Metric-based Fault Prediction for Spreadsheets

Patrick Koch, Konstantin Schekotihin, Dietmar Jannach, Birgit Hofer, and Franz Wotawa

Abstract—Electronic spreadsheets are widely used in organizations for various data analytics and decision-making tasks. Even though faults within such spreadsheets are common and can have significant negative consequences, today's tools for creating and handling spreadsheets provide limited support for fault detection, localization, and repair. Being able to predict whether a certain part of a spreadsheet is faulty or not is often central for the implementation of such supporting functionality.
In this work, we propose a novel approach to fault prediction in spreadsheet formulas, which combines an extensive catalog of spreadsheet metrics with modern machine learning algorithms. An analysis of the individual metrics from our catalog reveals that they are generally suited to discover a wide range of faults. Their predictive power is, however, limited when considered in isolation. Therefore, in our approach we apply supervised learning algorithms to obtain fault predictors that utilize all data provided by multiple spreadsheet metrics from our catalog. Experiments on different datasets containing faulty spreadsheets show that particularly Random Forests classifiers are often effective. As a result, the proposed method is in many cases able to make highly accurate predictions whether a given formula of a spreadsheet is faulty.

Index Terms—Spreadsheets, Fault Prediction, Machine Learning

1 Introduction

Spreadsheets are omnipresent in organizations, where they are used for various tasks, e.g., accounting, data analytics, or decision-making [2]. Like any other type of software, spreadsheets can contain faults leading to substantial economic losses.[1] Numerous academic approaches for spreadsheet fault prevention, detection, localization, and repair have been proposed [3]. The effectiveness of such approaches often depends on their ability to predict the likelihood that a certain formula of a spreadsheet is faulty.

Since decades, the necessity for accurate fault prediction has been widely recognized in general software development. Fault prediction models can provide a number of benefits when used to, e.g., *(i)* improve test processes by focusing on fault-prone modules, *(ii)* selecting among design alternatives, and *(iii)* identifying refactoring candidates [4]. Various fault prediction approaches have been developed which attempt to generate an oracle that is able to classify a certain piece of code as either faulty or correct.

A common approach is to frame the prediction task as a supervised classification problem and to apply machine learning (ML) techniques to solve this problem [5], [6], [7]. The training data required for these techniques is typically derived from collections of faulty software artifacts with explicitly labeled faults. *Software metrics* are commonly used as predictor variables (*features* in ML terminology) in such a problem formulation. The values of the predictor variables (*observations*) are gained by routines that compute measurable characteristics of the faulty programs.

Various metrics were explored for predicting faults in general software, e.g. the size, complexity, or even the modification history of a program [8], [9]. Given an output of metrics for a certain part of an input program (e.g., an individual statement), the ML task is then to learn a classification model that can make an accurate prediction whether this part is faulty or not.

As fault probabilities for general software, knowledge about possible faults in spreadsheet formulas can be used, e.g., to point the developer to potential problems during spreadsheet construction [10], to initialize and support debugging processes [11], [12], [13], or to guide algorithmic testing techniques [14]. However, only limited research exists that investigates methods to predict faults in spreadsheets. A major step in that direction was the adaptation of *code smells* to the spreadsheet domain [10], [11], [15], [16], [17], [18]. Like code smells for general software, *spreadsheet smells* point out "smelly" parts of spreadsheets that have an increased risk of containing faults. The smelly parts are detected using certain measurable characteristics of spreadsheets—*spreadsheet metrics*—such as the number of referenced empty cells, the average length of calculation chains, and the number of references to other spreadsheets.

One of the first successful applications of smell detection techniques for fault prediction in spreadsheets is a debugging algorithm that uses detected smells [15]. In our own recent work [1], we investigated the use of spreadsheet smells to train an ML-based fault predictor for spreadsheets. However, the obtained results were preliminary, as we focused on a limited set of spreadsheet metrics and we did not conclusively investigate the choice of an appropriate learning technique. In this work, we close this research gap and make the following contributions:

1) We present a catalog of 64 spreadsheet metrics, which were either proposed in the literature or designed based on our expertise. When assembling the catalog we tried to maximize the diversity of the metrics so that they cover as many faults occurring in practice as possible.
2) To assess the usefulness of our catalog, we evaluate the predictive power of each metric in isolation on three datasets using three learning algorithms. The results

- *Patrick Koch, Konstantin Schekotihin, and Dietmar Jannach are with AAU Klagenfurt, Austria. E-Mail: firstname.lastname@aau.at Corresponding author: Konstantin Schekotihin*
- *Birgit Hofer and Franz Wotawa are with TU Graz, Austria. E-Mail: {bhofer,wotawa}@ist.tugraz.at.*

1. See also http://www.eusprig.org/horror-stories.htm.

show that many formula faults in our spreadsheet collections can theoretically be identified by one of the metrics. However, the performance of predictors based on individual metrics is not satisfactory.

3) We therefore suggest an ML-based approach that uses all metrics of our catalog to train a classifier to predict faults in spreadsheet formulas. We specifically explore the use of Random Forests (*RF*) as a learning technique for two reasons: *(i)* *RF* is known to perform well on small-sized datasets, and *(ii)* the technique has a built-in mechanism for feature selection. A performance comparison of *RF* models with a number of alternative approaches on three spreadsheet collections shows that *RF* models represent the overall best choice across different scenarios in terms of measured *F1* values. Additional experiments demonstrate the importance of *RF*'s built-in feature selection mechanism.

The remainder of paper is organized as follows. We overview previous work in Section 2 and describe the proposed method in Section 3. Next, we lay out the general design of our experiments in Section 4. We then provide details and results for three experiments: one considering individual metrics as predictors (Section 5), one providing the results for the proposed approach (Section 6), and one on the importance of feature selection (Section 7). Threats to validity are outlined in Section 8. Section 9 concludes the paper and provides an outlook on future work.

2 Previous Work

We discuss related work from two perspectives: *(i)* traditional software engineering and *(ii)* spreadsheet specific.

2.1 Fault Prediction for Traditional Programs

Software fault prediction attracted many researchers. Catal [19] discussed 90 papers published between 1990 and 2009 that propose a variety of ML- and statistics-based approaches to fault prediction in software artifacts. Examples of common ML techniques range from logistic regression, over classification trees, to deep neural networks. Catal indicated that the considered approaches often suffered from insufficient data being available for learning, which is still a problem in today's application scenarios.

Radjenovic *et al.* [8] compared 106 studies related to fault prediction metrics in software published between 1991 and 2011. One main outcome is that the fault prediction performance of certain metrics can depend on a number of factors such as the type of the predicted faults or the used metrics. The authors found that object-oriented metrics were more successful fault predictors than traditional size and complexity metrics, and process metrics seemed to be more effective in detecting post-release faults in software.

Zeller [9] proposed a method that learns patterns from a larger pool of software projects and then uses these patterns to classify individual software components as faulty or correct. One main goal of this approach is to derive the rules for software smells from past projects. Similar to this work, our approach also has a learning step that consists of deriving classification models that combine multiple metrics in one prediction function. We do, however, rely on a larger catalog of pre-defined smells for this purpose.

2.2 Fault Prediction for Spreadsheets

The goal of spreadsheet fault prediction approaches is to automatically point out likely faulty cells. As in the software domain, the success of a prediction method depends strongly on the quality and applicability of the used metrics, heuristic algorithms or decision procedures.

Early approaches aimed at specific types of faults and used various heuristic decision procedures to identify exactly these types of faults. For instance, the fault prediction tools UCheck [12] and Dimension [20] first infer and assign "data types" to input cells (e.g., cells with numeric values) and formulas of a spreadsheet. These types are derived from the text values of header cells that are positioned in the same row or column as the related input cells. The tools then iteratively propagate the types of the input cells to the formulas that refer to those inputs. A formula cell is reported as possibly faulty if the propagated type of this cell does not match the derived type obtained from the header information. Other approaches, such as AmCheck [21], its successor CACheck [22], and EmptyCheck [23], focus on cell arrays. A cell array is defined as a set of columns or rows comprising formula cells that are functionally related and that are surrounded by "borders" of either empty cells or cells that contain fixed values. The first two tools predict a cell to be faulty (smelly), if its formula deviates significantly from other formulas of the same array, whereas EmptyCheck uses a clustering algorithm to identify empty cells that probably should contain a formula.

A drawback these approaches is that they are designed to detect specific types of faults such as incorrect data types or mismatching cells in an array, which can lead to a limited coverage and applicability [24]. Our method is based on an extensive set of spreadsheet characteristics, allowing for the detection of a variety of fault types. Moreover, it is easy to extend. If a new metric, e.g., for a new fault type, is added to our catalog, the underlying ML algorithm will automatically incorporate its output into the fault prediction model.

