Predicting Delays in Software Projects Using Networked Classification

Morakot Choetkiertikul*, Hoa Khanh Dam*, Truyen Tran† and Aditya Ghose*
*School of Computing and Information Technology
University of Wollongong, Australia
Email: {mc650,hoa,aditya}@uow.edu.au
†School of Information Technology
Deakin University, Australia
Email: truyen.tran@deakin.edu.au

***Abstract*—Software projects have a high risk of cost and schedule overruns, which has been a source of concern for the software engineering community for a long time. One of the challenges in software project management is to make reliable prediction of delays in the context of constant and rapid changes inherent in software projects. This paper presents a novel approach to providing automated support for project managers and other decision makers in predicting whether a subset of software tasks (among the hundreds to thousands of ongoing tasks) in a software project have a risk of being delayed. Our approach makes use of not only features specific to individual software tasks (i.e. local data) – as done in previous work – but also their relationships (i.e. networked data). In addition, using *collective* classification, our approach can *simultaneously* predict the degree of delay for a group of related tasks. Our evaluation results show a significant improvement over traditional approaches which perform classification on each task *independently*: achieving 46%–97% precision (49% improved), 46%–97% recall (28% improved), 56%–75% F-measure (39% improved), and 78%–95% Area Under the ROC Curve (16% improved).**

I. Introduction

Delays constitute a major problem in software projects [1]. Approximately one-third of IT projects went over the scheduled time, according to a recent study by Mckinsey and the University of Oxford in 2012 on 5,400 large scale IT projects [2]. In the Standish Group's well-known CHAOS report [3], the proportion of delayed projects recorded were even higher – at 82%. These studies have also shown that ineffective risk management is one of the main reasons for the high rate of overrun software projects. An important aspect of risk management is the ability to predict, at any given stage in a project, which tasks (among hundreds to thousands tasks) are at risk of being delayed. Foreseeing such risks allows project managers to take prudent measures to assess and manage the risks, and consequently reduce the chance of their project being delayed.

Making a reliable prediction of delays is therefore an important capability for project managers, especially when facing with the inherent dynamic nature of software projects (e.g. constant changes to software requirements). Current practices in software risk management however rely mostly on high-level, generic guidance (e.g. Boehm's "top 10 list of software risk items" [4] or SEI's risk management framework [5]) or highly subjective expert judgements. Thus, there is a strong need for providing automated, contextual support for identifying risks of delay in software projects.

In order to address that need, a number of recent proposals have leveraged data mining and machine learning technologies to predict delays in resolving software issues (e.g. [6]) or to estimate the fix time of a software bug (e.g. [7–11]). In line with a large body of work in data mining for software engineering, these approaches employ traditional machine learning classification techniques to perform classification on each issue or bug *independently* using its attributes or features. Such approaches do *not* take into account the role of the underlying network of inter-relationships between the tasks of resolving those bugs or issues. This is a gap, given the preponderance of task dependencies in software projects – approximately 57% of 11,851 tasks in the five open source projects selected for our study were related to at least one other task. These task dependencies form the *networked data* that we will seek to leverage in this work.

Networked data are seen in many different forms in our daily life, such as hyperlinked Web pages, social networks, communication networks, biological networks and financial transaction networks. They are used in various applications such as classifying Web pages [12], scientific research papers [13, 14], protein interaction and gene expression data [15]. We demonstrate that a similar class of networked data (i.e. networked tasks) can also provide valuable information for predicting delays in software projects. For example, if a task blocks another task and the former is delayed, then the latter is also at risk of getting delayed. This example demonstrates a common *delay propagation* phenomenon in (software) projects, which has not been considered by previous approaches.

We propose a novel approach to leverage task dependencies for predicting delays in software projects. This paper makes two main contributions:

- **A technique for constructing a task network of software development tasks**
  This technique enabled us to extract various relationships between tasks in software projects to build a task network. Task relationships can be explicit (those that are explicitly specified in the task records) or implicit (those that need to be inferred from other task information). Explicit relations usually determine the

order of tasks, while implicit relations reflect other aspects such as tasks assigned to the same developer, tasks affecting the same software component or similar tasks.

- **Predictive models to predict delays in software projects.**
  We developed accurate predictive models that can predict whether a software task is at risk of getting delayed. Our predictive models have three components: *local classifier*, *relational classifier* and *collective inference*. The local classifier uses *15* non-relational (i.e. local) features of a software task: discussion time, waiting time, task type, number of repetition tasks, percentage of delayed tasks that a developer involved with, developer's workload, task priority, number of comments, changing of priority, number of fix version, number of affect version, changing of description, number of votes, number of watches and reporter reputation. The relational classifier makes use of the task relations in a task network to predict if a task gets delayed based on the delay information of its neighbors. Finally, collective inference allows us to make such a prediction *simultaneously* for multiple related tasks. The performance of our predictive models were evaluated on five different open source projects. We achieved 46%-97% precision, 46%-97% recall, 56%-75% F-measure, and 78%-95% Area Under the ROC Curve. The evaluation results compared to the traditional approaches using only local classifiers show a significant improvement: it improves 49% precision, 28% recall, 39% F-measure, and 16% Area Under the ROC Curve over the traditional approaches.

The remainder of this paper is organized as follows. Section II describes a motivating example and Section III provides an overview of our approach. Section IV serves to describe how a task network is built for a software project. Section V presents our networked predictive models. Section VI reports the experimental evaluations of our approach. Threats to validity of our study are discussed in Section VII. Related work is discussed in Section VIII before we conclude and outline future work in Section IX.

II. Motivating example

Typically, a (software) project requires a number of activities or tasks be completed. Each task usually has an estimated due date. It is important for project managers to ensure as many tasks be completed in time (i.e. by their respective due date) as possible since this has implications to the overall progress of a project (e.g. releasing a major version on a specified date). However, in practice project will never execute exactly as it was planned due to uncertainty. For example, Figure 1 shows seven tasks (represented by their ID) extracted from the JBoss[1] project, only two of which (i.e. tasks JBIDE-1469 and JBAS-14) were completed on time whilst there were three delayed tasks (i.e. JBAS-15, JBWS-52 and JBOP-1). One of the main challenges in project management is therefore predicting which tasks have a risk of being delayed, giving the current situation of a project, in order to come up with measures to reduce or mitigate such a risk. In this example, assume that we are trying to predict if tasks JBAS-13 and JBAS-7 will be delayed.

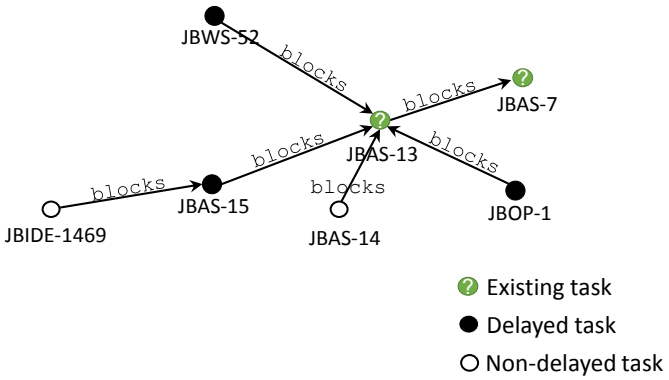

Fig. 1: An example of task dependencies in the JBoss project

Recent work [6] has proposed 16 risk factors contributing to the delay of a software task completion. Those factors reflect a range of attributes associated with a software task (which are referred to as *local features*) such as task type, task priority, the workload of developers assigned to the task, and so on. Based on these risk factors, predict models were built to predict if a task will be delayed. Like many tradition machine learning methods in software engineering, the work in [6] has treated tasks as being *independent*, which makes it possible to predict delay risks on a task-by-task basis. For example, using the 16 local features, the approach in [6] predicted that tasks JBAS-13 would not be delayed.

