Classifying Code Changes and Predicting Defects Using Change Genealogies

Kim Herzig
Saarland University
herzig@cs.uni-saarland.de

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

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

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

I. Introduction

Identifying bug fixes and using them to estimate or even predict software quality is a frequent task when mining version archives. The number of applied bug fixes serves as code quality metric identifying defect-prone and non-defect-prone code artifacts. But when is a set of applied code changes, we call it *change set*, considered a bug fix and which metrics should be used to building high quality defect prediction models? Most commonly, bug fixes are identified by analyzing commit messages—short, mostly unstructured pieces of plain text. Commit message containing keywords such as “fix” or “issue” followed by a bug report identifier, are considered to fix the corresponding bug report. Similar, most defect prediction models use metrics describing the structure, complexity, or dependencies of source code artifacts. Complex or central code is considered to be more defect-prone.

But commit messages and code metrics describe the state of software artifacts and code changes at a particular point in time, disregarding their genealogies that describe how the current state description came to be. There are approaches measuring historic properties of code artifacts [1]–[5] and using code dependency graphs [6], [7] but non of these approaches tracks the structural dependency paths of code changes to measure the centrality and impact of change sets, although change sets are those development events that make the source code look as it does. Herzig et al. [8] used so called change genealogy graphs to model structural dependencies between change sets. The authors used these change genealogy graphs to measure and analyze the impact of change sets on other, later applied change sets.

In this paper, we make use of *change genealogy graphs* to define a set of change genealogy network metrics describing the structural dependencies of change sets. We further investigate whether *change genealogy metrics* can be used to identify bug fixing change sets (without using commit messages and bug databases) and whether change genealogy metrics are expressive enough to build effective defect prediction models classifying source files to be defect-prone or not.

Regarding the identification of bug fixing change sets, our assumption is that change sets applying bug fixes show significant dependency differences when compared to change sets applying new feature implementations. We suspect that implementing and adding a new feature implies adding new method definitions that impact a large set of later applied code changes, which add code fragments adding method calls to these newly defined methods. In contrast, we suspect bug fixes to be relatively small rarely defining new methods but modifying existing features and thus to have a small impact on later applied code changes. The impact of bug fixes is to modify the runtime behavior of the software system rather than causing future change sets to use different functionality.

Similar, we suspect more central change sets—depending on a large set of earlier change sets and causing many later applied change sets to be dependent on itself—to be crucial to the software development process. Consequently, we suspect code artifacts that got many crucial and central code changes applied to be more defect prone than others.

More specifically, we seek to answer the following research questions in our study:

RQ1 How do bug fix classification models based on change genealogy metrics compare to classification models based on code complexity metrics (Section V)?

RQ2 How do defect prediction models compare with defect prediction models based on code complexity or code dependency network metrics (Section VI)?

We tested the classification and prediction abilities of our approaches on four open source projects. The results show that change genealogy metrics can be used to separate bug fixing from feature implementing change sets with an average precision of 72% and an average recall of 89%. Our results also show that defect prediction models based on change genealogy metrics can predict defect-prone source files with precision and recall values of up to 80%. On average the precision for change genealogy models lies at 69% and the average recall at 81%. Compared to prediction models based on code dependency network 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 [9]. A change genealogy is a directed 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 location of a failure. Alam et al. [11] 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 [12] used the original concept of change genealogies as defined by Brudaru and Zeller [9] to implement change genealogies modeling dependencies between added, modified, and deleted method definitions and method calls. Later, Herzig and Zeller [8] used this 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] and used by Herzig and Zeller [8].

Change Genealogies in a Nutshell

Our *change genealogy* framework models dependencies between individual change sets based on method definitions and method calls added, deleted by every change set. A Code change CS_N depends on an earlier change set CS_M if

- CS_N deletes a method definition added in CS_M,
- CS_N adds a method definition previously deleted in CS_M,
- CS_N adds a statement calling a method definition 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 sets 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)`. Note that there exists no change operation for the method definition `public C()...` nor for the class itself. Method calls and definitions are identified using their full qualified name and absolute position within the source code. Two change sets depend on each other if any of their applied change operations depend on each other.

