in contact with ProRail to point them to the location which causes errors. By analyzing the data of the past, trends can be found and used for future prevention. As mentioned, a near real-time analysis can be simulated by picking a close to current time analysis. In this way, the application can come close to a monitoring system but still functions as an analytics program. The program is, therefore, part of RTMA and not RTMO. This was also the desired functionality of the client.

Autonomous trend suggestions that indicate atomic/simple statistical findings:
There is a lot of available data and a lot of possible correlations to evaluate. In order to allow the users to focus on finding strong correlations, it is useful to indicate the current statistical outliers. This will be done by the autonomous trend suggestions. By this, an RE could be pointed only at the errors that have the highest likelihood of being a trend and is, therefore, a possible location-specific error. This aligns with the design goal of *quick insight into data*. The simple statistical findings can be placed next to some more complex analysis to gain a deeper insight of the actual trends found in the data. It is, for example, important to know how many times errors occurred in order to know their possible impact.

Find location-specific trends on the basis of diagnostic codes generated by trains:
The trends will be found on the premise of diagnostic codes because this is a level of abstraction that allows us to observe trends without focussing on the technical analysis of the trends too much. This is in line with the higher-level analysis required by the RE and fleet analyst. The trends will be focused on their location of occurrence, as the focus should not lie on general occurrences of trends. The assignment of the client was specifically focused on location-specific trends.

Order error codes per location dependent on the time of generation:
When analyzing the data in order to see trends arise, it is important to be able to only take error codes of one location. When finding that multiple instances of the same problem occur on the same location frequently, there is a high probability that the trains and the location do not go well together. Therefore, with this idea, a lot of information can be found in a short period of time. This aligns with the design goal of quick insight into the data.

View all found location-specific errors on a map, with approximated indications:
Having the location-specific errors visible on a map will give an overview of the location of the errors. This may be useful for figuring out connections between errors. A meeting with the end user has pointed out that they will probably not take the effort to look into a table, because they experienced that table based ways of displaying information tend to be overwhelming. Additionally, both the RE and FA thought that a map would provide faster insight as to where the trend occurs because a map works better with their intuition. This, therefore, aligns with the design goal of quick insight into the data.

Displaying additional information alongside the map:
In order to gain insight into the data, additional data needs to be presented in an interface. The additional data will comprise of what the end-users indicate they need. This data will be displayed using different levels of abstraction/summarization. The display of this additional data/information aligns with the design goal of quick insight into the data. Displaying this data will give the client the opportunity to check for the validity of the trends found. This additional data is also required if a trend is valid. When a trend is found and identified as a real problem, then the additional data needs to be sent to ProRail in order to allow them to fix the problem at the specified location. The information can help them to identify the problem at the location.

Filter out false positives generated by service tracks:
Trains generate a lot of diagnostic codes when they are under maintenance. If these diagnostic codes are not filtered out, the application will certainly show a lot of error trends at for example service locations. The end user is not interested in this data.

Noted should be, however, that recognizing diagnostic data generated by trains under maintenance is an entire research project on its own. NS employees have struggled with this problem earlier. Therefore, we will only implement this to the extent possible by implementing techniques given to us by the client. The need for this filter becomes obvious if imagined that if a trend is found there is a chance that someone of ProRail will go the location of the trend to fix the railway. If there is no problem to be found, then the money spent on the technician is wasted. Also, users could get the idea the application does not work properly and lose faith in the tool.

Filter out noise caused by quick consecutive error codes:
Multiple errors can be consecutively sent by a train, even though they are caused by the same defect and are basically indicating the same problem. This could be caused by multiple reasons but the information that needs to be retrieved from this is that there is only one problem causing the errors. This means that all the same errors in the same short timespan (in seconds) are squashed to one single error.

It could be the case that if no such filter is implemented, that some errors appear to be more severe due to its frequency. It would then be possible that one train sent most of the errors, of which all were caused by the same problem. By using a filter, only the problems are compared for their frequency of occurring. It might still be possible that not all of the double errors can be filtered out, as the line between sent double and two problems is blurry. It could be the case that a problem happened multiple times in a short time period without being a duplicate.

Display the severity of trends retrieved from the error severity (class):
This allows the RE to determine whether trends are worth looking at: is addressing the trend worth a business case? It can also be useful for the fleet analyst, in order to determine what priority things have in what is going on at that moment. Furthermore, it is likely a relatively easy functionality to implement,

because diagnostic codes have a severity level themselves. Giving the users an indication of what the priority is, aligns with the design goal of quick insight into the data.

Schedule the analysis to run automatically at a specified interval:
The RE wants to have a report/analysis every given interval, based on his specific use case of the system. By automatically generating a result at the start of a given period, the RE does not have to start the analysis and does not have to wait for it to finish. This improves the product, according to the design goal of manageability, where the user does not have to take additional actions to still get a result each period (e.g. day). It was desired by the client to get notified of results, to minimize their actions required.