In most cases, the tasks in a project are however related to each other, and the delay status (e.g. major delayed, minor delayed or non-delayed) of one task may have an influence on that of its related task. For example, there are several "blocking" relationships between the seven JBoss tasks in Figure 1, e.g. task JBIDE-1469 blocks JBAS-15, indicating that the former needs to be finished before the completion of the latter. The task dependencies form the *networked data* which contain interconnected tasks. Networked data provides additional, valuable information which can be used to improve the predictive performance of techniques solely relying local features. For example, the local features are not sufficiently to provide accurate prediction for task JBAS-13 – it was predicted as non-delayed but it is in fact a delayed task. On the other hand, by examining its relationships with other tasks whose delay status are known – JBAS-13 were blocked by 3 delayed tasks and 1 non-delayed task (see Figure 1) – we may be able to *infer* the risk of task JBAS-13 being delayed.

This example motivates the use of networked data to make *within-network estimation*: tasks for which delay status (e.g. delayed or non-delayed) is known are linked to tasks for which the delay status must be predicted. Here, the data has an important characteristics: tasks with known delay status are useful in two aspects. They serves not only as training data but also as *background knowledge* for the inference process. Hence, the traditional separation of data into training and test sets need to carefully take this important property into account. In addition, networked tasks support *collective classification*, i.e. the delay status of various related tasks can be predicted

[1] http://www.jboss.org

simultaneously. For example, the prediction of task JBAS-13 can be used to influence the estimation of task JBAS-7 as they are linked, thus we should do both predictions at the same time.

This example only demonstrates links that are explicitly specified in the tasks' record. In practice, there are many instances where tasks are related implicitly, e.g. assigned to the same software engineers, affecting the same software component, or similar to each other. Our approach is able to extract different types of explicit and implicit task relationships, and uses them for delay prediction.

III. Overview of our approach

Our approach leverages classification techniques in machine learning to predict the riskiness of a task being delayed. A given task is classified into one of the classes in $\{c_1, c_2, ..., c_k\}$ where each class c_i represents the risk impact in terms of the degree of delay, e.g. major delayed, minor delayed or non-delayed. Historical (i.e. completed) tasks are labeled, i.e. assigned to a class membership, based on examining the difference between their actual completion date and due date. For example, in our studies tasks completed by their due date are labeled as "non-delayed", whilst tasks finished more than 60 days after their due date are labeled as "major delayed".

The basic process of our approach is described in Figure 2. The process has two main phases: the learning phase and the execution phase. The learning phase involves using historical data from past tasks to train classifiers, which are then used for classify new tasks in the execution phase.

Our approach extracts data associated with software tasks to build a task network which is defined as below.

Definition 1 (Task network)**.** *A task network is a directed graph G = (V, E) where:*

- *each vertex $v \in V$ representing a software task in the form of $\langle ID, c, attrs \rangle$ where ID is a unique identifier of the task, c is the risk class, i.e. label (e.g. non-delayed, minor delayed or major delayed), which the task belongs to, and $attrs$ is a set of the task's attribute-value pairs $(attr_i, val_i)$ (i.e. local features).*
- *each edge $e \in E$ representing a link between tasks u and v in the form of $\langle \langle u, v \rangle, types, weights \rangle$ where $types$ is set of the link's type and $weigths$ is set of link's weight.*

The set of tasks (or nodes) V in a task network is further divided into two disjoint groups: tasks with known class labels, V^K, and tasks whose labels need to be estimated (unknown class), V^U which $V^U = V \setminus V^K$. Labeled tasks are used for training, and also serve as background knowledge for inferring the label of tasks in V^U.

A set of attributes ($attrs$) for a task are also extracted (see Table I). These features represents the local information of each individual task in the network. The local features are used to build a *local classifier* which treats tasks independently from each other. Traditional state-of-the-art classifiers (e.g. Random Forest [16], Multiclass Support Vector Machines [17], or Multiclass Logistic Regression [18]) can be employed for this purpose.

TABLE I: Local features of a software task

Feature	Short description
Discussion time	The period that a team spends on finding solutions to solve a task
Waiting time	The time when a task is waiting for being acted upon
Type	Task type
Task repetition	The number of times that an task is reopened
Priority	Task priority
Changing of priority	The number of times a task's priority was changed
No. comments	The number of comments from developers during the discussion time
No. fix versions	The number of versions for which a task was or will be fixed
No. affect versions	The number of versions for which a task has been found
Changing of description	The number of times in which the task description was changed
Reporter reputation	The measurement of the reporter reputation
Developer's workload	The number of opened tasks that have been assigned to a developer at a time
Per. of delayed tasks	The percentage of delayed tasks in all of the tasks which have been assigned to a developer
Number of votes	The number of developers who voted a task
Number of watches	The number of developers who watch a task

The second important component in our approach is the *relational classifier*. Unlike the local classifier, the relational classifier makes use of the relations between tasks in the network (represented by edges) to estimate a task's label using the labels of its neighbors. Relational classifier models exploit a phenomenon that is widely seen in relational data: the label of a node is influenced by the labels of its related nodes. Relational classifier models may also use local attributes of the tasks. Links between tasks in the network are established by extracting both explicit and implicit relations. Explicit relations refer to the task dependencies explicitly set by the developers (e.g. the block dependency). On the other hand, implicit relations can be inferred from the resources assigned to the tasks (e.g. assigned to the same developer) or the nature of the tasks. We will discuss these types of task relations in details in the next section. Each type of relationship can be assigned to a weight which quantitatively reflects the strength of the relationship.

Another novel aspect of our approach is the *collective inference* component which *simultaneously* classifies a set of related tasks. Details of these approaches will be provided in Section V.

IV. Task network construction

An important part of our approach is building a task network for past and current tasks. In most of modern task tracking system (e.g. JIRA), some dependencies between tasks are explicit recorded (i.e. in a special field) in the task reports and can be easily extracted from there. We refer to these dependencies as *explicit relationships*. There are however other types of task dependency that are not explicitly recorded (e.g. tasks assigned to the same developer), and we need to infer them from extracting other information of the tasks. These

