2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)

Pitfalls Analyzer: Quality Control for Model-Driven Data Science Pipelines

Gopi Krishnan Rajbahadur
Queen's University
Kingston, Canada
krishnan@cs.queensu.ca

Gustavo Ansaldi Oliva
Queen's University
Kingston, Canada
gustavo@cs.queensu.ca

Ahmed E. Hassan
Queen's University
Kingston, Canada
ahmed@cs.queensu.ca

Juergen Dingel
Queen's University
Kingston, Canada
dingel@cs.queensu.ca

***Abstract*—Data science pipelines are a sequence of data processing steps that aim to derive knowledge and insights from raw data. Data science pipeline tools simplify the creation and automation of data science pipelines by providing reusable building blocks that users can drag and drop into their pipelines. Such a graphical, model-driven approach enables users with limited data science expertise to create complex pipelines. However, recent studies show that there exist several data science pitfalls that can yield spurious results and, consequently, misleading insights. Yet, none of the popular pipeline tools have built-in quality control measures to detect these pitfalls. Therefore, in this paper, we propose an approach called Pitfalls Analyzer to detect common pitfalls in data science pipelines. As a proof-of-concept, we implemented a prototype of the Pitfalls Analyzer for KNIME, which is one of the most popular data science pipeline tools. Our prototype is model-driven, since the detection of pitfalls is accomplished using pipelines that were created with KNIME building blocks. To showcase the effectiveness of our approach, we run our prototype on 11 pipelines that were created by KNIME experts for 3 Internet-of-Things (IoT) projects. The results indicate that our prototype flags all and only those instances of the pitfalls that we were able to flag while manually inspecting the pipelines.**

***Index Terms*—Data science pipelines, model-driven engineering, quality control, data science pitfalls**

I. Introduction

Data science is the science of extracting knowledge and insights from the data. Data science pipelines are sequences of processing and analytic steps that are applied on data to extract such knowledge and insights. These data science pipelines are being widely used in various industries [1, 2, 3, 4] towards diverse use-cases. For instance, GE [5], SAP [6], Bosch [2], and Siemens [7] use a variety of data science pipelines to address problems related to the Internet-of-Things (IoT), Cyber-Physical Systems (CPS) and industrial automation. Particularly in the manufacturing industry, "Industry 4.0" has become a driving force that seeks to unite manufacturing, automation, and rich data sources. These efforts hold data science at their heart to foster better development and automation of IoT and CPS [8, 9]. Rüßmann et al. [9] predict such efforts would be worth close to 39 billion euros in the next 10 years.

However, data science requires deep expertise [10] to be effectively galvanized for real-world applications. Such a requirement have led industries and open source communities to come up with various *data science pipeline tools*, which support the creation and automation of pipelines. Examples of tools include Microsoft Azure Machine Learning Studio [11], IBM ThingWorx [12], Verizon ThingSpace [13], KNIME [14], Weka [15], and RapidMiner Studio [16]. Such pipeline tools leverage domain-specific graphical modeling languages (DSL) to enable the specification of data science pipelines. From a practical perspective, users specify pipelines by interconnecting building-blocks using graphical components that are provided by the tool. An example of a data science pipeline that is designed in KNIME is shown in Figure 3. Once the specification of the pipeline is completed, the pipeline tool automatically generates the low-level code that enables the execution of the pipeline. In other words, pipeline tools transform a user-specified data science pipeline into executable code (Figure 1). Ultimately, these tools enable users with limited data science and programming expertise to implement their data science pipelines with ease [17].

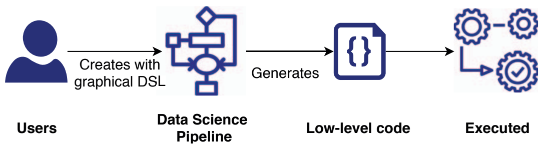


Fig. 1. Code generation in data science pipeline tools.

Unfortunately, data science pipeline tools encapsulate a serious problem. While they enable users with limited knowledge to create and automate pipelines easily, these tools do not offer ways to ensure the quality of the created pipelines and the generated results. Yet, prior research has shown that users with limited knowledge in data science tend to create pipelines that commonly lead to invalid results [17, 18]. Consider an example where a data science pipeline is being created by a non-expert user for forecasting and preventing failure of a pacemaker from associated sensor data. In such a pipeline, if the user fails to deal with correlated metrics in the data, it becomes impossible to ascertain which sensor truly indicates the presence of a problem, as correlation in the data yield spurious interpretations [17, 19]. Therefore, acting based on results of the pipeline could mean fixing the wrong aspect of the pacemaker, which in turn could have serious implications. In fact, similar problems have been encountered in practice by several industries. For instance, Google failed to predict the flu trends correctly, because they did not account for common data science pitfalls [20].

Hence, it is pivotal for expert and non-expert industry practitioners alike to avoid data science pitfalls. Mello et al. [21], Tantithamthavorn and Hassan [17] and Menzies and Shepperd [22] describe various pitfalls that non-expert practitioners stumble into when constructing data science pipelines. However, automating the detection of these pitfalls in data science pipelines is still an open research challenge.

Therefore, inspired by prior studies from the MDE community [23, 24, 25], we propose a novel approach called *Pitfalls Analyzer* that automates the detection of the pitfalls described by [17]. Our approach is model-driven and operates within the context of the domain-specific graphical modelling language employed by the pipeline tools. In this paper, we also describe the implementation details of our proposed approach in KNIME, which is one of the most popular pipeline tools [26]. Finally, we evaluate our implementation and showcase its effectiveness on 11 pipelines that were written by KNIME experts as part of 3 IoT projects.

Paper organization: The remainder of this paper is organized as follows. Section II provides a background on quality control in data science and discusses related work. Section III outlines our proposed approach to detected these pitfalls. Section IV describes the proof-of-concept implementation of our proposed approach in KNIME. Section V evaluates our prototype. Section VI discusses the limitations of both our approach and implementation. Finally, Section VII concludes our study and discusses future work opportunities.

II. Background and Related Work

A. Quality control in data science

Learners are at the heart of data science pipelines. A learner is a statistical or machine learning technique that forms a mathematical representation of the data it analyzes. This representation can be used to perform a variety of analytic tasks. Examples of popular learners include: Linear Regression, Logistic Regression, Decision Trees, and Random Forests. There are various factors that could impact the integrity of the result generated by the learner, which in turn impact the integrity of the results produced by a data science pipeline. For instance, prior research efforts have identified various issues on ensuring the quality of the data that is used in the data science process. For example [27, 28, 29] detail the perils of using data with poor quality as a part of a data science pipeline and focus on ensuring the quality of the collected data. Also, Hall [19] notes that correlated metrics that are present in a datasets could led to spurious learner interpretation. In turn, Cawley and Talbot [30] and Shepperd et al. [31] demonstrate the need for avoiding bias in the learners that are used in the data science pipelines.

While the aforementioned studies focus on individual aspects of the data science pipeline, Mello et al. [21], Menzies and Shepperd [22] and Tantithamthavorn and Hassan [17] describe data science pitfalls, i.e., incorrect ways of using data science processes and techniques. In this paper, we conceive an approach called *Pitfalls Analyzer* to detect the 8 pitfalls described by Tantithamthavorn and Hassan [17]. In the following, we briefly introduce these pitfalls. For a more thorough explanation (including examples and avoidance strategies), please refer to the original paper.

Pitfall P1 – Absence of control variables: Control variables are confounding variables that may affect the outcome of a data science pipeline or the interpretation of such an outcome. For example, it has been shown that large software modules are more likely to have more defects. Therefore, a learner (e.g., Logistic Regression) that aims to predict defects in future versions of a module should include the *size of the module* as a control variable.