In practice, designing a general heuristic procedure for fault prediction can be a tedious process. Given enough training data, ML methods can simplify this problem by finding a model that approximates such heuristics. A recent method by Singh *et al.* [25], called Melford, follows such an approach and uses ML to train a classifier for the automatic prediction of "number-where-formula-expected" faults in spreadsheets. In particular, Melford trains a neural network using specific abstract representations of faulty spreadsheets. An experimental evaluation revealed that the approach can classify instances in which a number is erroneously placed instead of a formula with high precision.

The applicability of Melford is still limited, as it currently can only recognize one fault type, and extending it to recognize others would require significant effort: first, one would have to design additional abstractions, which cover other fault types; second, an optimization of the neural network structure might also be required to allow for learning of an accurate classification model. In contrast, our approach does not rely on structural abstractions of faulty spreadsheets. Instead, we use the results of applying spreadsheet metrics, which encode the existing knowledge from research about possible problems in spreadsheets. As

the experiments show, our set of metrics is able to make highly accurate predictions for various types of faults.

Abreu *et al.* [11] proposed to use spreadsheet smells for fault prediction. Their tool, called FaultySheet Detective [15], uses the output of spreadsheet *smells*, such as complex formulas, missing inputs, or problematic dependencies [10], [16], [26], [27], to trigger a specific fault localization algorithm. Technically, their approach has two phases. In the first step, their technique computes two sets of cells: *(i)* the set of smelly cells and *(ii)* the set of output cells. Next, the algorithm determines calculation chains for all output cells and triggers a Spectrum-based Fault Localization algorithm for predicting the likelihood of each cell being faulty.

An advantage of Abreu *et al.*'s prediction model is that it does not require any learning phase to predict faults in formulas. An underlying assumption, however, is the availability of reliable smell detection thresholds, i.e., if for some cell the value of the metric used for detecting the smell is above the threshold, then this cell is considered as smelly. While for certain smells researchers suggested reasonable thresholds, e.g., [10], [16], the optimal threshold values are unknown in general and might depend on the given application domain. In contrast, our prediction model can deal with situations where the individual metrics are not necessarily strong predictors for the given domain. In fact, by combining multiple metrics in one prediction model and by automatically adjusting the weights through a learning procedure, we can achieve highly accurate predictions even in presence of weak individual predictors.

The ExceLint add-in [28] automatically finds faulty formulas using the assumption that spreadsheets follow "rectangular-like" patterns, i.e., formulas in the same row or column are very likely to have the same semantics. Therefore, any formula that does not fit into a rectangular-like layout is probably faulty. To find such layouts, ExceLint first transforms all formulas into a two-dimensional vector representation, called fingerprints. The value of each coordinate is defined by the sum of the relative column/row distances from the formula cell to other cells referred in the formula. Next, a binary decomposition algorithm finds regions of fingerprints by recursively dividing the spreadsheet into two parts such that each split minimizes the normalized Shannon entropy of the fingerprints in both subdivisions. Lastly, the tool generates repair suggestions for deviating formulas. Suggestions are generated that would lead to a moderate reduction of the overall entropy of the spreadsheet, which is typical for genuine fixes of faults.

ExceLint detects various fault types that occur due to erroneous formulas. However, since the approach does not take the structure of the formulas into account, it is unable to identify certain types of faults occurring, e.g., due to incorrectly applied functions (wrong order of arguments) or operators ("+" instead of "-"). Our approach, in contrast, can include these and other fault types by adding new metrics to the catalog that are indicative of the specific faults.

Of the described approaches only FaultySheet Detective and ExceLint are directly comparable with our method, since they are also designed to detect *various fault types* regarding formula cells of a spreadsheet. The other discussed techniques either focus on detecting a *single fault type* or detect faults in cells that do not contain formulas.

3 Metric-based Fault Prediction

The main idea of our approach is to use a given collection of spreadsheets containing labeled faults to train an ML model for fault prediction. The predictor variables of the ML problem correspond to an extensive set of metrics. The learning dataset is obtained by computing the values for the metrics (i.e., the observations) for the labeled spreadsheets.

Thus, formally, the *inputs* of our approach are *(i)* a set of training spreadsheets containing faults in which all formulas are either labeled as faulty/correct, and *(ii)* a set of metrics, where each metric takes a spreadsheet as input and returns a value (real number) for each formula of the spreadsheet. The *output* is a fault predicting classifier that, given a set of observations computed by our metrics for a formula of a previously unseen spreadsheet, returns the likelihood that this formula is faulty. The general *schema* to produce this classifier includes three main steps:

- Generation of a training and evaluation dataset from labeled spreadsheets using metrics from the catalog.
- Selection of a learning algorithm.
- Training (optimization) and evaluation.

Before we discuss the technical aspects for each step, we first present the details of a catalog of spreadsheet metrics, which we designed for the purpose of this work and which we use as a basis for our experimental evaluations later on. To support the reproducibility of our work, we share all used datasets and source code implementing spreadsheet metrics, ML optimization & training, and evaluation procedures for the trained ML models in an online appendix.[2]

3.1 Catalog of Metrics

We designed the catalog of 64 metrics shown in Table 1 based both on existing work from the literature and our expertise. We selected the metrics to provide a broad range of different measurements to cover a wide variety of faults occurring in practice. We first scanned previous research that discussed possible characteristics for assessing the quality of spreadsheets and extracted those ideas that seemed likely to be correlated with certain types of faults in spreadsheets. Not all identified characteristics could be applied directly. In such cases, the general idea of the metric was adapted so that it can be measured with a numeric value. For example, Metric 2 is conceptually based on the *Pattern Finder* [26] spreadsheet smell. The original heuristic checks if any other cell of the same type can be found within a certain range of the cell, whereas our implementation counts the number of cells that separate a given cell from the next cell of the same type in the same row. In addition, we designed novel metrics that either *(i)* cover basic spreadsheet features (e.g., Metric 1), *(ii)* extend on established ideas (e.g., Metric 29), or *(iii)* are based on personal experience from our previous research (e.g., Metric 4).

All metrics of the catalog can be divided into four categories, according to the spreadsheet part for which they are calculated: *(i) cell metrics* focus on characteristics of individual cells, *(ii) formula cell metrics* consider features derived

2. https://spreadsheet-research.github.io/Metric-based-Fault-Prediction-for-Spreadsheets/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2944604, IEEE Transactions on Software Engineering

from the content and the position of formula cells, *(iii) formula metrics* analyze formulas in isolation, and *(iv) worksheet metrics* take properties of the entire worksheet into account. Some of these metrics implicitly consider *structural* aspects of the spreadsheet, e.g., Metrics 9 or 10, which look for possible outliers in rows or columns, or Metric 61, which considers how a spreadsheet is organized in worksheets. Additional metrics, which specifically focus on structural aspects like the organization of calculations into blocks, can easily be added to our catalog.

Since our method aims to train fault predicting classifiers and many ML algorithms perform best with numeric inputs, we require every metric to output a real value for each formula cell in a given spreadsheet. Correspondingly, we modified the worksheet metrics which by definition return one value per worksheet, to assign this value to each cell of the worksheet. Overall, the provided metrics cover a wide range of faults, as will be shown in our experiments. However, the catalog can also be extended depending on the specific problem, e.g., to include metrics that consider formulas using custom functions declared in Visual Basic. A description of the underlying intuition of each metric of our catalog can be found in the online appendix. Generally, the metrics in our and related approaches serve as *heuristics* that are considered indicative of a fault. Therefore, a detected smell does not necessarily mean that a fault actually exists.

3.2 Preparing the Learning Dataset

Dataset preparation follows a straightforward process. The inputs of the process are a collection of faulty spreadsheets and a set of metrics. Each formula cell of every spreadsheet has a designated label (correct or faulty), and is provided as input to every metric of the catalog. We compute 64 values for each labeled formula cell. The results are stored in a table, where each row corresponds to a labeled formula cell, and each column corresponds to one of the 64 metrics. An additional column contains each formula cell's label. There are no missing entries in the table. This might be required by the learning algorithm. Accordingly, when alternative or additional metrics that have no defined value for some situations should be used with our approach, one can use data imputation techniques to fill eventually occurring gaps. We use this table to create labeled feature vectors. Figure 1 shows the general schema of the resulting dataset.

