Predicting Project Outcome Leveraging Socio-Technical Network Patterns

Didi Surian*, Yuan Tian†, David Lo†, Hong Cheng‡ and Ee-Peng Lim†
*School of Information Technologies
University of Sydney, Australia
Email: dsur5833@uni.sydney.edu.au
†School of Information Systems
Singapore Management University, Singapore
Email: {yuan.tian.2012,davidlo,eplim}@smu.edu.sg
‡Department of Systems Engineering and Engineering Management
The Chinese University of Hong Kong, Hong Kong
Email: hcheng@se.cuhk.edu.hk

***Abstract*—There are many software projects started daily; some are successful, while others are not. Successful projects get completed, are used by many people, and bring benefits to users. Failed projects do not bring similar benefits. In this work, we are interested in developing an effective machine learning solution that predicts project outcome (i.e., success or failures) from developer socio-technical network.**

To do so, we investigate successful and failed projects to find factors that differentiate the two. We analyze the socio-technical aspect of the software development process by focusing at the people that contribute to these projects and the interactions among them. We first form a collaboration graph for each software project. We then create a training set consisting of two graph databases corresponding to successful and failed projects respectively. A new data mining approach is then employed to extract discriminative *rich* patterns that appear frequently on the successful projects but rarely on the failed projects. We find that these automatically mined patterns are effective features to predict project outcomes. We experiment our solution on projects in SourceForge.Net, the largest open source software development portal, and show that under 10 fold cross validation, our approach could achieve an accuracy of more than 90% and an AUC score of 0.86. We also present and analyze some mined socio-technical patterns.

***Keywords*-software project; collaboration graph; discriminative pattern; graph mining**

I. Introduction

Some software projects, either open source or industrial, are successful. They are completed, used by many people, and bring benefits to the various stakeholders including users, developers, and companies. Many others however remain uncompleted, buggy, or are only used by a few. Investigating the nature of successful and failed projects could shed light to reasons or factors why some projects are successful while others fail. These factors in turn could be used to build a machine learning model to predict project outcome. Predicting project outcome is useful for planning, mitigation, and management reasons.

Due to the advent of Web 2.0, open source code, and software repositories, much information on past project successes and failures could easily be obtained. Features related to the various software projects could be extracted. This study aims to leverage these features to find discriminative patterns that differentiate successful from failed projects. A pattern is discriminative if it appears frequently in the successful projects but rarely in the failed projects and vice versa. Discriminative patterns characterize features closely associated to successful or failed projects. We use these patterns as features and build an effective classifier to predict project outcome.

First, we extract the socio-technical aspects of software development process by performing a longitudinal study on various snapshots of a super-repository. A super-repository contains much information including the people contributing to various projects and the number of downloads. Various snapshots of the super-repository could be taken to form a richly labeled graph that characterizes people involved in various projects and the nature of the relationships between them across time. We can characterize project success or failures based on the number of downloads. From this information, two sets of graph databases, one for successful and another for failed projects could be formed. Each graph in the database corresponds to a project with nodes corresponding to people involved in the project and edges corresponding to relationships among them. Various features, e.g., number of completed projects, number of collaborations, etc, could be attached as labels to the nodes and edges.

Next, to analyze these graphs, we extend past studies on discriminative graph mining that mine for subgraphs that frequently occur in one set of graphs but rarely in the opposing set [17], [5]. Past studies on discriminative graph mining are only able to mine from *simple* graphs with *single* labels on the nodes and edges. We extend their work to mine from *rich* graphs containing *multiple* labels on the nodes and edges. To realize this, we propose a novel approach to translate a rich graph into a simple one and adopt an existing algorithm [5] to mine for discriminative subgraphs from the

1534-5351/13 $26.00 © 2013 IEEE
DOI 10.1109/CSMR.2013.15

IEEE computer society

translated graphs. The results are then reverse translated to form rich graphs. We show that our translation is sound and complete.

Finally, based on the mined patterns, we train a machine learning model to predict project outcome. Based on a training data, a set of patterns would be mined, and the existence or absence of the patterns in each data instance is used as binary features for classifier construction. The mined patterns and the classifier are then used to predict instances on test data whose outcome is to be determined.

We experiment our solution on SourceForge.Net, one of the most popular and largest sites to download open-source software. Our experiments demonstrate that we can predict project outcome with an accuracy of over 90% and an AUC (Area under ROC) score of 0.86. We also show and analyze the set of top-20 most discriminative patterns that we mine.

We describe the contributions of this work as follows:

1. We introduce a new problem of predicting project outcome using socio-technical network patterns.
2. We extend a solution that mines discriminative simple graph patterns to mine discriminative *rich* graph patterns. We do so by proposing a translation process to map the problem to mining simple graph patterns and reverse translate the mined simple graph patterns to rich graph patterns.
3. We prove that our translation is sound and complete. This means that all mined patterns are discriminative and all discriminative rich graph patterns can be mined.
4. We show the scalability of our approach in analyzing SourceForge.Net dataset. Experiment results show that our approach can predict project outcome with high accuracy. We also present and analyze the mined discriminative patterns.

Section II introduces the concept of rich graph which is used to represent a software project. Section III outlines our overall framework. We zoom into the various components of the framework in the following sections. First, Section IV elaborates our approach to model the socio-technical aspects of projects. Next, Section V explains how discriminative patterns are mined. Finally, Section VI describes our process of utilizing mined patterns as effective features for project outcome prediction. Section VII presents our experiments. We present related work in Section VIII. We conclude and present future work in Section IX.

II. Preliminaries & Notations

We introduce the notions of simple graph and rich graph along with some notations in Definitions 2.1 & 2.2 respectively. The example for a simple graph and rich graph is shown in Figure 1(a) and 1(b) respectively. We use a rich graph to represent a software project based on the experience history of its developers.