Pitfall P2 – Not accounting for the impact of correlated variables on learner interpretation: Prior studies show that not accounting for correlated variables when building a learner can yield misleading insights (e.g., regarding variable importance) [19, 32, 33]. Hence, correlated variables should be accounted for (e.g., removed) from the data prior to building the learner.

Pitfall P3 – Not accounting for the impact of data rebalancing techniques on learner interpretation: Classifier learners (e.g., Decision Trees) tend to perform poorly when the number of data points belonging to each class differ substantially (i.e., when there is *class imbalance*). In this scenario, practitioners may use a family of techniques known as *data rebalancing*, which aims to bring the number of observations in each class closer together (e.g., by creating artificial data for the minority class). While such data rebalancing tends to improve classification performance, Turhan [34] observed that it can also induce a side-effect: bias in learned concepts (i.e., *concept drift*). In other words, the rebalanced dataset can have different statistical properties compared to the original one, thus possibly yielding misleading insights (e.g., regarding variable importance).

Pitfall P4 – Not experimenting with different learners or using default parameter settings for learners: Trying different learners for the same problem is a recommended practice, since there is no *silver bullet* learner, i.e., one that always works best in all possible situations [35]. Moreover, learners have hyper-parameters that need to be tuned in order to maximize their prediction performance [36]. Hence, not experimenting with different learners or using default parameters might lead to suboptimal prediction performance.

Pitfall P5 – Using threshold-dependent performance measures to measure the performance of a learner: The performance of a learner can be measured using threshold-dependent (e.g., F-measure) and threshold-independent (e.g., Area Under the receiving operating characteristic Curve (AUC)) performance measures. The former typically depend on a confusion matrix to be calculated, which in turn depend on a probability threshold to be generated. A threshold-dependent measure can be tricky to interpret, since its value might differ substantially depending on which specific threshold is used. To avoid spurious interpretation, threshold-independent measures should be favoured over threshold-dependent in most situations.

Pitfall P6 – Using 10-fold cross-validation to estimate the performance of a learner: Validation techniques estimate how accurately a learner will perform in practice. *10-fold cross validation* is a popular validation technique in which the original data is randomly partitioned into k equal sized subsamples (folds). Of the k subsamples, a single one is retained as the validation data for testing the model, and the remaining $k-1$ subsamples are merged together and used as training data. The cross-validation process is then repeated k times, with each of the k subsamples used exactly once as the validation data. Despite the popularity of 10-fold cross validation and its simplicity, there exist other validation techniques that are more accurate and stable (e.g., out-of-sample bootstrap) [37]. Hence, usage of 10-fold cross-validation in a pipeline should be dropped in favour of more powerful validation techniques.

Pitfall P7 – Using ANOVA Type-1 when interpreting a learner: The Analysis of Variance (ANOVA) is a statistical test that examines the importance of two or more variables on the outcome of a learner (e.g., defect-proneness). However, ANOVA Type-1 has a generally undesirable characteristic: it is sensitive to the ordering of the variables in the learner specification (e.g., logistic regression model). As a result, interpretation becomes coupled to the learner specification. Therefore, usage of ANOVA Type-1 should be dropped in favour of a more robust test that is insensitive to variable ordering (e.g., ANOVA Type-2).

Pitfall P8 – Interpreting a regression learner using the coefficients of its variables: In a regression learner (e.g., Linear Regression), each dependent variable typically has a coefficient attached to it. It is not uncommon for these variables to be on different scales or assume different ranges. For instance, while *total lines of code* is usually in the thousands (e.g., 50k), the *proportion of highly-active developers* ranges from 0 to 1. Big differences such as these often impact the coefficients of a regression learner. Hence, assuming that the coefficients of a regression learner represent the importance of each dependent variable is always prone to yield misleading conclusions.

B. Quality control in data science pipeline tools

Data science pipeline tools support the creation and automation of data science pipelines in a graphical, model-driven fashion. However, to the best of our knowledge, none of these tools implement any automated mechanism to ensure quality control of the pipelines produced. The closest effort we could identify has been from the KNIME community, which has provided a document [38] with "best-practices for the development, documentation and deployment of KNIME nodes, plug-ins and features." However, such a document does not address data science pitfalls.

C. Code verification

Use of static defect finder tools have been a fundamental part of software quality research [39]. These tools typically traverse program paths extracted from the code in order to find pre-specified problem patterns [39, 40, 41, 42]. A few other research efforts have focused on finding potentially problematic patterns in the source code (a.k.a., code smells) [43, 44].

Furthemore, there have been several efforts in the MDE community to provide error detection and quality control in model-driven solutions, though not catered towards pipeline tools or data science in general. The efforts that are closest to our work are the ones that demonstrate the use and applicability of static analysis for DSLs. Towards that end, Ruiz-Rube et al. [23] propose a model-driven interoperability method that enables standard static analyzer tools to be used to analyze systems that were written using DSLs. Their interoperability method produce several model transformations, to map between the grammar of the DSL and the static analyzer tool's target language. In turn, Heinze et al. [24] take a formalism-based approach to analyze DSLs. They proposed a method that takes a business process and generates a petri-net, which is later used to verify the correctness of the process and flag problems. Saad and Bauer [25] proposed a way to perform static analysis of model-driven DSLs through adoption of data flow-analysis. Their primary goal was to help the designers of the model-driven DSLs add static analysis capabilities to their DSLs. Several other studies [45, 46] also explore methods of enabling static analysis for DSLs. In summary, all these studies transform the model to a common form and check for a problem-causing pattern in the original model-driven DSL. Inspired by these approaches, we propose a novel, graphical, model-driven approach for identifying common data science pitfalls. In the next section, we describe such an approach.

III. Pitfalls Analyzer - Approach

In this section, we detail our approach called *Pitfalls Analyzer*, which aims to detect pitfalls in data science pipelines. Our approach focuses on the pitfalls described by Tantithamthavorn and Hassan [17], which were introduced in Section II-A. An overview of our approach is shown in Figure 2. In summary, we first extract the DAG (Directed Acyclic Graph) of the pipeline, which represents the control-flow of the pipeline. Next, we identify all the learners present in the DAG. Subsequently, we extract the execution sub-graph of each learner, which is made up of all paths leading to and from such a learner. Finally, we determine if any of the pitfalls occur in the execution sub-graph generated for each of the learners by searching for the presence of *anti-patterns*. Finally, we inform users about the presence/absence of pitfalls.

As depicted in Figure 2, our approach comprises four key steps, namely: *DAG extraction*, *Execution sub-graph extraction*, *Pitfalls identification*, and *Pitfalls reporting*. In the following sections, we explain each of these steps in more detail. We use the pipeline shown in Figure 3 as a running example to support the explanation of our approach.