Fig. 1: Structure of the training data

$Metric_1$	...	$Metric_m$	$Label$
$value_{1,1}$	...	$value_{m,1}$	correct
...	...	...	...
$value_{1,n}$	...	$value_{m,n}$	faulty

The absolute values returned by the metrics of our catalog have largely different ranges. One metric might, for example, return values between 0 and 1, whereas another one returns positive integers between zero and 1000. Therefore, before training a classifier, we re-scale all values of every feature ($Metric_i$) in the training set using a widely adopted standardization procedure: from each $value_{i,j}$ we subtract the mean $\overline{Metric_i}$ of all observations and then divide by their standard deviation S_i (z-scores).

3.3 Selecting a Learning Algorithm

Numerous algorithms can be applied for the described learning problem, from Logistic Regression, over Support Vector Machines, to recent Deep Learning techniques. Since it is challenging to predict the relative performance of different ML algorithms for specific datasets, we evaluate algorithms from different families. Some general considerations should, however, be taken into account when choosing an algorithm. First, regarding the *dataset sizes*, the number of labeled spreadsheets might be low in some applications. The chosen algorithm should therefore be stable also in cases when only limited data is available. Furthermore, the algorithm should scale well for cases where data is abundant. Second, the *importance of features* can vary across datasets. Our metric catalog is designed to cover a broad variety of fault types. However, if a specific spreadsheet collection does not contain certain fault types, then the learning algorithm should disregard the corresponding metrics.

Consequently, the used learning algorithm should *(i)* be known to perform well on small to medium learning datasets, and *(ii)* have an explicit or implicit mechanism to *automatically* select or emphasize the most informative metrics (*feature selection*). Taking these considerations into account, the Random Forests (*RF*) algorithm represents a suitable choice for the given learning problem [34]. RF is an ensemble approach, which uses the concept of *bagging* to train N decision trees on N randomly selected subsets of the training set. The final classification model is then determined as an average of the results obtained by the individual trees. Besides variance reduction through bagging, *RF* uses only a specifically sampled subset of features for each of the underlying decision trees during their construction. This leads to a better representation of the various features in different trees and to a reduction of the correlation between the features.

3.4 Training and Evaluation Aspects

Most machine learning algorithms have a number of hyperparameters that can be fine-tuned to achieve the best performance. In case of *RF*, for example, the number of trees N to use during learning is such a parameter. In the context of our experimental evaluation described below, we therefore applied *grid search* in combination with a 10-fold cross valuation procedure for all tested algorithms and for all datasets to determine the best parameters.[3]

As optimization goal we use the *F1* measure, i.e., the harmonic mean of precision and recall. Precision shows how many of the formula cells that were predicted to be faulty are actually faulty. Recall, in contrast, represents the proportion of true faults, which were correctly recognized as such. Since obtaining perfect recall can be accomplished by predicting that every formula is faulty, recall and precision are, as usually, combined in the *F1* measure.

Like in many other learning scenarios, the distribution of the labels in the learning data is very imbalanced. The large majority of the observations in our training datasets are labeled as "correct" whereas only a fraction of the

3. Grid search systematically explores a defined range of possible parameter values.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2944604, IEEE Transactions on Software Engineering

TABLE 1: Proposed Catalog of Spreadsheet Metrics

	Metrics computed per cell
1	Column number (position) of the cell in the worksheet.
2	Column distance to relative [11], [16], i.e., distance to the next cell in the same column that has the same cell type; we use a predefined and configurable maximum value for this metric. Adapted from *Pattern Finder* to provide a numeric score.
3	Row distance to relative [11], [16], i.e., distance to the next cell in the same row that has the same cell type; we use a predefined and configurable maximum value for this metric. Adapted from *Pattern Finder* to provide a numeric score.
4	Range references to cell, i.e., number of range references referring to this cell. Counts multiple references from the same cell individually.
5	Direct references to cell, i.e., number of direct references (not as part of range references or named ranges; multiples are counted individually.)
6	Number of any references to the cell, i.e., sum of metrics 4 and 5.
7	Number of direct and range references in other worksheets that refer to this cell.
8	Row number (position) of the cell in the worksheet.
9	Standard deviation (column) [11], [16], i.e., the absolute difference to the mean of the numeric values in the same column (for cells with numeric values).
10	Standard deviation (row) [11], [16], i.e., the absolute difference to the mean of the numeric values in the same row.
11	Successors [29], [30], [31], i.e., the number of cells that are referring to the cell by either direct or range references; also known as fan-in and visibility.
12	Number of successors in other worksheets [10].
	Metrics computed per formula cell
13	Complexity of conditional construct [29], calculated by adding the conditional complexity (Metric 35) to the complexity of conditional construct metrics (13) of referenced cells.
14	Dispersion of references [29], i.e., the exponential utility function of the weighted sum of the distances to references in the same worksheet.
15	Longest calculation chain that needs to be checked during evaluation of the cell[29], [27], [30].
16	Predecessors [30], i.e., the number of cells that are referenced by the cell by any reference type; also knows as fan-out and scope.
17	Distant predecessors [31], i.e., the number of predecessors that lie further away than a defined and configurable number of columns or rows from the formula.
18	Number of referenced cells that are empty (direct and range references) [16], [11].
19	Number of referenced cells that are in a different column (or different worksheet) [31].
20	Number of referenced cells that are in a different row (or different worksheet) [31].
21	Number of predecessors of this cell in other worksheets [10].
22	Number of referenced cells that only contain a single direct reference in their formula (Middleman predecessors) [10].
23	Number of referenced cells that are to the right or below the current cell, or in a worksheet with a greater sheet index (reverse predecessors) [31].
24	Cumulative range height growth [29], i.e., the sum of row heights (-1) of the ranges referenced by the cell; e.g., the reference A1:C5 adds a heights growth of 4.
25	Cumulative range width growth [29], [30], i.e., the sum of column widths (-1) of the ranges referenced by the cell; e.g., the reference A1:C5 adds a width growth of 2.
26	Number of different paths from input cells to the cell (reachability [29]).
27	Average calculation chain lengths of all references of the cell (avg. reachability [29]).
28	Number of references of this cell to other worksheets (counts multiple, coinciding references).
29	Sum of Manhattan distances between cell and referenced cells.
30	Column reference spreading [29], [30], i.e., the distance between the left-most and right-most cells of the set of referenced cells including the examined cell; we only consider referenced cells that are in the same worksheet as the examined cell.
31	Row reference spreading [29], [30], i.e., the distance of the top-most and bottom-most cell of the set of referenced cells including the examined cell; we only consider referenced cells that are in the same worksheet as the examined cell.
32	Worksheet reference spreading [30], i.e., the maximum worksheet distance between referenced cells.
33	Number of formulas located in the same worksheet whose formulas match in A1 notation (duplicated calculations [11])
34	Number of formulas located in the same worksheet whose formulas in R1C1 notation share a proper subtree without being the same formula [27], [32].
	Metrics computed per formula
35	Complexity of a formula based on conditional paths established by nested conditional operators in the abstract syntax tree (AST) of the formula (conditional complexity [29]). Non-conditional (sub-)expressions have complexity one, conditional operators (refer to list in Metric 36) have 2, conditionals double the complexity of nested conditionals (e.g., `=IF(A1,1,IF(A2,2,3))` has complexity 4).
36	Number of conditional operators (`AND`, `OR`, `XOR`, `NOT`, `IF`, `SWITCH`, `IFS`, `IFERROR`, `IFNA`, `COUNTIF`, `COUNTIFS`, `SUMIF`, `SUMIFS`) [31].
37	Number of AND and OR predicates (decision count [29], [30]).
38	Total number of operands and operators (i.e., sum of metrics 47 and 48).
39	Average depth of nodes in the formula's abstract syntax tree, e.g., `=A1+A2` has an average depth of 0.67 with the operator being the tree root and the two references being at depth 1 [29], [33].
40	Maximum depth of nodes in the formula's AST [29], [30], [31], [33], e.g., `=A1+A2` has a maximum depth of 1.
41	Number of references to (non-complex) ranges (e.g., `A1:B2`).
42	Number of binary operators (`+`,, `-`, `*`, `//`, `&`, `=`, `>=`, `>`, `<`, `<`, `<>`, `^`).
43	Number of direct references to individual cells (e.g., `A1`).
44	Number of constants in the formula [29].
45	Number of calls to built-in functions (e.g., `SUM`).
46	Number of references to names.
47	Number of operands [29], i.e., sum of metrics 44 and 52.
48	Number of operators [27], [29], [32], i.e., the sum of metrics 42, 45, 49, 50, and 53.
49	Number of parenthesis pairs.
50	Number of percent operators (`%`).
51	Number of combined range references (e.g., `A1:B2:C3`).
52	Number of references [27], [32], i.e., the sum of metrics 41, 43, 46, 51, and 54.
53	Number of unary operators (unary `+`,`-`).
54	Number of references in union form (e.g., `A1;B2` counts as one reference).
	Metrics computed per worksheet
55	Number of non-empty or referenced blank cells in the worksheet.
56	Number of formula cells in the worksheet.
57	Column number of the right-most non-empty cell in the worksheet.
58	Number of R1C1-unique formulas in the worksheet.
59	Position of the worksheet in the workbook's sheet tab.
60	Number of cells that are referenced by cells of this worksheet.
61	Number of cells in other worksheets that are referenced by cells of this worksheet.
62	Row number of the bottom-most non-empty cell in the worksheet.
63	Number of cells that refer to cells of this worksheet.
64	Number of cells in other worksheets that refer to cells of this worksheet.

formulas—below 4 %—is actually faulty. We therefore apply a *random oversampling* method [35] to counter this imbalance problem before training and evaluation. After the oversampling procedure, both classes (faulty and correct) appear equally often in the training data. The test dataset, however, remains unchanged.

