Noname manuscript No.
(will be inserted by the editor)

TuneR: A Framework for Tuning Software Engineering Tools with Hands-On Instructions in R

Markus Borg

Received: date / Accepted: date

Abstract Numerous tools automating various aspects of software engineering have been developed, and many of the tools are highly configurable through parameters. Understanding the parameters of advanced tools often require deep understanding of complex algorithms. Unfortunately, sub-optimal parameter settings limit the performance of tools and hinder industrial adaptation, but still few studies address the challenge of tuning software engineering tools. We present TuneR, an experiment framework that supports finding feasible parameter settings using empirical methods. The framework is accompanied by practical guidelines of how to use R to analyze the experimental outcome. As a proof-of-concept, we apply TuneR to tune ImpRec, a recommendation system for change impact analysis. Compared to the output from the default setting, we report a 20% improvement in the response variable. Moreover, TuneR reveals insights into the interaction among parameters, as well as non-linear effects. TuneR is easy to use, thus the framework has potential to support tuning of software engineering tools in both academia and industry.

Keywords software engineering tools · controlled experiment · parameter settings · configuration · recommendation system

1 Introduction

Tools that increase the level of automation in software engineering are often highly configurable through parameters. Examples of state-of-the-art tools that are highly configurable include EvoSuite for automatic test suite generation [37], FindBugs for static code analysis [7], and MyLyn, a task-oriented

Markus Borg
first address
Tel.: +123-45-678910
Fax: +123-45-678910
E-mail: fauthor@example.com

recommendation system in the Eclipse IDE [52]. However, the performance of these tools, as well as other tools providing decision support, generally depend strongly on the parameter setting used [11], often more so than the choice of the underlying algorithm [56]. The best parameter setting depends on the specific development context, and even within the same context it might change over time.

Finding feasible parameter settings is not an easy task. Automated tools in software engineering often implement advanced techniques such as genetic algorithms, dimensionality reduction, Information Retrieval (IR), and Machine Learning (ML). Numerous studies have explored how tool performance can be improved by tailoring algorithms and tuning parameters, for example in test data generation [38], test case selection [59], fault localization [2,90], requirements classification [28], and trace recovery [103,77]. We have previously published a systematic mapping study, highlighting the data dependency of IR-based trace recovery tools [18], and Hall *et al.* found the same phenomenon in a systematic literature review on bug prediction, stating that "models perform the best where the right technique has been selected for the right data, and these techniques have been tuned for the model rather than relying on default tool parameters" [41]. However, the research community cannot expect industry practitioners to have the deep knowledge required to fully understand the settings of advanced tools.

Feasible tuning of parameter settings is critical for successful transfer of Software Engineering (SE) tools from academia to industry. Unfortunately, apart from some work on Search-Based Software Engineering (SBSE) [34,5] there are few software engineering publications that specifically address parameter tuning. One could argue that academia should develop state-of-the-art tools, and the actual deployment in different organizations is simply a matter of engineering. However, we argue that practical guidelines for tuning SE tools, i.e., finding feasible parameter settings, are needed to support adaptation in industrial practise.

In this paper we discuss ImpRec [19], a Recommendation System for Software Engineering (RSSE) [81] developed to support Change Impact Analysis (CIA) in a company developing safety-critical software. ImpRec implements ideas from the area of Mining Software Repositories (MSR) to establish a semantic network of dependencies, and uses state-of-the-art IR to identify textually similar nodes in the network. The tool combines the semantic network and the IR system to recommend artifacts that are potentially impacted by an incoming issue report, and presents a ranked list to the developer. During development of the tool, we had to make several detailed design decisions, e.g., "how should distant artifacts in the system under study be penalized in the ranking function?" and "how should we weigh different artifact features in the ranking function to best reflect the confidence of the recommendations?". Answering such questions at design time is not easy. Instead we parametrized several decisions, a common solution that effectively postpones decisions to the tool user. We have deployed an early version of ImpRec in two pilot development teams to get feedback [23]. However, we did not want to force the

study participants to consider different parameter settings; instead we deployed ImpRec with a default setting based on our experiences. The question remains however; is the default setting close to the optimum?

We see a need for tuning guidelines for SE tools, to help practitioners and applied researchers to go beyond trial and pick-a-winner approaches. We suspect that three sub-optimal tuning strategies [32, pp. 211][71, pp. 4] dominate tuning of SE tools: 1) *ad hoc* tuning, 2) quasi-exhaustive search, and 3) Change One Parameter at a Time (COST) analysis. *Ad hoc tuning* might be a quick way to reach a setting, but non-systematic tuning increases the risk of deploying tools that do not reach their potential, therefore not being disseminated properly in industry. *Quasi-exhaustive search* might be possible if the evaluation does not require too much execution time, but it does not provide much insight in the parameters at play unless the output is properly analyzed. *COST analysis* is a systematic approach to tuning, but does not consider the effect of interaction between parameters.

We present TuneR, a framework for tuning parameters in automated software engineering tools. The framework consists of three phases: 1) Prepare Experiments, 2) Conduct Screening, and 3) Response Surface Methodology. The essence of the framework lies in space-filling and factorial design, established methods to structure experiments in DoCE and DoE, respectively. As a proof-of-concept, we apply TuneR to find a feasible parameter setting for ImpRec. For each step in TuneR, we present hands-on instructions of how to conduct the corresponding analysis using various packages for R [79]. Using TuneR we increase the accuracy of ImpRec's recommendations, with regard to the selected response variable, by 20%. We also validate the result by comparing the increased response to the outcome of a more exhaustive space-filling design.

The rest of this paper is structured as follows: Section 2 introduces the fundamental concepts in DoE and DoCE, and discusses how tuning of SE tools is different. Section 3 presents related work on finding feasible parameter setting for SE tools. In Section 4 we introduce ImpRec, the target of our tuning experiments. The backbone of the paper, the extensive presentation of TuneR, interweaved with the proof-of-concept tuning of ImpRec, is found in Section 5. In Section 6, we report from the exhaustive experiment on ImpRec parameter settings. Section 7 discusses our results, and presents the main threats to validity. Finally, Section 8 concludes the paper.

2 Background

This section introduces design of experiments, both of physical and simulated nature, and presents the terminology involved. Then we discuss how tuning of automated software engineering tools differ from traditional experiments. We conclude the section by reporting related work on experimental frameworks and parameter tuning in software engineering.

2.1 Design of Experiments

Design of Experiments (DoE) is a branch of applied statistics that deals with planning and analyzing controlled tests to evaluate the factors that affect the output of a process [71]. DoE is a mature research field, a key component in the scientific method, and it has proven useful for numerous engineering applications [45]. Also, DoE is powerful in commercialization, e.g., turning research prototypes into mature products ready for market release [74]. DoE is used to answer questions such as "what are the key factors at play in a process?, "how do the factors interact?", and "what setting gives the best output?".

We continue by defining the fundamental experimental terminology that is used throughout the paper. For a complete presentation of the area we refer to one of the available textbooks, e.g., Montgomery [71], Box et al. [24], and Dunn [32]. An *experiment* is a series of *experimental runs* in which changes are made to input variables of a system so that the experimenter can observe the output *response*. The input variables are called *factors*, and they can be either *design factors* or *nuisance factors*. Each design factor can be set to a specific *level* within a certain *range*. The nuisance factors are of practical significance for the response, but they are not interesting in the context of the experiment.

Dealing with nuisance factors is at the heart of traditional DoE. Nuisance factors are classified as *controllable*, *uncontrollable*, or *noise factors*. Controllable nuisance factors can be set by the experimenter, whereas uncontrollable nuisance factors can be measured but not set. Noise factors on the other hand can neither be controlled nor measured, and thus require more of the experimenter.

The cornerstones in the experimental design are *randomization*, *replication*, and *blocking*. Randomized order of the experimental runs is a prerequisite for statistical analysis of the response. Not randomizing the order would introduce a systematic bias into the responses. Replication means to conduct a repeated experimental run, independent from the first, thus allowing the experimenter to estimate the experimental error. Finally, blocking is used to reduce or eliminate the variability introduced by the nuisance factors. Typically, a block is a set of experimental runs conducted under relatively similar conditions.

Montgomery lists five possible goals of applying DoE to a process: 1) factor screening, 2) optimization, 3) confirmation, 4) discovery, and 5) robustness [71, pp. 14]. *Factor screening* is generally conducted to explore or characterize a new process, often aiming at identifying the most important factors. *Optimization* is the activity of finding levels for the design factors that produce the best response. *Confirmation* involves corroborating that a process behaves in line with existing theory. *Discovery* is a type of experiments related to factor screening, but the aim is to systematically explore how changes to the process affect the response. Finally, an experiment with a *robustness* goal tries to identify under which conditions the response substantially deteriorates. As the goal of the experiments conducted in this paper is to find the best re-

sponse for an automated software engineering tool by tuning parameters, i.e., optimization, we focus the rest of the section accordingly.

The traditional DoE approach to optimize a process involves three main steps: 1) factor screening to narrow down the number of factors, 2) using *factorial design* to study the response of all combinations of factors, and 3) applying *Response Surface Methodology* (RSM) to iteratively change the setting toward an optimal response [73]. Factorial design enables the experimenter to model the response as a *first-order model* (considering *main effects* and *interaction effects*), while RSM also introduces a *second-order* model in the final stage (considering also quadratic effects).

Different experimental designs have been developed to study how design factors affect the response. The fundamental design in DoE is a factorial experiment, an approach in which design factors are varied together (instead of one at a time). The basic factorial design evaluates each design factor at two levels each, referred to as a 2^k factorial design. Such a design with two design factors is represented by a square, where the corners represent the levels to explore in experimental runs (see A in Figure 1). When the number of design factors is large, the number of experimental runs required for a full factorial experiment might not be feasible. In a *fractional factorial experiment* only a subset of the experimental runs are conducted. Fractional factorial designs are common in practise, as all combinations of factors rarely need to be studied. The literature on fractional factorial designs is extensive, and we refer the interested reader to discussions by Montgomery [71] and Dunn [32].

All points in the experimental designs represent various levels of a design factor. In DoE, all analysis and model fitting are conducted in *coded units* instead of in *original units*. The advantage is that the model coefficients in coded units are directly comparable, i.e., they are dimensionless and represent the effect of changing a design factor over a one-unit interval [71, pp. 290]. We use 1 and -1 to represent the high and low level of a design factor in coded units.

Factorial design is a powerful approach to fit a first-order model to the response. However, as the response is not necessarily linear, additional experimental runs might be needed. The first step is typically to add a *center point* to the factorial design (cf. B in Figure 1). If quadratic effects are expected, e.g., indicated by experimental runs at the center point, the curvature needs to be better characterized. The most popular design for fitting a second-order model to the response is the *Central Composite Design* (CCD) [71, pp. 501] (cf. C in Figure 1). CCD complements the corners of the factorial design and the center point with axial points. A CCD is called *rotatable* if all points are at the same distance from the center point [53, pp. 50].

RSM is a sequential experimental procedure for optimizing a response (for a complete introduction we refer the reader to Myers' textbook [73]). In the initial optimization phase, RSM assumes that we operate at a point far from the optimum condition. To quickly move toward a more promising region of operation, the experimenter fits a first-order model to the response. Then, the operating conditions should be iteratively changed along the *path of steepest*

ascent. When the process reaches the region of the optimum, a second-order model is fitted to enable an analysis pinpointing the best point.

DoE has been a recommended practise in software engineering for decades. The approaches have been introduced in well-cited software engineering textbooks and guidelines, e.g., Basili et al. [10], Pfleeger [78], and Wohlin et al. [100]. However, tuning an automated software engineering tool differs from traditional experiments in several aspects, as discussed in the rest of this section.

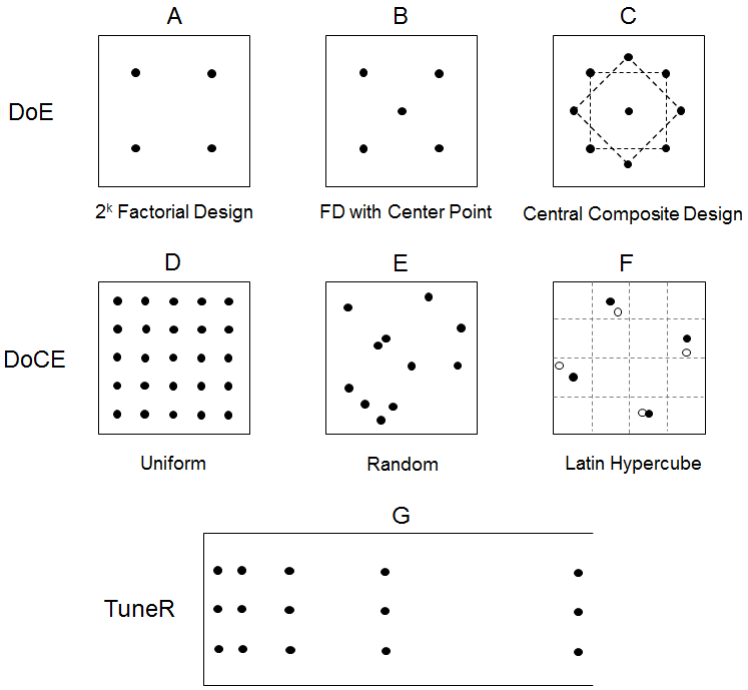

Fig. 1 Overview of experimental designs for two factors. Every point represents an experimental setting.

2.2 Design of Computer Experiments

DoE was developed for experiments in the physical world, but nowadays a significant amount of experiments are instead conducted as computer simulation models of physical systems, e.g., during product development [84]. Exploration using computer simulations shares many characteristics of physical experiments, e.g., each experimental run requires input levels for the design factors and results in one or more responses that characterize the process under study. However, there are also important differences between physical

experiments and experiments in which the underlying reality is a mathematical model explored using a computer.

Randomization, replication, and blocking, three fundamental components of DoE, were all introduced to mitigate the random nature of physical experiments. Computer models on the other hand, unless programmed otherwise, generate deterministic responses with no random error [96]. While the deterministic responses often originate from highly complex mathematical models, repeated experimental runs using the same input data generates the same response, i.e., replication is not required. Neither does the order of the experimental runs need to be randomized, nor is blocking needed to deal with nuisance factors. Still, assessing the relationship between the design factors and the response in a computer experiment is not trivial, and both the design and analysis of the experiment need careful thought.

Design of Computer Experiments (DoCE) focuses on *space-filling designs*. Evaluating only two levels of a design factor, as in a 2^k factorial design, might not be appropriate when working with computer models, as it can typically not be assumed that the response is linear [33, pp. 11]. Instead, interesting phenomena can potentially be found in all regions of the experimental space [71, pp. 524]. The simplest space-filling designs are *uniform design* (cf. D in Figure 1), in which all design points are spread evenly, and *random design* (cf. E in Figure 1). Another basic space-filling design is the *Latin Hypercube design*. A two-factor experiment has its experimental points in a latin square if there is only one point in each row and each column (cf. F in Figure 1), in line with the solution to a sudoku puzzle. A Latin hypercube is the generalization to an arbitrary number of dimensions. Latin Hypercubes can be combined with randomization to select the specific setting in each cell, as represented by white points in Figure 1.