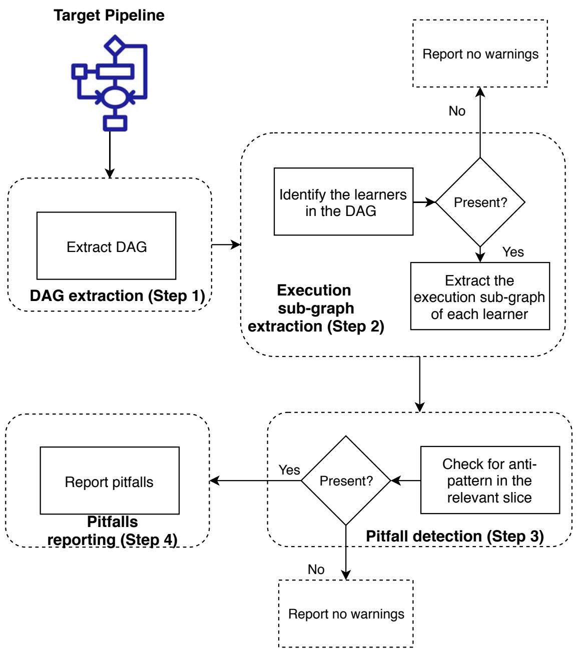

Target Pipeline
Extract DAG
DAG extraction (Step 1)
Identify the learners in the DAG
Present?
No
Report no warnings
Yes
Extract the execution sub-graph of each learner
Execution sub-graph extraction (Step 2)
Check for anti-pattern in the relevant slice
Present?
Yes
Report pitfalls
Pitfalls reporting (Step 4)
Pitfall detection (Step 3)
No
Report no warnings

Fig. 2. An overview of our approach.

A. *DAG Extraction (Step 1)*

A data science pipeline contains nodes and connections. Nodes represent the activities to be performed, while the connections between them dictate the flow of execution. Hence, a pipeline can be seen as a *Directed Acyclic Graph (DAG)* [47], where vertices represent nodes and edges represent connections. In this step, we extract the DAG from the target pipeline. In practice, most tools encode pipelines in metadata files. Therefore, DAG extraction can often be accomplished by parsing such a file. We highlight, however, that the metadata file is tool-specific and its format may vary (e.g., XML, JSON).

Running example: In the example depicted in Figure 3, the extracted DAG would contain all the 19 nodes of the pipeline along with their various interconnections.

B. *Execution sub-graph extraction (Step 2)*

Once the DAG of the pipeline is extracted, we extract the execution sub-graph of each learner node. This is done in two sub-steps as follows:

Identify the learners in the DAG: We identify all learner nodes in the extracted DAG. We do so because all of the pitfalls discussed in Section II are only meaningful if a learner is present in the pipeline. We stop our approach if none exists (check "Report no warnings" at the top right of Figure 2).

Running example: Our example pipeline depicted in Figure 3 contains 3 learner nodes, namely: Decision Tree Learner, Random Forest Learner, and Logistic Regression Learner.

Extract the execution sub-graph of each learner For each learner node, we need to extract its execution sub-graph. We define the execution sub-graph of a learner node as the sub-graph that contains all possible execution paths in the DAG leading from and to such a node. To obtain the execution paths *leading from* the learner node, we simply run a *Depth-First Search* (DFS) starting from the learner node. We call the sub-graph induced by the visited vertices a *forward slice* of the execution sub-graph. Similarly, to compute the execution paths *leading to* the learner node, we simply reverse all edges of the DAG and rerun DFS starting from the learner node. In this case, we call the sub-graph induced by the visited vertices a *backward slice* of the execution sub-graph.

The backward slice of a learner contains the nodes that regard *learner preparation*, since they are always executed before the learner is built. Analogously, the forward slice of a learner contains the nodes that regard *learner interpretation*, since they are executed after the learner is built.

Running example: In Figure 4, we show the backward slice of the execution sub-graph of the three learners (Random Forest, Decision Tree, and Logistic Regression) from the example pipeline depicted in Figure 3.

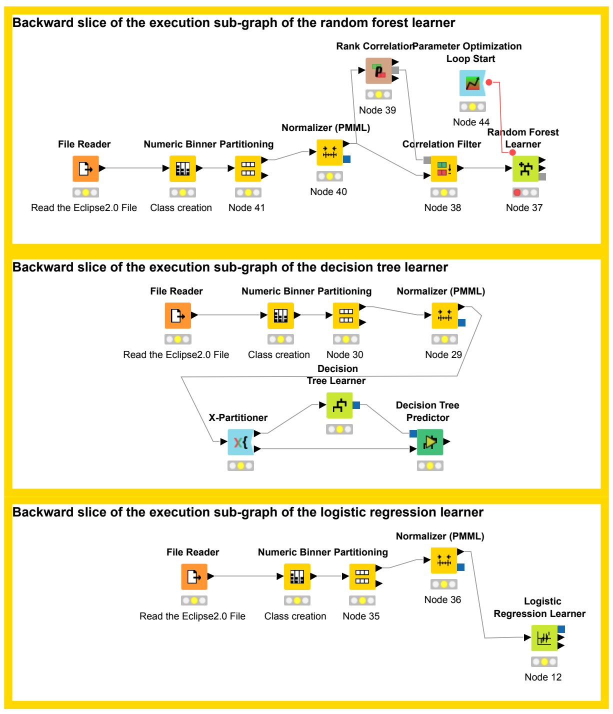


Fig. 4. Backward slices of the execution sub-graph of all the three learners from the example pipeline shown in Figure 3.

C. *Pitfall detection (Step 3)*

We detect pitfalls by searching for *anti-patterns* in the execution sub-graph of each learner node. Table I lists all the anti-patterns that we search for. If a pitfall should be avoided during learner preparation, we search for the anti-pattern in the backward slice of the learner. For instance, in order to avoid pitfall P2 ("not accounting for the impact of

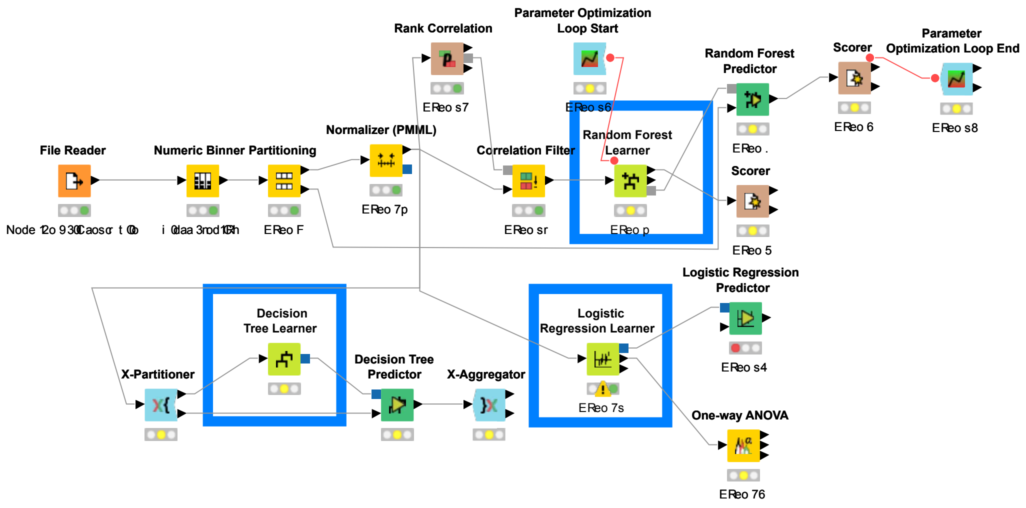


Fig. 3. An example pipeline with several pitfalls (learner nodes are highlighted).

correlated variables on learner interpretation"), the pipeline should contain a node that mitigates (e.g., removes) correlated variables before the learner is built. Hence, the anti-pattern for P2 is tested on the backward slice of learners. Analogously, if a pitfall should be avoided during learner interpretation, we search for the anti-pattern in the forward slice of the learner. In certain cases, searching for an anti-pattern requires inspecting both the backward and the forward slice of a learner (e.g., pitfall P3).