Not in scope

Other train series than the VIRM:
Each train has a different data structure. Due to the time constraint of ten weeks on the project, it is better to focus on one type of train and save time. The VIRM is the train series which has the most trains with an active data connection driving around and the employees currently working at the NS know the data this train type generates the best. The focus will be on the VIRM, in order to be able to use the knowledge of the employees. It was also the client's desire to focus on one train only to save time and spend the most time possible on finding the actual trends.

To still make it possible for the future to implement different train series, we will make sure that the data tables and their respective columns are not hardcoded into our application. In this way, a programmer could simply add more train series by indicating the required tables and columns for the new train serie. One should notice that location-specific errors can be train serie dependent. This means that, for example, there might be a situation in which the VIRM train serie has no troubles with a non-leveled rail, whilst the SLT does have troubles. Therefore, a distinct analysis between these different train types might be needed. This could in the implementation of this application be made possible by adapting the current train serie filter to the desired train serie. In other words, in the current version, the data will be filtered for VIRM trains. By adapting this filter to different train type, like the SLT, then the data can focus on this train specifically. Also should be noticed that location-specific errors can be train serie independent and therefore the filter for train series must be disabled. This can be done by simply disabling this filter. The application will be written as such, that the filter can be empty (i.e. what goes in, comes out).

Editing existing data in the database:
Editing the database requires precautionary measurements to prevent dangerous changes to the database from happening. For our application, edits to existing data can only be done if the data originated from our application. Meaning that only result data that is written to the database by our application can be affected and data generated by other applications or by the trains will be treated as read-only. This also means that appends can be done to create temporary or permanent files on the database. Temporary files can be cached to generate faster results, while permanent files can be the results of analyses.

Creating new error codes with the application:
Error codes and diagnostic codes are created by the train manufactures. There are options for creating new error codes but knowledge on the necessity of creating those is required. This knowledge is missing. Also, the trends found by the application can be displayed in the application itself and do not require to be defined as a new error code. Note that this does not say that result can be saved to the database, it only implies that existent data generated by trains is not edited.

To still make the application useful in the future, the ability to create these new codes should be

possible. Our program will allow an addition of this feature by not having any error codes hardcoded. This way, the analysis will not fail due to the problem not recognizing an error code.

Use raw sensor data (like temperature) for detecting error trends:
In order to be able to make such assumptions on the basis of the data, major knowledge about the raw sensor data is required. We do not have the required knowledge of the raw data or time to acquire such knowledge, to be able to implement this feature. The diagnostic codes the train generates when specific sensor values are beyond certain values will be used. In this way, we can use the knowledge of the train manufacturer to detect interesting sensor data.

For future expansion, it might be a good idea to have these raw sensor data available as extra information with which a correlation between the sensor data and error codes can be found. This correlation can be found using the earlier mentioned Pattern Mining method. For each sensor, the application would then find the k-most frequent items per error code. This method would be very expensive to execute and, therefore, would not be possible/desirable with the current implementation. The client suggested to ignore sensor data, also because of the complexity of the sensor data and the required knowledge for them. As a team, we also thought it would be better to first focus on the error codes themselves before diving into the more complex trends. We, therefore, ignore the raw sensor data and also do not add possibilities for future implementation of raw sensor data.

Arguably, it might even be more desirable to implement this Pattern Mining for raw sensor data inside the train computers, as these trends may even be train dependent (not even train serie dependent). This, however, does not directly enable a comparison of raw sensor data trends between trains (but does not make it impossible; findings of a train can be saved and compared externally). Further thoughts and research are interesting for the global vision of NS but lie outside of this research and should for now be ignored.

Analyze real-time diagnostic data and indicate on what conditions error trends are found:
Real-time diagnostic data can be beneficial for the errors that need fast responses to prevent huge damages. It is, however, a big feature that requires a lot of time and attention. This may not be worth it.

After discussing this approach, both the RE and FA indicated that day-by-day analysis would be a good place to start and that long-term analysis was more important. A day-by-day analysis is considered as good enough because there's a delay in finding the problem and getting the team on the work floor to address the problem. As such, real-time error detection will not be beneficial.
Additionally, a system for detecting long-term problems is currently not present in the NS environment, whereas the RTMO environment provides a kind of up to date analysis. As such the offline long-term approach is more valuable.

Literature Research

Research questions

In the literature research section, an answer to the following questions is given to the extent possible, in order to establish specific design-choices in the final approach and to help formulate the functionality in our MoSCoW list and the minimal viable product.

(Q1) What environments are currently being used by the NS, and how are they used to find trends?

(Q2) Which visualization techniques for displaying trends on a map exist?

(Q3) What frameworks exist for fetching big amounts of data from the database, with a small delay?
(Q3.1) How can that framework be used in order to maintain our *performance* design goal?

