Noname manuscript No.
(will be inserted by the editor)

High-level Software Requirements and Iteration Changes: A Predictive Model

Kelly Blincoe · Ali Dehghan · Abdoul-Djawadou Salaou · Adam Neal · Johan Linaker · Daniela Damian

Received: 07 Nov 2017 / Accepted: 19 Jul 2018

Abstract Knowing whether a software feature will be completed in its planned iteration can help with release planning decisions. However, existing research has focused on predictions of only low-level software tasks, like bug fixes. In this paper, we describe a mixed-method empirical study on three large IBM projects. We investigated the types of iteration changes that occur. We show that up to 54% of high-level requirements do not make their planned iteration. Requirements are most often pushed out to the next iteration, but high-level requirements are also commonly moved to the next minor or major release or returned to the product or release backlog. We developed and evaluated a model that uses machine learning to predict if a high-level requirement will be completed within its planned iteration. The model includes 29 features that were engineered based on prior work, interviews with IBM developers, and domain knowledge. Predictions were made at four different stages of the requirement lifetime. Our model is able to achieve up to 100% precision. We ranked the importance of our model features and found that some features are highly dependent on project and prediction stage. However, some features (e.g., the time remaining in the iteration and creator of the requirement) emerge as important across all projects and stages. We conclude with a discussion on future research directions.

Keywords software requirements · completion prediction · release planning · mining software repositories · machine learning

K. Blincoe
University of Auckland, New Zealand E-mail: kblincoe@acm.org

A. Dehghan, A. Salaou, · D. Damian
University of Victoria, BC, Canada E-mail: {dehghan, danielad}@uvic.ca

A. Neal
Persistent Systems, ON, Canada E-mail: nealadam@ca.ibm.com

J. Linaker
Lund University, Sweden E-mail: johan.linaker@cs.lth.se

1 Introduction

Planning when to release a software product and what new features or changes should be included in each release involves complex decisions [20,52]. Delivering frequent and incremental releases is a common practice of teams adopting agile practices [4]. Many agile teams deliver a new release after every software development cycle (or iteration) [34]. Frequent releases deliver value to the users quickly. In the highly competitive software environment, time-to-market for software features is paramount [22]. An early release can differentiate one product from its competition, while a late release allows competitors to grasp more of the market share [55]. Thus, late or incomplete releases can impact product success [4].

Planning releases to deliver new features as quickly and early as possible is, however, a challenging task. A recent mapping study indicates that implementing planned requirements on time is challenging because of difficulties in prioritizing requirements, estimating requirements' implementation effort, and understanding requirements' complexity [26]. Requirements can also be delayed due to growing technical debt accumulated from an inadequate, short-term planning horizon which is common on agile projects [26]. Since delivering features early is crucial for product success, software teams want to dedicate their time and resources to the requirements that will be completed in the current iteration and, as such, ready for the next release. Therefore, it would be useful, for release planning and resource allocation decisions, to know early when certain requirements will not make it into an iteration.

While there is an extensive body of literature on predicting various aspects of software releases and tasks, there are no studies that investigate the likelihood of a high-level requirement being implemented in its planned iteration. Much previous work has investigated ways to predict completion time [42, 21,1,36,15,30] and completion effort [58,5,45,12] of software tasks. However, this prior research focuses on low-level tasks such as bug fixes. In this paper, we focus on high-level requirements, which are typically broken into smaller software tasks. Predictions of high-level requirements can not be made simply by combining the predictions of all of their children, because the child tasks are often worked on in parallel and have dependencies that add additional coordination overhead.

Prior studies have also proposed models to predict the overall readiness of a software release [47,59,2,4]. This provides a very high-level view of when a release can be delivered. The work described in this paper differs from these studies since we focus on the completion of individual high-level requirements.

Predicting the completion of high-level requirements is important because often a high-level requirement is not released until all of its child tasks have been completed. Thus, project management decisions on all of the child tasks associated with a high-level requirement can depend on the likely completion time of the overall high-level task.

Furthermore, unlike tasks, high-level requirements are unique from a business and project management perspective. Delays in completing high-level requirements increase the need to manage customers' functionality delivery expectations. Any iteration changes to high-level requirements need to be appropriately managed in the project plan and discussed with the customers if necessary.

In this paper, we describe a mixed-method empirical study of requirements from three large projects at IBM. We used a combination of qualitative and quantitative methods. We worked closely with an IBM Analytics Architect (the fourth author) and other IBM practitioners. This close connection allowed us to obtain a concrete understanding of the IBM ecosystem and their workflow [41].

We developed predictive models to predict whether or not a high-level software requirement will be completed in its planned iteration. Our IBM partners requested that we develop a model that can optimize precision (at the expense of recall). The reason for this is that they do not want to make release planning or resource allocation decisions unless they can be certain the predictions are correct. They would prefer a model that provides a smaller set of very accurate predictions of requirements that should be replanned over a model that provides a large set of less accurate predictions. We used cost sensitive learning to satisfy this requirement of high precision. Thus, we developed two models during our investigation: 1) a cost insensitive model which does not aim to optimize precision and 2) a cost sensitive model which does aim to optimize precision. We compare the overall accuracy of the two models.

The development of our predictive model was guided by the following research questions:
RQ1: Can we predict whether or not a requirement will be completed within the planned iteration?
RQ2: Can we optimize the predictive model to maximize precision of predictions, while maintaining an acceptable recall?
RQ3: What is the relative importance of each model feature[1]?

We also aimed to better understand the types of changes that happen to high-level requirements. This part of our study was guided by the following research question:
RQ4: What happens to requirements that are not completed in their planned iteration?

Our study has four main contributions:

1. We analyze how often iteration changes occur and identify the types of iteration changes that commonly occur.
2. Through a process of feature engineering, we propose a predictive model that is capable of making predictions at different stages of a requirement lifetime and results in an F1-score between 0.55 and 0.80.

[1] Note that the term feature refers to a model feature, not to be confused with a software feature.

3. We optimize the predictive model to maximize precision of predictions to address IBM business interest. This model obtains precision values between 0.80 and 1.
4. We rank the engineered features according to their relative importance to our optimized model. This helps other researchers know what features to consider in their future studies. It also helps software organizations know what kind of data they should record for future analysis.

Our previous conference publication reported on some elements of this work [13]. However, this paper introduces numerous extensions to our work. Specifically, this paper extends our previous publications by investigating what happens to requirements when they do not make their planned iterations (RQ4). We identify eleven different types of iteration changes. Many high-level requirements are just moved to the next iteration when they are not completed in the current iteration, but they can also be moved to the next release, a later release, a previous release or put back on to the release or product backlog. We report on the frequency of each type of iteration change. We find that the type of work item has an effect on the type of iteration change that is made. We also restructured the paper, added additional details of the study throughout, and significantly expanded the related work and discussion sections, including a large discussion on potential avenues for future work enabled by this study.

2 Research Methodology

To answer our research questions, we used a mixed-methods approach comprised of interviews with practitioners, repository analysis, and development of predictive models. The study was led and designed by our research team in collaboration with an IBM Analytics Architect. The idea and the initial design of the study were based on a real business need at IBM and were initiated in a face-to-face meeting with an IBM program director and the IBM analytics architect. In this section, we discuss our research setting, dataset, and methodology.

2.1 Research Setting

We studied three large projects of an IBM product. The projects are codenamed as projects A, B and C due to confidentiality reasons. The product is an enterprise platform (referred to as the IBM enterprise platform henceforth). The platform is a software tool which provides a community where developers of the IBM ecosystem and customers of the product can collaborate and communicate. Members of this community can create, modify, resolve or comment on a work item. The platform is developed in Java. IBM adopts an agile methodology for development of this product.

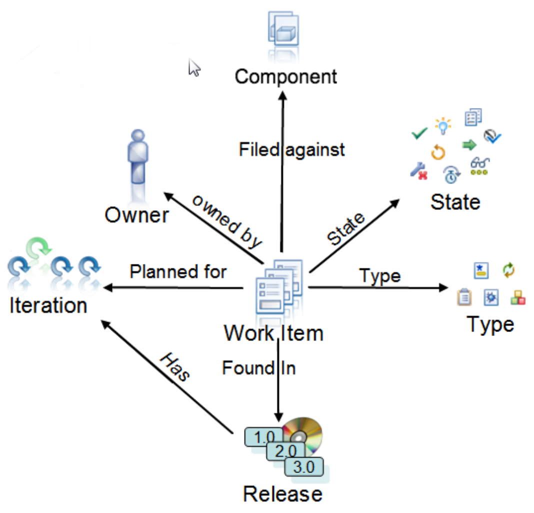


Fig. 1: IBM enterprise platform: Work Item attributes

We describe the context of the product according to the guidelines in [44].

– *Maturity:* The development was started in 2005 and the first release of the product was made in 2008. It has had six major releases and a large number of minor releases.
– *Quality:* Today, the development teams continue to add enhancements to the product and maintain the existing releases.
– *Size:* The product is large and provides many different components. Some indication of size is given by the number of developers and tasks (work items) in Table 1.
– *System type:* The product is available as a web application. In addition, clients are available for multiple platforms and technologies.
– *Customization:* The product has role-based licensing options, allowing customers to choose their desired components. The product itself provides many customization options allowing it to suit a wide-range of use cases.
– *Programming language:* The product is developed in Java.

To understand the research setting, we define some terminology:

WORK ITEMS represent work that needs to be done. The main attributes of a work item are illustrated in Figure 1. Work items have an owner (the assigned developer), are filed against specific components of the product, are planned for iterations, and have a state. The state of a work item describes its current status. A work item has a type attribute which defines the workflow of the work item. Work item types are: Plan Item, Story, Enhancement, Task and Defect. Work items vary in size from small chunks of work to very large. Typically, Plan Items are the largest work items and tasks and defects are the smallest.

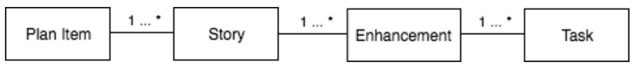


Fig. 2: Hierarchy of work items in IBM enterprise platform

Work items are also hierarchical, which means a work item could have one or multiple children or grandchildren. The ideal hierarchy of work items in the IBM enterprise platform is shown in Figure 2. Work items of type Defect could be a child of any of these other work item types. The hierarchy in practice, however, is sometimes inconsistent with this ideal structure. For instance, work items might have children with the same work item type (e.g., a Story could have a child of type Story), and, in some rare cases, this hierarchical order is violated.

Plan items are top level work items that represent software requirements and functionalities that should be included in next release(s). An example of a Plan Item is adding support for Visual Studio to a product. These are very large development efforts. These are most similar to Epics in agile methodologies. During the lifetime of a Plan Item, new children and grandchildren can always be added. A Plan Item is not completed until all of its children are completed. These are typically planned by high-level management across the entire project.