The anti-patterns presented in Table I are defined in terms of the *presence* or *absence* of specific nodes. We call them *anti-pattern nodes*. We search for specific anti-pattern nodes because, unlike traditional programming, in model-driven data science pipelines, tasks are accomplished by using readily available relevant nodes supported by the pipeline tool. In addition, we note that the exact anti-pattern nodes depend on the pipeline tool, as different tools may use different nodes or even different constructs. For instance, in KNIME [14], correlated variables are removed with a "Correlation Filter" node, whereas in RapidMiner [16], the same task is accomplished using a "Remove Correlated Variables" node.

Running example: For instance, to check if the example pipeline shown in Figure 3 runs into the correlation pitfall (P2), we search for a correlation filter node in the backward slice of each learner node. The backward slices are depicted in Figure 4. We can observe that the correlation pitfall is being avoided for the Random Forest learner. However, this pitfall is not being avoided for the Logistic Regression and the Decision Tree learners, since they do *not* have a correlation filter in the backward slice of their execution sub-graph. Therefore, we report the existence of the correlation pitfall (P2) for these two learners.

D. Pitfalls reporting

Once we identify the pitfalls, we issue a suitable warning alerting the pipeline creator about such pitfalls. To facilitate pipeline maintenance, we also identify which specific learner of the pipeline ran into the pitfall.

IV. Pitfalls Analyzer - Prototype

We use KNIME as our tool of choice to carry out the model-driven implementation of our proposed approach. The reason is twofold. First, KNIME is extensively used in both industry and academia [26, 48, 49]. Second, KNIME supports a wide variety of functionalities and provide several building blocks for creating pipelines. These two characteristics facilitate the translation of our approach to a concrete implementation. Indeed, our implementation leverages the MDE approach inherent to KNIME, thereby utilizing the domain-specific modeling and pipeline creation capabilities of KNIME.

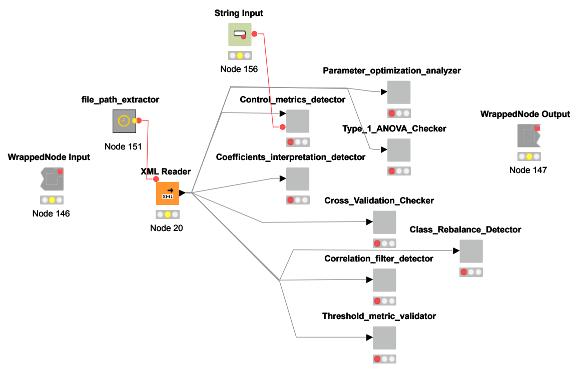


Fig. 6. Our prototype is implemented as a pipeline in KNIME with MDE.

Our prototype is a collection of KNIME pipelines. In Figure 6, we show the main pipeline of our prototype. Each pitfall shown in Table I is detected using a sub-pipeline . The main pipeline invokes these sub-pipelines via wrapped metanodes,

TABLE I
ANTI-PATTERNS THAT ARE SEARCHED FOR IN ORDER TO DETECT THE DATA SCIENCE PITFALLS DESCRIBED BY TANTITHAMTHAVORN AND HASSAN [17]

Pitfall	Anti-pattern	Anti-pattern nodes in KNIME	Tested Slice
P1	Absence of control metrics in the dataset used	Reader nodes	Backward
P2	Absence of nodes to removing correlated variables	Correlation Filter nodes	Backward
P3	Presence of class rebalancer nodes and subsequent use of threshold dependent measure computation nodes	SMOTE, Equal Size Sampling	Both
P4	Absence of parameter optimization nodes	Parameter Optimization Loop Start, Parameter Optimization Loop End	Both
P5	Absence of threshold independent performance measure nodes and presence of threshold dependent performance measure computation nodes	ROC Curve, Scorer	Forward
P6	Presence of cross validation nodes and absence of bootstrap validation nodes	X-Partitioner, X-Aggregator	Both
P7	Presence of Type-1 ANOVA computation nodes being fed by the learner's coefficients	One-way ANOVA	Forward
P8	Extraction of coefficients of a learner node	Regression Learners with an active coefficients port	Forward

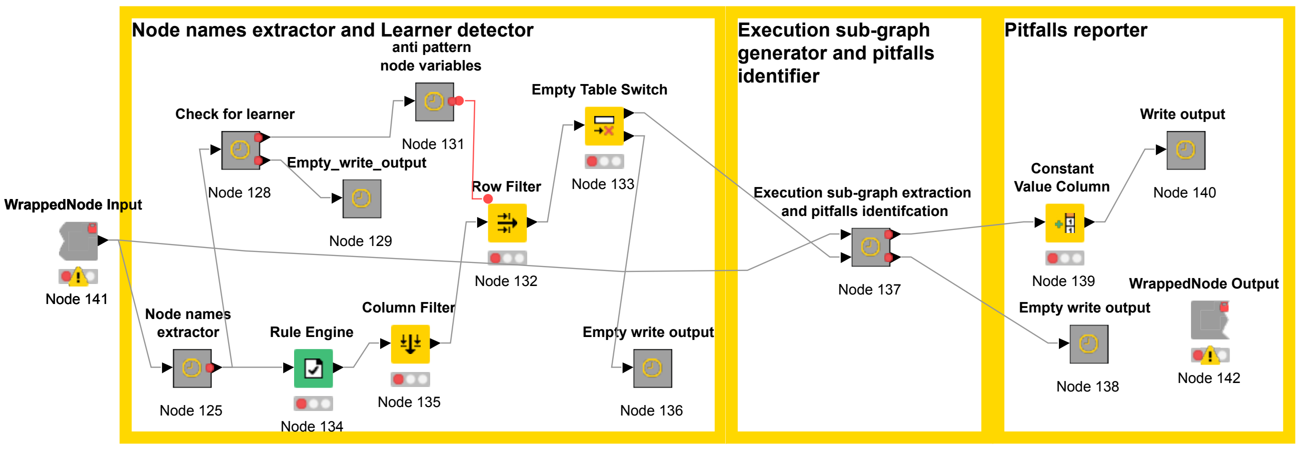


Fig. 5. Our implementation of the correlation pitfall (P2) detector.

which are represented in Figure 6 as grey rectangles (e.g., `Control_metrics_detector`).

As an illustrative example, Figure 5 shows our pipeline for the detection of the correlation pitfall (P2). This pipeline comprises several KNIME base nodes, such as "Rule engine" and "Row filter". We also use several wrapped nodes, which are made up of base nodes.

These wrapped nodes are different from the wrapped metanodes nodes (as seen in Figure 6). Wrapped nodes are typically meant for hiding parts of the pipeline to enhance comprehensibility, whereas wrapped metanodes are meant to promote reusability.

In the remainder of this section, we explain how we implemented each step of our approach. We rely on our correlation pitfall (P2) detector pipeline to support the explanation of our prototype.

A. *DAG Extraction (Step 1)*

For every pipeline created in KNIME, an associated XML file named `workflow.knime` is created in the working directory of the pipeline, which stores metadata of the pipeline. In particular, this file stores all details about the various nodes of the pipeline, the annotations, and the associated connections between the nodes. Therefore, we parse the `workflow.knime` to extract the DAG of the target pipeline (i.e., the pipeline to be analyzed for pitfalls). The `file_path_extractor` metanode and the `XML Reader` node shown in Figure 6 perform the DAG extraction.

B. *Execution sub-graph extraction (Step 2)*

The first two big yellow blocks of Figure 5 implement the *execution sub-graph extraction* step of our approach. This step contains two sub-steps: identifying the learners in the DAG and extracting the execution sub-graph of each learner (Figure 2). In the following, we describe these sub-steps.