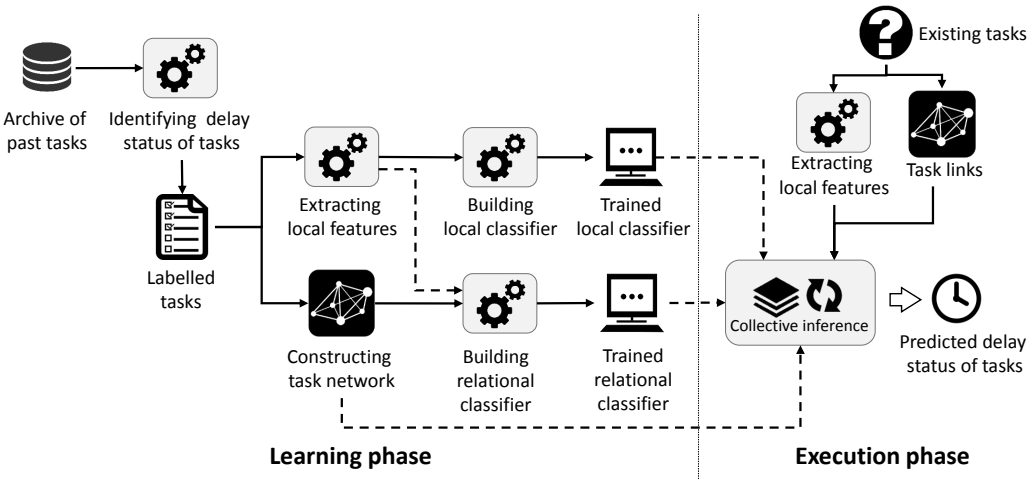

Fig. 2: An overview of our approach

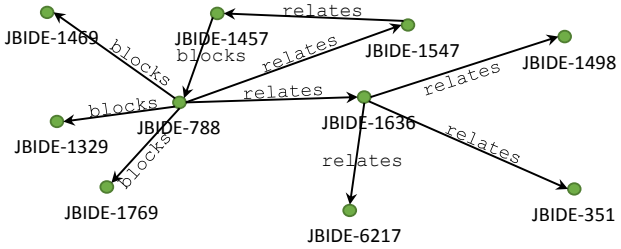

Fig. 3: Example of explicit task relationships in JBoss

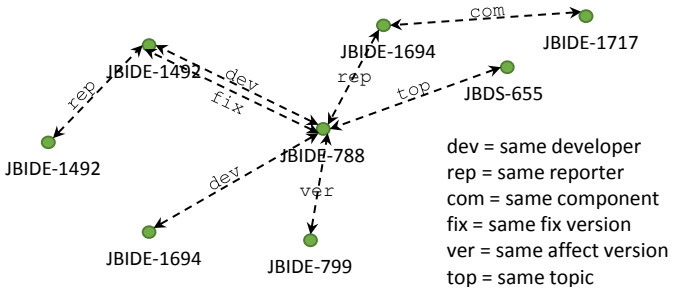

Fig. 4: Example of implicit task relationships in JBoss

are referred to as *implicit relationships*. We now discuss these types of relationships in details.

1) ***Explicit relationships****:* There are a number of dependencies among tasks which are explicitly specified in the task records. These typically determine the order in which tasks need to be performed. There are generally four different types of relationships of the preceding tasks to the succeeding tasks: finish to start (predecessor must finish before successor can start), start to start (predecessor must start before successor can start), finish to finish (predecessor must finish before successor can finish), and start to finish (predecessor must start before successor can finish). For example, blocking is a common type of relationships that is explicitly recorded in issue/bug tracking systems. Blocking tasks are software tasks that prevent other tasks from being resolved, which could fall into the finish to start or finish to finish category.

Figure 3 shows some explicit relationships between tasks in the JBoss project, which uses the JIRA task tracking system. JIRA provides the task link attribute to specify the relationship between two or more related tasks. The explicit relationships are extracted directly from the dataset. For example, JBIDE-788 blocks JBIDE-1469, which is represented by a directed edge connected the two nodes. In addition to blocking, JIRA also provides three other default types of task links: relates to, clones and duplicates. Figure 3 shows some examples of the "relates to" relationship, e.g. task JBIDE-788 relates to JBIDE-1547.

2) ***Implicit relationships****:* While explicit relationships are specified directly in the task reports, implicit relationship need to be inferred from other task information. There are different task information that can be extracted to identify a (implicit) relationship between tasks. We classified them into three groups as described below.

- *Resource-based relationship:* this type of relationships exists between tasks that share the same (human) resource. The resource here could be the developers assigned to perform the tasks or the same person who created and reported the tasks. Resource-based relationship is important in our context since a resource's skills, experience, reputation and workload may affect a chance of delayed tasks (i.e. a developer who causes a delay of a current task may do so again in the future). For example, from Figure 4, JBIDE-788 has a relationship with JBIDE-1694 since both of them are assigned to the same developer. Task JBIDE-788 is also related to JBIDE-1694 since they were reported by the same person.
- *Attribute-based relationship:* tasks can be related if some of their attributes share the same values. For example, there is a relationship between tasks performed on the same component since they may affect the same or related parts of code. For tasks recorded in JIRA, we extract this type of relationship by examining three

TABLE II: Datasets and networks' statistics

Project	Relationship	Num Nodes	Num Edges	Avg. node degree	Node Assort.
Apache	Explicit	496	246	1.597	0.256
	Implicit	496	27,460	55.362	0.246
	All	496	27,706	55.858	0.225
Duraspace	Explicit	1,116	563	1.700	0.257
	Implicit	1,116	383,677	343.796	0.240
	All	1,116	384,240	344.301	0.230
JBoss	Explicit	8,206	4,904	2.057	0.235
	Implicit	8,206	4,908,164	598.118	0.249
	All	8,206	4,913,068	598.716	0.247
Moodle	Explicit	1,439	1,283	3.055	0.222
	Implicit	1,439	197,176	137.022	0.215
	All	1,439	198,748	138.115	0.208
Spring	Explicit	597	222	1.219	0.250
	Implicit	597	63,430	106.247	0.249
	All	597	63,652	106.619	0.242

attributes: *affect version*, *fix version* and *component*. For example, JBIDE-788 and JBIDE-799 affects the same version while JBIDE-1694 and JBIDE-1717 affects the same component as shown in Figure 4.

- *Content-based relationship:* tasks can be similar in terms of how they are conducted and/or what they affect. The similarity may form an implicit relationship between tasks which can be established by extracting the description of the tasks. Different extraction techniques can be applied here, ranging from traditional information retrieval techniques to recent NLP techniques like topic modeling. We use Latent Dirichlet Allocation [19] to build a topic model representing the content of a software task. We then establish relationships between on the basis that related tasks share a significant number of common topics. Figure 4 shows some example of content-based relationships in JBoss, e.g. task JBIDE-788 has the same topic with JBDS-655. The common topics shared between these two tasks are "code, access control exception, and document types".

A task network is built by extracting both explicit and implicit links among the tasks. We employ a number of measures to describe different properties of a task network: the number of nodes, the number of edges, and the average node degree (i.e. the number of connections a node has to other nodes). In addition, assortativity coefficient [20] is used to measure the correlation between two nodes: the preference of network nodes to connect to other nodes that have similar or different degrees. Positive values of assortativity coefficient indicate a correlation between nodes of similar degree (e.g. highly connected nodes tends to be connected with other high degree nodes), while negative values indicate relationships between nodes of different degree (e.g. high degree nodes tend to connect to low degree nodes). As can be seen from Table II, the inclusion of implicit relationships significantly increases the density of the network tasks across all the five projects that we studied. By contrast, the assortativity coefficient remains nearly the same with or without implicit relationships.

A weight is also applied to each edge type in a task network. This allows us to better quantify the strength of a relationship between tasks. By default, each edge is equally assigned the weight of 1. However, different weights can also be applied to different types of relationships. More complex approaches can also be applied here. For example, the weights could be decreased over time to reflect the fading of the relationships, e.g. the tasks have been assigned to the same developer for long time ago.

V. Predictive Models

Our predictive models are built upon three components: local classifier (as done in previous work), relational classifier, and collective inference. Local classifiers treat tasks as being independent, making it possible to estimate class membership on a task-by-task basis. Relational classifiers posit that the class membership of one task may have an influence on the class membership of a related task in the network. Collective inference infers the class membership of all tasks simultaneously [21]. In the following we discuss the details of each components.

A. Local (non-relational) classifier

There are several available state-of-the-art algorithms and techniques that we could employ to develop local classifiers. We employ the state-of-the-art classifier which is Random Forest (RF) [16] – the best performing technique in our experiments.

