Predicting Defects Using Change Genealogies

Kim Herzig
Microsoft Research[†]
Cambridge, UK
kimh@microsoft.com

Sascha Just
Saarland University
Saarbrücken, Germany
just@st.cs.uni-saarland.de

Andreas Rau
Saarland University
Saarbrücken, Germany
rau@st.cs.uni-saarland.de

Andreas Zeller
Saarland University
Saarbrücken, Germany
zeller@cs.uni-saarland.de

***Abstract*—When analyzing version histories, researchers traditionally focused on single events: e.g. the change that causes a bug, the fix that resolves an issue. Sometimes however, there are *indirect effects* that count: Changing a module may lead to plenty of follow-up modifications in other places, making the initial change having an impact on those later changes. To this end, we group changes into *change genealogies*, graphs of changes reflecting their mutual dependencies and influences and develop new metrics to capture the spatial and temporal influence of changes. In this paper, we show that change genealogies offer good classification models when identifying defective source files: With a median precision of 73% and a median recall of 76%, change genealogy defect prediction models not only show better classification accuracies as models based on code complexity, but can also outperform classification models based on code dependency network metrics.**

***Index Terms*—Data mining; Predictive models; Software quality; Software engineering**

I. Introduction

Estimating and predicting software quality is a classic domain of analyzing version histories. The typical assumption is that modules that are *similar* to previously failure-prone modules would share the same bug-proneness, and therefore be predicted as likely to have bug fixes in the future. A similarity may be found due to fact that they share the same domain, the same dependencies, the same code churn, the same complexity metrics—or simply because they have a history of earlier bugs themselves. Sometimes, however, bugs are induced *indirectly:* Rather than being failure-prone by itself, a change in a module A may lead to changes in modules related to A, which in themselves may be risky and therefore induce bugs. Such indirect effects, however, would not be captured in A's metrics, which thus represent a local view of history, either spatially or temporally.

As an example, assume A is an abstract superclass of an authentication system. Now assume a change to A adds a new (flawed) authentication interface. This will have to be implemented in all subclasses of A; and any bugs later found will have to be fixed in these subclasses as well. All these bugs are caused by the initial change to A; yet, A will never show up as particularly bug-prone or be flagged as risky to change.

In this paper, we make use of *change genealogy graphs* to define a set of change genealogy network metrics describing the *structural* and *temporal* dependencies between change sets. Our assumption is that code changes and their impact on other, later applied code changes can differ substantially. We suspect that change sets that cause more dependencies to other, later applied change sets are likely to be more crucial with respect to the development process. The more dependencies a change set causes or relies on, the more complex the applied change and thus, the more critical the applied change. Consequently, we suspect code artifacts that got many crucial and central code changes applied to be more defect prone than others. Change genealogies thus would identify the interface change in module A as being the primary cause for the later bugs, and mark A as risky to change.

This is similar to the findings of Zimmermann and Nagappan who discovered that more central code artifacts tend to be more defect prone than others [30]; and indeed, these would mark A as potentially bug-prone because it is central. In our case, we do not measure the dependencies of the changed code artifacts but rather concentrate on the dependencies of the individual changes themselves, and can therefore determine how the initial change in A caused the later bug fixes in related modules. Specifically, we seek to answer the following research question:

> *Are change genealogy metrics effective in classifying source files to be defect-prone?*

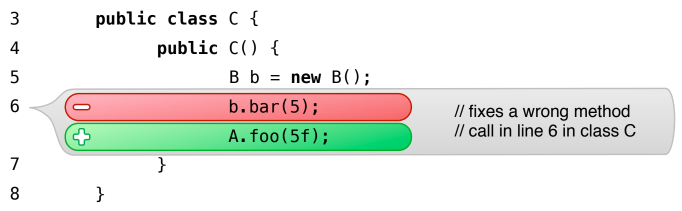
To answer this question, we ran experiments on four open source JAVA projects. The results show that defect prediction models based on change genealogy metrics can predict defective source files with precision and recall values of up to 80%. On median, change genealogy defect prediction models show a precision of 73% and a recall of 76%. Compared to prediction models based on code dependency network or code complexity metrics, change genealogy based prediction models achieve better precision and comparable recall values.

II. Background

A. Change Genealogies

Change Genealogies were first introduced by Brudaru and Zeller [6]. A change genealogy is a directed acyclic graph structure modeling dependencies between individual change sets. Change genealogies allow reasoning about the impact of a particular change set on other, later applied change sets. German et al. [10] used the similar concept of *change impact graphs* to identify change sets that influence the reported

[†]At the time of this study, Kim Herzig was PhD candidate at Saarland University. He is now employed at Microsoft Research Cambridge.



```
3       public class C {
4           public C() {
5               B b = new B();
6 -             b.bar(5);          // fixes a wrong method
  +             A.foo(5f);         // call in line 6 in class C
7           }
8       }
```

Fig. 1. Diff output corresponding to the table cell of column $CS4$ and row $File3$ shown in Figure 2. It also corresponds to the genealogy vertex $CS4$ shown in Figure 3.

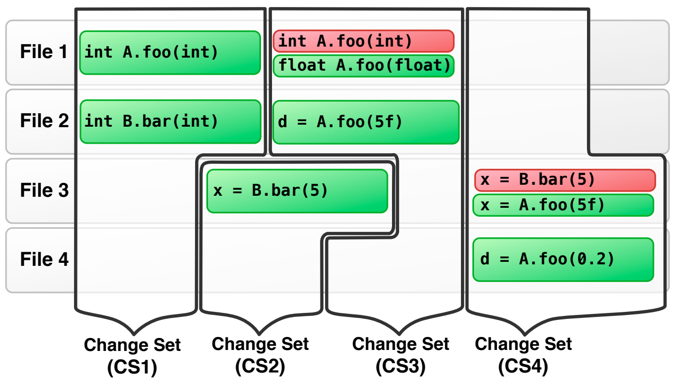


Fig. 2. We characterize change sets by method calls and definitions added or deleted. Changes depend on each other based on the affected methods.