*Definition 2.1 (***Simple Graph***):* A simple graph is a set of nodes N, edges E, and labels L. Each node and edge could be attached with *one* label from L. Each edge (u,v) is a pair of nodes in N. Since E is a set, each pair of nodes could be linked by only one edge. Given a node n we denote edges incident to it and its label by $n.Edges$ and $n.Label$ respectively. We denote the label of an edge e by $e.Label$. Given an edge e of a node n, we refer to the other node connected to n by e as $e.Target$.

*Definition 2.2 (***Rich Graph***):* A rich graph is a set of nodes N, node placeholders NP, edges E, edge placeholders EP, and labels L. Each node and edge contains *multiple* placeholders each of which contains one label from L. Each edge (u,v) is a pair of nodes in N. Each pair of nodes could be linked by one edge. Given a placeholder $np \in \{NP \cup EP\}$, we denote its labels by $np.Label$. Given a node n we denote edges incident to it and labels contained in its placeholders by $n.Edges$ and $n.Labels$ respectively. We denote the labels of an edge e contained in its placeholders by $e.Labels$. Given an edge e of a node n, we refer to the other node connected to n by e as $e.Target$.

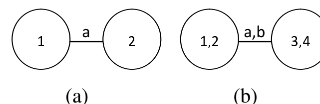


Figure 1. Example: (a) Simple Graph (b) Rich Graph

III. Overall Framework

Our framework has two phases: training and testing (or deployment). In the training phase, we extract discriminative features and learn a classifier from a training dataset containing projects with known outcomes (i.e., success or fail). In the testing phase, based on the existence or absence of the discriminative features, we apply the learned classifier to predict the outcome of the projects. Our framework is illustrated in Figure 2.

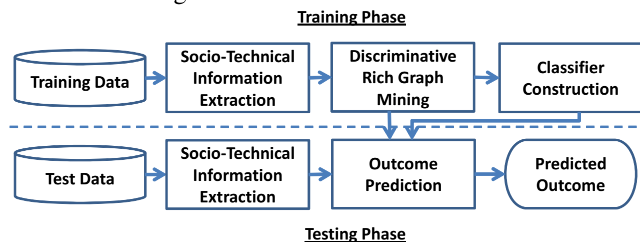


Figure 2. Overall Framework

During the training phase, we take as input a set of projects, and developers that work on them. For each project, we first extract socio-technical information from the developers working on the project in the form of a rich graph. Each node corresponds to a developer[1] and each edge

[1]A developer could possibly register himself using several usernames, however in this study we assume that each developer is identified by a unique username, which is intuitive for a developer to keep his reputation record. Studies on developer matching could potentially be applied [6], [8] but we leave this for future work.

corresponds to the relationship between two developers. We attach multiple labels capturing various information related to developers and their relationships. Next, we mine discriminative subgraph patterns that appear frequently in successful projects but rarely in failed projects (or vice versa). We treat the existence or absence of mined patterns as binary features (each is assigned either 0 (absence) or 1 (existence) value) and use them to build a classifier. The classifier and the patterns are forwarded to the test/deployment phase.

During the testing phase, for each project with unknown outcome, again we first extract its socio-technical information as a graph. We then check the existence or absence of the discriminative patterns in the graph and form binary features. The learned classifier is then used to predict the outcome of the project based on the binary features.

We provide more details on how socio-technical information is extracted in Section IV. Additional information on how discriminative patterns are mined are provided in Section V. We elaborate classifier construction and output prediction in Section VI.

IV. Modeling Socio-Technical Aspects of Projects

Many super-repositories, e.g., SourceForge.Net, provide information on the collaborations among software developers and contributors[2] over various projects. Each project is contributed by one or more persons. They could contribute to the projects either in terms of source code, ideas, planning, etc. Some of the developers of a particular project might have worked together before, while others work together for the first time. Some of the developers might have worked on one or more successful projects before while others have not. These socio-technical aspects of software development process potentially affect the success or failure of a project.

Various measures could be used to identify project success or failures. In our setting, we consider the number of downloads as an indicator of project success or failure. Intuitively, successful projects are downloaded a large number of times, while failed/unsuccessful projects are only downloaded a few number of times by users.

For each project, whether successful or not, we could extract some features representing the socio-technical aspects of the developers working on it. We consider 6 different features characterizing the co-contributors collaborating on the project and the socio-technical relationships between them. Three of them characterize a single developer; the other three characterize the relationship between two developers.

The first feature is the number of past successful projects a developer has before he joins the current project. Past successful projects potentially enrich a developer's experience. Intuitively, a project contributed by one or more experienced contributors is more likely to be successful than another with totally inexperienced contributors. We would like to investigate if this is the case. The first feature is defined in Definition 4.1.

*Definition 4.1 (***Past Successful Projects (PSP)):** Consider a developer D joining a project P at time t. The number of past successful projects, denoted as PSP, is the number of projects with D joining as a contributor before t, and are successful.

The second feature is the number of past unsuccessful projects a developer has before he joins the current project. It is possible that past failures correlate with future failures. On the other hand, developers could potentially learn from past failures to eventually contribute to a successful project. We define the second feature in Definition 4.2.

*Definition 4.2 (***Past Failed Proj. (PFP)):** Consider a developer D joining a project P at time t. The number of past failed projects, denoted as PFP, is the number of projects with D joining as a contributor before t, and are failed.

The third feature is the length of time a developer has joined the super-repository at the time he joins a project. A senior member could be more aware of the demand of the potential clients and could be more adept in selecting suitable group members to build a good software together. This feature is defined in Definition 4.3.

*Definition 4.3 (***Length of Membership (LOM)):** Consider a contributor D registered in the super repository at time s and joins a project P at time t. The length of membership, denoted as LOM, is the period of time from s to t, or mathematically, t - s.

The fourth feature characterizes the relationship between two developers. We consider the number of past successful collaborations two developers have before they start to work together[3] in the current project. Intuitively, if two developers have already had many successful collaborations before, the current collaboration would more likely be a successful one too. For such cases, the developers would likely know the working or coding style of one another well and are able to work well together thus contributing to project success. Definition 4.4 defines the fourth feature.