Lastly, the parameter-optimized models are evaluated using a 10-time, 10-fold cross-validation procedure to avoid overfitting [36]. In this procedure, the dataset is randomly partitioned into ten chunks, and ten models are built based on this partitioning. Each model is built using nine chunks as training set, and one as testing set. The *F1* performance of each model as measured on the testing set is averaged to balance for randomness. Moreover, to account for a potential random influence of the initial partitioning, the whole process is repeated with ten different random initial partitions, and the results of all models are averaged. As usual, parameter optimization, training, and evaluation are done independently for each dataset.

4 Empirical Evaluation

To assess the effectiveness of our metric-based fault prediction method, we designed and conducted a number of studies, focusing on the following research questions:

RQ1 What is the predictive performance of the *individual* metrics listed in Table 1? Are they suited to uncover the various types of faults in spreadsheets?

RQ2 What is the predictive performance of the *combined* model as proposed in this work? How do *RF* models compare to alternative ML techniques?

RQ3 What is the importance of an algorithm's capability to emphasize individual metrics for a given dataset (feature selection)?

4.1 Research Datasets

We made experiments with three collections of faulty spreadsheets covering different application scenarios: *(i) unrestricted*, represented by *Enron Errors*, *(ii) restricted*, demonstrated by *INFO1*, and *(iii) synthetic*, portrayed by a modified version of the *EUSES* corpus. Spreadsheets in these collections contain at least one faulty formula and all known faulty formulas of each spreadsheet are labeled as such. We assume all remaining formulas to be correct.

Enron Errors [37] is a subset of the Enron spreadsheet corpus [38] containing real-world faults. The Enron corpus itself consists of over 15,000 spreadsheets that were found in emails of Enron employees after the bankruptcy of the company in 2001. From this larger corpus, researchers extracted 26 faulty spreadsheets in a tool-supported process [37]. Since the spreadsheets of the corpus cover various application problems, we consider this case as *unrestricted*.

INFO1 [39] contains spreadsheets developed by civil engineering students during an Excel course. The students had to solve two tasks (construction-related calculations) by modeling spreadsheets. Consequently, the collection includes spreadsheets with similar, albeit long and complex computations, which is why we regard it as a *restricted* case.

The modified *EUSES* [13] corpus contains artificial faults that were injected into spreadsheets of the original *EUSES* corpus [40], such that each spreadsheet contains exactly one faulty formula. Faults were injected by randomly selecting one formula cell of each spreadsheet and modifying it using a randomly chosen *mutation* operator. The authors considered only operators from [41], which correspond to the introduction of mechanical faults, i.e., typos. Examples of the operators include alternations of arithmetic or logical connectives, replacement of formulas by constants, etc. Thus, while the dataset comprises real-world spreadsheets augmented with injected faults, the *synthetic* case represents a simulation of real-world scenarios occurring due to typos.[4]

The dataset statistics are presented in Table 2. In the *Enron Errors* dataset, the structure of individual formulas can deviate significantly from one spreadsheet to another; in the *INFO1* collection, in contrast, similar formulas are not uncommon; in the *EUSES* corpus, finally, the mutations often lead to what one could consider an accidental typo when the formula is entered. Since the percentage of faults for the *EUSES* corpus is lower than for the other datasets, we can use this dataset to investigate the performance of algorithms when there are only few faults.

TABLE 2: Statistics of the Research Datasets

Corpus	Scenario	Spreadsheets	Formulas	Faulty
Enron Errors	unrestricted	26	16.790	2.9 %
INFO1	restricted	119	174.493	3.0 %
EUSES	synthetic	576	284.109	0.2 %

4.2 Overview of the Performed Studies

We conducted three distinguished studies:

- *Study 1* (Section 5) addresses **RQ1** and evaluates the performance of classification models that are based on *one single* metric (feature).
- *Study 2* (Section 6) investigates **RQ2**. Corresponding to the approach presented in Section 3, we trained and optimized various classification models for each of the datasets. In particular, we compared the fault prediction performance achieved by a Random Forest model with the single-metric models (Study 1) and commonly used alternative models trained by Support Vector Machines, Adaptive Boosting, and Deep Neural Networks.
- *Study 3* (Section 7) considers **RQ3** and examines the importance of feature selection. Since feature selection is an implicit functionality of *RF*, we determined the impact of feature selection by combining a related algorithmic nearest-neighbor method [42] with an explicit feature selection technique.

5 Individual Metrics as Fault Predictors

5.1 Experiment Setup – Prediction Models

We investigated three models to assess the prediction performance of classifiers that are built on one single metric:

4. To check whether the mutations and, thus, the resulting dataset are realistic, we manually analyzed how many of the real-world faults in the *Enron Errors* corpus could be the result of the (repeated) application of the mutations. The analysis showed that 20 of the 31 *Enron Errors* faults could in theory be created through the mutations, which we see as a positive indicator regarding the usefulness of the synthetic dataset. The detailed results of this analysis can be found in the online appendix.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2944604, IEEE Transactions on Software Engineering

- *Threshold-based models* (TB). Similar to [11], *TB* takes a training dataset, a metric, and a specific threshold value T as input. A previously unseen formula is considered faulty if the computed metric value for this formula is above T. The optimal threshold for each classifier is determined by a grid search procedure among a set of candidate values based on the training set.
- *Logistic regression* (LR). *LR* [36] is a popular and well studied method to train classifiers with binary output on numerical data. Given a learning dataset, the training method optimizes the coefficients of a linear model to fit the given data. Provided with only one feature, the *LR* procedure determines a single threshold value to classify previously unseen formulas as either correct or faulty. In contrast to the *TB* approach, no predefined candidate values are needed for the *LR* approach.
- *Decision trees* (DT). Given the same inputs as the previous models, decision trees [43] can be used to derive a set of rules for fault prediction of the form: If for a given formula cell a value v of the metric is in the interval $(L \leq v \leq U)$, then return label $X \in \{faulty, correct\}$. The interval bounds L and U, as well as the corresponding labels are inferred automatically during training.

5.2 Evaluation Results and Analysis

Among the different single-metric classifiers, the ones based on decision trees (*DT*) performed best on all datasets. Figure 2 shows the results of these classifiers for the three datasets in terms of precision (horizontal axis), recall (vertical axis), and the *F1* measure (radial lines). For example, the *DT* classifier learned for Metric 4 (Range references to cell) for the *Enron Errors* dataset in Figure2a has a precision of 35 %, a recall of 28 %, and an *F1* score of 31 %, according to its position in relation to the radial lines.

The evaluation of *DT* models using individual metrics in many cases lead to a comparably high recall, often with values above 80 %. The performance in terms of precision is consistently low across all datasets, with the best score at around 35 % for the *Enron Errors* dataset. In most cases, however, the precision scores remained below 10 %. We thus conclude that classifiers based on individual metrics are able to recognize genuine faults, but also return many false positives, i.e., they predict a fault even though there is none.

In terms of recorded *F1* measures, *DT* models performed best on the *Enron Errors* dataset. This dataset represents the *unrestricted* scenario and contains a variety of different spreadsheets and fault types. Looking at individual metrics, metrics 61 and 4 are strong fault predictors for this dataset. Overall, there are 16 single-metric classifiers with *F1* values between 0.2 and 0.3, indicating that several metrics are somewhat relevant for this collection. The *F1* results for the *INFO1* and *EUSES* datasets are generally worse than those for *Enron Errors*. This is likely due to the *restricted* and *synthetic* nature of those datasets. The spreadsheets of *INFO1* were created by students for a given task and thus contain formulas that are structurally similar. The applied mutations to introduce faults into the *EUSES* corpus changed only small parts of given formulas. Therefore, for both datasets the differences between faulty and correct formulas tend to be small, which makes it difficult to learn a classification rule. For the *EUSES* corpus, a lower formula error rate (0.2 %) might have reduced the prediction performance.