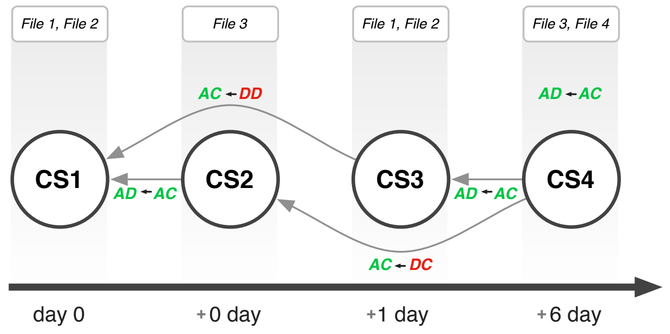


Fig. 3. Sample change genealogy derived from the change operations shown in Figure 2. $CS_i \rightarrow CS_j$ indicates that the change CS_j depends on CS_i.

location of a failure. Alam et al. [2] reused the concept of change dependency graphs [10] to show how changes build on earlier applied changes measuring the time dependency between both changes. In 2010, Herzig [15] used the original concept of change genealogies as defined by Brudaru and Zeller [6] to implement change genealogies modeling dependencies between added, modified, and deleted method definitions and method calls. Later, Herzig and Zeller [14] used method based change genealogy graphs to mine cause-effect chains from version archives using model checking. In this paper, we reuse the concept of genealogy graphs as defined and implemented by Herzig [12], [15].

Change Genealogies in a Nutshell

Change genealogy graphs model dependencies between individual change sets capturing how earlier changes enable and cause later ones. Dependencies between change sets are computed based on dependencies between added and deleted method definitions and method calls. Two change sets CS_N and the earlier change set CS_M depend on each other, if any of the applied change operations depend on each other:

- CS_N deletes a method definition added in CS_M,
- CS_N adds a method definition deleted in CS_M,
- CS_N adds a statement calling a method added in CS_M,
- CS_N deletes a method call added in CS_M.

For this purpose, we analyze the complete version history of a software project reducing every applied change set to a number of code *change operations* that added or deleted method calls (AC, DC) or added or deleted method definitions (AD, DD). The example change set shown in Figure 1 contains two change operations: one deleting the method call `b.bar(5)` and one adding `A.foo(5f)`. Method calls and definitions are identified using their full qualified name and absolute position within the source code.

The example change genealogy shown in Figure 3 corresponds to the artificial example history shown in Figure 2. Following the initial change set example in Figure 1 we can see that this change set causes two different dependencies for change genealogy vertex CS_4. Removing the method call to `B.bar(int)` makes CS_4 depending on CS_2 that added the just removed method call. CS_4 also depends on CS_3 containing a change operation deleting the method definition of `B.bar(int)`. Apart from dependencies between individual change sets, a change genealogy stores changed code artifacts (e.g. file names) as vertex annotations and the dependency types between vertices as edge annotations.

B. Network Metrics

Network metrics describing the dependency structure between individual code artifacts (e.g. source files) have shown to be powerful to express dependencies between source code artifacts such as methods and to predict software defects on file and package level [4], [24], [28], [30]. In this work, we use the concept of network metrics to express and measure dependency relations between change sets. Since these dependencies are already modeled within a change genealogy, we can reuse many network metrics used in previous studies.

C. Predicting Defects

The number of studies and approaches related to defect prediction is large and continues to grow. We reference only those approaches and studies that are closely related to this work. The given references are neither complete nor representative for the overall list of defect prediction models, their applications and related approaches.

One of the earliest attempts to predict defects was conducted by Basili et al. [3] using object-oriented metrics. Many studies investigated a large variety of different code metric types for defect prediction purposes. Ostrand et al. [22] used code metrics and prior faults to predict the number of faults for large industrial systems. Zimmermann et al. [31] demonstrated that higher code complexity leads to more defects. Besides code related metrics, there exist studies showing that change-related metrics [18], developer related metrics [23], organizational metrics [20] and process metrics [11] can be used to predict defect prone code artifacts.

The usage of code dependency information to build defect prediction models is not new either. Schröter et al. [26] used import statements to predict the number of defects for source files at design time. Shin et al. [27] and Nagappan and Ball [19] provided evidence that defect prediction models can improve when adding calling structure metrics.

Zimmermann and Nagappan [30] demonstrated that network metrics on code entity dependency graphs can be used to build precise defect prediction models. Code artifacts communicate with each used using method calls or shared variables. Modeling these communication channels results in a graph structure that can be used to apply network analysis on them. Later, Bird et al. [4] extended the set of network metrics by extending code dependency graph adding contribution dependency edges.

III. Change Genealogy Metrics

In Section II-A we briefly discussed the concept of change genealogies. Summarizing, change genealogies model dependencies (edges) between individual change sets (vertices). Similar to code dependency metrics [4], [30] we can use change set dependency graphs to define and compute change genealogy metrics describing the dependency structures between code changes instead of code artifacts.

Each change set applied to the software system is represented by a change genealogy vertex. Computing network metrics for each change genealogy vertex means to compute change set dependency metrics. Later, we will use this set of genealogy metrics to classify change sets as bug fixing or feature implementing using a machine learner and to predict defect-prone source code artifacts.

To capture as many of such dependency differences as possible, we implemented various genealogy dependency metrics of different categories.

A. EGO Network Metrics

Ego network metrics describe dependencies among change genealogy vertices and their direct neighbors. For every vertex we consider direct dependent or direct influencing change sets, only. Thus, this set of metrics measures the *immediate impact* of change sets on other change sets. Table I describes the implemented genealogy ego network metrics.

The metrics *NumDepAuthors* and *NumParentAuthors* refer to authorship of change sets. Code changes that fix a bug may dominantly depend on changes that were committed by the same developer. We suspect bug fixes to be fixed by code owners and that developers call their own functionality and methods more often than foreign ones. Thus, the number of authors of code change parents might be valuable. The last six metrics in Table I express temporal dependencies between change sets based on their commit timestamps.