*Definition 4.4 (***Past Successful Collab. (PSC)):** Consider t as the time when a developer $D1$ starts to collaborate with a developer $D2$ in a project P. The number of past successful collaborations, denoted as PSC, is the number of projects with $D1$ and $D2$ joining as contributors before t, and are successful.

The fifth feature is the number of past failed projects two developers have before they start to collaborate in the current project. Bad experiences could either lower morale or act as stepping stones towards successful collaborations. We define this in Definition 4.5.

[2]We use the terms developer and contributor interchangeably in this paper.

[3]We assume that a developer still contributes to a project if he is still recorded in the repository.

*Definition 4.5 (***Past Failed Collab. (PFC)***):* Consider t as the time when a developer $D1$ starts to collaborate with another developer $D2$ in a project P. The number of failed collaborations, denoted as PFC, is the number of projects with $D1$ and $D2$ joining as contributors before t, and are failed.

The last feature we consider is the length of time that has passed, since the first time two developers worked together, prior to the current project. A long collaboration history might increase the likelihood of developing a good project. On the other hand, although two developers have a long history, if they rarely work together, they might not collaborate well anymore. We present this feature in Definition 4.6.

*Definition 4.6 (***Len. of Collab. History (LCH)***):* Consider s as the time a developer $D1$ collaborates with another developer $D2$ for the first time. Let t be the the time $D1$ collaborates with $D2$ in the current project P. The length of $D1$'s and $D2$'s collaboration history when they collaborate on P, is the period of time from s to t, or mathematically, t - s.

For each project, we extract the above socio-technical features and express them as an undirected graph. Each node in the graph corresponds to a developer, while each edge corresponds to a relationship between two developers. We attach labels to the nodes and edges of the graph to capture the above six features. Three of them relate to the individual developers and are thus mapped to node labels. The other three relate to the relationships between two developers and are thus mapped to edge labels. Thus, every node is attached with PSP, PFP, and LOM. Also, every edge is attached with PSC, PFC, and LCH. We refer to such a graph as a *socio-technical graph*. Note that a socio-technical graph is a rich graph.

We would analyze a set of socio-technical graphs from many projects, some of which are successful projects while others are failed projects, to empirically validate whether any combinations of the above features could discriminate successful from failed projects and predict project outcome.

V. Mining Discriminative Rich Sub-Graph Patterns

In this section, we first describe discriminative sub-graph mining that could be applied to mine for discriminative success-failure patterns. We then present our solution that extends the state-of-the-art work on mining discriminative simple subgraphs to mine for discriminative *rich* subgraphs where each node and edge can contain multiple labels.

A. Discriminative Sub-Graph Mining

Given a set of socio-technical graphs representing a set of successful projects and another set representing failed projects, we aim to find discriminative subgraphs that could distinguish successful projects from failed ones. Intuitively, a subgraph is discriminative if it occurs frequently in the socio-technical graphs of the successful projects but rarely in those of the failed ones (and vice versa). Theoretically, we could design an objective function $F(g)$ to evaluate the discriminative score of a subgraph g. Then our goal becomes finding the optimal subgraph wrt. the objective function F. Formally, the mining problem is defined as:

*Definition 5.1 (***Mining Discriminative Subgraph***):* Given a set of graphs with class labels D=$\{g_i,y_i|1$<=i<=$n\}$, where g_i is a graph representing a project and $y_i \in \{-1,+1\}$ is the class label representing a successful or unsuccessful project, an objective function F which measures the discriminative score of a subgraph, find a subgraph g^* such that $g^* = \text{argmax}_g F(g)$.

In data mining, discriminative measures such as information gain, cross entropy and Fisher score are popularly used to evaluate the capacity of a feature in distinguishing instances from different classes. In this work, we use information gain as the objective function. A subgraph which occurs frequently in the socio-technical graphs of the successful projects but rarely in those of the failed ones will have a very large information gain score. Such a discriminative graph highlights the structural contrast between successful and failed projects. If we use c to denote the class label of the projects, and use g to represent a subgraph, then information gain of g is defined as:

$$IG(c|g) = H(c) - H(c|g) \quad (1)$$

where $H(c) = -\sum_{c_i \in \{0,1\}} p(c_i) \log p(c_i)$ is the entropy and $H(c|g) = -\sum p(g) \sum_{c_i \in \{0,1\}} p(c_i|g) \log p(c_i|g)$ is the conditional entropy given the subgraph g.

To efficiently mine the most discriminative subgraph from the successful and failed projects, we adopt a recently proposed graph mining algorithm LEAP [17]. We invoke the LEAP algorithm k times to mine the top-k discriminative subgraphs following a similar approach in [5]. The returned result is a ranked list of k discriminative subgraphs with decreasing information gain scores. We denote the discriminative subgraph mining operation by DGM.

B. Extending to Rich Sub-Graphs

The algorithms in [5], [17] described in Section V-A only mine from a set of simple graphs. In this section, we describe how we extend them to mine for discriminative rich subgraphs with multiple node and edge labels.

We handle the problem of mining discriminative rich subgraphs by introducing an equivalent simple graph representation of a rich graph. We thus propose a novel mapping between mining discriminative rich subgraphs to mining simple discriminative subgraphs. We show that the translation to simple graphs only increases the size of the rich subgraphs by a factor linear to the maximum number of labels per node and the maximum number of node labels per

edge. We also prove that all discriminative rich subgraphs could be mined using the approach in Section V-A from the corresponding simple graph translations.

Our process proceeds in the following steps:

- Convert the rich graphs to their corresponding simple graphs representations DB_REP.
- Mine discriminative subgraphs from DB_REP using the approach in Section V-A.
- Convert mined simple subgraph patterns to its corresponding rich subgraph patterns.

The next sub-section describes our translation process. A reverse translation process is presented next. We then analyze some properties of our translation process.