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 genealogies 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

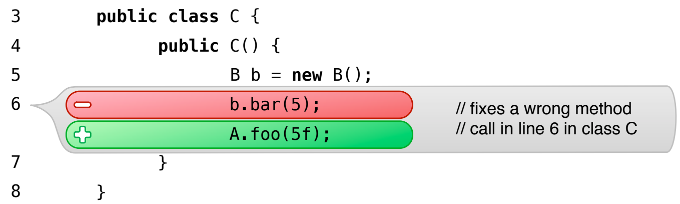

Fig. 1. Diff output corresponding to the table cell of column $CS4$ and row $File3$ shown in Figure 2. It also corresponds to the genealogy vertex $CS4$ shown in Figure 3.

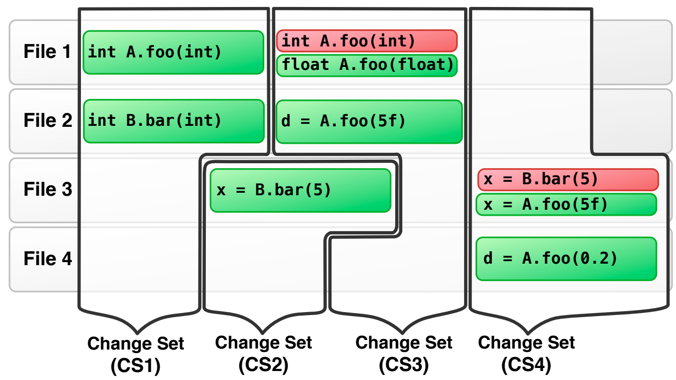

Fig. 2. We characterize change sets by method calls and definitions added or deleted. Changes depend on each other based on the affected methods.

to be powerful to express dependencies between source code artifacts such as methods and to predict software defects on file and package level [6], [7], [13], [14]. 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. *Change Classification*

Classifying change sets is common in mining version archives. There exist different approaches classifying change sets with respect to various aspects.

Classifying whether change sets are bug fixes or not is one of the earliest topics in mining version archives. Most approaches are based on commit message analysis. One of

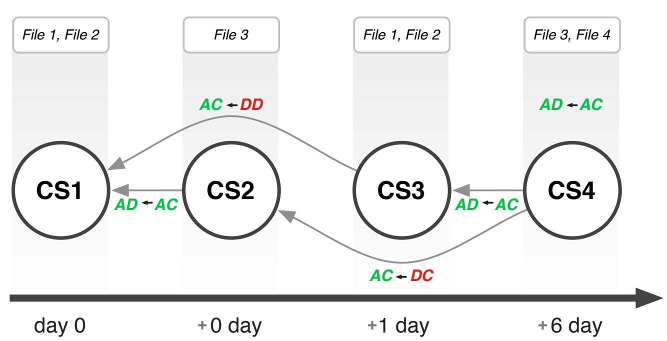

Fig. 3. Sample change genealogy derived from the change operations shown in Figure 2. $CS_i \rightarrow CS_j$ indicates that the change CS_j depends on CS_i.

the first approaches was presented by Čubranić and Gail [15]. The approach scans commit messages provided by developers for keywords indicating links to the issue ticket system (e.g. "Fixes bug id 7478"). These links will then be further filtered based on their activity, authorship and report date. Many approaches use a slightly modified version of this classification technique [16]–[19]. Lately, Murgia et al. [20] presented a technique using natural language processing to group commit messages sharing the same text features.

There exist a variety of approaches classifying change sets according to their impact on program execution [21], software architecture [22]–[25], and program execution [21]. Closely related is the approach of Fluri et al. who developed a framework capable of differentiating "between several types of changes on the method or class level" [26]. With their framework, the authors are able to assess the impact of a change set on other source code entities and whether the applied change set modifies the functionality of the software system or not. Although, our approach uses a similar abstraction layer, our aim is it to classify code changes with respect to their purpose: is a change set a bug fix or a feature implementation.

Kim et al. [19] classified change sets with respect to the likelihood the applied change set introduced a new software defect. Although this approach limits the search space for defect prediction models drastically, it cannot be used to identify bug fixes. Within their approach, the authors themselves used a commit message based approach to identify bug fixes.

Lately, Kawrykow and Robillard [27] identified so called non-essential changes—changes that are of "cosmetic nature, generally behavior-preserving, and unlikely to yield further insights into the roles of or relationships between the program entities they modify" [27].

D. Predicting Defects

Defect prediction models aim to predict the number and sometimes the location of defects to be fixed in near future. Such systems can be used to allocate quality assurance resource. 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. [28] using object-oriented metrics. Many studies investigated a large variety of different code metric types for defect prediction purposes. Ostrand et al. [29] used code metrics and prior faults to predict the number of faults for large industrial systems. Zimmermann et al. [17] demonstrated that higher code complexity leads to more defects. Besides code related metrics, there exist studies showing that change-related metrics [4], developer related metrics [30], organizational metrics [31] and process metrics [32] can be used to predict defect prone code artifacts.