TABLE I
Ego network metrics capturing direct neighbor dependencies.

Metric name	**Description**
NumParents	The distinct number of vertices being source of an incoming edge.
NumDefParents	The distinct number of vertices representing a method definition operation and being source of an incoming edge.
NumCallParents	The distinct number of vertices representing a method call operation and being source of an incoming edge.
NumDependants	The distinct number of vertices being target of an outgoing edge.
NumDefDependants	The distinct number of vertices representing a method definition operation and being target of an outgoing edge.
NumCallDependants	The distinct number of vertices representing a method call operation and being target of an outgoing edge.
AvgInDegree	The average number of incoming edges.
AvgOutDegree	The average number of outgoing edges.
NumDepAuthors	The distinct number of authors responsible for the direct dependents.
NumParentAuthors	The distinct number of authors that implemented the direct ascendants of this vertex.
AvgResponseTime	The average number of days between a vertex and all its children.
MaxResponseTime	The number of days between a vertex and the latest applied child.
MinResponseTime	The number of days between a vertex and the earliest applied child.
AvgParentAge	The average number of days between a vertex and all its parents.
MaxParentAge	The number of days between a vertex and the earliest applied parent.
MinParentAge	The number of days between a vertex and the latest applied parent.

B. GLOBAL Network Metrics

Global network metrics describe a wider neighborhood. Most global network metrics described in Table II can be computed for the global universe of vertices and dependencies. For practical reasons, we limited the maximal traversal depth to a maximal depth of five.

Metrics counting the number of global descendants or ascendants express the indirect impact of change sets on other change sets and how long this impact propagates though history. The set of inbreed metrics expresses dependencies between a change set and its children in terms of common ascendants or descendants. Code changes that depend on nearly the same earlier change sets as its children might indicate reverted or incomplete changes.

C. Structural Holes

The concept of structural holes was introduced by Burt [7] and measures the influence of actors in balanced social networks. When all actors are connected to all other actors, the network is well-balanced. As soon as dependencies between individual actors are missing ("structural holes") some actors are in advanced positions.

TABLE II
GLOBAL NETWORK METRICS.

Metric name	Description
NumParents†	The distinct number of vertices being part of on incoming path.
NumDefParents†	Like *NumParents* but limited to vertices that change method definitions.
NumCallParents†	Like *NumParents* but limited to vertices that change method calls.
NumDependants†	The distinct number of vertices being part of on outgoing path.
NumDefDependants†	Like *NumDependants* but limited to vertices that change method definitions.
NumCallDependants†	Like *NumDependants* but limited to vertices that change method calls.
Inbreed metrics:	
NumSiblingChildren	The number of children sharing at least one parent with this vertex.
AvgSiblingChildren	The average number of parents this vertex and its children have in common.
NumInbreedParents	The number of grandparents also being parents.
NumInbreedChildren	The number of grandchildren also being children.
AvgInbreedParents	The average number of grandparents also being parent.
AvgInbreedChildren	The average number of grandchildren also being children.

† maximal network *traversal depth* set to 5.

TABLE III
STRUCTURAL HOLES METRICS SIMILAR TO BURT [7].

Metric name	Description
EffSize	The number of vertices that connected to this vertex minus the average number of ties between these connected vertices.
InEffSize	The number of vertices that connected by incoming edges to this vertex minus the average number of ties between these connected vertices.
OutEffSize	The number of vertices that connected by outgoing edges to this vertex minus the average number of ties between these connected vertices.
Efficiency	norms EffSize by the number of ego-network vertices.
InEfficiency	norms InEffSize by the number of ego-network vertices.
OutEfficiency	norms OutEffSize by the number of ego-network vertices.

The *effective size* of a network is the number of change sets that are connected to a vertex minus the average number of ties between these connected vertices. The *efficiency* of a change set is its *effective size* normed by the number of vertices contained in the ego network. The higher the metric values for these metrics the closer the connection of a change set to its ego network. Table III lists the complete list of used structural hole metrics.

IV. DATA COLLECTION

The goal of our approach is to predict defective source files using change genealogy network metrics. To put the accuracy of such prediction models into perspective we compare our models against models based on code complexity [31] and models based on code dependency models by Zimmermann and Nagappan [30] (referred to as Z&N for sake of brevity).

TABLE IV
PROJECTS USED FOR EXPERIMENTS.

	HTTPCLIENT	JACKRABBIT†	LUCENE	RHINO
History length	6.5 years	8 years	2 years	13 years
Lines of Code	57k	66k	362k	56k
# Source files	570	687	2,542	217
# Code changes	1,622	7,465	5,771	2,883
# Mapped BUGs[1]	92	756	255	194
# Mapped RFEs[2]	63	305	203	38
	Change genealogy details			
# vertices	973	4,694	2,794	2,261
# edges	2,461	15,796	8,588	9,002

† considered only JACKRABBIT `content repository` (*JCR*).
[1] refers to mapped fixes to bug reports
[2] refers to implemented/enhanced feature requests

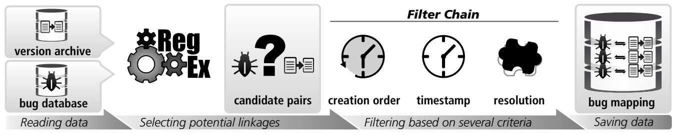


Fig. 4. Process of linking change sets to bug reports.

All models are evaluated on the four open-source JAVA projects HTTPCLIENT, LUCENE, RHINO, and JACKRABBIT. The projects differ in size from small (HTTPCLIENT) to large (LUCENE) allowing us to investigate whether the classification and prediction models are sensitive to project size. All projects are known in the research community and follow the strict and industry-like development processes of APACHE and MOZILLA. A brief summary of the projects and their genealogy graphs is presented in Table IV. Change genealogy graphs contain approximately as many vertices as applied change sets. The difference in the number of vertices and the number of change sets is caused by change sets that do not add or delete any method definition or method call (e.g. modifying the build system or code documentation).