1) Translation Process: A rich graph can have multiple node and edge labels. Our translation process converts a rich graph into a simple graph by performing node and edge *replication* operations. Each replica only retains *one* of the potentially many labels of the original node or edge that it replicates.

We first introduce the notions of translated simple graph in Definitions 5.2. In a translated simple graph, the replicas of the same rich graph's node (i.e., its *siblings*) are connected together with a special edge.

*Definition 5.2 (***Translated Simple Graph***):* A translated simple graph is a simple graph with a special edge named sibling-replicated edge (SRE). This edge connects all replicated nodes that are originating from the same node in the original rich graph.

Nodes and edges are replicated due to two reasons:

1) There are multiple node labels in a node of a rich graph. This node would be split into multiple simple graph nodes each with a single label.
2) There are multiple edge labels attached to an edge of a rich graph. As a simple graph does not allow for two edges between two nodes, either of the nodes connected by it would need to be replicated.

We refer to the replicas created due to the first reason as *NL-Replicas*. We refer to the ones created due to the latter reason as *EL-Replicas*. Our translation process first creates NL-Replicas. EL-Replicas are constructed next.

Creating NL-Replicas. To create NL-Replicas from a rich graph, for each node, we split it according to the number of labels that it has. The edges of the original node are transferred to each of its replicas. We also add SRE edges to connect all the nodes originating from the same rich node. The original rich nodes are then removed from the original graph. After all NL-Replicas have been created and rich nodes removed, all nodes in the graph would each have a single node label. The introduction of new edges ensures that the structures expressed in the rich graph are preserved after the introduction of NL-Replicas.

Figure 3 shows the pseudo-code realizing this[4]. We overlay the NL-Replicas on top of the original graph. We create the NL-Replicas one by one and eliminate the original nodes and edges step-by-step. Note that the order of which the nodes are being processed would not affect the NL-Replicas introduced. If two rich nodes $n1$ and $n2$ are connected, each of the NL-replicas of $n1$ is connected to all the NL-replicas of $n2$ (and vice versa). We illustrate the NL-Replicas creation process in Figure 4.

Procedure CreateNLReplicas
Inputs:
$G = (N, NP, E, EP, L)$: A rich graph with the set of nodes N, node placeholders NP, edges E, edge placeholders EP, and labels L
Output: G with nodes replaced with NL-Replicas
Method:
1: Let $Orig_N$ = Shallow copy of N
2: For each $n \in Orig_N$
3: Let $NSet[]$ = Create a node array of size $|n.Labels|$
4: For every ith node in $NSet$
5: Let $NSet[i].Label$ = $n.Labels[i]$
6: For every edge e in $n.Edges$
7: Add an edge with labels $e.Labels$ from $NSet[i]$ to $e.Target$
8: Add SREs that connect nodes in $NSet$ to one another
9: Remove n and all edges connected to it from N and E respectively
10: For every node n_{new} in $NSet[]$
11: Add n_{new} to N
12: Output (N, NP, E, EP, L)

Figure 3. Creation of NL-Replicas

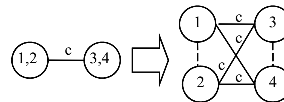


Figure 4. NL-Replicas: Illustration. Edges shown in dashed lines are SREs.

Creating EL-Replicas. To create EL-Replicas from a rich graph, for each edge with multiple labels, we replicate *one of the two nodes (or at most both nodes)* connected by it. Given a node n, connected to a multi-labeled edge e, we create EL-Replicas, by duplicating the node according to the number of labels e has. The original node would be connected with an edge with one of e's labels. Each of the newly introduced EL-Replica nodes would be connected with a new edge with one of the remaining e's labels. The newly introduced EL-Replica nodes are connected to the other replica nodes of the same original rich node by SREs. After all the EL-Replicas are created, the resultant graph would be a simple translated graph composed of NL-Replicas (nodes and edges) and EL-Replicas (nodes and edges).

[4]We only handle the case where there is no self-loop, which is not existent in software socio-technical dataset.

Procedure CreateELReplicas
Inputs:
$G = (N, NP, E, EP, L)$: A graph with NL-Replicas and rich edges
Output: An equivalent translated simple graph with NL- & EL-Replicas
Method:
1: Let $Orig_E$ = Shallow copy of E
2: For each $e \in Orig_E$
3: Let OTR = The node(s) conn. by e to be replicated
4: For every node n of OTR
5: Let $NSet$ = Replicate n, $|e.Labels|$ times
6: For every i^{th} node n' in $NSet$
7: Let l be the ith label in $e.Labels$
8: Add an edge from n' to $e.Target$ with label l
9: For every node n_{new} in $NSet[]$
10: Add n_{new} to N
11: Remove n and all edges connected to it from N and E respectively
12: Add SREs that connect nodes in $NSet$ to one another
13: Remove e from E
14: Output (N, E, L)

Figure 5. Creation of EL-Replicas

The remaining ambiguity is which of the two nodes should be replicated. We use the label of the two nodes to decide. Due to the creation of NL-Replicas, the nodes would have single labels. For a multi-labeled edge connecting two nodes $n1$ and $n2$, there are 3 cases[5]:

1) If $n1.Label < n2.Label$, we would create EL-Replicas of $n1$.
2) If $n1.Label > n2.Label$, we would create EL-Replicas of $n2$.
3) If $n1.Label = n2.Label$, we would create EL-Replicas for both $n1$ and $n2$.

The above cases are used to ensure that the same EL-Replicas are introduced no matter which edges are processed first.

Figure 5 shows the pseudo-code realizing this. We illustrate the EL-Replicas creation process in Figure 6.

Given a rich graph g, its translated simple graph is denoted as TL(g). Also, given a set of rich graphs DB, we denote the corresponding set of translated simple graphs as TL(DB). In the implementation, we combine the NL-Replicas and EL-Replicas creation process so that only one pass through the nodes in the graph is needed. An end-to-end example of how a rich graph is translated into a simple translated graph is shown in Figure 7.