Also RSM needs adaptation for successful application to computer experiments. There are caveats that need to be taken into consideration when transferring RSM from DoE to DoCE. Vining highlights that the experimenter need some information about starting points, otherwise there is a considerable risk that RSM ends up in a local optimum [96]. Moreover, bumpy response surfaces, which computer models might generate, pose difficulties for optimization. Consequently, a starting point for RSM should be in the neighborhood of an acceptable optimum. Finally, RSM assumes that there should be only few active design factors. Vining argues that both starting points and the number of design factors should be evaluated using screening experiments [96], thus screening is emphasized as a separate phase in TuneR.

2.3 Tuning Automated Software Engineering Tools

DoE evolved to support experiments in the physical world, and DoCE was developed to support experiments on computer models of physical phenomena. The question whether software is tangible or intangible is debated from both philosophical and juridical perspectives (see e.g., [72,13]), but no matter

what, there are differences between software and the entities that are typically explored using DoE and DoCE. Furthermore, in this paper we are interested in using experiments for tuning[1] a special type of software: tools for automated software engineering. We argue that there are two main underlying differences between experiments conducted to tune automated SE tools and traditional DoCE. First, automated SE tools are not computer models of anything in the physical world. Thus, we often cannot relate the meaning of various parameter setting to characteristics that are easily comprehensible. In DoCE however, we are more likely to have a pre-understanding of the characteristics of the underlying physical phenomenon. Second, a tuned automated SE tool is not the primary deliverable, but a means to an end. An automated SE tool is intended to either improve the software under development, or to support the ongoing development process [35]. In DoCE on the other hand, the simulation experiments tend to be conducted on a computer model of the product under development or the phenomenon under study.

Consequently, an experimenter attempting to tune an automated SE tool must consider some aspects that might be less applicable to traditional DoCE. The experimenter should be prepared for unexpected responses in all regions of the experiment space, due to the lack of connection between parameters and physical processes. Parameter ranges resulting in feasible responses might exist anywhere in the experiment space, thus some variant of space-filling designs need to be applied as in DoCE. However, responses from automated SE tools cannot be expected to behave linearly, as the response might display sudden steps in the response or asymptotic behavior. While certain peculiarities might arise also when calibrating physical processes, we believe that they could be more common while tuning automated SE tools. Other aspects that must be taken into consideration are execution time and memory consumption. An SE tool is not useful if it cannot deliver its output in a reasonable amount of time, and it should be able to do so with the memory available in the computers of the target environment.

When tuning an automated SE tool, we propose that it should be considered a black-box model (also recommended by Arcuri and Fraser [5]). We define a black-box model, inspired by Kleijnen, as "a model that transforms observable input into observable outputs, whereas the values of internal variables and specific functions of the tool implementation are unobservable" [53, pp. 16]. For any reasonably complex SE tool, we suspect that fully analyzing how all implementation details affect the response is likely to be impractical. However, when optimizing a black-box model we need to rely on heuristic approaches, as we cannot be certain whether an identified optimum is local or global. An alternative to heuristic approaches is to use metaheuristics (e.g., genetic algorithms, simulated annealing, or tabu search [4]), but such approaches require extensive tuning themselves.

[1] Adjusting parameters of a system is known as *calibration* when they are part of a physical process, otherwise the activity is called *tuning* [58].

The main contribution of this paper is *TuneR*, a heuristic experiment framework for tuning automated SE tools using R. TuneR uses a space-filling design to screen factors of a black-box SE, uniform for bounded parameters and a geometric sequence for unbounded parameters as shown in Figure 1. Once a promising region for the parameter setting has been identified, TuneR attempts to apply RSM to find a feasible setting. We complement the presentation of TuneR with a hands-on example of how we used it to tune the RSSE ImpRec.

3 Related Work on Parameter Tuning in Software Engineering

Several researchers have published papers on parameter tuning in software engineering. As the internals of many tools for automated SE involve advanced techniques, such as computational intelligence and machine learning, academic researchers must provide practical guidelines to support knowledge transfer to industry. In this section we present some of the most related work on tuning automated SE tools. All tools we discuss implement metaheuristics to some extent, a challenging topic covered by Birattari in a recent book [16]. He reports that *most tuning of metaheurstics is done by hand and by rules of thumb*, showing that such tuning is not only an issue in SE.

Parameter tuning is fundamental in Search-Based Software Engineering (SBSE) [5,38]. As SBSE is based on metaheuristics, its performance is heavily dependent on context-specific parameter settings. However, some parameters can be set based on previous knowledge about the problem and the software under test. Fraser and Arcuri refer to this as seeding, i.e., "any technique that exploits previous related knowledge to help solve the testing problem at hand" [38]. They conclude that *seeding is valuable in tuning SBSE tools*, and present empirical evidence that the more domain specific information that can be included in the seeding, the better the performance will be. In line with the recommendations by Fraser and Arcuri, we emphasize the importance of pre-understanding by including it as a separate step in TuneR.

Arcuri and Fraser recently presented an empirical analysis on how their tool EVOSUITE, a tool for test data generation, performed using different parameter settings [5]. Based on more than one million experiments, they show that *different settings cause very large variance* in the performance of their EVOSUITE, but also that *"default" settings presented in the literature result in reasonable performance*. Furthermore, they find that *tuning EVOSUITE using one dataset and then applying it on others brings little value*, in line with the No Free Lunch theorem by Wolpert [101]. Finally, they applied RSM to tune the parameters of EVOSUITE, but conclude that *RSM did not lead to improvements compared to the default parameter setting*. Arcuri and Fraser discuss the unsuccessful outcome of their attempt at RSM, argue that it should be treated as inconclusive rather than a negative result, and call for more studies on tuning in SE. Their paper is concluded by general guidelines on how to tune parameters. However, the recommendations are on a high-level,

limited to a warning on over-fitting, and advice to partition data into non-overlapping training and test sets. The authors also recommend using 10-fold cross-validation in case only little data is available for tuning purposes. Our work on TuneR complements the recommendations from Arcuri and Fraser, by providing more detailed advice on parameter tuning. Also, there is no conflict between the two sets of recommendations, and it is possible (and recommended) to combine our work with for example 10-fold cross validation.

Da Costa and Schoenauer also worked on parameter tuning in the field of software testing. They developed the software environment GUIDE to help practitioners use evolutionary computation to solve hard optimization problems [29]. GUIDE contains both an easy-to-use GUI, and support for parameter tuning. GUIDE has been applied to evolutionary software testing in three companies including Daimler. However, the parameter tuning offered by GUIDE is aimed for algorithms in the internal evolution engine, and not for external tools.

Biggers *et al.* highlighted that there are few studies on how to tune tools for feature location using text retrieval, and argue that it impedes deployment of such tool support [14]. They conducted a comprehensive study on the effects of different parameter settings when applying feature location using Latent Dirichlet Allocation (LDA). Their study involved feature location from six open source software systems, and they particularly discuss configurations related to indexing the source code. Biggers *et al.* report that *using default LDA settings from the literature on natural language processing is suboptimal in the context of source code retrieval.*

Thomas *et al.* addressed tuning of automated SE tools for fault localization [90]. They also emphasize the research gap considering tuning of tools, and acknowledge the challenge of finding a feasible setting for a tool using supervised learning. The paper reports from a large empirical study on 3,172 different classifier configurations, and show that *the parameter settings have a significant impact on the tool performance.* Also, Thomas *et al.* shows that ensemble learning, i.e., combining multiple classifiers, provides better performance than the best individual classifiers. However, design choices related to the combination of classifiers also introduce additional parameter settings [49].

Lohar *et al.* discussed different configurations for SE tools supporting trace retrieval, i.e., automated creation and maintenance of trace links [61]. They propose a machine learning approach, referred to as Dynamic Trace Configuration (DTC), to search for the optimal configuration during runtime. Based on experiments with data extracted from three different domains, they show that DTC can significantly improve the accuracy of their tracing tool. Furthermore, the authors argue that DTC is easy to apply, thus supporting technology transfer. However, in contrast to TuneR, DTC is specifically targeting SE tools for trace retrieval.

ImpRec, the tool we use for the proof-of-concept evaluation of TuneR, is a type of automated SE tool that presents output as a ranked list of recommendations, analogous to well-known IR systems for web search. Modern search engines apply ranking functions that match the user and his query with web

pages based on hundreds of features, e.g., location, time, search history, query content, web page title, content, and domain [102]. To combine the features in a way that yields relevant search hits among the top results, i.e., to tune the feature weighting scheme, Learning-to-Rank (LtR) is typically used in state-of-the-art web search [63]. LtR is a family of machine learning approaches to obtain feasible tuning of IR systems [60]. Unfortunately, applying LtR to the ranking function of ImpRec is not straightforward. The success of learning-to-rank in web search is enabled by enormous amounts of training data [85], manually annotated for relevance by human raters [95]. As such amounts of manually annotated training data is not available for ImpRec, and probably not for other automated SE tools either, TuneR is instead based on empirical experimentation. However, LtR is gaining attention also in SE, as showed by a recent position paper by Binkley and Lawrie [15].

4 ImpRec: An RSSE for Automated Change Impact Analysis

ImpRec is an SE tool that supports navigation among software artifacts [19], tailored for a development organization in the power and automation sector. The development context is safety-critical embedded development in the domain of industrial control systems, governed by IEC 61511 [46] and certified to a Safety Integrity Level (SIL) of 2 as defined by IEC 61508 [47]. The target system has evolved over a long time, the oldest source code was developed in the 1980s. A typical development project in the organization has a duration of 12-18 months and follows an iterative stage-gate project management model. The number of developers is in the magnitude of hundreds, distributed across sites in Europe, Asia and North America.

As specified in IEC 61511, the impact of proposed software changes should be analyzed before implementation. In the case company, the impact analysis process is integrated in the issue repository. Before a corrective change is made to resolve an issue report, the developer must store an impact analysis report as an attachment to the corresponding issue report. As part of the impact analysis, engineers are required to investigate the impact of a change, and document their findings in an impact analysis report according to a project specific template. The template is validated by an external certifying agency, and the impact analysis reports are internally reviewed and externally assessed during safety audits.

Several questions explicitly ask for trace links [17], i.e., "a specified association between a pair of artifacts" [39]. The engineer must specify source code that will be modified (with a file-level granularity), and also which related software artifacts need to be updated to reflect the changes, e.g., requirement specifications, design documents, test case descriptions, test scripts and user manuals. Furthermore, the impact analysis should specify which high-level system requirements cover the involved features, and which test cases should be executed to verify that the changes are correct, once implemented in the system. In the target software system, the extensive evolution has created a

complex dependency web of software artifacts, thus the impact analysis is a daunting work task.

ImpRec is an RSSE that enables reuse of knowledge captured from previous impact analyses [17]. Using history mining in the issue repository, a collaboratively created trace link network is established, referred to as the knowledge base. ImpRec then calculates the centrality measure of each artifact in the knowledge base. When a developer requests impact recommendations for an issue report, ImpRec combines IR and network analysis to identify *candidate impact*. First, Apache Lucene [67] is used to search for issue reports in the issue repository that are textually similar. Then, originating from the most similar issue reports, trace links are followed both to related issue reports and to artifacts that were previously reported as impacted. When a set of candidate impact has been identified, they are ranked according to a ranking function, and presented to developers in a ranked list in the ImpRec GUI. For further details on ImpRec, we refer to our previous publications [19, 23].

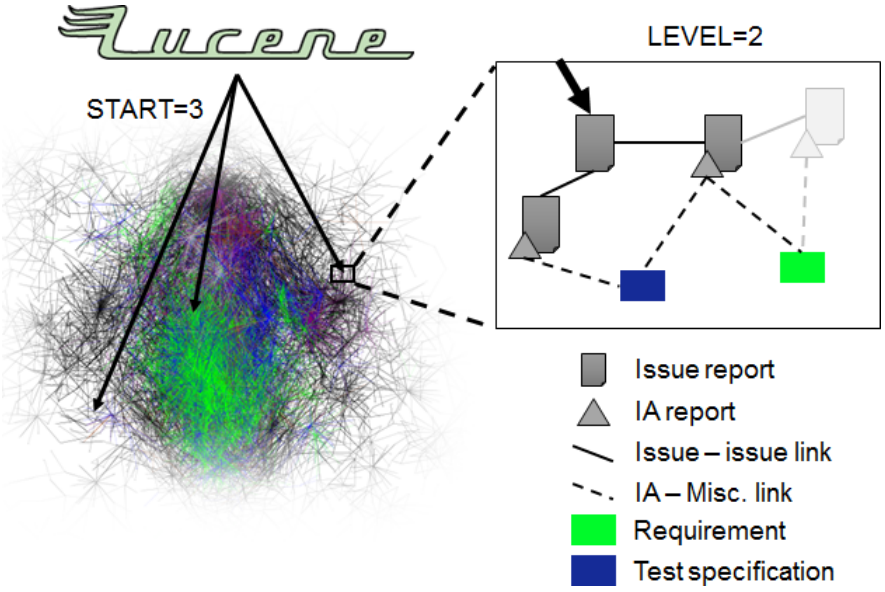

Fig. 2 Identification of candidate impact using ImpRec. Two related parameters (with an example setting) are targeted for tuning: 1) The number of starting points identified using Apache Lucene ($START$), and 2) the maximum number of issue-issue links followed to identify impacted artifacts ($LEVEL$).

This paper presents our efforts to tune four ImpRec parameters, two related to candidate impact identification, and two dealing with ranking of the candidate impact. Fig. 2 presents an overview of how ImpRec identifies candidate impact, and introduces the parameters $START$ and $LEVEL$. By setting the two parameters to high values, ImpRec identifies a large set of candidate impact. To avoid overwhelming the user with irrelevant recommendations, the

artifacts in the set are ranked. ImpRec assigns all artifacts a weight according to the following ranking function, introducing the other two parameters:

$$Weight = \frac{ALPHA * text\_sim + (1 - ALPHA) * node\_cent}{1 + links * PENALTY} \quad (1)$$

where $PENALTY$ is used to penalize distant artifacts and $ALPHA$ is used to set the relative importance of textual similarity and the centrality measure. $text\_sim$ is the similarity score of the corresponding starting point provided by Apache Lucene, $node\_cent$ is the centrality measure of the artifact in the knowledge base, and $links$ is the number of issue-issue links followed to identify the artifact (no more than $LEVEL-1$). The rest of this paper presents TuneR, and how we used it to tune $START$, $LEVEL$, $PENALTY$, and $ALPHA$.

5 TuneR: An Experiment Framework and A Hands-on Example

This section describes the three phases of TuneR, covering 11 steps. For each step in our framework, we first describe TuneR in general terms, and then we present a hands-on example of how we tuned ImpRec. Figure 3 shows an overview of the steps in TuneR.