The usage of code dependency information to build defect prediction models in now new either. Schöter et al. [33] used import statements to predict the number of defects for source files at design time. Shin et al. [34] and Nagapan and Ball [35] provided evidence that defect prediction models can benefit when adding calling structure metrics.

Zimmermann and Nagappan [6] 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. [7] 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 [6], [7] we can use change set dependency graph 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 measure dependencies between 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. Bug fixes might depend mainly of change sets that have the same author. The last six metrics in Table I express temporal dependencies between change sets based on their commit timestamp.

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 metric traversal depth to a maximal depth of five.

Metrics counting the number of global descendants or ascendents express the indirect impact of change sets on other change sets and how long this impact propagates though history. The set of inbreed metrics express dependencies between a change set and its children in terms of common ascendents

TABLE I
EGO NETWORK METRICS CAPTURING DIRECT NEIGBOUR 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 ascendents 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.

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 introduces by Burt [36] and measures the influence of actors in balanced social networks. In networks where each actor is connected to all other actors is well balanced. As soon as dependencies between individual actors are missing ("structural holes") some actors are in advanced positions.

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.

D. Change Metrics

The last set of metrics shown in Table IV measure the amount of cade changes applied by the corresponding change set and its neighbors in the ego network. In our case, we counted the different number of added and deleted method definitions and method calls. The intuition behind these metrics is that bug-fixes should in many cases be considerable smaller than other developer tasks such as feature implementations or code clean-ups [37].

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.
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 AS DEFINED BY BURT [36].

Metric name	**Description**
EffSize	The number of vertices that connected to this vertex minus the effective size 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 sizeof the number of vertices of the ego-network
InEfficiency	norms InEffSize by the sizeof the number of vertices of the ego-network
OutEfficiency	norms OutEffSize by the sizeof the number of vertices of the ego-network

IV. DATA COLLECTION

The goals of our approach are to classify bug fixing change sets independent from commit messages and bug databases and to predict defect prone source files, both using change genealogy network metrics. To reason about the precision of our classification and prediction models, we compare our models to state of the art benchmark models.

Mockus and Votta [37] (referred to as M&V for sake of

TABLE IV
CHANGE SIZE METRICS DESCRIBE A VERTEX USING THE NUMBER OF CHANGE OPERATIONS APPLIED BY THE CORRESPONDING CODE CHANGE.

Metric name	Description
ChangeSize	The number of change operations corresponding to the vertex.
NumAddOps	The number of adding change operations corresponding to the vertex.
NumDelOps	The number of deleting change operations corresponding to the vertex.
NumAddMethDefOps	The number of change operations adding method definitions corresponding to the vertex.
NumDelMethDefOps	The number of change operations deleting method definitions corresponding to the vertex.
NumAddCallOps	The number of change operations adding method calls corresponding to the vertex.
NumDelCallOps	The number of change operations deleting method calls corresponding to the vertex.
AvgDepChangeSize	The mean number of change operations changed by direct children.
MaxDepChangeSize	The maximal number of change operations changed by one of the direct children.
SumDepChangeSize	The total number of change operations changed by direct children.
AvgParentChangeSize	The mean number of change operations changed by direct parents.
MaxParentChangeSize	The maximal number of change operations changed by one of the direct parents.
SumParentChangeSize	The total number of change operations changed by direct parents.

TABLE V
PROJECTS USED FOR EXPERIMENTS.

	HTTPCLIENT	JACKRABBIT†	LUCENE	RHINO
History length	6.5 years	8 years	2 years	13 years
Lines of Code	57,143	65,764	362,128	56,084
# Source files	570	687	2,542	217
# Code changes	1,622	7,465	5,771	2,883
# Mapped `BUG` reports	92	756	255	194
# Mapped `RFE` reports	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 sub-project JACKRABBIT `content repository` (*JCR*).

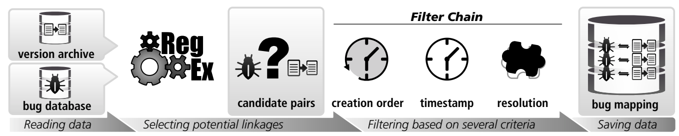

Fig. 4. Process of linking change sets to bug reports.

brevity) used code complexity metrics to identify bug fixing change sets. We will use a code complexity metric based change purpose classification models as benchmark model to compare against our change genealogy network metric models. Section IV-B contains details on the used complexity metrics.