2) Reverse Translation Process: The reverse translation operation is straightforward. We just need to merge every nodes connected by SRE together. These nodes map to the same original rich node. When we merge the nodes we take the union of their node labels. Due to the merging of the nodes, two nodes might have more than one edge connecting them. We would then merge the edges too by again taking

[5]Any arbitrary total ordering on the labels could be used to decide the cases.

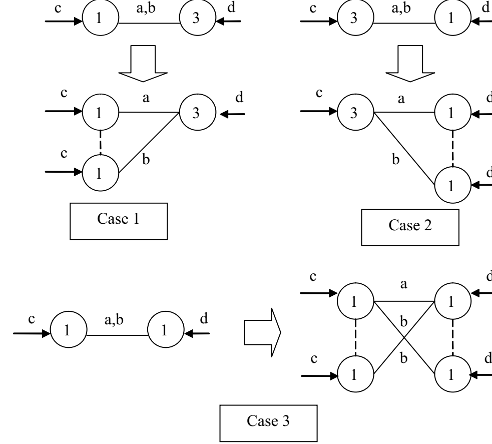


Figure 6. EL-Replicas: Illustration.

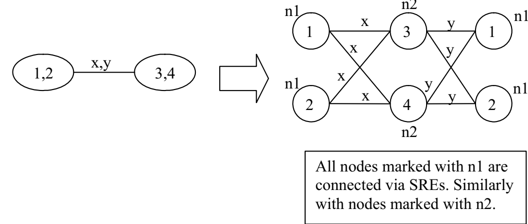


Figure 7. Translation Process

the union of their edge labels. Note that as the set union operation is commutative, associative, and distributive, it does not matter as to which nodes and edges are merged first. The reverse translation operation is deterministic, given one input graph, it would always produce one output graph no matter what nodes and edges are merged first.

We denote the reverse translation operation by RTL. Given a translated simple graph g, the corresponding rich graph after the reverse translation operation is performed is denoted by RTL(g). Also, given a set of translated simple graphs DB, we denote the corresponding set of rich graphs after the reverse translation operations are performed as RTL(DB).

3) Analysis: Theorem 1 assures the correctness of our translation process. We start by translating rich graphs to simple graphs, and then perform mining operation, and finally reverse translate the mined patterns to rich subgraph patterns. The whole process is sound and complete. It is sound as all resultant reverse translated rich subgraphs are discriminative. It is complete as no discriminative rich subgraphs are missed due to the translation and reverse translation processes. Due to space limitation, we move the proof along with its supporting lemmas and the low level details of the algorithm in an accompanying technical report [1].

*Theorem 1 (***Sound & Complete***):* All reverse translated subgraphs mined from $TL(DB_{rich})$ are discriminative. Also, all discriminative rich subgraphs could be mined from

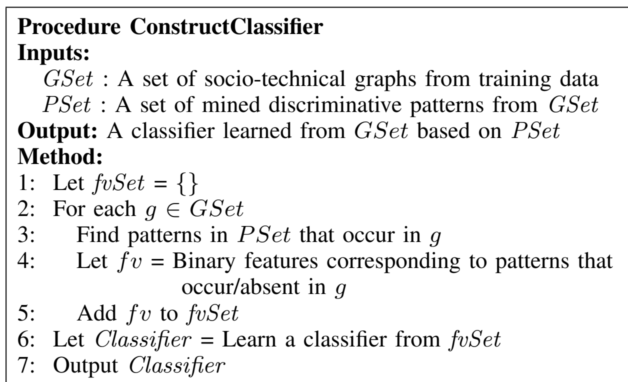
$TL(DB_{rich})$. In other words, the set of discriminative rich subgraphs is the set:

$$RTL(DGM(TL(DB_{rich})))$$

Next, let's analyze the size of the resultant translated graphs. Consider an arbitrary node n. Let e be the edge with the most labels in n. For this n, at most $n.Labels \times e.Labels$ new nodes are introduced. Thus, the number of nodes in the new translated graph grows linearly to the maximum number of labels per node and the maximum number of labels per edge.

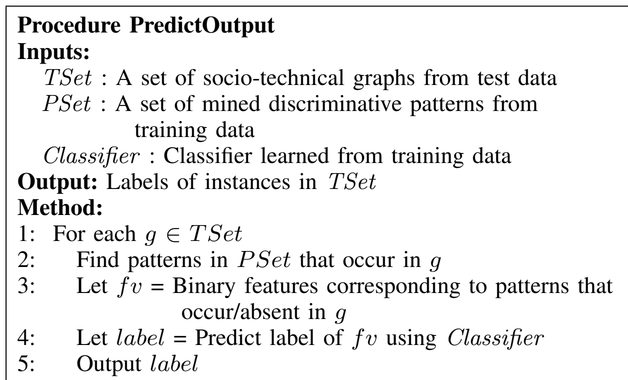
VI. Classifier Construction And Outcome Prediction

We consider the existence or absence of the discriminative patterns mined from the training dataset as binary features. Each of the training and testing data point (which is a software project), represented by its corresponding socio-technical graph, is then mapped to a set of binary features corresponding to the existence or absence of the patterns. We use LibSVM [4] to learn our classifier based on training data. This classifier is then used to assign labels to the test instances, in effect, classifying the project as successful or failed. We show the pseudocodes of classifier construction and outcome prediction in Figures 8 & 9.



