# Optimizing Modern Code Review Through Recommendation Algorithms

by

Giovanni Viviani

B. in Informatics, Università della Svizzera Italiana, 2014

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

**Master of Science**

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES
(Computer Science)

The University of British Columbia
(Vancouver)

August 2016

# Abstract

Software developers have many tools at their disposal that use a variety of sophisticated technology, such as static analysis and model checking, to help find defects before software is released. Despite the availability of such tools, software development still relies largely on human inspection of code to find defects. Many software development projects use code reviews as a means to ensure this human inspection occurs before a commit is merged into the system. Known as modern code review, this approach is based on tools, such as Gerrit, that help developers track commits for which review is needed and that help perform reviews asynchronously. As part of this approach, developers are often presented with a list of open code reviews requiring attention. Existing code review tools simply order this list of open reviews based on the last update time of the review; it is left to a developer to find a suitable review on which to work from a long list of reviews. In this thesis, we present an investigation of four algorithms that recommend an ordering of the list of open reviews based on properties of the reviews. We use a simulation study over a dataset of six projects from the Eclipse Foundation to show that an algorithm based on ordering reviews from least lines of code modified in the changes to be reviewed to most lines of code modified out performs other algorithms. This algorithm shows promise for eliminating stagnation of reviews and optimizing the average duration reviews are open.

# Preface

No part of this thesis has been published. This thesis is an original intellectual product of the author, G. Viviani.

# Table of Contents

# List of Tables

# List of Figures

# Acknowledgments

I would like to start thanking my parents and my sister for believing in me and supporting me from Europe over the past two years. They have been encouraging me to follow my dreams since I took the decision of studying Computer Science five years ago. Pursuing my studies on the other side of the planet has been a great source of stress for all of us, but I am glad to have their support.

Special thanks to my supervisor, Gail C. Murphy, for being the best mentor I could have asked for. Gail has been there to guide me whenever I felt lost and she has done an amazing job in teaching me what research is. She has also shown an incredible patience in dealing with my stubbornness and my habit of signing up for too many extra-curricular activities.

I would also thank my second reader, Reid Holmes, for taking the time to read this thesis and providing valuable feedback. Similarly, I want to acknowledge the whole Software Practices Lab for the support and for providing a great environment in which to work.

Finally, I want to thank all my friends, both in Canada and Switzerland, for keeping me sane during the past two years. Special mentions go to Jessica Wong and Ambra Garcea for enduring me during ups and downs without too many complaints.

*To my family, for providing all the support I have ever needed.*

# Chapter 1

# Introduction

For over forty years, software developers have used humans looking at the code as a means of reducing defects in the code. In the 1970s and 1980s, developers used a highly structured process, known as code inspection, in which groups of people met in person to review code line-by-line [6]. In recent years, advancements in tools and techniques have allowed this process to evolve to a more lightweight approach, defined by Baccheeli and Bird as modern code review [1].

The lightweight approach has been largely made possible by the invention of collaborative tools, such as Gerrit[1], that enable human reviewers to work asynchronously and remotely from each other. These tools enable code reviews to be performed for most, if not all, commits made to a source code repository both for open and closed source projects. Compared to the older style code inspections, most code reviews have a smaller size and involve less people [13].

In projects using a modern code review approach, every commit submitted is required to go through a review process that will decide if the change satisfies the project standards. Developers on a project are typically expected to contribute to open code reviews. On large projects, the number of open code reviews can become quite large, leading to a slow down in the release of changes. As an example, an analysis of six projects from the Eclipse ecosystem shows that there can be hundreds of code review open at a time; the `JGit` project currently has more than

[1] https://gerrit.googlecode.com, verified 20/7/16

150 open code reviews[2].

Despite developers spending time daily working on code reviews [12], code reviews go stagnant and remain stagnant for long periods of time. My analysis of the code review process for the Eclipse ecosystem showed that some code reviews can end up being left open for entire months without any update: in the `JGit` project, some code reviews are open for two years before being merged into the main branch.

In this thesis, I hypothesize that the stagnation of reviews is due, in part, to a lack of support by the tools, such as Gerrit, to help developers choose which code reviews to work on. Gerrit, and other similar tools, provide many features to aid code reviews. For example, the tools provide easy access to a list of files changed as part of a commit and user interfaces to visualize the difference of files in the commit to the current state of the file. However, these tools offer nearly no support for choosing the code review to analyze, but they rather display the reviews ordered by update timestamps.