(Q4) How can trends be established per location?
(Q4.1) How can trends be established autonomously?

(Q5) How can the data be filtered for noise and false positives?

Solutions

For the different solutions, we consider possible answers to the research questions, which are referenced accordingly. The answers to the research questions, which are based on the frameworks/techniques and algorithms analysis below, determine what a specific approach looks like. All questions mentioned in the section above will be answered in this section.,

(Q1) Environments

Q1 is already answered in the section ‘Current System’.

(Q2) Visualization:

Microsoft Power BI

As stated in the Current System chapter, the NS now primarily uses Microsoft Power BI for data visualization for reports and dashboards. Therefore, this option should at least be considered. An example of how Power BI can be used to display location-specific data can be seen in figure 1. In this figure, points are plotted on the map, where a box next to the track reported an error. Some data filters can be applied within the shown interface, such as selecting a timestamp or site name.

Power BI is a suite of analytics tools to analyze data and share insights in an attractive manner. The data behind the dashboard can also easily be looked into from the interface. It features data imports from hundreds of data sources, including Excel[15] sheets and databases such as the Haas HIVE[16] database, which is used by the NS to store RTM data. It provides a lot of built-in visuals to give insight into the data including tables, graphs, charts, and maps.

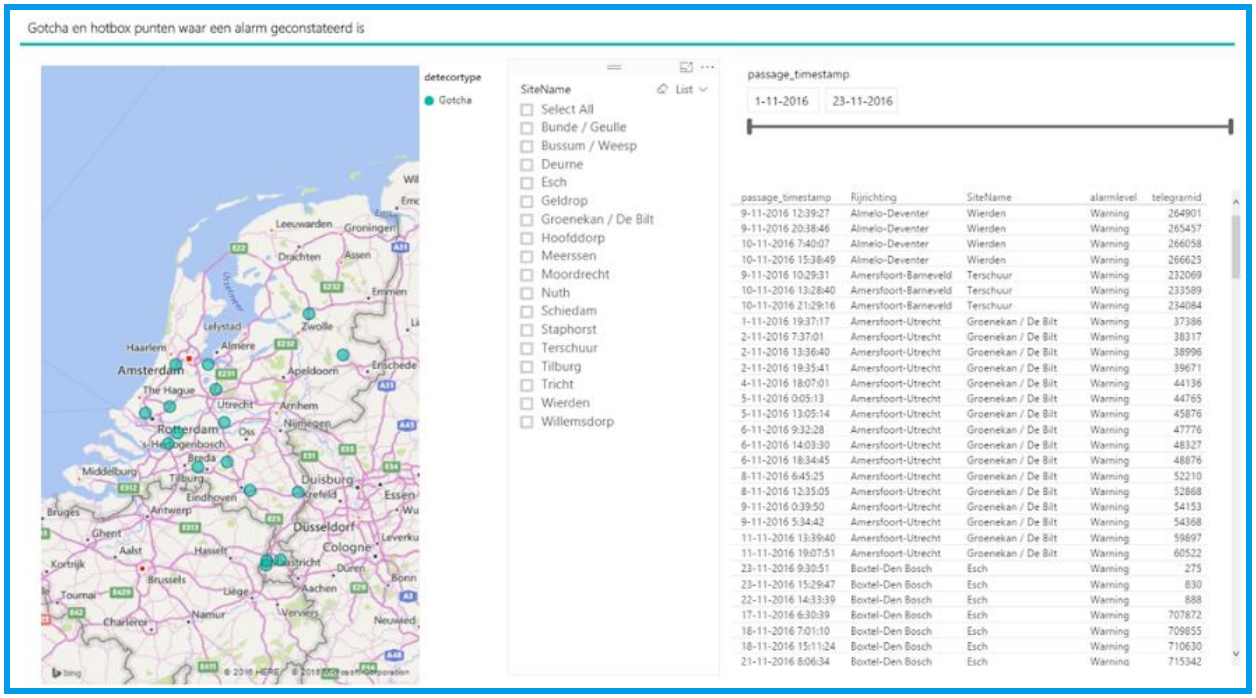

Figure 1: Example of Power BI interface used by NS

Other data visualization tools

Other visualization tools which will also satisfy our needs exist, such as Tableau Desktop[17] or Watson Analytics[18]. However, these need (expensive) commercial licenses to use them. The NS is, understandably, not willing to buy these licenses if they have another software solution themselves right now. Besides the extra costs, these tools do not add any functionality on top of Power Bi which would be needed for the application. Power BI has enough functionalities for our application.

