Condensing Class Diagrams by Analyzing Design and Network Metrics using Optimistic Classification

Ferdian Thung[1], David Lo[1], Mohd Hafeez Osman[2], and Michel R.V. Chaudron[3]
[1]School of Information Systems, Singapore Management University, Singapore
[2]Leiden University, Netherlands
[3]Chalmers University of Technology, Sweden
{ferdiant.2013,davidlo}@smu.edu.sg, hosman@liacs.nl, chaudron@chalmers.se

ABSTRACT

A class diagram of a software system enhances our ability to understand software design. However, this diagram is often unavailable. Developers usually reconstruct the diagram by reverse engineering it from source code. Unfortunately, the resultant diagram is often very cluttered; making it difficult to learn anything valuable from it. Thus, it would be very beneficial if we are able to condense the reverse- engineered class diagram to contain only the important classes depicting the overall design of a software system. Such diagram would make program understanding much easier. A class can be important, for example, if its removal would break many connections between classes. In our work, we estimate this kind of importance by using design (e.g., number of attributes, number of dependencies, etc.) and network metrics (e.g., betweenness centrality, closeness centrality, etc.). We use these metrics as features and input their values to our optimistic classifier that will predict if a class is important or not. Different from standard classification, our newly proposed optimistic classification technique deals with data scarcity problem by optimistically assigning labels to some of the unlabeled data and use them for training a better statistical model. We have evaluated our approach to condense reverse-engineered diagrams of 9 software systems and compared our approach with the state-of-the-art work of Osman et al. Our experiments show that our approach can achieve an average Area Under the Receiver Operating Characteristic Curve (AUC) score of 0.825, which is a 9.1% improvement compared to the state-of-the-art approach.

Categories and Subject Descriptors

D.2.7 [**Software**]: Software Engineering—*Distribution, Maintenance, and Enhancement*; H.2.8 [**Information Systems**]: Database Applications—*Data Mining*

General Terms

Algorithms, Experimentation

ICPC'14, June 2–3, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2879-1/14/06...$15.00
http://dx.doi.org/10.1145/2597008.2597157

Keywords

Design Metrics, Network Metrics, Optimistic Classification, Important Classes

1. INTRODUCTION

How could we understand a software system? Referring to software design is clearly an obvious choice. It informs us about the internal working of the software and how its components are connected to one another. Among design models specified in the Unified Modeling Language (UML), one of the most widely used model is class diagram. Class diagram specifies relationships (e.g., aggregation, composition, generalization, etc.) between classes in a software system. These relationships provide the basic knowledge needed to understand a software system. However, such class diagram documentation is often not kept up-to-date. It may be created and maintained properly at the beginning of a project, but is often abandoned somewhere in the middle of the project. Thus, it is harder to understand a software if a developer joins in the latter part of development due to the non existence of an up-to-date design documentation. The condition is even worse for legacy systems where such documentation is often no longer available.

When an up-to-date design is unavailable, a reverse engineered design is often generated as a replacement. Reverse engineering refers to the process of analyzing system components and behaviours in order to construct an abstract representation of high level design of a system. More specifically, reverse engineering class diagram means analyzing the implemented source code and creating a class diagram representation abstracting the attributes and methods in a class and the relationships between different classes. Many open source and commercial tools have been developed to perform this operation [1, 2, 3].

However, the reverse-engineered class diagram often contains too much detail that grows as the size of a software increases. When a reverse- engineered class diagram becomes too large, it provides little benefit towards program understanding. Fernandez-Saez et al. find that many subjects in their controlled experiment "did not consider RE [Reverse-Engineered] diagrams helpful" [4]. It was found that developers prefer forward design class diagram rather than its reverse-engineered counterpart. They found that it is much easier to find relevant information from forward design class diagrams and the sizes of these diagrams are often smaller than the reverse- engineered ones. Given a reverse-engineered class diagram, it is hard to identify which classes are more important than others. Although some Computer

Aided Software Engineering (CASE) tools let users remove properties from a class diagram, none is capable to identify the importance of a class in a diagram.

The lack of useful class diagrams for program understanding introduces a hurdle for a new software engineer that joins a software development team. The new engineer would have no good reference for understanding a software system. Given a good class diagram, the engineer can learn the overall architecture of a software system, know what design patterns are used in different parts of the system, or implement new classes that are consistent with the overall design of the system.

To address the problem described above, Osman et al. proposed an approach that can condense a reverse-engineered class diagram into another diagram that is close to a forward design diagram [5]. A diagram that is closer to the design documentation is likely to provide better program comprehension support. To do so, their proposed approach needs to identify important classes. The final condensed diagram can be constructed from the reverse-engineered diagram by keeping the important classes and discarding unimportant ones. Osman et al. computed values of various design features from source code, which include values of size and coupling metrics, and use these to predict if a class is important or not. They assume a partial knowledge about class importance exists (i.e., some classes have been labeled as important or not). They input the values of these features to a classification algorithm which creates a model that can predict if a class is important or not based on its features. They have investigated a number of classification algorithms and find that *random forest* performs the best.

In this work, we extend Osman et al.'s work with the goal of improving the accuracy of their proposed approach. We extend the set of features that Osman et al. used with network features. To obtain the values of these network features, we first create a network where the classes form the nodes of the network and the relationships among classes form the edges. These relationships include aggregation, composition, generalization, realization, and dependency. In this study, we do not differentiate these relationships. Thus, our graph only has one type of edge. We then compute the values of several standard network metrics to estimate the importance of a class based on the generated network structure. We also propose several other customized network metrics to better characterize the importance of a class. We use both design and network metrics as features and input the values of these features to our optimistic classification technique. Our newly proposed optimistic classification technique builds upon standard classification to deal with data scarcity problem by optimistically assigning labels (i.e., important or unimportant) to some of the unlabeled data points (i.e., classes whose importance are unknown) and use them for training a final statistical model. Our optimistic classification technique first learns a preliminary model from a training data set (i.e., classes whose importance are known) using a standard classifier (e.g., random forest) to assign probability scores to unknown data points. It then selects a small subset of the unknown data points which are likely to be important with high probability scores. These small subset of data points are optimistically assigned labels "important", merged with the original training data, and used to train a final statistical model.