I investigate whether an algorithm that suggests developers the order in which to work on code reviews can help avoid stagnation of reviews and improve the overall review process. I propose four algorithms for ordering open code reviews. I report on the effectiveness of each algorithm by using a simulation to show the effect that the algorithm would have in the order in which code review are solved and the average duration of resolution if the reviews were worked on in the suggested order. I found that an algorithm based on changes between the commit and existing code, when ordered from least changes (lines of code) to most changes, results in a lower average duration of open reviews and less stagnation.

This thesis makes the following contributions.

- Provides evidence to the problem of code review stagnation, identified by Rigby and Bird [13]
- It introduces four algorithms to order open code reviews to avoid stagnation and to reduce overall code review resolution time.
- It shows, through a simulation study, the effectiveness of each algorithm,

[2] Verified 2/7/2016

finding that an algorithm based on a syntactic analysis of the change reported in lines of code is the most likely approach to solve the code review stagnation problem.

I being by reviewing background and related work (Chapter 2). I then present my approach of recommending a code review ordering (Chapter 3). Chapter 4 presents the evaluation I run for validating my approach, followed by the results in Chapter 5. In Chapter 6, I address the threats to my results and I discuss possible future developments in Chapter 7. I finally summarize the work and draw some conclusion in Chapter 8.

# Chapter 2

# Background and Related Work

In this chapter, I describe the origin of code reviews, starting from the code inspection process up to the concept of lightweight code review. I also describe code review tools, with particular attention to Gerrit, the tool around which this research is built. I also cover research closely related to this thesis: time taken to complete code reviews and tools that use recommendation to improve aspect of the code review process. Finally, I provide some information on code review systems.

## 2.1 Code Inspection

The idea of code review can be attributed to Michael Fagan. Fagan defined an inspection process for code with two goals[5]:

- find and fix all product defects, and
- find and fix all development process defects that lead to product defects.

Figure 2.1 outlines the Fagan Inspection process. The Fagan Inspection is composed of 6 steps:

1. **Planning**: Materials meet the entry criteria; arrange the availability of the participants; arrange the place and time of the meeting.
2. **Overview**: Educate the group on the materials; assign the roles to the participants.

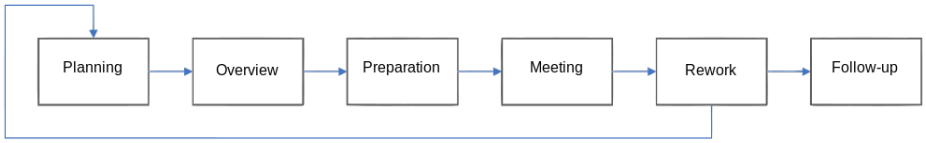



**Figure 2.1:** Basic model for Fagan Inspection

3. **Preparation**: Prepare for the review by learning the materials.
4. **Meeting**: Find the defects.
5. **Rework**: Rework the defects.
6. **Follow-up**: Assure that all defects have been fixed; assure that no other defect has been introduced.

In a follow-up paper, Fagan describes the advancement that has been made on the original concept of code inspection, identifying three main aspects Defect Detection, Defect Prevention and Process Management, and actions that can be taken to improve them[7].

Since the introduction of code inspection, researcher have studied how and why this process works and how to improve it. For further information, Kollanus and Koskinen provide a survey covering the research on code inspection, between 1980 and 2008[11].

### 2.1.1 Lightweight Code Review

In more recent years, the code inspection process has become more lightweight compared to the code inspection created by Fagan. This lightweight process has been named Modern Code Review by Bacchelli and Bird [1], to differentiate between the old process of inspection described by Fagan from the process that has been recently evolving. This process is being used by many companies, and it is characterized by being informal and tool-based.

Through its basis in tools, modern code review has the advantage of being asynchronous, whereas previous software inspection processes were synchronous. This

allows teams spread around the globe to continue their work without the problem of scheduling meetings.

## 2.2 Code Review Tools

Many code review tools have been developed to help developers with the code review process. Pre-commit and post-commit are the two different approaches that can be taken for code reviews. The major classification of the tools is based on the use of a pre-commit, like Gerrit[1], or a post-commit workflow, like Upsource[2]. In a pre-commit workflow, changes are reviewed before they are applied to the code, while in post-commit they are first applied and then reviewed.