We compare change genealogy defect prediction models against two benchmark model: models based on code complexity [17] and models based on code dependency models [6](referred to as Z&N for sake of brevity). Section IV-B and Section IV-C contain details on the used complexity and code dependency network metrics.

We evaluate our classification and prediction models on four open-source 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 V. 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 modifying code documentation).

A. Bugs

For both approaches, change classification and defect prediction, we need to know whether a change set applies a bug fix and the total number of applied bug fixes per source file. Once we identified bug fixing change sets and the corresponding bug report id, 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 the approach by Zimmermann et al. [17] (see also Figure 4):

1) Bug reports and change sets are read from the corresponding bug tracking system and version archive.
2) In a preprocess step we select potential candidates using regular expressions such as `[bug|issue|fixed]:?\s*#?\s?(\d+)` to search for potential bug report references in commit messages.
3) The pairs received from step 2) then pass 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. [38] containing a manual classified issue report type for each individual files 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. [38].

B. Complexity Metrics

We computed code complexity metrics for all source files of each projects trunk version using a commercial tool called JHAWK [39]. JHAWK computes classical object-oriented code complexity metrics for JAVA projects. Using JHAWK we computed the code complexity metrics listed in Table VI.

TABLE VI
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 [40].
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 [41].
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 *effect 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.

C. Network Metrics

Code dependency network metrics as proposed by Z&N express the information flow between code entities modeled by code dependency graph. The set of network metrics used in this work slightly differs from the original metrics set used by Z&N. We computed the used network metrics using the *R statistical software* [42] and the *igraph* [43] package. Using *igraph* we could not re-implement two of the 25 original network metrics: *ReachEfficiency* and *Eigenvector*. While we simply excluded *ReachEfficiency* from our network metric set, we substituted the *Eigenvector* by *alpha.centrality*—a metric that can be "considered as a generalization of eigenvector centrality to directed graphs" [44]. Table VII 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 graphs modeling dependencies between change sets over the entire project history. The metrics were computed using our self written MOZKITO [45] mining framework.

V. CLASSIFYING CODE CHANGES (RQ1)

In this first series of experiments we seek an answer to *RQ1*: can we use change genealogy metrics to identify bug fixing change sets and how to such code change classification models compare to classification models based on code complexity models?

TABLE VII
LIST OF CODE DEPENDENCY NETWORK METRICS.

Metric name	Description
Ego-network metrics (computed each 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	Betweeness 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 [44]
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.

Our goal is to build two sets of change set classification models for each subject project and to compare both sets of classification models against each other. For each classification model to be built, we need a data collection containing explanatory variables (metric values per change set) and the dependent variable classifying the corresponding change set as bug fixing or as feature adding (see Figure 5). Change genealogy metrics are already collected at change set level—each change set corresponds to exactly one change genealogy vertex. But code complexity metrics are collected on source file level. M&V used the difference in code complexity before and after the change set applied as metric. Following the idea of M&V, for each metric $\mathcal{M}$ we sum up the metric values over all source files at revision CS_{i-1} and subtract the same sum of metric values collected at revision CS_i. Doing this for every code complexity metrics, yields a set of code complexity metric values reflecting the amount of code complexity added or deleted by change set CS_i—we call this set *code complexity difference metrics*.

The columns containing the change genealogy metrics are

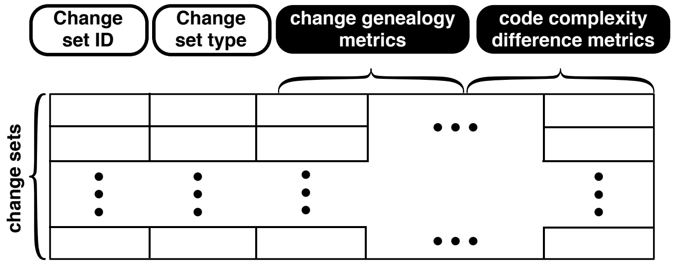

Fig. 5. Data collection used for change set classification purposes.

used to train the change genealogy classification model, the code complexity difference columns are used to train the benchmark model.

A. Experimental Setup (RQ1)

To train and test our classification models, we split our original data set as shown in Figure 5 into training and testing sub sets.