5.3 Discussion

While there is a clear indication that many metrics are actually able to recall faults in most cases, the precision of predictions returned by *DT* is low. With respect to **RQ1**, we conclude that the overall prediction performance of *DT*

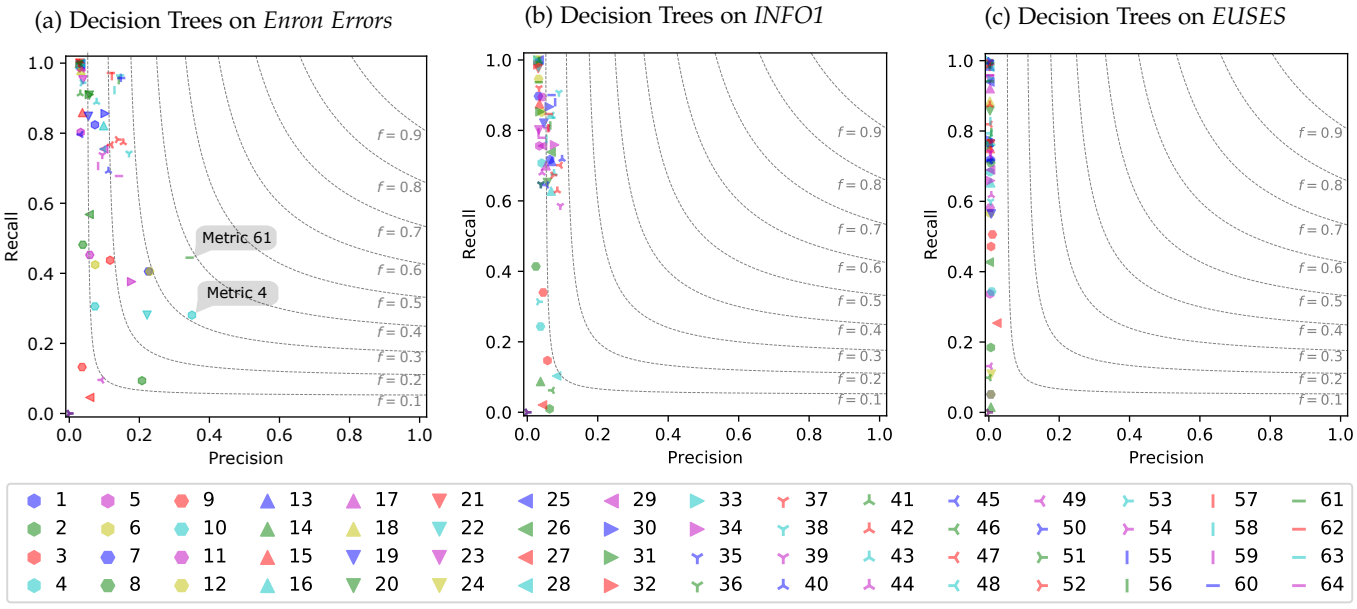

Fig. 2: Precision-recall performance for decision tree classifiers using single metrics on the *Enron Errors*, *INFO1* and *EUSES* corpora. The numbers in the legend correspond to the indices given in Table 1 and radial lines indicate *F1* measure values. Hexagons represent classifiers using metrics computed per cell, triangles represent formula cell-based metrics, inverted triangles are formula metrics, and bars indicate worksheet-based metrics.

models based on individual metrics is mostly inadequate for practical purposes. Consequently, we explore the combination of multiple metrics in the next section.[5]

6 Fault Prediction with Multiple Metrics

6.1 Experiment Setup – Prediction Models

Our stated hypothesis is that Random Forests (*RF*) algorithms are well-suited for the given problem. To validate this hypothesis, we compared the performance of predictors trained using *RF* and three alternative learning methods:[6]

- *Support Vector Machines (*SVM*)* [44]. *SVM*s have shown to be effective in a various domains, which is why we included them as baseline approach. Since training *SVM*s with traditional methods can be slow for larger datasets, we also use the faster but sometimes less accurate gradient descent method in its learning phase (*SVM SDG*). Training of *SVM*s depends highly on the value of the regularization parameter C, that determines the degree of importance to which the model might miss-classify the training data. Choosing an appropriate value for this parameter is important since it directly influences the ability of a model to generalize beyond the training dataset.
- *Adaptive Boosting (*AB*)* [45]. Similar to *RF*, this algorithm is a well-known ensemble technique. *AB* sequentially trains N "weak learners" (in our case decision trees) on weighted variants of the dataset. Thus, the algorithm corrects weaknesses of previously trained weak classifiers by emphasizing misclassified observations of the last iteration. Every newly trained weak learner tries to find a model that correctly classifies the most highly ranked observations first. Given an observation, an *AB* model determines a classification result as the weighted sum of the results of the underlying weak models, where the weights are determined by the prediction performance of the individual models.
- *Deep Neural Networks (*DNN*)* [46]. *DNN*s have been successfully applied to a wide range of classification problems. They were also used in Melford for the prediction of "number-where-formula-expected" faults [25]. In this study, we use a network architecture that is similar to the one used by Melford. In particular, our feed-forward neural network has N hidden layers, comprising 128 neurons each, and uses a rectified linear unit activation function. Instead of feeding abstractions to the network as in Melford, we provide our computed metric values to the 64 input neurons.

6.2 Evaluation Results and Analysis

The classification performance results for the three datasets are shown in Figure3. With the exception of the *SVM* models, the *F1* score results of the combined predictors are consistently higher than those of the predictors using individual metrics (Figure 2), even being consistently above 90 % for the *Enron Errors* case. In particular, the performance results in terms of precision, which were always below 40 % (and in most cases even lower) for the single metric classifiers, are largely improved on all datasets.

While the *SVM* method finished in reasonable time on the *Enron Errors* dataset, it exhibited unexpectedly long training times on the others.[7] Therefore, we switched to the *SVM SGD* variant of the algorithm that uses stochastic gradient descent instead of a complete optimization technique. As a result, we could train *SVM* models within an acceptable timeframe, but the performance of the obtained models decreased. However, given the results of both *SVM* variants on *Enron Errors* and of *SVM SGD* on the other datasets, we can exclude both variants from further discussion because of their low performance in comparison to the other methods.

TABLE 3: *F1* measure values observed in the study.

Algorithm	Enron Errors	INFO1	EUSES	Average
RF	0.98	0.77	0.61	**0.79**
AB	0.96	0.73	0.56	**0.75**
DNN	0.95	0.85	0.36	**0.72**
SVM SDG	0.35	0.21	0.03	**0.19**

6.3 Discussion

As highlighted above, all three of the more advanced models, *RF*, *AB*, and *DNN*, achieved very high *F1* scores for the *Enron Errors* dataset, with *RF* performing slightly better than the others. This indicates that combining different metrics as proposed in this work is beneficial and leads to a highly accurate predictions of faults (**RQ2**). Moreover, the results demonstrate that the success of our proposed method is not tied to one particular learning technique.

Similarly to the *Enron Errors* case, *RF*, *AB*, and *DNN* methods showed very good precision and recall on the other datasets. The ranking of the algorithms is not consistent over all experiments. For instance, *RF* performed best on the *EUSES* dataset and led to the second best results for the *INFO1* case. However, as presented in Table 3, on average *RF* outperformed all other learning methods. Therefore, we conclude that *RF* is an overall good algorithmic choice, providing stable performance across all tested datasets (**RQ2**).

Closer analysis shows that the *DNN* method excelled in terms of precision and recall on the *INFO1* dataset. On the *EUSES* dataset, in contrast, its precision score (27 %) was far below the values of *RF* (87 %) and *AB* (71 %). This indicates that the performance of *DNN*, while being generally good, largely depends on the characteristics of the dataset and/or fault types. More research is required to better understand the observed phenomena. From a practical perspective, the *DNN* method also has the disadvantage that it can be computationally complex. While training the *RF* classifier in our evaluation typically required a few hours on a single CPU, training a *DNN* usually required over a day for the bigger datasets. Differently from our expectation, however, we observed that the performance of *DNN* models does not

5. Since the best results, i.e., those obtained for *DT* classifiers, already sufficiently support the investigation of possible avenues for improvement, we omit a detailed discussion of the results of the *TB* and *LR* classifiers here. We refer the interested reader to the related plots and data in the provided online appendix.