B. Relational classifier

Relational classifiers make use of information about related tasks to estimate the label probability. For simplicity, we use only direct relations for class probability estimation:

$$P(c \mid G) = P(c \mid N_i)$$

where N_i is a set of the immediate neighbors of task v_i (i.e. those that are directly related to v_i) in the task network G, such that $P(c \mid N_i)$ is independent of $G \setminus N_i$.

This is based on a theoretical property known as the Markov assumption which states that given the neighborhood (also known as the Markov blanket), it is sufficient to infer about the current label without knowing the other labels in the network [22].

For developing a relational classifier, we employ two highly effective methods. One is Weighted-Vote Relational Neighbor (wvRN) [23] which is one of the best relational classification algorithms reported in [21]. The other is Stacked Graphical Learning [24], where classifiers are built in a stage-wise manner, making use of relational information in the previous stage.

*1) **Weighted-Vote Relational Neighbor**:* Weighted-Vote Relational Neighbor (wvRN) estimates class membership probabilities based on two assumption [25]. First, the label of a node depends only on its immediate neighbors. Second, wvRN relies on the principle of homophily which assumes that neighboring class labels were likely to be the same [26]. Thus,

wvRN estimates $P(c|v_i)$ as the (weighted) mean of the class membership of the tasks in the neighborhood (N_i):

$$P(c \mid v_i) = \frac{1}{Z} \sum_{v_j \in N_i} w(v_i, v_j) P(c \mid N_j)$$

where $Z = \sum_{v_j \in N_i} w(v_i, v_j)$ and $w(v_i, v_j)$ is the weight of the link between task v_i and task v_j. Our experiments applied the same weight of 1 to all relationship types, i.e. $w(v_i, v_j) = 1$. The optimized weights could be determined using the properties of a network topology such as assortativity coefficient [21, 27, 28]). We denote the prior class probability distributions from a relational classification as M_R.

*2) **Stacked Graphical Learning**:* One inherent difficulty of the weighted-voting method is the computation of the neighbor weights. Since there are multiple relations, estimating the weights are non-trivial. Stacked learning offers an alternative way to incorporate relational information.

The idea of stacking is to learn joint models by multiple steps, taking into relational information of the previous step to improve the current step. At each step, relational information together with local features are fed into a standard classifier (e.g., Random Forests). We consider relations separately and the contribution of each relation is learnt by the classifier through the relational features. The classifier is then trained. Its prediction on all data points (vertices in the network) will be then used as features of the next stage. We adapt the idea from [24]. Our contribution is in the novel use of Random Forests as a strong local classifier rather than linear classifiers as used in [24].The stacked learning algorithm is described in Algorithm 1. It returns T classifiers for Tsteps. At the first step, the local classifier is used. At subsequent steps, relational classifiers are trained on both local features and relation-specific averaged neighbor probabilities.

C. Collective inference

Collective inference is the process of inferring class probabilities simultaneously for all unknown labels in the network conditioned on the seen labels. We employ two methods: Relaxation Labeling (RL) [29] and Stacked Inference (SI). RL is applicable to any non-stagewise relational classifiers (e.g. wvRN described in Section V-B1). It has been found to achieve good performance in [21]. SI, on the other hand, is specific to stacked classifiers (e.g., see Section V-B2).

*1) **Relaxation Labeling**:* Relaxation Labeling (RL) has been shown to achieve good results in [21]. RL initializes the class probabilities using the local classifier model. RL then iteratively corrects this initial assignment if the neighboring tasks have labels that are unlikely according to the prior class distribution estimated by M_R (see Section V-B1). Algorithm 2 describes the Relaxation Labeling technique.

*2) **Stacked Inference**:* Following the stacked learning algorithm in Section V-B2, stacked inference is described in Algorithm 3. It involved T classifiers returned by the stack learning algorithm. At the first step, the local classifier is used to compute the class probabilities. At $T-1$ subsequent steps, relational classifiers receives both the local features and relation-specific weighted neighbor probabilities and outputs class probabilities. The final class probabilities are the outcome of the inference process.

VI. Evaluation

Tasks were collected from the JIRA task tracking system in five well-known open source projects: Apache, Duraspace, JBoss, Moodle, and Spring (see Table II), and divided into a training set and a test set. We try to mimic a real project management scenario that prediction on a current task is made using knowledge from the past tasks, the collected tasks in training set are those that were opened before the tasks in test set. The collected datasets are shown in Table II. We have made our datasets publicly available at: http://www.uow.edu.au/∼mc650/. Since the number of delayed tasks in our datasets is small, we chose to use two classes of delay: major delayed and minor delayed (and the non-delayed class).

Table III shows the number of tasks in training set and test set for each project. Major delayed tasks are those that have actual completed date (resolved date) greater than 30 days from planned to completed date and less than 30 days of delays is minor delayed. Note that the size of delayed can be defined by project managers who realize the impact of schedule overruns to the projects. Since (major/minor) delayed tasks are rare and imbalanced, we had to be careful in creating the training and test sets. Specifically, we placed 60% of the delayed tasks into the training set and the remaining 40% into the test set. In addition, we tried to maintain a similar ratio between delayed and non-delayed tasks in both test set and training set, i.e. stratified sampling.

TABLE III: Experimental setting

Project	Training set			Test set		
	Major	Minor	Non	Major	Minor	Non
Apache	10	52	236	6	34	158
Duraspace	23	71	575	16	47	384
JBoss	666	679	3,579	444	452	2,386
Moodle	42	52	770	28	34	513
Spring	13	34	310	8	22	207

A. Performance Measure

Reporting the average of precision/recall across classes is likely to overestimate the true performance, since our risk classes are ordinal and imbalanced and no-delays are the default and they are not of interest to the prediction of delayed tasks. Hence, our evaluation is focus on the predicting of risk classes as described below.

A confusion matrix is used to evaluate the performance of our predictive models. As a confusion matrix does not deal with a multi-class probabilistic classification, we reduce the classified tasks into two binary classes: delayed and non-delayed using the following rule:

$$C_i = \begin{cases} delayed, & if P(i, Maj) + P(i, Min) > P(i, Non) \\ non-delayed, & otherwise \end{cases}$$

Algorithm 1 The stacked learning algorithm (adapted from [24])

Train of the 1-st local classifiers on training nodes, ignoring relations
for step t=2,3,..,T **do**
    Compute the class probabilities for all data points using the (t-1)th classifier
    for each node i **do**
        for each relation r that this node has with its neighbor **do**
            if relation weight exist **then**
                Average all probabilities of its neighbors j who have the relation r with relation weight
            else
                Set relation weight to 1
                Average all probabilities of its neighbors j who have the relation r
            end if
            Prepare $k-1$ features using these averaged probabilities (k probabilities sum to 1)
        end for
        Concatenate all relational features together with the original features
    end for
    Train the t-th local classifier on *training nodes* and new feature sets.
end for
Output T classifiers (one local, T-1 relational)

Algorithm 2 The Relaxation Labeling algorithm (adapted from [25])

Use the 1-st classifier to predict the class probabilities using only local features
for step t=2,3,..,T **do**
    Estimate the prior class probabilities using the relational classifier, M_R, on the current state of network
    Reassign the class of each $v_i \in V^U$ according to the current class probabilities estimation
end for
Output the class probabilities of vertices with unknown labels.