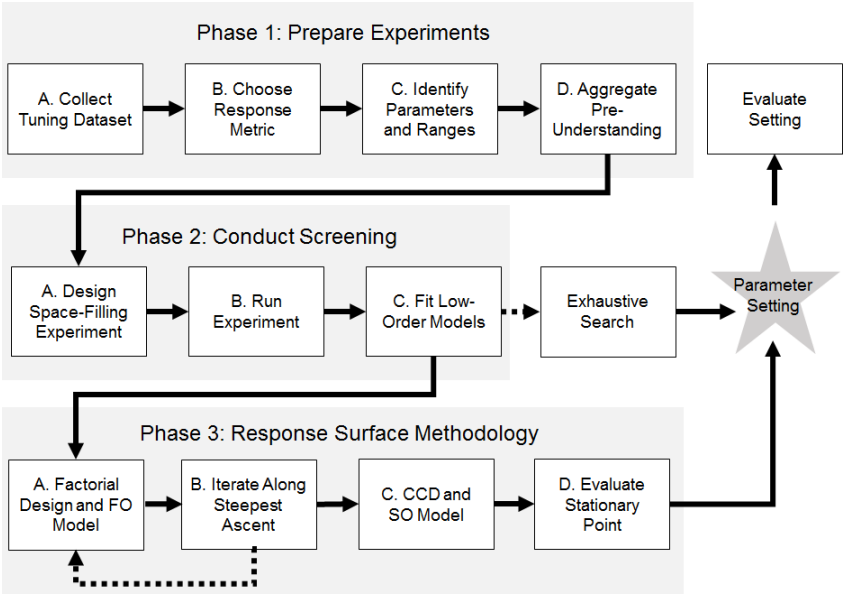

Fig. 3 Overview of TuneR. The three phases are depicted in gray boxes. Dotted arrows show optional paths.

5.1 Phase 1: Prepare Experiments

Successful experimentation relies on careful planning. The first phase of TuneR consists of four steps: A) Collect Tuning Dataset, B) Choose Response Metric, C) Identify Parameters and Ranges, and D) Aggregate Pre-Understanding. All four steps are prerequisites for the subsequent Screening phase.

5.1.1 A) Collect Tuning Dataset

Before any tuning can commence, a dataset that properly represents the target environment must be collected. The *content validity* of the dataset refers to the representativeness of the sample in relation to all data in the target environment [94]. Thus, to ensure high content validity in tuning experiments, the experimenter must carefully select the dataset, and possibly also sample from it appropriately, as discussed by Seiffert et al. [86]. Important decisions that have to be made at this stage include *how old data can be considered valid* and *whether the data should be pre-processed* in any way. While a complete discussion on data collection is beyond the scope of TuneR, we capture some of the many discussions on how SE datasets should be sampled and pre-processed in this section.

In many software development projects, the characteristics of both the system under development [68] and the development process itself [64] vary considerably. If the SE tool is intended for such a dynamic target context, then it is important that the dataset does not contain obsolete data. For example, Shepperd *et al.* discuss the dangers of using old data when estimating effort in software development, and the difficulties in knowing when data turns obsolete [87]. Jonsson *et al.* show the practical significance on time locality in automated issue assignment [49], i.e., how quickly the prediction accuracy deteriorates with old training data for some projects.

Preprocessing operations, such as data filtering, influence the performance of SE tools. Menzies *et al.* even warn that variation in preprocessing steps might be a major cause of conclusion instability when evaluating SE tools [69]. Shepperd *et al.* discuss some considerations related to previous work on publicly available NASA datasets, and conclude that the importance of preprocessing in general has not been acknowledged enough [88]. Regarding filtering of datasets, Lamkanfi and Demeyer show how filtering outliers can improve prediction of issue resolution times [55], a finding that has also been confirmed by AbdelMoez et al. [1]. Thus, if the SE tools will be applied to filtered data, then the dataset used for the tuning experiment should be filtered as well. Another threat to experimentation with tools implementing machine learning is the *dataset shift* problem, i.e., the distribution of data in the training set differs from the test set. Turhan discuss how dataset shift relate to conclusion instability in software engineering prediction models, and presents strategies to alleviate it [93].

The tuning dataset does not only need to contain valid data, it also needs to contain enough of it. A recurring approach in SE is to evaluate tools on

surrogate data, e.g., studying OSS development and extrapolating findings to proprietary contexts. Sometimes it is a valid approach, as Robinson and Francis have shown in a comparative study of 24 OSS systems and 21 proprietary software systems [82]. They conclude that the variation *among* the two categories is as big as *between* them, and, at least for certain software metrics, that there often exist OSS systems with characteristics that match proprietary systems. Several SE experiments use students as subjects, and Höst *et al.* show that it is a feasible approach under some circumstances [44]. However, the validity of experimenting on data collected from student projects is less clear, as discussed in our previous survey [22]. Another option is to combine data from various sources, i.e., complementing proprietary data from different contexts. Tsunoda and Ono recently highlighted some risks of this approach, based on a cross-company software maintenance dataset as an example [92]. They performed a statistical analysis of the dataset, and exemplified how easy it is to detect spurious relationships between totally independent data.

As ImpRec was developed in close collaboration with industry, and is a tool tailored for a specific context, the data used for tuning must originate from the same environment. We extracted all issue reports from the issue repository, representing 12 years of software evolution in the target organization [19]. As the issue reports are not independent, the internal order must be kept and we cannot use an experimental design based on cross-validation. Thus, as standard practise in machine learning evaluation, and emphasized by Arcuri and Fraser [5], we split the ordered data into non-overlapping training and test sets, as presented in Fig. 4. Instead, the training set was used to establish the knowledge base, and the test set was used to measure the ImpRec performance. The experimental design used to tune ImpRec is an example of *simulation* as presented by Walker and Holmes [98], i.e., we simulate the historical inflow of issue reports to measure the ImpRec response. Before commencing the tuning experiments, we analyzed whether the content of the issue reports had changed significantly over time. Also, we discussed the evolution of both the software under development, and the development processes, with engineers in the organization. We concluded that we could use the full dataset for our experiments, and we chose to not filter the dataset in any way.

5.1.2 B) Choose Response Metric

The next step in TuneR is to choose what metric to base the tuning on. TuneR is used to optimize a response with regard to a single metric, as it relies on traditional RSM, thus the response metric needs to be chosen carefully. Despite mature guidelines like the *Goal-Question-Metric* framework [9], the dangers of software measurements have been emphasized by several researchers [31, 50, 26]. However, we argue that selecting a metric for the response of an SE tool is a far more reasonable task than measuring the entire software development process based on a single metric. A developer of an SE tool probably already knows the precise goal of the tool, and thus should be able to choose or invent a feasible metric. Moreover, if more than one metric is important to the response,

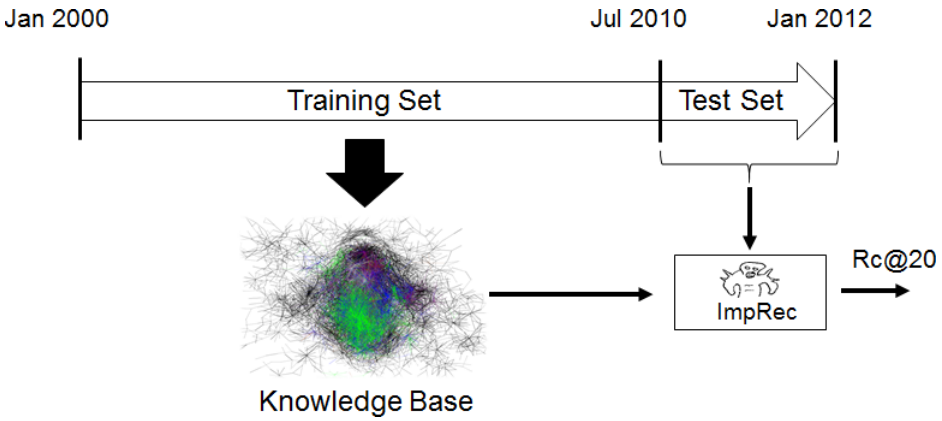

Fig. 4 Composition of the ImpRec tuning dataset into training and test sets. The knowledge base is established using issue reports from Jan 2000 to Jul 2010. The subsequent issue reports are used to simulate the ImpRec response, measured in Rc@20.

the experimenter can introduce a compound metric, i.e., a combination of individual metrics. On the other hand, no matter what metric is selected, there is a risk that naively tuning with regard to the specific metric leads to a sub-optimal outcome, a threat further discussed in Section 5.4.

Regarding the tuning of ImpRec, we rely on the comprehensive research available on quantitative IR evaluation, e.g., the TREC conference series and the Cranfield experiments [97]. In line with general purpose search systems, ImpRec presents a ranked list of candidates for the user to consider. Consequently, it is convenient to measure the quality of the output using established IR measures for ranked retrieval. The most common way to evaluate the effectiveness of an IR system is to measure precision and recall. Precision is the fraction of retrieved documents that are relevant, while recall is the fraction of relevant documents that are retrieved. As there is a trade-off between precision and recall, they are often reported pairwise. The pairs are typically considered at fixed recall levels (e.g., 0.1 ... 1.0), or at specific cut-offs of the ranked list (e.g., the top 5, 10, or 20 items) [66].

We assume that a developer is unlikely to browse too many recommendations from ImpRec. Consequently, we use a cut-off point of 20 to disregard all recommendations below that rank. While it is twice as many as the standardized page-worth output from search engines, CIA is a challenging task in which practitioners request additional tool support [30, 23], and thus we assume that engineers are willing to browse additional search hits. Also, we think that engineers can quickly filter out the interesting recommendations among the top 20 hits.

Several other measures for evaluating the performance of IR systems have been defined. A frequent compound measure is the F-score, a harmonized mean of precision and recall. Other more sophisticated metrics include Mean Average Precision (MAP) and Normalized Discounted Cumulative Gain (NDCG) [66].

However, for the tuning experiments in this paper, we decided to optimize the response wrt. *recall considering the top-20 results* (Rc@20).

5.1.3 C) Identify Parameters and Specify Ranges for Normal Operation

The third step of Phase 1 in TuneR concerns identification of parameters to vary during the tuning experiments. While some parameters might be obvious, maybe as explicit in settings dialogs or configuration files, other parameters can be harder to identify. Important variation points may be hidden in the implementation of the SE tool, thus identifying what actually constitutes a meaningful parameter can be challenging.

Once the parameters have been identified, the experimenter needs to decide what levels should be used. A first step is, in line with standard DoE practise [32, pp. 213], to identify what range represents "normal operation" of each parameter. Parameter variations within such a range should be large enough to cause changes in the response, but the range should not cover so distant values that the fundamental characteristics of the tool are altered. For some parameters, identification of the normal range is straightforward because of well-defined bounds, e.g., a real value between 0 and 1 or positive integers between 1 and 10. For other parameters, however, it is possible that neither the bounds nor even the sign is known. Parameters can also be binary or categorical, taking discrete values [32, pp. 208].

Regarding ImpRec, Section 4 already presented the four parameters $ALPHA$, $PENALTY$, $START$, and $LEVEL$. However, also the search engine library Apache Lucene is highly configurable. But as configuring Lucene is complex, see for example McCandless *et al.* [67, Ch. 2], and since the default setting yielded useful results in our previous work on issue reports [21], we choose to consider it as a black box with fixed parameters in this study, i.e., we use the default setting. The other four parameters of ImpRec on the other hand, do not have any default values, thus we must continue by specifying ranges for normal operation.

Table 5.1.3 shows how we specify the ranges for normal operation for the four parameters. $ALPHA$ represents the relative importance between textual similarities and centrality measures, i.e., it is a bounded real value between 0 and 1, and we consider the full range normal. $START$ is a positive integer, there must be at least one starting point, but there is no strict upper limit. We consider 200 to be the upper limit under normal operation, as we suspect larger values to generate imprecise recommendations and too long response times. $LEVEL$ and $PENALTY$ both deal with following links between issue reports in the knowledge base. Analogous to the argumentation regarding $START$, we suspect that assigning $LEVEL$ a too high value might be counter-productive. $LEVEL$ must be a positive integer, as 1 represents not following any issue-issue links at all. We decide to consider [0, 10] as the range for $LEVEL$ under normal operation. $PENALTY$ down-weights potential impact that has been identified several steps away in the knowledge base, i.e., impact with a high level. The parameter can be set to any non-negative number, but we assume

Parameter	Type of range	Normal range
$ALPHA$	Non-negative bounded continuous	[0-1]
$PENALTY$	Non-negative continuous	[0-5]
$START$	Positive discrete	[1-200]
$LEVEL$	Positive discrete	[1-10]

Table 1 The four parameters studied in the tuning experiment, and the values that represent their range for normal operation.

that a value between 0 and 5 represents normal operation. Already the level 5 would make the contribution of distant issue reports practically zero, see Equation 1.

5.1.4 D) Aggregate Pre-Understanding

Successful tuning of an SE tool requires deep knowledge. The experimenter will inevitably learn about the tool in the next two phases of TuneR, but probably there are already insights before the experimentation commences. In line with Gummesson's view [40, pp. 75], we value this *pre-understanding* as fundamental to reach deep understanding. The pre-understanding can provide the experimenter with a shortcut to a feasible setting, as it might suggest in what region the optimal setting is located. To emphasize this potential, TuneR consists of a separate step aimed at recapitulating what has already been experienced.

The development of ImpRec was inspired by test-driven development [12], thus we tried numerous different parameter settings. By exploring different settings in our trial runs during development, an initial parameter tuning evolved as a by-product of the tool development. While we performed this experimentation in an *ad hoc* fashion, we measured the output with regard to Rc@20, and recorded the results in a structured manner. Recapitulating our pre-understanding regarding the parameters provides the possibility to later validate the outcome of the screening in Phase 2 of TuneR.

The *ad hoc* experiments during development contain results from about 100 trial runs. We explored $ALPHA$ ranging from 0.1 to 0.9, obtaining the best results for high values. START had been varied between 3 and 20, and again high values appeared to be a better choice. Finally, we had explored LEVEL between 3 and 10, and PENALTY between 0 and 8. Using a high LEVEL and low PENALTY yielded the best results. Based on our experiences, we deployed ImpRec in the organization using the following default setting: $ALPHA = 0.83, START = 17, LEVEL = 7, PENALTY = 0.2$ (discussed in depth in another paper [23]). The default setting yield a response of Rc@20=0.41875, i.e., about 40% of the true impact is delivered among the top-20 recommendations. We summarize our expectations as follows:

- The ranking function should give higher weights to centrality measures than textual similarity ($0.75 < ALPHA < 1$)
- Many starting points benefit the identification of impact ($START > 15$)

- Following related cases several steps away from the starting point improves results ($LEVEL > 5$)
- We expect an interaction between $LEVEL$ and $PENALTY$, i.e., that increasing the number of levels to follow would make penalizing distant artifacts more important
- Completing an experimental run takes about 10-30 s, depending mostly on the value of $START$.

5.2 Phase 2: Conduct Screening Experiment

Phase 2 in TuneR constitutes three steps related to screening. Screening experiments are conducted to identify the most important parameters in a specific context [73, pp. 6] [32, pp. 240]. Traditional DoE uses 2^k factorial design for screening, using broad values (i.e., high and low values within the range of normal operation) to calculate main effects and interaction effects. However, as explained in Section 2.3, space-filling design should be applied when tuning SE tools. The three screening steps in TuneR are: A) Design Space-Filling Experiment, B) Run Experiment, and C) Fit Low-Order Models. Phase 2 concludes by identifying a *promising region*, i.e., a setting that appears to yield a good response, a region that is used as input to Phase 3.

5.2.1 A) Design a space-filling experiment

The first step in Phase 2 in TuneR deals with designing a space-filling screening experiment. The intention of the screening is not to fully analyze how the parameters affect the response, but to complement the less formal pre-understanding. Still, the screening experiment will consist of multiple runs. As a rule of thumb, Levy and Steinberg approximate that the number of experimental runs needed in a DoCE screening is ten times the number of parameters involved [58].