Procedure ConstructClassifier
Inputs:
$GSet$: A set of socio-technical graphs from training data
$PSet$: A set of mined discriminative patterns from $GSet$
Output: A classifier learned from $GSet$ based on $PSet$
Method:
1: Let *fvSet* = {}
2: For each $g \in GSet$
3: Find patterns in $PSet$ that occur in g
4: Let fv = Binary features corresponding to patterns that occur/absent in g
5: Add fv to *fvSet*
6: Let *Classifier* = Learn a classifier from *fvSet*
7: Output *Classifier*

Figure 8. Classifier Construction



Procedure PredictOutput
Inputs:
$TSet$: A set of socio-technical graphs from test data
$PSet$: A set of mined discriminative patterns from training data
Classifier : Classifier learned from training data
Output: Labels of instances in $TSet$
Method:
1: For each $g \in TSet$
2: Find patterns in $PSet$ that occur in g
3: Let fv = Binary features corresponding to patterns that occur/absent in g
4: Let $label$ = Predict label of fv using *Classifier*
5: Output $label$

Figure 9. Output Prediction

VII. Experiments

In this section, we describe our experimental settings, followed by our results.

A. Experimental Settings

We analyze SourceForge.Net, the largest open source software development portal. In particular, we use the database dumps of SourceForge.Net collected by Madey *et al.* [2]. From February 2005 onwards, Madey *et al.* collect SourceForge.Net database dumps monthly. We take 64 snapshots which are the dumps for the period starting from February 2005 until May 2010. Each snapshot has many tables and we focus on those containing the information on the various projects hosted in SourceForge.Net and the developers that work on those projects.

We initiate our experiment by extracting projects that have at least one developer from May 2010 snapshot. There are in total 227,922 projects with 289,316 registered developers. We divide the 227,922 projects into three groups: successful projects, failed projects, and others. We use the number of downloads to categorize projects. Projects with more than 100,000 downloads are considered successful. Projects with less than 100 downloads are considered failed. Projects with number of downloads between 100 and 100,000 are considered as belonging to the others group[6]. Unfortunately, SourceForge.net database dumps do not contain any accurate information on the number of downloads. Thus, we crawl the SourceForge.Net website to obtain the download history of each project. We find 2,448 projects (1.07% of all May 2010 projects) are categorized as successful, and 140,796 projects (61.77%) are categorized as failed.

We filter out projects with only one developer. From the 2,448 successful projects, 1,859 projects (75.94% of the successful projects) have more than one developer. Moreover from the 140,796 failed projects, 28,802 projects (20.46% of failed projects) have more than one developer. We also filter projects that exist on February 2005 as for those projects we could not ascertain the time the contributors join the project[7]. After we exclude those projects, we have 224 successful projects and 3,826 failed projects.

From these projects, we extract the socio-technical graphs along with the 6 features described in Section IV. To extract the features, we need to first determine the time when a developer joins a project. This information is not directly available from the dump. Fortunately, we have the monthly snapshots and by contrasting the reported developers in two consecutive months, we could find the *month* when a developer joins a project. For consistency, for all features, we use month granularity.

Four features: Past Successful Projects (PSP), Past Failed Projects (PFP), Past Successful Collaborations (PSC),

[6] We exclude this group from our analysis.
[7] This information is not recorded in SourceForge.Net dumps.

and Past Failed Collaborations (PFC), are obtained by analyzing the monthly dumps one by one. We need to compare the month when a developer or a pair of developers joins a past project with the month when the developer or the pair of developers joins the current project. If a past project under comparison is either successful or failed, the counts of the corresponding features among the four are updated.

Following our definition of Length of Membership (LOM), we count the period of time that has passed since a developer first registered in SourceForge until he joins the current project. SourceForge database dumps provide the time when a developer becomes a member of SourceForge. To compute the feature: Length of Collaboration History (LCH), we analyze the monthly dumps in chronological order and for relevant pairs of developers, we find the month that they first work together in a single project.

At the end of the above process, we have 224 socio-technical graphs corresponding to successful projects and 3,826 graphs corresponding to failed ones. These socio-technical graphs are rich graphs with multiple labels in the nodes and edges. The 224 successful project graphs have an average size of 3.76 nodes and 5.92 edges. The maximum number of nodes and edges for the successful project graphs are 32 nodes and 195 edges respectively. The 3,826 failed project graphs have an average size of 2.86 nodes and 3.98 edges. The maximum number of nodes and edges for the failed project graphs are 73 nodes and 1,081 edges respectively. We process our graph dataset following the procedure in Section V and mine rich subgraph patterns that differentiate successful and failed projects.

We run our translation algorithm on an Intel(R) Xeon(R) 3.17GHz server with 24 GB of RAM running 64-bit Windows Server Standard Edition, Service Pack 2. The algorithm is written in Visual C#.Net. The top-K discriminative graph mining algorithm (Top-K LEAP) [5] is written in C++ and is run on a Dell PowerEdge R900 server with 2.67GHz six-core CPU and 8GB main memory running Linux RedHat.

B. Experimental Results

We first describe the runtime of our approach and the sizes of the translated graphs. Next, we show the effectiveness of the mined patterns in predicting project outcome. Finally, we present our mined patterns.

Runtime & Translated Graph Size. The translation process for successful projects translates 224 rich graphs in 1.4 s. For the failed projects, the translation process translates 3,826 rich graphs in 9.7 s. The 224 resultant translated simple graphs for the successful projects have an average size of 31.54 nodes and 287.25 edges. The 3,826 resultant translated simple graphs for the failed projects have an average size of 23.93 nodes and 204.68 edges. In terms of the number of nodes, after translation, the graphs grow by 8.37-8.39 times, which is less than the number of node labels multiplied by the number of edge labels (i.e., $3 \times 3 = 9$). This is in line with our analysis presented in Section V-B3.

We run Top-K LEAP by Cheng *et al.* [5] on the translated simple graphs. It completes within 4 hours to mine 20 most discriminative graph patterns.

Project Outcome Prediction. We measure the effectiveness of our approach by classification accuracy and area under the ROC curve. Classification accuracy, defined as the percentage of projects correctly classified, is used as one measure. Due to the skewed class distribution, the measure AUC which is the area under a ROC curve is also used. ROC curve shows the trade-off between true positive rate and false positive rate for a given classifier [7]. A good classifier would produce a ROC curve as close to the top-left corner as possible. AUC is a measure of the model accuracy, in the range of $[0, 1.0]$. The best possible classifier would generate an optimal AUC value of 1.0.