Algorithm 3 The stacked inference algorithm

Use the 1-st classifier to predict the class probabilities using only local features
for step t=2,3,..,T **do**
    Prepare relational features using the neighbor probabilities computed from the previous step
    Use the t-th classifiers to predict the class probabilities using local features and relational features.
end for
Output the class probabilities of vertices with unknown labels.

where C_i is the binary classification of task i, and $P(i, Maj)$, $P(i, Min)$, and $P(i, Non)$ are the probabilities of task i classified in the major delayed, minor delayed, and non-delayed classes respectively. Basically, this rule determines that a task is considered as delayed if the sum probability of it being classified into the major and minor delayed classes is greater than the probability of it being classified into the non-delayed class. Note that our work on this paper focuses on predicting delayed and non delayed tasks. Our evaluations thus emphasize on measuring the performance of predicting whether tasks will cause a delay. We however acknowledge that the ability to distinguish between major and minor is also important. Hence, future work involves using several appropriate performance metrics (e.g. Macro-averaged mean absolute error [30]) to measure the performance of our models in distinguishing between the two delayed classes (major and minor delayed).

The confusion matrix is then used to store the correct and incorrect decisions made by a classifier. For example, if a task is classified as delayed when it truly caused a delay, the classification is a true positive (tp). If the task is classified as delayed when actually it did not cause a delay, then the classification is a false positive (fp). If the task is classified as non-delayed when it in fact caused a delay, then the classification is a false negative (fn). Finally, if the task is classified as non-delayed and it in fact did not cause a delay, then the classification is true negative (tn). The values stored in the confusion matrix are used to compute the widely-used Precision, Recall, and F-measure for the delayed tasks to evaluate the performance of the predictive models:

- Precision: The ratio of correctly predicted delayed task over all the tasks predicted as delayed task. It is calculated as:

$$pr = \frac{tp}{tp + fp}$$

- Recall: The ratio of correctly predicted delayed task over all of the actually task delay. It is calculated as:

$$re = \frac{tp}{tp + fn}$$

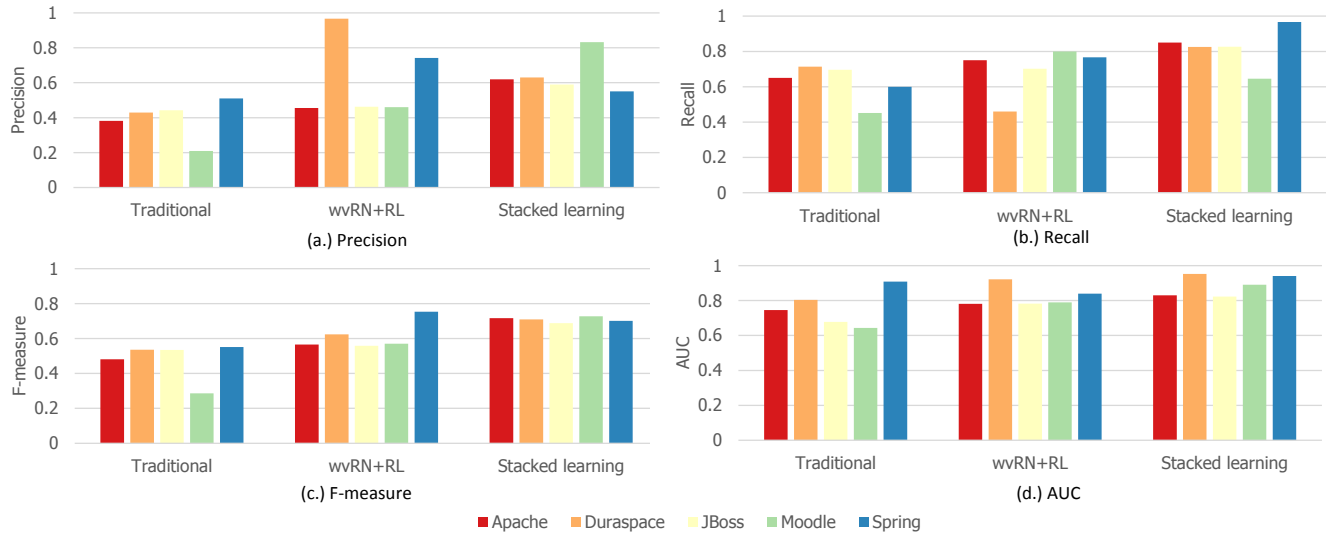

Fig. 5: Evaluation results of traditional classification, wvRN+RL, and stacked learning

- F-measure: Measures the weighted harmonic mean of the precision and recall. It is calculated as:

$$F - measure = \frac{2 * pr * re}{pr + re}$$

- Area Under the ROC Curve (AUC) is used to evaluate the degree of discrimination achieved by the model. The value of AUC is ranged from 0 to 1 and random prediction has AUC of 0.5. The advantage of AUC is that it is insensitive to decision threshold like precision and recall. The higher AUC indicates a better prediction.

B. Results

Comparison of different classification approaches: We compare three different settings: local classifier using Random Forrests (traditional classification), Weighted-Vote Relational Neighbor (wvRN) with Relaxation Labeling (RL), and stacking method (with stacked inference). Figure 5 shows the precision, recall, F-measure, and AUC achieved by three different classification approaches. The stacking method uses Random Forests as the base classifier. The evaluation results indicate that the predictive performance achieved by stacked learning is better and more consistent than traditional classification and relational classification using wvRN+RL. As can be seen in Figure 5(a.), stacked learning achieved the best precision of 0.65 (averaging across five projects), while the traditional classification achieved only 0.39 precision (averaging across five projects). It should however be noted that wvRN+RL achieved the highest precision of 0.97 for Duraspace. In addition, the precision achieved by stacked learning is more consistent and steady in all projects. By contrast, the performance of wvRN+RL are varied between projects. Relational classification with wvRN+RL is based on the principle of homophily, which may not always hold in some projects. This is reflected by its low performance in some cases (i.e. only 0.45 precision for Apache). On the other hand, stacked learning provides a more generalized approach to learn the relationships within networked data – it achieved above 0.5 precision across the five projects.

Stacked learning also outperforms the other classification approaches in terms of recall and F-measure: it achieved the highest recall of 0.83 and the highest F-measure of 0.71 (averaging across five projects) as can be seen in in Figure 5(b.) and 5(c.). The highest recall of 0.97 was also achieved by stack learning for the Spring project.

The degree of discrimination achieved by our predictive models is also high, as reflected in the AUC results. The AUC quantifies the overall ability of the discrimination between the delayed and non-delayed classes. As can be seen in Figure 5(d.), the average of AUC across all classifiers and across all projects is 0.83. All classifiers achieved more than 0.65 AUC while stacked learning is the best performer with 0.88 AUC (averaging across five projects) and 0.95 for Duraspace.

Overall, the evaluation results demonstrate the effectiveness of our predictive models, achieving on average 46%–97% precision, 46%–97% recall, 56%–76% F-measure, and 78%–95% Area Under the ROC Curve. Our evaluation results also show a significant improvement over traditional approaches (local classifiers): 49% improvement in precision, 28% in recall, 39% in F-measure, and 16% in Area Under the ROC Curve.