We train and test classification models for each subject project, once using change genealogy metrics and once using code complexity metric difference values using stratified sampling—the ratio of bug fixing change sets in the original data set is preserved in both training and testing data sets. This makes training and testing sets more representative by reducing sampling errors.

Next, we split the training and testing sets into sub sets each containing the columns change set type and change set identifier but one set containing change genealogy metrics only and one set containing code complexity difference metrics only. Splitting metric sets after creating testing and training sets, we create pairs of classification models using the same training and testing split but using different metrics data as feature vectors.

We repeatedly sample data sets 100 times in order to generate 100 independent training and testing sets. Each split is used to built one change genealogy and one code complexity model. In total, we test 200 independent prediction models for each project. Using such a repeated sampling reduces bias. A single sample may lead to a good or bad result by accident.

We conducted our experiments using the *R statistical software* [42] and more precisely Max Kuhn's R package *caret* [46]. This package provides helpful wrapper functions to several machine learning algorithms available in other packages. As machine learner we used a support vector machine with radial kernel. As evaluation measures, 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.

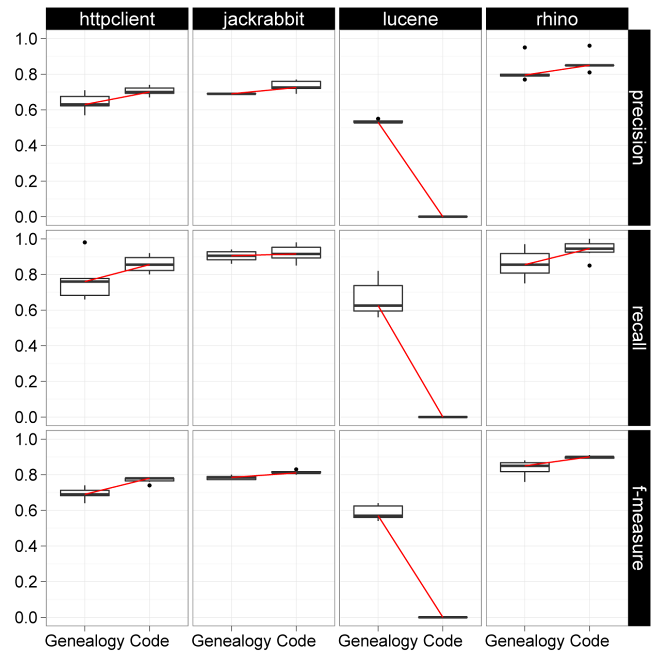

Fig. 6. Results from the classification experiment to separate bug fixing from feature implementing code changes.

B. Classification Quality

The results of the stratified repeated holdout setup are shown in Figure 6. Panels on the x-axis represent the subject projects. Each classification model ran on 100 stratified random samples on the two metric sets: change genealogy and complexity difference metrics.

The black line in the middle of each boxplot indicates the median value of the distribution. The red colored horizontal lines do not have any statistical meaning—they have been added to ease visual comparison. Additionally, we performed a non-parametric statistical test (Kruskal-Wallis) to statistically compare the results from the use of two pairs of metrics sets: change genealogy metrics vs. code complexity metric differences.

The results shown in Figure 6 show that the classification performances of both metric sets are close to each other, except for LUCENE. In all three cases code complexity difference metrics show statistically significant ($p < 0.05$) stronger classification results than change genealogy metrics, except for the recall values for JACKRABBIT. In summary, code complexity differences outperform change genealogy metrics on three out of four projects. For LUCENE we were no able to train a functional classification model using code complexity. Nearly all change sets applied to LUCENE modified code complexity only marginally. Thus, complexity metrics showed too little variance to allow classification model training. Over all projects, models based on change genealogy metrics show a median precision of 0.69 and a median recall of 0.81. Models based on complexity metrics showed a median precision of 0.72 and a median recall of 0.89.

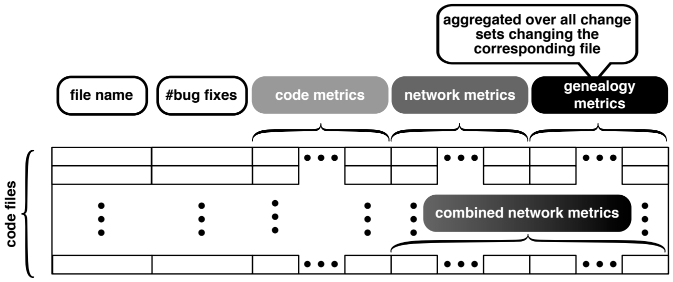

Fig. 7. Data collection used to classify bug fixing change sets and to predict defects for source files.