Stories are also high level work items, but they are breakdowns of Plan Items. A Plan Item typically is broken into multiple Stories. Team leads manage the completion of Stories. These are most similar to User Stories in agile methodologies. Using the example of adding support for Visual Studio to a product, this would be broken into multiple user stories for each feature that must be supported in Visual Studio. Stories and Plan items have many common characteristics in this platform, but Stories tend to be somewhat smaller than Plan Items.

Work item History: Every time a change happens to a direct attribute of a work item (for instance if the status or owner of a work item changes) or a new subscriber or comment is added to a work item, this change is recorded as a work item history. However, the addition or removal of children is not recorded as a work item history.

Iteration: A time-box during which development takes place. Iterations are hierarchical where top-level iterations represent releases and child iterations represent milestones within those releases. A milestone iteration is typically 1 to 4 weeks in length. The duration of release iterations vary from 1 month up to 1 year. Work items could be planned for a milestone or a release iteration, and if they don't get finished through an iteration, they could be replanned for another one.

Table 1: Summary of Projects

Attribute	**Project A**	**Project B**	**Project C**
start date	Jun 2006	Jan 2009	Jun 2006
end date	Oct 10, 2016	Oct 11, 2016	Oct 24, 2016
# work items	75k	71k	177k
# histories	816k	835k	1.73m
# plan items	839	749	447
# PI histories	20k	20k	19k
# stories	1,286	3,640	4,471
# S histories	16k	61k	50k
# comments	374k	312k	312k
# developers	594	481	796

2.2 Project Selection

The three projects were chosen, in consultation with IBM practitioners, based on their long duration, high-level of activity, and large number of Plan Items and Stories in their development and project management repositories. Table 1 shows some statistics of these three projects in our observation period. Note that as the type of a work item is subject to change, the number of Plan Items and Stories displayed in Table 1 is based on any work item that has at least one history of that type.

2.3 Repository Analysis

To investigate requirement iteration changes, we performed a repository analysis of the three IBM projects. We focused our analysis on Plan Items and Stories for two reasons: 1) IBM managers stated they would like to have predictions for both of these work item types, and 2) there are many inherent similarities between the two work item types in terms of measured attributes as well as the enterprise internal processes. We refer to Plan Items and Stories as requirements in the rest of this paper. However, as the two work item types do have some differences (stories are breakdowns of Plan Items), we perform our analysis on Plan Items and Stories separately.

We studied the requirements that are already completed and were planned for an iteration with an end date. These filters allowed us to validate our predictions based on actual completion times. Thus, the results displayed in Tables 4 and 5, show only the subset of requirements that have been completed.

2.4 Interviews

The study was done in close collaboration with IBM. As previously stated, the original idea for the study was initiated by IBM. Throughout the study, we held two face-to-face meetings with the IBM program director and the analytics architect and 16 semi-structured 30-60 minute interviews with the analytics architect and three other IBM practitioners. These interviews were used to drive the study design and validate our understanding of the repository data. Features for the predictive model were brainstormed and discussed in these interviews.

2.5 Model Development

Using feature engineering, we developed a model that predicts whether a high-level requirement will be completed during its planned iteration. The model features were derived from prior work, from suggestions made by IBM developers, and through domain knowledge. We used Random Forest (RF) as our learning algorithm due to its ability to handle different feature types, noise, missing values, and correlated features [60]. We used cost sensitive learning to favor precision over recall as desired by the IBM practitioners [56]. The development of the model is described in detail in Section 3.1.

We used a variety of tools and libraries for our analysis including RapidMiner [31] and WEKA [24]. The final implementation was done in Java code using WEKA libraries.

In the next section, we describe the data analysis and results in answering our research questions.

3 Data analysis and Results

3.1 Predictive Model

RQ1: Can we predict whether or not a requirement will be completed within the planned iteration?

In this section, we describe the development and evaluation of a model that predicts whether a high-level requirement will be completed during its planned iteration.

3.1.1 Feature Engineering

Feature engineering is the process of creating predictors for a machine learning algorithm in order to build a predictive model. Feature engineering was the most effort-consuming task in this study and involved iterative brainstorming, data visualization, digging into data, interviews with IBM software practitioners, and a review of existing literature. As a result, a set of 29 features were engineered. Each of these model features come from one or a multiple of these sources:

1. Prior work, chiefly in bug resolution time and effort prediction.
2. Suggestions made by IBM developers.

3. Domain knowledge achieved as a result of interviews with enterprise practitioners and studying the data.

The motivation behind choosing these features and their descriptions are stated in Section 3.1.2. Despite differences in practices across teams and inherent differences in the characteristics of the two work item types, the commonalities between them were high enough to enable us to use almost the same feature set for each dataset, except for a few minor exceptions which will be described in Section 3.1.2.

No Manual Feature Selection: We ran Pearson correlation analysis between each pair of features, the highest correlation values were between 0.6 and 0.8 which can still be handled by the RF algorithm. The RF algorithm is designed to be robust against correlated or non-informative features [60]. Thus, as one of the objectives of this research was ranking features based on their importance to the trained models, we did not exclude any of the engineered features.

No Automatic Feature Generation: Besides manual feature engineering, there are automatic techniques such as automatically generating large numbers of candidate features and selecting the best by their information gain, but these techniques can cause over-fitting [14], especially when dealing with a large number of attributes in raw data.

No Automatic Feature Subset Selection: Additionally, there are methods that automatically select the ideal subset of features. We used correlation-based features subset selection methods and wrapper for feature subset selection methods in the WEKA [24] library. We used these methods to select a subset out of the 29 manually engineered features for the RF classification model. For each dataset, we built ten subsets by varying the method parameters and kept features that appeared at least six times. We used these subset in our models and compared the results across subsets and with the original model. In all cases, the model with all 29 features outperformed those with automatic feature subset selection. Thus, the reported results use all 29 features.

3.1.2 Model Features

In this section, we introduce all the engineered features used in our analysis and motivate them. There are 29 features, which we categorized into 9 logical categories. Some features logically fit into more than one category. In those cases, we put them in the most relevant category only. In the following sections, we describe each category and the motivation for including each feature. We also cite prior work that has adopted similar features in other prediction models. Note that a citation right after the feature name indicates that the feature or a similar one was used in the referenced prior work, but it does not necessarily mean that they had the same motivation as in this work since we had different goals and settings in this study.

General Features

There are three features that are available early on in a requirement lifetime. We call these the general features.

– CREATOR_IDENTIFIER [42,21,1,36,30,12,23,49]: the stakeholder who created the requirement. Available at requirement creation.
– CREATION_MONTH [21,1,36,12]: the month the requirement was created. Available at requirement creation. Depending on the project and timeline of the corresponding team, it could capture factors such as workload of a team that could have impact on completion time of a requirement.
– OWNER_IDENTIFIER [42,21,1,23,36]: the developer assigned to a requirement. Available when the requirement is assigned to a developer.

Complexity Indicating Features

The more complex a requirement is, the more time it is likely to take. Misunderstood requirements often lead to technical debt and the need for rework [26]. Thus, we include features which can indicate the complexity of a requirement.

– SUBSCRIBER_COUNT [42,21,1,36,30]: the number of stakeholders subscribed to receive updates on a certain work item. Whenever someone comments on a work item, they are automatically subscribed to the work item. Stakeholders could also manually subscribe to or unsubscribe from work items. A high interest in receiving updates could indicate high complexity.
– FILED_AGAINST [42,21,1,36,12]: the software component against which the requirement is filed. Some components are inherently more complicated than others. For instance a database related component is likely to entail more complexity than a UI related component.
– ITERATION_CHANGE_COUNT: the number of times a requirement was replanned for a new iteration, as it suggests possible inadequate effort estimation [26]. If the requirement gets carried over to the next iteration multiple times, it could be an indication of high complexity.

Progress Implying Features

The current progress of a requirement will influence whether it will be completed in time or not. Thus, we have three features that enable the model to gauge current progress:

– ITERATION_DAYS_REMAINED [21,1,2,4]: the number of days remaining to the end of the planned iteration.
– DAYS_SINCE_CREATION [42,21,1]: the number of days since the work item was created.
– STATUS [21,1,23]: the current status of a work item, such as new, in exploration phase, in progress, in testing phase, etc.

Priority Implying Features

Having a higher priority usually helps a requirement receive more attention and activity and, thus, get completed earlier.

- PRIORITY: an explicit measure of importance from the developers perspective. Not available for all work items. In cases of missing values, priority was assumed to be normal, which is the most common priority.
- SEVERITY: an explicit measure of importance from the customers perspective. Not available for all work items. In cases of missing values, severity was assumed to be normal, which is the most common severity.
- DAYS_WITHOUT_OWNER: number of days the work item was not assigned to a developer since creation; many unassigned days could indicate low priority.
- DAYS_SINCE_LAST_COMMENT: the number of days since the last comment, many days between comments could indicate low priority.
- OWNER_CHANGE_COUNT [23,30,28]: the number of times a work item has been reassigned. Reassignments could be an indication of high priority as people reassign a task to find the optimal developer who can address the problem quickly [23]. However, too many reassignments could mean that no one is taking responsibility for handling the task [28], indicating low priority.
- DAYS_SINCE_LAST_OWNER: the number of days since the current work item owner was assigned. This is meant to capture a similar effect to the previous feature by considering how recent the last reassignment is.

Problem Change Indicating Features

A change in the problem definition of a requirement can impact completion time. Frequent changes could indicate additional changes in the future. This is captured by:

- SUMMARY_CHANGE_COUNT: the number of times the work item summary has changed.
- DESCRIPTION_CHANGE_COUNT: the number of times the work item description has changed.

Also, if the problem definition has changed recently, effort may still be underway to deal with the change. Thus, we consider:

- DAYS_SINCE_LAST_SUMMARY: the number of days since the last summary change.
- DAYS_SINCE_LAST_DESCRIPTION: the number of days since the last description change.

Previous studies attempted to capture changes in requirements by proposing basic features such as the number of total changes to bug attributes [21, 1] (equivalent to number of histories in our datasets), but to the best of our knowledge these particular four features are novel to this study.

Process Change Indicating Features

In addition to a change in problem definition, a change in the software process could also impact completion time. This is especially true if the process change is recent. It's not uncommon in the IBM enterprise platform for a Story to be transformed to a Plan Item and vice a versa. It also sometimes happens that a work item is created as a low-level requirement (Task) and then the developers decide that it should be transformed to a high-level requirement (Plan Item or Story). Such process changes could potentially delay implementation of a requirement.

- DAYS_SINCE_LAST_TYPE_S_P: number of days since the last time the requirement type was changed from Plan Item to Story or vice a versa.
- DAYS_SINCE_LAST_TYPE_CHILD: number of days since the last time the work item type was changed from a low-level work item type to a requirement.