We have done an experiment to measure the effectiveness of our approach to condense reverse-engineered class diagrams from nine software systems. We generate the features for each software system and split the data randomly into evenly sized training and testing data following the experimental procedure of Osman et al. [5]. We evaluate our approach on the testing data and repeat the process with different training and testing data ten times. We compare our approach with the state-of-the-art approach proposed by Osman et al. We use Area Under the Receiver Operating Characteristic Curve (AUC), which is a standard metric and was also used by Osman et al., as the evaluation metric. Since our process repeats for ten times, we get ten AUC scores, and we report the mean. Averaging across the 9 programs, our experiment shows that our approach can achieve an average AUC of 0.825. This is a 9.1% improvement to the result achieved by Osman et al.'s approach.

The contributions of our work are as follows:

1. We are the first to combine both design and network metrics to condense a reverse-engineered class diagram by predicting if a class is important or not.

2. We propose optimistic classification which optimistically assign labels to some unlabeled data and treat them as part of the training data to build a final statistical model.

3. We have evaluated our approach to condense reverse-engineered class diagrams from nine systems. These systems are previously used by Osman et al. [5] to evaluate their approach. The experiments show that our approach can achieve a high average AUC score of 0.825 and it improves the average AUC score achieved by Osman et al. by 9.1%.

The structure of the remainder of this paper is as follows. In Section 2, we describe our proposed approach. We then describe our experiments in Section 3. We present related work in Section 4. We finally conclude and mention future work in Section 5.

2. PROPOSED APPROACH

In this section, we first describe our overall framework. We then zoom-in to two components of our framework: our feature extractor component and our optimistic classification component.

2.1 Overall Framework

Figure 1 depicts the overall framework of our approach. It consists of two phases: training phase and deployment phase. In training phase, the goal is to learn a statistical model that can differentiate important reverse-engineered classes from unimportant ones. In this phase, we accept a set of training classes[1] whose labels (i.e., important or unimportant) are known. Based on these labeled data, we construct the statistical model. In deployment phase, we use the model that we construct in the training phase to predict whether an unlabeled reverse- engineered class is important or not.

In training phase, our framework first extracts the values of various features from the training reverse-engineered

[1]In this paper, we use the term class loosely to include concrete classes, abstract classes, and interfaces.

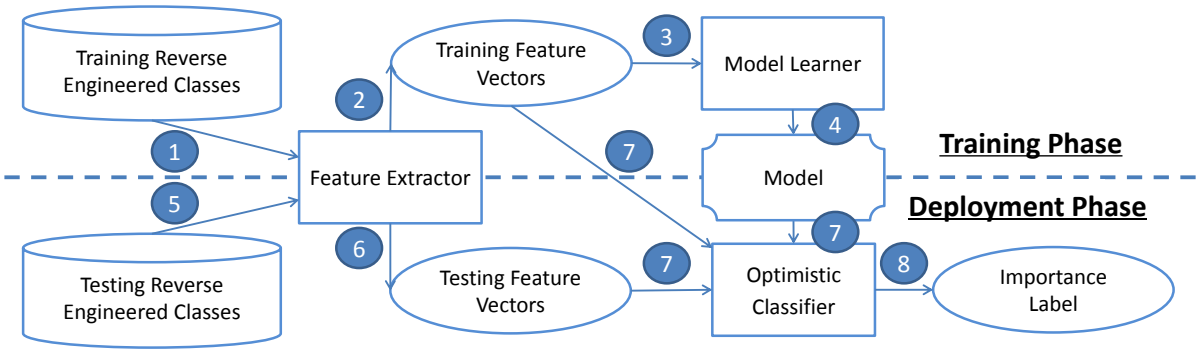

Figure 1: Overall Framework

classes of a software system (Step 1). A feature corresponds to a particular characteristics of a reverse-engineered class. Given a reverse-engineered class, we compute a score for each feature. The values of all these features form a feature vector. Our framework creates a feature vector for each training reverse-engineered class to form the set of *Training Feature Vectors* (Step 2). Each class in the training data has an importance label which is a binary value. Our *Model Learner* component takes these vectors as input and constructs a statistical model (*Model*) that is able to predict if a class is important or not from its feature vector (Steps 3-4).

In deployment phase, our framework first extracts features from a set of reverse-engineered testing classes whose importance labels are to be determined (Step 5). Similar to the training phase, our framework then creates a feature vector for each reverse-engineered class to form the set of *Testing Feature Vectors* (Step 6). The feature vectors are then input to *Optimistic Classifier* component. This component also takes two other inputs: *Model* and *Training Feature Vectors* (Step 7). This component would eventually produce a set of *Importance Labels* for the testing reverse-engineered classes (Step 8).

In Section 2.2, we explain the feature extraction process, which is performed by the *Feature Extractor* component. Section 2.3 elaborates the optimistic classification process that is performed by our *Optimistic Classifier* component.

2.2 Feature Extractor

The goal of the feature extractor component is to convert reverse-engineered classes in the training and testing data into *Training Feature Vectors* and *Testing Feature Vectors* respectively. A feature vector corresponds to a vector of values where each value corresponds to a particular characteristic of a reverse-engineered class. *Training Feature Vectors* are used to construct a statistical model, while *Testing Feature Vectors* are used to predict labels of reverse-engineered classes in the testing data.

We consider two kinds of features: design features and network features. We present design features in Section 2.2.1. Network features are then presented in Section 2.2.2.

2.2.1 Design Features

For design features, we use size and coupling metrics. Size metrics measure the size of a class in various ways. Coupling metrics measure the strengths of relationships that a class has with other classes. These metrics were used by Osman et al. to predict the importance of a class [5]. They experimented with three combinations of these metrics: size metrics only, coupling metrics only, and both of these metrics. They found that using both size and coupling metrics lead to the best performance. Thus in this work, we also use both size and coupling metrics as design features.

For completeness sake, we include a brief description of these metrics in Table 1. There are 5 size metrics and 6 coupling metrics.

2.2.2 Network Features

Network features include standard network metrics and our customized metrics. The standard metrics characterize the importance of a class in a network in which the nodes are classes and the edges are various relationships between pairs of classes. The customized metrics characterize the likelihood of a node to be important considering its neighbors in the network based on homophily principle [9]. The relationships in the network are derived considering object-oriented architecture in which classes are associated with other classes by means of aggregation, composition, generalization, realization, and dependency. Below are short definitions of the kinds of relationships that we consider in this work:

1. **Aggregation**

   Aggregation is a structural relationship between two classes in a system where one class owns another class. The owner class is often called container class while the owned class is called contained class. In this relationship, the destruction of a container class instance does not imply the destruction of contained class instance.

