Predicting Delivery Capability in Iterative Software Development

Morakot Choetkiertikul, Hoa Khanh Dam, Truyen Tran, Aditya Ghose, and John Grundy

Abstract—Iterative software development has become widely practiced in industry. Since modern software projects require fast, incremental delivery for every iteration of software development, it is essential to monitor the execution of an iteration, and foresee a capability to deliver quality products as the iteration progresses. This paper presents a novel, data-driven approach to providing automated support for project managers and other decision makers in predicting delivery capability for an ongoing iteration. Our approach leverages a history of project iterations and associated issues, and in particular, we extract characteristics of previous iterations and their issues in the form of features. In addition, our approach characterizes an iteration using a novel combination of techniques including feature aggregation statistics, automatic feature learning using the Bag-of-Words approach, and graph-based complexity measures. An extensive evaluation of the technique on five large open source projects demonstrates that our predictive models outperform three common baseline methods in Normalized Mean Absolute Error and are highly accurate in predicting the outcome of an ongoing iteration.

Index Terms—Mining software engineering repositories, Empirical software engineering, Iterative software development

1 Introduction

Late delivery and cost overruns have been common problems in software projects for many years. A recent study on 5,400 large scale IT projects by Mckinsey and the University of Oxford in 2012 [1] has found that on average large software projects run 66% over budget and 33% overtime. One in six of the 1,471 software projects studied by Flyvbjerg and Budzier [2] was a "black swan" – a term used to refer to projects with a budget overrun of 200% and a schedule overrun of almost 70%. These studies have also suggested that ineffective risk management is one of the main causes for such a high rate of overruns in software projects. Central to risk management is the ability to predict, at any stage in a project, if a software development team can deliver on time, within budget, and with all planned features.

Modern software development is mostly based on an incremental and iterative approach in which software is developed through repeated cycles (iterative) and in smaller parts at a time (incremental), allowing software developers to benefit from what was learned during development of earlier portions or versions of the software. Incremental and iterative development are essential parts of many popular software development methodologies such as (Rational) Unified Process, Extreme Programming, Scrum and other agile software development methods [3]. This is achieved by moving from a model where all software packages are delivered together (in a single delivery) to a model involving a series of incremental deliveries, and working in small iterations. Uncertainties, however, exist in software projects due to their inherent dynamic nature (e.g. constant changes to software requirements) regardless of which development process is employed. Therefore, an effective planning, progress monitoring, and predicting are still critical for iterative software development, especially given the need for rapid delivery [4]. Our focus here is on predicting the delivery capability of a *single iteration at a time*, rather than the whole software lifecyle as in traditional waterfall-like software development processes.

There has been a large body of work on building various prediction models to support software development. For example, existing effort estimation models (e.g. [5], [6], and [7]) was built to predict the effort required for developing a whole software, not a single iteration at a time. Other existing empirical work (e.g. [8], [9], [10], [11], and [12]) has only considered how prediction was done at the project level. It would however be much more valuable for project managers and decision makers to be provided with insightful and actionable information at the level of iterations. For example, being able to predict that an iteration is at risk of not delivering what has been planned allows project managers the opportunity adapt a current plan earlier, e.g. moving some features from the current iteration to the next one.

Our work in this paper aims to fill this gap. We focus on predicting delivery capability as to whether the target amount of work will be delivered at the end of an iteration. Our proposal, with its ability to learn from prior iterations to predict the performance of future iterations, represents an important advance in our ability to effectively use the incremental model of software development. To do so, we have developed a dataset of 3,834 iterations in five large open source projects, namely Apache, JBoss, JIRA, MongoDB, and Spring. Each iteration requires the completion of a number of work items (commonly referred to as *issues*), thus 56,687 issues were collected from those iterations.

- *M. Choetkiertikul, H. Dam, and A. Ghose are with the School of Computing and Information Technology, Faculty of Engineering and Information Sciences, University of Wollongong, NSW, Australia, 2522.*
  E-mail: {mc650, hoa, aditya}@uow.edu.au
- *T. Tran and J. Grundy is with the School of Information Technology, Deakin University, Victoria, Australia, 3216.*
  E-mail: {truyen.tran, j.grundy}@deakin.edu.au

To build a feature vector representing an iteration, we extracted fifteen attributes associated with an iteration (e.g. its duration, the number of participants, etc.). In addition, an iteration is also characterized by its issues, thus we also extracted twelve attributes associated with an issue (e.g. type, priority, number of comments, etc.) and a graph describing the dependency between these issues (e.g. one issue blocks another issue). The complexity descriptors of the dependency graph (e.g. number of nodes, edges, fan in and fan out) form a group of features for an iteration. The attributes of the set of issues in an iteration are also combined, using either statistic aggregation or bag-of-words method, to form another group of features for an iteration. Statistical aggregation looks for simple set statistics for each issue attribute such as max, mean or standard deviation (e.g. the maximum number of comments in all issues in a iteration). On the other hand, the bag-of-words technique clusters all the issues in a given project, then finds the closest prototype (known as "word") for each issue in an iteration to build a feature vector. The bag-of-words method therefore obviates the need for manual feature engineering. Our novel approach of employing multiple layers of features and automatic feature learning is similar to the notion of increasingly popular deep learning methods.

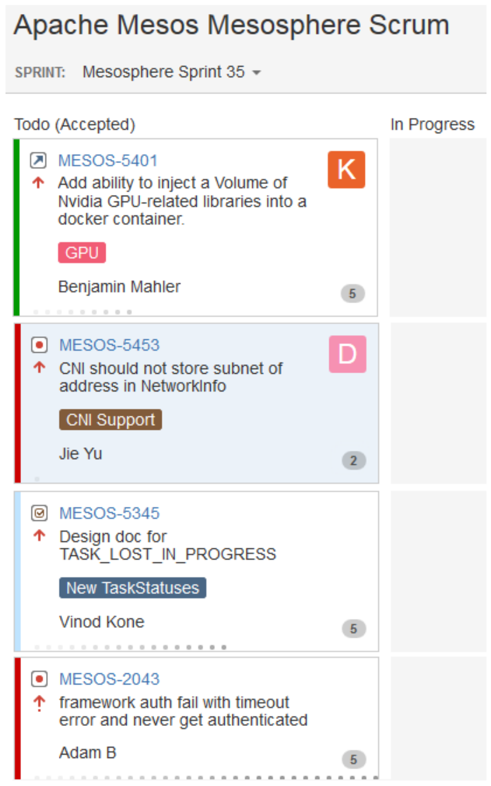
From the above we then developed accurate models that can predict delivery capability of an iteration (i.e. how much work was actually completed in an iteration against the target). Our predictive models are built based on three different state-of-the-art *randomized ensemble methods*: Random Forests, Stochastic Gradient Boosting Machines, and Deep Neural Networks with Dropouts. An extensive evaluation was performed across five projects (with over three-thousand iterations) and our evaluation results demonstrate that our approach outperforms three common baselines and performs well across all case studies.

The remainder of this paper is organized as follows. Section 2 discusses the motivation and the prediction tasks that our work addresses. Section 3 presents an overview of our approach, while Section 4 presents a comprehensive set of features and describes how these features are aggregated. Section 5 presents the predictive models we have developed. Section 6 reports on the experimental evaluation of our approach. Related work is discussed in Section 7 before we conclude and outline future work in Section 8.

2 Motivation

In iterative, agile software development, a project has a number of *iterations* (which are referred to as *sprints* in Scrum [13]). An iteration is usually a short (usually 2–4 weeks) period in which the development team designs, implements, tests and delivers a distinct *product increment*, e.g. a working milestone version or a working release. Each iteration requires the resolution/completion of a number of *issues*. For example, iteration *Mesossphere Sprint 35*[1] in the Apache project (see Figure 1) requires the completion of four issues: *MESOS-5401*, *MESOS-5453*, *MESOS-5445*, and *MESOS-2043*. At the beginning of the iteration (i.e. May 14, 2016), all of these issues were placed in the Todo list (or also referred to as the iteration or sprint backlog) . The iteration was scheduled to finish on May 30, 2016.

1. https://issues.apache.org/jira/secure/RapidBoard.jspa?rapidView=62&view=reporting&chart=sprintRetrospective&sprint=236


Apache Mesos Mesosphere Scrum

SPRINT: Mesosphere Sprint 35

Todo (Accepted)

In Progress

MESOS-5401
Add ability to inject a Volume of Nvidia GPU-related libraries into a docker container.
GPU
Benjamin Mahler
5

MESOS-5453
CNI should not store subnet of address in NetworkInfo
CNI Support
Jie Yu
2

MESOS-5345
Design doc for TASK_LOST_IN_PROGRESS
New TaskStatuses
Vinod Kone
5

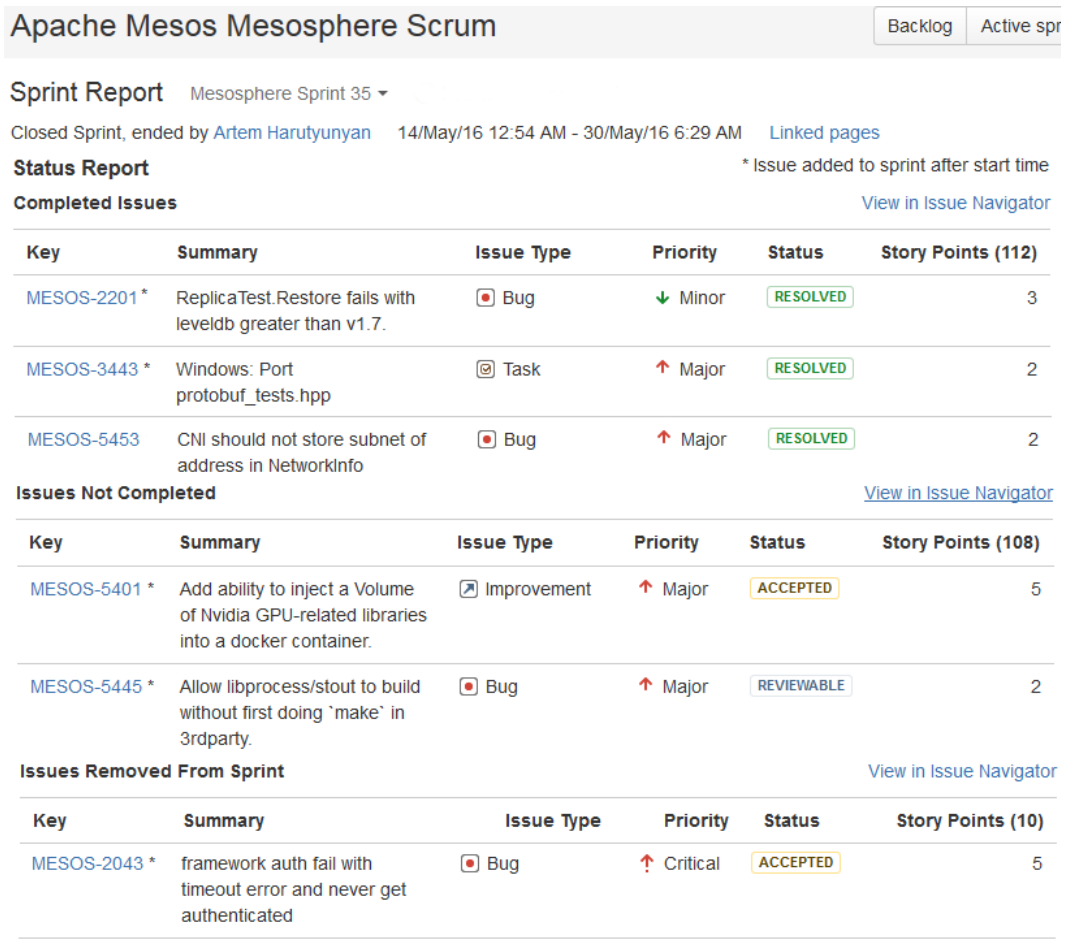
MESOS-2043
framework auth fail with timeout error and never get authenticated
Adam B
5

Fig. 1. An example of an iteration (at the beginning)

Planning is done before an iteration starts and focuses on determining its starting time, completion time and the issues to be resolved in this iteration. Agile approaches recommend that an iteration be *time-boxed* (i.e. have a fixed duration) [4]. During an iteration, issues can be added and removed from the iteration. At the end of an iteration, a number of issues are completed and there may also be a number of issues assigned to the iteration that remain incomplete/unresolved. These incomplete/unresolved issues may be assigned to future iterations.

Let t_{pred} refer to the time at which a prediction is being made (e.g. the third day of a 17-day iteration). Given time t_{pred} during an iteration, **we would like to predict the amount of work delivered at the end of an iteration (i.e. the number of issues resolved), relative to the amount of work which the team has originally committed to.** More specifically, let `Committed` be the set of issues that the team commits to achieve in the current iteration before time t_{pred}. Let `Delivered` be the set of issues actually delivered at the end of the iteration, and `NonDelivered` be the set of issues that the team committed to delivering but failed to deliver in this iteration. Note that `NonDelivered` includes issues from the `Committed` set but are removed from the iteration after prediction time t_{pred} and/or issues that are not completed when the iteration ends.

We illustrate and motivate this using the example in Figure 1. At the beginning, a team planned to deliver 4 issues which are *MESOS-5401*, *MESOS-5453*, *MESOS-5445*, and *MESOS-2043* from iteration *Mesossphere Sprint 35* which



Apache Mesos Mesosphere Scrum

Backlog | Active spr

Sprint Report Mesosphere Sprint 35

Closed Sprint, ended by Artem Harutyunyan 14/May/16 12:54 AM - 30/May/16 6:29 AM Linked pages

Status Report * Issue added to sprint after start time

Completed Issues View in Issue Navigator

Key	Summary	Issue Type	Priority	Status	Story Points (112)
MESOS-2201 *	ReplicaTest.Restore fails with leveldb greater than v1.7.	Bug	Minor	RESOLVED	3
MESOS-3443 *	Windows: Port protobuf_tests.hpp	Task	Major	RESOLVED	2
MESOS-5453	CNI should not store subnet of address in NetworkInfo	Bug	Major	RESOLVED	2

Issues Not Completed View in Issue Navigator

Key	Summary	Issue Type	Priority	Status	Story Points (108)
MESOS-5401 *	Add ability to inject a Volume of Nvidia GPU-related libraries into a docker container.	Improvement	Major	ACCEPTED	5
MESOS-5445 *	Allow libprocess/stout to build without first doing `make` in 3rdparty.	Bug	Major	REVIEWABLE	2

Issues Removed From Sprint View in Issue Navigator

Key	Summary	Issue Type	Priority	Status	Story Points (10)
MESOS-2043 *	framework auth fail with timeout error and never get authenticated	Bug	Critical	ACCEPTED	5

Fig. 2. An example of a closed iteration report

is a 17–day iteration. Assume that, on the second day, the team added 2 new issues to the iteration: *MESOS-2201* and *MESOS-3443*. Let assume that three days after the iteration started we would like to make a prediction; i.e. t_{pred} at day 3. The number of issues in `Committed` is then 6 issues (*MESOS-5401*, *MESOS-5453*, *MESOS-5445*, *MESOS-2043*, *MESOS-2201*, and *MESOS-3443*) at the time t_{pred} when the prediction is made. After the third day, issue *MESOS-2043* has been removed from the iteration, and at the end of the iteration, a team completed only 3 issues which are *MESOS-2201*, *MESOS-3443*, and *MESOS-5453*, while the remaining issues (*MESOS-5401* and *MESOS-5445*) were not resolved. Thus, the issues in `Delivered` are *MESOS-2201*, *MESOS-3443*, and *MESOS-5453* and the issues in `NonDelivered` are *MESOS-5401* and *MESOS-5445* which were not completed and *MESOS-2043* which was removed from the iteration (see Figure 2). We note that an issue's status (e.g. resolved or accepted) also corresponds to the associated iteration's report, e.g. *MESOS-2201* was resolved and was placed in the completed list while issue *MESOS-5445* was in the reviewing process.

