Predicting Relevance of Change Recommendations

Thomas Rolfsnes
Simula Research Laboratory
Oslo, Norway
thomgrol@simula.no

Leon Moonen
Simula Research Laboratory
Oslo, Norway
leon.moonen@computer.org

David Binkley
Loyola University Maryland
Baltimore, Maryland, USA
binkley@cs.loyola.edu

***Abstract*—Software change recommendation seeks to suggest artifacts (e.g., files or methods) that are related to changes made by a developer, and thus identifies possible omissions or next steps. While one obvious challenge for recommender systems is to produce *accurate* recommendations, a complimentary challenge is to *rank* recommendations based on their *relevance*. In this paper, we address this challenge for recommendation systems that are based on *evolutionary coupling*. Such systems use *targeted association-rule mining* to identify *relevant patterns* in a software system's change history. Traditionally, this process involves ranking artifacts using *interestingness measures* such as *confidence* and *support*. However, these measures often fall short when used to assess recommendation relevance.**

We propose the use of random forest classification models to assess recommendation relevance. This approach improves on past use of various interestingness measures by learning from previous change recommendations. We empirically evaluate our approach on fourteen open source systems and two systems from our industry partners. Furthermore, we consider complimenting two mining algorithms: CO-CHANGE and TARMAQ. The results find that random forest classification significantly outperforms previous approaches, receives lower Brier scores, and has superior trade-off between precision and recall. The results are consistent across software system and mining algorithm.

***Index Terms*—recommendation confidence, evolutionary coupling, targeted association rule mining, random forests.**

I. INTRODUCTION

When software systems evolve, the amount and complexity of interactions in the code grows. As a result, it becomes increasingly challenging for developers to foresee and reason about the effects of a change to the system. One proposed solution, change impact analysis [6], aims to identify software artifacts (e.g., files, methods, classes) affected by a given set of changes. The impacted artifacts form the basis for *change recommendations*, which suggests to a developer a list of artifacts that are related to their (proposed) changes to the code, supporting in the process of addressing change propagation and ripple-effects [17, 12, 40, 44].

One promising approach to change recommendation aims to identify potentially relevant items based on *evolutionary* (or *logical*) *coupling*. This approach can be based on a range of granularities of co-change information [14, 5, 33] as well as code-churn [15] or even interactions with an IDE [42]. Change recommendation based on evolutionary coupling has the intriguing property that it effectively taps into the inherent knowledge that software developers have regarding dependencies between the artifacts of a system. Our present work considers co-change information extracted from a version control system such as Git, SVN, or Mercurial.

One challenge faced by all recommender systems are false positives. This challenge becomes acute if developers come to believe that automated tools are "mostly wrong" [31]. Clearly, algorithms with higher accuracy will help address this challenge. However, we can also address this challenge with algorithms that assess the *relevance* of a proposed recommendation. In fact, the two approaches are complimentary. In this paper we hypothesize that *history aware* relevance prediction, which exploits earlier change recommendations to assess the relevance of a current recommendation, and ranks recommendations according to decreasing relevance, can help mitigate the challenge of false positives.

Our approach consists of training a *random forest classification model* [7] on previous change recommendations with *known relevance*. The model is used to give a *single likelihood estimate* of the relevance of future change recommendations. Automatic assessment of recommendation relevance frees developers from having to perform this time-consuming task. Our work facilitates further automation of the change recommendation process, only notifying the developer when relevant recommendations are available. Furthermore, our approach compliments existing research work on improving mining algorithm accuracy, as its application is independent of the mining algorithm used to generate recommendations.

Contributions: **(a)** We present twelve features describing aspects of *change sets*, *change histories*, and generated *change recommendations*. These features are used to build a random forest classification model for *recommendation relevance*. **(b)** We assess our model in a large empirical study encompassing sixteen software systems, including two from our industry partners *Cisco* and *KM*. Furthermore, change recommendations are generated using both CO-CHANGE and TARMAQ to assess the external validity of our approach. **(c)** We evaluate the importance of each of the twelve features used in our relevance classification model.

II. OVERALL APPROACH

The overarching goal of this paper is to attach appropriate *relevance scores* to change recommendations based on association rule mining. We consider this question of relevance from a developer viewpoint, where "relevant" means "useful for the developer". We therefore consider a change recommendation relevant if it contains a correct artifact as one of its top ten highest ranked artifacts.

We propose an approach that uses *classification based on random forests* [7] to learn from previous change recommendations in order to assess the current one. Thus, relevance of

a new recommendation is assessed based on the *known relevance* of historically similar and dissimilar recommendations.

Traditionally, the same interestingness measure used to rank the artifacts of a recommendation are also used to assess its relevance [37, 28, 16]. For example, an interestingness measure might weight artifacts on a range from 0 to 1, enabling an internal ranking. Given the ranking it is then up to the user to assess whether the recommendation is relevant. Naturally, if the top ranked artifacts have received weights close to the maximum (1 in this example), the recommendation is assumed relevant and will likely be acted upon. Recent work found that, in the context of software change recommendation, Agrawal's *confidence* interestingness measure [1] performs among the top-in-class when compared to more complex measures [30]. Considering this result, we use confidence as a baseline and set out to compare the relevance predictions given by our proposed approach against those based on confidence:

RQ 1 *Does classification based on random forests improve upon the use of confidence as a relevance classifier?*

To train our classification model, a range of features must be introduced that describe attributes related to a change recommendation. The better these features capture *key attributes* the better we can learn from previous recommendations and consequently the better we can assess the relevance of a current recommendation. Thus our second research question is

RQ 2 *What are the most important features for classifying change recommendation relevance?*

In our study we use random forests for their proven performance [9] and intrinsic ability to assess variable (feature) importance [7]. By answering our two research questions we seek to uncover whether change recommendation relevance is a viable venue for further research. If so, our approach may prove to be an important compliment to existing algorithms for change recommendation, helping to gain both better recommendations, and higher confidence in those recommendations.

III. Related Work

Recommendation Relevance: A shared goal in recommendation systems is uncovering *interesting* findings in a data-set, which means that an important research question concerns what actually characterizes the *interestingness* of a finding. Over the years, numerous measures for interestingness have been proposed [37, 28, 16, 24, 21]. A recent evaluation of 39 of these measures in the context of software change recommendation found that the traditional measures of *confidence* and *support* perform just as well as more recent and often more complex measures [30].

Cheetham and Price evaluated *indicators* (features) that can be used to provide confidence scores for *case based reasoning systems* [10, 11]. To provide a recommendation for a new case, the *k-nearest neighbor* algorithm was used, thus the evaluated features were tightly woven with the kind of output that the k-nearest neighbor algorithm produces. Example features include the "Number of cases retrieved with best solution" and "Similarity of most similar case." The features were tested for importance using leave-one-out testing in combination with the decision tree algorithm C4.5. In comparison, our use of random decision trees avoids the need for leave out testing of features as feature importance is internally accounted for.

Le et al. propose an approach for predicting whether the output of a bug localization tool is relevant [23, 22]. As for this paper, an output is considered relevant if a true positive is part of the top 10 recommended artifacts. While *change recommendation* can be seen as stopping faults before they happen, *bug localization* is a complementary approach for already existing faults. State of the art bug localization is based on comparing normal and faulty execution traces (spectrum based). In order to predict relevance, Le et al. identify 50 features related primarily to traces, but also considered the weights of recommended (suspicious) artifacts. These features were then used to train a classification model for relevance based on support vector machines.