Pre-commit allows for checking the quality of the changes before they are applied: this allows developers to assess that the code of the changes satisfies the project standards and that the changes do not introduce bugs in the code. The downside with this approach is that it lengthens the release period since the change has to be first approved. This means that developers are not able to work with the changes for a longer time, possibly slowing down the development time.

Post-commit, on the other hand, immediately applies the changes, and then the code is reviewed. This approach allows developers to continue working on new features while waiting for the review to be completed, allowing for a faster release cycle. The downside of this approach is that it is more likely to have bugs introduced in code and that there is no guarantee that the review will ever take place.

In this thesis, I focus on the pre-commit approach, particularly around the Gerrit tool.

### 2.2.1 Gerrit

Gerrit is a popular tool for supporting modern code review and the tool on which I focus in this thesis. To use Gerrit, developers can hook Gerrit up to the Git distributed version control system[3] hosting the source code for their system. After

[1] https://www.gerritcodereview.com/
[2] https://www.jetbrains.com/upsource/
[3] Git

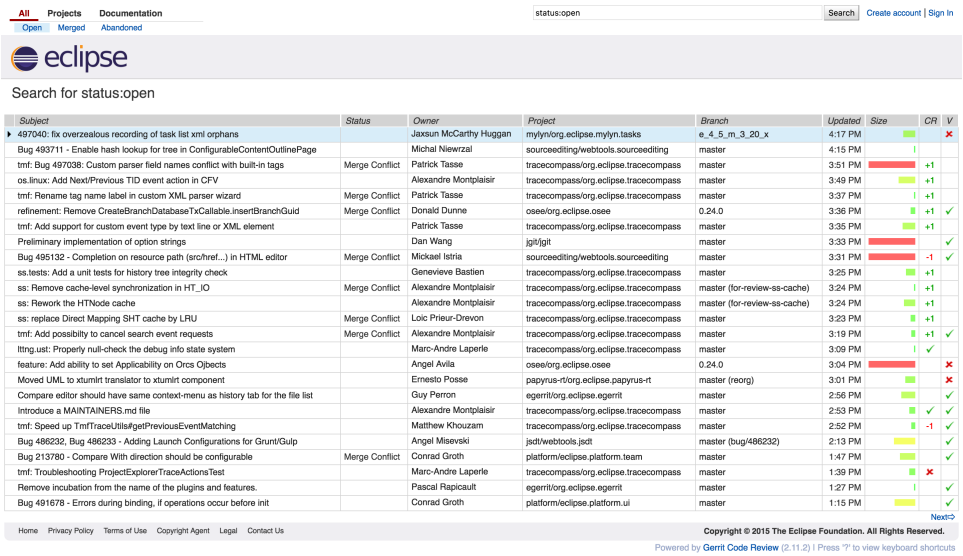



**Figure 2.2:** A sample of the main page of the Gerrit web interface

connecting to the web interface of Gerrit, developers are presented with a list of open code reviews for the project. The developer can then submit a new code review or select an existing code review on which to work. The developers' interactions with Gerrit are asynchronous from each other, providing several benefits:

- there is no need for meetings in which multiple developers convene to review code as was true for older software inspection approaches,
- developers are in charge of their own context switching and can work on code reviews when it fits into their work patterns, and
- developers in multiple timezones or different work patterns can collaborate on reviews.

Figure 2.2 shows a code review in Gerrit. When a review is created, Gerrit shows the same information available from Git about the commit to be reviewed. In addition, Gerrit enables a record of an activity stream of work on the review in the form of comments about the review and a list of patches applied to the original

commit. In many projects, Gerrit can be integrated with a continuous integration service such that for each new patch, integration is run and comments are added by the continuous integration tool to the review. Gerrit uses a pre-commit workflow for reviews. A Gerrit code review works in three stages:

- *Verified*: A change enter this phase as soon as it is verified that it can be merged without breaking the build. This is often done by a continuous integration system, but can also be done manually.
- *Approve*: Any developer can comment on a code review, however, only a specific set of developers, specified by the owner of the Gerrit server, is able to approve a change.
- *Merged*: A change that has been successfully merged in the repository will move to this final stage. It is still possible to comment on the review.