Predicting which issues would belong to the `Delivered` sets is difficult, and in some cases is impossible, e.g. some new issues in the `Delivered` set could be added after prediction time t_{pred}. **Hence, we propose to quantify the amount of work done in an iteration and use this as the basis for our prediction.** This approach reflects common practice in agile software development. For example, several agile methods (e.g. Scrum) suggest the use of *story points* to represent the effort, complexity, uncertainty, and risks involving resolving an issue [4]. Specifically, a story point is assigned to each issue in an iteration. The amount of work done in an iteration is then represented as a *velocity*, which is the total story points completed in the iteration. Velocity reflects how much work a team gets done in an iteration.

Definition 1 (Velocity). Velocity of a set of issues $\mathcal{I}$ is the sum story points of all the issues in $\mathcal{I}$:

$$velocity(\mathcal{I}) = \sum_{i \in \mathcal{I}} sp(i)$$

where $sp(i)$ is the story point assigned to issue i.

For example, as can be seen from Figure 2, there are six issues that were committed to be delivered from iteration *Mesossphere Sprint 35* in the Apache project at the prediction time t_{pred} (e.g. issue *MESOS-2201* has 3 story points). Thus, the committed velocity is 19, i.e. $velocity(\texttt{Committed}) = 19$. However, there are only three issues had been resolved in iteration *Mesossphere Sprint 35*. Thus, the velocity delivered from this iteration is 7, i.e. $velocity(\texttt{Delivered}) = 7$ (two issues have the story points of 2 and one of them has the story point of 3).

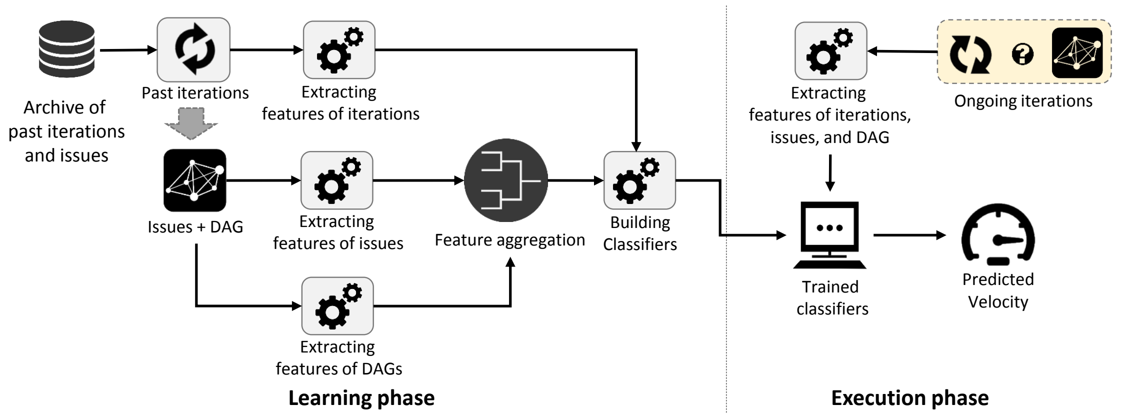

Archive of past iterations and issues
Past iterations
Extracting features of iterations
Issues + DAG
Extracting features of issues
Feature aggregation
Building Classifiers
Extracting features of DAGs
Learning phase
Extracting features of iterations, issues, and DAG
Ongoing iterations
Trained classifiers
Predicted Velocity
Execution phase

Fig. 3. An overview of our approach

We thus would like to predict the delivery capability in an iteration. Our approach is able to predict, given the current state of the project at time t_{pred}, what is the difference between the actual delivered velocity against the committed (target) velocity, defined as $velocity(\texttt{Difference})$:

$$velocity(\texttt{Difference}) = velocity(\texttt{Delivered}) - velocity(\texttt{Committed})$$

For example, the difference between the actual delivered velocity against the committed velocity of iteration *Mesossphere Sprint 35* was -12, i.e. $velocity(\texttt{Difference}) = -12$, because $velocity(\texttt{Delivered})$ was 7 and $velocity(\texttt{Committed})$ at t_{pred} was 19. This iteration delivered below the target, i.e. $velocity(\texttt{Committed}) > velocity(\texttt{Delivered})$. Note that $velocity(\texttt{Difference}) = 0$ does not necessarily imply that an iteration has delivered on all its commitments (in terms of the specific issues that were to be resolved) but instead it assesses the quantum of work performed.

3 Overview of our approach

Our approach consists of two phases: the *learning phase* and the *execution phase* (see Figure 3). The learning phase involves using historical iterations to build a predictive model (using machine learning techniques), which is then used to predict outcomes, i.e. $velocity(\texttt{Difference})$, of new and ongoing iterations in the execution phase. To apply machine learning techniques, we need to engineer features for the iteration. An iteration has a number of attributes (e.g. its duration, the participants, etc.) and a set of issues whose dependencies are described as a dependency graph. Each issue has its own attributes and derived features (e.g. from its textual description). Our approach separates the iteration-level features into three components: (i) iteration attributes, (ii) complexity descriptors of the dependency graph (e.g. the number of nodes, edges, fan-in, fan-out, etc.), and (iii) aggregated features from the set of issues that belong to the iteration. A more sophisticated approach would involve embedding all available information into an Euclidean space, but we leave this for future work.

Formally, issue-level features are vectors located in the same Euclidean space (i.e. the issue space). The aggregation is then a map of a set points in the issue space onto a point in the iteration space. The main challenge here is to handle sets which are unordered and variable in size (e.g. the number of issues is different from iteration to iteration). We propose two methods: statistical aggregation and bag-of-words (BoW). Statistical aggregation looks for simple set statistics for each dimension of the points in the set, such as maximum, mean or standard deviation. For example, the minimum, maximum, mean, and standard deviation of the number of comments of all issues in an iteration are part of the new features derived for the iteration. This statistical aggregation technique relies on manual feature engineering. On the other hand, the bag-of-words method automatically clusters all the points in the issue-space and finds the closest prototype (known as "word") for each new point to form a new set of features (known as bag-of-words, similarly to a typical representation of a document) representing an iteration. This technique provides a powerful, automatic way of learning features for an iteration from the set of issues in the layer below it (similar to the notions of deep learning).

For prediction models, we employ three state-of-the-art *randomized ensemble methods*: Random Forests, Stochastic Gradient Boosting Machines, and Deep Neural Networks (DNNs) with Dropouts to build the predictive models. Our approach is able to make a prediction regarding the delivery capability in an iteration (i.e. the difference between the actual delivered velocity against the committed (target) velocity). Next we describe our approach in more detail.

4 Feature extraction and aggregation

In this section, we describe our feature extraction from iteration and issue reports. Since an iteration has a set of issues to be resolved, we extract not only features of an iteration, but also features of an issue and the issue graph (i.e. dependency of issues) in an iteration. Most modern issue tracking systems (e.g. JIRA-Agile[2]) support an iterative, agile development which enables teams to use agile practices for their development to plan, collaborate, monitor and organize iterations and issues. Dependencies between issues are also explicitly recorded (i.e. issue links) in the issues reports which both iterative and issue reports can

2. https://www.atlassian.com/software/jira/agile

be easily extracted from there. We then employ a number of distinct techniques (feature aggregation using statistics, Bag-of-Words, and graph measures) to characterize an iteration using both features of an iteration and features of a number of issues associated to an iteration. These details will be discussed in this section.

4.1 Features of an iteration

Table 1 summarizes a range of features that are used to characterize an iteration in our study. The features cover three important areas of an iteration: the elapsed time (e.g. the planned duration), the amount of work, and the team. Prediction time (i.e. t_{pred}) is used as a reference point to compute a number of features reflecting the amount of work. These include the set of issues assigned to an iteration when it begins (i.e. start time), and the set of issues added or removed from the iteration between the start time and prediction time. In addition, we also leverage a number of features reflecting the current progress of the team up until prediction time: the set of issues which have been completed, the set of work-in-progress issues (i.e. issues that have been acted upon but not yet completed), and the set of issues on which the team has not started working yet (i.e. to-do issues). Agile practices also suggest using this approach for monitoring the progress of an iteration [13].

Figure 4 shows an example of an on-going iteration report (recorded in JIRA-Agile) of *Mesosphere Sprint 34*[3] in the Apache project. This iteration started from April 27, 2016 to May 11, 2016. This iteration has two issues in the Todo state (*MESOS-5272* and *MESOS-5222*), three issues in the In-progress state – those are all in the reviewing process (*MESOS-3739*, *MESOS-4781*, and *MESOS-4938*), and one issue has been resolved (*MESOS-5312*). These issues have story points assigned to them. For each of those sets of issues, we compute the set cardinality and velocity, and use each of them as a feature. From our investigation, among the under-achieved iterations across all case studies, i.e. $velocity(\texttt{Difference}) < 0$, 30% of them have new issues added after passing 80% of their planned duration (e.g. after 8^{th} day of a ten-day iterations), while those iterations deliver zero-issue. Specifically, teams added more $velocity(\texttt{Committed})$ while $velocity(\texttt{Delivered})$ was still zero. This reflects that adding and removing issues affects the deliverable capability of an on-going iteration. This can be a good indicator to determine the outcome of an iteration.

These features were extracted by examining the list of complete/incomplete issues of an iteration and the change log of an issue, e.g. which iteration an issue was added or removed on which date, and its status (e.g. an issue is in the set of work-in-progress issues or in the set of to-do issues) at prediction time. Figure 5 shows an example of an iteration report for the iteration named *Twitter Aurora Q2′ 15 Sprint 3* in the Apache project. The report provides a list of completed issues (e.g. *AURORA-274*) and uncompleted issues (e.g. *AURORA-698*), a list of added and removed issues during an iteration (e.g. *AURORA-1267*), and iteration details (e.g. state, start date, and planned end date). We can also identify when those issues were added or removed from the iteration by examining their change logs. Figure 6 shows an example of a change log of issue *AURORA-1267* which records that this issue has been added to *Twitter Aurora Q2′ 15 Sprint 3* on May 16, 2015 while this iteration was started one day earlier (i.e. May 15, 2015).

3. https://issues.apache.org/jira/secure/RapidBoard.jspa?rapidView=62

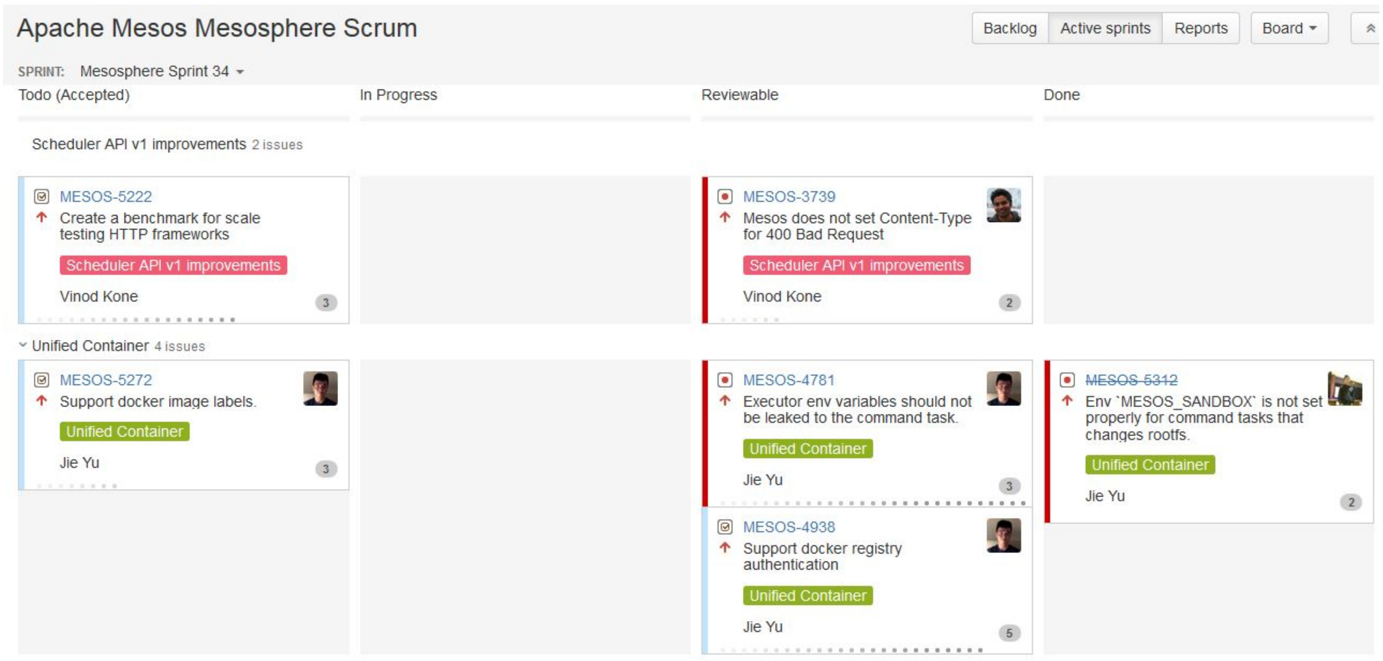
TABLE 1
Features of an iteration

Feature	Description
Iteration duration	The number of days from the start date to planned completion date
No. of issues at start time	The number of issues assigned to an iteration at the beginning
Velocity at start time	The sum of story points of issues assigned to an iteration at the beginning
No. of issues added	The number of issues added during an iteration (between start time and prediction time)
Added velocity	The sum of story points of issues added during an iteration (between start time and prediction time)
No. of issues removed	The number of issues removed during an iteration (between start time and prediction time)
Removed velocity	The sum of story points of issues removed during an iteration (between start time and prediction time)
No. of to-do issues	The number of to-do issues in an iteration by prediction time
To-do velocity	The sum of story points of to-do issues by prediction time
No. of in-progress issues	The number of in-progress issues in an iteration by prediction time
In-progress velocity	The sum of story points of in-progress issues by prediction time
No. of done issues	The number of done issues in an iteration by prediction time
Done velocity	The sum of story points of done issues by prediction time
Scrum master	The number of Scrum masters
Scrum team members	The number of team members working on an iteration

There are a number of features reflecting the team involved in an iteration. These include the number of team leads (e.g. Scrum masters) and the size of the team. Note that the team structure information is not explicitly recorded in most issue tracking systems. We thus conjecture that the number of team members is the number of developers assigned to issues in an iteration. The number of Scrum masters is the number of authorized developers who can manage issues (e.g. add, remove) in an iteration. Future work could look at other characteristics of a team including the expertise and reputation of each team member and the team structure.

4.2 Features of an issue

The issues assigned to an iteration also play an important part in characterizing the iteration. Figure 7 shows an example of an issue report of issue *AURORA-716* in the Apache project which the details of an issue are provided such as type, priority, description, and comments including a story points and an assigned iteration. Hence, we also extract a broad range of features representing an issue (see Table 2). The features cover different aspects of an issue including primitive attributes of an issue, issue dependency, changing of issue attributes, and textual features of an issue's description. Some of the features of an issue (e.g. number of issue links) were also adopted from our previous work [14].