C. Influential Metrics

The R package *caret* [46] allows computing the importance of individual metrics using the `filterVarImp` function. The function computes a ROC curve by first applying a series of cutoffs for each metric and then computing the sensitivity and specificity for each cutoff point. The importance of the metric is then determined by computing the area under the ROC curve. We a combined metrics set to compute variable importance for change genealogy and code complexity metrics and considered the top-10 most influential metrics for each metrics set for examination.

The most influential change genealogy metrics are dedicated to code age, the number of change set parents, and network efficiency. Bug fixing change sets seem to change older code while feature implementations are based on newer code fragments. It also seems universal that feature implementing change sets have more structural dependency parents than bug fixing ones.

The most influential complexity difference metrics show that the higher the impact of a change set on cyclomatic complexity of the underlying source code, the higher the chance that the change set is implementing a new feature. Thus, bug fixing change sets show smaller impact on code complexity than feature implementations. Surprisingly, metrics explicitly referring to the size of a change set, such as number of statements, are not among the top ten most influential complexity metrics.

VI. Predicting Defects (RQ2)

This series of experiments is dedicated to research question *RQ2*: how do defect prediction models compare with defect prediction models based on code complexity or code dependency network metrics? We do not aim to build the best prediction models possible and thus did not make any performance tuning optimizations when training the different prediction models. Our prediction models are trained to classify source code files as containing at least one defect or no defect.

A. Experimental Setup

We reuse the basic experimental setup as described in Section V. The only difference is the used data collection. To train and test classification models on code complexity and network metrics, we can reuse the originally generated set of metrics as described in Section IV-B and Section IV-C. The set of change genealogy metrics cannot be reused without modification. Change genealogy metrics are collected on change set basis but not on source file level. Thus, we have to convert the change genealogy metric set to the source file level. For each source file of the 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 resulting data collection is illustrated in Figure 7.

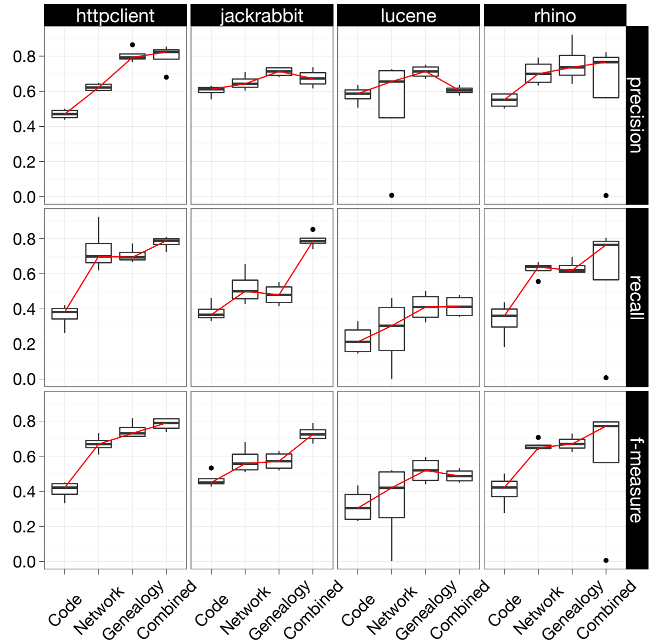

Fig. 8. Results from the repeated holdout experimental setup. Note that the *Combined* label refers to the combined metric set containing call graph network metrics and change genealogy metrics.

B. Prediction Results

Results from the defect prediction experimental setup are presented in Figure 8. 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, code dependency network metrics, change genealogy metrics, and a combined set *Combined* that contains code dependency and change dependency network metrics. For each run we computed precision, recall an F-measure values. The 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. Additionally, we performed a non-parametric statistical test (Kruskal-Wallis) to statistically compare the results.

The results shown in Figure 8 suggest that network metrics outperform code complexity metrics. Network metric prediction models show better precision and recall values for all four subject projects. Change genealogy models report up to 20% (on average 10%) less false positives (higher recall) when compared to code network metric models. At the same time, recall values for change genealogy models drop slightly in comparison to network metric based models. The statistical tests (Kruskal-Wallis) showed that the differences in classification performances are statistically significant ($p < 0.05$).