We perform 10-fold cross validation, where for each we keep 1/10 of the data for testing and the other for training. Under this setting, we are able to predict the labels of the projects with 94.99% accuracy and 0.86 AUC thus demonstrating the effectiveness of our proposed approach.

Most Discriminative Patterns. Figure 10 shows the top-20 most discriminative patterns sorted in a descending order of their discriminative scores. We also show the percentages of successful and failed projects exhibiting the patterns.

We could note the following from the mined patterns:

1) The most discriminative pattern is the pattern P1: 92.42% of the failed projects follow this pattern, whereas only 26.34% of the successful projects follow it. The pattern describes a collaboration including two contributors where none of them has a successful project before. The two contributors have also no past history of successful collaborations. The two developers are likely to be inexperienced and there is little positive socio-technical links between the two before the developers join the current project. Thus, from the dataset, empirically we observe that the lack of experience and the weak socio-technical ties between co-contributors seem to be some factors related to project failure.
2) More than 40% of the successful projects follow patterns P2 - P19 (18 patterns) whereas no more than 9% of the failed projects follow them. The patterns have one thing in common: there is at least 1 developer with 1 successful past project (PSP) or there is at least 1 successful collaboration in the past (PSC). Fourteen of the patterns (77.78%) specify that there is at least 1 developer with 1 past successful project (patterns P4 - P5, P7, and P9 - 19). Moreover, 12 of the 18 patterns (66.67%) show that there is at

#	Pattern	Successful Projects (%)	Failed Projects (%)	Score
1	(0.x.x) —0,x,x— (0,x,x)	26.34	92.42	0.0952
2	(x.0.x) —1,0,x— (x,0,x)	63.39	4.34	0.0923
3	(x.0.x) —1,x,x— (x.0.x)	63.39	4.34	0.0923
4	(1,x,x) —1.0.x— (x,0,x)	65.63	5.12	0.0917
5	(1.x.x) —x.0.x— (x.0.x)	67.41	5.80	0.0912
6	(1.0.x)	66.52	5.70	0.0898
7	(1.x.x) —1.x.x— (x.0.x)	66.07	5.59	0.0894
8	(1,x,x)	70.98	8.05	0.0861
9	(1.0.x) —1,0,x— (x.0.x)	60.71	4.34	0.0859
10	(1,0,x) —1,x,x— (x,0,x)	60.71	4.34	0.0859
11	(1.0.x) —x,0,x— (x,0,x)	62.05	4.78	0.0857
12	(1.x.x) —1,0,x— (x,0,x) —x,0,x— (x.0.x)	60.71	4.40	0.0855
13	(1,x,x) —1,x,x— (1,x,x) —x,0,x— (x,0,x)	48.66	4.57	0.0579
14	(1.x.x) —1.x.x— (1.x.x)	51.34	5.54	0.0576
15	(1,x,x) —1.0.x— (1,x,x)	48.21	4.52	0.0573
16	(1,x,x) —x,0,x— (1,x,x)	48.21	4.52	0.0573
17	(1,0,x) —1,0,x— (1,x,x)	47.77	4.47	0.0568
18	(1,0,x) —x.0.x— (1,x,x)	47.77	4.47	0.0568
19	(1,0,x) —1.x.x— (1.x.x)	47.77	4.52	0.0564
20	(0.x.x)	53.57	95.22	0.0523

(a.b.c) a = Number of Past Successful Projects (PSP)
b = Number of Past Failed Projects (PFP)
c = Length of Membership (LOM)

d,e,f d = Number of Past Successful Collaboration (PSC)
e = Number of Past Failed Collaboration (PFC)
f = Length of Collaboration History (LCH)

x denotes "don't care value", the feature could take any value

Figure 10. Top-20 Discriminative Patterns Mined

least 1 successful collaboration between the developers (patterns P2 - P4, P7, P9 - 10, P12 - P15, P17, and P19). This emphasizes that if developers have already had past successful collaborations, then the current collaboration would more likely be successful too.

3) The pattern with the highest proportion of successful projects exhibiting it is P8. It simply states that there is a contributor with one past successful project. For this pattern, the proportion of successful projects following it is almost 9 times more than that of failed projects. It highlights that a contributor's past positive experience is related with the likelihood of project success.
4) On the other hand, the pattern with the highest proportion of failed projects exhibiting it is P20. It is also a single node pattern. Opposite to P8, P20 represents a scenario where there is a developer with zero past successful projects. This seems to show that having one inexperienced member could lower the chance of project success. The decrement is not very much though as the difference between the proportion of successful projects exhibiting it is not very far from that of failed projects.
5) There are 2 patterns, i.e., pattern P12 and P13, that show collaborations between 3 developers. More than 60% of the successful projects follow pattern P12 and almost 50% of the successful projects follow pattern P13. However no more than 5% of failed project follow any of the two patterns. From these two 3-developer patterns, there are several things in common: 1) there is at least 1 developer with 1 past successful project, 2) there exists 1 past successful collaboration between two contributors, and 3) there is at least 1 developer with zero number of unsuccessful projects. Also, note that pattern P12 extends pattern P4 with one node. It extends P4 by adding the rightmost node. Adding a developer with no past experience of failed project seems to only slightly increase the likelihood of project success ($\frac{65.63}{5.12} \rightarrow \frac{60.71}{4.40}$ = 12.82 $\rightarrow$ 13.78).
6) It is interesting to note that none of top 20 most discriminative patterns include a concrete value for LOM (length of membership) and LCH (length of collaboration history). This result could be due to the fact that there is a large spread of possible values for LOM and LCH or that LOM and LCH have less influence on project success or failure.

Threats to Validity. In this study we have only analyzed projects in SourceForge.Net. In the future, we could reduce this threat of external validity by also analyzing projects in GitHub. In this study, we measure the success of a project by the number of downloads; other measures of success could also be employed. We also assume that a developer is identified by a unique user name and contributes to a project if he/she is listed in SourceForge.Net as a contributor.