Apache Mesos Mesosphere Scrum

Backlog | Active sprints | Reports | Board

SPRINT: Mesosphere Sprint 34

Todo (Accepted) | In Progress | Reviewable | Done

Scheduler API v1 improvements 2 issues

MESOS-5222
Create a benchmark for scale testing HTTP frameworks
Scheduler API v1 improvements
Vinod Kone 3

MESOS-3739
Mesos does not set Content-Type for 400 Bad Request
Scheduler API v1 improvements
Vinod Kone 2

Unified Container 4 issues

MESOS-5272
Support docker image labels.
Unified Container
Jie Yu 3

MESOS-4781
Executor env variables should not be leaked to the command task.
Unified Container
Jie Yu 3

MESOS-4938
Support docker registry authentication
Unified Container
Jie Yu 5

~~MESOS-5312~~
Env `MESOS_SANDBOX` is not set properly for command tasks that changes rootfs.
Unified Container
Jie Yu 2

Fig. 4. An example of an on-going iteration report

```
{"contents":{
 "completedIssues":[
   {"id": 12702203,
    "key": "AURORA-274"},
   {"id": 12724064,
    "key": "AURORA-556"},
   {"id": 12769421,
    "key": "AURORA-1047"},...],
 "incompletedIssues":[
   {"id": 12740639,
    "key": "AURORA-698"},...],
 "puntedIssues": [], ...
 "issueKeysAddedDuringSprint": {
   "AURORA-1267": true,
   "AURORA-1321": true,...}},
 "sprint": {
   "id": 127,
   "sequence": 127,
   "name": "Twitter Aurora Q2'15 Sprint 3",
   "state": "CLOSED",
   "startDate": "12/May/15 6:59 AM",
   "endDate": "26/May/15 4:00 AM",...}}
```

Fig. 5. Example of an iteration report in JSON format of the iteration named *"Twitter Aurora Q2'15 Sprint 3"* in the Apache project

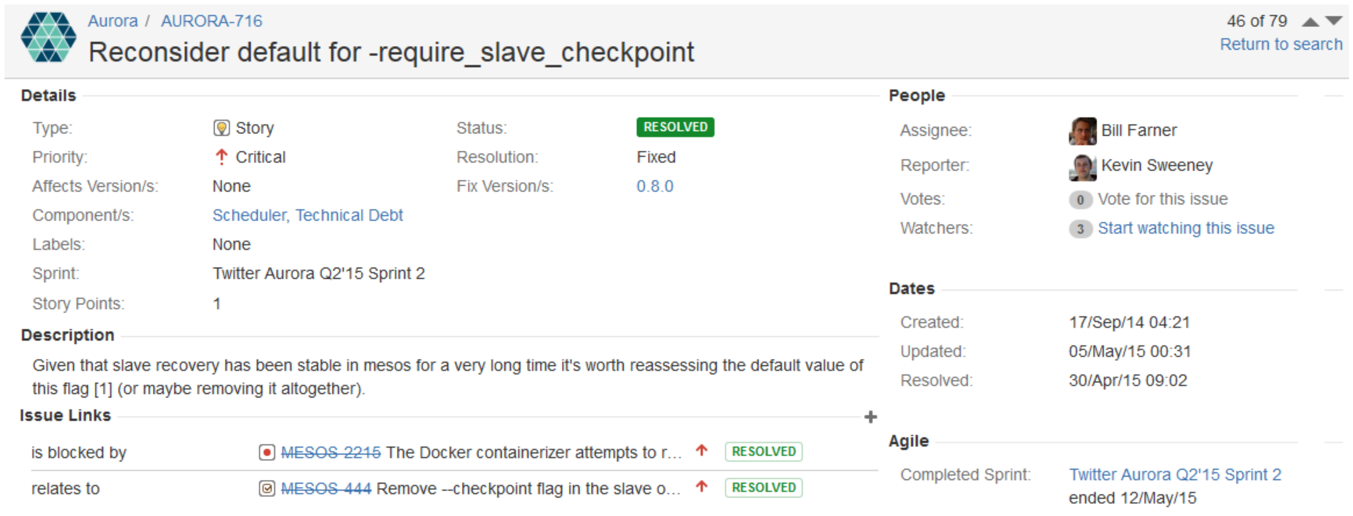
```
{"key": "AURORA-1267",
   "changelog": { "histories": [...
  {"id": "14758778",
   "created": "2015-05-16T00:31:55.018+0000",
   "items": [{
   "field": "Sprint",
   "fieldtype": "custom",
   "from": null,
   "fromString": null,
   "to": "127",
   "toString":"Twitter Aurora Q2'15 Sprint 3"}
   ]},...]}}
```

Fig. 6. Example of a change log of an issue in JSON format of issue *AURORA-1267*

It is important to note that we used the time when a prediction is made (prediction time t_{pred}) as the reference point when extracting the values of *all* the features. By processing an issue's change log during both training and testing phases we collected the value which a feature had just before the prediction time. For example, if the final number of comments on an issue is 10, and there were no comments at the time when the prediction was made, then the value of this feature is 0. The underlying principle here is that: when making a prediction, we try to use only information available just *before* the prediction time. The purpose of doing this is to prevent using "future" data when making a prediction – a phenomenon commonly referred in machine learning as information leakage [15].

4.2.1 Primitive attributes of an issue

These features are extracted directly from the issue's attributes, which include type, priority, number of comments, number of affect versions, and number of fix versions. Each issue will be assigned a type (e.g task, bug, new feature, improvement, and story) and a priority (e.g. minor, major, and critical). These indicate the nature of the task associated with resolving the issue (e.g. new feature implementation or bug fixing) and the order in which an issue should be attended with respect to other issues (e.g. a team should concerns critical priority issues more than issues with major and minor priority). These attributes might affect the delivery of an iteration. For example, in the Apache project approximately 10% of the under-achieved iterations have at least one *critical* priority issue.


Aurora / AURORA-716

Reconsider default for -require_slave_checkpoint

46 of 79

Return to search

Details

Type:	Story	Status:	RESOLVED
Priority:	Critical	Resolution:	Fixed
Affects Version/s:	None	Fix Version/s:	0.8.0
Component/s:	Scheduler, Technical Debt		
Labels:	None		
Sprint:	Twitter Aurora Q2'15 Sprint 2		
Story Points:	1		

Description

Given that slave recovery has been stable in mesos for a very long time it's worth reassessing the default value of this flag [1] (or maybe removing it altogether).

Issue Links

is blocked by	~~MESOS-2215~~ The Docker containerizer attempts to r...	RESOLVED
relates to	~~MESOS-444~~ Remove --checkpoint flag in the slave o...	RESOLVED

People

Assignee:	Bill Farner
Reporter:	Kevin Sweeney
Votes:	0 Vote for this issue
Watchers:	3 Start watching this issue

Dates

Created:	17/Sep/14 04:21
Updated:	05/May/15 00:31
Resolved:	30/Apr/15 09:02

Agile

Completed Sprint:	Twitter Aurora Q2'15 Sprint 2 ended 12/May/15

Fig. 7. An example of an issue report of issue *AURORA-716* in the Apache project

Previous studies (e.g. [16]) have found that the number of comments on an issue indicates the degree of team collaboration, and thus may affect its resolving time. The "affect version" attribute of an issue specifies versions in which an issue (e.g. bug) has been found, while the "fix version" attribute indicates the release version(s) for which the issue was (or will be) fixed. One issue can be found in many versions. An issue with a high number of affect versions and fix versions needs more attention (e.g. an intensive reviewing process) to ensure that the issue is actually resolved in each affect version and does not cause new problems for each fix version. For example, in Apache, we found that 80% of the over-achieved iterations have issues assigned to only one fix version. We also found that 75.68% of the issues were assigned at least one affect version and fix version.

4.2.2 *Dependency of an issue*

We extract the dependency between the issues in a term of the number of issue links. Issue linking allows teams to create an association between issues (e.g. an issue resolution may depend on another issue). Figure 7 also shows an example of issue links of issue *AURORA-716* in the Apache project for which it has been blocked by issue *AURORA-MESOS-2215*. There are several relationship types for issue links (e.g. relate to, depend on). In our approach we consider all types of issue links and use the number of those links as features. Moreover, blocker is one of the issue linking types that indicates the complexity of resolving issue since these blocker issues block other issues from being completed (i.e. all blocker issues need to be fixed beforehand). As such they directly affects the progress and time allocated to solve other issues [17], [18], [19]. The blocker relationship is thus treated separately as two features: number of issues that are blocked by this issue and number of issues that block this issue. We found that there are more than 30% of issues have at least one relationship. We also note that when counting the number of links between issues, we count each link type separately. For example, if there are three different types of links between issues A and B, the number of links counted would be 3.

4.2.3 *Changing of issue attributes*

Previous research, e.g., [19], [20], has shown that changing of an issue's attribute (e.g. priority) may increase the issue resolving time and decrease the deliverable capability which it could be a cause of delays in software project. In our study, there are three features reflecting changing of issue's attributes which are: the number of times an issue priority was reassigned, the number of times in which a fix version was changed, and the number of times in which an issue description was changed. The changing of an issue's priority may indicate the shifting of its complexity. In addition, the changing of the fix version(s) reflects some changes in the release planning for which it affects directly the planning of on-going iterations. In particular, changing the description of an issue could indicate that the issue is not stable and could also create misunderstanding. These may consequently have an impact on the issue resolution time.

4.2.4 *Textual features of an issue's description*

An issue's description text can provide good features since it explains the nature of an issue. A good description helps the participant of an issue understand its nature and complexity. We have a employed readability measure to derive textual features from the textual description of an issue. We used Gunning Fox [21] to measure the complexity level of the description in terms of a readability score (i.e. the lower score, the easier to read). Previous studies (e.g. [22]) have found that issues with high readability scores were resolved more quickly. We acknowledge that there are more advanced techniques with which to derive features from textual data (e.g. word2vec), use of which we leave for our future work.

4.3 Feature Aggregation

As previously stated, to characterize an iteration, we extracted both the features of an iteration and the features of issues assigned to it (i.e. one iteration associates with a number of issues). Feature aggregation derives a new set of features from the issues (i.e. a number of issues) for

TABLE 2
Features of an issue

Feature	Description
Type	Issue type
Priority	Issue priority
No. of comments	The number of comments
No. of affect versions	The number of versions for which an issue has been found
No. of fix versions	The number of versions for which an issue was or will be fixed
Issue links	The number of dependencies of an issues
No. of blocking issues	The number of issues that block this issue for being resolved
No. of blocked issues	The number of issues that are blocked by this issue
Changing of fix versions	The number of times in which a fix version was changed
Changing of priority	The number of times an issue's priority was changed
Changing of description	The number of times in which an issue description was changed
Complexity of description	The read ability index (Gunning Fog [21]) indicates the complexity level of a description which is encoded to easy and hard

TABLE 3
Statistical aggregated features for an issue's feature k

Function	Description
min	The minimum value in V_k
max	The maximum value in V_k
mean	The average value across V_k
median	The median value in V_k
std	The standard deviation of V_k
var	The variance of V_k (measures how far a set of numbers is spread out)
range	The difference between the lowest and highest values in V_k
frequency	The summation of the frequency of each categorical value

an iteration by aggregating those features of the issues assigned to the iteration. We discuss here two distinct feature aggregation techniques (i.e. statistical aggregation and Bag-of-Words) we use to characterize an iteration using features of the issues assigned to it. The complexity descriptors of a graph describing the dependencies among those issues also form a set of features representing the iteration. These aggregation approaches aim to capture the characteristics of issues associated to an iteration in different aspects which each of them could reflects the situation of an iteration. The features of an iteration and its aggregated features of the issues are then fed into a classifier to build a predictive model that we discuss in Section 5.

4.3.1 Statistical aggregation

Statistical aggregation aggregates the features of issues using a number of statistical measures (e.g. max, min, and mean) which aims to capture the statistical characteristics of the issues assigned to an iteration. For each feature k in the set of features of an issue (see Table 2), we have a set of values $V_k = \{x_1^k, x_2^k, ..., x_n^k\}$ for this feature where x_i^k ($i \in [1..n]$) is the value of feature k of issue x_i in an iteration, and n is the number of issues assigned to this iteration. Applying different statistics over this set V_k (e.g. min, max, mean, median, standard deviation, variance, and frequency) gives different aggregated features. Table 3 shows eight basic statistics that we used for our study. Note that the categorical features (e.g. type and priority) are aggregated by summing over the occurrences of each category. For example, minimum, maximum, mean, and standard deviation of number of comments, and frequency of each type (e.g. number of issues in an iteration that are "bug" type) are the features among the aggregated features that characterize an iteration.

4.3.2 Feature aggregation using Bag-of-Words

The above-mentioned approach require us to *manually* engineer and apply a range of statistics over the set of issues in an iteration in order to derive new features characterizing the iteration. Our work also leverages a new feature learning approach known as Bag-of-Words, which has been widely used in computer vision for image classification (e.g. [23]), to obviate the need for manual feature engineering. Here, new features for a project's iteration can be *automatically* learned from all issues in the project. We employ unsupervised K-means clustering to learn features for an iteration. Specifically, we apply K-means clustering to all the issues extracted from a given project, which gives us k issue clusters whose centers are in $\{C_1, C_2, ..., C_k\}$. The index of the closest center to each issue in an iteration forms a word in a bag-of-words representing the iteration. The occurrence count of a word is the number of issues closest to the center associated with the word. For example, assume that an iteration has three issues X, Y and Z, and the closest cluster center to X is C_1 while the closest center to Y and Z is C_2. The bag-of-words representing this iteration is a vector of occurrence counts of the cluster centers, which in this case is 1 for C_1, 2 for C_2 and 0 for the remaining clusters' centers. Note that in our study we derived 100 issue clusters ($k = 100$) using *kmeans*[4] package from Matlab. Future work would involve evaluation using a different number of issue clusters.

This technique provides a powerful abstraction over a large number of issues in a project. The intuition here is that the number of issues could be large (hundreds of thousands to millions) but the number of issue types (i.e. the issue clusters) can be small. Hence, an iteration can be characterized by the types of the issues assigned to it. This approach offers an efficient and effective way to learn new features for an iteration from the set of issues assigned to it. It can also be used in combination with the statistical aggregation approach to provide an in-depth set of features for an iteration.

4.3.3 Graph descriptors

Dependencies often exist between issues in an iteration. These dependency of issues are explicitly recorded in the form of issue links (e.g. relate to, depend on, and blocking). Blocking is a common type of dependency that is recorded in issue tracking systems. For example, blocking issues are those that prevent other issues from being resolved. Such dependencies form a directed acyclic graph (DAG) which depicts how the work on resolving issues in an iteration

4. http://au.mathworks.com/help/stats/kmeans.html

should be scheduled (similar to the popular activity precedence network in project management). Figure 8 shows an example of a DAG constructed from nine issues assigned to the iteration *Usergrid 20*[5] in the Apache project. However note that we consider only the relationships among issues in the same iteration.

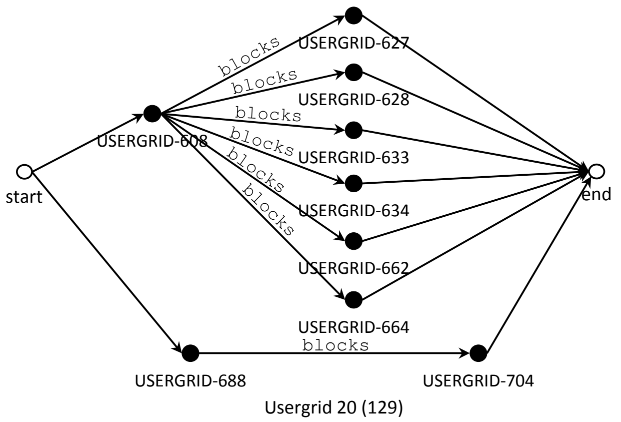


Fig. 8. Example of a DAG of issues in the iteration "Usergrid 20" in the Apache project

The descriptors of complexity of such a DAG of issues provide us with a rich set of features characterizing an iteration. These include basic graph measures such as the number of nodes in a graph, the number of edges, the total number of incoming edges, and so on (see [24] for a comprehensive list). Table 4 lists a set of graph-based features that we currently use. For example, from Figure 8, among the aggregated features using graph-based feature aggregation for the iteration *Usergrid 20*, the number of nodes equals 9 and the number of edges equals 7. We acknowledge that the dependencies between issues in an iteration may not exist (e.g. no issue link between issues in an iteration) however the aggregated features from this approach can be combined with the features from our other techniques to characterize an iteration in terms of dependencies between issues assigned to it. Future work would involve exploring some other measures such as graph assortativity coefficient [25].