Identifying the learners in the DAG (sub-step 1): The first sub-step is implemented via the `node names extractor` and the `check for learner` metanodes (Figure 5). Figure 7 shows the expansion of the `node names extractor` metanode. In this pipeline, we identify the various node names, unique ids, and node types from the target pipeline using XPath and other base nodes. Once the node names and their respective types are obtained, we check whether any of these nodes are learners. We accomplish this task with the check for learner pipeline (Figure 8).

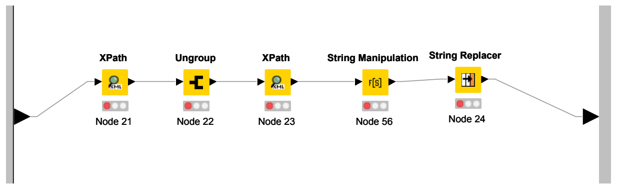


Fig. 7. Expansion of the node names extractor metanode from Figure 5.

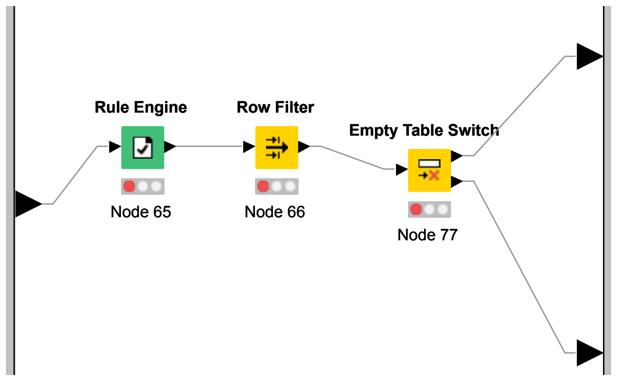


Fig. 8. Expansion of the Check for learner metanode from Figure 5.

Extracting the execution sub-graph of each learner (sub-step 2): The execution sub-graph is extracted as part of the Execution sub-graph extraction and pitfalls identification metanode from Figure 5. The expansion of this metanode is shown in Figure 9. We iteratively query the DAG of the target pipeline using XPath in order to extract the backward and forward slices of the execution sub-graph of each learner. This mechanism is coded in R, inside the "Table to R" node. We highlight that determining the backward and forward slices of each execution sub-graph is computationally costly. Hence, we only compute these slices when they are actually needed. To accomplish this goal, we determine which anti-pattern nodes have been found by the node names extractor metanode. If the presence (or absence) of an anti-pattern node denotes an anti-pattern (as given by Table I), we compute the slice.

C. *Pitfall detection (Step 3)*

Pitfall detection happens as part of the Execution sub-graph extraction and pitfalls identification metanode, which is expanded in Figure 9. To detect pitfalls, we loop through the anti-pattern nodes to check for their presence/absence (as given by Table I) in the execution path of the learner. For example, in our correlation pitfall detector pipeline, we search for

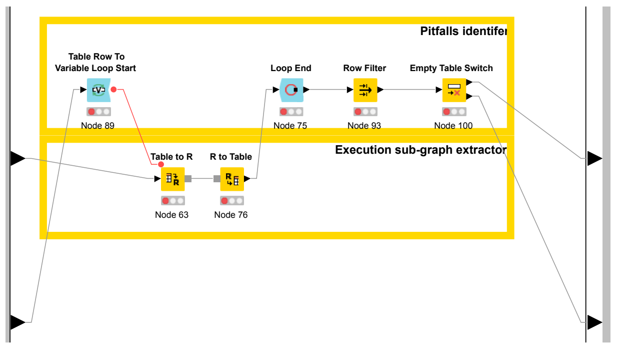


Fig. 9. Expansion of the Execution sub-graph extraction and pitfalls identification metanode from Figure 5.

the existence of the correlation filter node in the backward slice of the execution sub-graph of all the learners in the target pipeline. The "Table to Row Variable Loop start" and the "Loop End" node loops through the anti-pattern nodes to be checked for in the relevant slice of the execution sub-graph.

D. *Pitfalls reporting (Step 4)*

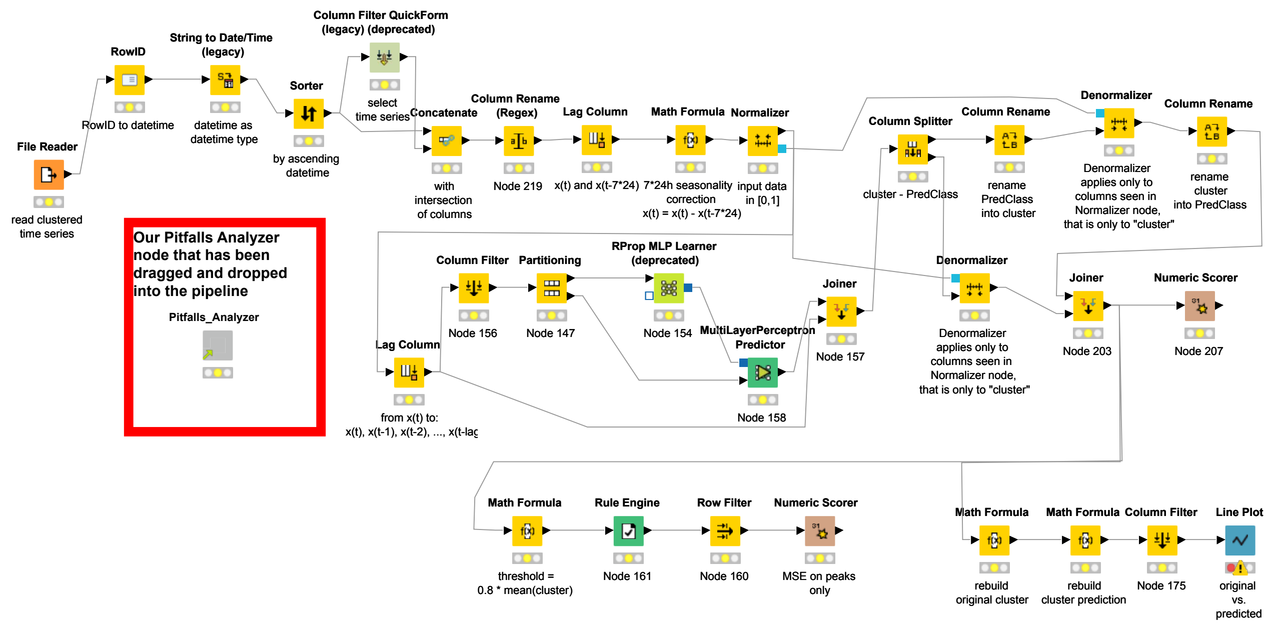
In this step, we report which pitfalls were detected. We also attach the id of the associated learner. We currently issue this report via a log message. The nodes that implement this report are shown in the right-most yellow block of Figure 5.

V. Evaluation

We evaluate the effectiveness of our Pitfalls Analyzer prototype by using it to analyze existing KNIME IoT data science pipelines. The KNIME website lists three IoT use cases, with several projects for each of them[1] created by KNIME experts. We choose one project from each use case, namely: Electricity Consumption Prediction, Rotor Failure Detection, and Bikeshare Predictive Analytics. Each project is comprises multiple data science pipelines, which are used to address the various tasks of the project.

We test our Pitfalls Analyzer on the aforementioned data science projects. Two of the four authors manually analyze each of the chosen projects (and all the pipelines that they contain) carefully with our experience to see how many of the pitfalls mentioned in Section II-A each project contains. Next, we run our Pitfalls Analyzer on these projects to evaluate if our pitfall analyzer is able to successfully identify all of the manually identified pitfalls. We finally discuss the implications of our results. We note that we do not check if the studied pipelines exhibits control metrics pitfall (P1), as it requires prior knowledge of what the control metrics pertaining to a specific project should be (as mentioned in Section II-A). Hence, our prototype does not flag any of the pipelines for the control metric pitfall.