Rule Aggregation, Clustering, and Filtering: Rolfsnes et al. propose aggregation of association rules to combine their evidence (or interestingness) [35]. Aggregation likely increases recommendation relevance: for example, consider three rules, one recommending A with confidence 0.8 and two recommending B with confidence 0.7 and 0.6 respectively. Traditional approaches would use the highest ranking rule and thus prioritize A over B. Rule aggregation combines the evidence for B and thus leads to recommending B over A.

Several authors propose methods to discover the most informative rules in a large collection of mined association rules, by either clustering rules that convey redundant information [38, 26, 20], or by pruning *non-interesting* rules [41, 3]. The overall idea is that the removal of rules will reduce the noise in the recommendations made using the remaining rules. However, in contrast to rule aggregation, and to the approach proposed in this paper, recommendations will be based on only part of the available evidence. It remains open to future work to investigate how this affects their relevance.

Parameter Tuning: Recent research highlighted that the configuration parameters of data mining algorithms have a significant impact on the quality of their results [27]. In the context of association rule mining, several authors have highlighted the need for thoughtfully studying how parameter settings affect the quality of generated rules [43, 25, 18].

Moonen et al. investigate how the quality of software change recommendation varied depending on association rule mining parameters such as transaction filtering threshold, history length, and history age [30, 29]. In contrast to that work, which focused on configurable parameters of the algorithm, this paper considers *non-configurable features* of the query, the change history, and the recommendation history.

IV. Generating Change Recommendations

A. Software Change Recommendation

Recommender (or recommendation) engines are information filtering systems whose goal is to predict *relevant* items for a specific purpose [32]. A common use of recommender systems is in marketing where these systems typically leverage

a shopper's previous purchases and the purchases of other shoppers to predict items of potential interest to the shopper.

In the context of software engineering, these systems typically leverage a developer's previous changes together with the changes made by other developers to predict items of interest. *Software change recommendation* takes as input a set of changed entities, referred to as a *change set* or *query*, and predicts a set of entities that are also likely in need of change. These entities may be any software *artifact*, such as files, methods, models, or text documents. The study described in this paper considers both files and methods as potential artifacts. We extract these artifacts from the version control history of a software system. Thus, software recommendation helps answering questions such as *"Given that files f_1 and f_2 and method m changed, what other files and methods are likely to need to be changed?"*

A common strategy for change recommendation is to capture the *evolutionary coupling* between entities [14]. In this application, entities are considered coupled iff they have changed together in the past. The key assumption behind evolutionary coupling is that the more frequently two or more entities change together, the more likely it is that when one changes, the others will also have to be changed. In the context of software change recommendation, evolutionary coupling is commonly captured using association rule mining [44].

B. Targeted Association Rule Mining

Association rule mining is an unsupervised learning technique aimed at finding patterns among items in a data set [1]. Association rule mining was first applied for market basket analysis, to find patterns (*rules*) describing items people typically purchase together. In this context, the data set is expressed as a list of transactions, where each transaction consists of a set of items. For example, in the domain of grocery stores, items likely include products such as "milk" and "cookies", and mining a rule such as "cookies" $\rightarrow$ "milk", uncovers the tendency of people who buy cookies (the rule *antecedent*) to also buy milk (the rule *consequent*). This provides valuable knowledge, for example suggesting that placing these grocery items in close proximity will increase sales.

Shortly after the introduction of association rule mining, Srikant et al. acknowledged that for most applications only a few specific rules are of practical value [36]. This led to the development of *constraint-based rule mining* where only rules that satisfy a given constraint are mined. Typically, constraints specify that particular items must be involved in the rule's antecedent. For example, consider a software engineer who recently made a change to file *x*. A constraint could specify that rule antecedents must contain *x*, thus limiting recommendation to those involving *x*. Constraints are usually specified by the user in the form of a *query*, at which the mining process is said to be *targeted*. The resulting *Targeted Association Rule Mining Algorithms* filter from the history all transactions unrelated to the query, producing a more focused set of rules. Doing so provides a significant reduction in execution time [36].

To rank the resulting rules, numerous *interestingness measures* have been proposed [24, 16]. Such measure attempt to quantify the likelihood that a rule will prove useful. The first interestingness measures introduced, *frequency*, *support*, and *confidence*, are also the most commonly used [1]. It is worth formalizing these three. Each is defined in terms of a *history*, $\mathcal{H}$, of transactions and an association rule $A \rightarrow B$, where A and B are disjoint sets of items. To begin with rule *frequency* is the number of times the antecedent and consequent have changed together in the history:

Definition 1 (Frequency)

$$\textit{frequency}(A \rightarrow B) \stackrel{\textit{def}}{=} |\{T \in \mathcal{H} : A \cup B \subseteq T\}|$$

Second, the *support* of a rule is its relative frequency with respect to the total number of transactions in the history:

Definition 2 (Support)

$$\textit{support}(A \rightarrow B) \stackrel{\textit{def}}{=} \frac{\textit{frequency}(A \rightarrow B)}{|\mathcal{H}|}$$

Finally, *confidence* is its relative frequency of the rule with respect to the number of historical transactions containing the antecedent A:

Definition 3 (Confidence)

$$\textit{confidence}(A \rightarrow B) \stackrel{\textit{def}}{=} \frac{\textit{frequency}(A \rightarrow B)}{|\{T \in \mathcal{H} : A \subseteq T\}|}$$

Support and confidence are often combined in the *support-confidence framework* [1], which first discards rules below a given support threshold and then ranks the remaining rules based on confidence. Thresholds were originally required to minimize the number of potential rules, which can quickly grow unwieldy. However, the constraints introduced by targeted association rule mining greatly reduce the number of rules and thus do not depend on a support threshold for practical feasibility.

C. Association Rule Mining Algorithms

A targeted association rule mining algorithm takes as input a query Q and restricts the antecedents of the generated rules to be various subsets of Q. The variation here comes from each algorithm attempting to best capture relevant rules. Consider, for example, the query $Q = \{a, b, c, d\}$. Potential rules include $a \rightarrow X$ and $b, c \rightarrow Y$. In fact, the set of possible antecedents is given by the power-set of Q.

One of most well known algorithms, ROSE, limits the number of rules by requiring that the antecedents are equal to the query, Q [44]. Thus for $\{a, b, c, d\}$, ROSE rules are all of the form $a, b, c, d \rightarrow X$, where X is only recommended if there exists one of more transactions where X changed together with *all* the items of the query. At the other end of the spectrum, CO-CHANGE partitions Q into singletons and considers only rules for each respective singleton [19]. Thus it produces rules of the form $a \rightarrow x$ and $b \rightarrow x$.

While ROSE and CO-CHANGE makes use of the largest and smallest possible rule antecedents, TARMAQ identifies the largest subset of Q that has changed with something else in

the past [34]. Thus TARMAQ may return the same rules as CO-CHANGE (when the history is made up of only two-item transactions) or the same rules as ROSE (when Q is a proper subset of at least one transaction). However, TARMAQ can also exploit partial matches (e.g., a history including transactions larger than two items, but none having Q as proper subset).