Creating our own interface

It is also a possibility to create a custom visualization interface using frameworks. The best solution, in that case, would be to build a web interface displaying the found trends. That way, existing visualization frameworks which can plot data on maps can be used such as D3.js[19], Leaflet[20], or geoplotlib[21]. This is

[15] *https://products.office.com/nl-nl/excel*
[16] *http://hive.apache.org/*
[17] *https://www.tableau.com/products/desktop*
[18] *https://www.ibm.com/watson-analytics*
[19] *https://d3js.org/*

however not our preference, because we did a similar thing in the Context Project course of our Bachelor. It is also more interesting for the client if we put more effort in finding trends rather than displaying trends. Furthermore, it is desired by the client to have all tools in the same interface. Because of these reasons, this solution will not be chosen.

(Q3) Database:

Big Data Frameworks

Jupyter is an open source notebook the analysts at the NS use to write code, analyze datasets and build various machine learning models. Jupyter is integrated into the NS Big Data domain and supports various programming languages, including Python. The NS Big Data domain includes the following frameworks: Apache Beeswax (Hive UI), Apache Hive, Apache Pig, HCatalog, Apache Spark.

All previously mentioned frameworks are related to the use of the MapReduce paradigm (Dean, Ghemawat. 2008). This makes all frameworks suitable achieving the *performance* design goal. The former four are integrated within the HUE environment and make use of Apache Hadoop as the underlying framework. They also make use of the HDFS as their filesystem and use MapReduce to obtain the results for queries. Apache Spark, in its PySpark form, is accessed through Jupyter, needs to use a distributed file system, and uses its own implementation for cluster computations, different than MapReduce. Because of the latter PySpark generally is 10-100 times faster than the MapReduce implementation Hadoop uses (Williams, 2017; The Apache Software Foundation, 2018). Regarding the level of abstraction, the former four are generally higher level programming frameworks.

In choosing the framework to use for the final product, the *manageability* design goal appears to be the most significant, which would lead to the usage of PySpark as primary Big Data framework. Manageability is warranted by choosing for PySpark as the current analysts at the NS use Python and this framework. The *performance* design goal is also warranted this way, as Spark has better performance than any of the Hadoop alternatives. Furthermore, besides performance and manageability Hadoop oriented frameworks and Spark based frameworks appear to be virtually similar regarding functionality.

(Q4) Trend detection:

In the following section different possible algorithms are explained and checked for their usability in the application.

Statistical Analysis

Considering the repetition of events as a trend in the data, statistical analysis would be able to provide us with such trends by e.g. counting how often an error code occurs. The available data can be searched and data about various features can be collected in order to provide us with information about trends in the dataset.

[20] *http://leafletjs.com/*
[21] *https://github.com/andrea-cuttone/geoplotlib*

As a basis for finding statistical trends in the data, Simple Statistical Trend (SST) analysis would be a place to start. Simple means that the statistical information relies on one atomic feature, e.g. diagnostic_report_code value, in combination with a time frame and location. As an example, given the diagnostic_report_code (error code) 'ATB032' and a time frame '1 year ago - 2 weeks ago' the (hypothetical) Statistical Analysis Engine (SAE) will count the number of times ATB032 occurred for each location and return the locations in order of frequency. Various relevant atomic features could be used in order to detect a number of SSTs.

An advantage of this way of detecting trends is that it provides insight as to why trends are indicated as trends. That is, the count of certain features can be displayed to the user. Depending on the implementation of the SAE, further useful information such as histograms can also be given using Power BI. Such information can be of good use for the technical analysts. In its simple form, it is an easy, yet powerful system to build.

A way to extend this SST system could be sequential pattern mining for item sets. This could be useful for a type of causal analysis of trends and could also enable the system to indicate complex trends for certain locations, as it could indicate what item-set frequently occurred.

Sequential Pattern Mining

Sequential pattern mining concerns finding statistically relevant patterns between different data examples, where the values are sequentially delivered (Mabroukeh & Ezeife, 2010). In this technique, there is a distinction between string mining and itemset mining. The first focuses on long strings of characters in which one can search for a pattern within the string. The second focuses on discovering frequent itemsets and the order in which they appear. The latter seems to be more applicable in the situation of the trend system, where errors sent by a single train can be seen as a single itemset.

An example in which you would want to use itemset mining, as described by Han, Cheng, Xin & Yan (2007), is when you want to find trends in shopping carts, where the customer who buys milk is more likely to also buy bread. Here the combination of milk and bread is a frequent itemset. When a customer buys a cell phone, then a charger and then a memory card, then this is a sequential frequent itemset (Han et al., 2007). The latter is what can be applied in the NS application. If first error A is thrown, then error B and then error C and this seems to happen more often, then there is a sequential frequent itemset possible for these error combinations. If this frequent itemset is confirmed to exist, then the occurrence of an error A can guarantee with a probability P that errors B and C will follow.

Methodologies

For the sequential pattern mining, we cover two different methodologies: Apriori and FP-growth. To find the most suitable algorithm, we first have to distinguish the different methods from each other.