Stakeholder Characteristics

Through the interviews with IBM practitioners, we learned that the types of stakeholders involved in a work item can impact its completion time. To account for this, we consider:

- DAYS_SINCE_LAST_DE_COMMENT: number of days since a distinguished engineer (DE) commented on the work item. In IBM, DE is a title reserved for very respected developers. If a DE participates in the discussion of a work item, it will likely receive prompt attention.
- COMPONENT_RESOLVER: the total number of work items that the work item owner has resolved in the same component up until the modified date of the corresponding history of each requirement. This indicates the expertise of the owner in the domain of the problem.
- CREATOR_TEAM_RELATIONSHIP [23,36]: the relationship of the work item creator to the assigned team. Prior work suggests that bugs reported by people on the same team are likely to get fixed faster [23]. For this feature, the creator could be an IBM developer from the same team, an IBM developer from another team, or a customer from outside the company.

Stakeholder Communication

Communication between stakeholders of a work item can impact completion time. A large volume of communication may indicate high complexity and, thus, result in longer completion time [23]. On the other hand, increased communication can indicate a good information flow and a high level of awareness and engagement and, thus, results in shorter completion time [23]. Communication in this enterprise platform usually happens via commenting on work items, so we consider:

- COMMENT_COUNT [42,21,1,36,30]: number of comments on the work item.

– COMMENTER_COUNT [30]: the number of unique stakeholders involved in the discussion on a work item.

Child Features

As described in Section 2.1, work items are hierarchical in the IBM enterprise platform. Requirements normally have children and grandchildren. Our model considers these hierarchical dependencies since the number of children a requirement has can impact its completion time for a variety of reasons, e.g.:

1. child work items are a breakdown of the requirement,
2. the number of children impacts the effort required for planning and breaking down the requirement as well as the effort required to integrate and test the children,
3. each child has its own idle time (such as time spent on triaging and assignment).

The type of a child work item can play an important role in this effect since plan items and stories often involve a more significant amount of work. We also want to consider the progress of the children, since a child work item that has been completed or is near completion should not greatly impact the completion time of the parent. Therefore, we consider:

– SAME_TYPE_CHILD_COUNT_NEW: the number of first descendant child work items of the same type as the requirement, with progress status New
– LARGE_SIZE_CHILD_COUNT_NEW : the number of first descendant child work items of type Story (or Enhancement if requirement type is Story), with progress status New
– MEDIUM_SIZE_CHILD_COUNT_NEW: the number of first descendant child work items of type Enhancement (or Task if requirement type is Story), with progress status New

Table 2 summarizes the model features. As can be seen, while many features are based on other prediction models, a large number of features are new to this study. In addition, the table illustrates there are five features that are specific to only high-level requirements, the focus of this study. All of the features related to children (LARGE_TYPE_CHILD_COUNT_NEW, MEDIUM_TYPE_CHILD_COUNT_NEW, SAME_TYPE_CHILD_COUNT_NEW) are applicable only to high-level requirements, since low-level requirements do not have children. In addition, the process change indicating features (DAYS_SINCE_LAST_TYPE_S_P and DAYS_SINCE_LAST_TYPE_CHILD) are specific to high-level requirements as they relate to requirements changing to (or from) a high-level requirement type.

3.1.3 Learning Algorithm

We used Random Forest (RF) as our learning algorithm. RF has shown high performance on many types of datasets compared to many well-known algorithms such as SVM and Naive Bayes [18]. It is robust to noise in data, is

Table 2: Model Features.

Feature	**Based on**	**Applicable to (requirement level)**	
		Low	**High**
CREATOR_IDENTIFIER	[42,21,1,36,30,12,23,49]	✓	✓
CREATION_MONTH	[21,1,36,12]	✓	✓
OWNER_IDENTIFIER	[42,21,1,23,36]	✓	✓
SUBSCRIBER_COUNT	[42,21,1,36,30]	✓	✓
FILED_AGAINST	[42,21,1,36,12]	✓	✓
ITERATION_CHANGE_COUNT	—	✓	✓
ITERATION_DAYS_REMAINED	[21,1,2,4]	✓	✓
DAYS_SINCE_CREATION	[42,21,1]	✓	✓
STATUS	[21,1,23]	✓	✓
PRIORITY	—	✓	✓
SEVERITY	—	✓	✓
DAYS_WITHOUT_OWNER	—	✓	✓
DAYS_SINCE_LAST_COMMENT	—	✓	✓
OWNER_CHANGE_COUNT	[23,30,28]	✓	✓
DAYS_SINCE_LAST_OWNER	—	✓	✓
SUMMARY_CHANGE_COUNT	—	✓	✓
DESCRIPTION_CHANGE_COUNT	—	✓	✓
DAYS_SINCE_LAST_SUMMARY	—	✓	✓
DAYS_SINCE_LAST_DESCRIPTION	—	✓	✓
DAYS_SINCE_LAST_TYPE_S_P	—		✓
DAYS_SINCE_LAST_TYPE_CHILD	—		✓
DAYS_SINCE_LAST_DE_COMMENT	—	✓	✓
COMPONENT_RESOLVER	—	✓	✓
CREATOR_TEAM_RELATIONSHIP	[23,36]	✓	✓
COMMENT_COUNT	[42,21,1,36,30]	✓	✓
COMMENTER_COUNT	[30]	✓	✓
SAME_TYPE_CHILD_COUNT_NEW	—		✓
LARGE_SIZE_CHILD_COUNT_NEW	—		✓
MEDIUM_SIZE_CHILD_COUNT_NEW	—		✓

applicable to datasets with a mixture of continuous, semi-continuous and categorical features, and is capable of handling missing values as well as correlated features [60].

Model Parameters: In RF, there are three main parameters. The number of trees, the maximum depth of each tree, and the the number of randomly selected features at each split. For the sake of the model performance, we tune those parameters by trying different combinations. All datasets obtain the best result using 100 trees with a maximum depth of 50 depth and 5 randomly selected features.

3.1.4 Prediction Stages

We made predictions at four different stages for each project and work item type: the first day of creation of a requirement and the end of the first, second and third quarter of the planned iteration. This allows us to make predictions at different stages of a requirement lifetime, something that is beneficial to enterprises. It also enables us to compare the significance of features at different points in time. The selected stages are meaningful and derived from the actual needs of the enterprise. Another benefit to this approach is that training and testing data points will be automatically selected based on the same percentage of the progress of requirements. Therefore, these data points are more meaningful and related to each other, assuming that requirements at the same time slot of their corresponding iteration are approximately at the same point of their progress. For simplicity, we will onwards refer to these four prediction stages with 0th, 1st, 2nd and 3rd short form notations.

Considering that work items in this platform might have many histories within the same quarter of an iteration, we defined the history selection criteria as following:

- 0th: The first history of a work item.
- 1st, 2nd and 3rd: The last history of a work item within the corresponding quarter of its planned iteration.

As a result of having three different projects (A, B and C), two different requirement types (Plan Item, Story) and four different prediction stages (0th, 1st, 2nd, 3rd), we have 24 datasets in total. For simplicity from now on, we will refer to a specific dataset by the Project-Type-Stage notation. For instance B-plan-2nd will refer to the dataset of Plan Items of project B at the end of the 2nd quarter of their planned iteration.

3.1.5 Binary Classification Problem

In terms of the prediction outcome, we formulated our analysis as a binary classification problem. We compared the completion date of a requirement to the end date of the planned iteration. Equation (1) shows the derived prediction outcome:

$$iteration\_met = \begin{cases} YES, & \text{if } completion\_date \leq end\_date \\ NO, & \text{otherwise} \end{cases} \tag{1}$$

As we assumed that requirements are supposed to be completed within their planned iteration, a trend we confirmed from our examination of the historical data of the projects, the NO class normally made up the minority of

Table 3: Ratio of requirements that are not completed in their planned iteration

dataset	**0th**	**1st**	**2nd**	**3rd**
A - plan	0.53	0.44	0.49	0.49
B - plan	0.30	0.54	0.52	0.32
C - plan	0.35	0.38	0.39	0.39
A - story	0.50	0.38	0.40	0.40
B - story	0.44	0.20	0.19	0.18
C - story	0.50	0.37	0.36	0.36

Table 4: Cost Insensitive Model results

		Plan Items						Stories					
proj	stage	n	NO%	prec	recall	F1	WA	n	NO%	prec	recall	F1	WA
A	0th	316	53%	.81	.70	.75	.78	521	50%	.72	.69	.71	.71
	1st	393	44%	.85	.74	.79	.82	769	38%	.75	.60	.67	.71
	2nd	462	49%	.84	.74	.79	.81	842	40%	.72	.62	.66	.69
	3rd	468	49%	.85	.75	.80	.82	873	40%	.73	.61	.67	.70
B	0th	231	30%	.65	.50	.57	.61	2020	44%	.68	.67	.67	.67
	1st	224	54%	.68	.73	.70	.69	2472	20%	.81	.53	.64	.74
	2nd	287	52%	.75	.71	.73	.74	2759	19%	.86	.58	.69	.79
	3rd	473	32%	.73	.45	.56	.66	2925	18%	.85	.59	.69	.78
C	0th	77	35%	.73	.44	.55	.66	1644	50%	.73	.73	.72	.72
	1st	267	38%	.73	.63	.68	.70	1999	37%	.68	.53	.60	.64
	2nd	280	39%	.72	.65	.69	.71	2310	36%	.70	.50	.58	.65
	3rd	287	39%	.70	.59	.64	.67	2422	36%	.70	.51	.59	.65

requirements. Thus, NO was considered as the positive class for our learners to predict. Table 3 displays the ratio of requirements that are not completed in their planned iterations for all datasets.

3.1.6 Model Validation

We used K-fold cross validation to evaluate our model. K-fold cross validation is a common technique for evaluating the performance of a predictive model. Values such as 10 or 5 as number of folds (k) are typically known as good choices according to the variance-bias trade-off [19,17]. In our experiments, we used 10-fold cross validation.

Table 4 shows the performance of our cost insensitive learning process in each of the 24 datasets measured by precision and recall of the positive class. We also report the F1-score, which is the harmonic mean of precision and recall, as well as the Weighted Arithmetic mean (WA) as defined in section 3.2.2. From Table 4, it is evident by the NO% column, which shows the ratio of the positive class to the dataset size, that there is some skewness in our datasets. Despite this, our predictions result in precisions higher than 0.65 for all projects, work item types and stages. In some cases, precision is as high as 0.86. F1-scores are also high with all datasets obtaining an F1-score of at least 0.55 and some datasets achieving scores as high as 0.80. The one dataset that received a lower F1-score, C-plan-0th, is the smallest dataset.

Comparing the results between the two work item types, we observe fairly similar prediction performances for both Plan Items and Stories.

One key factor to take into account when comparing results, however, is the balance of the dataset denoted by the column NO%. We expect to have better results when the dataset is more balanced since it allows for enough positives in the training set. We see a trend of having a more balanced dataset in the 0th prediction stage for most projects and work item types.