While it may not be immediately evident, TARMAQ is defined such that its recommendations are identical to those of ROSE, *when* ROSE is able to produce a recommendation. On the other hand, TARMAQ can produce recommendations far more often than ROSE [34]. As a result, we performed our empirical study using CO-CHANGE and TARMAQ, as the behavior of ROSE is subsumed by that of TARMAQ. As CO-CHANGE mines only singleton rules and TARMAQ potentially mines rules which maximize the antecedent with respect to the query, they together cover a large range of possible recommendations.

V. OVERVIEW OF MODEL FEATURES

This section introduces the features that we use to build our random forest classification model. We consider three categories of features:

- features of the query,
- features of the change history, and
- features of the recommendation.

It is worth noting that the features describing the query are known a priori, while features of the change history and the change recommendation are only known after a recommendation has been generated. Fortunately, a change recommendations can be generated in mere milliseconds and the corresponding feature set can therefore be included without incurring undo computational expense.

A. Features of the Query

Query Size: The first feature of a query we consider is simply its size. For example, if a single method is changed the query size is 1, if two different methods are changed, the query size is 2 and so on. Furthermore, some files may not be able to be parsed for fine-grained change information, changes to these files only ever increase the query size by 1. Throughout the rest of the paper we use the term *artifact* as a generic way of referring to both (unparsable) files and (parsable) methods. We hypothesize that *query size* may be important for relevance as when it increases one of the following is likely occurring: **(a)** The developer is working on a complex feature, requiring code updates in several locations. Here increased query size indicates *specificity*. **(b)** On the other hand, if a developer is not compartmentalizing the work and thus is working on multiple features at the same time, increased query size indicates *chaos* as unrelated artifacts are added to the same commit.

Number of Files Changed/Number of Methods Changed: We record the granularity of changes in two separate features: the number of files changed and the number of methods changed. For example, if the methods $m1$ and $m2$ change in the file $f1$, and the method $m3$ change in the file $f2$, we record that 3 methods and 2 files have changed. By including these metrics of query granularity, we aim to capture the *specificity* of the corresponding change recommendation.

Number of New Artifacts: If a new file or new method is included in a query, we know that nothing has changed together with it previously, thus from an *evolutionary perspective*, the new artifact is uncoupled from all other artifacts. The presence of new artifacts in combination with known artifacts adds uncertainty and is therefore considered as a feature.

B. Features of the Change History

Whenever an existing artifact, a, is changed, a list of *relevant transactions* can be extracted from the change history. A transaction is relevant if it contains the changed artifact. From these transactions mining algorithms identify other artifacts that typically changed with a, forming the basis for the resulting change recommendation.

Number of Relevant Transactions: The number of relevant transactions is the number of transactions with at least one artifact from the query. This metrics provides a measure of the *churn rate* (i.e., how often the artifacts change).

Mean Size of Relevant Transactions: While the number of relevant transactions tells us how often the artifacts found in a query change, it does not tell us how often they change with other artifacts, the *mean size of relevant transactions* attempts to capture this feature.

Mean/Median Age of Relevant Transactions: Two age related features included involve the change in dependencies between artifacts as software evolves (e.g., because of a refactoring) and *code decay*, where artifacts become "harder to change" (a known phenomena in long lived software systems [13]). The feature "age of relevant transactions" attempts to capture these two age related aspects. Note that two features are actually used: the *mean age* and the *median age*.

Overlap of Query and Relevant Transactions: If there are transactions that exhibit large overlap with the query, this might indicate highly relevant transactions [44]. We capture this through the "overlap percentage". Note that the percentage reports the single largest match rather than the mean.

C. Features of the Recommendation

A recommendation boils down to a prioritized list of association rules giving the *potentially affected* artifacts. However, different mining algorithms may return different lists. While the features described so far are independent of the mining algorithm, in this section we consider features that are aspects of the recommendation and thus the particular algorithm used.

Confidence and Support of Association Rules: A recommendation is constructed from *association rules* of the form A $\rightarrow$ B. Here "*A*" includes changed artifacts while "*B*" the recommended artifacts. To be able to distinguish rules, weights can be provided through *interestingness measures*. One way of providing these weights uses the support and confidence interestingness measures (Definitions 2 and 3). Traditionally, interestingness measures are used in isolation to judge whether a recommendation is relevant [44, 40, 34]. In this paper we extend their use by considering aggregates of the top 10 rules

in order to indicate recommendation relevance. We include three aggregates: The *top 10 mean confidence*, the *top 10 mean support* and the *maximum confidence*. Each feature is meant to capture the likelihood of whether there exist at least one relevant artifact in the top ten.

Number of Association Rules: If a recommendation consists of a large number of rules, two non-mutually exclusive situations may exist: **(a)** the query is large and the contained artifacts have changed with something else in the past, or **(b)** at least one artifact of the query has changed with a large number of other artifacts in the past. In either case, a large recommendation is a symptom of non-specificity and may thus prove a valuable feature for classifying true negatives.

VI. Experiment Design

Our empirical study is designed to answer one primary question: *can we predict if a recommendation contains relevant artifacts?* To answer this question we generate a large *recommendation oracle*, over which we train random forest classification models using the features described in section V. Finally, we evaluate the resulting models by comparing their performance against two confidence based predictions of relevance. These aim to function as a baseline for whether a developer would act upon a given recommendation.

1) **Maximum Confidence:** a recommendation is predicted as relevant if the confidence of the artifact with the highest confidence is larger than a given threshold.
2) **Mean Confidence 10:** a recommendation is predicted as relevant if the mean confidence of the top ten artifacts is greater than a given threshold.

The rationale behind *maximum confidence* mimics a developer who is only willing to consider a recommendation's highest ranked artifact, while that of *mean confidence 10* mimics a developer who is willing to consider the recommendation's top ten artifacts.

Our study encompasses change recommendations generated from the change history of sixteen different software systems with varying characteristics. Two of these systems come from our industry partners, *Cisco* and *KM*. *Cisco* is a worldwide leader in the production of networking equipment, We analyze the software product line for professional video conferencing systems developed by *Cisco*. *KM* is a leader in the production of systems for positioning, surveying, navigation, and automation of merchant vessels and offshore installations. We analyze the common software platform *KM* uses across various systems in the maritime and energy domain.

The other fourteen systems are the well known open-source projects reported in Table I along with change history statistics illustrating their diversity. The table shows that the systems vary from medium to large size, with over $280\,000$ unique files in the largest system. For each system, we extracted up to the $50\,000$ most recent transactions. This number of transactions covers vastly different time spans across the systems, ranging from almost twenty years in the case of HTTPD, to a little over ten months in the case of the Linux kernel. In the table, we report the number of unique files changed throughout the $50\,000$ most recent transactions, as well as the the number of unique artifacts changed. These artifacts include, in addition to file-level changes, method-level changes for certain languages.[1] Finally, the last column of Table I shows the programming languages used in each system, as an indication of heterogeneity.

The remainder of this section first explains the setup used to generate change recommendations using Co-Change and Tarmaq. It then details how these recommendations are used to train and evaluate models for relevance prediction. The results of our study are presented in section VII.