Models trained on feature vectors combining code dependency and change dependency network metrics show better precision values for HTTPCLIENT and RHINO but worse precision values for LUCENE when compared to models trained on change genealogy metrics, only. The precision values for LUCENE even drop below the precision values of the corresponding network metric models. But interestingly, models trained using the combined metric sets show better recall values for all four projects. For three out of four projects, the recall values are considerable increased (HTTPCLIENT, JACKRABBIT, RHINO).

C. Influential Metrics

We used the same strategy as described in Section V-C to determine top-10 most influential metrics. For three out of four projects (HTTPCLIENT, JACKRABBIT, RHINO) seven of the ten most influential metrics are change genealogy metrics. Only for LUCENE the top-10 most influential metrics contains no change genealogy metric.

We observed three different patterns with respect to presence and ranking of network and change genealogy metrics. Each of the four top-10 most influential metric sets contained one of the *EffSize* or *Efficiency* metrics as the most important network metrics. For HTTPCLIENT, JACKRABBIT, and RHINO the top two most influential metrics were change genealogy metrics describing the relation between a change set and its dependencies to earlier applied change sets (outgoing dependencies). The number and type of the dependency parents as well as the time span between the change set and its parents seem to be crucial. The higher the number of parents and the longer the time span between a change set and its parents the higher the probability to add new defects. Thus, code entities that got applied many change set combining multiple older functions together are more likely to be defect prone than other.

VII. THREADS TO VALIDITY

Empirical studies like this one have threats to validity. We identified three noteworthy threats:

Change Genealogies. First and most noteworthy, change genealogies model only a 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 [38] also apply to this study.

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

Study subject. Third, 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.

REFERENCES

[1] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, "Predicting fault incidence using software change history," *IEEE Trans. Softw. Eng.*, vol. 26, no. 7, pp. 653–661, Jul. 2000.

[2] A. E. Hassan and R. C. Holt, "The top ten list: Dynamic fault prediction," in *Proceedings of the 21st IEEE International Conference on Software Maintenance*, ser. ICSM '05. IEEE Computer Society, 2005, pp. 263–272.

[3] S. Kim, T. Zimmermann, E. J. Whitehead Jr., and A. Zeller, "Predicting faults from cached history," in *Proceedings of the 29th international conference on Software Engineering*, ser. ICSE '07. IEEE Computer Society, 2007, pp. 489–498.

[4] 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*, ser. ICSE '08. ACM, 2008, pp. 181–190.

[5] 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*, ser. ISSRE '10. IEEE Computer Society, 2010, pp. 309–318.

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

[7] 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*, ser. ISSRE '09. IEEE Computer Society, 2009, pp. 109–119.

[8] 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*, ser. ISSRE '11. IEEE Computer Society, 2011, pp. 60–69.

[9] 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*, ser. RSSE '08. ACM, 2008, pp. 30–32.

[10] D. M. German, A. E. Hassan, and G. Robles, "Change impact graphs: Determining the impact of prior code changes," *Inf. Softw. Technol.*, vol. 51, no. 10, pp. 1394–1408, Oct. 2009.

[11] 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*, ser. WCRE ’09. IEEE Computer Society, 2009, pp. 21–30.

[12] K. S. Herzig, “Capturing the long-term impact of changes,” in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ser. ICSE ’10. ACM, 2010, pp. 393–396.

[13] 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*, ser. ESEM ’11. IEEE Computer Society, 2011, pp. 215–224.

[14] 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*, ser. PROMISE ’09. ACM, 2009, pp. 5:1–5:9.

[15] D. Čubranić and G. C. Murphy, “Hipikat: recommending pertinent software development artifacts,” in *Proceedings of the 25th International Conference on Software Engineering*, ser. ICSE ’03. IEEE Computer Society, 2003, pp. 408–418.

[16] M. Fischer, M. Pinzger, and H. Gall, “Populating a release history database from version control and bug tracking systems,” in *Proceedings of the International Conference on Software Maintenance*, ser. ICSM ’03. IEEE Computer Society, 2003, pp. 23–32.

[17] T. Zimmermann, R. Premraj, and A. Zeller, “Predicting defects for Eclipse,” in *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, ser. PROMISE ’07. IEEE Computer Society, 2007, pp. 9–.

[18] A. E. Hassan, “Automated classification of change messages in open source projects,” in *Proceedings of the 2008 ACM symposium on Applied computing*, ser. SAC ’08. ACM, 2008, pp. 837–841.