5 Predictive Models

Our predictive models can predict the difference between the actual delivered velocity against the committed (target) velocity for an iteration, i.e. $velocity(\texttt{Difference})$. To do so, we employ regression methods (supervised learning) where the outputs reflect the deliverable capability in an iteration e.g., the predicting of $velocity(\texttt{Difference})$ will be equal to 12. The extracted features of the historical iterations (i.e. training set) are used to build the predictive models. Specifically, a feature vector of an iteration and an aggregated feature vector of issues assigned to the iteration are concatenated and fed into a regressor.

5. https://issues.apache.org/jira/secure/RapidBoard.jspa?rapidView=23&view=reporting&chart=sprintRetrospective&sprint=129

TABLE 4
Features of a DAG of issues in an iteration

Graph measure	Description
number of nodes	The number of issues in DAG
number of edges	The number of links between issues in DAG
sum of fan in	The total number of incoming links of issues in DAG
sum of fan out	The total number of outgoing links of issues in DAG
max of fan in	The maximum number of incoming links of issues in DAG
max of fan out	The maximum number of outgoing links of issues in DAG
mean of fan in	The average of numbers of incoming links across all issues in DAG
mean of fan out	The average of numbers of outgoing links across all issues in DAG
mode of fan in	The number of incoming links that appear most often in DAG
mode of fan out	The number of outgoing links that appear most often in DAG
avg. node degree	The degree distribution of nodes and edges of DAG

We apply the currently most successful class of machine learning methods, namely *randomized ensemble methods* [26], [27], [28]. Ensemble methods refer to the use of many regressors to make their prediction [28]. Randomized methods create regressors by randomizing data, features, or internal model components [29]. Randomizations are powerful regularization techniques which reduce prediction variance, prevent overfitting, are robust against noisy data, and improve the overall predictive accuracy [30], [31].

We use the following high performing regressors that have frequently won recent data science competitions (e.g. Kaggle[6]): *Random Forests* (RFs) [32], *Stochastic Gradient Boosting Machines* (GBMs) [33], [34] and *Deep Neural Networks with Dropouts* (DNNs) [35]. All of them are ensemble methods that use a divide-and-conquer approach to improve performance. The key principle behind ensemble methods is that a group of "weak learners" (e.g. classification and regression decision trees) can together form a "strong learner". Details for these predictive models used are provided in Appendix A.

The source code in Matlab of the three feature aggregation techniques: Statistical aggregation, Bag-of-Words, and Graph-based feature aggregation, and the three randomized ensemble methods: Random Forests, Stochastic Gradient Boosting Machines, and Deep Neural Networks with Dropouts have been made available online at http://www.dsl.uow.edu.au/sasite/index.php/agile/.

6 Evaluation

This section discusses an extensive evaluation that we have carried out of our approach. We describe how data is collected and processed for our study, the experimental setting, discuss the performance measures, and report our results. Our empirical evaluation aims to answer the following research questions:

RQ1 *Do the feature aggregation approaches improve the predictive performance?*

6. https://www.kaggle.com

We build predictive models using the three feature aggregation approaches and compare them against a predictive model using only the information extracted directly from the attributes of an iteration (i.e. features of an iteration). This is to evaluate whether the aggregated features of issues offer significant improvement. We also investigate which combinations of the feature aggregation approaches are the best performer (e.g. combining the aggregated features from statistical feature aggregation and Bag-of-words aggregation approach).

RQ2 *Do randomized ensemble methods improve the predictive performance compared to a traditional regression model?*
We employ Support Vector Machine (SVM) as a representative for traditional regression models. SVM is the most important (deterministic) classifier in the year 2000s [36]. Its predictive power has been challenged, only recently, by randomized and ensemble techniques (e.g, see the recent comparative study proposed by Fernández-Delgado et al [37]). SVM has been widely used in software analytics, such as defect prediction, effort estimation, and bug localization. Its regression version, Support Vector Regression (SVR), is also known for being highly effective for regression problems. Thus, we employ SVR to build predictive models to evaluate whether our randomized ensemble methods perform better than the traditional regression model. We also find the best randomized ensemble method in predicting the difference between actual achieved and target velocity in an iteration.

RQ3 *Are the purposed randomized ensemble method and the feature aggregation suitable for predicting the difference between the actual delivered velocity against the target velocity of an iteration?*
This is our sanity check as it requires us to compare our purposed prediction model with three common baseline benchmarks used in the context of effort estimation: Random Guessing, Mean Effort, and Median Effort. Random guessing performs random sampling (with equal probability) over the set of iterations with known difference (between target and actual achieved velocity), chooses randomly one iteration from the sample, and uses the target vs. actual difference velocity of that iteration as the prediction of the new iteration. Random guessing does not use any information associated with the new iteration. Thus any useful prediction model should outperform random guessing. Mean and Median Effort predictions are commonly used as baseline benchmarks for effort estimation. They use the mean or median target vs. actual difference of the past iterations to predict the difference of the new iterations.

RQ4 *Does the time of making a prediction (t_{pred}) affect the predictive performance?*
We want to evaluate the predictive performance from the different prediction times (t_{pred}) to confirm our hypothesis that the later we predict, the more accuracy we gain. Specifically, we evaluate the predictive performance from four different prediction times: at the beginning of an iteration, and when it progresses to 30%, 50%, and 80% of its planned duration. We acknowledge that making a prediction as late as at 80% of an iteration duration may not be particularly useful in practice. However, for the sake of completeness we cover this prediction time to sufficiently test our hypothesis. Note that our experiments in RQ1-RQ3 were done at the prediction time when an iteration has progressed to 30% of its planned duration (e.g. make a prediction at the third day of a 10-day iteration).

RQ5 *Can the output from the predictive model (i.e. the difference between the actual delivered velocity against the target velocity) be used for classifying the outcomes of an iteration (e.g. an under-achieved iteration)?*
Rather than the difference between the actual achieved and the target velocity, the outcomes of iterations can also be classified into three classes: below the target – *under achieved*, i.e. $velocity(\texttt{Committed}) > velocity(\texttt{Delivered})$, or above the target – *over achieved*, i.e. $velocity(\texttt{Committed}) < velocity(\texttt{Delivered})$, or the same as the target – *achieved*, i.e. $velocity(\texttt{Committed}) = velocity(\texttt{Delivered})$.
We want to evaluate whether the output from the predictive models (i.e. the difference velocity) can be used to classify the outcome of an iteration in terms of the three classes: negative outputs, i.e. $velocity(\texttt{Difference}) < 0$, are in the *under-achieved* class, positive outputs, i.e. $velocity(\texttt{Difference}) > 0$, are in the *over-achieved* class, and zero, i.e. $velocity(\texttt{Difference}) = 0$, is in the *achieved* class. This method can also accommodate a tolerance margin e.g. outputs from -1 to 1 are considered in the achieved class. A finer-grained taxonomy (e.g. different levels of over achieved or under achieved) for classifying iterations can also be accommodated to reflect the degree of difference between the target and the actual delivered velocity.

6.1 Data collecting and processing

We collected the data of past iterations (also referred to as sprints in those projects) and issues from five large open source projects which follow the agile Scrum methodology: Apache, JBoss, JIRA, MongoDB, and Spring. The project descriptions and their agile adoptions have been reported in Table 5.

All five projects use *JIRA-Agile* for their development which is a well-known issue and project tracking tool that supports agile practices. We use the Representational State Transfer (REST) API provided by JIRA to query and collect iteration and issue reports in JavaScript Object Notation (JSON) format. Note that the JIRA Agile plug-in supports both the Scrum and Kanban practices, but we collected only the iterations following the Scrum practice. From the JIRA API, we were also able to obtain issues' change log and the list of complete/incomplete issues of each iteration.

We initially collected 4,301 iterations from the five projects and 65,144 issues involved with those iterations from *February 28, 2012* to *June 24, 2015*. The former date is the date when the first iteration (among all the case studies)

TABLE 5
Project description

Project	Brief description	Agile adoption	Year of adoption	Repository
Apache	A web server originally designed for Unix environments. There are more than fifty sub-projects under the Apache community umbrella (i.e. Aurora, Apache Helix, and Slider).	Almost 50% of the issues in the sub-projects that are applied the Scrum method (e.g. Aurora) have been assigned to at least one iteration.	2006	https://issues.apache.org/jira/
JBoss	An application server program which supports a general enterprise software development framework.	JBoss community has described their iterative development process in the developer guide.$^+$ There are more than ten sub-projects that are applied agile methods.	2004*	https://issues.jboss.org/
JIRA	A project and issue tracking system including the agile plug-in that provides tools to manage iterations following agile approaches (e.g. Scrum, Kanban) provided by Atlassian.	Recently, Atlassian reported their success in applying agile methods to improve their development process.^{x}	N/A	https://jira.atlassian.com/
MongoDB	A cross platform document-oriented database (NoSQL database) provided by MongoDB corporation and is published as free and open-source software.	More than 70% of the issues in the main sub-projects (e.g. Core MongoDB, MongoDB tool) have been assigned to at least one iteration	2009	https://jira.mongodb.org/
Spring	An application development framework that contains several sub-projects in their repository i.e. Spring XD and Spring Data JSP.	Almost 70% of the issues in the core sub-projects (e.g. Spring XD) have been assigned to at least one iteration.	N/A	https://jira.spring.io/

$^+$:http://docs.jboss.org/process-guide/en/html/, *: identified from the developer guided published date,
^{x}:http://www.businessinsider.com.au/atlassian-2016-software-developer-survey-results-2016-3

TABLE 6
Descriptive statistics of the iterations of the projects in our datasets

Project	# iterations	# issues	# days/iteration				# issues/iteration				# team members /iteration			
			min/max	mean	median	SD	min/max	mean	median	SD	min/max	mean	median	SD
Apache	348	5,826	3/21	12.02	14	10.12	3/128	15.39	7	8.24	3/21	4.6	4	3.4
JBoss	372	4,984	3/49	12.52	13	7.4	4/122	7.14	5	8.54	2/20	2.6	2	3.11
JIRA	1,873	10,852	4/60	10.5	9	6.03	3/88	5.5	5	6.2	2/12	3.71	2	2.07
MongoDB	765	17,528	3/43	20	18	36	3/180	15	9	21.4	2/30	4.12	3	5.02
Spring	476	17,497	3/49	14.28	14	7.21	3/161	20.71	17	20.11	3/15	5.4	4	4.37
Total	3,834	56,687												

# iterations: number of iterations, # issues: number of issues, # days/iteration: number of days per iteration,
# issues/iteration: number of issues per iteration, # team members/iteration: number of team members per iteration

was created and the later is the date when we finished collecting the data. Hence, this window ensures that we did not miss any data up to the time of our collection. The data was preprocessed by removing duplicate and demonstration iterations as well as future and ongoing iterations. We also removed iterations that have zero resolved issues where those iterations are ignored by teams (e.g. no activity recorded in the issue report, all issues have been removed).

In total, we performed our study on 3,834 iterations from the five projects, which consist of 56,687 issues. Table 6 shows the number of iterations and issues in our datasets and summarizes the characteristics of the five projects in terms of the iteration length, number of issues per iteration, number of team members per iteration in terms of the minimum, maximum, mean, median, and standard deviations (STD). The iteration length across the five projects tends to be in the range of 2 to 4 weeks. All projects have almost the same team size, while the number of issues per iteration varies. For example, the mean number of issues per iteration in MongoDB is 15 issues, while it is only 5.5 issues in JIRA. We have made our dataset publicly available at http://www.dsl.uow.edu.au/sasite/index.php/agile/.

6.2 Experimental setting

All iterations and issues collected in each of the five case studies were used in our evaluations. As discussed in Section 2, we would like to predict the difference between the actual delivered velocity against the target velocity. For example, if the output of our model is -5, it predicts that the team will deliver 5 story points below the target. Table 7 shows the statistical descriptions of the difference between the actual delivered against the target velocity of the five projects in terms of the minimum, maximum, mean, median,

TABLE 7
Descriptive statistics of the difference between the actual delivered velocity against the target velocity in each project

Project	$velocity(\texttt{Difference})$						
	min	max	mean	median	mode	SD	IQR
Apache	-81	49	-9.05	-5	0	17.16	15.00
JBoss	-96	30	-6.03	-1	0	14.83	4.00
JIRA	-83	117	-2.82	0	0	11.37	3.00
MongoDB	-67	50	-0.99	0	0	11.54	5.00
Spring	-135	320	23.77	8.50	4	51.66	32.00

mode, standard deviations (SD), and interquartile range (IQR).

We used ten-fold cross validation in which all iterations were sorted based on their start date. After that, an iteration i^{th} in every ten iterations is included in fold i^{th}. With larger data sets, we could choose the sliding window setting, which mimics a real deployment scenario, to ensure that prediction on a current iteration is made by using knowledge from the past iterations. We leave for future work since we believe that our findings in the current experimental setting still hold [38]. The time when the prediction is made may affect its accuracy and usefulness. The later we predict, the more accurate our prediction gets (since more information has become available) but the less useful it is (since the outcome may become obvious or it is too late to change the outcome). We later address the varying of the prediction time in RQ4.

6.3 Performance Measure

We adapted a recently recommended measure: Mean Absolute Error (MAE) [6] to measure the predictive performance of our models in a term of the errors between the actual and the predicted $velocity(\texttt{Difference})$. However, different projects have different $velocity(\texttt{Difference})$ ranges (see Table 7). Thus, we needed to normalize the MAE (by dividing it with the interquartile range) to allow for comparisons of the MAE across the studied project. Similarly to the work in [39], we refer to this measure as Normalized Mean Absolute Error (NMAE). To compare the performance of two predictive models, we also applied statistical significance testing on the absolute errors predicted by the two models using the Wilcoxon Signed Rank Test [40] and employed the Vargha and Delaneys $\hat{A}_{12}$ statistic [41] to measure whether the effect size is interesting. Later in RQ5, we used a number of traditional metrics: precision, recall, F-measure, Area Under the ROC Curve (AUC) to measure the performance in classifying the outcome of an iteration. We also used another metric called Macro-averaged Mean Absolute Error (MMAE) [42] to assess the distance between actual and predicted classes since the classes is ordinal, we can order them, e.g. *over achieved* is better than *achieved*. The traditional class-based measures (Precision/Recall) do not take into account the ordering between classes, and Matthews Correlation Coefficient (MCC) [43], which performs well on imbalanced data. Details of these measures and our statistical testing used are provided in Appendix B.

6.4 Results

We report here our evaluation results in answering our research questions RQs 1-5.

6.4.1 *Benefits of the feature aggregations of issues **(RQ1)***

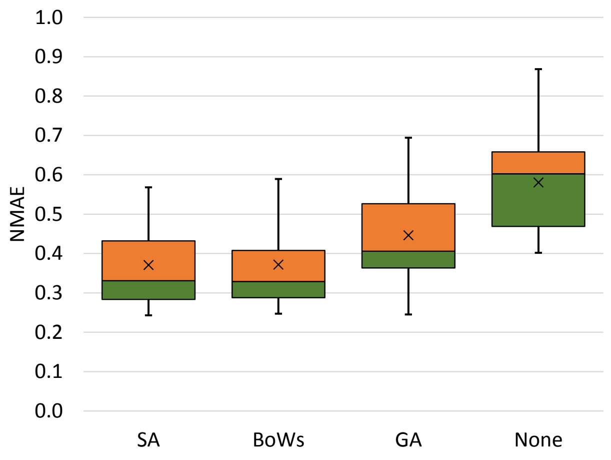


Fig. 9. Evaluation results on the three aggregated features and the features of iterations (None). X is a mean of NMAE averaging across all projects and all regression models (the lower the better).