6. Optimal values of hyperparameters found by the grid search for each algorithm are listed in the online appendix.

7. The optimization procedure did not finish within several weeks for the *INFO1* and *EUSES* datasets.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2944604, IEEE Transactions on Software Engineering

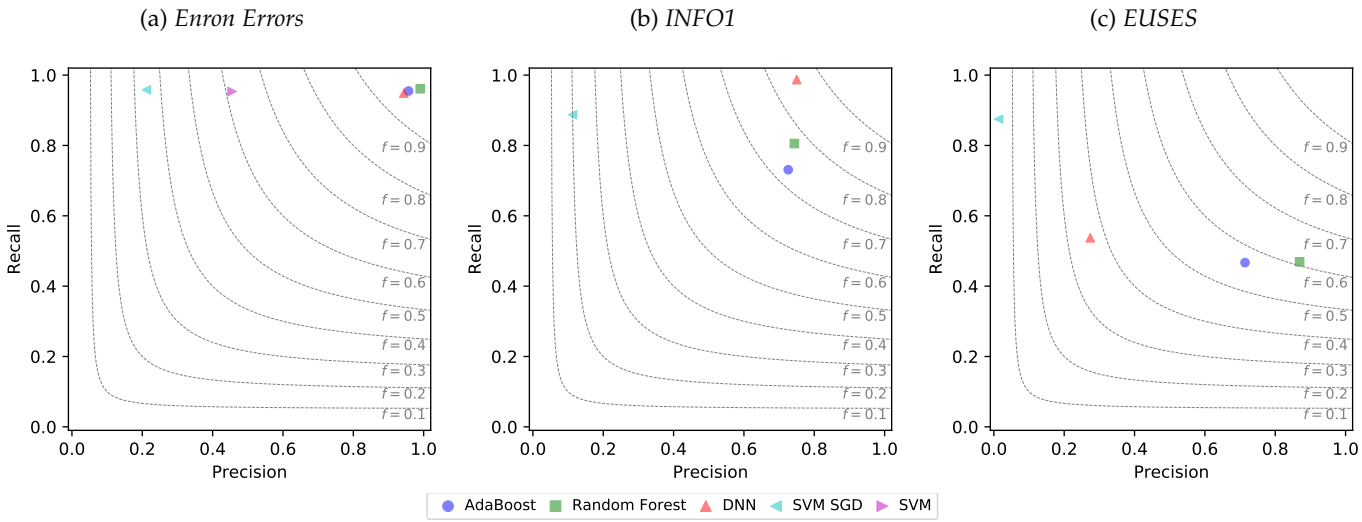

Fig. 3: Precision-recall results for different classifiers based on multiple metrics. Radial lines indicate *F1* measure values.

depend on the amount of input data, and it was among the best-performing methods for the smallest dataset.

Finally, comparing results across all datasets reveals generally lower accuracy results for the *synthetic EUSES* dataset. This trend was also observed for the single-metric models in the first study. We thus assume that the specific types of artificially injected faults in *EUSES* are difficult to capture by the metrics of our catalog. However, our catalog is open for extension to include metrics that are tuned to such fault types. Generally, the accuracy results for the synthetic dataset also depend on the chosen fault injection strategy, which can lead to easier or more difficult problems. Nevertheless, in presence of randomly injected faults, our approach proved its usefulness, as the combination of multiple metrics led to significant increases in accuracy.

6.4 Evaluation on Combined Dataset

In previous experiments, the models were evaluated on spreadsheets with the same origin as the training data. The results confirmed that fault prediction on organization-specific data/corpora is effective. To test the effectiveness of our approach when the training data includes spreadsheets from other sources, we created a *combined dataset* by merging all datasets from Table 2. We then made four measurements in which we trained classifiers on a training set derived from this *combined dataset*. The trained models were evaluated on four test datasets *(i)* three sampled from the three individual corpora, and *(ii)* one sampled from the *combined dataset*.

The first three experiments led to *F1* scores of 97 %, 77 %, and 59 % for the *Enron Errors*, *INFO1*, and *EUSES* datasets resp.[8] In comparison with the results from Table 3, the fault predictors were able to mostly maintain their performance levels even when the training sets comprise additional and possibly irrelevant data from other sources. The *RF* method therefore seems robust enough to learn classification models that can effectively discriminate between faults of different origin. The fourth experiment also supports this observation as the fault predictor scored an *F1* value of about 77 %.

8. See the online appendix for detailed results.

These results indicate that our method is suitable for two deployment scenarios: First, it can be used in organization-specific settings, where the fault predictor is trained on a collection of domain-specific spreadsheets. Second, providers of debugging tools might include models that are pre-trained on general spreadsheet collections in their products.

6.5 Comparison with Previous Work

To provide a point of reference for the proposed approach, we compared the evaluation results with our own previous work and with related efforts by other researchers. In comparison to our previous work in which our models were based on a smaller catalog of *smell metrics* [1], using the extensive metric catalog proposed in our current work led to large improvements in terms of prediction performance. In particular, using the new catalog in combination with an *AB* model led to an improvement of the *F1* value from 43 % to 95 % on the *Enron Errors* dataset. Furthermore, in this work we explored using *RF* models for predicting faults, which further improved classification performance.

As pointed out in Section 2, we mainly regard FaultySheet Detective [15] and ExceLint [28] as comparable with our approach. For the sake of comparison, we applied both tools to identify potentially faulty cells in the spreadsheets of the *Enron Errors* dataset. [9] We then compared the detected faults with the ground truth data consisting of 630 known faulty formulas that were used in our experiments. According to this evaluation, FaultySheet Detective highlights 3,586 cells as potentially faulty, revealing 179 of the known true faults for an *F1* value of 8 %. Likewise, ExceLint highlights 99 cells as potentially faulty, revealing 20 of the known true faults for an *F1* value of 5 %. A further manual application of the two tools to randomly drawn samples of the *INFO1* and *EUSES* datasets yielded similar results.

While the prediction performance of FaultySheet Detective in our analysis seems low, this is actually expected as

9. Since tool application and result analysis have to be processed by hand for both, FaultySheet Detective and ExceLint, we limited the exhaustive analysis to the Enron Errors dataset.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2944604, IEEE Transactions on Software Engineering

the tool is not designed to directly pinpoint faulty cells. Instead, its goal is to reveal those cells that are the most relevant ones to all detected smelly cells. Likewise, ExceLint can only reveal faults where cells diverge from proper rectangular spreadsheet layouts. While this approach can reveal highly relevant cases, not all existing faults fall into this category. In conclusion, the comparison shows that our proposed approach provides a clear benefit in scenarios that require the detection of diverse fault types in spreadsheets.

7 The Importance of Feature Selection

7.1 Experiment Setup – Prediction Models

The main hypothesis of **RQ3** is that learning approaches like *RF*, which implicitly perform some form of feature selection, are favorable for the given fault prediction task. In our case, this means that the learning technique would, depending on the specifics of the dataset, only consider the most relevant features in the training phase. Metrics that are not suited for a given collection of faulty spreadsheets are automatically disregarded. Feature selection (or elimination) is usually applied to reduce noise in the data, which subsequently leads to a performance increase of the learned models.

To validate this hypothesis, we performed a study on the *Enron Errors* dataset, which used k-Nearest-Neighbors (*k-NN*) in combination with *recursive feature elimination* (*RFE*) [47], a feature selection method that is conceptually similar to the *RF* method [42], [48]. *K-NN* assigns the label to a new learning example that is the most common one among the k nearest neighbors with respect to Minkowski distance. The *RFE* process works as follows. In the first iteration of the process, the *k-NN* method is evaluated in terms of the average *F1* measure using a feature set that encompasses *all* available features (metrics). *RFE* then evaluates all feature subsets that can be constructed by removing one of the features. From these subsets, the one with the best *F1* score is selected and the process is continued recursively until no more improvement can be made. If our hypothesis is true and feature selection is important, we expect a significant performance difference when comparing the results for the *k-NN* method with and without applying *RFE*.

7.2 Evaluation Results and Analysis

7.2.1 Value of Feature Selection

Figure 4 shows the performance of the plain *k-NN* method, the *k-NN* method with feature selection using *RFE*, and the *RF* model for the *Enron Errors* dataset. In this experiment, hyper-parameter optimization for *k-NN* revealed 3 to be the best value for k. The results indicate that feature selection has a significant positive effect on the prediction performance of *k-NN* (according to a Wilcoxon test with $p < 0.001$). The same observation was made for the *INFO1* dataset.[10] Regarding **RQ3**, we conclude that an algorithm's ability to focus on a subset of the available features through feature selection is important to obtain a high *F1* score.

Generally, the scalability of *k-NN* with feature selection is however low, which is why we did not make the measurement for the *EUSES* dataset. In practice, one would therefore rather only rely on the *RF* model and not use the *RFE* procedure that we used here for the analysis of the importance of feature selection.