Apriori works by observing a downward closure property, called apriori, among the most frequent k-itemsets (Agrawal and Srikant, 1994). The idea behind this method is that an itemset can only be frequent if its sub-itemsets are also frequent. This means that the database can be scanned for frequent 1-itemsets, and using these itemsets generate candidate frequent 2-itemsets and check and obtain by scanning the database again if these itemsets are frequent. This process can be iterated until there are no combinations of itemsets are possible with size *k*. The Apriori technique can be used to find frequently combined error

codes but may generate performance issues when executed on the full database with a range of, for example, a year. In such a case the number of combinations can become significantly large. In such a case there are for each *k,* $(n-1)^k/2$ possible combinations, where *n* is the number of unique values. Note that there are no double values in sets and, therefore, no combinations with two of the same elements are possible and only half of the combinations are unique.

The second methodology, FP-growth, differs from the Apriori method in the second step. Unlike Apriori, FP-growth uses a suffix tree structure to encode the objects in the dataset, without explicitly generating a candidate set (Han, Pei & Yin, 2000). Normally, generating a candidate set can be an expensive task. After the second step, frequent itemsets can be obtained from the FP-tree.
The Spark library has an implementation of the FP-growth algorithm[22], with parallel capabilities, proposed by Li, Wang, Zhang, Zhang & Chang (2008). This library makes sure that when the algorithm is executed on a cluster, it will be more efficiently solved. The application will eventually run on the clusters of the NS. As the FP-growth algorithm can be parallelized, as well as it does not need an expensive candidate selection, it is more suitable than Apriori for our application. In this way, faster results can be obtained.

Defining sets

All of the definitions above assume that there are sets of items. In the case of the database of the NS, there are tables with errors generated by the trains. These errors should, therefore, be clustered in some way to form sets, which can then be checked for the presence of frequently occurring combinations of items. For creating these sets, there are different options. Itemsets could be defined as errors in the same timetable (i.e. the same train between stations A and B), errors generated on the same railway section, or errors generated in the same timespan of *t* seconds (where *t* can for now be an arbitrary integer). The second option, errors generated in the same railway section, seems to be more relevant. Errors generated in the same timetable are too broad to be pointed as location-specific, they would be more likely to be timetable specific or specific on the train. The size of the sets of errors generated in the same timespan of *t* seconds would depend on the speed of the train (as faster traveling trains cover a longer distance in the same time) and the question remains for choosing the right value of *t*. Railway sections enable the application to focus on trends only specific to a location (railway sections) but at the same time does not require it to learn a value *t*. Learning this *t* would require validation data, which is currently not available (yet). Still, this *t* could be interesting for future work, when this application has generated enough data to function as validation data, where *t* can be minimized as such that the location-specific trends can be narrowed down to smaller sections of the railway.
For now, the best choice is to cluster all items (errors) in the same set based on the railway section they were sent on.

Sequential Pattern Mining seems to be useful for our application, as it enables the application to find combinations of errors that are frequent dependent on their location. It could be the case that some combinations are logical to be frequent, these can be filtered out by comparing the frequent itemsets of different locations. Both the Reliability Engineer or Fleet Analyst can decide if these combinations are logical. Sometimes they will have to check this with their third parties to be sure. If, for example, most locations seem to have the same frequent itemset, then this itemset is likely not to be a location-specific

[22] *https://spark.apache.org/docs/2.3.0/mllib-frequent-pattern-mining.html*

trend. There is a high likelihood that it is a general trend. Still, these kind of trends can be useful, as general trends are also important to the interest of the NS. Frequent itemsets can after being identified, be coupled with optional delays of the train. If there was a delay, it might be caused by the occurrence of the frequent itemset in the timetable. Coupling with the delay is only possible when the delay is known to the system at the time of the analysis. The drawback of the use of Sequential Pattern Mining is that the trends must be frequent and thus less frequent combinations might be missed. To find these more infrequent combinations, different approaches might be desired.

Neural Networks

Trends may be identified/classified by the comparison of many factors, neural networks offer a way of classifying multidimensional data by means of relevant feature extraction, on the basis of seeing examples, in the case of supervised learning (Ojha, 2017). NS has an extensive dataset containing many meaningful features (section: available data), which is useful for machine learning tasks in general. In this dataset, there are no examples of trends, which makes the application of supervised neural networks challenging. Trends have an additional feature, which is their chronological dependency. Recurrent neural networks (RNNs) have the capability to capture such (long-term) chronological dependencies (Boden, 2002) which can be used for time series prediction (Cai, 2007).