2. **Composition**

   Composition is a structural relationship similar with aggregation. However, it is a stronger relationship since the destruction of a container class instance would also imply the destruction of a contained class instance.

3. **Generalization**

   Generalization represents an inheritance relationship between two classes in a system where one class is a parent (i.e., super-class) and the other is the child (i.e., sub-class of the parent class). The child inherits attributes and methods from the parent.

Table 1: Size and Coupling Metrics

Metrics	Category	Description
NumAttr	Size	The number of attributes in a class.
NumOps	Size	The number of methods in a class. Also known as WMC in [6] and NM in [7].
NumPubOps	Size	The number of public methods in a class. Also known as NPM in [7].
Setters	Size	The number of methods whose names start with 'set'.
Getters	Size	The number of methods whose names start with 'get', 'is', or 'has'.
DepOut	Coupling	The number of dependencies where a class uses other classes.
DepIn	Coupling	The number of dependencies where a class is used by other classes.
ECAttr	Coupling	The number of times a class is externally used as an attribute type. This is a version of OAEC +AAEC in [8].
ICAttr	Coupling	The number of attributes in a class having another class or interface as their types. This is a version of OAIC+AAIC in [8].
ECPar	Coupling	The number of times a class is externally used as a parameter type. This is a version of OMEC+AMEC in [8].
ICPar	Coupling	The number of parameters in a method of a class having another class or interface as their types. This is a version of OMIC+AMIC in [8].

4. **Realization**

   Realization represents an implementation of an interface. Interface is a contract specifying the methods that a class must implement.

5. **Dependency**

   Dependency represents a relationship where a class depends on another class for its implementation. This includes a class that is used as a type of a parameter in a method that another class has.

Based on the above relationships, we want to construct a customized directed network linking all the classes. An edge in the network connecting two nodes, corresponding to two classes, means that the two classes are associated by one of the above mentioned relationship types. The edge is a directed one and its direction is determined based on the relationship type. Consider a class C, we define 3 types of edges incident to it: (1) If C is a child of parent P, then there exists a link from P to C; (2) If C implements interface I, then there exists a link from I to C; (3) If C has any of the remaining relationships with class K, then there exists a link from C to K. These links are illustrated in Figure 2.

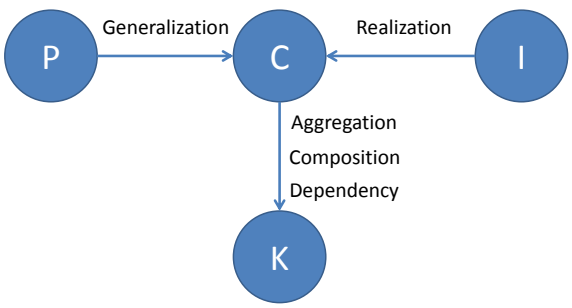

Figure 2: Possible Edges Incident to Class C

Using the class network that we have constructed, we then compute some standard and customized network metrics. The following is a list of standard network metrics that we consider as features in our approach. The standard metrics are used to measure the importance of a node (i.e., a class) in a network.

1. **Barycenter Centrality**

   Barycenter centrality is defined based on the sum of shortest distances of node v to all other nodes in a network. The barycenter centrality of node v is computed using the following formula:

   $$baryC(v) = \frac{1}{\sum_{u \neq v} sdist(v,u)}$$

   In the equation, $sdist(v,u)$ refers to the shortest distance from node v to node u.

2. **Betweenness Centrality**

   Betweenness centrality is defined based on the number of shortest paths between all possible pairs of other nodes that pass through node v. The betweenness centrality of node v is formulated as follows:

   $$betweenC(v) = \sum_{a \neq b \neq v} \frac{spath(a,b,v)}{spath(a,b)}$$

   In the equation, $spath(a,b,v)$ refers to the number of shortest paths between node a and node b that pass through node v. $spath(a,b)$ refers to the number of shortest paths between node a and node b.

3. **Closeness Centrality**

   Closeness centrality is defined based on the mean shortest distance of node v to all the other nodes in a network. The closeness centrality of node v is computed using the following formula:

   $$closeC(v) = \frac{n-1}{\sum_{u \neq v} sdist(v,u)}$$

In the equation, $sdist(v, u)$ refers to the shortest distance from node v to node u. n refers to the number of nodes in the graph.

4. **Eigenvector Centrality**

   Eigenvector centrality measures the importance of node v based on the importance of its neighboring nodes. The eigenvector centrality $EigenC$ for a network is measured using the following formula:

$$EigenC(\alpha, \beta) = \alpha(I - \beta R)^{-1} R1$$

   In the equation, α is a scaling vector for normalizing the score, I is the identity matrix, R is the adjacency matrix representing the network, β is the weighting factor for the adjacency matrix, and 1 is a matrix where the contents of all its cells are ones. Since the value of this metric is often very small, in this work, we compute the reciprocal of this metric.

5. **Hyperlink-Induced Topic Search (HITS)**

   HITS is an algorithm for ranking nodes using two different scores: hub and authority score. A node with a high hub score represents a node that links to many other nodes and a node with a high authority score represents a node that is linked by many different nodes. These scores are computed by the following formulas:

$$hub(v) = \sum_{i=1}^{n} auth(v)$$

$$auth(v) = \sum_{i=1}^{n} hub(v)$$

   In the equation, n refers to the number of node in a network, $hub(v)$ refers to the hub score for node v, and $auth(v)$ refers to the authority score for node v. Notice that the definition is a recursive one. To actually arrive with the hub and authority scores for all nodes, one must first assign an initial value of 1 as hub and authority scores to each of the nodes in the network. The scores would then be updated iteratively until they converge (i.e., there is no further change in any node's hub and authority scores in the entire network). Both hub and authority values are then normalized. Since the values of this metric may be very small, in this work, we compute the reciprocal of this metric.

6. **PageRank**

   PageRank is an algorithm for measuring node importance proposed by Brin and Page [10]. It suggests that nodes with more incoming links are more important than nodes with less incoming links. It computes the probability that a random walker visits a node from an arbitrary node. Initially, all nodes are assigned with the same initial probability. The scores are then iteratively updated. The PageRank score of node v at iteration i can be computed following the formula:

$$PR(v, i) = \frac{1 - r}{T} + r \sum_{u \in K(v)} \frac{PR(q, i - 1)}{|L(u)|}$$

   In the equation, r is the probability that a random walker continues to visit other nodes (a.k.a. the *damping factor*), T is the number of nodes in the network, $K(v)$ is the set of nodes that link to v, and $L(u)$ is the set of nodes that u links to. The iteration continues until all the scores converge. Since the value of these two metrics are often very small, in this work, we compute the reciprocals of these metrics.