A. Bugs

For our approach we need to identify bug fixing change sets and the total number of applied bug fixes per source file. Mapping bug report to change sets, we can associate change sets with modified source files and count the distinct number of fixed bug reports per source file.

To associate change sets with bug reports, we followed a similar approach as described by Zimmermann et al. [31] (see also Figure 4):

1) Bug reports and change sets are read from the corresponding bug tracking system and version archive.
2) In a pre-process step we select potential candidates using regular expressions such as to search for po-

TABLE V
SET OF CODE COMPLEXITY METRICS USED.

Identifier	Description
NOM	Total number of methods per source file.
LCOM	Lack of cohesion of methods in source file.
AVCC	Cyclomatic complexity after McCabe [17].
NOS	Number of statements in source file.
INST$^{\Sigma}$	Number of class instance variables.
PACK	Number of imported packages.
RCS$^{\oslash}$	Total response for class (#methods + #distinct method calls).
CBO$^{\oslash}$	Couplings between objects [8].
CCML	Number of comment lines.
MOD$^{\oslash}$	Number of modifiers for class declaration.
INTR$^{\Sigma}$	Number of implemented interfaces.
MPC$^{\oslash}$	Represents coupling between classes induced by message passing.
NSUB$^{\Sigma}$	Number of sub classes.
EXT$^{\Sigma}$	Number of external methods called.
FOUT$^{\Sigma}$	Also called *fan out* or *efferent coupling*. The number of other classes referenced by a class.
F-IN$^{\Sigma}$	Also called *fan in* or *afferent coupling*. The number of other classes referencing a class.
DIT$^{\wedge}$	The maximum length of a path from a class to a root class in the inheritance structure.
HIER$^{\Sigma}$	Number of class hierarchy methods called.
LMC$^{\Sigma}$	Number of local methods called.

Σ aggregated using the sum of all metric values of lower order granularity. $\oslash$ aggregated using the mean value. $\wedge$ aggregated using the max value. Otherwise directly computed on file level.

TABLE VI
LIST OF CODE DEPENDENCY NETWORK METRICS.

Metric name	Description
Ego-network metrics (computed for incoming, outgoing, and undirected dependencies; descriptions adapted from Z&N):	
Size	# nodes connected to the ego network
Ties	# directed ties corresponds to the number of edges
Pairs	# ordered pairs is the maximal number of directed ties
Density	% of possible ties that are actually present
WeakComp	# weak components in neighborhood
nWeakComp	# weak components normalized by size
TwoStepReach	% nodes that are two steps away
Brokerage	# pairs not directly connected. The higher this number, the more paths go through ego
nBrokerage	Brokerage normalized by the number of pairs
EgoBetween	% shortest paths between neighbors through ego
nEgoBetween	Betweenness normalized by the size of the ego network
Structural metrics (descriptions adapted from Z&N):	
EffSize	# entities that are connected to an entity minus the average number of ties between these entities
Efficiency	Normalizes the effective size of a network to the total size of the network
Constraint	Measures how strongly an entity is constrained by its neighbors
Hierarchy	Measures how the constraint measure is distributed across neighbors. When most of the constraint comes from a single neighbor, the value for hierarchy is higher
Centrality metrics (computed each for incoming, outgoing, and undirected dependencies; descriptions adapted from Z&N):	
Degree	# dependencies for an entity
nDegree	# dependencies normalized by number of entities
Closeness	Total length of the shortest paths from an entity (or to an entity) to all other entities
Reachability	# entities that can be reached from a entity (or which can reach an entity)
alpha.centrality†	Generalization of eigenvector centrality [5]
Information	Harmonic mean of the length of paths ending in entity
Betweenness	Measure for a entity in how many shortest paths between other entities it occurs
nBetweenness	Betweenness normalized by the number of entities

† Metrics not used by Z&N.

tential bug report references in commit messages (e.g. `[bug|issue|fixed]:?\s*#?\s?(\d+)`).

3) The pairs received from step 2) are passed through a set of filters checking
   a) that the bug report is marked as resolved.
   b) that the change set was applied after the bug report was opened.
   c) that the bug report was marked as resolved not later than two weeks after the change set was applied.

To determine a set of ground truth identifying the real purpose of change sets we use a data set published by Herzig et al. [13] containing a manually classified issue report type for more than 7,400 issue report. Instead of using the original issue report type to identify bug reports, we used the manual classified issue report type as published by Herzig et al. [13].

B. Complexity Metrics

Complexity metrics were used to train and evaluate the classification benchmark model based on code complexity churn metrics as well as the code complexity defect prediction benchmark model. We computed code complexity metrics for all source files of each projects trunk version using a commercial tool called JHAWK [1]. Using JHAWK we computed classical object-oriented code complexity metrics listed in Table V.

C. Network Metrics

Code dependency network metrics as proposed by Z&N express the information flow between code entities modeled by code dependency graphs. The set of network metrics we used slightly differs from the original metrics set used by Z&N. We computed the network metrics using the *R statistical software* [25] and the *igraph* [9] package. We could not re-use two of the 25 original network metrics: *ReachEfficiency* and *Eigenvector*. While we simply excluded *ReachEfficiency* from our network metric set, we substituted the *Eigenvector* metric by *alpha.centrality*—a metric that can be "considered as a generalization of eigenvector centrality to directed graphs" [5]. Table VI lists all code dependency network metrics used in this work. Metrics carry the same metric name than the corresponding metric as described by Z&N.