Considering the purpose of our system, time series prediction is not something that is needed. The purpose is to find existing trends, that already happened, not to predict trends that may happen. However, theoretically one could use RNNs to assign a severity to a currently detected trend by estimating how likely the observed pattern results in more error codes. This would add some amount of value to indicating the severity of existing trends, especially those that recently occurred as the cause of the trend is more likely to be present still and they have a more likely chance to reoccur. Given this a priori knowledge of recently detected trends having a higher chance of happening again the value of RNNs once again limited. Predicting how likely a trend is to reoccur would focus on the predictive power of RNNs, which is not in line with the scope of this product, and add limited value to indicating longer-term trends, especially considering the time needed to implement and train RNNs. Furthermore, assuming RNNs would be able to classify trends, they could fall short in being able to explain the reasoning for indicating a trend (Goodman, 2017) and what factors lead to calling it a trend. This explanation is important for analysts to be able to address (the cause of) the trend.

Primarily due to the scope of this project, using RNNs is not recommended, however there is potential when looking at the type of product described in the *vision* section above. Neural networks could be used to classify raw sensor data as diagnostic error codes. This may be done by using technical analysts to train a neural network which selects key features in the NS dataset in order to determine accurate boundary conditions for specific diagnostic error codes. This could save the technical analysts the time and effort of trying to determine what boundary conditions lead to what diagnostic error codes. As additional functionality outside of the current scope, the RNNs ability to forecast trends could well be used in order to predict what problems may pop up in the near future, such that resources can be allocated more efficiently and punctuality can be increased due to faster reaction times.

(Q5) Data filtering:

When visualizing the current errors, high frequencies are found in Onnen and Maastricht, among others. The reason for this is that these two cities are places where trains can go for maintenance (“Locaties langs het spoor”, n.d.). In these service locations, the mechanics need to test whether certain sensors are working and will thus trigger all kinds of errors. It is, however, not visible in the error codes whether these errors are made by the mechanics in the service locations or that these are legitimate errors. This makes it thus very hard to filter these false positives out of the database. The conditions to know whether the errors are legitimate are still unknown for NS. As mentioned in the scope section, finding out these conditions is an entire research project on its own. Only the conditions already given by NS will be used to filter. The quickest way of actually filtering these will be using the SQL WHERE query (“SQL for Beginners: Filtering Your Data”, n.d.).

Another way that noise can occur is when the trains send out quick consecutive errors for the same problem. This noise needs to be filtered by merging records that represent the same problem into one. What needs to be determined is how much time or traveled distance needs to be in between two of the same error messages in order for it to be displayed as a separate error. For determining this, empirical research is needed. The exact duration has to be found when the program is in use. When using a long duration, multiple problems might be merged into one, which is a bad thing. However, when using a short duration, too many error codes may still be displayed.

Conclusion to the research

For the interface, Power BI will be used. For data-storage and data processing we will use Pyspark in the Jupyter environment based on a Hive database. For autonomous trends, we use statistical analysis for finding the STTs in the data and FP-growth Sequential Pattern Mining to find frequent occurring combinations of errors per railway section. For data filtering, the conditions for filtering maintenance errors will be executed to the extent possible, only with what NS has provided us with. For filtering the multiple errors a train sends out for the same problem, empirical research is needed to determine the time or distance traveled between errors to merge them into one.

Use cases

- As a user, I want to activate the analysis when I want to, in order to make an analysis when I want it and not put too much load on the servers.
- As an FA, I want to get a report of a daily analysis sent to me each day, in order to always be up-to-date on the newest trends.
- As an RE, I want to get a report of monthly analysis sent to me each month, in order to gain insight into current trends without having to think of activating the system.
- As a user, I want to change the timeframe for analysis (i.e. analyze from time *x* until time *y*), in order to see trends of different time spans than daily or monthly.
- As a user, I want to be able to filter the results of an analysis per location, in order to be able to gain insight into location-specific information.
- As a user, I want to be able to filter the results of an analysis, in order to be able to gain insight into the information I think to be relevant.
- As a user, I want to have trends that can be found in the data autonomously suggested to me, in order to gain insight into a reasonable amount of trends without having to scan through the data manually.
- As a user, I want to be able to find trends based on the frequency of errors occurring together, in order to find which errors are correlated to each other.
- As a user, I want to filter data from service tracks signals, in order to make my data more accurate for the analysis.
- As a user, I want to filter data from repeating errors, in order to make my data more accurate for the analysis.
- As a user, I want to see the trends visible on a map, in order to get a quick insight on the location of the trends.
- As a user, I want to gain insight into more specific information related to the found trends, in order to verify the validity of the trends and be able to provide additional information to others (e.g. ProRail).
- As a user, I want to click the trends visible on a map, in order to see the additional information about the trend.
- As a user, I want to have my indicators on a map as accurate as possible, in order to send the GPS coordinates of my trends as accurate as possible to those who have to fix the problem on the pointed location (e.g. ProRail).
- As a user, I want to have the application integrated into the software I am familiar with: Power BI, in order to have all my analysis tools in the same environment and visualize the results of my tools on the same map.
- As a user, I want to get an indication of how severe a detected trend is based on the class of errors, in order to be able to prioritize my work better.