A. Generating Change Recommendations

Establishing the Ground Truth: The change history of a software system exactly describes, transaction by transaction, how to (re)construct the current state of the system. Consequently, we can assume the majority of the time that each transaction has some intention behind it, and that the changes in the transaction have some meaningful relation to each other. In fact, if this assumption is completely misguided, recommendations based on change histories would degenerate to random prediction, which is clearly not the case.[2] Thus, given a subset Q of transaction C, a good recommendation algorithm should identify the complement $E = C/Q$ as the set of impacted items. Here E captures the ground truth on whether a change recommendation is truly relevant because it includes those artifacts that actually changed alongside Q. For this reason, E is used when evaluating the change recommendation generated for query Q.

Sampling Strategy: Construction of the change scenarios involves two steps:

- sampling transactions and
- generating queries from each sampled transaction.

We start by fetching the $50\,000$ most recent transactions from each subject system. From these, we then determine the $10\,000$ most recent transactions whose size is between 2 and 300. The minimum number, 2, ensures that there is always a potential impact set for any given query, while the maximum number, 300, covers at least 99% of all transactions while aiming to omit large changes such as licensing changes. From each sampled transaction C, the impact set E is randomly determined, but ensured to consist of at least ten items.

Ranking Rules: The rules mined by each algorithm all share the property that their support is at least one because we operate with an absolute minimal support constraint. By including "all rules" like this, we ensure that both high frequency as well as low frequency (rare) rules are included in the recommendations [39].

The support measure is not otherwise used for ranking. When support is used for ranking, special care needs to be taken as rare (infrequent) rules always rank lower then more frequent rules. This is a result of the *downward closure*

[1] We currently extract method-level change information from files of C, C++, C#, and Java code.

[2] The chance of randomly predicting a correct method is $1/(\text{number of methods})$, which for any sizable system approaches zero.

TABLE I
CHARACTERISTICS OF THE EVALUATED SOFTWARE SYSTEMS (BASED ON OUR EXTRACTION OF THE LAST 50 000 TRANSACTIONS FOR EACH).

Software System	History (in yrs)	Number of unique files	Number of unique artifacts	Avg. transaction size (artifacts)	Languages used*
CPython	12.05	7725	30090	4.52	Python (53%), C (36%), 16 other (11%)
Mozilla Gecko	1.08	86650	231850	12.28	C++ (37%), C (17%), JavaScript (21%), 34 other (25%)
Git	11.02	3753	17716	3.13	C (45%), shell script (35%), Perl (9%), 14 other (11%)
Apache Hadoop	6.91	24607	272902	47.79	Java (65%), XML (31%), 10 other (4%)
HTTPD	19.78	10019	29216	6.99	XML (56%), C (32%), Forth (8%), 19 other (4%)
Liferay Portal	0.87	144792	767955	29.9	Java (71%), XML (23%), 12 other (4%)
Linux Kernel	0.77	26412	161022	5.5	C (94%), 16 other (6%)
MySQL	10.68	42589	136925	10.66	C++ (57%), C (18%), JavaScript (16%), 24 other (9%)
PHP	10.82	21295	53510	6.74	C (59%), PHP (13%), XML (8%), 24 other (20%)
Ruby on Rails	11.42	10631	10631	2.56	Ruby (98%), 6 other (2%)
RavenDB	8.59	29245	47403	8.27	C# (52%), JavaScript (27%), XML (16%), 12 other (5%)
Subversion	14.03	6559	46136	6.36	C (61%), Python (19%), C++ (7%), 15 other (13%)
WebKit	3.33	281898	397850	18.12	HTML (29%), JavaScript (30%), C++ (26%), 23 other (15%)
Wine	6.6	8234	126177	6.68	C (97%), 16 other (3%)
Cisco	2.43	64974	251321	13.62	C++, C, C#, Python, Java, XML, other build/config
KM	15.97	35111	35111	5.08	C++, C, XML, other build/config

* languages used by open source systems are from http://www.openhub.net, percentages for the industrial systems are not disclosed.

property: any subset of a rule must have equal or larger support relative to the origin rule [2]. For example, given the two rules $r_1 = \{a\} \rightarrow \{x\}$ and $r_2 = \{a, b\} \rightarrow \{x\}$, r_2 cannot have higher support than r_1.

By using the confidence measure to rank rules, both rare and non-rare rules may be ranked highly. Still, the frequency of a pattern continues to inform the relevancy of a rule. To this end, recall that *the top 10 mean support* is included as a feature in our prediction model.

B. Evaluation of Relevance Prediction

Blocked Cross-Validation: *Last block* validation is a frequently used scheme for evaluating prediction models. Here the data is split into two blocks, with the first block being used to train the model and the second block being used to evaluate it. This setup has the advantage of respecting the temporal nature of the data. However, a drawback is that not all data is used for both training and prediction [4]. To address this a traditional cross-validation setup may be used. However, doing so violates the temporal nature of time series data, and, in the worst case, may invalidate the results [4]. Because in time series data future data naturally depends on prior data, we use *blocked cross-validation* to preserve the temporal order between training and evaluation. In our configuration we partition each set of transactions into ten equally sized blocks, preserving temporal order between blocks. We then train nine random forest classification models for each software-system and mining algorithm combination. As illustrated in Figure 1, each random forest is trained on an incrementally larger subset of the available recommendations. In total, 16 systems $*$ 2 algorithms $*$ 9 forests $= 288$ random forests are trained.

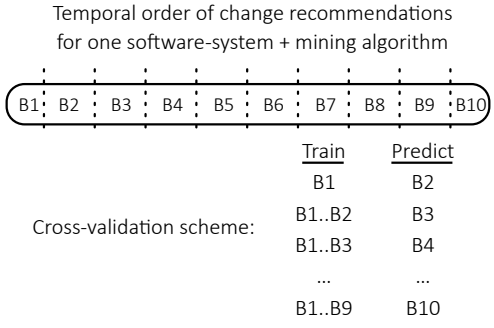

Fig. 1. The blocked cross-validation scheme used in our study. Notice that all blocks except B1 and B10 are used for both training and prediction

Measuring Relevance: The *random forest* and *confidence based* classification models have probabilistic interpretations. The confidence interestingness measure itself is given by the conditional likelihood $P(B|A)$, where B is the recommended artifact and A is one or more artifacts that changed (i.e., artifacts from the query) [1]. The maximum and mean confidence models use this information to capture the likelihood that a developer will act upon a given change recommendation. Here a "0" indicates very unlikely while a "1" indicates very likely. In the case of random forests, the likelihood is the result of the votes obtained from the collection of trees making up the random forest [7]. Each decision tree casts a single vote. As each tree is built from a random selection of change recommendations, the end likelihood is an indication of internal consistency within the data set for a particular scenario. In other words, if a certain scenario always results in a certain outcome, it is very likely that similar new scenarios will have the same outcome. Finally, in the *evaluation sets* used throughout our study we encode the possible classes in a similar way, the binary options are either 0 (not correct in top 10) or 1 (correct in top 10).