Besides the standard network metrics, we also create some customized network metrics specialized for our classification problem. Following the setting of Osman et al., we assume that a partial knowledge about class importance exists; this would mean that for some of the classes in the network, we know whether they are important or not [5]. Based on this partial knowledge, we want to compute some customized metrics which try to characterize whether a node is an important one or not based on homophily principle. In this way, the customized metrics are *supervised* ones (i.e., created based on knowledge of class labels in the training data) while the standard metrics are *unsupervised* ones. These supervised metrics are particularly useful for our optimistic classification technique (see Section 2.3). Below is a list of our customized supervised network metrics that are used as features in our approach:

1. **Important Neighbor Proportion**

   This metric measures the proportion of known important classes among the neighbors of a class in the network.

2. **Unknown Neighbor Proportion**

   This metric measures the proportion of classes whose importance are unknown among neighbors of a class.

3. **Shortest Distance to Known Important Classes**

   This metric measures the shortest distance to any one of the known important classes.

4. **Neighbor Existence**

   This is a boolean metric which describes whether a class in the network has a neighbor or not.

We summarize the above network features in Table 2.

2.3 Optimistic Classification

In the training data, the number of reverse-engineered classes labeled as important is often small. Thus, it is often hard to generate a good statistical model which distinguishes between important and unimportant classes. Our optimistic classification technique addresses this data scarcity issue by optimistically assigning labels to some testing data points whose labels are unknown.

The technique takes as input a preliminary model learned from the training data. It then applies the model to each testing data point (i.e., each reverse-engineered class whose importance is unknown) to assign to each data point a probability for it to be important. It then tries to refine the preliminary model using some of the testing data that have the highest probabilities to be important. Our classification technique *optimistically* assumes that the reverse-engineered test classes with the highest probabilities are indeed important. Thus, we add these reverse-engineered classes to the

Table 2: Standard and Customized Network Metrics

Metrics	Description
Barycenter	The barycenter centrality score of a class in the network.
Betweenness	The betweenness centrality score of a class in the network.
Closeness	The closeness centrality score of a class in the network.
Eigenvector	The eigenvector centrality score of a class in the network.
Hub	The hub score of a class in the network.
Authority	The authority score of a class in the network.
PageRank	The page rank score of a class in the network.
PropImportant	The proportion of a class neighbors that are known to be important.
PropUnknown	The proportion of a class neighbors whose importance are unknown.
SDistToDesign	The shortest distance to a known important neighbor.
HasNeighbor	A boolean value indicating whether a class has a neighbor(s) or not.

```
1: Input:
2:    TrainingFeatureVectors = feature vectors of training
                               reverse-engineered classes
3:    TestingFeatureVectors = feature vectors of testing
                              reverse-engineered classes
4:    Model = a learned model in the training phase
5:    k = parameter for picking top-k scores
6: Output:
7:    Importance labels for TestingFeatureVectors
8: Method:
9: Labels = {}
10: Scores = {}
11: for all fv ∈ TestingFeatureVectors do
12:    Add prob(Model, fv) to Scores
13: end for
14: TopScores = Pick top-k Scores
15: for all fv ∈ TestingFeatureVectors do
16:    Let score = prob(Model, fv)
17:    if score ∈ TopScores then
18:       fv.Label ="Important"
19:       Add fv to TrainingFeatureVectors
20:    end if
21: end for
22: Update TrainingFeatureVectors
23: Update TestingFeatureVectors
24: Model' = Learn a model from TrainingFeatureVectors
25: for all fv ∈ TestingFeatureVectors do
26:    label = classify(Model', fv)
27:    Labels = Labels ∪ {label}
28: end for
29: return Labels
```

Figure 3: Optimistic Classification Algorithm

training data and retrain a new model from the combined data. As we are likely to have more important reverse-engineered classes than before in the updated training data, it is expected that a better model could be learned and a better classification accuracy can be achieved.

Figure 3 shows the pseudocode of our proposed technique. It takes as input *TrainingFeatureVectors*, *TestingFeatureVectors*, and a *Model* learned from the *TrainingFeatureVectors*. Using the *Model*, it first computes a probability score, for each reverse-engineered class, whose feature vector appears in *TestingFeatureVectors* (lines 11-13). It then picks the top-k probability scores (line 14). Next, it iterates the classes that have corresponding entries in *TestingFeatureVectors* again, and the classes having one of the top-k probability scores are labeled as "important" and added to the *TrainingFeatureVectors* (lines 15-21). In this way, we optimistically assume that these classes are indeed important. In lines 22-23, we update the feature vectors in *TrainingFeatureVectors* and *TestingFeatureVectors* based on the additional important classes. In particular, the values of the features corresponding to the customized network metrics, which are described in Section 2.2.2, might need to be updated. The algorithm then learns a new model based on the updated *TrainingFeatureVectors* (line 24). Finally, it performs classification using the new model on *TestingFeatureVectors* (lines 25-28). The resultant *importance labels* are then output (line 29).

3. EXPERIMENTS & ANALYSIS

In this section, we describe our dataset, evaluation measure, and experimental settings. We then list our research questions followed by our experiment results which answer these questions. We finish by discussing some interesting points and threats to validity.

3.1 Dataset

We use the same dataset that was used in Osman et al. work [5]. The dataset consists of 9 projects that are chosen based on these criteria: the project should be open source to ensure replicability of the findings, the project should contain more than 50 classes, the forward design class diagram should exist. The chosen projects and their characteristics are shown in Table 3.

For each project, we generate a reverse-engineered class diagram using Magic Draw version 1.0 (academic evaluation license).[2] Our goal is to compress this diagram by identifying important classes. To get the ground truth labels (i.e., important or unimportant), we use the same procedure employed by Osman et al. [5]. The reverse-engineered and forward design class diagrams are compared. A class that exists both in the reverse- engineered and forward design class diagrams is labeled as an important class. A class that exists in the reverse-engineered class diagram, but not in the forward design class diagram is labeled as unimportant class.