- As a user, I want to focus my application first on the VIRM train type, in order to verify that there are location-specific trends to be found which can later be ported to different train types.
- As a developer, I want to keep in mind that there are more train types (and therefore not hardcode any train specific elements in my program), in order to make sure that the program could later be extended to support multiple train types.
- As a developer, I want to save/cache the outcome of analyses, in order to prevent double analyses from happening and increase the performance of the application.
- As a developer, I want my application to be written in Python in combination with PySpark, in order to use the same language as the people that will maintain the program and, therefore, increase the maintainability.
- As a developer, I want a minimal tested line coverage of 75%, in order to have a slight assurance of my application to be of quality.
- As a developer, I want to use version control, in order to be able to return to previous versions when needed.
- As a developer, I want my code to be documented in English, in order to enable any developer to maintain the code without requiring the code maintainer to be Dutch.

Requirements

In this section, requirements will be explained and ranked according to the MoSCoW[23] methodology. Using this methodology the software engineering tasks can clearly be prioritized and communicated to the client, thereby also serving as an implicit contract.

Must

- The user shall be able to click a button to start the analysis.
- The user shall be able to analyze the VIRM train type.
- The user shall be able to adjust the analysis time frame.
- The user shall be able to filter the results based on a specified location.
- The user shall be provided trends based on the number of times an error reoccurs.
- The user shall be able to see how many different trains a trend occurred for a given location.
- The user shall be able to see the timeframe in which the trend occurred.
- The user shall be provided trends based on the frequency of errors occurring together (k-frequent itemsets) in the trains on the same location.
- The user shall be able to see the locations of found trends on a map based on GPS coordinates.
- The user shall be able to gather more information regarding a trend when clicking on the trend on the map.
- The application shall be able to point diagnostic error codes to a railway section between two time table points where they were sent from.
- The user shall be able to view the generated results in Power BI.
- The user shall be able to receive an automatically scheduled analysis report of the last 24 hours, that will be generated daily.

[23] *https://www.agilebusiness.org/content/moscow-prioritisation*

- The application shall use the existing service track filter to filter out diagnostic codes send from service tracks.
- The application shall have a filter to filter out repeated diagnostic codes repeatedly send which indicate the same problem.
- The user shall be able to terminate an analysis process by pressing a terminate button.
- The results of the analysis shall be stored in a database in order to be able to display them using Power BI.

Should

- The user shall be able to receive an automatically scheduled analysis report that summarizes the month, that will be generated monthly.
- The user shall be able to open a view in Power BI displaying individual errors that lead to the trend.
- The user shall be able to share last found results through email.
- The user shall be able to filter on the class of the errors.
- The user shall be able to filter on the rolling stock names of errors.
- The user shall be able to filter on the diagnostic report code of errors.
- The user shall be able to filter on the rolling stock number of errors.
- The user shall be able to filter on the frequency of errors following a specific trend on a specific location, i.e. show all trends that re-occurred more than *x* times.
- The user shall be displayed the frequency relative to the amount of train traffic for a given location.

Could

- The user shall be able to receive an *automatically* scheduled analysis report of a *user-specified* time interval.
- The user shall be able to see the severity of trends calculated based on the severity of the error classes in the trends.
- The user shall be able to filter on the severity of errors.
- The user shall be able to disable repeated diagnostic error filtering.
- The user shall be provided with a histogram displaying the number of occurrences of diagnostic error codes over time for a specific location.
- The user shall be able to see which diagnostic codes frequently occur for a location in a chart.
- The user shall be able to find trends based on Machine Learning methods.
- The application shall smoothen GPS coordinates based on a map matching algorithm.
- The user shall be able to export the result to a CSV to enable the user to open the data from Excel.
- The user shall be able to add his own hypothetic trends and shall be able to verify the hypothesis with the data.
- The user shall be displayed an error indicating that no recent data was added.
- The user shall be displayed an error indicating that no recent trends were found.
- The user shall be displayed an error indicating that no connection with the database could be established.
- The user shall be displayed an error indicating that the data used for analysis has changed format in a way that leads to the system not working.

Won't

- The user shall not be able to edit existing data from the application.

- The user shall not be able to get real-time analysis based on the incoming data.
- The user shall not be able to analyze different train types.
- The user shall not be able to use raw sensor data to find trends.
- The user shall not be able to edit existing data generated by trains using the application.

Definition of Done

The minimum viable product is the most pared-down version of the product that can still be released. The vision of the product should be visible and the functionality should have enough value to the NS. The minimal viable product consists out of the musts in the MoSCoW method above. All functionality described in the should and could are optionally done when the time allows it. The won't functionalities are not implemented and may only be included in future recommendations. The definition of done is when we implemented all the functionality in the must section.