[1]https://www.knime.com/white-papers



File Reader
read clustered time series
RowID
RowID to datetime
String to Date/Time (legacy)
datetime as datetime type
Sorter
by ascending datetime
Column Filter QuickForm (legacy) (deprecated)
select time series
Concatenate
with intersection of columns
Column Rename (Regex)
Node 219
Lag Column
x(t) and x(t-7*24)
Math Formula
7*24h seasonality correction x(t) = x(t) - x(t-7*24)
Normalizer
input data in [0,1]
Column Splitter
cluster - PredClass
Column Rename
rename PredClass into cluster
Denormalizer
Denormalizer applies only to columns seen in Normalizer node, that is only to "cluster"
Column Rename
rename cluster into PredClass
Our Pitfalls Analyzer node that has been dragged and dropped into the pipeline
Pitfalls_Analyzer
Column Filter
Node 156
Partitioning
Node 147
RProp MLP Learner (deprecated)
Node 154
MultiLayerPerceptron Predictor
Node 158
Lag Column
from x(t) to: x(t), x(t-1), x(t-2), ..., x(t-lag
Joiner
Node 157
Denormalizer
Denormalizer applies only to columns seen in Normalizer node, that is only to "cluster"
Joiner
Node 203
Numeric Scorer
Node 207
Math Formula
threshold = 0.8 * mean(cluster)
Rule Engine
Node 161
Row Filter
Node 160
Numeric Scorer
MSE on peaks only
Math Formula
rebuild original cluster
Math Formula
rebuild cluster prediction
Column Filter
Node 175
Line Plot
original vs. predicted

Fig. 10. A pipeline that is a part of the Electricity Consumption Prediction project. Our Pitfalls Analyzer node has been dragged and dropped into the pipeline (yellow rectangle).

A. Projects

Table II shows a summary of the characteristics of the projects that we evaluated. In the following, we briefly introduce each project.

TABLE II
CHARACTERISTICS OF THE PROJECTS USED FOR EVALUATION

Project	#Pipelines	Nodes	#Pitfalls	Pitfalls present
ECP	7	490	4	P2, P4, P5, P8
RFD	3	126	4	P2, P4, P5, P8
BPA	1	29	4	P2, P3, P4, P5

RFD- Rotor Failure Detection, ECP - Electricity Consumption Prediction, BPA - Bikeshare Predictive Analytics. Pitfalls detected via manual analysis.

Electricity Consumption Prediction (ECP) is the largest project that we use for evaluation. It is made up of 7 pipelines and a total of 490 nodes. ECP is a project that uses KNIME to classify the electric usage behaviour of Irish consumers [50]. The electric consumption behavior of various participating customers were collected through smart meters that are installed in the home and businesses of these participating consumers. The smart meter data of over 5,000 customers were collected over 2 years. In this project, consumer usage behaviors are first clustered using K-Means clustering [51] to identify common clusters. Then, Silipo and Winters [50] identify the behavior of each consumer as belonging to one of these aforementioned clusters using several learners.

Figure 10 shows one of the pipelines that make up the project. From the figure, we observe that the pipeline exhibits several pitfalls. For instance, the pipeline uses a "scorer" node to compute threshold-dependent performance measures from a learner in the pipeline. Furthermore, it lacks a node for the computation of a threshold-independent performance metric, thereby exhibiting a threshold-dependent metric pitfall (P5). Similarly, we can observe that the pipeline also exhibits a correlation pitfall (P2), an optimization pitfall (P4) and threshold-dependent metric pitfall (P5).

Rotor Failure Detection (RFD) is a project for predicting the failure likelihood of a rotor ahead of time using anomaly detection [52]. Rotors have sensors associated with them. In this project, along with the the time series data associated with 28 sensors that are attached to 8 different positions of a rotor. The signals are collected over approximately 2 years and leverages pipeline analysis.

Bikeshare Predictive Analytics (BPA) is a project to predict how to best restock the bike stations for a bike sharing business[2] [53]. The sensor data from each bike is feed into a decision tree learner to determine when to restock a biking station so that the biking station does not run out of bikes.

B. Results and Implications

Results: Our Pitfalls Analyzer successfully identifies all the pitfalls that we manually identified. We run our prototype on all the data science pipelines in each of three studied projects and we aggregate the detected pitfalls in each project and present them in Table III. As we can observe from Table III, all the pitfalls that were manually identified in each of the projects (as given in Table II) have been marked with a ×. These results showcase the usefulness and effectiveness of our

[2]https://www.capitalbikeshare.com/

Pitfalls Analyzer. We also observe that the same pitfalls occur repeatedly across multiple pipelines, which further reinforces the need for our Pitfalls Analyzer.

TABLE III
PITFALLS DETECTED IN THE EVALUATED PROJECTS

Project	P1	P2	P3	P4	P5	P6	P7	P8	Total
ECP		×		×	×			×	4
RFD		×		×	×			×	4
BPA		×	×	×	×				4

RFD- Rotor Failure Detection, ECP - Electricity Consumption Prediction, BPA - Bikeshare Predictive Analytics

Implications: Pitfalls Analyzer is considerably more scalable than manual inspection.

The 11 pipelines across the 3 projects comprise a total of 645 nodes, which we were able to analyze for pitfalls in under a minute with our Pitfalls Analyzer. The same task took approximately 1 hour to be accomplished manually by 2 of the 4 authors. Furthermore, if the pipelines are modified in the future, they would have to be inspected manually, which could be very inefficient, cumbersome, and error-prone. Therefore, an automated solution like our prototype is considerably more scalable than manual inspection. Furthermore, our approach could identify the pitfalls described in Section II for any type of learner (e.g, deep neural networks etc.,) within the ambit of the platform within which our pitfalls analyzer is implemented.

Pitfalls Analyzer is extensible. In addition to analyzing the common data science pitfalls, one could extend our Pitfalls Analyzer to support more general pipeline analyses with our approach. Essentially, our approach allows for static analysis of the graphical data science pipelines. One could extend our approach to ensure for *pipeline best practices* and *pipeline efficiency*. For instance, usage of short node names, flagging usage of deprecated nodes, enforcing annotations for nodes, and more could be enforced by extending our approach. Consider the pipeline that is part of the ECP project that is shown in Figure 10, which has a deprecated node as a part of it. Extending our approach, one could flag and recommend the user to remove such a node. In other words, one could create tools similar to LINT [54] or coverity [55] for pipeline analysis.

The KNIME implementation of our Pitfalls Analyzer, even while being just a prototype, successfully identifies all the pitfalls that we were manually able to identify in 3 openly available KNIME projects for IoT predictive analytics.

VI. LIMITATIONS

Our anti-patterns do not capture all manifestations of the pitfall. Our approach detects the presence of pitfalls in a pipeline using the anti-patterns shown in Table I. However, there are other ways in which these pitfalls could be committed in a pipeline construction. For instance, the presence of a parameter optimization loop node in KNIME does not necessarily indicate that all (relevant) learner parameters have been fine-tuned (pitfall P4). Similarly, the presence of a correlation filter node does not ensure a proper threshold for the elimination of correlated metrics was chosen.

Our approach cannot find pitfalls committed as a part of the code written using code snippet nodes. Many of the data science pipeline creation tools support code snippet nodes. These nodes allow scripts written in other programming languages to be natively executed as part of the pipeline. For instance, KNIME has an "R Source" node which supports native execution of nodes with scripts written in R as a part of the pipeline. Similarly, Rapidminer has "Execute Python" process which allows for scripts that are written in Python to be a part of the pipeline, were the pitfalls could be committed. Currently, our approach does not support these nodes, which would require programming-language specific static analysis tools.