Several aspects influence the details of the space-filling design, and we discuss four considerations below. First, parameters of different types (as discussed in Phase 1, Step 2) require different experimental settings. The space of categorical parameters can only be explored by trying all levels. Bounded parameters on the other hand can be explored using uniform space-filling designs as presented in Section 2.2. Unbounded parameters however, at least when the range of normal operation is unknown, requires the experimenter to select values using other approaches. Second, our pre-understanding from Phase 1, Step 4 might suggest that some parameters are worth to study using more fine-granular values than others. In such cases, the pre-understanding has already contributed with a preliminary sensitivity analysis [83, pp. 189], and the design should be adjusted accordingly. Third, the time needed to perform the experiments limits the number of experimental runs, in line with discussions on search budget in SBSE [42]. Certain parameter settings might require longer execution times than others, and thus require a disproportional amount

of the search budget. Fourth, there might be known constraints at play, forcing the experimenter to avoid certain parameter values. This phenomenon is in line with the discussion on non-safe settings in DoE [32, pp. 254].

Unless the considerations above suggest special treatment, we propose the following rules-of-thumb as a starting point:

- Restrict the search budget for the screening experiment to a maximum of 48 h, i.e., it should not require more than a weekend to execute.
- Use the search budget to explore the parameters evenly, i.e., for an SE tool with i parameters, and the search budget allows n experimental runs, use $\sqrt[i]{n}$ values for each parameter.
- Apply a uniform design for bounded parameters, i.e., spread the parameter values evenly.
- Use a geometric series of values for unbounded parameters, e.g., for integer parameters explore values $2^i,\ i = 0,\ 1,\ 2,\ 4,\ 8$ ...

When screening the parameters of ImpRec, we want to finish the experimental runs between two workdays (4 PM to 8 AM, 16 h) to enable an analysis of the results on the second day. Based on our pre-understanding, we predict that on average four experimental runs can be completed per minute, thus about 3,840 experimental runs can be completed within the 16 h search budget. As we have four parameters, we can evaluate about $\sqrt[4]{3,840} \approx 7.9$ values per parameter, i.e., 7 rounded down.

Table 5.2.1 shows the values we choose for screening the parameters of ImpRec. $ALPHA$ is a relative weighting parameter between 0 and 1. We use a uniform design to screen $ALPHA$, but do not pick the boundary values to avoid divisions by zero. $PENALTY$ is a positive continuous variable with no upper limit, and we decide to evaluate several magnitudes of values. A penalty of 8 means that the contribution of distant artifacts to the ranking function is close to zero, thus we do not need to try higher values. $START$ and $LEVEL$ are both positive discrete parameters, both dealing with how many impact candidates should be considered by the ranking function. Furthermore, our pre-understanding shows that the running time is proportional to the value of $START$. As we do not know how high values of $START$ are feasible, we choose to evaluate up to 512, a value that represents about 10% of the full dataset. Exploring such high values for $LEVEL$ does not make sense, as there are no such long chains of issue reports. Consequently, we limit $LEVEL$ to 64, already a high number. In total, this experimental design, constituting 3,430 runs, appears to be within the available search budget.

5.2.2 B) Run Screening Experiment

When the design of the screening experiment is ready, the next step is to run the experiment. To enable execution of thousands of experimental runs, a stable experiment framework for automatic execution must be developed. Several workbenches are available that enable reproducible experiments, e.g., frameworks such as Weka [36] and RapidMiner [43] for general purpose data

Parameter	#Levels	Values
ALPHA	7	0.01, 0.17, 0.33, 0.5, 0.67, 0.83, 0.99
PENALTY	7	0.01, 0.1, 0.5, 1, 2, 4, 8
START	10	1, 2, 4, 8, 16, 32, 64, 128, 256, 512
LEVEL	7	1, 2, 4, 8, 16, 32, 64

Table 2 Screening design for the four parameters *ALPHA*, *PENALTY*, *START*, and *LEVEL*.

mining, and SE specific efforts such as the TraceLab workbench [51] for traceability experiments, and the more general experimental Software Engineering Environment (eSSE) [91]. Furthermore, the results should be automatically documented as the experimental runs are completed, in a structured format that supports subsequent analysis.

We implement a feature in an experimental version of ImpRec that allows us to execute a sequence of experimental runs. Also, we implement an evaluator class that compares the ImpRec output to a 'gold standard' (see the 'static validation' in our parallel publication [23] for a detailed desctiption), and calculates established IR measures, e.g., precision, recall, and MAP at different cut-off levels. Finally, we print the results of each experimental run as a separate row in a file of Comma Separated Values (CSV). Below we show an excerpt of the resulting csv-file, generated from our screening experiment. The first four columns show the parameter values, and the final column is the response measured in Rc@20.

Listing 1 screening.csv generated from the ImpRec screening experiment.

```
alpha, penalty, start, level, resp
0.01, 0.01, 1, 1, 0.059375
0.01, 0.01, 1, 2, 0.078125
0.01, 0.01, 1, 4, 0.1125
0.01, 0.01, 1, 8, 0.115625
0.01, 0.01, 1, 16, 0.115625
...
(3,420 additional rows)
...
0.99, 8, 512, 4, 0.346875
0.99, 8, 512, 8, 0.315625
0.99, 8, 512, 16, 0.31875
0.99, 8, 512, 32, 0.321875
0.99, 8, 512, 64, 0.328125
```

5.2.3 C) Fit Low-order Polynomial Models

The final step in Phase 2 of TuneR involves analyzing the results from the screening experiment. A recurring observation in DoE is that only a few factors dominate the response, giving rise to well-known principles such as the '80-20 rule' and 'Occam's razor' [54, pp. 157]. In this step, the goal is to find the simplest polynomial model that can be used to explain the observed response.

If neither a first nor second-order polynomial model (i.e., linear and quadratic effects plus two-way interactions) fits the observations from the screening experiment, the response surface is complex. Modelling a complex response surfaces is beyond the scope of TuneR, as it requires advanced techniques such as neural networks [73, pp. 446], splines, or kriging [48]. If low-order polynomial models do not fit the response, TuneR instead relies on quasi-exhaustive space-filling designs (see Fig. 3). We discuss this further in Section 6, where we use exhaustive search to validate the result of the ImpRec tuning using TuneR.

When a low-order polynomial model has been fit, it might be possible to simplify it by removing parameters that do not influence the response much. The idea is that removal of irrelevant and noisy variables should improve the model. Note, however, that this process known as subset selection in linear regression, has been widely debated among statisticians, referred to as "fishing expeditions" and other derogatory terms (see for example discussions by Lukacs et al. [62] and Miller [70, pp. 8]). Still, when tuning an SE tool with a multitude of parameters, reducing the number of factors might be a necessary step for computational reasons. Moreover, working with a reduced set of parameters might reduce the risk of overfitting [3]. A standard approach is stepwise backward elimination [80, pp. 336], i.e., to iteratively remove parameters until all that remain have a significant effect on the response. While parameters with high p-values are candidates for removal [89, pp. 277], all such operations should be done with careful consideration. We recommend visualizing the data (cf. Fig. 5 and 6), and trying to understand why the screening experiment resulted in the response. Also, note that any parameter involved in interaction or quadratic effects must be kept.

To fit low-order polynomial models for ImpRec's response surface, we use the R package *rsm* [57], and the package *visreg* [25] to visualize the results. Assuming that screening.csv has been loaded to screening, Listing 2 and 3 fit a first-order and second-order polynomial model, respectively.

Listing 2 Fitting a first-order polynomial model with *rsm* [57]. The results are truncated.

```
> FO_model <- rsm(resp ~ FO(alpha, penalty, start, level), ...
    data=screening)
> summary(FO_model)

Call:
rsm(formula = resp ~ FO(alpha, penalty, start, level), data = ...
    screening)

              Estimate Std. Error t value Pr(>|t|)
(Intercept) 2.4976e-01 4.2850e-03 58.2855   <2e-16 ***
alpha       4.9432e-02 5.7393e-03  8.6129   <2e-16 ***
penalty     8.8721e-04 7.0248e-04  1.2630   0.2067
start       1.2453e-04 1.2052e-05 10.3327   <2e-16 ***
level       6.9603e-05 8.8805e-05  0.7838   0.4332
---
Signif. codes:  0    ***    0.001    **    0.01    *    0.05 ...
       .    0.1         1

Multiple R-squared:  0.05076,   Adjusted R-squared:  0.04965
F-statistic: 45.79 on 4 and 3425 DF,  p-value: < 2.2e-16

Analysis of Variance Table

Response: resp
                                  Df Sum Sq Mean Sq F value    Pr(>F)
FO(alpha, penalty, start, level)   4  2.234 0.55859  45.789 < 2.2e-16
Residuals                       3425 41.782 0.01220
Lack of fit                     3425 41.782 0.01220
Pure error                         0  0.000
```

The second order model fits the response better than the first order model; the lack of fit sum of squares is 29.1841 versus 41.782 (cf. Listing 3:62 and Listing 2:25). Moreover, Listing 3:44-47 show that the parameters $PENALTY$, $START$, and $LEVEL$ have a quadratic effect on the response. Also, interaction effects are significant, as shown by alpha:start, penalty:start, and start:level (cf. Listing 3:38-43). Fig. 5 visualizes[2] how the second order model fits the response, divided into the four parameters. As each data point represents an experimental run, we conclude that there is a large spread in the response. For most individual parameter values, there are experimental runs that yield an Rc@20 between approximately 0.1 and 0.4. Also, in line with Listing 3, we see that increasing $START$ appears to improve the response, but the second order model does not fit particularly well.

[2] R command: > *visreg*(*SO_model*) etc.

Listing 3 Fitting a second-order polynomial model with *rsm* [57]. The results are truncated.

```
> SO_model <- rsm(resp ~ SO(alpha, penalty, start, level), ...
    data=screening)
> summary(SO_model)
Call:
rsm(formula = resp ~ SO(alpha, penalty, start, level), data = ...
    screening)

                 Estimate  Std. Error  t value  Pr(>|t|)
(Intercept)    2.1502e-01  6.1700e-03  34.8493 < 2.2e-16 ***
alpha          2.6868e-02  1.8997e-02   1.4143 0.1573604
penalty        4.1253e-03  2.4574e-03   1.6787 0.0932935 .
start          1.2814e-03  4.1704e-05  30.7247 < 2.2e-16 ***
level          1.2045e-03  3.2053e-04   3.7579 0.0001742 ***
alpha:penalty -4.5460e-04  1.7894e-03  -0.2541 0.7994640
alpha:start    3.3458e-04  3.0698e-05  10.8993 < 2.2e-16 ***
alpha:level    5.5608e-05  2.2620e-04   0.2458 0.8058257
penalty:start  3.3783e-06  3.7573e-06   0.8991 0.3686588
penalty:level  6.7390e-05  2.7687e-05   2.4340 0.0149839 *
start:level   -4.9485e-06  4.7499e-07 -10.4182 < 2.2e-16 ***
alpha^2       -1.1659e-02  1.7181e-02  -0.6786 0.4974522
penalty^2     -5.8485e-04  2.7071e-04  -2.1604 0.0308128 *
start^2       -2.5851e-06  7.3816e-08 -35.0212 < 2.2e-16 ***
level^2       -1.2702e-05  4.4041e-06  -2.8840 0.0039508 **
---
Signif. codes:  0    ***    0.001    **    0.01    *    0.05 ...
       .    0.1         1

Multiple R-squared:  0.337, Adjusted R-squared:  0.3342
F-statistic:   124 on 14 and 3415 DF,  p-value: < 2.2e-16

Analysis of Variance Table

Response: resp
                                     Df  Sum Sq Mean Sq F value    ...
                                         Pr(>F)
FO(alpha, penalty, start, level)      4  2.2343 0.55859  65.363 < ...
    2.2e-16
TWI(alpha, penalty, start, level)     6  2.0014 0.33356  39.032 < ...
    2.2e-16
PQ(alpha, penalty, start, level)      4 10.5963 2.64907 309.983 < ...
    2.2e-16
Residuals                          3415 29.1841 0.00855
Lack of fit                        3415 29.1841 0.00855
Pure error                            0  0.0000
```

Listing 3 suggests that all four parameters are important when modelling the response surface. The statistical significance of the two parameters $START$ and $LEVEL$ is stronger than for α and $PENALTY$. However, $ALPHA$ is involved in a highly significant interaction effect (alpha:start in Listing 3:39). Also, the quadratic effect of $PENALTY$ on the response is significant (penalty^2 in Listing 3:45). Consequently, we do not simplify the second order model of the ImpRec response by reducing the number of parameters.

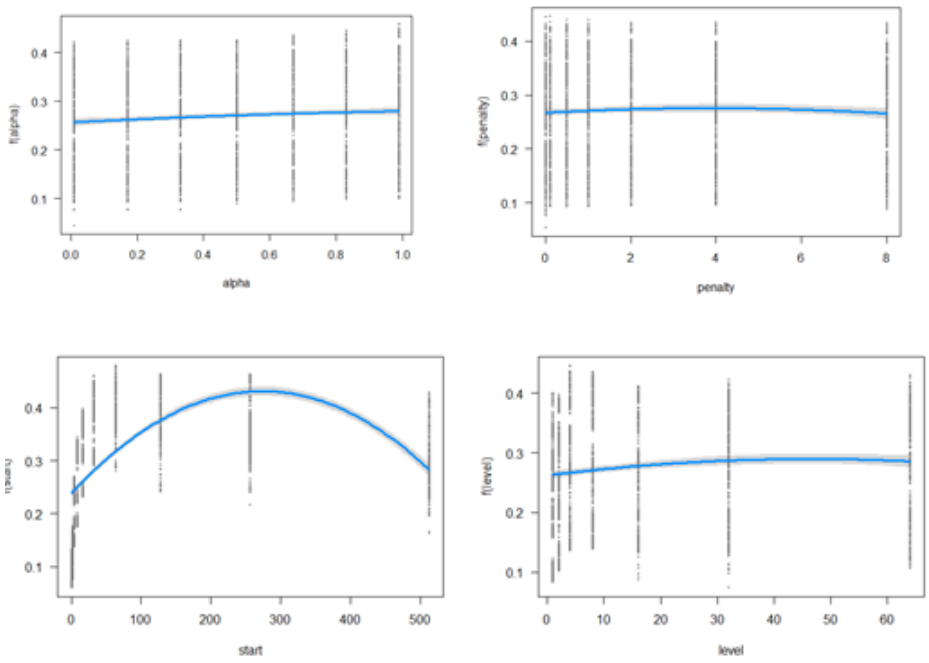

Fig. 5 Visualization of the second order model using *visreg* [25].

Fig. 6 displays boxplots of the response per parameter, generated with *ggplot2*[3] [99]. Based on the boxplots, we decide that a promising region for further tuning appears to involve high α values, $START$ between 32 and 128, and $LEVEL = 4$. The parameter value of $PENALTY$ however, does not matter much, as long as it is not too small, thus we consider values around 1 promising. An experimental run with the setting $\alpha = 0.9, PENALTY = 1, START = 64, LEVEL = 4$ gives a response of Rc@20=0.46875, compared to 0.41875 for the default setting. Thus, this 11.9% increase of the response confirms the choice of a promising region.