VII. RESULTS AND DISCUSSION

We organize the discussion of our results around the two research questions proposed in section V. Throughout this section we will consider prediction performance for each individual software system, and for each of the mining algorithms: CO-CHANGE and TARMAQ. By doing so we get an

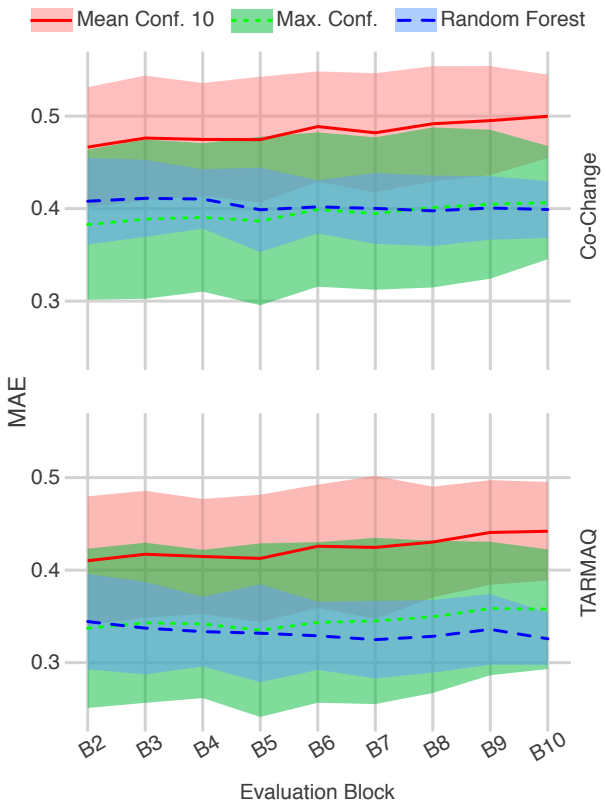

Fig. 2. Descriptive view of errors using the Mean Absolute Error (MAE). Each line shows the mean MAE over all 16 systems, and the ribbon shows the standard deviation.

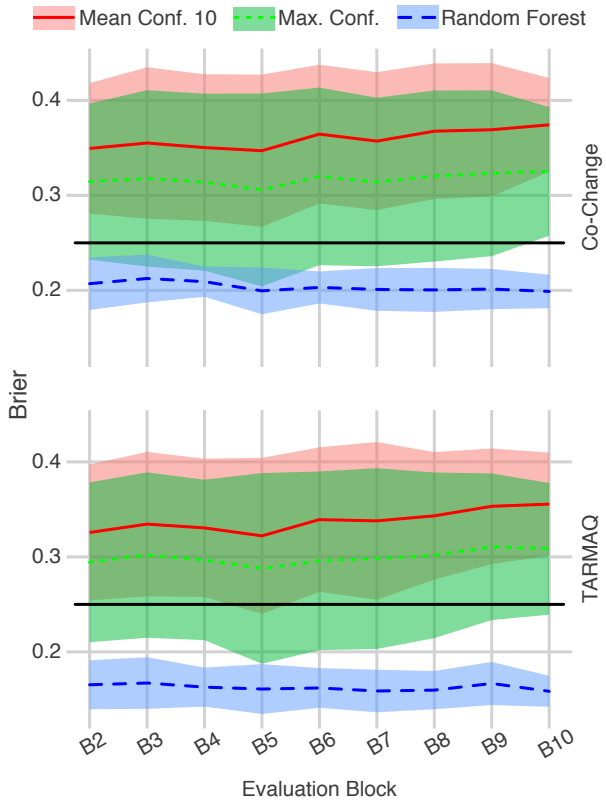

Fig. 3. Accuracy of classification models using Brier scores. Each line shows the mean MAE over all 16 systems, and the ribbon shows the standard deviation. Models below the horizontal black line tend to classify correctly with regards to a 0.5 classification threshold.

indication of how generalizable the results are to other systems and other algorithms. In the following we briefly introduce each performance metric before presenting the corresponding results.

A. RQ 1: Comparison to Confidence as a Relevance Predictor

While comparing the three classification models, we focus on two aspects of their relevance predictions:

1) the *error* with respect to the actual relevance, and
2) the *performance* across different classification thresholds.

We start with a descriptive view of the errors exhibited by each classification model, for this we use the Mean Absolute Error (MAE). In our case, MAE measures the mean absolute difference between the *actual* and *predicted* value for whether there is a correct artifact in the top ten artifacts of a recommendation. For example, given a certain feature as input, the random forest might give the output *0.67*, indicating a 67% likelihood that the resulting recommendation will be relevant. If in actuality, we know that for this scenario there is indeed a correct artifact in the top ten the *prediction error* would be $1 - 0.67 = 0.33$ because we encode "knowing" as "1". Note that lower is better. Figure 2 shows the MAE across the 16 software systems for the two mining algorithms. To reduce clutter, we present summarized information where each line is the mean over all systems and the ribbon indicates the standard deviation. For this first look at the data we have also preserved results for each *evaluation block*. This enables a view into error fluctuations across time. First, there is no apparent overall trend across the evaluation blocks. This is good news as it provides evidence that the analysis is stable across time. This is also supported by fitting linear regression lines (left out to minimize clutter). The *random forest* model shows less variance in error across systems and algorithms, while for some systems the *maximum confidence* model exhibits less overall error. For the change recommendations where the actual relevance was 1 (Correct in Top 10), the *maximum confidence* model frequently matches the prediction exactly and therefore minimizes the error for these recommendations.

In terms of *accuracy* of each classification model a *proper scoring rule* must be used [8]. For proper scoring rules, the maximum score is achieved if the *prediction distribution* exactly matches the *actual distribution*. One such scoring rule is the *brier score*. In the case of binary classification, which is our task, the brier score is the *mean squared error*:

$$BS = \frac{1}{N}\sum_{i=1}^{N}(p_i - a_i)^2$$

Where p_i is the predicted relevance for scenario i, and a_i is the actual relevance (1 or 0). Figure 3 presents the Brier scores for each classification model across each evaluation block, software system, and mining algorithm. Note that lower

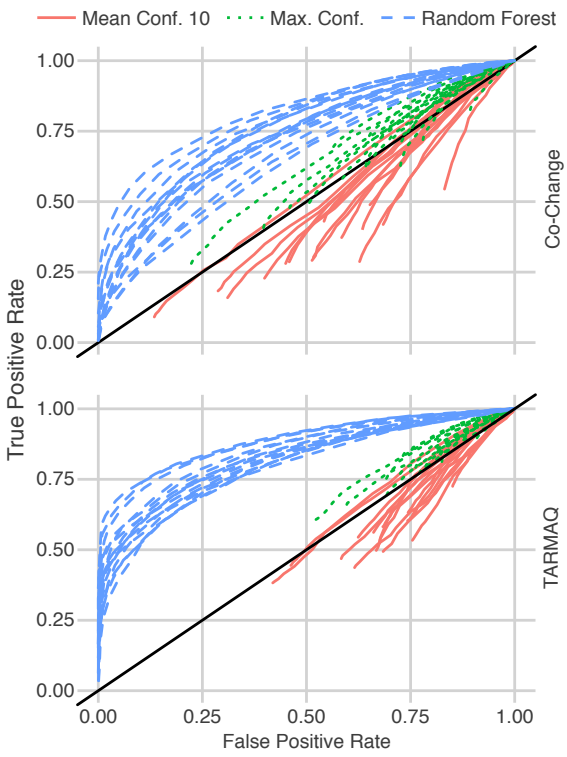

Fig. 4. ROC curves for the prediction models trained for each software system and algorithm.