10. For this dataset, *k-NN* was even slightly better than the *RF* model.

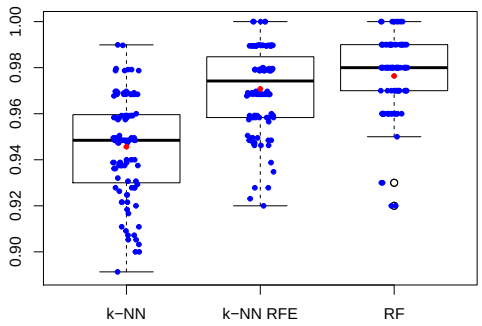

Fig. 4: *F1* measure for the *Enron Errors* dataset for *k-NN* (with and without *RFE*) and the *RF* model.

7.2.2 Analysis of Selected Features

An analysis of the feature importance weights, as produced by the *RF* model, led to the following main insights.[11]

Distribution of Importance Weights: We observed that *RF* was very successful in identifying key metrics in each dataset. For the *Enron Errors* and *EUSES* datasets, the 15 most important features accumulated more than two thirds of the total importance weight (slightly more for *INFO1*). Likewise, one third to one half of the metrics obtained very low weights, as they were identified to be not relevant for the corpus. Nonetheless, according to additional experiments, retaining only a small number—e.g., 10—of the most important metrics and removing the rest is detrimental to prediction performance, i.e., metrics with comparably low weights also contribute to high precision and recall results.

Ranking of Metrics: The relative importance of the metrics varies across datasets, and there is no distinct set of generally "best" metrics. This observation emphasizes both the value of having an extensive catalog of metrics and the need of using learning techniques that identify the important metrics automatically. In addition, the outcome of the metric selection process can also be helpful to provide explanations for the fault predictions. Another analysis showed that all *types* of metrics (e.g., worksheet metrics) appeared in the set of the most important metrics for each dataset. This, again, speaks for having a broad catalog of metrics. A side observation here was that *worksheet* metrics were consistently among the most important ones for all datasets. In fact, six of ten worksheet metrics led to significantly higher average readings for the worksheets of *Enron Errors* that contain faults. This confirms a common conjecture that larger and more complex worksheets are more likely to contain faults. A final observation is that in a global ranking of the metrics by *RF* weights over all datasets using the Borda count method, 7 of the top 10 metrics were novel and introduced in this work: **56**, **1**, 34, **63**, **60**, **58**, 30, **55**, 10, **57**. Most of the successful novel metrics were worksheet-related, which confirms the importance of considering this types of metrics that capture complexity-related aspects.

11. The exact values per dataset as well as additional charts are provided in the online appendix.

8 Threats to Validity

One possible threat to the *internal validity* of our research lies in the correctness of the used software for data preparation and the implementation of spreadsheet metrics, learning, and evaluation procedures. To minimize these risks, we mostly relied on existing software libraries provided for the Python programming language. Furthermore, we provide all source code of our metrics implementation, learning and evaluation scripts, as well as the datasets online for inspection and reproduction by other researchers.

As for threats to *external* validity, the main concern is the representativeness of the faults in the used datasets with respect to the overall population of spreadsheet faults. To minimize this risk, we relied on three different datasets in our experiments. The spreadsheets of the *Enron Errors* collection have been used in several empirical studies, and the specific set of recorded faults was obtained in a systematic and reproducible manner [37]. We therefore consider the risk that these faults are not representative as low. The spreadsheets of the *INFO1* collection contain faults that were introduced by students performing routine computations for civil engineering. While the situation in which the spreadsheets were created was artificial, the problem setting and development task itself was one that could also happen in the real world in an architectural firm. Consequently, we are confident that the resulting faults are also representative to a certain extent. Lastly, the *EUSES* dataset contains artificially injected faults that often resemble potential mechanical errors (typos) that users make when creating a spreadsheet.

The performance measures that we use for evaluating our approach, pose another potential threat to the external validity, as improvements in terms of generic performance measures might not necessarily translate into tangible benefits for end users. However, as presented above, there exists ample evidence of fault prediction methods providing various benefits in the domain of general software development [4], as well as in the spreadsheet domain [10], [11], [12], [13], [14]. Many of the fault prediction methods that are used by those approaches were evaluated using generally accepted performance measures like precision and recall, as is common in contemporary research on fault prediction techniques. We are thus confident that our chosen evaluation approach is suitable to assess of the benefit of our method and facilitates comparison with existing research.

9 Summary, Applications and Future Work

In this work, we have investigated how spreadsheet metrics can be successfully used for automated fault prediction based on machine learning. Two main insights of our research are that it is essential to *(i)* consider a combination of various metrics to achieve high prediction accuracy and *(ii)* that the choice of the learning technique can matter.

Generally, accurate fault prediction models can serve as a basis for a variety of quality assurance tools. One possible immediate application of our approach lies in an extension of the existing "Smart Tag" functionality of MS Excel. Smart Tags are small cell markers in MS Excel that point users to potential problems in their formulas, e.g., when a range reference seems incomplete. A future version of Smart Tags could also highlight cells that have a high fault probability according to the learned model. Once such a cell is inspected, the system can present a limited set of user-oriented explanations for those metrics that surpass a certain threshold for the given formula. A main advantage of such an approach would be that users of MS Excel are already familiar with the provided interaction mechanism. Alternatively, fault probabilities can also be used in more complex debugging environments to provide tangible user benefit. Specifically, our goals include the implementation of our approach within our debugging frameworks [49], [50], and to evaluate its usefulness through a user study.

From an algorithmic perspective, we plan further investigations regarding the use of neural approaches, which performed well on some of the datasets. We also plan to investigate the usefulness of the presented method for finding faults in non-formula cells such as the "number-where-formula-expected" fault considered by Melford. Finally, we see our catalog of spreadsheet metrics as a solid starting point that leads to satisfying accuracy results. Further extensions of the catalog, including variants of the proposed metrics, are possible to further increase the practical usefulness of our approach. Moreover, an in-depth analysis of the contribution of specific metrics and metric types will prove useful for understanding and improving prediction results.

10 Acknowledgment

The work was funded by the Austrian Science Fund (FWF), project I2144: *DEbugging Of Spreadsheet programs (DEOS)*.

References

[1] P. W. Koch, K. Schekotihin, D. Jannach, B. Hofer, F. Wotawa, and T. Schmitz, "Combining spreadsheet smells for improved fault prediction," in *ICSE (NIER)*, 2018, pp. 25–28.
[2] R. R. Panko and N. Ordway, "Sarbanes-Oxley: What About all the Spreadsheets?" *CoRR*, vol. abs/0804.0797, 2008.
[3] D. Jannach, T. Schmitz, B. Hofer, and F. Wotawa, "Avoiding, finding and fixing spreadsheet errors - A survey of automated approaches for spreadsheet QA," *J. Syst. Softw.*, vol. 94, pp. 129–150, 2014.
[4] C. Catal and B. Diri, "Investigating the effect of dataset size, metrics sets, and feature selection techniques on software fault prediction problem," *Inf. Sci.*, vol. 179, no. 8, pp. 1040–1058, 2009.
[5] F. A. Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino, "Comparing and experimenting machine learning techniques for code smell detection," *Empirical Software Engineering*, vol. 21, no. 3, pp. 1143–1191, 2016.
[6] F. Palomba, M. Zanoni, F. A. Fontana, A. D. Lucia, and R. Oliveto, "Smells like teen spirit: Improving bug prediction performance using the intensity of code smells," in *ICSME*, 2016, pp. 244–255.
[7] W. Ma, L. Chen, Y. Zhou, and B. Xu, "Do we have a chance to fix bugs when refactoring code smells?" in *SATE*, 2016, pp. 24–29.
[8] D. Radjenovic, M. Hericko, R. Torkar, and A. Zivkovic, "Software fault prediction metrics: A systematic literature review," *Information & Software Technology*, vol. 55, no. 8, pp. 1397–1418, 2013.
[9] A. Zeller, "Learning from 6,000 projects: Mining models in the large," in *SCAM*, 2010, pp. 3–6.
[10] F. Hermans, M. Pinzger, and A. van Deursen, "Detecting and visualizing inter-worksheet smells in spreadsheets," in *ICSE*, 2012, pp. 441–451.
[11] R. Abreu, J. Cunha, J. P. Fernandes, P. Martins, A. Perez, and J. Saraiva, "Smelling faults in spreadsheets," in *ICSME*, 2014, pp. 111–120.
[12] R. Abraham and M. Erwig, "UCheck: A spreadsheet type checker for end users," *J. Vis. Lang. Comput.*, vol. 18, no. 1, pp. 71–95, 2007.
[13] B. Hofer, A. Riboira, F. Wotawa, R. Abreu, and E. Getzner, "On the empirical evaluation of fault localization techniques for spreadsheets," in *FASE*, ser. LNCS, vol. 7793, 2013, pp. 68–82.