We summarize the results from screening the ImpRec parameters as follows:

- Centrality values of artifacts are more important than textual similarity when predicting impact ($ALPHA$ close to 1). Thus, previously impacted artifacts (i.e., artifacts with high centrality in the network) are likely to be impacted again.
- The low accuracy of the textual similarity is also reflected by the high parameter value of $START$; many starting points should be used as compensation.

[3] R commands for the $START$ parameter:
$> start\_box <- ggplot(screening, aes(factor(start), resp))$
$> start\_box + geom\_boxplot()$

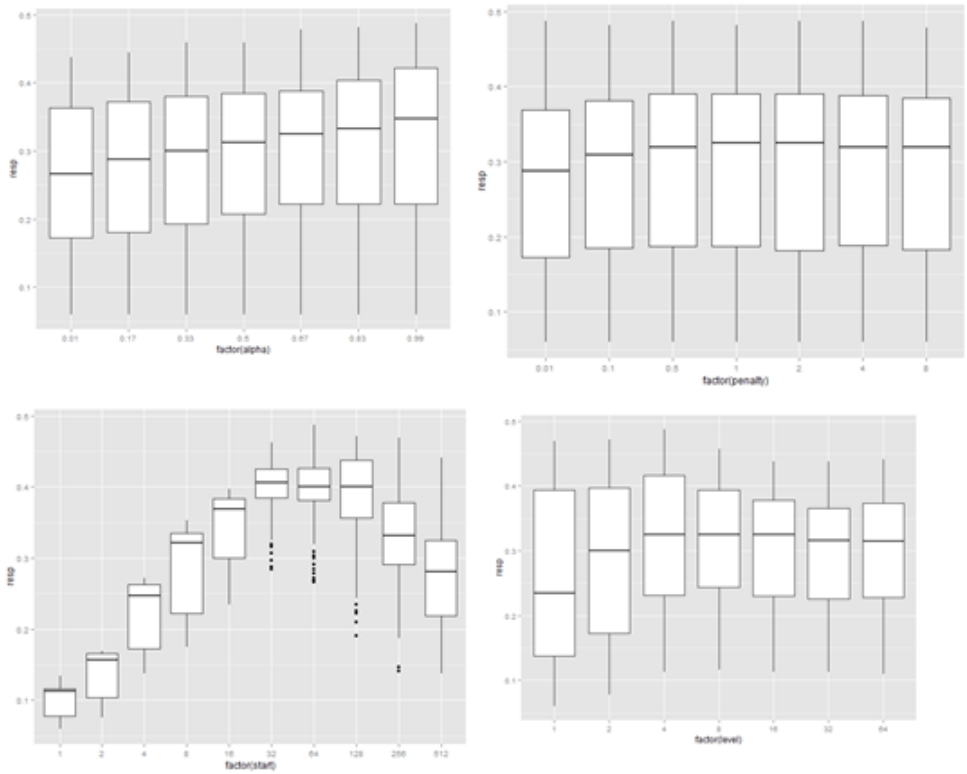

Fig. 6 Value of the response for different parameter settings. Note that the x-axis is only linear in the first plot (ALPHA).

- Regarding $LEVEL$ and $PENALTY$ we observe that following a handful of issue-issue links is beneficial, trying even broader searches however is not worthwhile.
- Also, severely penalizing distant artifacts does not benefit the approach, i.e., most related issues are meaningful to consider.
- A promising region, i.e., a suitable start setting for Phase 3, appears to be around $ALPHA = 0.9, PENALTY = 1, START = 64, LEVEL = 4$.

5.3 Phase 3: Apply Response Surface Methodology

The third phase in TuneR uses RSM to identify the optimal setting. The first part of RSM is an iterative process. We use a factorial design to fit a first-order model to the response surface, and then gradually modify the settings along the most promising direction, i.e., the *path of the steepest ascent*. Then, once further changes along that path do not yield improved responses, the intention is to pin-point the optimal setting in the vicinity. The pin-pointing is based on analyzing the stationary points of a second-order fit of that particular region

of the response surface, determined by applying an experiment using CCD (cf. Fig. 1). We describe Step 1 and 2 (i.e., the iterative part) together in the following subsection, and present Step 3 and 4 in the subsequent subsections.

When applying RSM, an important aspect is to use an appropriate *coding transformation*. The way the data are coded affects the results of the steepest ascent analysis. If all coded variables in the experiment vary over the same range, typically -1 and 1, each parameter gets an equal share in potentially determining the steepest ascent path [57].

5.3.1 Step 1 and 2: Factorial designs, First-order Models, and Iteration

Iteration of the first two steps is intended to quickly move toward the optimum. To find the direction, we design an experiment using 2^k factorial design and fit a first-order model of the response surface. The factorial design uses the outcome from Phase 2 as the center point, and for each parameter, we select a high value and a low value, referred to as the *factorial range* [32]. Selecting a feasible factorial range is one of the major challenges in RSM, another one is to select an appropriate *step size*.

Selecting a suitable factorial range for a computer experiment is a bit different than for a physical experiment. In traditional DoE, a too narrow range generates a factorial experiment dominated by noise. While noise is not a threat in experiments aimed at tuning SE tools, a too narrow range will instead not show any difference in the response at all. On the other hand, the range can also be too broad, as the response surface might then be generalized too much. Dunn reports that selecting extreme values is a common mistake in DoE, and suggests selecting 25% of the extreme range as a rule-of-thumb [32]. Since the number of tuning experiments typically is not limited in the same way as physical experiments, it is possible to gradually increase the factorial range until there is a difference in the response.

The factorial experiment yields the direction of the steepest ascent, but the next question is how much to adjust the setting in that direction, i.e., the step size. Again we want the difference to be large enough to cause a change in the response in a reasonable amount of experiments, but not so large that we move over an optimum. A good decision relies on the experimenter's understanding of the parameters involved in the SE tool. Otherwise, a rule-of-thumb is to choose a step size equal to the value of the largest coefficient describing the direction of the steepest ascent [8].

For tuning ImpRec, we decide to fit a first-order model in the region: $ALPHA = 0.9 \pm 0.05, PENALTY = 1 \pm 0.5, START = 64 \pm 4, LEVEL = 4 \pm 1$. Our experience from the screening experiments suggests that these levels should result in a measurable change in the response. Table 5.3.1 shows the 2^k factorial design we apply, and the results from the 16 experimental runs. We report the experimental runs in Yates' standard order according to the DoE convention, i.e., starting with low values, and then alternating the sign of the first variable the fastest, and the last variable the slowest [71, pp. 237]. Finally, we store the table, except the coded variables, in $rsm1\_factorial.csv$.

	Coded variables				Natural variables				
Exp. Run	x1	x2	x3	x4	ALPHA	PENALTY	START	LEVEL	Resp.
1	-1	-1	-1	-1	0.85	0.8	60	3	0.468750
2	1	-1	-1	-1	0.95	0.8	60	3	0.481250
3	-1	1	-1	-1	0.85	1.2	60	3	0.468750
4	1	1	-1	-1	0.95	1.2	60	3	0.478125
5	-1	-1	1	-1	0.85	0.8	68	3	0.478125
6	1	-1	1	-1	0.95	0.8	68	3	0.484375
7	-1	1	1	-1	0.85	1.2	68	3	0.475000
8	1	1	1	-1	0.95	1.2	68	3	0.484375
9	-1	-1	-1	1	0.85	0.8	60	5	0.471875
10	1	-1	-1	1	0.95	0.8	60	5	0.478125
11	-1	1	-1	1	0.85	1.2	60	5	0.471875
12	1	1	-1	1	0.95	1.2	60	5	0.478125
13	-1	-1	1	1	0.85	0.8	68	5	0.468750
14	1	-1	1	1	0.95	0.8	68	5	0.484375
15	-1	1	1	1	0.85	1.2	68	5	0.468750
16	1	1	1	1	0.95	1.2	68	5	0.481250

Table 3 First RSM iteration, 2^k factorial design for the four parameters ALPHA, PENALTY, START, and LEVEL.

Listing 4 shows the analysis of the results, conducted in coded variables. The standard coding transformation from a natural variable v_N to a coded variable v_C in DoE is [32, pp. 245]:

$$v_c = \frac{v_n - center_v}{\Delta_v/2} \tag{2}$$

where Δ_v is the factorial range of v_n, and $center_v$ its center point. For the four parameters of ImpRec, the coding is presented in Listing 4 on line 2.

Listing 4 reveals that $x1$ and $x3$ (i.e., $ALPHA$ and $START$ in coded values) affect the response the most. As visualizing the response surface in more than two variables is difficult, Fig. 7 shows the contour plot[4] wrt. $x1$ and $x3$, generated using *visreg* [25]. Our experiments show that higher response can be achieved if we increase $ALPHA$ and $START$, and decrease $PENALTY$ and $LEVEL$. We decide to use the step size provided by the direction of the steepest ascent in original units, as it already constitutes actionable changes to the parameters (cf. Listing 4:95). Table 5.3.1 shows the experimental results when gradually changing the ImpRec settings in the direction: $(+0.046, -0.0223, +1.338, -0.111)$. Note that $START$ and $LEVEL$ are integer parameters and thus rounded off accordingly (highlighted in italic font), and that $ALPHA$ has a maximum value of 1 (or 0.99 for practical reasons). We observe that the response continuously improves until step 10 (in bold font in Table 5.3.1). Two additional steps in the same direction confirm the decreased response.

[4] R command: $> visreg(rsm1\_model, "x1", "x3")$

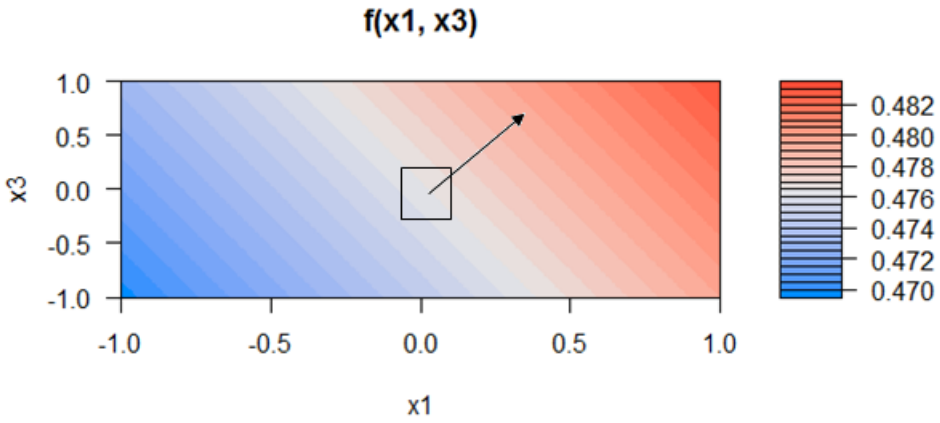

Fig. 7 Contour plot displaying the two most important parameters (ALPHA and START) in coded variables, generated using *visreg* [25]. We have added an arrow showing the direction of the steepest ascent.

Listing 4 Using *rsm* [57] to find the direction of the steepest ascent.

```
> rsm1 <- read.csv("rsm1_factorial.csv")
> rsm1_coded <- coded.data(rsm1, x1¬(alpha-0.9)/0.05, ...
    x2¬(penalty-1)/0.2, x3¬(start-64)/4, x4¬(level-4)/1)
> rsm1_model <- rsm(resp¬FO(x1, x2, x3, x4), data=rsm1_coded)
> summary(rsm1_model)

Call:
rsm(formula = resp ¬ FO(x1, x2, x3, x4), data = rsm1_coded)

               Estimate  Std. Error  t value  Pr(>|t|)
(Intercept)  0.47636719  0.00069429 686.1210 < 2.2e-16 ***
x1           0.00488281  0.00069429   7.0328 2.175e-05 ***
x2          -0.00058594  0.00069429  -0.8439   0.41668
x3           0.00175781  0.00069429   2.5318   0.02788 *
x4          -0.00058594  0.00069429  -0.8439   0.41668
---
Signif. codes:  0    ***    0.001    **    0.01    *    0.05 ...
       .    0.1         1

Multiple R-squared:  0.8389,    Adjusted R-squared:  0.7804
F-statistic: 14.32 on 4 and 11 DF,  p-value: 0.0002442

Analysis of Variance Table

Response: resp
                 Df     Sum Sq    Mean Sq F value    Pr(>F)
FO(x1, x2, x3, x4)  4 0.00044189 1.1047e-04  14.324 0.0002442
Residuals          11 0.00008484 7.7130e-06
Lack of fit        11 0.00008484 7.7130e-06
Pure error          0 0.00000000

Direction of steepest ascent (at radius 1):
        x1         x2         x3         x4
 0.9291177 -0.1114941  0.3344824 -0.1114941
```

Step	$ALPHA$	$PENALTY$	$START$	$LEVEL$	Resp.
0	0.9	1	64	4	0.46875
1	0.946456	0.977701	*65.33793*	*3.888506*	0.471875
2	0.992912	0.955402	*66.67586*	*3.777012*	0.4875
3	0.99	0.933104	*68.01379*	*3.665518*	0.4875
4	0.99	0.910805	*69.35172*	*3.554024*	0.4875
5	0.99	0.888506	*70.68965*	*3.442529*	0.490625
6	0.99	0.866207	*72.02758*	*3.331035*	0.49375
7	0.99	0.843908	*73.36551*	*3.219541*	0.5
8	0.99	0.821609	*74.70344*	*3.108047*	0.50625
9	0.99	0.799311	*76.04137*	*2.996553*	0.50625
10	**0.99**	**0.777012**	**77.37929**	**2.885059**	**0.49375**
11	0.99	0.754713	*78.71722*	*2.773565*	0.490625
12	0.99	0.732414	*80.05515*	*2.662071*	0.490625

Table 4 Iterative exploration along the direction of the steepest ascent. The tenth step, presented in bold font, results in a decreased response. Values in italic font are rounded off to the nearest integer value.

	Coded variables			Natural variables			
Exp. Run	$x1$	$x2$	$x3$	$PENALTY$	$START$	$LEVEL$	Resp.
1	-1	-1	-1	0.76	76	2	0.465625
2	1	-1	-1	0.84	76	2	0.5
3	-1	1	-1	0.76	80	2	0.4625
4	1	1	-1	0.84	80	2	0.490625
5	-1	-1	1	0.76	76	4	0.465625
6	1	-1	1	0.84	76	4	0.5
7	-1	1	1	0.76	80	4	0.4625
8	1	1	1	0.84	80	4	0.490625

Table 5 Second RSM iteration, 2^k factorial design for the three parameters $PENALTY$, $START$, and $LEVEL$.

```
96
97 Corresponding increment in original units:
98       alpha     penalty       start       level
99  0.04645588 -0.02229882  1.33792946 -0.11149412
```

The second iteration starts where the first ended, i.e., using the tenth step in Table 5.3.1 as its center point. The parameter $ALPHA$ is already at its maximum value, thus we focus on $PENALTY$, $START$, and $LEVEL$. We decide to use the following factorial ranges: $PENALTY = 0.80 \pm 0.04, START = 78 \pm 2$, and $LEVEL = 3 \pm 1$. Table 5.3.1 shows the corresponding 2^k factorial experiment. We store the table, except the coded variables, in $rsm2\_factorial.csv$. Listing 5 shows the analysis of the results, including the coding transformation of the parameters.