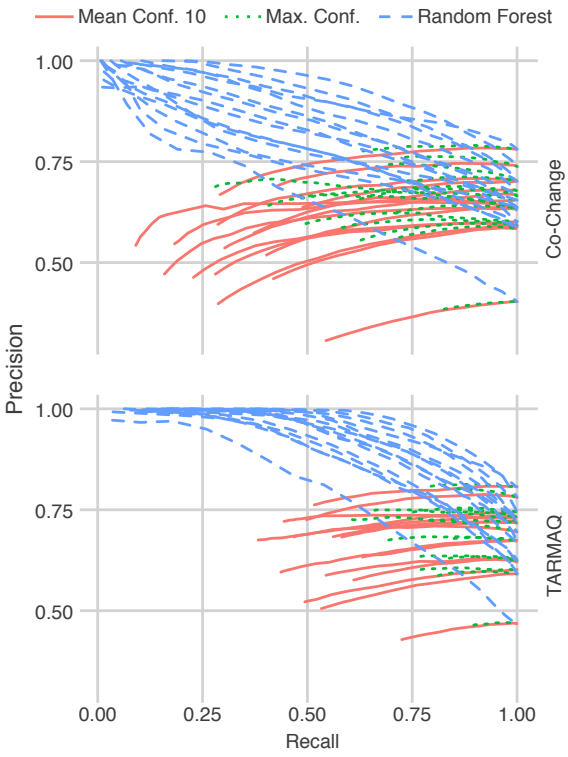

Fig. 5. Precision/Recall curves for the prediction models trained for each software system and algorithm.

is better. In the figure, the horizontal black line at y = 0.25 indicates the brier score of a *neutral classification model*. A neutral model always makes the prediction 0.5, where relevance and non-relevance are equally likely. Brier scores below the line indicate a prediction model that tends to be on the "right side of the midpoint". We clearly see the random forest model being consistently more accurate across algorithms and software systems.

While the *error* informs us about the overall fit of a prediction model, it does not capture performance across different *classification thresholds*. When classification models are used in practice, thresholds must be set to balance true positives against false positives and false negatives. To investigate the ability of our three prediction models in these terms, we consider the ROC curve and the Precision/Recall curve. First, the ROC curve in Figure 4 plots the True Positive Rate ($TPR = \frac{TP}{TP+FN}$) against the False Positive Rate ($FPR = \frac{FP}{FP+TN}$) at classification thresholds from 0 to 1. It is immediately apparent that the confidence based models *do not meaningfully respond to different classification thresholds*, as data points are not evenly spread across the x-axis. Furthermore, there are strong linear relationships between their TPRs and FPRs. This was also reflected in corresponding Pearson correlation coefficients, where all coefficients were calculated to be *0.97* or higher. Intuitively, the effect we see is that as the classification threshold is lowered, the confidence models for recommendation relevance classify comparably increased amounts of recommendations both correctly and incorrectly. For example, both TPR and FPR increase similarly. Furthermore, observe that the range and domain of the TPR and FPR for the confidence models do not fully extend between 0 and 1. This is the result of a high percentage of change scenarios being given the relevance "1", and these scenarios being evenly split between True Positives (TPs) and False Positives (FPs). In other words, the lack of even distribution of data-points results in less variation in TPR and FPR, which again is reflected in the range and domain. To further support our findings we compared the partial Area Under the Curve (pAUC) between the ROC of the random forests and each of the confidence based models. We used `roc.test` from the R package `pROC`, the significance test is based on bootstrapped sampling of multiple AUCs and computing their difference. Across all software systems and for both CO-CHANGE and TARMAQ the pAUC were significantly larger ($p < 0.05$) for the random forest model. Thus, the random forest classifier consistently provide better estimates of relevance across various thresholds compared to the purely confidence based methods explored.

As laid out earlier, relevance prediction can be performed in a *background* thread that only notifies the developer when there is a high likelihood for a true positive recommendation. In this application, the positive class (correct in top ten) is therefore of the most interest. Notifications that there are

no relevant artifacts would be of less use. With this view, the *precision* ($\frac{TP}{TP+FP}$) of the classification models become imperative. The task is to find an appropriate classification threshold that makes true positives likely (high precision), while still maintaining practicality in that recommendations can be regularly made (high recall). Figure 5 shows the precision/recall curves for our three prediction models for each software system and mining algorithm. First, the abnormality in slope, range and domain for the confidence models can again be attributed to the weak connection to threshold-changes. Furthermore, while one usually expects a decrease in precision as recall increases, this does not necessarily *need* to be the case. The trends for the confidence models in Figure 5 are the result of having slightly higher concentrations of positive classifications than negative classifications on lower thresholds. As thresholds are lowered further, more recommendations become TPs, and the ratio between TPs and FPs actually increases. An implication of this is that at least for the confidence measure, its value cannot be used directly as an indication of relevancy.

Turning to the random forest models, these exhibit greater defined behavior, where negative classifications are primarily located in lower likelihood thresholds, thus precision decreases as recall increases. In terms of recommending concrete classification thresholds for our random forest model we suggest that this should be adjusted with respect to the *system domain knowledge* of the developer for which the recommendation is made. In Table II we have provided the mean precision and recall across software systems for the example thresholds at 0.5 and 0.9. As developers become more acquainted with a system, they should also be able to better differentiate relevant and non-relevant recommendations. As such, experienced developers might afford a higher rate of recall at the cost of lower precision. A classification threshold of 0.50 becomes reasonable for this group, assuming TARMAQ is used. For inexperienced developers one wants to minimize confusion, and therefore maximize the precision of change recommendations. Thus, a threshold such as 0.9 might be appropriate for these developers, resulting in change recommendations only having false positives in about 1 to 3 per 100 recommendations.

B. RQ 2: Analysis of Features

Having empirically established that the random forest classifications models are superior at predicting change recommendation relevance, we next consider which features bring the most value to the models. Breiman introduced the concept of *variable importance* for random forests [7]. Once the decision trees of the random forests have been built, the process can self-assess the importance of each feature. The basic idea is to observe if randomly permuting a feature changes prediction performance [7]. Averaging the accuracy changes over all trees gives the *mean decrease in accuracy* (when permuted) for each feature.

TABLE II
EXAMPLES OF PRECISION AND RECALL FOR THE RANDOM FOREST CLASSIFICATION MODEL. THE STANDARD DEVIATION (SD) CAPTURES FLUCTUATIONS BETWEEN SOFTWARE SYSTEMS.