[14] G. Rothermel, L. Li, C. DuPuis, and M. M. Burnett, "What you see is what you test: A methodology for testing form-based visual programs," in *ICSE*, 1998, pp. 198–207.

[15] R. Abreu, J. Cunha, J. P. Fernandes, P. Martins, A. Perez, and J. Saraiva, "FaultySheet Detective: When smells meet fault localization," in *ICSME*, 2014, pp. 625–628.

[16] J. Cunha, J. P. Fernandes, H. Ribeiro, and J. Saraiva, "Towards a catalog of spreadsheet smells," in *ICCSA (4)*, ser. LNCS, vol. 7336, 2012, pp. 202–216.

[17] W. Dou, S. Cheung, C. Gao, C. Xu, L. Xu, and J. Wei, "Detecting table clones and smells in spreadsheets," in *SIGSOFT FSE*, 2016, pp. 787–798.

[18] P. W. Koch, B. Hofer, and F. Wotawa, "On the refinement of spreadsheet smells by means of structure information," *JSS*, vol. 147, pp. 64–85, 2019.

[19] C. Catal, "Software fault prediction: A literature review and current trends," *Expert Syst. Appl.*, vol. 38, no. 4, pp. 4626–4636, 2011.

[20] C. Chambers and M. Erwig, "Automatic detection of dimension errors in spreadsheets," *J. Vis. Lang. Comput.*, vol. 20, no. 4, pp. 269–283, 2009.

[21] W. Dou, S. Cheung, and J. Wei, "Is spreadsheet ambiguity harmful? detecting and repairing spreadsheet smells due to ambiguous computation," in *ICSE*, 2014, pp. 848–858.

[22] W. Dou, C. Xu, S. C. Cheung, and J. Wei, "CACheck: Detecting and repairing cell arrays in spreadsheets," *IEEE Trans. Software Eng.*, vol. 43, no. 3, pp. 226–251, 2017.

[23] L. Xu, S. Wang, W. Dou, B. Yang, C. Gao, J. Wei, and T. Huang, "Detecting faulty empty cells in spreadsheets," in *SANER*, 2018, pp. 423–433.

[24] R. Zhang, C. Xu, S. C. Cheung, P. Yu, X. Ma, and J. Lu, "How effectively can spreadsheet anomalies be detected: An empirical study," *J. Syst. Softw.*, vol. 126, pp. 87–100, 2017.

[25] R. Singh, B. Livshits, and B. Zorn, "Melford: Using neural networks to find spreadsheet errors," Microsoft, Tech. Rep. MSR-TR-2017-5, 2017.

[26] J. Cunha, J. P. Fernandes, P. Martins, J. Mendes, and J. Saraiva, "Smellsheet detective: A tool for detecting bad smells in spreadsheets," in *VL/HCC*, 2012, pp. 243–244.

[27] F. Hermans, M. Pinzger, and A. van Deursen, "Detecting code smells in spreadsheet formulas," in *ICSM*, 2012, pp. 409–418.

[28] D. W. Barowy, E. D. Berger, and B. G. Zorn, "Excelint: automatically finding spreadsheet formula errors," *PACMPL*, vol. 2, no. OOPSLA, pp. 148:1–148:26, 2018.

[29] A. Bregar, "Complexity metrics for spreadsheet models," *CoRR*, vol. abs/0802.3895, 2008.

[30] K. Hodnigg and R. T. Mittermeir, "Metrics-based spreadsheet visualization: Support for focused maintenance," *CoRR*, vol. abs/0809.3009, 2008.

[31] F. Hermans, M. Pinzger, and A. van Deursen, "Measuring spreadsheet formula understandability," *CoRR*, vol. abs/1209.3517, 2012.

[32] ——, "Detecting and refactoring code smells in spreadsheet formulas," *Emp. Softw. Engin.*, vol. 20, no. 2, pp. 549–575, 2015.

[33] T. Reschenhofer, B. Waltl, K. Shumaiev, and F. Matthes, "A conceptual model for measuring the complexity of spreadsheets," *CoRR*, vol. abs/1704.01147, 2017.

[34] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.

[35] N. V. Chawla, N. Japkowicz, and A. Kotcz, "Editorial: special issue on learning from imbalanced data sets," *SIGKDD Explorations*, vol. 6, no. 1, pp. 1–6, 2004.

[36] G. James, D. Witten, T. Hastie, and R. Tibshirani, *An introduction to statistical learning*. Springer, 2013.

[37] T. Schmitz and D. Jannach, "Finding errors in the enron spreadsheet corpus," in *VL/HCC*, 2016, pp. 157–161.

[38] F. Hermans and E. R. Murphy-Hill, "Enron's spreadsheets and related emails: A dataset and analysis," in *ICSE (2)*, 2015, pp. 7–16.

[39] E. Getzner, B. Hofer, and F. Wotawa, "Improving spectrum-based fault localization for spreadsheet debugging," in *QRS*, 2017, pp. 102–113.

[40] M. F. II and G. Rothermel, "The EUSES spreadsheet corpus: a shared resource for supporting experimentation with spreadsheet dependability mechanisms," *ACM SIGSOFT Softw. Eng. Notes*, vol. 30, no. 4, pp. 1–5, 2005.

[41] R. Abraham and M. Erwig, "Mutation operators for spreadsheets," *IEEE Trans. Software Eng.*, vol. 35, no. 1, pp. 94–108, 2009.

[42] Y. Lin and Y. Jeon, "Random forests and adaptive nearest neighbors," *J. Am. Stat. Assoc.*, vol. 101, no. 474, pp. 101–474, 2002.

[43] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone, *Classification and Regression Trees*. Wadsworth, 1984.

[44] C. Cortes and V. Vapnik, "Support-vector networks," *Machine Learning*, vol. 20, no. 3, pp. 273–297, 1995.

[45] R. E. Schapire, "A brief introduction to boosting," in *IJCAI*, 1999, pp. 1401–1406.

[46] I. J. Goodfellow, Y. Bengio, and A. C. Courville, *Deep Learning*. MIT Press, 2016.

[47] I. Guyon and A. Elisseeff, "An introduction to variable and feature selection," *Journal of Machine Learning Research*, vol. 3, pp. 1157–1182, 2003.

[48] L. E. Peterson, "K-nearest neighbor," *Scholarpedia*, vol. 4, no. 2, p. 1883, 2009.

[49] D. Jannach, A. Baharloo, and D. Williamson, "Toward an integrated framework for declarative and interactive spreadsheet debugging," in *ENASE*. SciTePress, 2013, pp. 117–124.

[50] P. W. Koch and K. Schekotihin, "Fritz: A tool for spreadsheet quality assurance," in *VL/HCC*. IEEE, 2018, pp. 285–286.

Patrick Koch is a Ph.D. candidate (Doctorate in Technical Sciences) at AAU Klagenfurt and scientific project assistant at Graz University of Technology. He received his Master's Degree in Software Development and Business Management from the Graz University of Technology, Austria, in 2016. His research area is focused on static analysis and AI-based techniques for debugging and quality assurance of spreadsheets.

Konstantin Schekotihin is an associate professor of Intelligent Systems at the AAU Klagenfurt, Austria. His research focus lies mainly on various aspects of knowledge representation and reasoning systems including machine learning, knowledge acquisition and maintenance, reasoning techniques as well as different applications such as configuration, planning, and recommendation.

Dietmar Jannach is a full professor of Information Systems at the AAU Klagenfurt, Austria. His main research area lies in the application of artificial intelligence technology to practical problems, with a particular focus on recommender systems in e-commerce, AI-based testing and debugging of spreadsheets, and on engineering problems of knowledge-intensive systems.

Birgit Hofer works as researcher at Graz University of Technology. She received a Ph.D. degree in Computer Science (2013) and a Master's degree (2009) from the same University. Her main research interests are automatic fault localization and correction in imperative and object-oriented software and spreadsheets, with a particular focus on spectrum-based fault localization, model-based debugging, and genetic programming approaches.

Franz Wotawa received an MSc and a PhD from Vienna University of Technology in 1994 and 1996 respectively. He is currently a professor of software engineering at Graz University of Technology. His research interests include model-based and qualitative reasoning, theorem proving, mobile robots, verification and validation, and software testing and debugging.