Our approach could potentially fall prey to false positives and false negatives: Though we were able to successfully identify all the pitfalls in our evaluation, our approach can potentially report the existence of a pitfall when none exists and vice-versa. For instance, if the parameter optimization of a learner was achieved through general looping nodes, our approach will not be able to identify that parameter optimization is being performed (as we look for a specific anti-pattern). Similarly, a dataset in which rebalancing of the data was already performed elsewhere could be fed into the pipeline. Our approach would not be able to identify such a pitfall and would not report it to the user.

Our approach suffers from the limitations of static analysis. We cannot guarantee that pitfalls connected to the absence of anti-pattern nodes (e.g., Pitfall P2) would be avoided if an anti-pattern node is simply present in the execution sub-graph. For instance, if a Correlation Filter node is positioned after a decision construct (e.g., if-then-else constructs), it is possible that this filter will not be executed during runtime.

There are justified cases of anti-patterns pertaining to a pitfall that could be used in a data science pipeline. As Menzies and Shepperd [22] outline, while the pitfalls that we identify are general data science bad practices, there are instances where their use could be justified. For instance, when users are not planning to use learners to make decisions, they could choose not to remove correlated variables for improved performance of their learners. Another case is that some of our pitfalls (P1, P2, P3, P7, and P8) apply only when the learner is being used for interpretation rather than simple prediction i.e., When the learner in the pipeline is being used to understand the underlying root cause rather than for simple prediction tasks. However, our Pitfalls Analyzer reports the warnings irrespective of the focus. In that case, acting upon the pitfalls is at the discretion of the user.

We only focus on the checking of the pitfalls described by Tantithamthavorn and Hassan [17]. As we described in Section II-A, Menzies and Shepperd [22] and Mello et al. [21] also identify several data science pitfalls. Our rationale for choosing to detect the pitfalls described by Tantithamthavorn and Hassan [17] in lieu of others are as follows. First, these pitfalls are more quantitative in nature and thus lend themselves to automation more straightforwardly. Second, these pitfalls are deal with core aspects of data science, whereas the other studies also investigate pitfalls that are more domain-specific.

In summary, our approach and prototype cannot capture all manifestations of the pitfalls and has the aforementioned limitations. However, our goal was to create a proof-of-concept and help pipeline creators avoid common data science pitfalls in their pipelines. More generally, we hope our fledgling efforts could prove to be a fertile ground for a plethora of future studies to improve upon our approach.

VII. Conclusion and Future work

Data science has become very prevalent in various industries. Limited availability of data science experts and the availability of easy data science pipeline creation and automation tools enable non-expert users to apply advanced data science approaches with limited prior experience. However, prior research shows that they could be potentially making mistakes that invalidate the findings of their data science pipelines. Therefore, in this paper we propose, a novel MDE approach to detect the various data science pitfalls elucidated by Tantithamthavorn and Hassan [17].

We hope that our approach will enable users to detect potential pitfalls in the data science pipelines that they create. In addition, we provide a prototype implementation of our approach in a common and popularly used pipeline tool called KNIME. We also evaluate the effectiveness of our prototype by evaluating it on 3 openly available KNIME IoT predictive analytics projects. As part of future work, we foresee the following research opportunities:

Implementing our approach in other tools: Though in this paper we only showcase the implementation of our approach in KNIME, our approach could be generalized to other data science pipeline creation tools that support nodes (units that accomplish a specified function) similar to KNIME. On the one hand, our approach would be hard to replicate in tools like WEKA [15], as they do not support the wide variety of nodes that KNIME supports. For instance, our implementation makes extensive use of looping, code snippet nodes, and XPath nodes. However, WEKA does not support similar nodes, as it supports the creation of simpler pipelines. It would be hard to implement our approach in such constrained tools.

On the other hand, there are plenty of other widely used tools that encompass capabilities similar to those of KNIME, including RapidMiner Studio [16] and Microsoft Azure Machine Learning Studio [11]. Hence, by extracting the metadata of the created pipeline and finding nodes that behave similarly to those of KNIME, one could bootstrap the implementation our approach in these other pipeline tools.

References

[1] M. Marjani, F. Nasaruddin, A. Gani, A. Karim, I. A. T. Hashem, A. Siddiqa, and I. Yaqoob, "Big iot data analytics: architecture, opportunities, and open research challenges," *IEEE Access*, vol. 5, pp. 5247–5261, 2017.

[2] C. Gröger, "Building an industry 4.0 analytics platform," *Datenbank-Spektrum*, vol. 18, no. 1, pp. 5–14, 2018.

[3] J. Lee, H.-A. Kao, and S. Yang, "Service innovation and smart analytics for industry 4.0 and big data environment," *Procedia Cirp*, vol. 16, pp. 3–8, 2014.

[4] C. Lewis, Z. Lin, C. Sadowski, X. Zhu, R. Ou, and E. J. Whitehead Jr, "Does bug prediction support human developers? findings from a google case study," in *Proceedings of the 2013 international conference on Software Engineering*. IEEE Press, 2013, pp. 372–381.

[5] N. Dedić and C. Stanier, "Towards differentiating business intelligence, big data, data analytics and knowledge discovery," in *International Conference on Enterprise Resource Planning Systems*. Springer, 2016, pp. 114–122.

[6] M. Gualtieri, A. Rowan Curran, K. TaKeaways, and M. To, "The forrester wave(tm): Big data predictive analytics solutions, q1 2013," *Forrester research*, 2013.

[7] J. Ekström, "Outcome based business model siemens osakeyhtiö," 2018.

[8] H. Karre, M. Hammer, M. Kleindienst, and C. Ramsauer, "Transition towards an industry 4.0 state of the leanlab at graz university of technology," *Procedia manufacturing*, vol. 9, pp. 206–213, 2017.

[9] M. Rüßmann, M. Lorenz, P. Gerbert, M. Waldner, J. Justus, P. Engel, and M. Harnisch, "Industry 4.0: The future of productivity and growth in manufacturing industries," *Boston Consulting Group*, vol. 9, no. 1, pp. 54–89, 2015.

[10] T. H. Davenport and D. Patil, "Data scientist," *Harvard business review*, vol. 90, no. 5, pp. 70–76, 2012.

[11] R. Barga, V. Fontama, W. H. Tok, and L. Cabrera-Cordon, *Predictive analytics with Microsoft Azure machine learning*. Springer, 2015.

[12] P. Solutions, "Platform technology: Thingworx. 2016," *URL: https://www. thingworx. com/(cited on page 25)*.

[13] M. E. Anderson, "Technical trade-offs of iot platforms," in *Autonomous Systems: Sensors, Vehicles, Security, and the Internet of Everything*, vol. 10643. International Society for Optics and Photonics, 2018, p. 1064316.

[14] M. R. Berthold, N. Cebron, F. Dill, T. R. Gabriel, T. Kötter, T. Meinl, P. Ohl, K. Thiel, and B. Wiswedel, "Knime-the konstanz information miner: version 2.0 and beyond," *AcM SIGKDD explorations Newsletter*, vol. 11, no. 1, pp. 26–31, 2009.

[15] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: an update," *ACM SIGKDD explorations newsletter*, vol. 11, no. 1, pp. 10–18, 2009.

[16] M. Hofmann and R. Klinkenberg, *RapidMiner: Data mining use cases and business analytics applications*. CRC Press, 2013.

[17] C. Tantithamthavorn and A. E. Hassan, "An experience report on defect modelling in practice: Pitfalls and challenges," in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*. ACM, 2018, pp. 286–295.