Based on Table 3, we can see that the proportion of important classes in a reverse-engineered class diagram range from 3.45% to 47.45%. xUML and Maze have relatively bal-

[2] `http://www.nomagic.com/products/magicdraw.html`

Table 3: Our Dataset

ID	Project	Description	Total Classes in Reversed Engineered Diagram (S)	Total Classes in Forward Design Diagram (D)	D:S ratio as %
1.	ArgoUML[3]	UML diagramming application.	903	44	4.87
2.	JGAP[4]	A framework for performing genetic algorithms and genetic programming.	171	18	10.52
3.	JPMC[5]	A collection of automated intelligent agents in financial sector.	121	24	19.83
4.	JavaClient[6]	A framework for developing robotics applications.	214	57	26.64
5.	Mars[7]	An application for creating simulation of possible human settlement in planet Mars.	840	29	3.45
6.	Maze[8]	An application for solving maze puzzles.	59	28	47.45
7.	Neuroph[9]	A framework for developing neural network architectures.	161	24	14.90
8.	Wro4J[10]	An application for optimizing web resources.	87	11	12.64
9.	xUML[11]	A software for producing executable and testable system from a specified data model and associated state machines.	84	37	44.05

anced distributions of important and unimportant classes. The other projects, however, have very unbalanced distributions of important and unimportant classes, with important classes being the minority. Thus, in 7 out of the 9 projects, we have a data scarcity problem for classes labeled as important.

3.2 Evaluation Measure

We use Area Under the Receiver Operating Characteristic Curve (AUC) to evaluate our prediction performance. The same measure was used by Osman et al. to evaluate their approach [5]. AUC measures the ability of a classification algorithm to correctly rank classes as important or unimportant. Receiver Operating Characteristic Curve (ROC) is a two-dimensional measure of classification performance. It is a plot of the true positive rate versus false positive rate. The larger an ROC area is, the better a classification algorithm is in term of its ability to classify classes correctly as important or unimportant. The AUC score range from 0 to 1, with 1 representing perfect prediction performance.

The AUC score is used because this measure is suitable for highly imbalanced data [11]. In 7 out of the 9 projects, we have imbalanced data (i.e., there are substantially more unimportant than important classes). AUC is able to address the issue of favoring models that trivially predict the majority outcome label (i.e., "unimportant") for all data points. Many past software engineering works also use AUC as an evaluation metric, e.g., [12, 13, 14] and an AUC score above 0.7 is considered reasonable [13, 14].

3.3 Experimental Settings

For the *Model Learner* component, which converts *Training Feature Vectors* to a preliminary *Model*, we use the random forest classification algorithm. Random forest basically constructs a number of decision trees based on different subset of features and perform classification based on each decision tree. The classification from each tree are counted and the majority is chosen as the classification output of random forest. We use the implementation of random forest available in Weka [15]. In Osman et al.'s work, random forest has been shown to be the best performing algorithm [5]. In our experiments, we want to investigate whether adding network features and employing optimistic classification help.

To compute the values of the design features which correspond to size and coupling metrics, we use SDMetrics version 2.2 (academic license).[12] To compute the values of the network features, we need to first create a network of classes. To create this network, we make use of SDMetrics Open Core API [13] to parse a UML class diagram and extract classes (which would correspond to nodes in the network) and relationships of interest (which would correspond to edges in the network). To compute the standard network measures, we make use of Java Universal Network/Graph Framework (JUNG).[14]

We set the parameter k of our optimistic classification algorithm (shown in Figure 3) to 5% of the total number of data points in the testing data. It means that we only

[3] http://argouml.tigris.org
[4] http://sourceforge.net/projects/jgap
[5] http://jpmc.sourceforge.net
[6] http://java-player.sourceforge.net
[7] http://mars-sim.sourceforge.net
[8] http://code.google.com/p/maze-solver
[9] http://neuroph.sourceforge.net
[10] http://code.google.com/p/wro4j
[11] http://code.google.com/p/xuml-compiler
[12] http://www.sdmetrics.com
[13] http://www.sdmetrics.com/OpenCore.html
[14] http://jung.sourceforge.net

optimistically consider top 5% classes in the testing data to be important classes and use them to retrain the model.

To evaluate our proposed approach, we need to divide our dataset (i.e., reverse-engineered classes) into training and testing data. We follow the same procedure used by Osman et al. to create these training and test data and to evaluate our approach. For each project, 50% randomly selected reverse-engineered classes are used as training data and the rest are used as testing data. We use the test data to evaluate the performance of our approach and Osman et al.'s approach. For reliability, we repeat this process 10 times using different training and test data that are randomly constructed, and report the average performance across the 10 repetitions.

3.4 Research Questions

To demonstrate the effectiveness of our approach over the state-of-the-art work, we investigate the following research questions:

Research Question 1. How effective is our proposed approach in recovering important classes in a reverse- engineered class diagram?

The bottom line of a classification-based technique is its accuracy. A better technique should achieve a higher accuracy on various data. To answer this question, we measure the effectiveness of our approach in predicting important classes in terms of AUC for each of the 9 programs. We also compare and contrast the AUC scores that are achieved by our approach with those that are achieved by Osman et al.'s approach.

Research Question 2. Individually, are the network metrics and optimistic classification techniques helpful to improve the effectiveness of our approach?

Our work extends Osman et al.'s approach, by introducing two additional things. First, we introduce the network metrics. We also introduce a new classification approach, namely optimistic classification. In this research question, we are interested to evaluate whether each of these two additions is helpful to boost effectiveness of our proposed approach. To answer this question, we create an instance of our approach that use network metrics but not optimistic classification. We then compare this instance with Osman et al.'s approach and our full approach.

Research Question 3. What are the most discriminative features in classifying the important classes?

Not all features are equally important in the classification process. Some features may have an edge over the others in discriminating between important and unimportant classes. We are interested to find out the best features that can discriminate between different classes. To answer this research question and identify discriminative features, we measure information gain score of each feature. Information gain has often been used before to measure the importance of a feature [16, 17, 18]. Information gain is also used by the random forest algorithm to build each decision tree [18].

3.5 RQ1: Effectiveness of Our Approach

We show the AUC scores of Osman et al.'s approach (baseline) and our approach (ours) in Table 4. For Osman et al.'s approach, we show the results for the random forest algorithm which is the best performing algorithm. From the table, the AUC scores of our approach range from 0.757-0.915. We achieve the lowest AUC score for ArgoUML and the best AUC score for Neuroph. Averaging across the 9 programs, our average AUC is 0.825.