D. Genealogy Metrics

We discussed the set of genealogy metrics in Section III. To compute these metrics, we constructed change genealogy

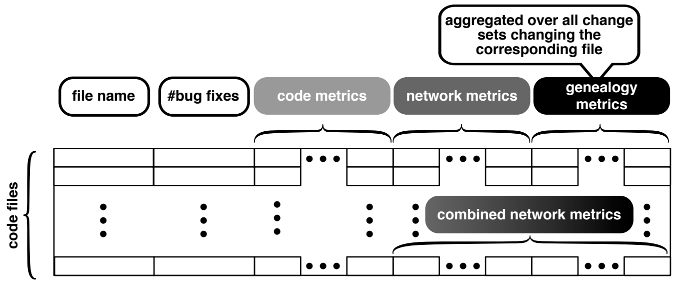


Fig. 5. Data collection used to predict defects for source files.

graphs modeling dependencies between change sets over the entire project history. But change genealogies model dependencies between individual changes. Thus, change genealogy network metrics express dependencies between change sets and not between source files, which we need in order to build classification models for source files. To transform change genealogy metrics into source file metrics, for each source file of a project, we aggregate all change genealogy metric values over all change sets that modified the corresponding file. We used three different aggregation functions: mean, max, and sum. The intuition behind aggregating change genealogy metrics to source file level is that we suspect central change sets as more crucial with respect to the development process. Consequently, we suspect code artifacts being changed by such crucial change sets to be more defect prone than others.

The resulting data collection is illustrated in Figure 5. For each source file of a project—identified by its full qualified file name—we know the distinct number of bugs that were fixed using change sets touching the corresponding file and three sets of metric values: code complexity metrics, network metrics as described and discussed by Z&N, and aggregated change genealogy metrics.

V. Experimental Setup

Our goal is to test multiple classification models based on different metrics set to classify defective source files for our four open-source software projects under investigation and to compare these classification models against each other. We consider source files as defective if at least one change set fixing a bug report modified the source file.

For each prediction model to be built, we require a data collection containing explanatory variables (metric values per source file) and the dependent variable identifying the number of fixed bugs in the corresponding source files (see Figure 5). For each project, we train and test four different sets of classification model based on: one model based on code complexity metrics (*CM*), one model based on code dependency network metrics (*NM*), one model based on genealogy network metrics (*GM*), and one model based on a combined set code dependency and change genealogy network metrics (*Comb.*).

Each prediction model requires training and testing sets. To achieve comparable prediction models we split the original data set as shown in Figure 5 twice:

First split: stratified random sampling We split the overall data set into 2/3 training and 1/3 testing sets using stratified sampling—the ratio of source files being marked as defective (bug count larger than zero) in the original dataset is preserved in both training and testing subsets. Stratified sampling makes training and testing sets more representative since it reduces sampling errors.

Second split: by metric columns Next, we split training and testing sets by metric columns. For each training and testing set received from the first splitting step, we produce four training and four testing sets each containing the metric columns required to train and test one of the four different prediction models: code complexity metrics (*CM*), code dependency network metrics (*NM*), genealogy metrics (*GM*), combined set code dependency and change genealogy network metrics (*Comb.*).

Splitting the metric columns apart after splitting training and testing sets apart ensures that each metric set is validated on the exact same training and testing sets and thus ensures a fair comparison between the individual metric sets. Further, we repeatedly sample the original data sets 100 times (cross-fold) using the above splitting scheme in order to generate 100 independent training and testing sets per metric set. Thus, in total we generate 400 different training and testing sets per subject project.

We conducted our experiments using the R statistical software [25] and more precisely Max Kuhn's R package caret [16]. The caret package provides helpful functions and wrapper methods to train, tune, and test various classification models. Table VII lists the models we used for our classification results. Each model is optimized by the caret package by training models using different tune parameters (please see caret manual[1] for more details). "The performance of held-out samples is calculated and the mean and standard deviations is summarized for each combination. The parameter combination with the optimal resampling statistic is chosen as the final model and the entire training set is used to fit a final model" [16]. The level of performed optimization can be set using the *tuneLength* parameter. We set this number to five. We further apply the following optimization steps before training and testing the individual classification models:

Remove constant metric columns. Metrics that show near zero variance and thus can be considered as constant. The caret package provides the *nearZeroVar* function that identifies those explanatory variables that show no significant variance. Such metrics will not contribute to the final prediction or classification model.

Remove highly inter-correlated metric columns. If two explanatory variables are correlated with other variables they add no new information dimension. Using the caret function *findCorrelation* we remove any metric column that showed a correlation higher than 0.9 with any other metric column.

[1]http://cran.r-project.org/web/packages/caret/caret.pdf

TABLE VII
LIST OF MODELS USED FOR PREDICTION EXPERIMENTS.

Model*	**Description**
k-nearest neighbor (*knn*)	*This model finds k training instances closest in Euclidean distance to the given test instance and predicts the class that is the majority amongst these training instances.*
Logistic regression (*multinom*)	*This is a generalized linear model using a logic function and hence suited for binomial regression, i.e. where the outcome class is dichotomous.*
Recursive partitioning (*rpart*)	*A variant of decision trees, this model can be represented as a binomial tree and popularly used for classification tasks.*
Support vector machines (*svmRadial*)	*This model classifies data by determining a separator that distinguishes the data with the largest margin. We used the radial kernel for our experiments.*
Tree Bagging (*treebag*)	*Another variant of decision trees, this model uses bootstrapping to stabilize the decision trees.*
Random forest (*randomForest*)	*An ensemble of decision tree classifiers. Random forests grow multiple decision trees each "voting" for the class on an instance to be classified.*

*For better understanding, we advise the reader to refer to specialized machine learning texts such as by Wittig and Frank [29].

Rescaling and centering data. To minimize the effects of large metric values on the classification models, we rescaled all metric columns (training and testing data) into the value range [0,1].

As evaluation measure we report *precision*, *recall*, and *F-measure*. Each of these measures is a value between zero and one. A precision of one indicated that the classification model did not produce any false positives; that is classified non bug fixes as bug fixes. A recall of one would imply that the classification model did not produce any false negatives—classified a bug fix not as such. The F-measure represents the harmonic mean of precision and recall.

To measure whether the reported classification performance differences are statistical significant, we performed a Kruskal-Wallis test—a non-parametric statistical test to statistically compare the results. The statistical tests showed that all reported differences in classification performances are statistically significant ($p < 0.05$).