Listing 5 Second RSM iteration, using *rsm* [57] to find a new direction of the steepest ascent.

```
> rsm2 <- read.csv("rsm2_factorial.csv")
> rsm2_coded <- coded.data(rsm2, x2¬(penalty-0.80)/0.04, ...
    x3¬(start-78)/2, x4¬(level-3)/1)
> rsm2_model <- rsm(resp¬FO(x2, x3, x4), data=rsm2_coded)
> summary(rsm2_model)

Call:
rsm(formula = resp ¬ FO(x2, x3, x4), data = rsm2_coded)

               Estimate  Std. Error t value  Pr(>|t|)
(Intercept)  4.7969e-01  7.8125e-04     614 4.222e-11 ***
x2           6.7465e-18  7.8125e-04       0   1.00000
x3          -3.1250e-03  7.8125e-04      -4   0.01613 *
x4           1.5625e-02  7.8125e-04      20 3.688e-05 ***
---
Signif. codes:  0    ***    0.001    **    0.01    *    0.05 ...
       .    0.1         1

Multiple R-squared:  0.9905,    Adjusted R-squared:  0.9833
F-statistic: 138.7 on 3 and 4 DF,  p-value: 0.0001695

Analysis of Variance Table

Response: resp
               Df     Sum Sq    Mean Sq F value    Pr(>F)
FO(x2, x3, x4)  3 0.00203125 0.00067708  138.67 0.0001695
Residuals       4 0.00001953 0.00000488
Lack of fit     4 0.00001953 0.00000488
Pure error      0 0.00000000

Direction of steepest ascent (at radius 1):
           x2            x3            x4
 4.233906e-16 -1.961161e-01  9.805807e-01

Corresponding increment in original units:
   penalty      start      level
 0.0000000 -0.3922323  0.9805807
```

Listing 5 shows that the direction of the steepest ascent involves changing the value of $START$ and $LEVEL$, but not $PENALTY$. We also know that the setting $(0.99, 0.80, 76, 3)$ yields 0.50625 (cf., step 9 in Table 5.3.1). Table 5.3.1 shows the results from iteratively moving from this setting along the direction of the steepest ascent. As we do not observe any increases in the response when changing the two integer parameters $START$ and $LEVEL$, we conclude that this ImpRec setting is in the region of the maximum. In the next TuneR step, the goal is to pin-point the exact position of the stationary point.

5.3.2 Step 3: CCD and a second-order polynomial model

The final step in RSM is to fit a second-order polynomial model to the region close to the maximum, and to locate the stationary point. The most popular

Step	$ALPHA$	$PENALTY$	$START$	$LEVEL$	Resp.
0	0.99	0.80	76	3	0.50625
1	0.99	0.80	*75.60777*	*3.980581*	0.5
2	0.99	0.80	*75.21554*	*4.961161*	0.5
3	0.99	0.80	*74.8233*	*5.941742*	0.48125
4	0.99	0.80	*74.43107*	*6.922323*	0.45625

Table 6 Iterative exploration along the new direction of the steepest ascent. No changes result in an increased response, indicating that the current ImpRec setting is close to the optimum. Values in italic font are rounded off to the nearest integer value.

	Coded variables		Natural variables		
Exp. Run	$x3$	$x4$	$START$	$LEVEL$	Resp.
1	-1	-1	72	2	0.453125
2	1	-1	80	2	0.4625
3	-1	1	72	4	0.490625
4	1	1	80	4	0.490625
5	0	0	76	3	0.50625
6	-1.414	0	*70.344*	3	0.490625
7	+1.414	0	*81.656*	3	0.48125
8	0	-1.414	76	*4.414*	0.5
9	0	+1.414	76	*1.586*	0.465625

Table 7 Central composite design for the two parameters $START$ and $LEVEL$. Values in italic font are rounded off to the nearest integer value.

design for fitting a second-order model is CCD [71, pp. 501]. In traditional DoE, it is recommended to conduct three to five experimental runs at the center point. When tuning an SE tool, we do not need more than one, thus the only choice for the experimental design is the distance of the *axial runs*. As presented in Fig. 1, we recommend a rotatable design, i.e., that all settings in the tuning experiment should be at the same distance from the center point.

In the CCD experiment for tuning ImpRec, we focus on the two parameters $START$ and $LEVEL$. Listing 5:134 shows that these two parameters dwarf $PENALTY$ in this region. Furthermore, the parameter $ALPHA$ is already at its maximum value. Table 5.3.2 shows the CCD experiment and the corresponding responses. We store the table, except the coded variables, in *ccd.csv*. Listing 6 shows the analysis of the results, including the coding transformation of the parameters. In the final step in Phase 3 of TuneR, the outcome of the CCD experiment is analyzed.

Listing 6 Using *rsm* [57] to fit a second-order model of the response surface in the vicinity of the optimal response.

```
> ccd <- read.csv("ccd.csv")
> ccd_coded <- coded.data(ccd, x3¬(start-76)/2, x4¬(level-3)/1)
> ccd_model <- rsm(resp¬SO(x3, x4), data=ccd_coded)
> summary(ccd_model)

Call:
rsm(formula = resp ¬ SO(x3, x4), data = ccd_coded)

               Estimate  Std. Error  t value  Pr(>|t|)
(Intercept)  0.50614821  0.00268418 188.5672 4.745e-09 ***
x3          -0.00027574  0.00068784  -0.4009 0.7090065
x4           0.01666667  0.00163740  10.1788 0.0005247 ***
x3:x4       -0.00117188  0.00100270  -1.1687 0.3074152
x3^2        -0.00223432  0.00039648  -5.6354 0.0048794 **
x4^2        -0.02310668  0.00268905  -8.5929 0.0010078 **
---
Signif. codes:  0    ***    0.001    **    0.01    *    0.05 ...
       .    0.1         1

Analysis of Variance Table

Response: resp
            Df     Sum Sq    Mean Sq    F value    Pr(>F)
FO(x3, x4)   2 0.00166925 0.00083463 5.1884e+01  0.001378
TWI(x3, x4)  1 0.00002197 0.00002197 1.3659e+00  0.307415
PQ(x3, x4)   2 0.00137822 0.00068911 4.2838e+01  0.001990
Residuals    4 0.00006435 0.00001609
Lack of fit  3 0.00006435 0.00002145 4.4547e+29 1.101e-15
Pure error   1 0.00000000 0.00000000

Stationary point of response surface:
        x3         x4
-0.1573278  0.3646356

Stationary point in original units:
    start     level
75.685344  3.364636

Eigenanalysis:
$values
[1] -0.002217888 -0.023123113

$vectors
         [,1]      [,2]
x3 -0.9996068 0.0280393
x4  0.0280393 0.9996068
```

5.3.3 Step 4: Evaluate stationary point

The purpose of the previous TuneR step was to locate a stationary point in the vicinity of the optimal setting. The stationary point can either represent a maximum response, a minimum response, or a saddle point. The nature of

the stationary point is given by the signs of the *eigenvalues*: for a maximum response all are negative, and for a minimum response all are positive. Thus, if the eigenanalysis resulted in a maximum point, the tuning experiments have resulted in a pin-pointed optimal setting for the SE tool. If the eigenvalues have different signs on the other hand, then the CCD experiment located a saddle point. The experimenter then should then perform *ridge analysis* [75], i.e., conducting additional experimental runs following the ridge in both directions.

Regarding the stationary point identified for the tuning of ImpRec, it is located close to $START = 76, LEVEL = 3$ as shown in Listing 6:170. The eigenanalysis gives that it represents a maximum point (cf. Listing 6:174). Fig. 8 shows a visualization[5] using *visreg* [25] of the response surface in this region. visualizes the response surface in this region, confirming that a setting representing a maximum response has been identified. Thus, we conclude the parameter tuning of ImpRec as follows: $ALPHA = 0.99, PENALTY = 0.80, START = 76, LEVEL = 3$. Using the new parameter setting of ImpRec, we obtain $Rc@20 = 0.50625$ compared to $Rc@20 = 0.41875$ using the default settings of ImpRec ($ALPHA = 0.83, PENALTY = 0.2, START = 17, LEVEL = 7$). Applying TuneR has thus improved the Rc@20 for ImpRec by 20.9% on this particular dataset.

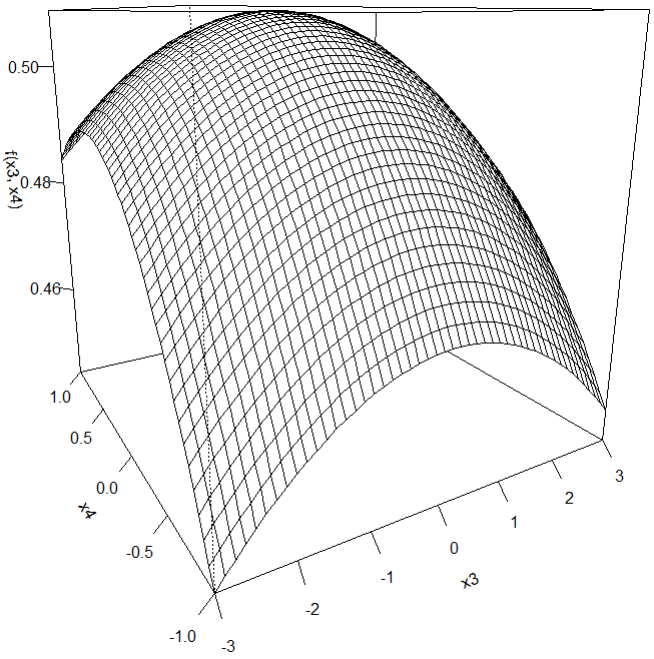

Fig. 8 Visualization of the response surface wrt. $x3$ and $x4$, i.e., $START$ and $LEVEL$ in coded variables. $ALPHA$ and $PENALTY$ are fixed to 0.99 and 0.80, respectively.

[5] R command: $> visreg2d(ccd_model, "x3", "x4", plot.type = "persp")$

5.4 Evaluate the setting

The final activity in TuneR is to perform an evaluation of the new parameter setting. Optimization based on a single response metric might result in a far too naive perspective, thus a more holistic analysis must be employed to determine the value of the new parameter setting. Numbers typically do not cover all aspects of a studied phenomenon [65], and there is a risk that the experimenter pushes the setting too much based on quantitative metrics, squeezing percentages without considering overall values of the process the SE tool is intended to support. The final activity of TuneR aims at taking a step back, and consider the bigger picture.

Fig. 9 shows a comparison of the ImpRec evaluation using the default setting (dashed line) and the tuned setting (solid line). The four subplots show the cut-off, N, of the ranked output list on the x-axis, and the following IR measures:

A: **Precision**, displaying the decrease that is characteristic to IR evaluations [27].
B: **Recall**, including the target metric for the tuning experiments: Rc@20.
C: **F_1-score**, the (balanced) harmonic mean of recall and precision.
D: **MAP**, an IR measure that combines recall with the performance of ranking.

The evaluation reveals that while the tuning has resulted in increases with regard to recall (cf. Fig. 9, subplot B), the improvements have been paid by other metrics. Indeed, TuneR has increased the target metric Rc@20 by 20.9%. Moreover, the response for higher N is even higher, reaching as high as 0.544 for Rc@43-50 (an increase by 27.0%). However, at low N the default setting actually reaches a higher recall, and first at Rc@11 the tuned setting becomes becomes better. To further add to the discussion, the three subplots A, C, and D all show that the default setting outperforms the tuned setting. For MAP@N, the difference between the default setting and the tuned setting actually increases for large N.

The evaluation of the tuned parameter setting for ImpRec, and the identified trade-off, shows the importance of the final step in TuneR. It is not at all clear from Fig. 9 whether the new parameter setting would benefit an engineer working with ImpRec. While we have carefully selected the response metrics, the trade-off appear to be bigger than expected. Not only is the trade-off between recall and precision evident, but also the trade-off within Rc@N; only after the cut-off N=11 the recall benefits from the new setting. Our work is an example of a purely quantitative *in silico* evaluation, conducted as computer experiments without considering the full operational context of ImpRec [20]. To fully understand how switching between the two settings affect the *utility* [6] of ImpRec, human engineers actually working with the tool must be studied. We report from such an *in situ* study in another paper [23], in which we deployed ImpRec in two development teams in industry, one with the default setting and one with the tuned setting.

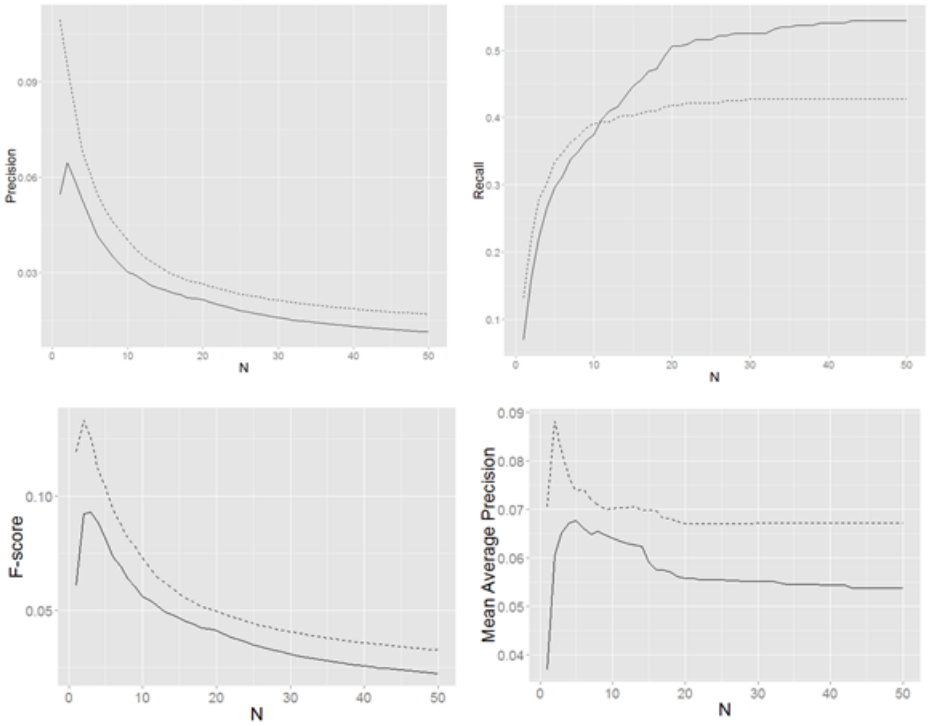

Fig. 9 Comparison of ImpRec output with default settings (dashed line) and tuned settings (solid line). Subplots clockwise from the first quadrant: Recall@N, MAP@N, F_1-score@N, and Precision@N,

6 Tuning ImpRec using Exhaustive Search

If the screening experiments of TuneR (Phase 2) fails to identify actionable regularities in the response surface, i.e., there is considerable lack of fit for both first and second-order models, the experimenter might decide to design an experiment of a more exhaustive nature. However, as an exhaustive amount of experimental runs is likely to be computationally expensive, a first try should be to investigate if a low-order polynomial model fit for the promising part of the response surface. If that is the case, Phase 3 could still be applicable in that region of the response surface. Otherwise, at least if the set of critical parameters has been reduced, a more exhaustive space-filling design (i.e., a brute force approach [76]) might be the remaining option to find a tuned setting. The purpose of this section is twofold. First, we present the design of a fine-granular space-filling design for tuning ImpRec. Second, the result of the exhaustive search acts as a proof-of-concept of TuneR, as we compare the results to the outcome from Phase 3.