Answer to RQ1: We developed models that were able to accurately predict whether a requirement was completed within the planned iteration. We obtained precision scores of up to .86 and F1-scores up to .80 (see Table 4).

3.2 Precision Optimization

RQ2: Can we optimize the predictive model to maximize precision of predictions, while maintaining an acceptable recall?

IBM managers stated they are interested in having the highest possible precision, even though it might result in lower recall values. They stated that a precision of below 0.8 would be of little utility in practice and a precision of 0.9 or above would be ideal for them. At the same time, they pointed out that having a high precision, a recall value around 0.2 or 0.3 would be good enough. The reason for this is that they would like to be alerted of the most relevant requirements which are most likely to not be completed in their planned iteration. They want to ensure they are not reallocating time or resources away from a requirement that may actually be completed on time.

3.2.1 Cost Sensitive Learning

To favor precision over recall, we used cost sensitive learning with a high penalty for false positives to make our models more cautious when making predictions on the positive class. There exist many techniques for cost sensitive learning, most of which are described in a literature review by He and Garcia [25]. These techniques can be classified into two general categories: 1) cost sensitive learning which performs resampling or sample-weighting on the minority class to make the data more balanced; and 2) approaches that minimize the expected cost of classification utilizing the confidence of the base classifier in predictions. We use a sample-reweighting technique through the CostSensitiveClassiffier class of the WEKA [24] library, which is an implementation of the approach introduced by Ting [56]. We determined the false positive penalty by considering the original class balance to make it balanced. In addition, we increased the false positive penalty in order to comply with the goal of maximizing precision of predictions. As a result, a false positive penalty between 3 to 5 was applied depending on the skewness of dataset.

3.2.2 Weighted Arithmetic Mean

Based on the specified ideal precision and recall values desired by IBM, we used a weighted arithmetic mean to measure the performance of our cost sensitive model, shown in Equation (2).

$$WA = (3 \times precision(NO) + recall(NO))/4 \tag{2}$$

Table 5: Cost Sensitive Model results

		Plan Items					Stories				
proj	stage	n	NO%	prec	recall	WA	n	NO%	prec	recall	WA
A	0th	316	53%	.95	.47	.83	521	50%	.87	.36	.74
	1st	393	44%	.95	.47	.83	769	38%	.87	.39	.75
	2nd	462	49%	.93	.56	.84	842	40%	.83	.38	.72
	3rd	468	49%	.91	.66	.85	873	40%	.89	.38	.73
B	0th	231	30%	1	.24	.81	2020	44%	.86	.31	.72
	1st	224	54%	.88	.13	.69	2472	20%	.94	.24	.76
	2nd	287	52%	.95	.24	.77	2759	19%	.93	.34	.78
	3rd	473	32%	.94	.11	.73	2925	18%	.95	.33	.80
C	0th	77	35%	1	.08	.77	1644	50%	.85	.51	.76
	1st	267	38%	.86	.12	.67	1999	37%	.80	.27	.67
	2nd	280	39%	.82	.13	.65	2310	36%	.86	.15	.68
	3rd	287	39%	.86	.17	.69	2422	36%	.88	.17	.70

We still report performance metrics such as precision and recall. We use WA as defined above to compare performance of the models developed for RQ1 and RQ2. We call the model which optimizes precision the cost sensitive model and the original model the cost insensitive model.

3.2.3 Model Validation

Similar to RQ1, we used 10-fold cross validation to evaluate our model. Table 5 shows the precision, recall and weighted arithmetic mean (WA) of our cost sensitive learning process for each dataset. We see similar skewness across the datasets in regards to the NO% as seen in RQ1. We obtain precision vales of at least 0.80 for all projects, work item types and stages. In some cases, precision goes as high as 1. Of course, recall is lower than seen in RQ1, with a median recall of 0.29 Though, in some cases, recall values are as high as 0.66.

When comparing the WA of the cost sensitive model (Table 5) and the cost insensitive model (Table 4) results, we observe that without any exception, WA of the cost sensitive predictions are always equal to or greater than cost insensitive predictions. This shows that the cost sensitive models are overall more accurate while providing the high precision values desired by IBM.

Answer to RQ2: We developed models that achieved higher precision values, up to 1 (see Table 5), and obtained higher overall performance compared to RQ1 as measured by the WA scores.

Table 6: Feature Importance Ranking: Frequency each feature occurs in top 10 most important features across the 24 datasets.

Feature	**count**	**specific to high-level requirements**
iteration_days_remained	24	
owner_identifier	24	
creator_identifier	24	
filed_against	23	
creation_month	22	
status	21	
severity	14	
priority	14	
creator_team_relationship	13	
days_since_creation	9	
component_resolver	9	
days_since_last_comment	7	
days_without_owner	7	
iteration_change_count	6	
subscriber_count	5	
medium_size_child_count_new	4	✓
days_since_last_owner	3	
comment_count	3	
summary_change_count	3	
days_since_last_summary	2	
large_size_child_count_new	1	✓
owner_change_count	1	
creator_team_relationship	1	
description_change_count	1	
same_type_child_count_new	1	✓
days_since_last_type_s_p	1	✓
days_since_last_type_child	1	✓
days_since_last_de_comment	1	
days_since_last_description	1	

3.3 Feature Importance (RQ3)

RQ3: What is the relative importance of each model feature?

To rank feature importance, for each dataset, we use Weka wrapper subset attribute evaluator method with *BestFirst* as the search method and Random Forest as the classifier. Other techniques, such as Cohen's f^2 test [54] or Chi-square measure [35], can be used to measure the effect size on or relevance of features to the class feature out of the box. However they do not indicate a feature importance in the trained model and, thus, we did not employ these measures.

Table 6 shows the number of times each feature occurs in the top ten most important features across the 24 datasets. When interpreting these results, one should consider that the importance of the features in each dataset can be influenced by:

– Other dominant features. These feature importance values are relative to other features within the same dataset. A change in relative importance does not indicate a change in the absolute importance of that feature.
– The availability rate and cardinality of a feature. Some features might have a high number of missing values in some stages. It is also possible that some features have similar values and a low cardinality and, thus, provide lower information gain. While RF can handle these missing values, it could impact the feature importance in that dataset.
– The randomness factor of the random forest algorithm in selecting subsets of training samples as well as features when building individual trees [9].

As can be seen in Table 6, there are a subset of features which are stable across all or nearly all datasets. ITERATION_DAYS_REMAINED, OWNER_IDENTIFIER and CREATOR_IDENTIFIER are important in all datasets, which is in alignment with the findings of prior work [23]. We also found that FILED_AGAINST, CREATION_MONTH, and STATUS have very high importance in most datasets. The importance of the remaining features varies across projects and prediction stages.

Using the results shown in Table 6, we divided the attributes into four groups based on their frequency of occurrence:

1. The top 3 features: ITERATION_DAYS_REMAINED, OWNER_IDENTIFIER, and CREATOR_IDENTIFIER
2. The next three features: FILED_AGAINST, CREATION_MONTH, and STATUS
3. The next three features: SEVERITY, PRIORITY, and CREATOR_TEAM_RELATIONSHIP
4. all remaining features.

We then reran our cost sensitive predictive model (from RQ2) four times. We started by using only the most important features (group 1) and then, on each subsequent run, added the next most significant group of features to investigate if using only a smaller subset of our model features would produce similar results as those obtained with all model features. The median WA for each model is shown in Table 7. As can be seen, with the addition of each group of features, the median WA increases. We used Wilcoxon Signed-rank test to test if this increase was significant. The results, shown in Table 8 shows that the addition of groups 2 and 4 is indeed significant with a medium and large effect size, respectively. Thus, using only the most important model features does not result in as good of results as using the full set of features.

Table 9 shows detailed information on relative feature importance for the model trained on each dataset.

Table 7: Results of Cost Sensitive Predictive Model using subsets of Features

Groups	**median WA**
1	0.6835
1+2	0.7025
1+2+3	0.7065
1+2+3+4	0.7415

Table 8: Significance of adding each group of features (using the Wilcoxon Signed-rank test)

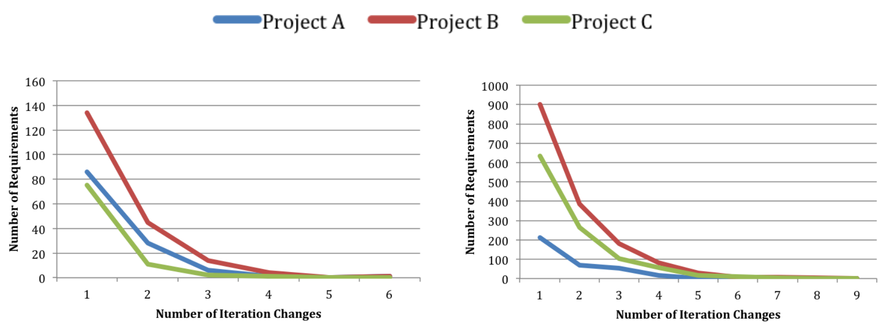
Group	**Z**	**p**	**r**
2	2.93	<0.01	0.42
3	0.50	0.63	0.07
4	4.11	<0.001	0.59

Many of the most important features are also applicable to low-level requirements and are not unique to high-level requirements. Many of these also were inspired by previous work on low-level requirements, but we have demonstrated their applicability on high-level requirements in this study. Though, as can be seen in Table 6, there are features specific to high-level requirements, like the MEDIUM_SIZE_CHILD_COUNT_NEW, that are sometimes very important for making predictions.

Answer to RQ3: Feature importance is project and prediction stage dependent. However, some features, such as ITERATION_DAYS_REMAINED, OWNER_IDENTIFIER and CREATOR_IDENTIFIER, are consistently ranked higher than others. Table 6 shows the relative importance of the 29 features included in our models.