VI. PREDICTION RESULTS

Classification results are presented in Figure 6. Panels across the x-axis in the figure represent the subject projects. The four prediction models used were run on 100 stratified random samples on four metric sets: complexity metrics (*CM*), code dependency network metrics (*NM*), change genealogy metrics (*CGM*), and a combined set of combined code dependency and change dependency network metrics (*Comb.*). For each run we computed precision, recall an F-measure values.

TABLE VIII
MEDIAN PREDICTION PERFORMANCE MEASURES AS SHOWN IN FIGURE 6.

Metric set	**HTTPCLIENT**	**JACKRABBIT**	**LUCENE**	**RHINO**
		Precision		
CM	0.47	0.61	0.59	0.55
NM	0.62	0.64	0.66	0.70
GM	0.79	0.71	0.71	0.74
Comb.	0.82	0.67	0.60	0.77
		Recall		
CM	0.38	0.37	0.21	0.36
NM	0.70	0.50	0.30	0.64
GM	0.69	0.48	0.41	0.62
Comb.	0.79	0.79	0.41	0.76
		F-Measure		
CM	0.42	0.45	0.30	0.42
NM	0.67	0.56	0.42	0.65
GM	0.73	0.57	0.52	0.67
Comb.	0.79	0.72	0.49	0.77

The boxplots in the figure reflect the distribution of the corresponding performance measure. For each distribution the best performing model (see Section V) is stated under the corresponding. The boxplot black line in the middle of each boxplot indicates the median value of the corresponding distribution. Larger median values indicate better performance on the metrics set for the project based on the respective evaluation measure. Note that the red colored horizontal lines connecting the medians across the boxplots do not have any statistical meaning—they have been added to aid visual comparison of the performance of the metrics set. An upward sloping horizontal line between two boxplots indicates that the metrics set on the right performs better than the one of the left and vice versa.

A. General Observations

In general, the variance of classification results between individual models seems to be high, especially for models trained and tested on the combined set of network and genealogy metrics. Although, in most cases, treebag models provided the best classification performances, it seems to be important to not only compare individual metric sets but also individual classification models as this may heavily impact the performance results. For the rest of this section, we will use the median performance (black line in box plots and shown inTable VIII) across all models as comparison basis, unless stated otherwise.

B. Comparing Complexity and Network Metrics

As expected, network metrics (*NM*) outperform code complexity metrics (*CM*). Across all projects and network metric models show better precision and recall values. Except for LUCENE, network metric models even show a lower variance border that lies above the upper variance border for complexity metrics, although network metric models seem to show larger variances and thus seem to be more dependent upon the used algorithm to train the individual model. Overall, the result confirms the initial findings by Zimmermann and Nagappan [30].

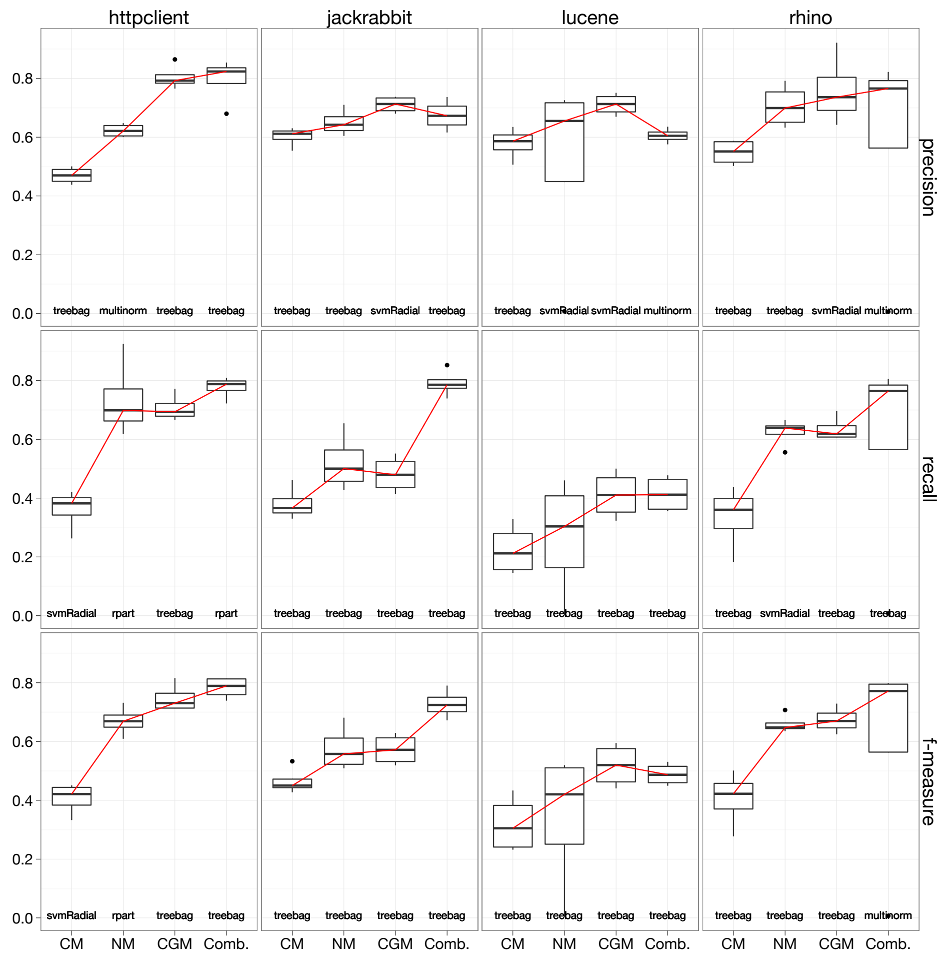


Fig. 6. Results from the repeated holdout prediction setup.

C. Comparing Network and Genealogy Metrics

When comparing the average classification performances between code dependency network (*NM*) and genealogy (*GM*) it is evident that genealogy metrics outperform network metrics with respect to precision values. For all projects, genealogy metric models report higher precision values. The median precision of network metric models across all projects lies at 0.64 whereas the median precision of models based on change genealogies lies at 0.74. That is an increase of 10% for the median precision. Thus, models based on change genealogy metrics tend to report 10% less false positives when compared to network metric models.