From the results, it is clear that our approach can improve Osman et al.'s approach for all of the 9 programs. The AUC improvements range from 2.1% (JavaClient) to 17.5% (JPMC). Averaging across the 9 programs, our approach improves the AUC score of Osman et al.'s approach by 9.1%. For 5 out of the 9 programs, the AUC score improvement is close to or larger than 10%. These results show that our approach is more effective than the state-of-the-art approach.

Table 4: Effectiveness of Our Approach

ID	Project	AUC(baseline)	AUC(ours)	Improv.
1.	ArgoUML	0.655	0.757	15.6%
2.	JGAP	0.748	0.797	6.6%
3.	JPMC	0.692	0.813	17.5%
4.	JavaClient	0.844	0.862	2.1%
5.	Mars	0.766	0.845	10.3%
6.	Maze	0.674	0.767	13.8%
7.	Neuroph	0.835	0.915	9.6%
8.	WroJ	0.742	0.763	2.8%
9.	xUML	0.847	0.905	6.9%
	Average	**0.756**	**0.825**	**9.1%**

3.6 RQ2: Benefits of Network Metrics and Optimistic Classification

Our approach is build on top of Osman et al's approach by the addition of two new steps: the use of network metrics as features and the use of optimistic classification. Table 5 shows the improvement made when we include network metrics in the set of features and do not employ optimistic classification. From the table, we can see that on average, the AUC increases from 0.756 (Osman et al.'s approach) to 0.810 (our approach without optimistic classification). Thus there is an improvement of 7.1%. The AUC improvements range from 2.6% (JavaClient) to 13.7% (JPMC). These results show that our network metrics are effective to improve classification performance.

Table 5: Improvement by Using Network Metrics

No	Project	AUC(before)	AUC(after)	Improv.
1.	ArgoUML	0.655	0.737	12.5%
2.	JGAP	0.748	0.778	4.0%
3.	JPMC	0.692	0.787	13.7%
4.	JavaClient	0.844	0.866	2.6%
5.	Mars	0.766	0.797	4.1%
6.	Maze	0.674	0.727	7.9%
7.	Neuroph	0.835	0.918	9.9%
8.	Wro4J	0.742	0.780	5.1%
9.	xUML	0.847	0.900	6.3%
	Average	**0.756**	**0.810**	**7.1%**

Table 6 compares the results of our approach (with design and network metrics) with and without optimistic classification. On average, AUC is increased from 0.810 (random forest) to 0.825 (optimistic classification). Thus there is a 1.9% improvement in terms of AUC. The AUC improvements range from -2.2% (Wro4j) to 6.0% (Mars). We notice that there are several projects where our optimistic classi-

fication approach achieves negative improvements (the performance without optimistic classification is better). However, we only see this for 3 projects: JavaClient, Neuroph, and Wro4j. It means that for 67% of the projects, optimistic classification can improve performance. Thus overall, the results show that our optimistic classification is effective. Admittedly, the improvement gained by using optimistic classification is less than the improvement gained by using network metrics.

Table 6: Improvement by Optimistic Classification

No	Project	AUC(before)	AUC(after)	Improv.
1.	ArgoUML	0.737	0.757	2.7%
2.	JGAP	0.778	0.797	2.4%
3.	JPMC	0.787	0.813	3.3%
4.	JavaClient	0.866	0.862	-0.5%
5.	Mars	0.797	0.845	6.0%
6.	Maze	0.727	0.767	5.5%
7.	Neuroph	0.918	0.915	-0.3%
8.	Wro4J	0.780	0.763	-2.2%
9.	xUML	0.900	0.905	0.6%
	Average	**0.810**	**0.825**	**1.9%**

3.7 RQ3: Most Discriminative Features

To find the most discriminative features across the projects, for each training and test data pair, we compute the information gain score for each feature. For each project, we compute the average information gain score per feature across the 10 training and test data pairs. We then pick the top-5 features with the highest average information gain scores for each project. We exclude features having zero information gain score. We then compute the number of times each feature appears in the top-5 lists (#Appearance). We show the list of the top-10 features based on their number of appearances in Table 7.

From the table, our network features dominate the list (7 out of 10 most discriminative features) and appear in the top-5 most discriminative features. It shows that the network features can discriminate important classes better than design features. This fact partly explains the improvement made by using network features. For design features, *Dep_In*, *EC_Par* , and *EC_Attr* discriminate better than the other design features.

Table 7: Top-10 Most Discriminative Features

ID	Feature	Category	#Appearance
1.	Authority	Network	5 (55.6%)
2.	Barycenter	Network	4 (44.5%)
3.	Betweenness	Network	4 (44.5%)
4.	Eigenvector	Network	3 (33.4%)
5.	PropImportant	Network	3 (33.4%)
6.	Dep_In	Design	3 (33.4%)
7.	EC_Par	Design	3 (33.4%)
8.	PageRank	Network	3 (33.4%)
9.	Hub	Network	3 (33.4%)
10.	EC_Attr	Design	3 (33.4%)

3.8 Discussion

Various network metrics are computed as features in our proposed approach. The computation of a network metric generally has a high complexity. For example, to compute closeness centrality, the shortest distance between each pair of nodes must be computed. This translates to a complexity of $O(n^3)$. However, the adjacency matrix corresponding to the network representing a class diagram is typically very sparse and the number of classes in most software system is not astronomical – compared with social network having millions to billions of nodes. Thus, running time is typically not a problem. In our experiment, we can compute all network metrics for the entire dataset in under ten seconds.

In our optimistic classification step, we pick the top 5% test instances to optimistically retrain the classification model. As an alternative, we can also set a fixed absolute threshold (e.g., 0.99) and pick only test instances whose probability scores are higher than this threshold. We have tried this alternative and it performs worse than picking top 5% test instances. This may be due to the different characteristics of each software system and different discriminative power of network or design features when applied in different software systems which causes the optimal absolute threshold to vary for different software systems.

3.9 Threats to Validity

Threats to Internal Validity. Threats to internal validity relates to errors and biases. We have rechecked our implementation. Still, there could be errors that we do not notice. Also, since we reuse Osman et al.'s dataset, our study suffers from the same possibility of errors in the collection of the dataset. For example, the documentation might be outdated or may not contain the most relevant classes. To reduce experimenter bias, we choose to reuse the same dataset, evaluation measure, and experiment settings that was used to evaluate Osman et al.'s work.