Classification Threshold		0.5			0.9		
Algorithm		Mean		SD	Mean		SD
CO-CHANGE	Precision	0.775	±	0.075	0.971	±	0.020
	Recall	0.684	±	0.046	0.100	±	0.074
TARMAQ	Precision	0.868	±	0.062	0.993	±	0.008
	Recall	0.735	±	0.032	0.277	±	0.102

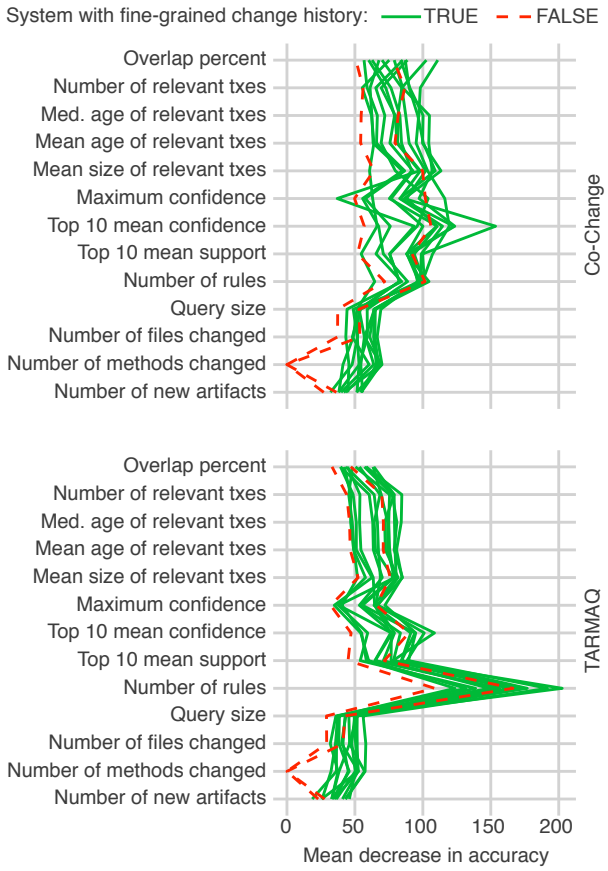

Fig. 6. Feature importance as determined by mean decrease in accuracy. Each line represents a separate software system, the line color & style indicates if a fine-grained change history was available.

The corresponding plot for the features included in our model is provided in Figure 6. For two of the studied software systems (KM and Rails), we only have file-level change information. These two systems are shown using the dashed (red) lines. Naturally, permuting the "*Number of methods changed*" feature does not change accuracy for these two systems, as the value is always 0, as reflected in Figure 6.

To begin with TARMAQ was constructed to produce a focused recommendation that matches the query as closely as possible [34]. This is evident in the figure where the "*Number of rules generated*" being an essential feature for TARMAQ. Thus for TARMAQ, variation in the number of rules generated meaningfully correlates with recommendation relevance; if a large subset of the query has changed with something else in the past, this results in fewer possible rule antecedents and thus fewer rules, which evidently increases the likelihood of

a relevant recommendation. For CO-CHANGE this feature has less importance. For the remaining features, Figure 6 shows a rather clear picture; the query based features (the bottom four) are the least important, the attributes they represent are better captured by other features. Of the interestingness measure based features the *"Top 10 mean confidence"* proved most useful. Finally, all history related features are comparably important.

The high degree of co-variance between software systems suggests that *model transfer* to other systems is viable. That is, classification models learned on one or more systems, should be reusable for a new (unknown) system. If this can be shown to work reliably, systems that are early in their evolution can still benefit from models generated from more mature systems. However, care must be taken to adapt the feature variation of the random forest to fit the variation found in new software system.

C. Threats to Validity

Implementation: We implemented and thoroughly tested all algorithms and our technique for model classification in Ruby and R respectively, However, we can not guarantee the absence of implementation errors.

Variation in Software Systems: We have sought to obtain generalizable results by evaluating over a range of heterogeneous software systems, however, we also uncovered that relevance prediction performance varies between systems. Future work should investigate the effects of system characteristics on prediction performance.

Mining Algorithms Studied: In our evaluation we have studied classification models for recommendations generated using both the CO-CHANGE and TARMAQ mining algorithms. For both algorithms we achieve strong results. However, we acknowledge that comparable results cannot be guaranteed for other mining algorithms.

Using Other Interestingness Measures: In our study we focused on the *confidence* interestingness measure, thus our results are limited to this measure. As such, future work should investigate the use of other interestingness measures, both for comparison to the random forest predictor, as well as being included as part of the model. We also envision that a variation of the relevance prediction presented here might be an interestingness measure recommender, thus essentially creating an ensemble of measures where the most relevant is used at a given time.

Recommendations Used for Training and Evaluation: We train and evaluate our classification model over a constructed set of change recommendations. Each recommendation is the result of executing randomly sampled a query from an existing transaction where the complement of the query and the source transaction is used as the *expected outcome*. However, this approach does not account for the actual order in which changes were made before they were committed to the versioning system. As a result, it is possible that queries contain elements that were actually changed later in time than elements of the expected outcome. As such, we cannot guarantee that recommendations used in training and evaluation exactly reflect each system's evolution.

VIII. CONCLUDING REMARKS

Change recommendation is clearly an asset to a software developer maintaining a complex system. However, its practical adoption faces two challenges: (a) recommendations must be both accurate and relevant. We believe that both challenges can be effectively addressed using *historically proven change recommendations*.

This paper shows that random forest classification using the 12 features that describe aspects of the *change set* (query), *change history* (transactions) and generated *change recommendations* is viable. We compare the random forest model against the state-of-the-art (based on *confidence*). We evaluate our approach in a large empirical study across 16 software systems and two change recommendation algorithms. Our findings are as follows:

Finding 1: The random forest classification model consistently outperforms the confidence based models in terms of accuracy (Brier scores).

Finding 2: The random forest classification model achieves significantly larger area under ROC curve than both confidence based models.

Finding 3: While the confidence measure is appropriate for ranking of artifacts, the values themselves should not be interpreted in isolation as overall estimates of recommendation relevance.

Finding 4: The importance of model features may varies between algorithms. For example, the relevance of TARMAQ recommendations is best predicted by considering the number of rules generated, while this feature is less important for CO-CHANGE. However, the remaining features studies showed consistent importance between algorithms.

Directions for Future Work

Looking forward, it would be of interest to study the effects of using stricter and looser definitions of relevance (e.g., relevant if correct in top three vs top twenty). Furthermore, rather than classification, relevance can also be studied as a regression problem, predicting other recommendation metrics such as precision, recall, average precision etc. In addition, we plan to study the behavior of relevance classification models over other interestingness measures, and investigate the viability of model transfer between software systems. Finally, we plan to look into improving the classification model by including features such as the number of relevant transactions authored by core contributors vs occasional contributors, the weighting recent relevant transactions higher than older transactions, and the text similarity scores between change set and relevant transactions.

ACKNOWLEDGMENT

This work is supported by the Research Council of Norway through the EvolveIT project (#221751/F20) and the Certus SFI (#203461/030). Dr. Binkley is supported by NSF grant IIA-1360707 and a J. William Fulbright award.

REFERENCES

[1] R. Agrawal, T. Imielinski, and A. Swami. "Mining association rules between sets of items in large databases". In: *ACM SIGMOD International Conference on Management of Data*. ACM, 1993, pp. 207–216.

[2] R. Agrawal and R. Srikant. "Fast Algorithms for Mining Association Rules". In: *International Conference on Very Large Data Bases (VLDB)*. 1994, pp. 487–499.

[3] E. Baralis et al. "Generalized association rule mining with constraints". In: *Information Sciences* 194 (2012), pp. 68–84.

[4] C. Bergmeir and J. M. Benítez. "On the use of cross-validation for time series predictor evaluation". In: *Information Sciences* 191 (2012), pp. 192–213.