At the same time, the median recall values for change genealogy metrics (0.58) lie slightly below the recall values of models trained on network metrics (0.59). Thus, using genealogy network metrics to classify defective source files reduces the number of false positives by about 10% while reporting nearly exactly the same amount of false negatives. That also explains the overall better F-measures for genealogy metric models across all projects.

D. The Combined Set of Network Metrics

Combining code dependency and change dependency network metrics to classify code changes led to a mixed result. While precision values for HTTPCLIENT and RHINO were the best precision values when compared to all other metric sets, the precision dropped for the remaining two projects to code dependency metric level. But interestingly, models trained using the combined metric sets show better recall values for all four projects. Except for LUCENE the recall values are considerable increased: up to 29% while the median improvement lies at 10.5%. Comparing the F-measures across all projects and all metric sets, the combined network metric set showed the best results, except for LUCENE where the combined network metrics showed slightly lower F-measures than the genealogy network Metrics.

Overall, project size seems to have no impact on prediction accuracy.

☛ *Models based on change genealogy metrics report less false positives (higher precision).*

☛ *Combining code dependency and change genealogy metrics increases recall but decreases precision.*

TABLE IX
TOP 10 MOST INFLUENTIAL METRICS PER PROJECT IN DECREASING ORDER.

HTTPCLIENT	JACKRABBIT	LUCENE	RHINO
NumParentAuthors (*GM*)	OutDegree (*NM*)	closeness (*NM*)	OutEfficiency (*GM*)
NumCallParents (*GM*)	OutEfficiency (*GM*)	twoStepReach (*NM*)	Efficiency (*GM*)
NumParents (*GM*)	AvgChildrenOut (*GM*)	effSize (*NM*)	ChildrenParents (*GM*)
OutEffSize (*GM*)	NumCallParents (*GM*)	efficiency (*NM*)	AvgParentAge (*GM*)
NumInbreedChildren (*GM*)	AvgInbreedChildren (*GM*)	tiesOut (*NM*)	MinResponseTime (*GM*)
EffSize (*GM*)	AvgSiblingChildren (*GM*)	NumDefDependents (*GM*)	NumParents (*GM*)
NumDependants (*GM*)	NumCallParents (*GM*)	NumChildrenParents (*GM*)	MinParentAge (*GM*)
NumCallDependents (*GM*)	InEffeciency (*GM*)	NumCallDependents (*GM*)	outDegree (*NM*)
NumSiblingChildren (*GM*)	NumParents (*GM*)	NumSiblingChildren (*GM*)	MaxResponseTime (*GM*)
MaxParentAge (*GM*)	Efficiency (*GM*)	MaxParentAge (*GM*)	AvgInbreedChildren (*GM*)

E. *Influential Metrics*

To identify the overall most influential metrics, we used the `filterVarImp` function provided by the *caret* [16] package. This function computes a ROC curve by first applying a series of cutoffs for each metric before computing the sensitivity and specificity for each cutoff point. The importance of the metric is then determined by measuring the area under the ROC curve. We use a combined metrics set containing all metrics (*CM*, *NM*, and *GM*) to compute variable importance of these metrics. The top-10 most influential metrics are shown in Table IX in decreasing order.

The top-10 most influential for HTTPCLIENT and RHINO are all change genealogy metrics. The top most influential metrics for JACKRABBIT is a code dependency metric measuring the number of outgoing edges. The top-5 most influential metrics for LUCENE are all code dependency metrics and explains the exceptional trends in the overall classification performance comparison for this project.

We observed three different patterns with respect to presence and ranking of network and change genealogy metrics:

Efficiency network metrics are in the top-10 most influential metrics for every of the four projects. For LUCENE the code dependency efficiency metric contributed (*effsize* and *efficiency*) while for the other three projects the change genealogy efficiency metrics (*EffSize* and *Efficiency*) made the difference.

The number of genealogy parents seems to be crucial. The more change genealogy parents the higher the likelihood of bugs. For all projects, the top-10 set of most influential metrics contains at least one metric expressing the number of change parents.

The time span between changes seems to be crucial as well. For three out of four projects the timespan between a change and its parents is important. Combined with the previous finding, it seems that source files that got applied many change set combining multiple older functions are more likely to be defect prone than others.

☛ *Network efficiency seems to be crucial for prediction models relying on network metrics.*

☛ *Applying many changes combining multiple older functions seems to raise defect likelihood.*

VII. THREATS TO VALIDITY

Empirical studies like this one have threats to validity.

Change Genealogies. First and most noteworthy, change genealogies model only dependencies between added and deleted method definitions and method calls. Disregarding change dependencies not modeled by change genealogies might have an impact on change dependency metrics. More precise change dependency models might lead to different change genealogy metric values and thus might change the predictive accuracy of the corresponding classification and prediction models.

Number of bugs. Computing the number of bugs per file is based on heuristics. While we applied the same technique as other contemporary studies do, there is a chance the count of bugs for some files may be an approximation.

Issue reports. We reused a manual classified set of issue reports to determine the purpose of individual change sets. The threats to validity of the original manual classification study [13] also apply to this study.

Non-atomic change sets. Individual change sets may refer to only one issue report but still apply code changes serving multiple development purposes (e.g. refactorings or code cleanups). Such non-atomic changes introduce noise into the change genealogy metric sets and thus might bias the corresponding classification models.

Study subject. The projects investigated might not be representative, threatening the external validity of our findings. Using different subject projects to compare change genealogy, code dependency, and complexity metrics might yield different results.

VIII. CONCLUSION

When it comes to learning from software histories, looking for indirect effects of events can make significant differences. As presented in this paper, software genealogies provide wider and deeper insights on the long-term impact of changes, resulting in better change classification report significantly less false positives when compared to code network metric and code complexity models.