Initial Release Plan

We plan to do a small release every week. This initial plan is to give a coarse idea of which features we want to implement in which release.

Week 3: (07-05 - 13-05)

- Get permissions to the required frameworks, databases and libraries.
- Establish version control and scrum board software to use as a team.
- Design high-level architecture and make corresponding UML.
- Establish a connection between PySpark and the NS data warehouse.
- Use PySpark to execute queries to obtain basic location-specific results
- Start sequential pattern mining.

Week 4: *release 1.0*

- Store result fetched with Pyspark in a database.
- Establish a connection between the database and Power BI
- Preliminary design for the GUI in Power BI
- Display some data in Power BI in a table and map.
- Design more detailed architecture and make corresponding UML
- Determine valuable statistics to gather and design how to collect and store these statistics.
- Implement and improve sequential pattern mining algorithm.

Week 5: (21-05 - 27-05)

- Allow the user to interact through Power BI with the analysis.
- Allow the user to click items on a map and show basic additional information.
- Start filtering repeating diagnostic codes and service tracks.
- Establish a system for scheduling an analysis.

Week 6: release 2.0

- Improve filters for diagnostic codes and service tracks.
- Implement a system for displaying information about individual errors that lead to the trend.
- Implement filters on already processed data, as referred to in the *should* section in the MoSCoW
- Implement a system for sharing an analysis.
- Improve the way data is visualized on the map.
- Explore the possibility of ML techniques.

Week 7: (04-06 - 10-06)

- Implement additional visualization techniques, such as histograms/graphs, for the information of individual errors that lead to the trend.
- Extend the valuable statistics with additional valuable statistics.
- Implement system to indicate the severity of a trend.

Week 8: (11-06 - 17-06)

This week will be used as a buffer for the planning and could be used, if time allows, to implement some additional features. For example, the item *explore the possibility of ML techniques* in week 6 could require the existence of such a buffer.

Week 9: release 3.0 (final)

- Optimize the implemented *must* and *should* functionality.
- Finish programming documentation.
- Submit the final code to SIG (22-06-2018).

Week 10: (25-06 - 01-07)

- Finish final report (25-06-2018)
- Prepare BEP presentation.

References

1. Boden, M. (2002). A guide to recurrent neural networks and backpropagation. *the Dallas project*.
2. Cai, X., Zhang, N., Venayagamoorthy, G. K., & Wunsch, D. C. (2007). Time series prediction with recurrent neural networks trained by a hybrid PSO–EA algorithm. *Neurocomputing, 70*(13-15), 2342-2353. doi:10.1016/j.neucom.2005.12.138
3. Dean, J., & Ghemawat, S. (2008). MapReduce. *Communications of the ACM, 51*(1), 107. doi:10.1145/1327452.1327492
4. Goodman, B., Flaxman, S. (2017). European Union Regulations on Algorithmic Decision-Making and a "Right to Explanation". *AI Magazine, 38*(3), 50. doi:10.1609/aimag.v38i3.2741
5. Han, J., Cheng, H., Xin, D. & Yan, X. (2007). Frequent pattern mining: current status and future directions. *Data Mining and Knowledge Discovery, 15*(1), 55-86. doi: 10.1007/s10618-006-0059-1
6. Han, J., Pei, J. & Yin, Y. (2000). Mining Frequent Patterns without Candidate Generation. *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, 1-12. doi: 10.1145/335191.335372
7. Han, J., Wang, Y., Zhang, D., Zhang, M. & Chang, E. Y. (2008). PFP: Parallel FP-Growth for Query Recommendation. *Proceedings of the 2008 ACM Conference on Recommender Systems*, 107-114. doi: 10.1145/1454008.1454027
8. Locaties langs het spoor (n.d.). Retrieved from https://www.ns.nl/over-ns/treinonderhoud/locaties.html
9. Mabroukeh, N. R., Ezeife, C. I. (2010). A taxonomy of sequential pattern mining algorithms. *ACM Computing Surveys, 43*(1), 1-41. doi:10.1145/1824795.1824798
10. Ojha, V. K., Abraham, A., & Snášel, V. (2017). Metaheuristic design of feedforward neural networks: A review of two decades of research. *Engineering Applications of Artificial Intelligence, 60*, 97-116. doi:10.1016/j.engappai.2017.01.013
11. SQL for Beginners: Filtering Your Data. (n.d.). Retrieved from http://www.dnndev.com/learn/guide-legacy/sql-for-beginners/filtering-data
12. The Apache Software Foundation. (2018). Apache Spark™ - Unified Analytics Engine for Big Data. Retrieved May 2, 2018, from https://spark.apache.org/
13. Williams, M. (2017, August 30). Apache Spark vs. MapReduce. Retrieved May 2, 2018, from https://dzone.com/articles/apache-spark-introduction-and-its-comparison-to-ma