Table 9: Relative Variable Importance Ranking per Dataset

Feature	Plan Items												Stories											
	A				B				C				A				B				C			
	0th	1st	2nd	3rd	0th	1st	2nd	3rd	0th	1st	2nd	3rd	0th	1st	2nd	3rd	0th	1st	2nd	3rd	0th	1st	2nd	3rd
creator_identifier	1	1	1	1	1	1	1	3	2	2	2	2	1	1	1	1	1	2	1	1	1	1	1	1
creation_month	4	5	5	4	12	6	5	11	5	4	4	4	4	5	7	6	4	4	3	3	5	5	4	4
owner_identifier	5	2	2	2	4	4	3	4	3	3	3	3	2	2	2	2	3	3	4	4	4	3	3	3
subscriber_count	7	12	10	9	11	12	13	12	14	14	15	14	6	16	18	10	14	17	13	13	14	12	12	17
filed_against	2	3	4	3	2	2	2	2	4	5	7	6	3	3	3	12	2	1	2	2	3	2	2	2
iteration_days_remained	3	4	3	5	3	3	4	1	1	1	1	1	5	7	6	3	5	5	5	5	2	4	5	5
days_since_creation	12	13	7	7	14	14	14	5	11	12	13	12	13	8	8	16	11	7	8	8	7	13	13	12
status	13	6	6	6	10	9	8	9	12	7	8	7	14	6	5	5	10	6	6	6	8	6	6	6
priority	8	8	12	14	6	11	9	15	8	8	10	9	7	11	14	13	8	21	23	22	9	7	7	8
severity	10	10	13	15	5	8	7	16	6	6	9	8	12	12	15	14	7	20	22	21	10	9	9	9
days_without_owner	11	11	15	17	9	13	12	8	9	10	12	11	9	26	23	20	12	14	15	28	6	10	11	19
days_since_last_comment	15	15	16	19	8	5	15	6	10	11	16	15	11	9	11	8	15	28	25	20	12	11	10	20
owner_change_count	14	14	18	25	15	15	11	19	15	15	14	13	15	27	28	26	13	15	21	24	15	15	15	23
summary_change_count	17	17	9	10	13	17	17	25	13	13	18	17	10	25	27	28	17	18	17	15	13	14	14	28
description_change_count	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	16	18	17	29	29	29	29
days_since_last_description	26	26	26	27	24	24	26	26	24	24	26	26	24	24	25	25	26	12	19	14	24	24	24	26
days_since_last_summary	27	27	8	8	26	26	27	27	26	26	27	27	26	23	24	22	27	27	14	19	26	26	26	25
iteration_change_count	28	28	27	28	27	27	28	28	27	27	6	28	27	21	10	9	28	10	10	10	27	27	27	24
days_since_last_owner	25	25	28	12	28	28	25	10	28	28	5	5	28	20	22	24	25	13	12	12	28	28	28	21
days_since_last_type_s_p	23	7	25	24	25	25	23	24	25	25	28	25	25	28	21	23	23	24	28	29	25	25	25	22
days_since_last_type_child	18	23	24	22	23	23	18	22	23	23	25	23	23	19	20	21	18	25	29	26	23	23	23	14
days_since_last_de_comment	22	18	23	23	18	18	22	23	18	18	24	18	18	15	17	17	22	22	27	23	18	18	18	13
creator_team_relationship	9	9	14	16	7	10	10	17	7	9	11	10	8	13	16	15	9	19	20	16	11	8	8	10
component_resolver	6	22	20	20	22	22	6	20	22	22	23	22	22	4	4	4	6	8	7	7	22	22	22	11
comment_count	19	19	21	13	19	19	19	13	19	19	20	19	19	14	19	19	19	9	9	9	19	19	19	18
commenter_count	20	20	11	11	20	20	20	7	20	20	21	20	20	17	12	18	20	11	11	11	20	20	20	16
same_type_child_count_new	21	21	22	21	21	21	21	21	21	21	22	21	21	18	13	11	21	23	26	25	21	21	21	15
large_size_child_count_new	24	24	19	26	17	7	24	14	17	17	19	24	17	22	26	27	24	26	24	27	17	17	17	27
medium_size_child_count_new	16	16	17	18	16	16	16	18	16	16	17	16	16	10	9	7	16	29	16	18	16	16	16	7



(b) Plan Items

(c) Stories

Fig. 3: Number of iteration changes for the requirements that do not make their first planned iteration.

3.4 Iteration Change Analysis

> *RQ4:* What happens to requirements that are not completed in their planned iteration?

To answer this research question, we investigate how often requirements are changed from their planned iterations and what iterations they are moved to. We identify a set of iteration change types and examine the frequency of each type. Finally, we investigate if there is a relationship between the type of iteration change and various properties of the work item.

3.4.1 Amount of Iteration Changes

We examined the amount of iteration changes for each requirement. Table 3 shows the ratio of requirements that are not completed in their planned iteration. Somewhere between 18% and 54% of requirements have at least one iteration change in each dataset.

We then examined those requirements which did not make their planned iteration to determine how many iteration changes they typically have. As can be seen in Figure 3, the majority of high-level requirements that are not completed in their first planned iteration have only one or two changes to their planned iteration. Only 7% of Plan Items and 19% of Stories have their planned iterations changed three or more times.

3.4.2 Types of Iteration Changes

To better understand the scheduling considerations of the high-level requirements that do not make their planned iterations, we analyzed the planned iteration changes. Over the three projects and two work item types, there were 6,530 iteration changes. As described in section 2.1, the iterations are

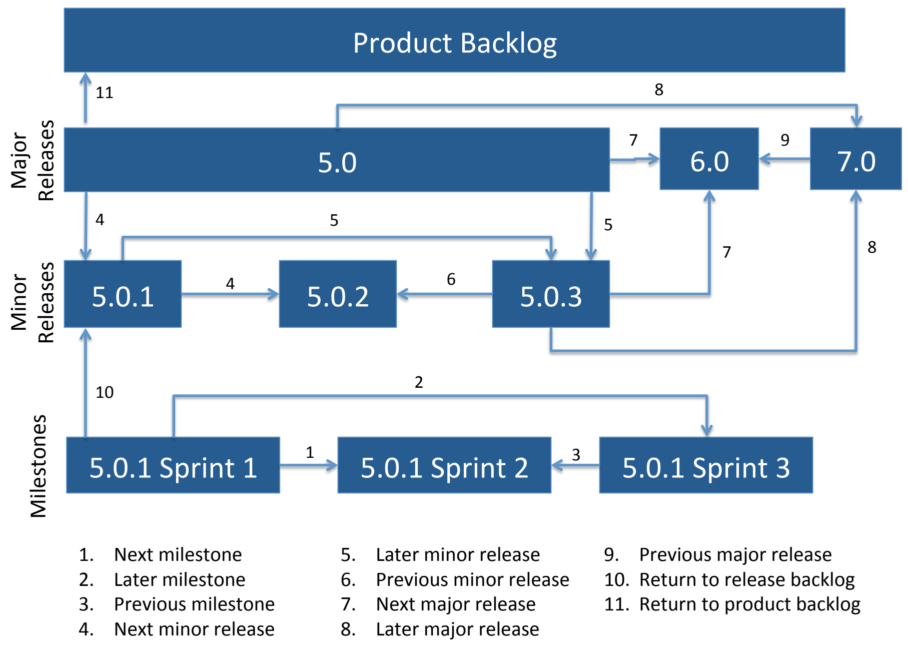


Fig. 4: Types of Iteration Changes.

hierarchical, so milestone iterations will have a parent release. Also, the IBM teams mostly used a standard naming convention for iterations (i.e. X.X.X is a minor release and X.X is a major release). Due to this, we were able to automatically identify the type of iteration changes that occur for 97% of the changes. We created a script, which performed pattern matching on the iteration name and considered the iteration hierarchy and start dates to identify the type of changes. We also manually reviewed the categorized changes to ensure the script was accurate in its categorization. Once the iteration changes were categorized, we examined the frequency of each type of change.

Some of the changes were part of the normal workflow for many requirements:

– Change from release to milestone within that release. This occurs when a requirement is changed from a given release to a specific development iteration within that release.
– Plan from backlog. This occurs when a requirement is taken from the main product backlog and planned for a specific release or milestone.

As these are expected in the normal workflow, these types of changes were not considered in our analysis. After excluding these types of iteration changes, we are left with 5,038 iteration changes, which we included in our analysis.

As depicted in Figure 4, we found the following types of iteration changes (their relative frequency is shown in Table 10):

1. Next milestone. Overall, the most common type of iteration change is pushing a requirement out from a milestone to the subsequent milestone of the release. For example, if a requirement was planned for release 5.0.1 sprint 1 and was not completed during that iteration, it would be rescheduled for 5.0.1 sprint 2.
2. Later milestone. Occasionally, requirements are pushed from a milestone to a later milestone within the same release that is not the subsequent milestone. For example, if a requirement changes from 5.0.1 Sprint 1 to 5.0.1 Sprint 3.
3. Previous milestone. Requirements can also be pulled forward to a sooner milestone than originally planned. For example, if a requirement was planned for release 5.0.1 sprint 2 and was changed to 5.0.1 sprint 1.
4. Next minor release. Another type of iteration change is pushing a requirement out from a release to the subsequent minor release. For example, if a requirement was planned for release 5.0.1 and was not completed during that release, it would be rescheduled for 5.0.2.
5. Later minor release. Requirements can be pushed from a release to a later minor release that is not the subsequent minor release. For example, a requirement could be replanned from 5.0.1 to 5.0.3.
6. Previous minor release. Requirements can also be moved from a minor release to the previous minor release. For example, if a requirement was scheduled for 5.0.3 and moved forward to 5.0.2.
7. Next major release. Another type of requirement iteration change is when a requirement is pushed out to the next major release. We found that happens most frequently when a requirement is planned for but not completed in the last minor release of the previous release. Occasionally, this also occurs from one of the earlier minor releases or from a milestone of one of the previous minor releases.
8. Later major release. Requirements can be pushed from a release to a later major release that is not the subsequent major release. For example, a requirement could be replanned from 4.0 to 6.0.
9. Previous major release. Requirements can also be moved from a major release to the previous major release. For example, if a requirement was scheduled for 5.0 and moved forward to 4.0.
10. Return to release backlog. If a requirement is not completed during its planned milestone, it is put back on the backlog for the release.
11. Return to product backlog. If a requirement is not completed during its planned milestone or release, it could also be put back on the overall product backlog.

As seen in Table 10, the frequency of iteration change types varies across the different work item types. Stories are most likely to be replanned for the next milestone, with this type of iteration change accounting for 59% of the changes. The types of changes for Plan items, which are the highest level of requirements, are more evenly distributed with next milestone, next minor

Table 10: Frequency of iteration change types.

Type	**Plan Items**	**Stories**	**Overall**
1. Next milestone	16%	59%	55%
2. Later milestone	5%	7%	6%
3. Previous milestone	3%	9%	9%
4. Next minor release	14%	6%	6%
5. Later minor release	2%	1%	1%
6. Previous minor release	7%	1%	1%
7. Next major release	11%	2%	3%
8. Later major release	6%	1%	2%
9. Previous major release	3%	1%	1%
10. Return to release backlog	14%	8%	9%
11. Return to product backlog	19%	5%	7%

release, next major release, return to release backlog, and return to product backlog occurring fairly evenly.

3.4.3 Effect of Work Item Properties on Type of Change

To understand why certain types of iteration changes are made, we used multinomial logistic regression to model the relationship between the type of iteration change identified and various work item properties.