The usefulness of collective inference: The second aspect of our evaluation focuses on evaluating the predictive performance achieved by using collective inference. To do so, we have setup two experiments: one using wvRN and the other using both wvRN and RL. Figure 6 shows the comparison of the precision, recall, F-measure, and AUC achieved by the relational classification with collective inference (wvRN+RL) and without collective inference (only wvRN). Overall, the predictive performance achieved by relational classification with collective inference is better than that without collective inference in all measures. As can be seen in Figure 6, the relational classification with collective inference achieves the highest precision of 0.61, recall of 0.70, F-measure of 0.62,

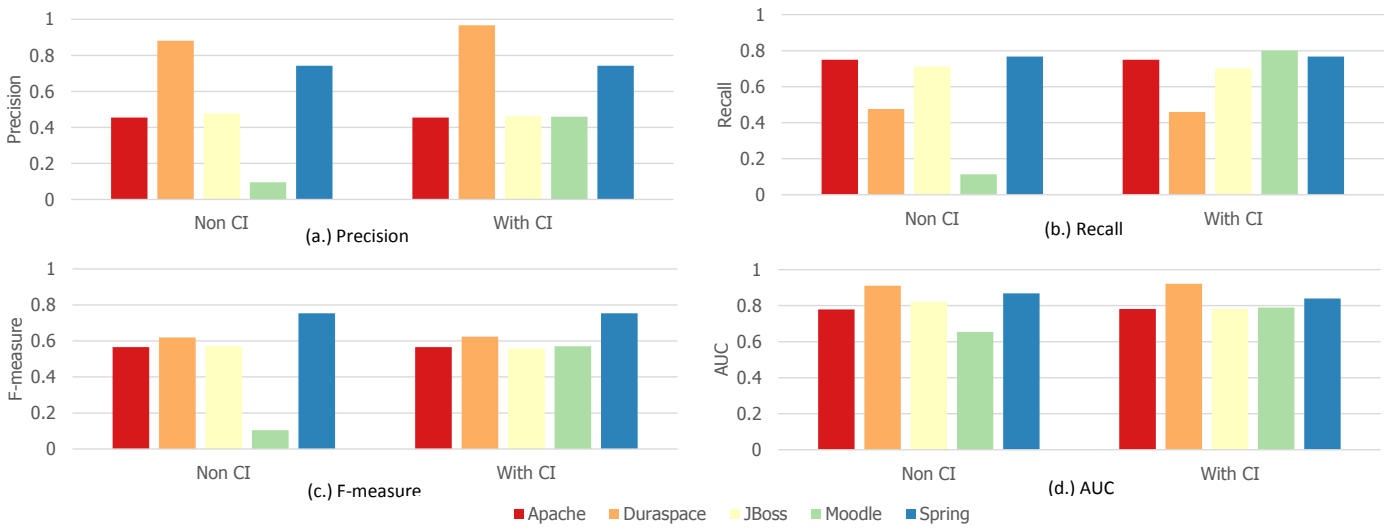

Fig. 6: Evaluation results of relational classifier with collective inference and without collective

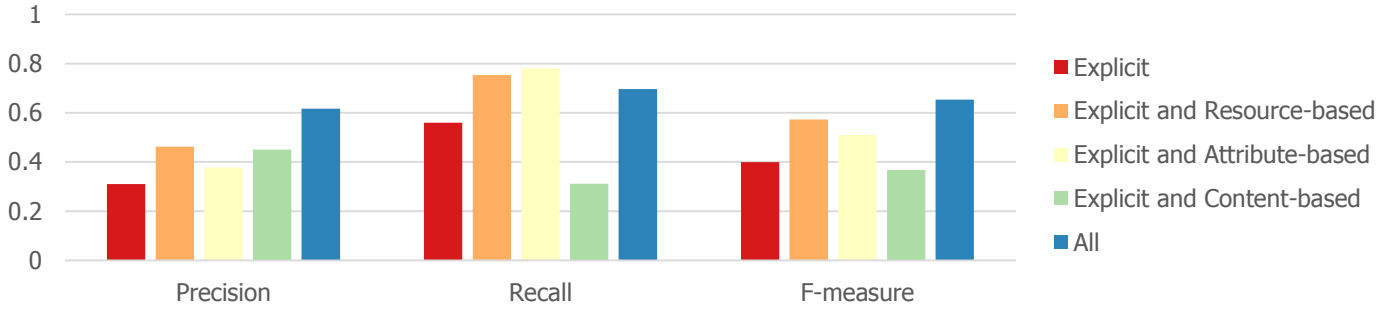

Fig. 7: Evaluation results on different sets of relationships

and 0.82 AUC (averaging across five projects). Although, the predictive performance of Relaxation Labeling is lower than stacked learning as we discussed earlier, the evaluation results still support that collective inference significantly improve the performance of relational classifiers. However, collective inference applied on top of the wvRN still follows a strong assumption of homophily theory and as a result, it causes an inconsistent predictive performance among different projects.

The influence of explicit and implicit relationships: We have also performed a number of experiments to evaluate the predictive performance achieved by different sets of relationships. We have tested with five different combinations: networks with explicit relationships, networks with explicit and resource-based relationships, networks with explicit and attribute-based relationships, networks explicit and content-based relationships, and networks with all explicit and implicit relationships. As can be seen from Figure 7, the highest predictive performance is achieved by using both explicit and implicit relationships: it achieved the highest precision of 0.62 and the highest recall of 0.70 (averaging across five projects). By contrast, the networks using only explicit relationships achieved the lowest precision, i.e. 0.31, while the networks using explicit and content-based relationships produced the lowest recall. In general, using both explicit relationships and implicit relationships (resource-based, attribute-based, and content-based) significantly increases the predictive performance: 66.23 % increased in precision and 21.62 % increased in recall (compare to using only explicit relationships).

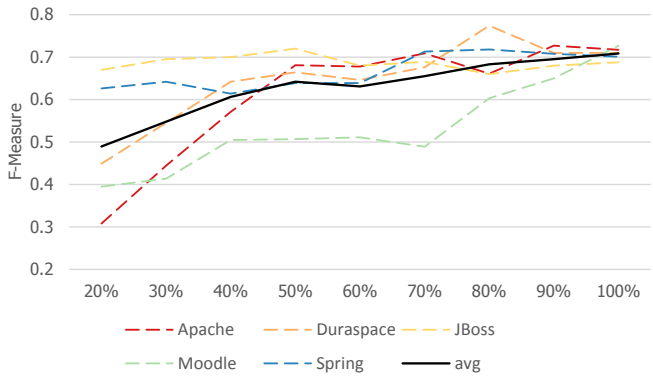

Fig. 8: Evaluation results on different sizes of training data

The effect of the size of training data: We have also performed a number of experiments to assess the proportion of past tasks (i.e. labeled tasks) is needed to achieve a good predictive performance. Specifically, in these experiments,

given a data set, $G = (V, E)$, V^K (i.e. labeled tasks) is created by selecting samples of 20% – 100% of the training set (see Table III). The test set, V^U, is then defined as $V \setminus V^K$. Figure 8 shows the predictive performance from samples of 20% to 100% of V in terms of F-measure. The results clearly demonstrate that F-measure (averaging across five projects) increases as more labeled data is used for training.

VII. Threats to validity

One relational setting involves the use of wvRN, which assumes the homophily property among tasks, that is, related tasks should have similar delay risk. This is a strong assumption and may not hold in reality, and this has been revealed in our experiments. We have addressed this threat by proposing stacked learning approach which does not rely on the the homophily assumption but rather estimates the contribution of separate relationships.