For tuning of ImpRec, we design a uniform space-filling design. Table 8 shows the levels we explore in the experimental runs. The screening experiment

Parameter	#Levels	Values
$ALPHA$	21	0.01, 0.05, 0.10, 0.15, ... , 0.95, 0.99
$PENALTY$	11	0.01, 0.5, 1, 1.5, ... , 5
$START$	90	1, 2, 3, ... , 90
$LEVEL$	9	1, 2, ... , 9

Table 8 Uniform space-filling design for exhaustive approach to tuning of ImpRec. The design requires 187,110 experimental runs, compared to 3,430 in the screening experiment (cf. Table 5.1.3).

	Rc@20	#Settings
1	0.5375	12
2	0.534375	72
3	0.53125	60
4	0.528125	72
5	0.525	108
6	0.521875	238
7	0.51875	96
8	0.515625	120
9	0.5125	238
10	0.509375	83

Table 9 Top 10 results from the exhaustive experiment. The third column shows how many different settings that yield the response.

described in Section 5.2 shows that $ALPHA$ appears to be more important than $PENALTY$, thus we study it with a finer granularity. $START$ and $LEVEL$ are both positive integer parameters, and we choose to explore them starting from their lowest possible values. As the nature of the issue-issue links is unlikely to result in issue chains longer than five, setting the highest value to 9 is already a conservative choice. The potential of large $START$ on the other hand is less clear, but Fig. 6 suggests that values between 16 and 128 result in the best Rc@20. However, large $START$ require infeasible execution times, thus we restrict the parameter to 90 for practical reasons.

Table 6 shows the best results from running the exhaustive tuning experiments. In total, the experiments required 1,253 hours (about 52 days) to complete on a desktop computer[6], with an average of 24 s per experimental run. The best result we obtain in the exhaustive experiments is Rc@20=0.5375, a response we get from 12 different settings, a value that is 6.2% better than what we found using the three main phases of TuneR (Rc@20=0.50625). By looking at the 12 settings yielding Rc@20=0.5375, we note that $START = 51$ and $LEVEL = 3$ provide the best results. However, regarding the two remaining parameters, the pattern is less clear; $ALPHA$ varies from 0.6 to 0.99, and $PENALTY$ is either at low range (0.5 or 1.5) or at high range (4.5 or 5). Figure 10 summarizes the exhaustive experiment by presenting the distribution of responses per setting, as well as the execution times.

[6] Intel Core i5-2500K quad-core CPU 3.30 GHz with 8 GB RAM.

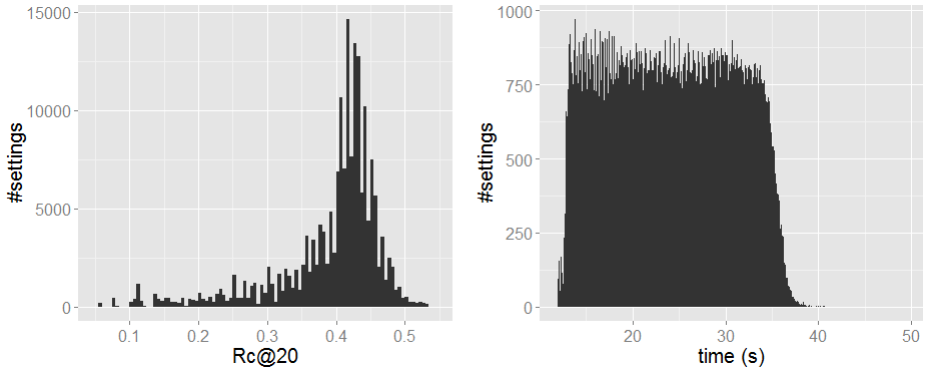

Fig. 10 Distribution of results from the exhaustive experiment. The y-axes show the number of settings that resulted in the output. The left figure displays Rc@20, and the right figure shows the execution time.

7 Discussion

Finding feasible parameter settings for SE tools is a challenging, but important, activity. SE tools are often highly configurable through parameters, but there is typically no silver bullet; there is not one default parameter setting that is optimal for all contexts. However, often advanced approaches are implemented in state-of-the-art SE tools. As a result of the tools' inherent complexity, academic researchers have published numerous papers on how to improve tool output by trying different settings and tuning internal algorithms. Consequently, SE tool developers cannot expect end users to understand all the intricate details of their implementations. Instead we argue that applied researchers need to provide guidelines to stimulate dissemination of SE tools in industry, i.e., to support transfer of research prototypes from academia to industry.

An established approach to improve processes is to use experiments. However, traditional DoE was developed for physical processes, much different from the application of SE tools. In this paper we introduced TuneR, a framework for tuning SE tools, using a combination of space-filling designs and RSM. Several researchers have presented advice and guidelines on how to tune various SE tools, but they typically address a specific family of tools, e.g., SBSE [5, 38], evolutionary software testing [29], LDA for feature location [14], and trace retrieval [61]. TuneR is instead a general framework, that can be applied for most types of SE tools.

As a proof-of-concept, and as a demonstration of TuneR's ease of use, we presented a detailed step-by-step tuning of the RSSE ImpRec. Using TuneR we obtain a considerable increase in the response variable of ImpRec, even though we considered the default setting already good enough for tool deployment in industry (see our industrial case study for further details [23]). We selected

the default setting[7] based on *ad hoc* tuning during development of ImpRec, but using TuneR resulted in a 20.9% higher response, i.e., an improvement from Rc@20=0.41875 to Rc@20=0.50625. Thus, in contrast to the Arcuri and Fraser's inconclusive results from tuning an SBSE tool [5], we demonstrate that RSM can be a component in successful tuning of SE tools.

Applying TuneR to tune an SE tool provides insights beyond what a feasible parameter setting. Thanks to the screening phase, TuneR identifies the most important parameters, both in terms of main effects and interaction effects. Especially interaction effects is missed when tuning tools using less structured experimental designs, e.g., COST analysis and *ad hoc* tuning. During tuning of ImpRec, we found that two interactions were significant: 1) positive interaction between $ALPHA$ and $START$, and 2) negative interaction between $START$ and $LEVEL$. Thus, if a high number of issue reports are used as starting points, then the ranking function should give more weight to the centrality measure than the textual similarity. Furthermore, if the number of starting points is high, then the number of links to follow in the knowledge base should be decreased.

Although resulting in a considerable improvement in the response, we found that the tuned setting[8] obtained from TuneR still was not optimal. Using exhaustive experiments, we identified settings that yield even higher responses, reaching as high as Rc@20=0.5375. However, running exhaustive experiments come at a high computational cost, and it is not certain that there is enough return on investment. In our example, we used more than 50 days of computation time for the exhaustive experiments, in total conducting 187,110 experimental runs, to find a 6.2% higher response (Rc@0=0.5375) compared to the TuneR setting. Moreover, we explored only four parameters in the exhaustive experiments. For other SE tools the number of parameters might be higher, and the combinatorial explosion quickly leads to infeasible exhaustive experimental designs. To mitigate this problem, the screening phase of TuneR could be used to identify the dominating parameters, in line with common practise in traditional DoE [71,32].

The exhaustive experiments revealed 12 different settings yielding the top response. A clear pattern in the 12 settings was found; to obtain the best results, $START$ and $LEVEL$ were set to 51 and 3, respectively. At the same time however, $ALPHA$ and $PENALTY$ could be set to several different combinations of values. Based on the screening phase of TuneR, we concluded that $ALPHA$ should be set to high values, as "centrality values are more important than textual similarity, i.e., previously impacted artifacts are likely to be impacted again" (see Section 5.2.3). In hindsight, with the knowledge obtained from the exhaustive experiment, it appears that early fixing $ALPHA$ to 0.99 was not necessarily the right decision, as high responses apparently can be obtained for a range of $ALPHA$ values. Experimentation is an iterative process, and the experimenter's knowledge gradually increases. Based on

[7] Default setting: $ALPHA = 0.83, START = 17, LEVEL = 7, PENALTY = 0.2$

[8] Tuned setting: $ALPHA = 0.99, PENALTY = 0.80, START = 76, LEVEL = 3$

the updated understanding of $ALPHA$, a next step could be to do another TuneR screening focusing on $0.6 \leq ALPHA \leq 0.99$.

We acknowledge two main threats to the validity of the tuned ImpRec setting we obtain through TuneR. First, there is always a threat that *focusing on a single response metric might be an oversimplification*, as discussed in Section 5.1.2. In Section 5.4, we show that while the tuned setting leads to an improved response in Rc@20, with regard to most other metrics we apply for the evaluation of the new setting, the output was better for the default setting. Whether Rc@20 is the best target metric is not certain, even though we posit that it reflects an important quality aspect of ImpRec, resulting in maximization of true impact among a manageable amount of recommendations. An alternative response metric could be MAP@20, also reported in the evaluation in Section 5.4, a metric that also considers the ranking of the true output among the top-20 recommendations. We stress that it is important to validate the response metric from the start, otherwise TuneR will move the setting in a direction that does not bring value.

Second, while we carefully selected four parameters for the tuning experiments, *there might be additional important parameters at play*. For example, the IR approach we apply in ImpRec could be adjusted in numerous ways, yet we consider the involved variation points as fixed. Apache Lucene, the integrated IR solution, is highly configurable, but as we have successfully used it for a similar task before (duplicate detection of issue reports [21]), we make the assumption that it performs well out-of-the-box. Other potentially useful approaches related to the IR, which we did not explore in this paper, is to perform pre-processing, e.g., stop word removal, stemming, and dimensionality reduction. However, as TuneR resulted in an increased response, also close to what the exhaustive experiment yielded, we argue that our selection of parameters was valid.

Furthermore, there are also some threats to the validity of the overall TuneR framework. While our goal when developing TuneR was to present a framework generally applicable to tuning of SE tools, the *external validity* [100] *of the approach is still uncertain*. We have only presented one single proof-of-concept, i.e., the tuning of the RSSE ImpRec, thus we need to conduct additional tuning experiments, with other SE tools, to verify the generalizability. We plan to continue evolving TuneR, and two involved activities we particularly want to focus on improving are: 1) guidelines regarding *parameter subset selection* when fitting low-order polynomial models during screening (Section 5.2.3), and 2) the *step size selection* in the RSM phase (Section 5.3.1). Finally, we argue that TuneR is easy to use, especially since we present hands-on examples in R, but the only way to validate the usability is by letting others try the framework.

8 Conclusion

In this paper we have presented TuneR, an experimental framework for tuning Software Engineering (SE) tools. TuneR build on methods from Design of Experiments (DoE) and Design of Computer Experiments (DoCE), two established fields with numerous successful applications in various engineering disciplines [45]. However, both DoE and DoCE have been developed to address experiments on phenomenon with a representation in the physical world, either directly (DoE) or indirectly through computer models (DoCE). We have discussed how tuning of SE tools is different from traditional experimentation, and how *TuneR combines space-filling designs and factorial designs to identify a feasible parameter setting*.

As a proof-of-concept, we applied TuneR to tune ImpRec, a recommendation system for change impact analysis, to a specific proprietary context. For all TuneR steps, we have provided detailed instructions on how to analyze the experimental output using various R packages. Using TuneR, we *increased the accuracy of the ImpRec recommendations by 20%* with regard to recall among the top-20 candidates. To validate the tuned setting, we also applied a more exhaustive space-filling design, trying in total 187,110 parameter settings. We found a parameter setting yielding a 6% higher response, but running the experiment required more than 50 days of computation time. Thus, we consider the proof-of-concept successful, as TuneR resulted in a similar response in a fraction of the time.

A major threat when tuning an SE tool is that the selected response metric, i.e., the target for optimization, does not fully capture the overall value of the tool. Optimizing a response might come at a price; increases in one metric might be paid by decreases in other metrics. The tuning of ImpRec is an example of this trade-off, and we show how precision, F_1-score, and mean average precision decrease with the new tuned setting. Even recall at lower cut-off points, i.e., when considering ten or fewer recommendations from ImpRec, yields decreased results with the tuned parameter setting. From this observation, we *stress the importance of carefully selecting the response metric*, and to *properly evaluate the consequences of the tuned parameter setting*, before deploying the tuned SE tool.

Acknowledgements This work was funded by the Industrial Excellence Center EASE - Embedded Applications Software Engineering[9].

References

1. W. AbdelMoez, M. Kholief, and F. Elsalmy. Improving bug fix-time prediction model by filtering out outliers. In *Proc. of the International Conference on Technological Advances in Electrical, Electronics and Computer Engineering*, pages 359–364, 2013.

[9] http://ease.cs.lth.se

2. R. Abreu, P. Zoeteweij, and A. van Gemund. An evaluation of similarity coefficients for software fault localization. In *Proc. of the 12th Pacific Rim International Symposium on Dependable Computing*, pages 39–46, 2006.
3. C. Andersen and R. Bro. Variable selection in regression - a tutorial. *Journal of Chemometrics*, 24(11-12):728–737, 2010.
4. N. Ansari and E. Hou. *Computational Intelligence for Optimization*. Springer, 2012.
5. A. Arcuri and G. Fraser. Parameter tuning or default values? an empirical investigation in search-based software engineering. *Empirical Software Engineering*, 18(3):594–623, 2013.
6. I. Avazpour, T. Pitakrat, L. Grunske, and J. Grundy. Dimensions and metrics for evaluating recommendation systems. In M. Robillard, W. Maalej, R. Walker, and T. Zimmermann, editors, *Recommendation Systems in Software Engineering*, pages 245–273. Springer, 2014.
7. N. Ayewah, D. Hovemeyer, J. Morgenthaler, J. Penix, and W. Pugh. Using static analysis to find bugs. *IEEE Software*, 25(5):22–29, 2008.
8. T. Bartz-Beielstein, C. Lasarczyk, and M. Preuss. The sequential parameter optimization toolbox. In T. Bartz-Beielstein, M. Chiarandini, L. Paquete, and M. Preuss, editors, *Experimental Methods for the Analysis of Optimization Algorithms*, pages 337–362. Springer, 2010.
9. V. Basili. Software modeling and measurement: the goal/question/metric paradigm. Technical Report CS-TR-2956, University of Maryland, 1992.
10. V. Basili, R. Selby, and D. Hutchens. Experimentation in software engineering. *Transactions on Software Engineering*, 12(7):733–743, 1986.
11. M. Baz, B. Hunsaker, P. Brooks, and A. Gosavi. Automated tuning of optimization software parameters. Technical report, Dept. of Industrial Engineering, University of Pittsburgh, 2007.
12. K. Beck. *Test-driven Development: By Example*. Addison-Wesley Professional, 2003.
13. D. Berry. *The philosophy of software: code and mediation in the digital age*. Palgrave Macmillan, Basingstoke, 2011.
14. L. Biggers, C. Bocovich, R. Capshaw, B. Eddy, L. Etzkorn, and N. Kraft. Configuring latent dirichlet allocation based feature location. *Empirical Software Engineering*, 19(3):465–500, 2014.
15. D. Binkley and D. Lawrie. Learning to rank improves IR in SE. In *Proc. of the International Conference on Software Maintenance and Evolution*, pages 441–445, 2014.
16. M. Birattari. *Tuning Metaheuristics - A Machine Learning Perspective*. Springer, 2009.
17. M. Borg, O. Gotel, and K. Wnuk. Enabling traceability reuse for impact analyses: A feasibility study in a safety context. In *Proceedings of the 7th International Workshop on Traceability in Emerging Forms of Software Engineering*, 2013.
18. M. Borg and P. Runeson. IR in software traceability: From a bird's eye view. In *Proc of the 7th International Symposium on Empirical Software Engineering and Measurement*, pages 243–246, 2013.
19. M. Borg and P. Runeson. Changes, evolution and bugs - recommendation systems for issue management. In M. Robillard, W. Maalej, R. Walker, and T. Zimmermann, editors, *Recommendation Systems in Software Engineering*, pages 477–509. Springer, 2014.
20. M. Borg, P. Runeson, and L. Broden. Evaluation of traceability recovery in context: A taxonomy for information retrieval tools. In *Proc. of the 16th International Conference on Evaluation & Assessment in Software Engineering*, pages 111–120, 2012.
21. M. Borg, P. Runeson, J. Johansson, and M. Mantyla. A replicated study on duplicate detection: Using {Apache Lucene} to search among {Android} defects. In *Proc. of the 8th International Symposium on Empirical Software Engineering and Measurement*, pages 8:1–8:4, 2014.
22. M. Borg, K. Wnuk, and D. Pfahl. Industrial comparability of student artifacts in traceability recovery research - an exploratory survey. In *Proc. of the 16th European Conference on Software Maintenance and Reengineering*, pages 181–190, 2012.