Threats to External Validity. Threats to external validity relates to the generalizability of our findings. We have tried to reduce these threats to external validity by evaluating our approach on 9 different programs. In the future, we plan to reduce these threats further by considering additional programs of various sizes written in multiple programming languages.

Threats to Construct Validity. Threats to construct validity relates to the suitability of our evaluation metric. We have made use of AUC, which is a standard metric in data mining [18] and it is designed to evaluate imbalanced data [11]. Many studies in software engineering also use AUC as evaluation metric, e.g., [12, 13, 14]. AUC was also used by Osman et al. to evaluate their proposed approach. Thus, we believe there is little threat to construct validity.

4. RELATED WORK

The most related work to ours is the recent study by Osman et al. [5]. Aside from this work, there are a number of works that also assign importance to classes in various ways. We highlight these works in Section 4.1. We also highlight some software engineering works that employ classification techniques in Section 4.2

4.1 Assigning Importance to Classes

Zaidman and Demeyer [19] proposed the use of HITS web mining techniques to identify key classes in a system. Dynamic analysis of the source code was used in their study as the input of the proposed method. The validation of this study was done manually by comparing the result of the approach with the classes specified in the software doc-

umentation. Our approach extends Zaidman and Demeyer approach, by using not only HITS metric, but also other network metrics (standard and customized) and the design metrics to classify important classes.

Perin et al. [20] proposed the use of PageRank algorithm to rank software artifacts. Two case studies were used: Pharo Smalltalk system and Moose reengineering environment. They considered important classes to be classes mentioned in the documentation. Our approach extends Perin et al.'s approach, by using not only PageRank metric, but also other network and design metrics. We integrate all these metrics together by using an optimistic classification technique.

Steidl et al. [21] investigated the usage of a number of network metrics including PageRank, HITS, and several others to identify important classes of a system. They would like to analyze which metrics and settings are better to identify important classes. Our approach extends Steidl et al.'s approach in several ways: we use a classification-based approach to combine all the network metrics together to predict if a class is important or not, we propose a number of new network metrics that characterize the importance of a class based on homophily principle, and we combine the network metrics with design metrics to improve performance.

Hammad et al. proposed an approach that assigns importance scores to classes and sets of collaborating classes based on the number of times these classes were changed in a version control system [22]. To assign importance scores to classes, they identified commits made to a version control system that impacted design. They then counted, for each class, the number of such commits that include changes to the class. To assign importance scores to a set of collaborating classes, they used frequent itemset mining algorithm. Different from Hammad et al.'s work, we consider a different criteria to judge the importance of a class. Our goal is to reduce a reverse-engineered class diagram such that it is closer to a forward design diagram. Our work is motivated by the recent study of Fernandez-Saez et al. which finds that forward design class diagram is more useful than its reverse-engineered counterpart [4].

Bieman et al. [23] proposed a method to identify and visualize classes that are frequently changed. They presented measurements for class change-proneness, i.e., local change-proneness, pair change coupling and sum of pair coupling. They identified change clusters based on the measures. These clusters often reside around key components. They then construct a change-prone class diagram and change architecture diagram for visualization. Different from Bieman et al.'s work we do not capture change-prone classes, rather our goal is to reduce a reverse-engineered class diagram such that it is closer to a forward design diagram.

4.2 Other Related Works

There are many software engineering studies that also employed classification techniques to predict various information [24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35]. We highlight some of them below. The survey here is by no means complete.

Giger et al. used a decision tree based algorithm to categorize bug reports based on their resolution time into two classes: "slow" and "fast" [24]. Menzies and Marcus used a classification algorithm named Ripper to predict the severity of bug reports from NASA [25]. Lamkanfi et al. investigated the effectiveness of a number of classification algorithms in predicting the severity of bug reports [26]. Tian et al. used a classification engine named GRAY which extends linear regression to predict the priority of bug reports [27]. Thung et al. used SVM to categorize bug reports into three families [28].

Maiga et al. used support vector machine (SVM) as a classification algorithm to detect anti-patterns which are instances of poor design that causes problems for program comprehension [31]. Hou and Mo used Naive Bayes to assign semantic labels to discussions in various software forums [32]. Swapna et al. used a special variant of support vector machine (SVM) that uses Hidden Markov Model (HMM) to categorize posts in software forums [33]. Le and Lo used an extended Support Vector Machine (SVM) to predict if a fault localization instance is effective or not [34]. Prasetyo et al. used an SVM to categorize if a microblog is relevant to engineering software system or not [35].

5. CONCLUSION AND FUTURE WORK

Recently, Osman et al. have proposed an approach for condensing a reverse-engineered class diagram by using design metrics as input to a standard classification technique. In this work, we want to improve the effectiveness of this state-of-the-art approach. To achieve our goal, we introduce additional features which include standard and customized network metrics which characterize the importance of a class and its likelihood to be important based on the homophily principle. We also propose a new classification technique which we refer to as optimistic classification. Different from standard classification, optimistic classification *optimistically* assigns labels to some unlabeled data, and use these newly labeled data to generate a better statistical model. To investigate the effectiveness of our proposed approach, we have conducted an experiment on reverse-engineered classes of 9 programs, which were also used to evaluate Osman et al.'s approach. Our approach achieves an average AUC score of 0.825 which improves the average AUC score achieved by Osman et al.'s approach by 9.1%. Each of the steps in our approach also performs well. By adding network metrics to Osman et al.'s approach we can achieve an AUC of 0.810 which is a 7.1% improvement. Our optimistic classification technique further improves performance by another 1.9%. We have also measured the discriminativeness of each feature using information gain and created a list of top-10 most discriminative features. We find that our network metrics are on the top of the list. It explains the improvement that we achieve by adding the network metrics. We also found several design features that can discriminate better compared to other features in the same category. These feature are *Dep_In*, *EC_Par*, and *EC_Attr*.

As future work, we plan to improve the effectiveness of our approach further. To achieve that goal, we would like to investigate and propose additional metrics that could be used to differentiate between important and unimportant classes better. We also plan to reduce some threats to validity. One way is by evaluating our approach on more reverse-engineered class diagrams from additional programs of various sizes and written in a variety of programming languages.

Dataset. We make our dataset publicly available and it can be downloaded from: `http://sites.google.com/site/classdiag/dataset.zip`.

6. REFERENCES

[1] MagicDraw. http://www.nomagic.com/products/magicdraw.html.

[2] eUML2. http://www.soyatec.com/euml2/com.soyatec.uml.doc/.