We have attempted to identify all possible relationships among typical software tasks. However, we acknowledge that the implicit relationships we have inferred are by no means comprehensive to represent all task dependencies. Another threat to our study is that our data set has the class imbalance problem (over 90% of the total data are non-delayed tasks), which may affect a classifier's ability to learn to identify delayed tasks. We have used stratified sampling to mitigate this problem. We however acknowledge such a sampling approach could be an external threat to validity. Further experiments to evaluate sampling techniques used in practice are thus needed. In addition, patterns that hold in the train data may not reflect the situation in the test, e.g. the team and management having changed their approach or managed the risks they perceived. To address this threat, instead of splitting the data randomly (as done in traditional settings), we deliberately chose the time to split training and test sets to mimic a real deployment.

We have considered 11,851 task reports from the five projects which differ significantly in size, complexity, development process, and the size of community. Although these are real data, we however cannot claim that our data set would be representative of all kinds of software projects, especially in commercial settings. Although open source projects and commercial projects share similarities in many aspects, they are also different in the nature of contributors, developers and projects stakeholders. For example, open source contributors are free to join and leave the communities (i.e. high turn over rate), while developers in the commercial setting tend to be stable and fully commit to deliver the projects progress. Hence, further study is need to understand how our predict models perform for commercial projects.

VIII. Related Work

An automated risk prediction mainly supports software risk management which is under the umbrella of project management and crucial to the project success rate. Risk management consists of two main activities: risk assessment and risk control. Our current work focuses on risk assessment, which is a process of identifying risks, analyzing and evaluating their potential effects in order to prioritize them [4, 31]. Risk control aims to develop, engage, and monitor risk mitigation plans [32].

There are a number of works on applying statistical and machine learning techniques to use in different aspects of risk management. For example, Letier *et al.* [33] used a statistical decision analysis approach to provide a statistical support on complex requirements and architecture. The work in [34] analyzed the correlation and causality of risk factors using Bayesian network.

Our work also related to the works on predicting and mining bug reports, for example, mining bug reports for fix-time prediction (e.g. [7–11]), blocking bug prediction (e.g. [35]), re-opened bug prediction (e.g. [36, 37]), severity/priority prediction (e.g. [38, 39]), delays in the integration of a resolved issue to a release (e.g. [40]), bug triaging (e.g. [41–44]), and duplicate bug detection ([45–50]). Another groups of the works on predicting is mining source code to predict software defects, for example, mining change history (e.g. [51]), and personalized defect prediction (e.g. [52])

Anothe thread of related work resides in the use of networked data such as predicting software quality using social network analysis (e.g. [53–56]), predicting software evolution in terms of estimating bug severity, efforts, and defect-prone releases using Graph-based analysis (e.g. [20]), and predicting software defects using network analysis on dependency graphs (e.g. [57]). Those approaches mostly work at the level of source code and have not addressed delay prediction at the task level as in our work.

IX. Conclusions and Future Work

In this paper, we have proposed a novel approach to predict whether a number of existing tasks in a software project are at risk of being delayed. Our approach exploits not only features specific to individual tasks but also the relationships between the tasks (i.e. networked data). We have developed several prediction models using local classifiers, relational classifiers and collective inference. The evaluation results demonstrate a strong predictive performance of our networked classification techniques compared to traditional approaches: achieving 49% improvement in precision, 28% improvement in recall, 39% improvement in F-measure, and 16% improvement in Area Under the ROC Curve. In particular, the stacked graphical learning approach consistently outperformed the other techniques across the five projects we studied.

The results from our experiments indicate that the relationships between tasks have an impact on the predictive performance. Hence, as part of future work, we will investigate which types of task relationships should be selected to give optimal results and how this can be done automatically for software projects. A related future investigation would involve applying different weights to different task relationships and assessing their impact to the results. We also plan to investigate if there are any other kinds of implicit relationships which can be inferred from the task information and the general context of a software project. Our future work would involve expanding our study to commercial software projects and other large open source projects to further assess our predictive models. Finally, delay prediction which has been addressed in this paper is just only the first part of the solution. The next task is making actionable recommendations such as which tasks having a risk of being delayed should be dealt with first, and which measures could be used to mitigate the risks.

REFERENCES

[1] B. Flyvbjerg and A. Budzier, "Why Your IT Project May Be Riskier Than You Think," *Harvard Business Review*, vol. 89, no. 9, pp. 601–603, 2011.

[2] B. Michael, S. Blumberg, and J. Laartz, "Delivering large-scale IT projects on time, on budget, and on value," 2012.

[3] S. Group, "Chaos report," West Yarmouth, Massachusetts: Standish Group, Tech. Rep., 2004.

[4] B. W. Boehm, "Software risk management: principles and practices," *Software, IEEE*, vol. 8, no. 1, pp. 32–41, 1991.

[5] M. J. Carr and S. L. Konda, "Taxonomy-Based Risk Identification," Software Engineering Institute, Carnegie Mellon University, Tech. Rep. June, 1993.

[6] M. Choetkiertikul, H. K. Dam, T. Tran, and A. Ghose, "Characterization and prediction of issue-related risks in software projects," in *Proceedings of 12th Working Conference on Mining Software Repositories (MSR-2015)*, 2015, pp. 280–291.

[7] C. Weiss, R. Premraj, T. Zimmermann, and A. Zeller, "How Long Will It Take to Fix This Bug?" in *Proceedings - ICSE 2007 Workshops: Fourth International Workshop on Mining Software Repositories, MSR 2007*. IEEE, May 2007.

[8] E. Giger, M. Pinzger, and H. Gall, "Predicting the fix time of bugs," in *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering - RSSE '10*. ACM Press, May 2010, pp. 52–56.

[9] L. D. Panjer, "Predicting Eclipse Bug Lifetimes," in *Fourth International Workshop on Mining Software Repositories (MSR'07:ICSE Workshops 2007)*. IEEE, May 2007, pp. 29–29.

[10] L. Marks, Y. Zou, and A. E. Hassan, "Studying the fix-time for bugs in large open source projects," in *Proceedings of the 7th International Conference on Predictive Models in Software Engineering - Promise '11*. New York, New York, USA: ACM Press, Sep. 2011, pp. 1–8.

[11] P. Bhattacharya and I. Neamtiu, "Bug-fix time prediction models," in *Proceeding of the 8th working conference on Mining software repositories - MSR '11*. New York, New York, USA: ACM Press, May 2011, p. 207.

[12] J. Neville and D. Jensen, "Collective Classification with Relational Dependency Networks," in *Proceedings of the Second International Workshop on Multi-Relational Data Mining*, 2003, pp. 77–91.

[13] B. Taskar, V. Chatalbashev, and D. Koller, "Learning Associative Markov Networks," *Proc. of the International Conference on Machine Learning*, pp. 102–110, 2004.

[14] Q. Lu and L. Getoor, "Link-based classification," in *ICML*, vol. 3, 2003, pp. 496–503.

[15] E. Segal, R. Yelensky, and D. Koller, "Genome-wide discovery of transcriptional modules from DNA sequence and gene expression," *Bioinformatics*, vol. 19, pp. 273–282, 2003.

[16] L. Breiman, "Random forests," *Machine learning*, pp. 5–32, 2001.

[17] C. Science, R. Holloway, and L. Egham, "Support Vector Machines for Multi-Class Pattern Recognition," in *European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning*, vol. 99, 1999, pp. 219–224.

[18] D. Böhning, "Multinomial logistic regression algorithm," *Annals of the Institute of Statistical Mathematics*, vol. 44, no. 1, pp. 197–200, 1992.

[19] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent Dirichlet Allocation," *Journal of Machine Learning Research*, vol. 3, no. 4-5, pp. 993–1022, 2012.