23. M. Borg, K. Wnuk, B. Regnell, and P. Runeson. Supporting change impact analysis using a recommendation system - an industrial case study in a safety-critical context. *Submitted to a journal*, 2015.
24. G. Box, S. Hunter, and W. Hunter. *Statistics for Experimenters: Design, Innovation, and Discovery, 2nd Edition*. Wiley, Hoboken, N.J, 2 edition, 2005.
25. P. Breheny and W. Burchett. Visualization of regression models using visreg. Technical report, University of Kentucky, 2013.
26. M. Brown and D. Goldenson. Measurement and analysis: what can and does go wrong? In *Proc. of the 10th International Symposium on Software Metrics*, pages 131–138, 2004.
27. M. Buckland and F. Gey. The relationship between recall and precision. *Journal of the American Society for Information Science*, 45(1):12–19, 1994.
28. A. Casamayor, D. Godoy, and M. Campo. Identification of non-functional requirements in textual specifications: A semi-supervised learning approach. *Information and Software Technology*, 52(4):436–445, 2010.
29. L. Da Costa and M. Schoenauer. Bringing evolutionary computation to industrial applications with GUIDE. In *Proc. of the 11th Annual Conference on Genetic and Evolutionary Computation*, pages 1467–1474, 2009.
30. J. de la Vara, M. Borg, K. Wnuk, and L. Moonen. Survey on safety evidence change impact analysis in practice: Detailed description and analysis. Technical Report 18, Simula Research Laboratory, 2014.
31. C. Dekkers and P. McQuaid. The dangers of using software metrics to (mis) manage. *IT Professional*, 4(2):24–30, 2002.
32. K. Dunn. Design and analysis of experiments. In *Process Improvement Using Data*, pages 207–288. 294-34b8 edition, 2014.
33. K.-T. Fang, R. Li, and A. Sudjianto. *Design and Modeling for Computer Experiments*. CRC Press, Boca Raton, FL, 2006.
34. R. Feldt and P. Nordin. Using factorial experiments to evaluate the effect of genetic programming parameters. In *Proc. of the European Conference EuroGP 2000*, pages 271–282, 2000.
35. A. Fisher. *CASE: Using Software Development Tools*. Wiley, New York, 2 edition, 1991.
36. E. Frank, M. Hall, G. Holmes, R. Kirkby, B. Pfahringer, I. Witten, and L. Trigg. Weka - a machine learning workbench for data mining. In O. Maimon and L. Rokach, editors, *Data Mining and Knowledge Discovery Handbook*, pages 1305–1314. Springer, 2005.
37. G. Fraser and A. Arcuri. EvoSuite: Automatic test suite generation for object-oriented software. In *Proc. of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, pages 416–419, 2011.
38. G. Fraser and A. Arcuri. The seed is strong: Seeding strategies in search-based software testing. In *Proc. of the 5th International Conference on Software Testing, Verification and Validation*, pages 121–130, 2012.
39. O. Gotel, J. Cleland-Huang, J. Huffman Hayes, A. Zisman, A. Egyed, P. Grunbacher, A. Dekhtyar, G. Antoniol, J. Maletic, and P. Mader. Traceability fundamentals. In J. Cleland-Huang, O. Gotel, and A. Zisman, editors, *Software and Systems Traceability*, pages 3–22. Springer, 2012.
40. E. Gummesson. *Qualitative Methods in Management Research*. SAGE Publications, 1999.
41. T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell. A systematic literature review on fault prediction performance in software engineering. *Transactions on Software Engineering*, 38(6):1276–1304, 2012.
42. M. Harman. The current state and future of search based software engineering. In *Proc. of FOSE '07 Future of Software Engineering*, pages 342–357, 2007.
43. M. Hofmann and R. Klinkenberg, editors. *RapidMiner: Data Mining Use Cases and Business Analytics Applications*. CRC Press, 2013.
44. M. Host, B. Regnell, and C. Wohlin. Using students as subjects - a comparative study of students and professionals in lead-time impact assessment. *Empirical Software Engineering*, 5(3):201–214, 2000.

45. L. Ilzarbe, M. Alvarez, E. Viles, and M. Tanco. Practical applications of design of experiments in the field of engineering: A bibliographical review. *Quality and Reliability Engineering International*, 24(4):417–428, 2008.
46. {International Electrotechnical Commission}. *IEC 61511-1 ed 1.0, Safety instrumented systems for the process industry sector*. 2003.
47. {International Electrotechnical Commission}. *IEC 61508 ed 1.0, Electrical/electronic/programmable electronic safety-related systems*. 2010.
48. D. Jones. A taxonomy of global optimization methods based on response surfaces. *Journal of Global Optimization*, 21(4):345–383, 2001.
49. L. Jonsson, M. Borg, D. Broman, K. Sandahl, S. Eldh, and P. Runeson. Automated bug assignment: Ensemble-based machine learning in large scale industrial contexts. *Under revision in Empirical Software Engineering*, 2015.
50. C. Kaner and W. Bond. Software engineering metrics: What do they measure and how do we know? In *In Proc. of 10th International Symposium on Software Metrics*, 2004.
51. E. Keenan, A. Czauderna, G. Leach, J. Cleland-Huang, Y. Shin, E. Moritz, M. Gethers, D. Poshyvanyk, J. Maletic, J. Huffman Hayes, A. Dekhtyar, D. Manukian, S. Hossein, and D. Hearn. TraceLab: An experimental workbench for equipping researchers to innovate, synthesize, and comparatively evaluate traceability solutions. In *Proc. of the 34th International Conference on Software Engineering*, pages 1375–1378, 2012.
52. M. Kersten and G. Murphy. Using task context to improve programmer productivity. In *Proc. of the 14th International Symposium on Foundations of Software Engineering*, pages 1–11, 2006.
53. J. Kleijnen. Low-order polynomial regression metamodels and their designs: basics. In *Design and Analysis of Simulation Experiments*, pages 15–71. Springer, 2008.
54. J. Kleijnen. Screening designs. In *Design and Analysis of Simulation Experiments*. Springer, 2008.
55. A. Lamkanfi and S. Demeyer. Filtering bug reports for fix-time analysis. In *Proc. of the 16th European Conference on Software Maintenance and Reengineering*, pages 379–384, 2012.
56. N. Lavesson and P. Davidsson. Quantifying the impact of learning algorithm parameter tuning. In *Proc. of the 21st National Conference on Artificial Intelligence - Volume 1*, pages 395–400, 2006.
57. R. Lenth. Response-surface methods in {R}, using rsm. *Journal of Statistical Software*, 32(7):1–17, 2009.
58. S. Levy and D. Steinberg. Computer experiments: A review. *AStA Advances in Statistical Analysis*, 94(4):311–324, 2010.
59. Z. Li, M. Harman, and R. Hierons. Search algorithms for regression test case prioritization. *IEEE Transactions on Software Engineering*, 33(4):225–237, 2007.
60. T.-Y. Liu. *Learning to Rank for Information Retrieval*. Springer, 2011.
61. S. Lohar, S. Amornborvornwong, A. Zisman, and J. Cleland-Huang. Improving trace accuracy through data-driven configuration and composition of tracing features. In *Proc. of the 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 378–388, 2013.
62. P. Lukacs, K. Burnham, and D. Anderson. Model selection bias and {F}reedmans paradox. *Annals of the Institute of Statistical Mathematics*, 62(1):117–125, 2010.
63. C. Macdonald, R. Santos, and I. Ounis. The whens and hows of learning to rank for web search. *Information Retrieval*, 16(5):584–628, 2013.
64. R. Madachy. *Software Process Dynamics*. Wiley, 2007.
65. K. Malterud. The art and science of clinical knowledge: evidence beyond measures and numbers. *The Lancet*, 358(9279):397–400, 2001.
66. C. Manning, P. Raghavan, and H. Schutze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.
67. M. McCandless, E. Hatcher, and O. Gospodnetic. *Lucene in Action*. Manning Publications, Greenwich, 2 edition, 2010.
68. T. Mens. Introduction and roadmap: History and challenges of software evolution. In T. Mens and S. Demeyer, editors, *Software Evolution*, pages 1–11. Springer, 2008.
69. T. Menzies and M. Shepperd. Special issue on repeatable results in software engineering prediction. *Empirical Software Engineering*, 17(1-2):1–17, 2012.

70. A. Miller. *Subset Selection in Regression*. CRC Press, 2002.
71. D. Montgomery. *Design and Analysis of Experiments*. Wiley, 8 edition, 2013.
72. K. Moon. The nature of computer programs: Tangible? goods? personal property? intellectual property? *European Intellectual Property Review*, 31(8):396–407, 2009.
73. R. Myers, D. Montgomery, and C. Anderson-Cook. *Response Surface Methodology: Process and Product Optimization Using Designed Experiments*. Wiley, 2009.
74. M. Nandagopal, K. Gala, and V. Premnath. Improving technology commercialization at research institutes: Practical insights from NCL innovations. In *Proc. of the Innovation Educators' Conference*, Hyderabad, India, 2011.
75. G. Neddermeijer, G. van Oortmarssen, N. Piersma, and R. Dekker. A framework for response surface methodology for simulation optimization. In *Proc. of the 32nd Conference on Winter Simulation*, WSC '00, pages 129–136, 2000.
76. J. Nievergelt. Exhaustive search, combinatorial optimization and enumeration: Exploring the potential of raw computing power. In *Proc. of the 27th Conference on Current Trends in Theory and Practice of Informatics*, pages 18–35, 2000.
77. R. Oliveto, M. Gethers, D. Poshyvanyk, and A. De Lucia. On the equivalence of information retrieval methods for automated traceability link recovery. In *International Conference on Program Comprehension*, pages 68–71, 2010.
78. S. Pfleeger. Experimental design and analysis in software engineering. *Annals of Software Engineering*, 1(1):219–253, 1995.
79. R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2008. ISBN 3-900051-07-0.
80. A. Rencher. *Methods of Multivariate Analysis*. Wiley, 2002.
81. M. Robillard and R. Walker. An introduction to recommendation systems in software engineering. In M. Robillard, W. Maalej, R. Walker, and T. Zimmermann, editors, *Recommendation Systems in Software Engineering*, pages 1–11. Springer, 2014.
82. B. Robinson and P. Francis. Improving industrial adoption of software engineering research: a comparison of open and closed source software. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement*, pages 21:1–21:10, 2010.
83. T. Santner, B. Williams, and W. Notz. *The Design and Analysis of Computer Experiments*. Springer, 2003.
84. M. Sarcar, K. Mallikarjuna Rao, and K. Lalit Narayan. *Computer Aided Design and Manufacturing*. PHI Learning Pvt. Ltd., 2008.
85. D. Sculley. Large scale learning to rank. In *In Proc. of the NIPS 2009 Workshop on Advances in Ranking*, 2009.
86. C. Seiffert, T. Khoshgoftaar, J. Van Hulse, and A. Folleco. An empirical study of the classification performance of learners on imbalanced and noisy software quality data. *Information Sciences*, 259:571–595, 2014.
87. M. Shepperd, C. Schofield, and B. Kitchenham. Effort estimation using analogy. In *Proc. of the 18th International Conference on Software Engineering*, pages 170–178, 1996.
88. M. Shepperd, Q. Song, Z. Sun, and C. Mair. Data quality: Some comments on the NASA software defect datasets. *Transactions on Software Engineering*, 39(9):1208–1215, 2013.
89. P. Teetor. *R Cookbook*. O'Reilly Media, 2011.
90. S. Thomas, M. Nagappan, D. Blostein, and A. Hassan. The impact of classifier configuration and classifier combination on bug localization. *IEEE Transactions on Software Engineering*, 39(10):1427–1443, 2013.
91. G. Travassos, P. dos Santos, P. Neto, and J. Biolchini. An environment to support large scale experimentation in software engineering. In *Proc. of the 13th International Conference on Engineering of Complex Computer Systems*, pages 193–202, 2008.
92. M. Tsunoda and K. Ono. Pitfalls of analyzing a cross-company dataset of software maintenance and support. In *Proc. of the 15th International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*, pages 1–6, 2014.
93. B. Turhan. On the dataset shift problem in software engineering prediction models. *Empirical Software Engineering*, 17(1-2):62–74, 2012.

94. J. Urbano. Information retrieval meta-evaluation: Challenges and opportunities in the music domain. In *Proc. of the 12th International Society for Music Information Retrieval Conference*, pages 597–602, 2011.
95. S. Vaidhyanathan. *The Googlization of Everything: (And Why We Should Worry)*. University of California Press, 2012.
96. G. Vining. Adapting response surface methodology for computer and simulation experiments. In H. Tsubaki, S. Yamada, and K. Nishina, editors, *The Grammar of Technology Development*, pages 127–134. Springer, 2008.
97. E. Voorhees. *TREC: Experiment and Evaluation in Information Retrieval*. MIT Press, 2005.
98. R. Walker and R. Holmes. Simulation - a methodology to evaluate recommendation systems in software engineering. In M. Robillard, W. Maalej, R. Walker, and T. Zimmermann, editors, *Recommendation Systems in Software Engineering*, pages 301–327. Springer, 2014.
99. H. Wickham. *ggplot2: Elegant Graphics for Data Analysis*. Springer, 2009.
100. C. Wohlin, P. Runeson, M. Host, M. Ohlsson, B. Regnell, and A. Wesslen. *Experimentation in Software Engineering: A Practical Guide*. Springer, 2012.
101. D. Wolpert and W. Macready. No free lunch theorems for optimization. *Transactions on Evolutionary Computation*, 1(1):67–82, 1997.
102. H. Zaragoza and M. Najork. Web search relevance ranking. In L. Liu and T. Oszu, editors, *Encyclopedia of Database Systems*, pages 3497–3501. Springer, 2009.
103. X. Zou, R. Settimi, and J. Cleland-Huang. Improving automated requirements trace retrieval: A study of term-based enhancement methods. *Empirical Software Engineering*, 15(2):119–146, 2010.