[3] Class Visualizer. http://www.class-visualizer.net/.

[4] Ana M. Fernandez-Saez, Michel R. V. Chaudron, Marcela Genero, and Isabel Ramos. Are Forward Designed or Reverse-Engineered UML Diagrams More Helpful for Code Maintenance?: A Controlled Experiment. In *International Conference on Evaluation and Assessment in Software Engineering*, pages 60–71, 2013.

[5] Mohd Hafeez Osman, Michel R. V. Chaudron, and Peter van der Putten. An Analysis of Machine Learning Algorithms for Condensing Reverse Engineered Class Diagrams. In *International Conference on Software Maintenance*, pages 140–149, 2013.

[6] Shyam R. Chidamber and Chris F. Kemerer. A Metrics Suite for Object Oriented Design. *IEEE Trans. Software Eng.*, 20(6):476–493, 1994.

[7] Al Lake and Curtis Cook. Use of factor analysis to develop OOP software complexity metrics. In *Proc. 6th Annual Oregon Workshop on Software Metrics, Silver Falls, Oregon*, 1994.

[8] Lionel C. Briand, Premkumar T. Devanbu, and Walcélio L. Melo. An Investigation into Coupling Measures for C++. In *International Conference on Software Engineering*, pages 412–421, 1997.

[9] Jiliang Tang, Huiji Gao, Xia Hu, and Huan Liu. Exploiting homophily effect for trust prediction. In *International Conference on Web Search and Data Mining*, pages 53–62, 2013.

[10] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual Web search engine. In *International Conference on World-Wide Web*, pages 107–117, 1998.

[11] Foster J. Provost, Tom Fawcett, and Ron Kohavi. The Case against Accuracy Estimation for Comparing Induction Algorithms. In *International Conference on Machine Learning*, pages 445–453, 1998.

[12] Ahmed Lamkanfi, Serge Demeyer, Emanuel Giger, and Bart Goethals. Predicting the severity of a reported bug. In *Mining Software Repositories*, pages 1–10, 2010.

[13] Stefan Lessmann, Bart Baesens, Christophe Mues, and Swantje Pietsch. Benchmarking Classification Models for Software Defect Prediction: A Proposed Framework and Novel Findings. *IEEE Trans. Software Eng.*, 34(4):485–496, 2008.

[14] Daniele Romano and Martin Pinzger. Using source code metrics to predict change-prone Java interfaces. In *International Conference on Software Maintenance*, pages 303–312, 2011.

[15] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The weka data mining software: an update. *SIGKDD Explorations*, 11(1):10–18, 2009.

[16] Hong Cheng, Xifeng Yan, Jiawei Han, and Chih-Wei Hsu. Discriminative Frequent Pattern Analysis for Effective Classification. In *International Conference on Data Enginering*, pages 716–725, 2007.

[17] Xifeng Yan, Hong Cheng, Jiawei Han, and Philip S. Yu. Mining significant graph patterns by leap search. In *SIGMOD Conference*, pages 433–444, 2008.

[18] J. Han and M. Kamber. *Data Mining Concepts and Techniques*. Morgan Kaufmann, 2nd edition, 2006.

[19] Andy Zaidman and Serge Demeyer. Automatic identification of key classes in a software system using webmining techniques. *Journal of Software Maintenance*, 20(6):387–417, 2008.

[20] Fabrizio Perin, Lukas Renggli, and Jorge Ressia. Ranking Software Artifacts. In *Workshop on FAMIX and Moose in Reengineering*, 2010.

[21] Daniela Steidl, Benjamin Hummel, and Elmar Jürgens. Using Network Analysis for Recommendation of Central Software Classes. In *Working Conference on Reverse Engineering*, pages 93–102, 2012.

[22] Maen Hammad, Michael L. Collard, and Jonathan I. Maletic. Measuring Class Importance in the Context of Design Evolution. In *International Conference on Program Comprehension*, pages 148–151, 2010.

[23] James M. Bieman, Anneliese Amschler Andrews, and Helen J. Yang. Understanding Change-Proneness in OO Software through Visualization. In *International Workshop on Program Comprehension*, pages 44–53, 2003.

[24] Emanuel Giger, Martin Pinzger, and Harald Gall. Predicting the fix time of bugs. In *International Workshop on Recommendation Systems for Software Engineering*, 2010.

[25] T. Menzies and A. Marcus. Automated Severity Assessment of Software Defect Reports. In *International Conference on Software Maintenance*, 2008.

[26] A. Lamkanfi, S. Demeyer, Q.D. Soetens, and T. Verdonck. Comparing Mining Algorithms for Predicting the Severity of a Reported Bug. In *European Conference on Software Maintenance and Reengineering*, 2011.

[27] Yuan Tian, David Lo, and Chengnian Sun. Drone: Predicting priority of reported bugs by multi-factor analysis. In *International Conference on Software Maintenance*, 2013.

[28] Ferdian Thung, David Lo, and Lingxiao Jiang. Automatic defect categorization. In *Working Conference on Reverse Engineering*, 2012.

[29] Hongyu Zhang, Liang Gong, and Steve Versteeg. Predicting bug-fixing time: an empirical study of commercial software projects. In *International Conference on Software Engineering*, 2013.

[30] Michael Gegick, Pete Rotella, and Tao Xie. Identifying security bug reports via text mining: An industrial case study. In *Mining Software Repositories*, pages 11–20, 2010.

[31] Abdou Maiga, Nasir Ali, Neelesh Bhattacharya, Aminata Sabane, Yann-Gaël Guéhéneuc, and Esma Aïmeur. SMURF: A SVM-based Incremental Anti-pattern Detection Approach. In *Working Conference on Reverse Engineering*, pages 466–475, 2012.

[32] Daqing Hou and Lingfeng Mo. Content Categorization

of API Discussions. In *International Conference on Software Maintenance*, pages 60–69, 2013.

[33] Swapna Gottipati, David Lo, and Jing Jiang. Finding relevant answers in software forums. In *International Conference on Automated Software Engineering*, pages 323–332, 2011.

[34] Tien-Duy B. Le and David Lo. Will fault localization work for these failures? an automated approach to predict effectiveness of fault localization tools. In *nternational Conference on Software Maintenance*, 2013.

[35] Philips Kokoh Prasetyo, David Lo, Palakorn Achananuparp, Yuan Tian, and Ee-Peng Lim. Automatic classification of software related microblogs. In *International Conference on Software Maintenance*, 2012.