The work item properties used in this analysis are a subset of the model features which are described in Section 3.1.1. We excluded model features in this analysis for the following reasons:

- Feature is too sparse. For example, DAYS_SINCE_LAST_DE_COMMENT is only available for those work items which had a DE comment, making this feature sparse. The sparse features were: DAYS_SINCE_LAST_DE_COMMENT, DAYS_SINCE_LAST_TYPE_STORY_PLAN, and DAYS_SINCE_LAST_TYPE_CHILD.
- Feature is categorical with a large number of categories. If all of the categorical features were cross-tabulated, it would result in a table too sparse for meaningful logistic modeling. Further, the categorical features with many categories would produce results that were very specific to the projects being studied. For example, we are not interested in identifying specific task assignees (OWNER_IDENTFIER) that have a relationship with certain types of iteration changes. The categorical features that were excluded were: OWNER_IDENTIFIER, CREATOR_IDENTIFIER, FILED_AGAINST, and CREATION_MONTH.
- Multicolliniarity. We check correlations between all pairs of features, and removed one of the features in the pair if the correlation was 0.5 or higher. This removed: COMPONENT_RESOLVER, DAYS_PAST_SINCE_CREATION, COMMENTER_COUNT, and SUBSCRIBER_COUNT.

We also included two additional features; project and work item type. These were not included in the feature set described in 3.1.1 since the previous analysis (RQ1-3) was done separately for each work item type and each project.

For this multinomial logistic regression, we computed each feature at the time of the iteration change. For example, the iteration change count would be the number of times the iteration changed prior to the change under investigation.

Table 11 shows the results of multinomial logistic regression where the type of iteration change was the output variable. We used the change type of Next milestone as the baseline. We chose this as the baseline since it is the overall most common type of iteration change. Table 11 reports the regression coefficients and the p-values (computed using Wald tests). In the table, we have bolded the regression coefficients that show the largest log odds and also have significant p-values.

We found work item type is one of the most significant predictors. Work item type is a two-level categorical variable (Plan Item or Story). The results in Table 11 show the regression coefficients for when the work item type is Plan Item (compared to Story). The log odds of being changed to the next minor or major release, a later major release or being put back on the release or product backlog (all bigger delays) compared to being changed to the next milestone increases significantly for plan items (compared to stories). On the other hand, the log odds of being changed to a previous milestone vs. being changed to the next milestone decreases significantly for plan items compared to stories. This confirms that the differences in frequency of iteration change types across work item types (see Table 10) is indeed significant.

Not surprisingly, we see an increase in PRIORITY is associated with a decrease in the log odds of the work item change of later major release. PRIORITY does not have any other significant effects.

Interestingly, we see that an increase in OWNER_CHANGE_COUNT is associated with a decrease in the log odds of the work item change type of previous major release. We also see that an increase in SAME_TYPE_CHILD_COUNT_NEW is associated with both a decrease in the log odds of a change to the later milestone release and an increase in the log odds of a change to the previous major or previous minor release. This is surprising as it would be expected that having additional child tasks or changes in task ownership would be more likely to cause delays. Further investigation is needed to better understand the reasons for these relationships.

Answer to RQ4: We identified eleven different types of iteration changes including: next milestone, next release, later release, previous release and return to backlog. The majority of requirements are simply planned for the next milestone when they are not completed in an iteration. The work item type is the most significant predictor on the type of iteration change that is made.

Table 11: Impact of various work item properties on iteration change type

	Prev. major	**Prev. minor**	**Prev. milestone**	**Later milestone**	**Release backlog**
Work item type (Plan Item)	-0.20 ***	0.26 ***	**-1.97** ***	-0.38 ***	**2.64** ***
Project	0.63	-0.73	-0.53 *	-0.38	0.04
iteration_change_count	-0.44 **	-0.13	0.09	-0.07	-0.14
iteration_days_remained	<0.01 ***	<0.01 ***	<0.01 ***	<0.01 ***	<0.01 ***
Status	-0.29 ***	0.12 ***	-0.02 ***	-0.06 ***	-0.32 ***
Priority	-0.35 ***	-0.46 ***	-0.22 ***	0.04 ***	-0.66 ***
Severity	-0.12	0.34	-0.20	0.17	-0.10
Days_without_owner	<-0.01	<-0.01	<-0.01	-0.01	<-0.01
Days_since_last_comment	0.01 ***	0.01 ***	0.01 ***	<0.01 ***	0.01 ***
Owner_change_count	**-2.51** **	-0.48	-0.36	-0.64 *	<0.01
Days_since_last_owner	0.02 ***	0.01 ***	0.01 ***	0.01 ***	0.01 ***
Summary_change_count	0.01	0.02	-0.34 *	0.01	-0.04
Description_change_count	0.04	<-0.01	0.08 **	-0.01	-0.12
Days_since_last_summary	-0.09	-0.13	<-0.01	<-0.01	<0.01
Days_since_last_description	0.01 **	0.01	<0.01	<0.01	<-0.01
Creator_Team_Relationship	0.06	-0.96	-0.16	0.01	-0.27
Comment_count	0.01	0.01	-0.05 *	<-0.01	-0.01
Same_type_child_count_new	**1.42** ***	-0.16 ***	**1.01** ***	**-1.61** ***	0.17 ***
large_size_child_count_new	0.27	-0.14	-0.94	-0.12	-0.17
medium_size_child_count_new	-0.18	-0.73	-0.11	-0.18	-0.01
	Next minor	**Later minor**	**Next major**	**Later major**	**Product backlog**
Work item type (Plan Item)	**2.63** ***	-0.41 ***	**3.40** ***	**3.02** ***	**2.13** ***
Project	0.30	0.15	0.48	-0.57	-0.73 **
iteration_change_count	-0.11	0.17	-0.10	-0.37 *	-0.09
iteration_days_remained	<0.01 ***	<0.01 ***	<0.01 ***	<0.01 ***	<0.01 ***
Status	-0.48 ***	-0.24 ***	-0.64 ***	-0.33 ***	-0.55 ***
Priority	-0.29 ***	-0.57 ***	-0.30 ***	**-1.45** ***	-0.82 ***
Severity	0.13	-0.85	-0.92	0.51	0.06
Days_without_owner	<-0.01	<-0.01	<0.01	<-0.01	<0.01
Days_since_last_comment	0.01 ***	0.01 ***	0.01 ***	0.01 ***	0.01 ***
Owner_change_count	-0.25	**-1.65** *	-0.14	-1.15	-0.38
Days_since_last_owner	0.01 ***	0.01 ***	0.01 ***	0.01 ***	0.01 ***
Summary_change_count	0.12	-0.03	0.07	0.22	0.06
Description_change_count	0.02	-0.07	0.04	-0.08	0.03
Days_since_last_summary	-0.01 **	0.01	<-0.01	<-0.01	<0.01
Days_since_last_description	<0.01 *	-0.01	<0.01	<-0.01	<0.01
Creator_Team_Relationship	-0.22	0.51	0.16	0.90	0.28
Comment_count	0.03 **	-0.01	0.05 **	0.01	-0.02
Same_type_child_count_new	-0.51 ***	-0.35 ***	0.54 ***	0.45 ***	-0.29 ***
large_size_child_count_new	-0.48	-0.03	-0.15	0.12	-0.39
medium_size_child_count_new	0.03	0.03	-0.15	-0.16	-0.05

(* p <0.05, ** p <0.01, *** p <0.001)

4 Related Work

Although no other studies propose methods to predict whether a high-level requirement will be completed in a planned iteration, there are related areas of research. Below, we present related work covering prediction on various aspects of software tasks and releases.

4.1 Software Task Predictions

Prior studies on low-level software tasks (i.e. bug fixes and issue resolutions) have focused on predicting resolution time [42,21,1,36,15,30], effort involved [58,5,45,12], and probability of completion [23]. However, none of them have investigated completion time of high-level requirements. While high-level requirements often have smaller child tasks, they cannot be estimated by simply combining the estimates of their children due to parallel work, dependencies, coordination overhead, etc. Our prediction model draws on many of the techniques and features used in these previous approaches, tailored to work with high-level requirements.

Past studies employ various machine learning techniques to make their predictions. Unsupervised learning techniques [61,5,45], kNN [58], linear [42], probabilistic [42,1] and decision tree [42,21] classifiers as well as random forest [36,30] and other ensemble learning techniques [12] are among the popular ones. In our study, we employ random forest due to its ability to handle a mixture of continuous, semi-continuous and categorical features and its robustness against noise, missing values and correlated features [60].

Various features are used to make predictions on software tasks. A number of studies utilize task meta attributes, such as task creator, owner, priority, severity, etc. to make predictions [42,21,1,23]. Some also apply text analysis techniques on text attributes [58,48,5,45,12] and some adopt social network analysis techniques [15]. Yet, not all of the studies have come to the same conclusions in regards to which features are best to include. Guo et al. found that bug reporter's reputation has a significant impact on bug resolution time [23]. However, Bhattacharya and Neamtiu [8] found no such correlation and stated that importance of features are highly project dependent. Inspired by these past studies, we identify 29 features for our models (described in detail in Section 3.1.2).

Marks et al. [36] and Kikas et al. [30] both classified their engineered features into several groups based on the source of features and measured relative importance of features to their models. Likewise, we rank the importance of the engineered features to our models, and also classify our features into nine categories based on their potential impact on completion time.

Making early predictions can be useful, but predictions likely become more accurate as more data becomes available. Some of the studies made predictions at different stages. For example, Giger et al. [21] made predictions at six different stages of a bug lifetime. Kikas et al. [30] made predictions at different stages of an issue lifetime. Similarly, we make predictions at various stages of a requirement lifecycle.

4.2 Software Release Readiness

Prior studies have also investigated the overall readiness of a software release. A software release is a collection of new and/or changed features that form a new product [52]. The decisions around when to release a new product and what features should be included in each release are complex [20]. Previous research has proposed various checklists that product managers can use to assess the readiness of a release [37,6,50,39,51]. The checklists contain criteria such as checking for open bugs, ensuring adequate test coverage, verifying personnel are available on the day of the release, and ensuring updated documentation is available for users [37]. These checklists can be helpful in assessing how ready a software project is for release, but the criteria often involve subjective decisions.

In addition to these checklists, a variety of quantitative measures have been proposed to assist managers in release readiness decisions. For example, metrics to predict the number of software defects remaining in a software release have been proposed [10,43,38]. This can be estimated by considering the current rate of defect discovery [10,43], comparing the defect density of the current release to previous releases [38], or purposely inserting defects into the code to see how many the testers find (called defect seeding) [38]. Using such metrics, managers can estimate how many defects are still left in the release, which can help managers know if the release is ready. Similarly, it has been proposed to consider code maturity and stability to assist with release readiness decisions by measuring the amount of code changes [43] or changes in code complexity and coupling over time [57].