VIII. Related Work

There has been a number of work in software engineering that leverages social network among developers. One of

the early work is the work by Bird *et al.* that extracts a social network from developer email communication [3]. They find that the level of email activity strongly correlates with the level of activity in the source code. Many recent studies also show the power of analyzing social network to predict for failures [15], [11]. This work complements past studies by recovering discriminative developer collaboration patterns that differentiate successful and failed projects in a large super-repository of open-source projects, i.e., SourceForge.Net, and utilizing them to predict project outcome.

Lungu *et al.* propose an approach to visualize a super-repository [9]. A related visualization study is also performed by Sarma *et al.* [13]. Surian *et al.* mine for frequent patterns of collaborations [14]. We extend their study by mining for discriminative patterns from a super-repository containing thousands of diversified projects. Furthermore, we also extend the study by investigating multiple snapshots of SourceForge.Net instead of only a single snapshot.

Madey *et al.* show the power law relationship between the number of nodes in collaboration clusters and their frequency in a snapshot of SourceForge.Net [10]. Xu *et al.* investigate the small-world phenomenon on a snapshot of SourceForge.Net [16]. Verner *et al.* interview software developers in various countries and identify factors that relate with project success. Cheng *et al.* [5] extend the approach by Yan *et al.* [17] to mine for top-k most discriminative graph patterns. Procaccino *et al.* perform a survey to find how developers view project success [12]. In this work, we extend the above studies to mine for discriminative patterns from *rich* rather than *simple* graphs. We also consider a new problem domain by mining discriminative graphs to find socio-technical collaboration features that relate to and could be leveraged to predict project outcome.

IX. Conclusion

In this work, we perform a longitudinal study of projects in SourceForge.Net and detect discriminative graph patterns from developer and their collaboration history that differentiate successful from failed projects. These socio-technical patterns form effective features that could be leveraged to predict project outcome. We take 64 snapshots of SourceForge collected on the period from February 2005 to May 2010. We extract some socio-technical features from the 64 snapshots and model them in the form of graphs. Each project is mapped to a graph which is labeled as either successful or failed depending on the number of downloads. We propose a new graph mining problem of extracting the top-k most discriminative *rich* graph patterns from a graph database. To address this problem we built upon an existing top-k most discriminative *simple* graph mining algorithm by proposing a novel translation strategy. We show that the translation could conserve some properties that allow for all discriminative patterns to be mined. We apply our mining solution on the socio-technical graphs from SourceForge. We show that these patterns could classify project outcome with an accuracy of over 90% and an AUC score of 0.86. Additionally, we present and analyze the top-20 most discriminative patterns that we mine. As a future work, we plan to consider industrial datasets in addition to SourceForge.Net data.

Acknowledgement. This research is supported by the Singapore National Research Foundation under its International Research Centre @ Singapore Funding Initiative and administered by the IDM Programme Office, and the Hong Kong Research Grants Council (RGC) General Research Fund (GRF) Project No. CUHK 411310. We would like to thank Greg Madey for sharing with us the SourceForge.Net dataset. This work was done while the first author was with School of Information Systems, Singapore Management University.

References

[1] "Mining rich graphs: A graph transformation approach," http://www.mysmu.edu/faculty/davidlo/papers/richgraphtr.pdf.

[2] M. Antwerp and G. Madey, "Advancess in the sourceforge research data archive (SRDA)," in *OSS*, 2008.

[3] C. Bird, A. Gourley, P. Devanbu, M. Gertz, and A. Swaminathan, "Mining email social networks," in *MSR*, 2006.

[4] C.-C. Chang and C.-J. Lin, *LIBSVM: a library for support vector machines*, 2001, software available at http://www.csie.ntu.edu.tw/∼/cjlin/libsvm.

[5] H. Cheng, D. Lo, Y. Zhou, X. Wang, and X. Yan, "Identifying bug signatures using discriminative graph mining," in *ISSTA*, 2009, pp. 141–152.

[6] M. Goeminne and T. Mens, "A comparison of identity merge algorithms for software repositories," in *Science of Computer Programming*, 2012.

[7] J. Han and M. Kamber, *Data Mining: Concepts and Techniques* (2nd ed.). Morgan Kaufmann, 2006.

[8] E. Kouters, B. Vasilescu, A. Serebrenik, and M. V. D. Brand, "Who's who in gnome: using lsa to merge software repository identities," in *ICSM*, 2012.

[9] M. Lungu, M. Lanza, T. Girba, and R. Heeck, "Reverse engineering super-repositories," in *WCRE*, 2007.

[10] G. Madey, V. Freeh, and R. Tynan, "The open source software development phenomenon: An analysis based on social network theory," in *AMCIS*, 2002.

[11] M. Pinzger, N. Nagappan, and B. Murphy, "Can developer social networks predict failures?" in *FSE*, 2008.

[12] J. Procaccino and J. Verner, "Software developers' views of end-users and project success," *Commun. ACM*, vol. 52, pp. 113–116, 2009.

[13] A. Sarma, L. Maccherone, P. Wagstrom, and J. Herbsleb, "Tesseract: Interactive visual exploration of socio-technical relationships in software development," in *ICSE*, 2009.

[14] D. Surian, D. Lo, and E.-P. Lim, "Mining collaboration patterns from a large developer network," in *WCRE*, 2010.

[15] T. Wolf, A. Schroter, D. Damian, and T. Nguyen, "Predicting build failures using social network analysis on developer communication," in *ICSE*, 2009.

[16] J. Xu, Y. Gao, S. Christley, and G. Madey, "A topological analysis of the open source software development community," in *HICSS*, 2005.

[17] X. Yan, H. Cheng, J. Han, and P. S. Yu, "Mining significant graph patterns by scalable leap search," in *SIGMOD*, 2008, pp. 433–444.