## 2.3 Code Review Completion Time

My initial investigation on the Eclipse dataset indicated that code review stagnation is a phenomenon present in several projects. Similar findings were made by Rigby et al. [15] when they report that if a code review is not closed immediately, it will not be reviewed and will become stagnant. Even in the other cases, some code reviews can take a long period of time before they are resolved. This observation was also made by Rigby and Bird, who observed that 50% of the reviews they studied have a duration of almost a month [13].

Jiang et al. noticed that the integration time of a code review was particularly dependent on the experience of the developer that created it[10]. Similarly, Bosu and Carver observed that code reviews submitted by newcomers to a project tend to receive feedback more slowly, another instance where code reviews may be in the queue for longer than desired [4].

## 2.4 Code Review Recommendation

Rigby and Storey investigated OSS projects, interviewing developers to find out how they were selecting code reviews to be reviewed, finding that often developers tend to review code they are familiar with or that they have edited in the

past[14]. Subsequent work has considered the use of recommenders to speed code review resolution by automatically selecting reviewers.Balachandran proposes Review Bot, a tool that recommends reviewers based on the line change history of the code requiring a review, reporting an accuracy in finding the correct reviewer of 60-92%[2]. Thongtanunam et al. propose a similar approach, in which the expertise is computed from the similarity between the path of the files changed in the code review and the path of files changed in code reviews reviewed by the reviewer in the past[16][17]. They empirically show on historical data from a selection of open source projects that the recommender can accurately recommend the correct reviewer for 79% of the reviews within the top ten recommendations generated. Zanjani et al. presented a new approach, chRev, based on the expertise of the reviewers[19]. In this approach, the expertise is calculated with a combination of previous comments made by the reviewer on reviews on the code being reviewed, number of workdays spent on those comments and period of time since the last comment. Baysal and colleagues take a different approach to recommendation by showing a developer reviews only related to issues on which they have worked[3]. My recommendation approach aims to improve the process of code reviews from a different angle. Instead of assigning the reviewer to the code review, I aim to assign the code review to the reviewer. My approach also differs in being agnostic to the developer asking for the recommendation: as a result, my recommender is not sensitive to the ebb and flow of developers joining and leaving an open source project or a company involved in a closed source project.

# Chapter 3

# Recommending Code Review Ordering

I propose the use of a recommender to help reduce stagnation of code reviews and reduce the duration of time that code reviews remain open. The recommender uses information from open code reviews to rank the code reviews and suggest which review should be worked on next. The goal of the recommender is to optimize the overall handling of reviews. A recommender for code review ordering could be easily integrated into code review tools, such as Gerrit, that present a list of open reviews to developers, without requiring any change in the front end. In addition to an overall reduction in time reviews are open, software developers could also benefit from reducing or eliminating the time necessary to choose a review on which to work. The algorithms I investigate in this thesis are based on metrics that are accessible from the code review as soon as the review is created. I consider two metrics to embed in algorithms: 1) lines of code modified by the code change causing the review to occur and 2) edit actions representing syntax changes to files involved in the change.

## 3.1 Lines of Code

A simple, but still often turned to metric when considering code, is a count of lines of code. A benefit of using the number of lines of code in a code change to

represent the complexity of the change is the simplicity of computing the metric. The idea behind using lines of code for ordering code reviews is that the more lines of code that have been changed, likely the harder it will be for a reviewer to understand the scope of the change.

For my investigation of recommenders, I use the lines of code as reported by Gerrit for a change. Gerrit, like Git, uses information from running `diff` on the files associated with the change. Gerrit records the number of lines inserted and deleted in each file mentioned in the change. When using lines of code in a code review ordering recommender, I sum the number of lines added and the number of lines removed for each file in the code review.

## 3.2 Edit Actions

The simplicity of lines of code as a proxy metric for complexity is that one single line change can be more complex than to understand a many line, simple change.

Another way to assess the complexity of change in code is through *edit actions*, as explored by Falleri et al. [8]. Edit actions are based on a syntactic understanding of the change, and are computed from the AST[1] of the code. Edit actions can be of four possible types:

- an *addition*, indicating that nodes have been added to the syntax tree,
- a *deletion*, indicating that nodes have been removed from the syntax tree,
- an *update*, indicating that a syntax node (and its children) have been modified, but remains of the same type, and
- a *move*, indicating that the syntax node (and its children) have been moved to another parent in the syntax tree, without having been modified.