We compared the predictive performance (using NMAE) achieved for the three feature aggregation approaches: statistical aggregation (SA), Bag-of-Words (BoWs), and graph-based aggregation (GA) against the predictive model using only the features of an iteration (None). Figure 9 shows the NMAE achieved for each of the aggregated features, averaging across all the projects and all the regression models. **The analysis of NMAE suggests that the predictive performance achieved from using the feature aggregation approaches (i.e. SA, BoWs, and GA) consistently outperforms the predictive model using only the features of iterations (i.e. None).** The predictive models using statistical aggregation, Bag-of-Words, and graph-based aggregation achieve an accuracy of 0.371, 0.372, and 0.446 NMAE respectively, while the predictive models using only the features of iterations achieve only 0.581 NMAE.

Table 8 shows the results of the Wilcoxon test (together with the corresponding A^{12} effect size) to measure the statistical significance and effect size (in brackets) of the improved accuracy achieved by the aggregated features over the features of iterations. In *all* cases, the predictive models using the feature aggregation approaches significantly outperform the predictive models using only the features of iterations ($p < 0.001$) with effect size greater than 0.5.

We also performed a range of experiments to explore the best combination of aggregated features. There are four possible combinations: SA+GA, BoWs+GA, SA+BoWs, and the combination of all of them (All). For example, SA+GA combines a feature vector of an iteration, a feature vector of issues obtained from statistical aggregation (SA), and a feature vector of issues obtained from graph-based aggregation (GA). As can be seen in Figure 10, in most cases the combination of two or more feature aggregation approaches produced better performance than a single aggregation approach. **However, the best performer varies**

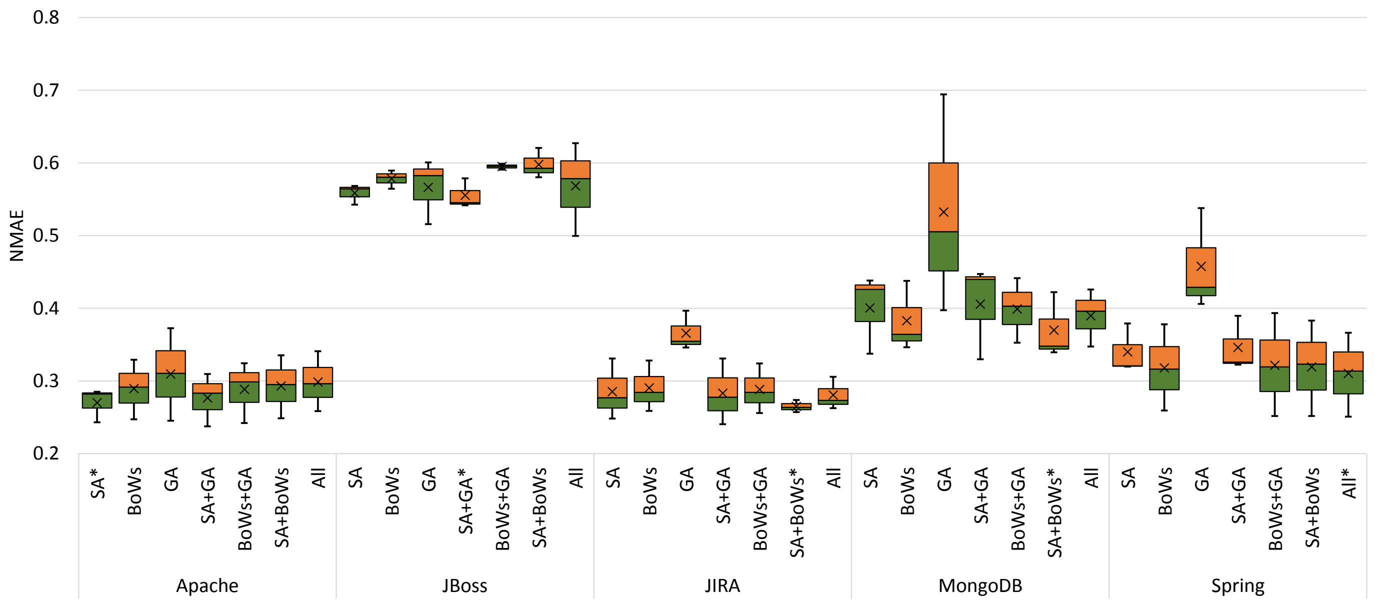


Fig. 10. Evaluation results on all the combinations of the aggregated features of issues. In each project, the best performer is marked with *. X is a mean of NMAE averaging across all regression models (the lower the better).

TABLE 8
Comparison of the predictive models between with and without the aggregated features using Wilcoxon test and A^{12} effect size (in brackets)

Project	With agg. vs	Without agg.	
Apache	SA	<0.001	[0.61]
	BoWs	<0.001	[0.60]
	GA	<0.001	[0.56]
JBoss	SA	<0.001	[0.59]
	BoWs	<0.001	[0.60]
	GA	<0.001	[0.54]
JIRA	SA	<0.001	[0.62]
	BoWs	<0.001	[0.56]
	GA	<0.001	[0.52]
MongoDB	SA	<0.001	[0.61]
	BoWs	<0.001	[0.60]
	GA	<0.001	[0.53]
Spring	SA	<0.001	[0.60]
	BoWs	<0.001	[0.61]
	GA	<0.001	[0.54]

between projects. For example, SA+GA outperforms the others in JBoss – it achieves 0.555 NMAE while the others achieve 0.558 - 0.598 NMAE (averaging across all regression models), while SA+BoWs is the best performer in JIRA – it achieves 0.265 NMAE while the others achieve 0.285 - 0.366 NMAE (averaging across all regression models). These results suggest that the three approaches are distinct and complementary to each other: the statistical aggregation covers the details, the Bag-of-Words technique addresses the abstraction, while the graph-based approach reflects the network nature of issues in an iteration as also shown in our previous work [18]. We rely not just on features coming directly from the attributes of an object (i.e. an iteration), like most of existing software analytics approaches, but also the features from the parts composing the object (i.e. the issues). The latter features are powerful since they are automatically learned and aggregated, and capture the graphical structure of the object.

Answer to RQ1: Feature aggregation offers significant improvement in predictive performance.

6.4.2 *Benefits of the randomized ensemble methods **(RQ2)***

TABLE 9
Comparison of the three randomized ensemble methods against the traditional SVR using Wilcoxon test and A^{12} effect size (in brackets)

Project	Method vs	SVR	
Apache	RF	<0.001	[0.61]
	Deep Nets.	<0.001	[0.61]
	GBMs	<0.001	[0.67]
JBoss	RF	<0.001	[0.58]
	Deep Nets.	<0.001	[0.59]
	GBMs	<0.001	[0.66]
JIRA	RF	<0.001	[0.63]
	Deep Nets.	<0.001	[0.63]
	GBMs	<0.001	[0.71]
MongoDB	RF	<0.001	[0.66]
	Deep Nets.	<0.001	[0.70]
	GBMs	<0.001	[0.82]
Spring	RF	<0.001	[0.64]
	Deep Nets.	<0.001	[0.66]
	GBMs	<0.001	[0.73]

To answer RQ2, we focus on the predictive performance achieved from different regression models. For a fair comparison we used only one combination of aggregated

TABLE 10
Comparison of GBMs against RF and Deep nets. using Wilcoxon test and A^{12} effect size (in brackets)

Project	Method vs	RF		Deep Nets.	
Apache	GBMs	<0.001	[0.62]	<0.001	[0.60]
JBoss	GBMs	<0.001	[0.60]	<0.001	[0.60]
JIRA	GBMs	<0.001	[0.59]	<0.001	[0.62]
MongoDB	GBMs	<0.001	[0.75]	<0.001	[0.71]
Spring	GBMs	<0.001	[0.67]	<0.001	[0.65]

features that performs best in most cases, SA+BoWs, as reported in RQ1. Figure 11 shows the NMAE achieved by the three randomized ensemble methods: Random Forests (RF), Deep Neural Networks with Dropouts (Deep Net.), and Stochastic Gradient Boosting Machines (GBMs), and the traditional Support Vector Regression (SVR), averaging across all projects.

Overall, all the three ensemble methods that we have employed performed well, producing much better predictive performance than the traditional SVR. They achieve 0.392 NMAE averaging across the three ensemble methods, while SVR achieves 0.621 NMAE. The results for the Wilcoxon test to compare the ensemble methods against the tradition regression model is shown in Table 9. The improvement of the ensemble methods over the traditional regression model is significant ($p < 0.001$) with the effect size greater than 0.5 all cases. **The best performer is Stochastic Gradient Boosting Machines (GBMs).** GBMs achieved 0.369 NMAE averaging across all projects as confirmed by the results of the Wilcoxon test with the corresponding A^{12} effect size in Table 10: GBMs perfomed significantly better than RF and Deep Nets. ($p < 0.001$) with effect size greater than 0.5 in all cases.

Answer to RQ2: Randomized ensemble methods significantly outperform traditional methods like SVR in predicting delivery capability.

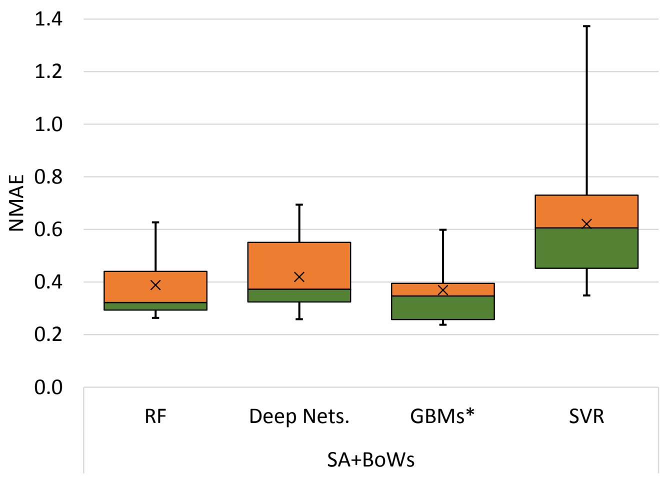


Fig. 11. Evaluation results on different regression models with SA+BoWs. The best performer is marked with *. X is a mean of NMAE averaging across all projects (the lower the better).

6.4.3 Sanity check (RQ3)

To answer RQ3, we employed the best randomized ensemble method: GBMs (identified by RQ2), and the best combination of the aggregated features in each project: SA for Apache, SA+GA for JBoss, SA+BoWs for JIRA and MongoDB, and the combined of all (All) for Spring (identified by RQ1). Figure 12 shows the predictive performance achieved from GBMs with the best aggregated features and the three baseline methods: Random, Mean, and Median in each project. **Our analysis of this evaluation result suggests that the predictive performance obtained with our approach is better than those achieved by using Random, Mean, and Median in all projects.** Averaging across all the project, our approach (GBMs with the best aggregated features in each project) achieves an accuracy of 0.349 NMAE, while the base of the baselines achieve only 0.702 NMAE. The NMAE produced by our model is higher for JBoss than that for other projects. JBoss is also the project that the baseline methods (Random, Mean and Median) struggled with the most. There are a few reasons which explain this phenomenon. Firstly, JBoss has the smallest number of issues among the five studied projects. Small training data may affect the predictive power of a model. Secondly, JBoss has the largest range of story points assigned to issues – it has a standard deviation of 3.27, comparing to 1.71–2.34 standard deviation in the other projects. The high standard deviation indicates that the issue story points in JBoss are spread out over a large range of values. This could make all the models struggle since issue story points directly affect the velocity of an iteration.

Table 11 shows the results of the Wilcoxon test (together with the corresponding A^{12} effect size) to measure the statistical significance and effect size (in brackets) of the improved accuracy achieved by our approach over the baselines: Random Guessing, Mean, and Median. In *all* cases, our approach significantly outperforms the baselines ($p < 0.001$) with (large) effect sizes greater than 0.65.

Answer to RQ3: our approach significantly outperforms the baselines, thus passing the sanity check required by RQ3.

6.4.4 The impact of prediction time (RQ4)

We also varied the prediction time (at the beginning of the iteration, and when it progresses to 30%, 50% and 80% of its planned duration) to observe its impact on predictive performance. Note that the extracted features also correspond to the prediction time (e.g. the number of comments of an issue when an iteration progresses to 80% may greater than that at the beginning of the iteration). Moreover, the difference between the target velocity and actual delivered velocity could be dynamic at different prediction times.

For example, the actual delivered velocity of the iteration named *Mesosphere Sprint 13* in the Apache project is 72, i.e. $velocity(\texttt{Delivered}) = 72$. At the beginning of this iteration, team planned to deliver 15 velocity, i.e. $velocity(\texttt{Committed}) = 15$. The difference between actual delivered velocity and target velocity of this iteration when the prediction is done at the beginning of the iteration (0%) is 57, i.e. $velocity(\texttt{Difference}) = 57$. When this iteration

TABLE 11
Comparison on the predictive performance of our approach against the baseline benchmarks using Wilcoxon test and A^{12} effect size (in brackets)

Project	Model vs	Random		Mean		Median	
Apache	GBMs (SA)	<0.001	[0.82]	<0.001	[0.81]	<0.001	[0.77]
JBoss	GBMs (SA+GA)	<0.001	[0.86]	<0.001	[0.86]	<0.001	[0.66]
JIRA	GBMs (SA+BoWs)	<0.001	[0.88]	<0.001	[0.86]	<0.001	[0.67]
MongoDB	GBMs (SA+BoWs)	<0.001	[0.86]	<0.001	[0.85]	<0.001	[0.80]
Spring	GBMs (All)	<0.001	[0.83]	<0.001	[0.83]	<0.001	[0.79]

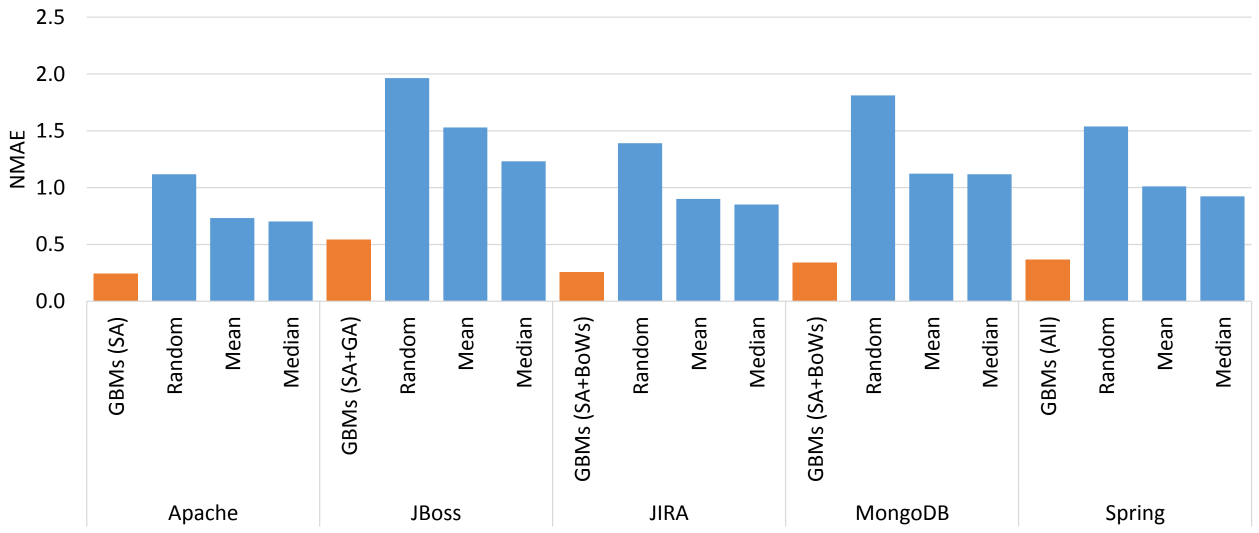


Fig. 12. Evaluation result of the GBMs with the best aggregated features in each project and the three baseline benchmarks