In our future work, we will continue to take a broad, long-term view on software evolution. With respect to change genealogies, our work will focus on the following topics:

Graph patterns and metrics. Besides temporal change rules [15], we can also search for specific *patterns* in the genealogy graph, such as identifying changes that trigger the most future changes, changes with the highest long-term impact on quality or maintainability, or bursts of changes in a short time period. Such patterns may turn out to be excellent change predictors [21].

More features and rules. Genealogies may be annotated with additional features, such as authors or metrics, in order to allow more predictive patterns ("Whenever Bob increases the cyclomatic complexity above 0.75, the module requires refactoring")

Prediction rationales. Predictions based on change genealogies build on a wide number of factors that are harder to comprehend than isolated metrics. We are working on *providing rationales* based on past history to precisely explain why a module is predicted to fail, and to suggest short- and long-term consequences.

To learn more about our work, visit our Web site:

http://softevo.org/change_genealogies/

Acknowledgments. We thank the reviewers for their constructive comments.

REFERENCES

[1] Jhawk 5 (release 5 version 1.0.1). Available at: http://www.virtualmachinery.com/jhawkprod.htm.

[2] O. Alam, B. Adams, and A. E. Hassan. A study of the time dependence of code changes. In *Proceedings of the 2009 16th Working Conference on Reverse Engineering*, WCRE '09, pages 21–30. IEEE Computer Society, 2009.

[3] V. R. Basili, L. C. Briand, and W. L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Trans. Softw. Eng.*, 22(10):751–761, Oct. 1996.

[4] C. Bird, N. Nagappan, H. Gall, B. Murphy, and P. Devanbu. Putting it all together: Using socio-technical networks to predict failures. In *Proceedings of the 2009 20th International Symposium on Software Reliability Engineering*, ISSRE '09, pages 109–119. IEEE Computer Society, 2009.

[5] P. Bonacich. Power and centrality: A family of measures. *American journal of sociology*, 1987.

[6] I. I. Brudaru and A. Zeller. What is the long-term impact of changes? In *Proceedings of the 2008 international workshop on Recommendation systems for software engineering*, RSSE '08, pages 30–32. ACM, 2008.

[7] R. S. Burt. *Structural holes: The social structure of competition*. Harvard University Press, Cambridge, MA, 1992.

[8] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, June 1994.

[9] G. Csardi and T. Nepusz. The igraph software package for complex network research. *InterJournal*, Complex Systems:1695, 2006.

[10] D. M. German, A. E. Hassan, and G. Robles. Change impact graphs: Determining the impact of prior code changes. *Inf. Softw. Technol.*, 51(10):1394–1408, Oct. 2009.

[11] A. E. Hassan. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 78–88. IEEE Computer Society, 2009.

[12] K. Herzig. *Mining and Untangling Change Genealogies*. PhD thesis, Universität des Saarlandes, dez 2012.

[13] K. Herzig, S. Just, and A. Zeller. It's not a bug, it's a feature: How misclassification impacts bug prediction. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 392–401, Piscataway, NJ, USA, 2013. IEEE Press.

[14] K. Herzig and A. Zeller. Mining Cause-Effect-Chains from Version Histories. In *Proceedings of the 2011 IEEE 22nd International Symposium on Software Reliability Engineering*, ISSRE '11, pages 60–69. IEEE Computer Society, 2011.

[15] K. S. Herzig. Capturing the long-term impact of changes. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, pages 393–396. ACM, 2010.

[16] M. Kuhn. *caret: Classification and Regression Training*, 2011. R package version 4.76.

[17] T. J. McCabe. A complexity measure. *IEEE Trans. Software Eng.*, 2(4):308–320, 1976.

[18] R. Moser, W. Pedrycz, and G. Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 181–190. ACM, 2008.

[19] N. Nagappan and T. Ball. Using software dependencies and churn metrics to predict field failures: An empirical case study. In *Proceedings of the First International Symposium on Empirical Software Engineering and Measurement*, ESEM '07, pages 364–373. IEEE Computer Society, 2007.

[20] N. Nagappan, B. Murphy, and V. Basili. The influence of organizational structure on software quality: an empirical case study. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 521–530. ACM, 2008.

[21] N. Nagappan, A. Zeller, T. Zimmermann, K. Herzig, and B. Murphy. Change bursts as defect predictors. In *Proceedings of the 2010 IEEE 21st International Symposium on Software Reliability Engineering*, ISSRE '10, pages 309–318. IEEE Computer Society, 2010.

[22] T. J. Ostrand, E. J. Weyuker, and R. M. Bell. Where the bugs are. In *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '04, pages 86–96. ACM, 2004.

[23] M. Pinzger, N. Nagappan, and B. Murphy. Can developer-module networks predict failures? In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, SIGSOFT '08/FSE-16, pages 2–12. ACM, 2008.

[24] R. Premraj and K. Herzig. Network versus code metrics to predict defects: A replication study. In *Proceedings of the 2011 International Symposium on Empirical Software Engineering and Measurement*, ESEM '11, pages 215–224. IEEE Computer Society, 2011.

[25] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, 2010.

[26] A. Schröter, T. Zimmermann, and A. Zeller. Predicting component failures at design time. In *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, ISESE '06, pages 18–27. ACM, 2006.

[27] Y. Shin, R. Bell, T. Ostrand, and E. Weyuker. Does calling structure information improve the accuracy of fault prediction? In *Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*, MSR '09, pages 61–70. IEEE Computer Society, 2009.

[28] A. Tosun, B. Turhan, and A. Bener. Validation of network measures as indicators of defective modules in software systems. In *Proceedings of the 5th International Conference on Predictor Models in Software Engineering*, PROMISE '09, pages 5:1–5:9. ACM, 2009.

[29] I. H. Witten and E. Frank. Data mining: practical machine learning tools and techniques with java implementations. *SIGMOD Rec.*, 31(1):76–77, Mar. 2002.

[30] T. Zimmermann and N. Nagappan. Predicting defects using network analysis on dependency graphs. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 531–540. ACM, 2008.

[31] T. Zimmermann, R. Premraj, and A. Zeller. Predicting defects for Eclipse. In *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, PROMISE '07, pages 9–. IEEE Computer Society, 2007.