The edit script of a change is defined as the sequence of edit actions necessary to move from one version of the file to next one. Computing the minimum edit script for file in a given change, defined as the edit script that requires the least number of edit actions, is an NP-hard problem. To approximate the edit script for each file in a change described in a review, I use the Gumtree[2] tool. Gumtree uses a

[1] Abstract Syntax Tree

[2] https://github.com/GumTreeDiff/gumtree/wiki

heuristic algorithm to calculate the minimum edit script from the AST of the code. Figure 3.1 shows an example of the minimum edit script generated by Gumtree for a simple piece of Java code that has been changed. In this example, yellow represents an *update*, green an *addition* and blue a *move*.

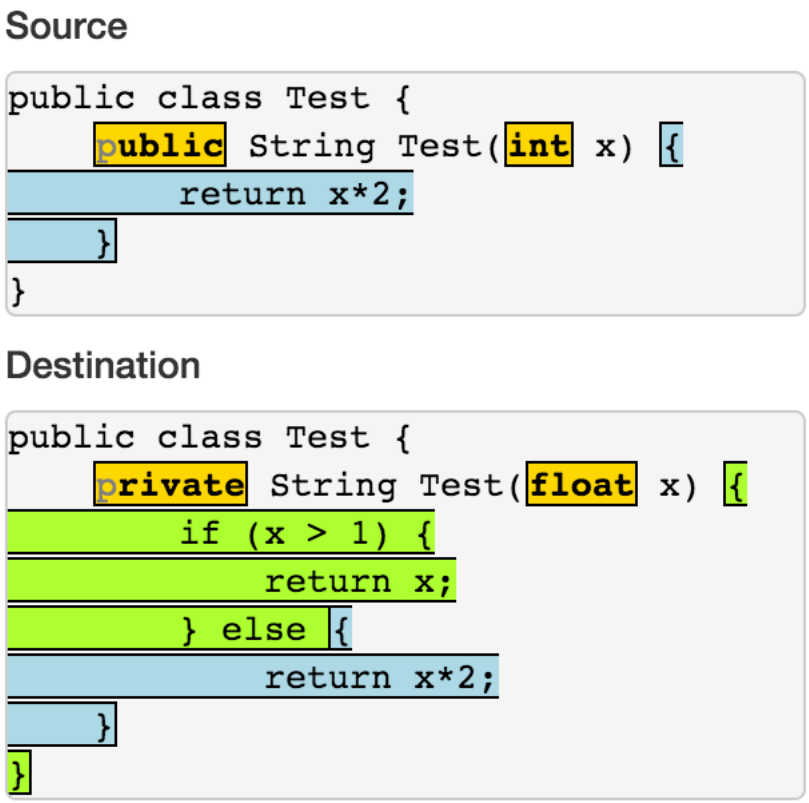



**Figure 3.1:** Output example of the Gumtree tool. Yellow represents an *update*, green an *addition* and blue a *move*.

Edit actions allow us to separate movements and changes in files from additions and deletions, as compared to using lines of code. However, the use of Gumtree is limited. First of all, Gumtree requires a parsers to generate the ASTs. Currently, Gumtree supports only 4 languages: Java, C, JavaScript and Ruby. Gumtree uses a heuristic algorithm, computing only an approximation of the solution. There are some extreme cases in which Gumtree returns an errorenous results or is not able

to compute the results. These cases seem to be caused by specific conditions that appear relatively rarely.

## 3.3 Recommendation Algorithms

I decided to initially opt for a greedy approach: I use the above mentioned metrics to define four recommendation algorithms:

1. lines of code that orders the code reviews from least lines of code changed to most lines of code changed ($R_{locMin}$),
2. lines of code that orders the code from most lines of code changes to least changed ($R_{locMax}$),
3. edit actions that orders the code reviews from least edit actions in the review to most edit actions ($R_{editMin}$), and
4. edit actions that orders the code reviews from most edit actions in the review to least edit actions ($R_{editMax}$).

I investigate the ordering of code reviews from both least to most of each metric and vice versa as there is no basis to determine whether tackling the likely hardest review first is better than last.