TABLE 12
The descriptive statistics of the difference between actual delivered velocity against the target velocity from the different prediction time

Project	Prediction Time(%)	$velocity(\texttt{Difference})$				
		min	max	mean	median	STD
Apache	0	-81	57	-7.25	-4	17.56
	30	-81	49	-9.05	-5	17.16
	50	-81	40	-10.00	-5	16.97
	80	-81	30	-10.86	-6	16.69
JBoss	0	-89	30	-5.12	-1	14.78
	30	-96	30	-6.03	-1	14.83
	50	-96	30	-6.20	-2	15.05
	80	-106	10	-6.35	-2	13.97
JIRA	0	-74	147	-1.61	0	12.19
	30	-83	117	-2.82	0	11.37
	50	-83	96	-3.28	0	11.09
	80	-86	72	-4.07	0	10.68
MongoDB	0	-67	88	1.61	0	12.78
	30	-67	50	-0.99	0	11.54
	50	-67	48	-2.27	-1	11.16
	80	-85	44	-3.65	-1	11.29
Spring	0	-131	332	34.72	15	63.57
	30	-135	320	23.77	8.5	51.66
	50	-135	316	18.79	4	47.92
	80	-147	312	11.07	1	42.13

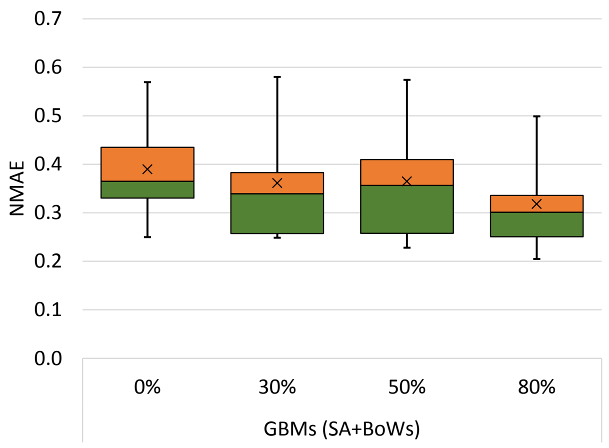


Fig. 13. Evaluation results on varying prediction time. X is a mean of NMAE averaging across all projects (the lower the better).

progressed to 80% of its planned duration, 33 velocity were added and planned to deliver in this iteration, i.e. $velocity(\texttt{Committed}) = 48$. Thus, $velocity(\texttt{Difference})$ of this iteration when the prediction is done at 80% of the planned duration is 24. Table 12 shows the statistical description of $velocity(\texttt{Difference})$ in the different prediction time. The decreasing of STD of the different velocity in the later prediction time in all cases shows that teams may adjust the target velocity of ongoing iterations corresponding to the remaining time of iterations rather than extend the duration.

We again used the best performer (i.e. GBMs with SA+BoWs) in most cases to perform this experiment. Figure 13 shows NMAE achieved in predicting $velocity(\texttt{Difference})$ from the four different prediction times (i.e. at the beginning of the iteration, and when it progresses to 30%, 50%, and 80% of its planned duration) obtained from running GBMs using the combination of statistical aggregation and Bag-of-Words aggregation approaches averaging across all projects. **We observe that the predictive performance achieved from the predictions made at a later time in an iteration is better than those that the predictions made at the beginning of the iterations (0%)** – the prediction at 30%, 50%, and 80% of the iteration's duration achieves an average of 0.348 NMAE while the prediction at the beginning of the iteration achieves 0.390 NMAE, averaging across all projects. This confirms our earlier hypothesis that the latter we predict, the more accuracy we could gain since more information has become available. That phenomenon is however not consistently seen with the predictive performance of the prediction made at 50% of the iterations duration being slightly lower than those that made at 30% of the iterations duration. The prediction made at 80% of the iterations duration does however achieve the highest predictive performance it achieves 0.318 NMAE, averaging across all projects. For the sake of completeness, our experiments covered a range of prediction times from 0% to 80% to sufficiently test a hypothesis that the latter we predict, the more accuracy we could gain. We however acknowledge that making a prediction at 80% might be less useful in practice since the outcomes have become obvious and/or it might be too late to change the outcomes.

We investigated further to see when predictions could be made while not losing too much predictive power. To do so, we analyzed the improvement of the predictive performance between different prediction time intervals. We found that when we delayed making a prediction by 30%, we gained only 7-12% improvement in predictive performance. In fact, there was not much difference in terms of predictive power when making a prediction at the 30% or 50% marks. This result suggests that it is reasonably safe to make a prediction early, even at the beginning of an iteration.

Answer to RQ4: The time when a prediction is made affects the predictive performance, but only small improvement is gained when we delay making the prediction.

6.4.5 *Classifying the outcomes of an iteration* ***(RQ5)***

We defined a tolerance margin to classify the outcomes of an iteration based on the statistical characteristics of each project. To maintain a relative balance between the classes, we used the 33^{th} and the 66^{th} percentile of $velocity(\texttt{Difference})$ as the tolerance margin: an iteration having $velocity(\texttt{Difference})$ below the 33^{th} percentile falls into *under-achieved* class, an iteration having $velocity(\texttt{Difference})$ above the 66^{th} percentile falls into *over-achieved* class, otherwise an iteration falls into *achieved* class. Table 13 shows the number of iterations in each class according to the 33^{th} and the 66^{th} percentile from each project. For example, in the Apache project, an iteration falls into *under-achieved* class if $velocity(\texttt{Difference})$ is below -11, and falls into *over-achieved* class if $velocity(\texttt{Difference})$ is over 0.

In this experiment we also used GBMs with SA+BoWs and applied these margins to the predicted value of $velocity(\texttt{Difference})$. For example, in the Apache project, an iteration is predicted as *under-achieved* if the predicted value is below -11 (i.e. the 33^{th} percentile). Figure 14 shows the precision, recall, F-measure, and AUC achieved for each of five open source projects. **These evaluation results demonstrate the effectiveness of our predictive models in predicting the outcomes of iterations across the five projects**, achieving on average 0.79 precision, 0.82 recall, and 0.79 F-measure. The degree of discrimination achieved by our predictive models is also high, as reflected in the AUC results – the average of AUC across all projects is 0.86. The AUC quantifies the overall ability of the discrimination between classes. Our model performed best for the Spring project, achieving the highest precision (0.87), recall (0.85), F-measure (0.86), and AUC (0.90). The result from using MCC as a measure (Figure 15) also corresponds to the other measures. As can be seen from Figure 15, our approach achieved over 0.5 MCC in all cases – our approach achieves 0.71, averaging across all projects. MMAE is used to assess the performance of our models in terms of predicting ordered outcomes. As can be seen from Figure 16, our approach achieved 0.20 MMAE, averaging across all projects.

Answer to RQ5: Our predictive model is also highly accurate in classifying the outcomes of an iteration.

TABLE 13
Number of iterations in each class in each project

Project	Percentile		Class		
	33^{th}	66^{th}	Over	Achieved	Under
Apache	-11	0	105	135	108
JBoss	-4	0	47	233	92
JIRA	-3	0	218	1211	444
MongoDB	-2	0	244	273	248
Spring	0	20	162	162	152

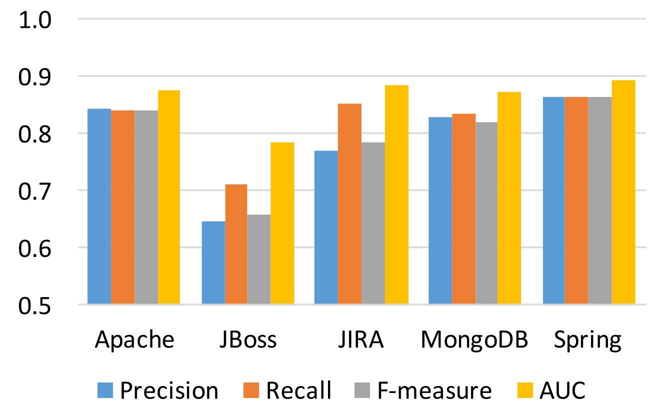


Fig. 14. Evaluation results on predicting the outcomes of iterations in terms of precision, recall, F-measure, and AUC from each project (the higher the better)

TABLE 14
Top–10 most important features with their normalized weight

Apache		JBoss		JIRA	
To-do velocity	1.00	To-do velocity	1.00	To-do velocity	1.00
Velocity at start time	0.93	Velocity at start time	0.65	Cluster 5 (BoWs)	0.61
No. of issues at start time	0.53	No. of to-do issues	0.45	Cluster 53 (BoWs)	0.37
No. of to-do issues	0.50	No. of issues at start time	0.37	Cluster 93 (BoWs)	0.35
Type (Epic)	0.32	Priority (Critical)	0.27	Type (Story)	0.28
Changing of desc. (var)	0.23	Changing of fix versions (max)	0.19	Velocity at start time	0.26
Changing of desc. (mean)	0.20	Type (Feature Request)	0.16	Type (Major)	0.25
In-progress velocity	0.19	No. of comments (std)	0.15	Type (Improvement)	0.24
Type (Story)	0.19	No. of team mem.	0.14	No. of comments (std)	0.22
No. of team mem.	0.18	No. of fix versions (var)	0.14	Cluster 27 (BoWs)	0.20
MongoDB		**Spring**			
Velocity at start time	1.00	Type (Major)	1.00		
In-progress velocity	0.95	Done velocity	0.85		
To-do velocity	0.94	In-progress velocity	0.82		
No. of in-progress issues	0.71	No. of edges	0.76		
Added velocity	0.67	Sum of fan in	0.75		
No. of issues at start time	0.47	Sum of fan out	0.74		
Cluster 43 (BoWs)	0.46	No. of comments (std)	0.68		
Priority (Hard)	0.38	Cluster 56 (BoWs)	0.59		
Cluster 21 (BoWs)	0.36	Mean of fan in	0.58		
Done velocity	0.35	Added velocity	0.53		

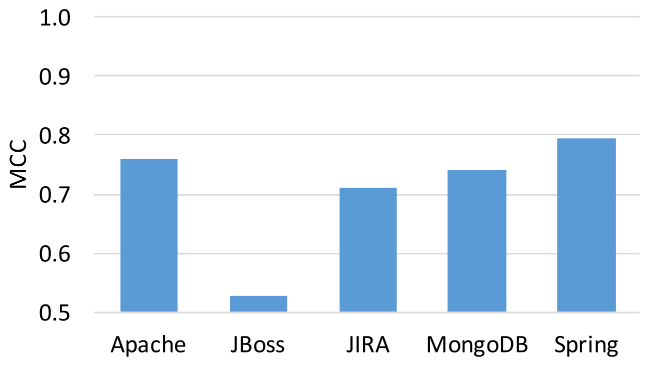


Fig. 15. Matthews Correlation Coefficient (MCC) results from each project (the higher the better)

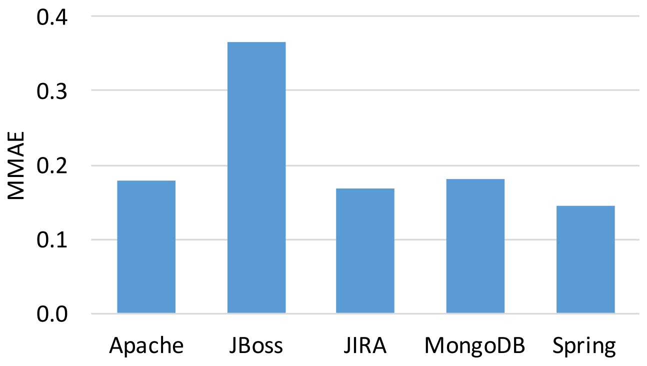


Fig. 16. Macro-averaged Mean Absolute Error (MMAE) results (the lower the better)

6.4.6 *Important features*

Table 14 reports the top-10 most important features and their weight obtained from running Random Forests with the combination of all aggregated features of issues. The weights here reflect the discriminating power of a feature since they are derived from the number of times the feature is selected (based on information gain) to split in a decision tree [44]. The weights are normalized in such a way that the most important feature has a weight of 1 and the least important feature has a weight of 0. **We observe that iteration features and statistical aggregation features are dominant in the top-10 list.** In many projects (e.g. Apache, JBoss, and MongoDB) iteration features such as the number of to-do issues, to-do velocity, and velocity at start time, are good predictors for foreseeing how an iteration will achieve against the target. It also corresponds to our results for finding the best combinations of aggregated features in RQ1. For example, in the JIRA and MongoDB projects, there are several aggregated features from statistical aggregation (SA) and Bag-of-Words aggregation (BoWs) that have high discriminating power since SA+BoWs performs best in those projects. In addition, the changing of the other issue attributes (e.g. fix versions, description) are also in the top-10 in several projects (e.g. Apache and JBoss). In Spring, the graph-based aggregated features (e.g. sum of fan in/out) are good predictors. In the JBoss project, the features related to comments and number of team members are important for predicting the difference between the actual delivered velocity and the target velocity which may suggests that team collaboration is an important factor.

6.5 Implications and lessons learned

Results from our evaluations on five large open source projects suggest that our approach is highly reliable in predicting delivery capability at the iteration level. It allows project managers and other decision makers to quickly foresee, at any given time during an ongoing iteration, if their team is at risk of not meeting the target deliverable set for this iteration. Predicting delivery capability early allows the team to deploy mitigation measures such as appropriate changes affecting the values of the top-5 predictors presented the previous section. Effective project management should also identify situations where an unexpected future event might present an opportunity to be exploited. Our approach supports iterative software development by forecasting whether the team is likely to deliver more than what has been planned for in an iteration. Knowing these opportunities early allows the team to plan for accommodating extra high-priority issues into the current iteration.

Story points are used to compute velocity, a measured use for the teams delivery capability per iteration. In practice, story points are however developed by a specific team based on the teams cumulative knowledge and biases, and thus may not be useful outside the team (e.g. in comparing performance across teams). Hence, the trained models are specific to teams and projects.

Our evaluation also demonstrates the high performance of the three ensemble methods used in building our prediction models. This suggests that ensemble methods such as Random Forests or Gradient Boosting Machines can be highly recommended for building software analytics models. Deep learning neural networks are only effective where there are significantly large amounts of data for training, which might not be the case for a range of software engineering problems.

One of the key novelties in our approach is deriving new features for an iteration from aggregating the features of its issues and their dependencies. These features can be derived by using a range of statistics over the issues' features or automatically learned using the bag-of-word approach. Our experimental results demonstrate the effectiveness of this approach and that they are complementary to each other. These results suggest that these feature aggregations techniques can be useful in other software analytics settings where features are located in different layers (similarly to iterations and issues).

In addition, there is well-established knowledge in machine learning that model accuracy depends critically on informative and comprehensive features extracted from data. It is also known that features are more useful when there are less redundancies. In our data, there are two main separate structures: the attributes associated within each issue and the dependency structure between issues. Our three feature sets are obtained through (a) aggregation of issue attributes, hence capturing the salient characteristics within a sprint, (b) building Bag-of-Words which is essentially the well-known vector quantization technique by finding distinct exemplars via k-means, hence reducing redundancies, and (b) exploiting graph characteristics, hence adding complementary information. Our experiments demonstrate that combining those three feature sets yields the best performance, thus confirming the prior knowledge. The increase in performance cannot be explained just by the increase in model complexity. This is because a more complex model will definitely fit the training data better, but it is more likely to hurt performance on test data due to the classic problem known as overfitting. If our goal is to derive a highly accurate model, then model complexity should not be a problem, as long as the model generalises better than simpler alternatives.