[19] S. Kim, J. E. James Whitehead, and Y. Zhang, “Classifying software changes: Clean or buggy?” *IEEE Transactions on Software Engineering*, vol. 34, no. 2, pp. 181–196, March/April 2008.

[20] A. Murgia, G. Concas, M. Marchesi, and R. Tonelli, “A machine learning approach for text categorization of fixing-issue commits on cvs,” in *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM ’10. ACM, 2010, pp. 6:1–6:10.

[21] K. Chatterjee, L. de Alfaro, V. Raman, and C. Sánchez, “Analyzing the impact of change in multi-threaded programs,” in *Proceedings of the 13th international conference on Fundamental Approaches to Software Engineering*, ser. FASE’10. Springer-Verlag, 2010, pp. 293–307.

[22] J. Díaz, J. Pérez, J. Garbajosa, and A. L. Wolf, “Change impact analysis in product-line architectures,” in *Proceedings of the 5th European conference on Software architecture*, ser. ECSA’11. Springer-Verlag, 2011, pp. 114–129.

[23] M. Lee, A. J. Offutt, and R. T. Alexander, “Algorithmic analysis of the impacts of changes to object-oriented software,” in *Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 34’00)*, ser. TOOLS ’00. IEEE Computer Society, 2000, pp. 61–.

[24] B. J. Williams and J. C. Carver, “Characterizing software architecture changes: A systematic review,” *Inf. Softw. Technol.*, vol. 52, pp. 31–51, January 2010.

[25] A. Tang, A. Nicholson, Y. Jin, and J. Han, “Using bayesian belief networks for change impact analysis in architecture design,” *J. Syst. Softw.*, vol. 80, pp. 127–148, January 2007.

[26] B. Fluri and H. C. Gall, “Classifying change types for qualifying change couplings,” in *Proceedings of the 14th IEEE International Conference on Program Comprehension*, ser. ICPC ’06. IEEE Computer Society, 2006, pp. 35–45.

[27] D. Kawrykow and M. P. Robillard, “Non-essential changes in version histories,” in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE ’11. ACM, 2011, pp. 351–360.

[28] V. R. Basili, L. C. Briand, and W. L. Melo, “A validation of object-oriented design metrics as quality indicators,” *IEEE Trans. Softw. Eng.*, vol. 22, no. 10, pp. 751–761, Oct. 1996.

[29] 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*, ser. ISSTA ’04. ACM, 2004, pp. 86–96.

[30] 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*, ser. SIGSOFT ’08/FSE-16. ACM, 2008, pp. 2–12.

[31] 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*, ser. ICSE ’08. ACM, 2008, pp. 521–530.

[32] A. E. Hassan, “Predicting faults using the complexity of code changes,” in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE ’09. IEEE Computer Society, 2009, pp. 78–88.

[33] 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*, ser. ISESE ’06. ACM, 2006, pp. 18–27.

[34] 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*, ser. MSR ’09. IEEE Computer Society, 2009, pp. 61–70.

[35] 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*, ser. ESEM ’07. IEEE Computer Society, 2007, pp. 364–373.

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

[37] A. Mockus and L. G. Votta, “Identifying reasons for software changes using historic databases,” in *Proceedings of the International Conference on Software Maintenance (ICSM’00)*, ser. ICSM ’00. IEEE Computer Society, 2000, pp. 120–.

[38] K. Herzig, S. Just, and A. Zeller, “It’s not a bug, it’s a feature: How misclassification impacts bug prediction,” Universität des Saarlandes, Saarbrücken, Germany”, Tech. Rep., August 2012.

[39] “Jhawk 5 (release 5 version 1.0.1),” available at: http://www.virtualmachinery.com/jhawkprod.htm.

[40] T. J. McCabe, “A complexity measure,” *IEEE Trans. Software Eng.*, vol. 2, no. 4, pp. 308–320, 1976.

[41] S. R. Chidamber and C. F. Kemerer, “A metrics suite for object oriented design,” *IEEE Trans. Softw. Eng.*, vol. 20, no. 6, pp. 476–493, Jun. 1994.

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

[43] G. Csardi and T. Nepusz, “The igraph software package for complex network research,” *InterJournal*, vol. Complex Systems, p. 1695, 2006. [Online]. Available: http://igraph.sf.net

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

[45] “Mozkito (0.4-snapshot),” available at: http://mozkito.org.

[46] M. Kuhn, *caret: Classification and Regression Training*, 2011, R package version 4.76. [Online]. Available: http://cran.r-project.org/web/packages/caret/caret.pdf