[18] B. Kitchenham and E. Mendes, "Why comparative effort prediction studies may be invalid," in *Proceedings of the 5th international Conference on Predictor* ***Models*** *in Software Engineering*. ACM, 2009, p. 4.

[19] M. A. Hall, "Correlation-based feature selection for machine learning," 1999.

[20] D. Lazer, R. Kennedy, G. King, and A. Vespignani, "The parable of google flu: traps in big data analysis," *Science*, vol. 343, no. 6176, pp. 1203–1205, 2014.

[21] M. M. Mello, J. K. Francer, M. Wilenzick, P. Teden, B. E. Bierer, and M. Barnes, "Preparing for responsible sharing of clinical trial data," 2013.

[22] T. Menzies and M. Shepperd, "bad smells in software analytics papers," *Information and Software Technology*, 2019.

[23] I. Ruiz-Rube, T. Person, J. M. Dodero, J. M. Mota, and J. M. Sánchez-Jara, "Applying static code analysis for domain-specific languages," *Software & Systems Modeling*, pp. 1–16, 2019.

[24] T. S. Heinze, W. Amme, and S. Moser, "Static analysis and process model transformation for an advanced business process to petri net mapping," *Software: Practice and Experience*, vol. 48, no. 1, pp. 161–195, 2018.

[25] C. Saad and B. Bauer, "Data-flow based model analysis and its applications," in *Proceedings of the 16th International Conference on Model-Driven Engineering Languages and Systems - Volume 8107*, 2013, pp. 707–723.

[26] C. Idoine, P. Krensky, A. Linden, and E. Brethenoux, "Magic quadrant for data science and machine learning platforms (id: G00354456)," Gartner Research, Tech. Rep., January 2019. [Online]. Available: https://www.gartner.com/en/documents/3899464

[27] M. Lease, "On quality control and machine learning in crowdsourcing," in *Workshops at the Twenty-Fifth AAAI Conference on Artificial Intelligence*, 2011.

[28] N. Wickramage, "Quality assurance for data science: Making data science more scientific through engaging scientific method," in *2016 Future Technologies Conference (FTC)*. IEEE, 2016, pp. 307–309.

[29] D. Jensen, J. Neville, and M. Hay, "Avoiding bias when aggregating relational data with degree disparity," in *Proceedings of the 20th International Conference on Machine Learning (ICML-03)*, 2003, pp. 274–281.

[30] G. C. Cawley and N. L. Talbot, "On over-fitting in model selection and subsequent selection bias in performance evaluation," *Journal of Machine Learning Research*, vol. 11, no. Jul, pp. 2079–2107, 2010.

[31] M. Shepperd, D. Bowes, and T. Hall, "Researcher bias: The use of machine learning in software defect prediction," *IEEE Transactions on Software Engineering*, vol. 40, no. 6, pp. 603–616, 2014.

[32] J. Jiarpakdee, C. Tantithamthavorn, and A. E. Hassan, "The impact of correlated metrics on the interpretation of defect models," 2019.

[33] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan, "An empirical study of the impact of modern code review practices on software quality," *Empirical Software Engineering*, vol. 21, no. 5, pp. 2146–2189, 2016.

[34] B. Turhan, "On the dataset shift problem in software engineering prediction models," *Empirical Software Engineering*, vol. 17, no. 1-2, pp. 62–74, 2012.

[35] D. H. Wolpert, W. G. Macready *et al.*, "No free lunch theorems for optimization," *IEEE transactions on evolutionary computation*, vol. 1, no. 1, pp. 67–82, 1997.

[36] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "The impact of automated parameter optimization on defect prediction models," *IEEE Transactions on Software Engineering*, 2018.

[37] ——, "An empirical comparison of model validation techniques for defect prediction models," *IEEE Transactions on Software Engineering*, vol. 43, no. 1, pp. 1–18, 2017.

[38] KNIME, "Knime noding guidelines," *KNIME*, 2015. [Online]. Available: https://www.knime.com/sites/default/files/inline-images/noding_guidelines.pdf

[39] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, "A few billion lines of code later: using static analysis to find bugs in the real world," *Communications of the ACM*, vol. 53, no. 2, pp. 66–75, 2010.

[40] Y. Xie and A. Aiken, "Context-and path-sensitive memory leak detection," in *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5. ACM, 2005, pp. 115–125.

[41] W. R. Bush, J. D. Pincus, and D. J. Sielaff, "A static analyzer for finding dynamic programming errors," *Software: Practice and Experience*, vol. 30, no. 7, pp. 775–802, 2000.

[42] M. Das, S. Lerner, and M. Seigle, "Esp: Path-sensitive program verification in polynomial time," in *ACM Sigplan Notices*, vol. 37, no. 5. ACM, 2002, pp. 57–68.

[43] A. Saboury, P. Musavi, F. Khomh, and G. Antoniol, "An empirical study of code smells in javascript projects," in *2017 IEEE 24th international conference on software analysis, evolution and reengineering (SANER)*. IEEE, 2017, pp. 294–305.

[44] Z. Soh, A. Yamashita, F. Khomh, and Y.-G. Guéhéneuc, "Do code smells impact the effort of different maintenance programming activities?" in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1. IEEE, 2016, pp. 393–402.

[45] H. Prhofer, F. Angerer, R. Ramler, H. Lacheiner, and F. Grillenberger, "Opportunities and challenges of static code analysis of iec 61131-3 programs," in *Proceedings of 2012 IEEE 17th International Conference on Emerging Technologies & Factory Automation (ETFA 2012)*, 2012, pp. 1–8.

[46] A. Mandal, D. Mohan, R. Jetley, S. Nair, and M. D'Souza, "A generic static analysis framework for domain-specific languages," in *2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2018, pp. 27–34.

[47] K. Thulasiraman and M. N. Swamy, *Graphs: theory and algorithms*. Wiley Online Library, 1992.

[48] M. O. Gokalp, K. Kayabay, M. A. Akyol, P. E. Eren, and A. Koçyiğit, "Big data for industry 4.0: A conceptual framework," in *2016 International Conference on Computational Science and Computational Intelligence (CSCI)*. IEEE, 2016, pp. 431–434.

[49] C. Dietz and M. R. Berthold, "Knime for open-source bioimage analysis: a tutorial," in *Focus on Bio-Image Informatics*. Springer, 2016, pp. 179–197.

[50] R. Silipo and P. Winters, "Time series prediction of smart energy data," *Knime Whitepaper*, 2013. [Online]. Available: https://files.knime.com/sites/default/files/inline-images/knime_bigdata_energy_timeseries_whitepaper.pdf

[51] J. A. Hartigan and M. A. Wong, "Algorithm as 136: A k-means clustering algorithm," *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, vol. 28, no. 1, pp. 100–108, 1979.

[52] A. I. Silipo, Rosaria and P. Winters, "Time series prediction of smart energy data," *Knime Whitepaper*, 2018. [Online]. Available: https://files.knime.com/sites/default/files/181212_Whitepaper_Anomaly_Detection_Predictive_Maintenance_KNIME.pdf

[53] R. Silipo and P. Winters, "Predicitve analytics on bike share data," *Knime Whitepaper*, 2018. [Online]. Available: https://files.knime.com/sites/default/files/181212_Whitepaper_Anomaly_Detection_Predictive_Maintenance_KNIME.pdf

[54] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh, "Using static analysis to find bugs," *IEEE software*, vol. 25, no. 5, pp. 22–29, 2008.

[55] P. Louridas, "Static code analysis," *IEEE Software*, vol. 23, no. 4, pp. 58–61, 2006.