In a systematic literature review, Alam et al. [3] found that the majority of these measures are related to product quality and testing metrics. However, release readiness decisions are multi-dimensional. A survey of software practitioners found that managers use four main measures in their decision to ship a release: feature completion rate, bug fix rate, defect find rate and build success [3]. Some multi-dimensional metrics have been proposed. Asthan and Olivieri [6] proposed a set of metrics along five dimensions: software functionality, operational quality, known remaining defects, testing scope and stability, and reliability. Satapathy [53] proposed an aggregate measure, called ShipIt, which measures release readiness by considering how much progress has been made in all stages of the software lifecycle compared to the release plan. However, both of these aggregate large amounts of information into single measures, which can be difficult to understand.

To assist practitioners, methods have been proposed that use predictive analysis and various machine learning techniques to predict release readiness. Early models focused on the single dimension of defects. Quah [47] developed a model that predicts the number of software defects in a release along with the amount of time it will take to make code changes to fix those defects using neural networks. Wild and Brune [59] also developed a model to predict the number of defects that will be found in testing, but they used linear response theory. More recently, multi-dimensional predictive models have been proposed which consider various features, such as release duration and number of open requirements and defects [2,4]. These models formulate software release readiness as a binary classification problem, so it is easy to interpret the result.

Our work differs from these studies as we focus on the completion of individual high-level requirements and do not try to predict readiness of a full release.

5 Discussion

In this study, we investigated requirement iteration changes. We identified eleven types of iteration changes that occur. Requirements can be moved to another milestone, minor release, major release or returned to the release or product backlog. For milestone, minor release and major release changes, requirements can be moved forward to a previous iteration or pushed out to the subsequent or a later iteration. We found that Stories are most often just moved to the next milestone. We also found that 19% of Stories have their planned iterations changed three or more times. In contrast, Plan Items are more or less equally likely to be moved to the next milestone, the next minor release, the next major release, returned to the release backlog, or returned to the product backlog. Only 7% of Plan Items have their planned iterations changed three or more times. This indicates that the team is making more realistic replanning decisions for Plan Items. Only very few Stories are moved to later (not subsequent) milestones or releases. Having better support on when a requirement will be completed would be useful for managers to make better decisions on replanning Stories.

As a first step in providing such support, we developed and evaluated a predictive model that predicts the likelihood of requirement implementation within the planned iteration for high-level requirements, filling a gap in software release planning research. Existing research has proposed models for bug fix effort estimation and release readiness prediction. However, no existing research investigates completion of high-level requirements. Our predictive model uses machine learning, includes a set of 29 features, and makes predictions at four stages of a requirement lifetime. This model was validated on three projects with satisfying results. IBM is investigating incorporating this method into their task management software to enable their software teams to make better decisions regarding task scheduling and resource allocation. The analysis has also made them aware of how important it is to ensure all data related to tasks is kept up-to-date and complete, since such data is not only useful to the tasks themselves, but can be used for analysis such as the study described in this paper. Due to this, they are now considering what other task meta-data can be captured and stored to enable future analysis.

We have also ranked the features according to their relative importance to the trained model. Although we observed that the importance of some features is dependent on prediction stage and project even within the same company, we showed that there are some features that are always consistently ranked higher than others across the different projects and stages of development. The implication of these results are a set of candidate features to receive further attention by researchers and practitioners. Many of the features were inspired by previous work, but we have shown their utility in predictions for high-level requirements. We also proposed many new features, some of which are specific to high-level requirements, like the three child features described in Section 3.1.2.

Another implication of this study is related to the precision-recall trade-off. In general, we would like to have predictive models with both high precision and recall, and there is a general belief that models with low precision or recall are practically useless [62]. However, as Menzies et al. [40] argue, there are certain practical use cases for models with low precision and high recall, such as in the defect prediction domain. In our study, however, our claim for the usefulness of models with high precision and relatively low recall is based on our industrial partner's interest for high precision at the expense of recall. We also showed that machine learning techniques, in particular cost sensitive learning, could be adopted to satisfy such requirements.

5.1 Threats to Validity

Like any empirical study, our paper is prone to some threats to validity [16]. We describe them below, together with our mitigation strategies. The first threat, to external validity, is the potential lack of generalizability of our findings due to the close connection with the data and respondents from only one software organization. We mitigated this threat by studying three different projects and by adopting techniques and features that have been empirically validated in previous work. Although all studied projects are within the same company, we believe many of the proposed features are general enough to be applicable to projects from other companies. Besides the features themselves, we grouped the 29 features in nine categories which, due to a higher abstraction level, represent a starting point for other organizations to engineer their own features. This study should be replicated in other organizations to validate generalizability.

The second threat, to construct validity, concerns our potential misinterpretation of the workflows within the IBM ecosystem. This was mitigated by prolonged close connection with IBM practitioners and through iterative and feedback-driven design of our model and machine learning techniques.

The third threat, to internal validity, relates to whether the results really do follow from the data in our study. Specifically, whether the iteration change types are valid, whether the features are meaningful to our prediction outcome, and whether the performance estimations are realistic. The eleven iteration change types were identified by only one person familiar with software development practices. However, the types are all logical and align with standard software development practices. To count the frequency of occurrence for each iteration change type, we were able to use pattern matching and validated the results through manual analysis by someone familiar with software development practices. For the model development, we performed manual feature engineering through iterative cycles of conducting interviews and using the domain knowledge of the data and avoided automatic means in feature selection to avoid bias. Moreover, we performed LOOCV as recommended in effort estimation literature [32] to reduce bias and variance of our performance estimations.

The fourth threat to validity concerns the reliability and reproducibility of our findings. We have done our best to provide a clear methodology and specify design decisions, so that as long as other researchers study the same data following the same study settings, the same results should be achieved.

5.2 Future Work

There are several avenues for future research directions. We summarize some potential research direction heres.

Additional features. Firstly, the proposed model could be further improved by including new features. Some possible new feature ideas include:

- *Length of text attributes.* Longer descriptions and summaries may indicate a deeper understanding of the problem. Conversely, long comments may indicate misunderstanding or differences of opinions.
- *Severity of changes in summary and description.* We considered only the number of times the summary or description were changed. However, some of these changes could be trivial. More severe changes could indicate more problematic requirements.
- *Additional child features.* While we considered the count of same type, large and medium size children, it may be beneficial to include additional child features. For example, the number of all children for each requirement type may be useful.
- *Additional communication and coordination related features.* For example, geographical distribution of project stakeholders was found useful in [23].
- *Sentiment analysis on comments.* Extremely negative comments could indicate problematic requirements.

Future work can include these proposed features to validate these assumptions and ascertain if better accuracy is obtained with their inclusion.

Ensemble techniques. Secondly, using ensemble techniques could allow even more additional features to be included. Previous work on bug effort estimation had success using text similarity techniques to make predictions based on similar bugs [58,45]. Such techniques could be tested to identify if they can also be applied on high-level requirements. Although most previous models using text analysis solely relied on text attributes, there has been prior work that integrated them with other methods successfully using ensemble techniques [12] or by transforming text attributes into numeric features [64,30]. Ensemble learning techniques have also been used successfully in effort estimation literature [33,7,27]. Utilizing such methods, would allow the inclusion of different sources of data. Combining the estimates from various sources is likely to produce better estimates compared to each individual source [29].

Feature importance. In this study, we considered feature importance of individual features only. However, future work could consider relationships between features and rank pairs or groups of features.

Effort estimation. Finally, another research direction would be to predict completion time of high-level requirements. As mentioned previously, there is a large amount of research that has done such effort estimation for low-level tasks such as bug fixes. Models have been developed to predict both completion time [42,21,1,36,15,30] and completion effort [58,5,45,12] of software tasks. High-level requirements typically have a collection of child tasks as the work is broken into smaller, more manageable tasks. Combining the predictions of all child tasks would not produce accurate results since child tasks are often worked on in parallel and have dependencies, adding additional coordination overhead and delays.

Thus, since this is the first study to investigate completion of high-level requirements, we focused on the binary classification problem of whether a high-level requirement can be completed within its planned iteration. Future research should build on these models to predict completion time of high-level requirements.

Cross-project predictions. While we found that the important features varied quite a bit between projects, there were some features that were universally important across all projects. Future work could investigate if it is feasible to make predictions across project teams. If a small set of universally important features were identified, being able to make predictions across projects would allow predictions to be made on newer projects without the need to large amounts of historical data. Future work can investigate methods to produce accurate cross-project predictions.

Multi-objective optimization. In our case, IBM had specific requirements for the weight of precision compared to recall, allowing us to utilize a weighted arithmetic mean to ensure the model performed according to the needs to IBM. However, these weights may not be applicable to other projects outside of this study. Future work could investigate ways to incorporate multi-objective optimization, such as Differential evolution (DE) [46], Non-dominated Sorting Genetic Algorithm II (NSGA-II) [11], or the Multiobjective Evolutionary Algorithm based on Decomposition (MOEA/D) [63], to ensure the balance between precision and recall is optimized.

6 Conclusion

In this work, we studied three large IBM projects using a combination of qualitative and quantitative methods. We investigated iteration changes of high-level software requirements. We found that 7% to 19% of requirements have their planned iterations changed three or more times. We identified eleven different types of iteration changes. The most common change types are moving to the next milestone, next minor release, next major release or returning to the release or product backlog. Better support for predicting when a high-level requirement will be completed can help project managers make better planning decisions.

We also proposed a predictive model to predict whether a high-level requirement will be completed in its planned iteration or not. Our model made machine learning based predictions at four meaningful stages of a requirement lifetime. We engineered a set of 29 features and put them in nine logical categories. Our model obtained precision scores of up to 0.86 and F1-scores of up to 0.80. We then modified the learning process to maximize precision of predictions according to the business interest of our industrial partner, IBM, and achieved precision values between 0.82 and 1 while retaining a recall value between 0.08 and 0.66 depending on the project and prediction stage. We ranked features based on their relative importance to the trained model and observed that although importance of features is highly project and prediction stage dependent, there are certain features, such as time remained to the end of iteration, creator of the requirement, and owner of the requirement that have high importance in all projects and prediction stages.

Acknowledgements Special thanks to Fabio Calefato from University of Bari, Alan Yeung from Persistent Systems and the IBM team.