[20] P. Bhattacharya, M. Iliofotou, I. Neamtiu, and M. Faloutsos, "Graph-based Analysis and Prediction for Software Evolution," in *Proceedings of the 34th International Conference on Software Engineering (ICSE 2012)*, 2012, pp. 419–429.

[21] S. a. Macskassy and F. Provost, "Classification in Networked Data: A toolkit and a univariate case study," *Journal of Machine Learning Research*, vol. 8, no. December 2004, pp. 1–41, 2005.

[22] S. L. Lauritzen, *Graphical models*. Oxford University Press, 1996.

[23] S. a. Macskassy and F. Provost, "A simple relational classifier," *Proceeding of the 2nd Workshop on Multi-Relational Data Mining (MRDM 03)*, pp. 64–76, 2003.

[24] Z. Kou and W. W. Cohen, "Stacked Graphical Models for Efficient Inference in Markov Random Fields," *SIAM International Conference on Data Mining*, pp. 533–538, 2007.

[25] S. a. Macskassy, "Relational classifiers in a non-relational world: Using homophily to create relations," *Proceedings - 10th International Conference on Machine Learning and Applications, ICMLA 2011*, vol. 1, pp. 406–411, 2011.

[26] M. McPherson, L. Smith-Lovin, and J. M. Cook, "Birds of a Feather: Homophily in Social Networks," *Annual Review of Sociology*, vol. 27, no. 1, pp. 415–444, 2001.

[27] P. Vojtek and M. Bieliková, "Homophily of neighborhood in graph relational classifier," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 5901 LNCS, pp. 721–730, 2010.

[28] B. Golub and M. O. Jackson, "How homophily affects the speed of learning and best-response dynamics," *Quarterly Journal of Economics*, vol. 127, no. 3, pp. 1287–1338, 2012.

[29] R. a. Hummel and S. W. Zucker, "On the foundations of relaxation labeling processes." *IEEE transactions on pattern analysis and machine intelligence*, vol. 5, no. 3, pp. 267–287, 1983.

[30] S. Baccianella, a. Esuli, and F. Sebastiani, "Evaluation Measures for Ordinal Regression," *2009 Ninth International Conference on Intelligent Systems Design and Applications*, 2009.

[31] Xu Ruzhi, Q. leqiu, and Jing Xinhai, "CMM-based software risk control optimization," in *Proceedings Fifth IEEE Workshop on Mobile Computing Systems and Applications*. IEEE, 2003, pp. 499–503.

[32] M. Choetkiertikul and T. Sunetnanta, "A Risk Assessment Model for Offshoring Using CMMI Quantitative Approach," in *2010 Fifth International Conference on Software Engineering Advances*. IEEE, Aug. 2010, pp. 331–336.

[33] E. Letier, D. Stefan, and E. T. Barr, "Uncertainty, risk, and information value in software requirements and architecture," in *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014*. New York, New York, USA: ACM Press, May 2014, pp. 883–894.

[34] Y. Hu, X. Zhang, E. Ngai, R. Cai, and M. Liu, "Software project risk analysis using Bayesian networks with causality constraints," *Decision Support Systems*, vol. 56, pp. 439–449, Dec. 2013.

[35] H. Valdivia Garcia, E. Shihab, and H. V. Garcia, "Characterizing and predicting blocking bugs in open source projects," in *Proceedings of the 11th Working Conference on Mining Software Repositories - MSR 2014*. ACM Press, May 2014, pp. 72–81.

[36] T. Zimmermann, N. Nagappan, P. J. Guo, and B. Murphy, "Characterizing and predicting which bugs get reopened," in *34th International Conference on Software Engineering (ICSE), 2012*. IEEE Press, Jun. 2012, pp. 1074–1083.

[37] E. Shihab, A. Ihara, Y. Kamei, W. M. Ibrahim, M. Ohira, B. Adams, A. E. Hassan, and K.-i. Matsumoto, "Studying re-opened bugs in open source software," *Empirical Software Engineering*, vol. 18, no. 5, pp. 1005–1042, Sep. 2012.

[38] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals, "Predicting the severity of a reported bug," in *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*. IEEE, May 2010, pp. 1–10.

[39] T. Menzies and A. Marcus, "Automated severity assessment of software defect reports," *2008 IEEE International Conference on Software Maintenance*, pp. 346–355, Sep. 2008.

[40] D. Alencar, S. L. Abebe, and S. Mcintosh, "An Empirical Study of Delays in the Integration of Addressed Issues."

[41] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?" in *Proceeding of the 28th international conference on Software engineering - ICSE '06*. New York, New York, USA: ACM Press, May 2006, p. 361.

[42] J. Anvik and G. C. Murphy, "Reducing the effort of bug report triage," *ACM Transactions on Software Engineering and Methodology*, vol. 20, no. 3, pp. 1–35, Aug. 2011.

[43] M. M. Rahman, G. Ruhe, and T. Zimmermann, "Optimized assignment of developers for fixing bugs an initial evaluation for eclipse projects," in *2009 3rd International Symposium on Empirical Software Engineering and Measurement*. IEEE, Oct. 2009, pp. 439–442.

[44] G. Murphy and D. Čubranić, "Automatic bug triage using text categorization," in *Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering*, 2004.

[45] P. Runeson, M. Alexandersson, and O. Nyholm, "Detection of Duplicate Defect Reports Using Natural Language Processing," in *29th International Conference on Software Engineering (ICSE'07)*. IEEE, May 2007, pp. 499–510.

[46] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun, "An approach to detecting duplicate bug reports using natural language and execution information," in *Proceedings of the 30th international conference on Software engineering*, 2008, pp. 461–470.

[47] N. Bettenburg, R. Premraj, and T. Zimmermann, "Duplicate bug reports considered harmful ... really?" in *2008 IEEE International Conference on Software Maintenance*. IEEE, 2008, pp. 337–345.

[48] C. Sun, D. Lo, S. C. Khoo, and J. Jiang, "Towards more accurate retrieval of duplicate bug reports," *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pp. 253–262, Nov. 2011.

[49] N. Jalbert and W. Weimer, "Automated duplicate detection for bug tracking systems," in *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*. IEEE, 2008, pp. 52–61.

[50] A. T. Nguyen, T. T. T. N. Nguyen, D. Lo, and C. Sun, "Duplicate bug report detection with a combination of information retrieval and topic modeling," *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering - ASE 2012*, p. 70, 2012.

[51] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "Detecting bad smells in source code using change history information," *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013 - Proceedings*, pp. 268–278, 2013.

[52] T. Jiang, L. Tan, and S. Kim, "Personalized defect prediction," *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013 - Proceedings*, pp. 279–289, 2013.

[53] N. Bettenburg and A. E. Hassan, "Studying the impact of dependency network measures on software quality," *Empirical Software Engineering*, 2012.

[54] T. Wolf, A. Schröter, D. Damian, and T. Nguyen, "Predicting build failures using social network analysis on developer communication," *Proceedings - International Conference on Software Engineering*, pp. 1–11, 2009.

[55] A. Meneely, L. Williams, W. Snipes, and J. Osborne, "Predicting failures with developer networks and social network analysis," *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering - SIGSOFT '08/FSE-16*, p. 13, 2008.

[56] W. Hu and K. Wong, "Using citation influence to predict software defects," in *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, May 2013, pp. 419–428.

[57] T. Zimmermann and N. Nagappan, "Predicting defects using network analysis on dependency graphs," in *Proceedings of the 13th international conference on Software engineering - ICSE '08*. New York, New York, USA: ACM Press, May 2008, p. 531.