A major contribution of our work is demonstrating the utility of randomized methods to avoid the need for feature selection and reduction. While we acknowledge that a small set of independent features would be easy to understand, realistic problems are often complex enough to warrant a comprehensive feature set. The use of feature aggregation techniques has given us an extensive set of features to characterize a software development iteration. Although feature selection could be employed to filter out "weak" predictors, feature selection can be unstable: each selection method, running on different data samples, can produce a different subset of features. In modern machine learning techniques, feature correlation is no longer a crucial issue [26]. Randomized methods such as those used in this paper need neither feature selection nor dimensionality reduction. This is because at each training step, only a small random subset of features is used – this also breaks the correlation between any feature pair since correlated features are much less likely to be in the same subset [45], [46], [47]. In addition, when making a prediction, a combination of many classifiers are used, each of which works on a smaller feature set.

6.6 Threats to validity

There are a number of threats to the validity of our study, which we discuss below.

Threats to construct validity: Construct validity concerns whether independent and dependent variables from which the hypothesized theory is constructed are relevant. We mitigated these threats by using real world data from iterations and issues recorded in several large open source projects. We collected iterations, all issues associated to these iterations, and all the relevant historical information available to ensure. The ground-truth (i.e. the difference between the actual delivered velocity and the target velocity) is based on the story points assigned to issues. Those story points were estimated by teams, and thus may contain human biases. However, story points are currently the best practices for measuring the delivery capability of a team, and are widely used in the industry. Hence, using story points makes our approach relevant to current industry practices.

Threats to conclusion validity: We tried to minimize threats to conclusion validity by carefully selecting unbiased error measures and applied a number of statistical tests to verify our assumptions [41] and following recent best practices in evaluating and comparing predictive models regarding effort estimation [41], [48]. In terms of predicting the three outcomes of an iteration (i.e. classification), our performance measures were also carefully designed against reporting bias towards majority classes. For example, while the F-measure is a balance between recall (often low for minority class) and precision (often high for minority class), we also employed the MCC performance measure which is insensitive to class imbalance. We used the MMAE performance measure for ordered classes. We however acknowledge that other techniques could also be used such as doing statistical undersampling, or artificially creating more samples for the undersampled class.

Threats to internal validity: Our great concern for threats to internal validity is data preprocessing. We found that around 30% of the issues across the five projects have not been assigned a story point (missing data). We filled those missing values using the mean of story points in each project. Our future work will investigate the use of other imputation techniques to handle such missing data. We also removed outliers (i.e. the difference velocity) and iterations involved with zero issue. We carefully processed the issue's change log to extract the features regard to a prediction time to prevent the leaking [15]. In addition, we also tried to avoid instability from applying additional data preprocessing algorithms (e.g. feature engineering) [49] by employing the ensemble randomized methods which overcome these problems (e.g. feature correlation, feature selection) [26].

Threats to external validity: We have considered almost 4,000 iterations and 60,000 issues from five large open source projects, which differ significantly in size, complexity, team of developers, and the size of community. All iteration and issue reports are real data that were generated during from the software development in open source settings. We however acknowledge that our data set may not be representative of all kinds of software projects, especially in commercial settings (although open source projects and commercial projects are similar in many aspects). Further investigation to confirm our findings for other open source and for closed source projects is needed.

7 Related work

Today's agile, dynamic and change-driven projects require different approaches to planning and estimating [4]. A number of studies have been dedicated to effort estimation in agile software development. Estimation techniques that rely on experts' subjective assessment are commonly used in practice, but they tend to suffer from the underestimation problem [50], [51]. Some recent approaches leverage machine learning techniques to support effort estimation for agile projects. The work in [52] developed an effort prediction model for iterative software development setting using regression models and neural networks. Differing from traditional effort estimation models (e.g. COCOMO [53], [54]), this model is built after each iteration (rather than at the end of a project) to estimate effort for the next iteration. The work in [55], [56] built a Bayesian network model for effort prediction in software projects which adhere to the agile Extreme Programming method. Their model however relies on several parameters (e.g. process effectiveness and process improvement) that require learning and extensive fine tuning.

Bayesian networks are also used [57] to model dependencies between different factors (e.g. sprint progress and sprint planning quality influence product quality) in a Scrum-based software development project in order to detect problems in the project. Our work predicts not only the committed velocity against the actual achieved velocity but also the delivered against non-delivered velocity. More importantly, we use a comprehensive aggregation of features at both the iteration and issue levels, which represents the novelty of our work.

In our previous work [14] we developed models for characterizing and predicting delays at the level of issues. Several approaches have also been proposed to predict the resolving time of a bug or issue (e.g. [16], [58], [59], [60], [61]). Those work however mainly focus on waterfall software development processes. This work, in contrast, aims to predict the quantum of achieved works at the level of iterations (and can be extended to releases as well) in agile software development. Here, we leverage feature/representation learning techniques which were not used in those previous work.

Graph-based characterization, one of the techniques we employed here, has also been used for building predictive models in software engineering. The study in [62] shows the impact of dependency network measures on post-release failures. The work in [25] built graphs representing source code, module and developer collaboration and used a number of metrics from those graphs to construct predictors for bug severity, frequently changed software parts and failure-prone releases. Similarly, Zimmermann and Nagappan [63] built dependency graphs for source code and used a number of graph-based measures to predict defects. Those approaches mostly work at the level of source code rather than at the software task and iteration level as in our work. Our recent work [18] proposed an approach to construct a network of software issues and used networked classification for predicting which issues are a delay risk. Those approaches however did not specifically look at the iterative and agile software development settings as done here in our work.

Our work is also related to the work on predicting and mining bug reports, for example, blocking bug prediction (e.g. [19]), re-opened bug prediction (e.g. [64], [65]), severity/priority prediction (e.g. [66], [67]), delays in the integration of a resolved issue to a release (e.g. [68]), bug triaging (e.g. [69], [70], [71]), duplicate bug detection ([72], [73], [74], [75], [76], [77]), and defect prediction (e.g. [78], [79].

8 Conclusions and future work

Iterative software development methodologies have quickly gained popularity and gradually become the mainstream approach for most software development. In this paper, we have proposed a novel approach to delivery-related risk prediction in iterative development settings. Our approach is able to predict how much work gets done in the iteration. Our approach exploits both features at the iteration level and at the issue level. We also used a combination of three distinct techniques (statistical feature aggregation, bag-of-words feature learning, and graph-based measures) to derive a comprehensive set of features that best characterize an iteration. Our prediction models also leverage state-of-the-art machine learning randomized ensemble methods, which produce a strong predictive performance.

In terms of future work, we plan to enrich our feature set with more features characterizing a development team, e.g. team structure, and a developer's skill, and workload. We will also explore how to use graph embedding to map a graph of issues into a vector using unsupervised learning. This is an alternative way to characterize an iteration using features of the issues (and their inter-dependencies) assigned to it. Performing additional experiments with different training and test sets (using the sliding window approach) is also part of our future work. We will also look into expanding our study to commercial, closed source software projects.

REFERENCES

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

[2] 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.

[3] L. Williams, "What agile teams think of agile principles," *Communications of the ACM*, vol. 55, no. 4, p. 71, 2012.

[4] M. Cohn, *Agile estimating and planning*. Pearson Education, 2005.

[5] A. Mockus, D. Weiss, and P. Z. P. Zhang, "Understanding and predicting effort in software projects," in *Proceedings of the 25th International Conference on Software Engineering (ICSE)*, vol. 6. IEEE, 2003, pp. 274–284.

[6] F. Sarro, A. Petrozziello, and M. Harman, "Multi-objective Software Effort Estimation," in *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, 2016, pp. 619–630.

[7] E. Kocaguneli, T. Menzies, and J. W. Keung, "On the value of ensemble effort estimation," *IEEE Transactions on Software Engineering*, vol. 38, no. 6, pp. 1403–1416, 2012.

[8] L. Huang, D. Port, L. Wang, T. Xie, and T. Menzies, "Text mining in supporting software systems risk assurance," in *Proceedings of the IEEE/ACM international conference on Automated software engineering (ASE)*. New York, USA: ACM Press, 2010, pp. 163 – 167.

[9] Z. Xu, B. Yang, and P. Guo, "Software Risk Prediction Based on the Hybrid Algorithm of Genetic Algorithm and Decision Tree," *Advanced Intelligent Computing Theories and Applications. With Aspects of Contemporary Intelligent Computing Techniques*, vol. 2, pp. 266–274, 2007.

[10] 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, 2013.

[11] C. Fang and F. Marle, "A simulation-based risk network model for decision support in project risk management," *Decision Support Systems*, vol. 52, no. 3, pp. 635–644, 2012.

[12] M. N. Moreno García, I. R. Román, F. J. García Peñalvo, and M. T. Bonilla, "An association rule mining method for estimating the impact of project management policies on software quality, development time and effort," *Expert Systems with Applications*, vol. 34, no. 1, pp. 522–529, 2008.

[13] H. F. Cervone, "Understanding agile project management methods using Scrum," *OCLC Systems & Services: International digital library perspectives*, vol. 27, no. 1, pp. 18–22, 2011.

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

[15] S. Kaufman and C. Perlich, "Leakage in Data Mining : Formulation , Detection , and Avoidance," *ACM Transactions on Knowledge Discovery from Data (TKDD)*, vol. 6(4), no. 15, pp. 556–563, 2012.

[16] L. D. Panjer, "Predicting Eclipse Bug Lifetimes," in *Proceedings of the 4th International Workshop on Mining Software Repositories (MSR)*, 2007, pp. 29–32.

[17] X. Xia, D. Lo, E. Shihab, X. Wang, and X. Yang, "ELBlocker: Predicting blocking bugs with ensemble imbalance learning," *Information and Software Technology*, vol. 61, pp. 93–106, 2015.

[18] M. Choetkiertikul, H. K. Dam, T. Tran, and A. Ghose, "Predicting delays in software projects using networked classification," in *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015, pp. 353 – 364.

[19] 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)*. ACM Press, 2014, pp. 72–81.

[20] X. Xia, D. Lo, M. Wen, E. Shihab, and B. Zhou, "An empirical study of bug report field reassignment," in *Proceedings of the Conference on Software Maintenance, Reengineering, and Reverse Engineering*, 2014, pp. 174–183.

[21] D. R. McCallum and J. L. Peterson, "Computer-based readability indexes," in *Proceedings of the ACM'82 Conference*. ACM, 1982, pp. 44–48.

[22] P. Hooimeijer and W. Weimer, "Modeling bug report quality," in *Proceedings of the 22 IEEE/ACM international conference on Automated software engineering (ASE)*. ACM Press, nov 2007, pp. 34 – 44.

[23] P. Tirilly, V. Claveau, and P. Gros, "Language modeling for bag-of-visual words image categorization," in *Proceedings of the 2008 international conference on Content-based image and video retrieval*, 2008, pp. 249–258.

[24] E. Zegura, K. Calvert, and S. Bhattacharjee, "How to model an internetwork," in *Proceedings of the 15th Annual Joint Conference of the IEEE Computer Societies Networking the Next Generation (INFOCOM)*, vol. 2, 1996, pp. 594–602.

[25] 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)*. IEEE Press, 2012, pp. 419–429.

[26] L. Rokach, "Taxonomy for characterizing ensemble methods in classification tasks: A review and annotated bibliography," *Computational Statistics and Data Analysis*, vol. 53, no. 12, pp. 4046–4072, 2009.

[27] D. O. Maclin and R., "Popular Ensemble Methods: An Empirical Study," vol. 11, pp. 169–198, 1999.

[28] T. Dietterich, "Ensemble methods in machine learning," in *Proceedings of the 1st International Workshop on Multiple Classifier Systems*, vol. 1857. Springer, 2000, pp. 1–15.

[29] T. G. Dietterich, "An experimental comparison of three methods for constructing ensembles of decision trees: Bagging, boosting, and randomization," *Machine learning*, vol. 40, no. 2, pp. 139–157, 2000.

[30] M. Galar, A. Fernandez, E. Barrenechea, H. Bustince, and F. Herrera, "A review on ensembles for the class imbalance problem: Bagging-, boosting-, and hybrid-based approaches," *IEEE Transactions on Systems, Man and Cybernetics Part C: Applications and Reviews*, vol. 42, no. 4, pp. 463–484, 2012.

[31] S. M. Halawani, I. A. Albidewi, and A. Ahmad, "A Novel Ensemble Method for Regression via Classification Problems," *Journal of Computer Science*, vol. 7, no. 3, pp. 387–393, 2011.

[32] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[33] J. H. Friedman, "Greedy function approximation: a gradient boosting machine," *Annals of Statistics*, vol. 29, no. 5, pp. 1189–1232, 2001.

[34] J. H. Friedsman, "Stochastic gradient boosting," *Computational Statistics & Data Analysis*, vol. 38, no. 4, pp. 367–378, 2002.

[35] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, pp. 1929–1958, 2014.

[36] X. Wang and A. McCallum, "Topics over time: a non-markov continuous-time model of topical trends," in *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2006, pp. 424–433.

[37] M. Fernández-Delgado, E. Cernadas, S. Barro, D. Amorim, and D. Amorim Fernández-Delgado, "Do we Need Hundreds of Classifiers to Solve Real World Classification Problems?" *Journal of Machine Learning Research*, vol. 15, pp. 3133–3181, 2014.

[38] K. H. Esbensen and P. Geladi, "Principles of proper validation: Use and abuse of re-sampling for validation," *Journal of Chemometrics*, vol. 24, pp. 168–187, 2010.

[39] K. Goldberg, T. Roeder, D. Gupta, and C. Perkins, "Eigentaste: A Constant Time Collaborative Filtering Algorithm," *Information Retrieval*, vol. 4, no. 2, pp. 133–151, 2001.

[40] K. Muller, "Statistical power analysis for the behavioral sciences," *Technometrics*, vol. 31, no. 4, pp. 499–500, 1989.

[41] A. Arcuri and L. Briand, "A Hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering," *Software Testing, Verification and Reliability*, vol. 24, no. 3, pp. 219–250, 2014.

[42] S. Baccianella, A. Esuli, and F. Sebastiani, "Evaluation measures for ordinal regression," in *Proceedings of the 9th International Conference on Intelligent Systems Design and Applications (ISDA)*. IEEE, 2009, pp. 283–287.

[43] D. M. W. Powers, "Evaluation: From Precision, Recall and F-Factor to ROC, Informedness, Markedness & Correlation David," *Journal of Machine Learning Technologies*, vol. 2, no. 1, pp. 37–63, 2011.

[44] R. Genuer, J.-M. Poggi, and C. Tuleau-Malot, "Variable selection using random forests," *Pattern Recognition Letters*, vol. 31, no. 14, pp. 2225–2236, 2010.

[45] K. Tumer and N. C. Oza, "Input decimated ensembles," *Pattern Analysis and Applications*, vol. 6, no. 1, pp. 65–77, 2003.

[46] K. Tumer and J. Ghosh, "Error Correlation and Error Reduction in Ensemble Classifiers," *Connection Science*, vol. 8, no. 3-4, pp. 385–404, 1996.

[47] E. Tuv, "Feature Selection with Ensembles , Artificial Variables , and Redundancy Elimination," *Journal of Machine Learning Research*, vol. 10, pp. 1341–1366, 2009.

[48] M. Shepperd and S. MacDonell, "Evaluating prediction systems in software project estimation," *Information and Software Technology*, vol. 54, no. 8, pp. 820–827, 2012. [Online]. Available: http://dx.doi.org/10.1016/j.infsof.2011.12.008

[49] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking Classification Models for Software Defect Prediction: A Proposed Framework and Novel Findings," *IEEE Transactions on Software Engineering*, vol. 34, no. 4, pp. 485–496, jul 2008.

[50] M. Usman, E. Mendes, F. Weidt, and R. Britto, "Effort Estimation in Agile Software Development: A Systematic Literature Review," in *Proceedings of the 10th International Conference on Predictive Models in Software Engineering (PROMISE)*, 2014, pp. 82–91.