References

1. Abdelmoez, W., Kholief, M., Elsalmy, F.M.: Bug fix-time prediction model using naïve bayes classifier. In: Computer Theory and Applications (ICCTA), 2012 22nd International Conference on, pp. 167–172. IEEE (2012)
2. Al Alam, S.D., Karim, M.R., Pfahl, D., Ruhe, G.: Comparative analysis of predictive techniques for release readiness classification. In: Realizing Artificial Intelligence Synergies in Software Engineering (RAISE), 2016 IEEE/ACM 5th International Workshop on, pp. 15–21. IEEE (2016)
3. Alam, A., Didar, S., Nayebi, M., Pfahl, D., Ruhe, G.: A two-staged survey on release readiness. In: Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering, pp. 374–383. ACM (2017)
4. Alam, A., Didar, S., Pfahl, D., Ruhe, G.: Release readiness classification: An explorative case study. In: Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, p. 27. ACM (2016)
5. Assar, S., Borg, M., Pfahl, D.: Using text clustering to predict defect resolution time: a conceptual replication and an evaluation of prediction accuracy. Empirical Software Engineering **21**(4), 1437–1475 (2016)
6. Asthana, A., Olivieri, J.: Quantifying software reliability and readiness. In: Communications Quality and Reliability, 2009. CQR 2009. IEEE International Workshop Technical Committee on, pp. 1–6. IEEE (2009)

7. Azhar, D., Riddle, P., Mendes, E., Mittas, N., Angelis, L.: Using ensembles for web effort estimation. In: Empirical Software Engineering and Measurement, 2013 ACM/IEEE International Symposium on, pp. 173–182. IEEE (2013)
8. Bhattacharya, P., Neamtiu, I.: Bug-fix time prediction models: can we do better? In: Proceedings of the 8th Working Conference on Mining Software Repositories, pp. 207–210. ACM (2011)
9. Breiman, L.: Random forests. Machine learning **45**(1), 5–32 (2001)
10. Brettschneider, R.: Is your software ready for release? IEEE Software **6**(4), 100 (1989)
11. Deb, K., Pratap, A., Agarwal, S., Meyarivan, T.: A fast and elitist multiobjective genetic algorithm: Nsga-ii. IEEE transactions on evolutionary computation **6**(2), 182–197 (2002)
12. Dehghan, A., Blincoe, K., Damian, D.: A hybrid model for task completion effort estimation. In: Proceedings of the 2nd International Workshop on Software Analytics, pp. 22–28. ACM (2016)
13. Dehghan, A., Neal, A., Blincoe, K., Linaker, J., Damian, D.: Predicting likelihood of requirement implementation within the planned iteration: an empirical study at ibm. In: Proceedings of the 14th International Conference on Mining Software Repositories, pp. 124–134. IEEE Press (2017)
14. Domingos, P.: A few useful things to know about machine learning. Communications of the ACM **55**(10), 78–87 (2012)
15. Duc, A.N., Cruzes, D.S., Ayala, C., Conradi, R.: Impact of stakeholder type and collaboration on issue resolution time in oss projects. In: IFIP International Conference on Open Source Systems, pp. 1–16. Springer (2011)
16. Easterbrook, S., Singer, J., Storey, M.A., Damian, D.: Selecting empirical methods for software engineering research. In: Guide to advanced empirical software eng., pp. 285–311. Springer (2008)
17. Efron, B.: Estimating the error rate of a prediction rule: improvement on cross-validation. Journal of the American Statistical Association **78**(382), 316–331 (1983)
18. Fernández-Delgado, M., Cernadas, E., Barro, S., Amorim, D.: Do we need hundreds of classifiers to solve real world classification problems. J. Mach. Learn. Res **15**(1), 3133–3181 (2014)
19. Fortmann-Roe, S.: Understanding the bias-variance tradeoff (2012)
20. Franch, X., Ruhe, G.: Software release planning. In: Proceedings of the 38th International Conference on Software Engineering Companion, pp. 894–895. ACM (2016)
21. Giger, E., Pinzger, M., Gall, H.: Predicting the fix time of bugs. In: Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering, pp. 52–56. ACM (2010)
22. Gueorguiev, S., Harman, M., Antoniol, G.: Software project planning for robustness and completion time in the presence of uncertainty using multi objective search based software engineering. In: Proceedings of the 11th Annual conference on Genetic and evolutionary computation, pp. 1673–1680. ACM (2009)
23. Guo, P.J., Zimmermann, T., Nagappan, N., Murphy, B.: Characterizing and predicting which bugs get fixed: an empirical study of microsoft windows. In: Software Engineering, 2010 ACM/IEEE 32nd International Conference on, vol. 1, pp. 495–504. IEEE (2010)
24. Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I.H.: The weka data mining software: an update. ACM SIGKDD explorations newsletter **11**(1), 10–18 (2009)
25. He, H., Garcia, E.A.: Learning from imbalanced data. IEEE Transactions on knowledge and data engineering **21**(9), 1263–1284 (2009)
26. Heikkilä, V.T., Damian, D., Lassenius, C., Paasivaara, M.: A mapping study on requirements engineering in agile software development. In: 41st Euromicro Conference on Software Engineering and Advanced Applications (SEAA), pp. 199–207. IEEE (2015)
27. Idri, A., Hosni, M., Abran, A.: Systematic literature review of ensemble effort estimation. Journal of Systems and Software **118**, 151–175 (2016)
28. Jeong, G., Kim, S., Zimmermann, T.: Improving bug triage with bug tossing graphs. In: Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, pp. 111–120. ACM (2009)

29. Jorgensen, M.: What we do and don't know about software development effort estimation. IEEE software **31**(2), 37–40 (2014)
30. Kikas, R., Dumas, M., Pfahl, D.: Using dynamic and contextual features to predict issue lifetime in github projects. In: Proceedings of the 13th International Conference on Mining Software Repositories, pp. 291–302. ACM (2016)
31. Klinkenberg, R.: RapidMiner: Data mining use cases and business analytics applications. Chapman and Hall/CRC (2013)
32. Kocaguneli, E., Menzies, T.: Software effort models should be assessed via leave-one-out validation. Journal of Systems and Software **86**(7), 1879–1890 (2013)
33. Kocaguneli, E., Menzies, T., Keung, J.W.: On the value of ensemble effort estimation. IEEE Transactions on Software Engineering **38**(6), 1403–1416 (2012)
34. Lindstrom, L., Jeffries, R.: Extreme programming and agile software development methodologies. Information systems management **21**(3), 41–52 (2004)
35. Liu, H., Setiono, R.: Chi2: Feature selection and discretization of numeric attributes. In: Tools with artificial intelligence, 1995. proceedings., seventh international conference on, pp. 388–391. IEEE (1995)
36. Marks, L., Zou, Y., Hassan, A.E.: Studying the fix-time for bugs in large open source projects. In: Proc. of the 7th International Conf. on Predictive Models in Software Engineering, p. 11. ACM (2011)
37. McBride, M.: Is your team ready to release? In: Managing Projects in the Real World, pp. 171–182. Springer (2014)
38. McConnell, S.: Gauging software readiness with defect tracking. IEEE Software **14**(3), 136 (1997)
39. McConnell, S.: Software project survival guide. Pearson Education (1998)
40. Menzies, T., Dekhtyar, A., Distefano, J., Greenwald, J.: Problems with precision: A response to" comments on'data mining static code attributes to learn defect predictors'". IEEE Transactions on Software Engineering **33**(9) (2007)
41. Minku, L.L., Mendes, E., Turhan, B.: Data mining for software engineering and humans in the loop. Progress in Artificial Intelligence **5**(4), 307–314 (2016)
42. Panjer, L.D.: Predicting eclipse bug lifetimes. In: Proceedings of the Fourth International Workshop on mining software repositories, p. 29. IEEE Computer Society (2007)
43. Pearse, T., Freeman, T., Oman, P.: Using metrics to manage the end-game of a software project. In: Software Metrics Symposium, 1999. Proceedings. Sixth International, pp. 207–215. IEEE (1999)
44. Petersen, K., Wohlin, C.: Context in industrial software engineering research. In: Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement, pp. 401–404. IEEE Computer Society (2009)
45. Pfahl, D., Karus, S., Stavnycha, M.: Improving expert prediction of issue resolution time. In: Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering, p. 42. ACM (2016)
46. Price, K., Storn, R.M., Lampinen, J.A.: Differential evolution: a practical approach to global optimization. Springer Science & Business Media (2006)
47. Quah, T.S.: Estimating software readiness using predictive models. Information Sciences **179**(4), 430–445 (2009)
48. Raja, U.: All complaints are not created equal: text analysis of open source software defect reports. Empirical Software Engineering **18**(1), 117–138 (2013)
49. Ramarao, P., Muthukumaran, K., Dash, S., Murthy, N.B.: Impact of bug reporter's reputation on bug-fix times. In: Information Systems Engineering (ICISE), 2016 International Conference on, pp. 57–61. IEEE (2016)
50. Rothman, J.: Release criteria: Is this software done? STQE magazine (2002)
51. Rothman, J.: Measurements to reduce risk in product ship decisions (2014)
52. RUHE, G.: Software release planning. In: Handbook Of Software Engineering And Knowledge Engineering: Vol 3: Recent Advances, pp. 365–393. World Scientific (2005)
53. Satapathy, P.R.: Evaluation of software release readiness metric [0, 1] across the software development life cycle. Department of Computer Science & Engineering, University of California, Riverside (2013)
54. Selya, A.S., Rose, J.S., Dierker, L.C., Hedeker, D., Mermelstein, R.J.: A practical guide to calculating cohens f2, a measure of local effect size, from proc mixed. Frontiers in psychology **3**, 111 (2012)

55. Seymour, J.: Software delays: Truth or consequences. PC Magazine **7**(12) (1988)
56. Ting, K.M.: An instance-weighting method to induce cost-sensitive trees. IEEE Transactions on Knowledge and Data Engineering **14**(3), 659–665 (2002)
57. Ware, M., Wilkie, F.G., Shapcott, M.: The use of intra-release product measures in predicting release readiness. In: Software Testing, Verification, and Validation, 2008 1st International Conference on, pp. 230–237. IEEE (2008)
58. Weiss, C., Premraj, R., Zimmermann, T., Zeller, A.: How long will it take to fix this bug? In: Proceedings of the Fourth International Workshop on Mining Software Repositories, p. 1. IEEE Computer Society (2007)
59. Wild, R., Brune, P.: Determining software product release readiness by the change-error correlation function: on the importance of the change-error time lag. In: System Science (HICSS), 2012 45th Hawaii International Conference on, pp. 5360–5367. IEEE (2012)
60. Yang, F., Wang, H.z., Mi, H., Cai, W.w., et al.: Using random forest for reliable classification and cost-sensitive learning for medical diagnosis. BMC bioinformatics **10**(1), S22 (2009)
61. Zeng, H., Rine, D.: Estimation of software defects fix effort using neural networks. In: Computer Software and Applications Conference, 2004. COMPSAC 2004. Proceedings of the 28th Annual International, vol. 2, pp. 20–21. IEEE (2004)
62. Zhang, H., Zhang, X.: Comments on" data mining static code attributes to learn defect predictors". IEEE Transactions on Software Engineering **33**(9) (2007)
63. Zhang, Q., Li, H.: Moea/d: A multiobjective evolutionary algorithm based on decomposition. IEEE Transactions on evolutionary computation **11**(6), 712–731 (2007)
64. Zhou, Y., Tong, Y., Gu, R., Gall, H.: Combining text mining and data mining for bug report classification. Journal of Software: Evolution and Process (2016)