[5] D. Beyer and A. Noack. "Clustering Software Artifacts Based on Frequent Common Changes". In: *International Workshop on Program Comprehension (IWPC)*. IEEE, 2005, pp. 259–268.

[6] S. Bohner and R. Arnold. *Software Change Impact Analysis*. CA, USA: IEEE, 1996.

[7] L. Breiman. "Random Forests". In: *Machine Learning* 45.1 (2001), pp. 5–32.

[8] A. Buja, W. Stuetzle, and Y. Shen. "Loss functions for binary class probability estimation and classification: structure and application". 2005.

[9] R. Caruana and A. Niculescu-Mizil. "An empirical comparison of supervised learning algorithms". In: *Proceedings of the 23th International Conference on Machine Learning* (2006), pp. 161–168.

[10] W. Cheetham. "Case-Based Reasoning with Confidence". In: *European Workshop on Advances in Case-Based Reasoning (EWCBR)*. Lecture Notes in Computer Science, vol 1898. Springer, 2000, pp. 15–25.

[11] W. Cheetham and J. Price. "Measures of Solution Accuracy in Case-Based Reasoning Systems". In: *European Conference on Case-Based Reasoning (ECCBR)*. Lecture Notes in Computer Science, vol 3155. Springer, 2004, pp. 106–118.

[12] D. Cubranic et al. "Hipikat: a project memory for software development". In: *IEEE Transactions on Software Engineering* 31.6 (2005), pp. 446–465.

[13] S. Eick et al. "Does code decay? Assessing the evidence from change management data". In: *IEEE Transactions on Software Engineering* 27.1 (2001), pp. 1–12.

[14] H. Gall, K. Hajek, and M. Jazayeri. "Detection of logical coupling based on product release history". In: *IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 1998, pp. 190–198.

[15] H. Gall, M. Jazayeri, and J. Krajewski. "CVS release history data for detecting logical couplings". In: *International Workshop on Principles of Software Evolution (IWPSE)*. IEEE, 2003, pp. 13–23.

[16] L. Geng and H. J. Hamilton. "Interestingness measures for data mining". In: *ACM Computing Surveys* 38.3 (2006).

[17] A. E. Hassan and R. Holt. "Predicting change propagation in software systems". In: *IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2004, pp. 284–293.

[18] N. Jiang and L. Gruenwald. "Research issues in data stream association rule mining". In: *ACM SIGMOD Record* 35.1 (2006), pp. 14–19.

[19] H. Kagdi et al. "Blending conceptual and evolutionary couplings to support change impact analysis in source code". In: *Working Conference on Reverse Engineering (WCRE)*. 2010, pp. 119–128.

[20] S. Kannan and R. Bhaskaran. "Association Rule Pruning based on Interestingness Measures with Clustering". In: *Journal of Computer Science* 6.1 (2009), pp. 35–43.

[21] T.-d. B. Le and D. Lo. "Beyond support and confidence: Exploring interestingness measures for rule-based specification mining". In: *International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2015, pp. 331–340.

[22] T.-D. B. Le, D. Lo, and F. Thung. "Should I follow this fault localization tool's output?" In: *Empirical Software Engineering* 20.5 (2015), pp. 1237–1274.

[23] T.-D. B. Le, F. Thung, and D. Lo. "Predicting Effectiveness of IR-Based Bug Localization Techniques". In: *2014 IEEE 25th International Symposium on Software Reliability Engineering*. IEEE, 2014, pp. 335–345.

[24] P. Lenca et al. "On selecting interestingness measures for association rules: User oriented description and multiple criteria decision aid". In: *European Journal of Operational Research* 184.2 (2008), pp. 610–626.

[25] W. Lin, S. A. Alvarez, and C. Ruiz. "Efficient Adaptive-Support Association Rule Mining for Recommender Systems". In: *Data Mining and Knowledge Discovery* 6.1 (2002), pp. 83–105.

[26] B. Liu, W. Hsu, and Y. Ma. "Pruning and summarizing the discovered associations". In: *SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*. ACM, 1999, pp. 125–134.

[27] O. Maimon and L. Rokach. *Data Mining and Knowledge Discovery Handbook*. Ed. by O. Maimon and L. Rokach. Springer, 2010, p. 1383.

[28] K. McGarry. "A survey of interestingness measures for knowledge discovery". In: *The Knowledge Engineering Review* 20.01 (2005), p. 39.

[29] L. Moonen et al. "Exploring the Effects of History Length and Age on Mining Software Change Impact". In: *IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 2016, pp. 207–216.

[30] L. Moonen et al. "Practical Guidelines for Change Recommendation using Association Rule Mining". In: *International Conference on Automated Software Engineering (ASE)*. Singapore: IEEE, 2016.

[31] C. Parnin and A. Orso. "Are automated debugging techniques actually helping programmers?" In: *International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2011, p. 199.

[32] P. Resnick and H. R. Varian. "Recommender systems". In: *Communications of the ACM* 40.3 (1997), pp. 56–58.

[33] R. Robbes, D. Pollet, and M. Lanza. "Logical Coupling Based on Fine-Grained Change Information". In: *Working Conference on Reverse Engineering (WCRE)*. IEEE, 2008, pp. 42–46.

[34] T. Rolfsnes et al. "Generalizing the Analysis of Evolutionary Coupling for Software Change Impact Analysis". In: *International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2016, pp. 201–212.

[35] T. Rolfsnes et al. "Improving change recommendation using aggregated association rules". In: *International Conference on Mining Software Repositories (MSR)*. ACM, 2016, pp. 73–84.

[36] R. Srikant, Q. Vu, and R. Agrawal. "Mining Association Rules with Item Constraints". In: *International Conference on Knowledge Discovery and Data Mining (KDD)*. AASI, 1997, pp. 67–73.

[37] P.-N. Tan, V. Kumar, and J. Srivastava. “Selecting the right interestingness measure for association patterns”. In: *International Conference on Knowledge Discovery and Data Mining (KDD)*. ACM, 2002, p. 32.

[38] H. Toivonen et al. “Pruning and Grouping Discovered Association Rules”. In: *Workshop on Statistics, Machine Learning, and Knowledge Discovery in Databases*. Heraklion, Crete, Greece, 1995, pp. 47–52.

[39] M.-C. Tseng and W.-Y. Lin. “Mining Generalized Association Rules with Multiple Minimum Supports”. In: *Lecture Notes in Computer Science (LNCS)*. Vol. 2114. 2001, pp. 11–20.

[40] A. T. T. Ying et al. “Predicting source code changes by mining change history”. In: *IEEE Transactions on Software Engineering* 30.9 (2004), pp. 574–586.

[41] M. J. Zaki. “Generating non-redundant association rules”. In: *SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*. ACM, 2000, pp. 34–43.

[42] M. B. Zanjani, G. Swartzendruber, and H. Kagdi. “Impact analysis of change requests on source code based on interaction and commit histories”. In: *International Working Conference on Mining Software Repositories (MSR)*. 2014, pp. 162–171.

[43] Z. Zheng, R. Kohavi, and L. Mason. “Real world performance of association rule algorithms”. In: *SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*. ACM, 2001, pp. 401–406.

[44] T. Zimmermann et al. “Mining version histories to guide software changes”. In: *IEEE Transactions on Software Engineering* 31.6 (2005), pp. 429–445.