[51] M. Usman, E. Mendes, and J. Börstler, "Effort estimation in agile software development: A survey on the state of the practice," in *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering*, 2015, pp. 1–10.

[52] P. Abrahamsson, R. Moser, W. Pedrycz, A. Sillitti, and G. Succi, "Effort prediction in iterative software development processes – incremental versus global prediction models," *1st International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pp. 344–353, 2007.

[53] B. W. Boehm, R. Madachy, and B. Steece, *Software cost estimation with Cocomo II*. Prentice Hall PTR, 2000.

[54] O. Benediktsson, D. Dalcher, K. Reed, and M. Woodman, "COCOMO-Based Effort Estimation for Iterative and Incremental Software Development," *Software Quality Journal*, vol. 11, pp. 265–281, 2003.

[55] P. Hearty, N. Fenton, D. Marquez, and M. Neil, "Predicting Project Velocity in XP Using a Learning Dynamic Bayesian Network Model," *IEEE Transactions on Software Engineering*, vol. 35, no. 1, pp. 124–137, 2009.

[56] R. Torkar, N. M. Awan, A. K. Alvi, and W. Afzal, "Predicting software test effort in iterative development using a dynamic Bayesian network," in *Proceedings of the 21st IEEE International Symposium on Software Reliability Engineering – (Industry Practice Track)*. IEEE, 2010.

[57] M. Perkusich, H. De Almeida, and A. Perkusich, "A model to detect problems on scrum-based software development projects," *The ACM Symposium on Applied Computing*, pp. 1037–1042, 2013.

[58] P. Bhattacharya and I. Neamtiu, "Bug-fix time prediction models: can we do better?" in *Proceedings of the 8th working conference on Mining software repositories (MSR)*. ACM, 2011, pp. 207–210.

[59] 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)*. ACM, 2010, pp. 52–56.

[60] 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)*. ACM Press, 2011, pp. 1–8.

[61] C. Weiss, R. Premraj, T. Zimmermann, and A. Zeller, "How Long Will It Take to Fix This Bug?" in *Proceedings of the 4th International Workshop on Mining Software Repositories (MSR)*, 2007, pp. 1–8.

[62] N. Bettenburg and A. E. Hassan, "Studying the impact of dependency network measures on software quality," in *Proceedings of the International Conference on Software Maintenance (ICSM)*, 2012, pp. 1–10.

[63] T. Zimmermann and N. Nagappan, "Predicting defects using network analysis on dependency graphs," in *Proceedings of the 13th international conference on Software engineering (ICSE)*, 2008, pp. 531–540.

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

[65] 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.

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

[67] T. Menzies and A. Marcus, "Automated severity assessment of software defect reports," in *Proceedings of the International Conference on Software Maintenance (ICSM)*. IEEE, 2008, pp. 346–355.

[68] D. Alencar, S. L. Abebe, S. Mcintosh, D. Alencar da Costa, S. L. Abebe, S. Mcintosh, U. Kulesza, and A. E. Hassan, "An Empirical Study of Delays in the Integration of Addressed Issues," in *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2014, pp. 281–290.

[69] 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, 2011.

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

[71] G. Murphy and D. Čubranić, "Automatic bug triage using text categorization," in *Proceedings of the 16th International Conference on Software Engineering & Knowledge Engineering (SEKE)*, 2004, pp. 92–97.

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

[73] 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 (ICSE)*, 2008, pp. 461–470.

[74] N. Bettenburg, R. Premraj, and T. Zimmermann, "Duplicate bug reports considered harmful . . . really?" in *Proceedings of the International Conference on Software Maintenance (ICSM)*, 2008, pp. 337–345.

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

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

[77] A. T. Nguyen, T. T. N. Nguyen, D. Lo, and C. Sun, "Duplicate bug report detection with a combination of information retrieval and topic modeling," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2012, pp. 70–79.

[78] 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," in *Proceeding of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2013, pp. 268–278.

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

[80] S. D. Conte, H. E. Dunsmore, and V. Y. Shen, *Software Engineering Metrics and Models*. Redwood City, CA, USA: Benjamin-Cummings Publishing Co., Inc., 1986.

[81] T. Foss, E. Stensrud, B. Kitchenham, and I. Myrtveit, "A simulation study of the model evaluation criterion MMRE," *IEEE Transactions on Software Engineering*, vol. 29, no. 11, pp. 985–995, 2003.

[82] B. Kitchenham, L. Pickard, S. MacDonell, and M. Shepperd, "What accuracy statistics really measure," *IEE Proceedings - Software*, vol. 148, no. 3, p. 81, 2001.

[83] M. Korte and D. Port, "Confidence in software cost estimation results based on MMRE and PRED," *Proceedings of the 4th international workshop on Predictor models in software engineering (PROMISE)*, pp. 63–70, 2008.

[84] D. Port and M. Korte, "Comparative studies of the model evaluation criterions mmre and pred in software cost estimation research," in *Proceedings of the 2nd ACM-IEEE international symposium on Empirical software engineering and measurement*. ACM, 2008, pp. 51–60.

[85] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*, 2011, pp. 1–10.
[86] R. M. Everson and J. E. Fieldsend, "Multi-class ROC analysis from a multi-objective optimisation perspective," *Pattern Recognition Letters*, vol. 27, no. 8, pp. 918–927, 2006.
[87] B. W. Matthews, "Comparison of the predicted and observed secondary structure of T4 phage lysozyme," *Biochimica et Biophysica Acta (BBA)-Protein Structure*, vol. 405, no. 2, pp. 442–451, 1975.
[88] G. Jurman, S. Riccadonna, and C. Furlanello, "A comparison of MCC and CEN error measures in multi-class prediction," *PLoS ONE*, vol. 7, no. 8, pp. 1–8, 2012.

PLACE PHOTO HERE

Morakot Choetkiertikul Biography text here.

PLACE PHOTO HERE

Hoa Khanh Dam Biography text here.

PLACE PHOTO HERE

Truyen Tran Biography text here.

PLACE PHOTO HERE

Aditya Ghose Biography text here.

PLACE PHOTO HERE

John Grundy Biography text here.

Appendix A
Predictive Models

A.1 Random Forests

Random forests (RFs) [32] uses decision trees as weak learners and typically works as follows. First, a subset of the full dataset is randomly sampled (i.e. 200 out of 1,000 iterations) to train a decision tree. At each node of the decision tree, we normally search for the best feature across *all* the features (predictor variables) to split the training data. Instead of doing so, at each node of a decision tree, RFs randomly selects a subset of candidate predictors and then find the best splitting predictor for the node. For example, if there are 200 features, we might select a random set of 20 in each node, and then split using the best feature among the 20 available, instead of the best among the full 200 features. Hence, RFs introduces randomness not just into the training samples but also into the actual trees growing.

This process is repeated: a different random sample of data is selected to train a second decision tree. The predictions made by this second tree is typically different from those of the first tree. RFs continues generating more trees, each of which is built on a slightly different sample and producing slightly different predictions each time. We could continue this process indefinitely, but in practice 100 to 500 trees are usually generated. To make predictions for a new data, RFs combines all separate predictions made by each of the generated decision tree typically by averaging the outputs across all trees.

In our work, we trained 500 regression trees using the randomeforest-matlab[7] package for Matlab.

A.2 Stochastic Gradient Boosting Machines

RFs grows independent decision trees (which thus can be done in parallel) and simply takes the average of the predictions produced by those trees as the final prediction. On the other hand, gradient boosting machines (GBMs) [33], [34] generates trees and adds them to the ensemble in a *sequential* manner. The first tree is generated in the same way as done in RFs. The key difference here is the generation of the second tree which aims at minimizing the prediction errors produced by the first tree (thus the trees are not independent to each other as in RFs). Both the first and second trees are added to the ensemble but different weights are assigned to each of them. This process is repeated multiple times: at each step, a new tree is trained with respect to the error of the whole ensemble learnt so far and is then added to the ensemble. The final ensemble is used as a model for predicting the outcome of new inputs.

Unlike RFs, a weak learner in GBMs can be not just only regression trees but also any other regression learning algorithms such as neural networks or linear regression. In our implementation, regression trees are used as weak learners and 100 trees were generated.

A.3 Deep Neural Networks with Dropouts

Neural networks have long been used in many prediction tasks where the input data has many variables and noises. The neural network is organized in a series of layers: the bottom layer accepting the input, which is projected to a hidden layer, which in turn projects to an output layer. Each layer consists of a number of computation units, each of which is connected to other units in the next layer. Deep neural networks (DNNs) are traditional artificial neural networks with multiple hidden layers, which make them very expressive models that are capable of learning highly complicated relationships between their inputs and outputs. Limited training data may however cause overfitting problem, where many complicated relationships exist in the training data but does not occur in the real test data.

Building an ensemble of many differently trained neural networks would alleviate the overfitting problem (as seen in RFs and GBMs). To achieve the best performance, each individual neural network in the ensemble should be different from each other in terms of either the architecture or the data used for training. *Dropout* [35] is a simple but scalable technique which, given a main network, builds an ensemble from many variants of this network. These variant networks are generated by temporarily removing one random unit (and its incoming and outgoing connections) at a time from the main network. Hence, a neural network with n units can be used to generate an ensemble of 2^n networks. After being trained, those 2^n networks can be combined into a single neural network to make predictions for the new inputs.

7. https://code.google.com/archive/p/randomforest-matlab/

Appendix B
Performance measures

B.1 Normalized Mean Absolute Error (NMAE)

There are a range of measures used in evaluating the accuracy of a predictive model in teams of effort estimation. Most of them are based on the Absolute Error, (i.e. $|ActualDiff - EstimatedDiff|$). where $AcutalDiff$ is the real $velocity(\texttt{Difference})$ and $EstimatedDiff$ is the predicted $velocity(\texttt{Difference})$ given by a predictive model. Mean of Magnitude of Relative Error (MRE) and Prediction at level l [80], i.e. Pred(l), have also been used in effort estimation. However, a number of studies [81], [82], [83], [84] have found that those measures bias towards underestimation and are not stable when comparing the models. Thus, the *Mean Absolute Error (MAE)* has recently been recommended to compare the performance of effort estimation models [6].

Since different projects have different $velocity(\texttt{Difference})$ ranges, we need to normalize the MAE (by dividing it with the interquartile range) to allow for comparisons of the MAE across the studied project. The Normalized Mean Absolute Error (NMAE) [39] is defined as:

$$NMAE = \frac{\frac{1}{N}\sum_{i=1}^{N}|ActualDiff_i - EstimatedDiff_i|}{IQR}$$

where N is the number of iterations used for evaluating the performance (i.e. test set), $ActualDiff_i$ is the actual $velocity(\texttt{Difference})$, $EstimatedDiff_i$ is the predicted $velocity(\texttt{Difference})$ for the iteration i, and IQR is the distance between the $velocity(\texttt{Difference})$ at 75^{th} and 25^{th} percentile (i.e. $75^{th}percentile - 25^{th}percentile$).

We assess the $velocity(\texttt{Difference})$ produced by the predictive models using NMAE. To compare the performance of two predictive models, we tested the statistical significance of the absolute errors achieved with the two models using the Wilcoxon Signed Rank Test [40]. The Wilcoxon test is a safe test since it makes no assumptions about underlying data distributions. The null hypothesis here is: "the absolute errors provided by a predictive model are significantly less that those provided by another predictive model". We set the confidence limit at 0.05 (i.e. $p < 0.05$).

In addition, we also employed a non-parametric effect size measure, the Vargha and Delaney's $\hat{A}_{12}$ statistic [41] to assess whether the effect size is interesting. The $\hat{A}_{12}$ measure is chosen since it is agnostic to the underlying distribution of the data, and is suitable for assessing randomized algorithms in software engineering generally [85] and effort estimation in particular [6]. Specifically, given a performance measure (e.g. the Absolute Error from each prediction in our case), the $\hat{A}_{12}$ measures the probability that predictive model M achieves better results (with respect to the performance measure) than predictive model N using the following formula: $\hat{A}_{12} = (r_1/m - (m+1)/2)/n$ where r_1 is the rank sum of observations where M achieving better than N, and m and n are respectively the number of observations in the samples derived from M and N. If the performance of the two models are equivalent, then $\hat{A}_{12} = 0.5$. If M perform better than N, then $\hat{A}_{12} > 0.5$ and vice versa. All the measures we have used here are commonly used in evaluating effort estimation models [6], [85].

B.2 Precision/Recall/F-measures/AUC

A confusion matrix is used to store the correct and incorrect classifications for each individual class made by a predictive model. For example, the confusion matrix for class *under achieved* in predicting the target velocity against the actual velocity delivered is constructed as follows. If an iteration is classified as *under achieved* when it truly delivered below than the target, the classification is a true positive (tp). If the iteration is classified as *under achieved* when it is actually *over achieved* or *achieved*, then the classification is a false positive (fp). If the iteration is classified as not *under achieved* when it in fact deliver below than the target, then the classification is a false negative (fn). Finally, if the iteration is classified as not *under achieved* and it in fact is did not deliver below the target, then the classification is true negative (tn). The values of each individual class classification stored in the confusion matrix are used to compute the widely used Precision, Recall, and F-measure [86]. In addition, we used another measure, Area Under the ROC Curve (AUC), to evaluate the degree of discrimination achieved by the model.

- Precision: The ratio of correctly predicted as a given class over all the iteration predicted as a given class. It is calculated as:

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

- Recall: The ratio of correctly predicted as a given class over all of the actually iteration in a given class. It is calculated as:

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

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

$$F - measure = \frac{2 * precision * recall}{precision + recall}$$

- 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 predictor.

B.3 Matthews correlation coefficient (MCC)

To compare the performance between two cases, using only F-measure can mislead the interpretation (e.g. very high precision but very poor recall), especially in cases of class imbalance. We thus also used Matthews correlation coefficient (MCC) [87]. MCC takes into account all true and false positives and negatives values (tp, tn, fp, and fn) and summarizes into a single value. It is also generally known as a balanced measure which can be used even if the classes are very different sizes [88]. MCC is defined as:

$$MCC = \frac{tp \times tn - fp \times fn}{\sqrt{(tp + fp) \times (tp + fn) \times (tn + fp) \times (tn + fn)}}$$

MCC has a range of -1 to 1 where -1 indicates a completely wrong classifier while 1 indicates a completely correct classifier, and 0 is expected for a prediction that no better than random.

B.4 Macro-averaged Mean Absolute Error (MMAE)

Since the classes in each prediction task could be considered as ordinal, we can order them, e.g. *over achieved* is better than *achieved*, and *achieved* is better than *under achieved*. However, the traditional class-based measures (e.g. Precision and Recall) do not take into account the ordering between classes. Hence, we also used another metric called Macro-averaged Mean Absolute Error (MMAE) [42] to assess the distance between actual and predicted classes. MMAE is suitable for ordered classes and insensitive to class imbalance. For example, as we discussed that the ordering of our classes is based on the performance of an outcome. Let y^i be the true class and $\hat{y}^i$ be the predicted class of iteration i in the task. Let n_k be the number of true cases with class k where $k \in \{1, 2, 3\}$ – there are 3 classes in our classification – i.e., $n_k = \sum_{i=1}^{n} \delta\left[y^i = k\right]$ and $n = n_1 + n_2 + n_3$. The Macro-averaged Mean Absolute Error is computed as follows.

$$\text{MMAE} = \frac{1}{3} \sum_{k=1}^{3} \frac{1}{n_k} \sum_{i=1}^{n} \left|\hat{y}^i - k\right| \delta\left[y^i = k\right]$$

For example, if the actual class is *over achieved* ($k = 1$), and the predicted class is *under achieved* ($k = 3$), then an error of 2 has occurred. Here, we assume that the predicted class is the one with the highest probability, but we acknowledge that other strategies can be used in practice. We however note that the ordering of the classes can be changed based on the project settings. For example, the achieved class may be more preferred than the over achieved class since over-achieving is not suggested over staying within budget.