# Chapter 4

# Simulation

The best way to evaluate the different algorithms for recommending the order in which to proceed with code reviews would be to build the algorithms into a standard code review tool, put the algorithms into use for an extended period of time and compare against the history of duration times for code reviews of the project. Such an evaluation is very costly in terms of people's time and effort.

Instead of starting with this costly approach, I chose to compare the effectiveness of the algorithms using a simulation approach. Simulation approaches are often used to evaluate recommendation systems for software development [18]. With a simulation approach, I apply an algorithm to order code reviews for historical project data and compute estimates of the completion time of the reviews if such an ordering was used. A simulation allows us to initially reproduce the original process and observe the performances of the various algorithms.

I start my description of the simulation by explaining the project data used. I then explain how I compute the actual duration ($D_a$) and the actual effort ($E_a$) for each code review in a project's history. Using these values, I can define the simulation: the goal of the simulation is to estimate a duration ($D_e$) for each code review if the algorithm's ordering was respected. I defer a discussion of the threats to validity until after I present the results (Chapter 6).

**Table 4.1:** The Eclipse Foundation dataset

| Project | # Code Reviews | # Lines Of Code |
|---|---|---|
| egit | 4,636 | 16,563 |
| org.eclipse.linuxtools | 4,441 | 239,176 |
| jgit | 4,255 | 168,864 |
| org.eclipse.sirius | 2,504 | 382,459 |
| org.eclipse.osee | 2,451 | 696,706 |
| org.eclipse.tracecompass | 1,894 | 196,344 |

## 4.1 Eclipse Foundation Data

The data I use for the simulation study comes from the Gerrit repository of the Eclipse Foundation[1]. I sorted Eclipse projects by the number of successfully merged code reviews and chose the top six for study. I took this approach to ensure enough data to see trends in the ordering algorithms simulated. Table 4.1 shows the project used in the simulation study. The data reported for code reviews is from 2010-08-19 to 2016-03-06.

Before beginning the simulation, I had to clean the dataset to remove code reviews with confusing metadata. These reviews were easily identifiable because their creation date happened after the merge date. All of the faulty entries belong to the first 5,000 reviews and have the same creation date (2012-2-10), suggesting the data problem might be related to the introduction of Gerrit for Eclipse Foundation projects.

All of the projects in the simulation are written in Java and represent a variety of project sizes from tens of thousands of lines to code to half a million of lines of code (Table 4.1).

## 4.2 Actual Duration

Actual duration ($D_a$) is the time from the creation of a code review to its completion as obtained from the historical data of Gerrit. The actual duration of a review is computed as the difference between the time the code review was opened and the

[1] https://eclipse.org/org/foundation/, verified 3/7/16

time the code review was merged, computed using the timestamps saved in the code review. I report $D_a$ in hours.

Duration is an important factor in code reviews: a code review that has been left open for too long is at risk to become stagnant. In some cases, an open code review might delay the project, particularly if it affects critical parts of the code.

## 4.3 Actual Effort

To run the simulation, I need to determine how much effort is needed to resolve a given code review and how much effort is available from project personnel on a particular day to work on code reviews. I define effort of a given review ($E_a$) as the sum of the number of messages, number of patches and number of developers involved in that code review.

Effort is an important factor in my evaluation because it allows us to quantify the amount of work that was put on the code review. A code review that involved more reviewers and went through multiple iterations required more work than a code review with a single version and only one reviewer. In using this definition, I am dependent on the externalized activity that happened on a code review before it was merged. I discuss the impact of this choice on the validity of the results in Chapter 6.

To give a sense of the data on which I run simulations, Figure 4.1 shows the actual duration ($D_a$) (Y-axis) in relation to the actual effort ($E_a$) (X-axis) of each code review in the `JGit` project. An analysis of the plot indicates that $D_a$ is less than a day for half of the code reviews; these code reviews are all plotted on the bottom left of the graph as they typically also have a small $E_a$ value. The overall average $D_a$ is higher because of the code reviews plotted on the right side of the graph. In Figure 4.1, I have also plotted regression curves for degrees 1, 2 and 3. I will use these regression lines later in the thesis to compare to the results of the recommendation algorithms.

## 4.4 Effort per Hour

For the simulation, I also need to determine, for each project, the amount of effort developers on the project spend on average over a unit of time. I use effort per hour

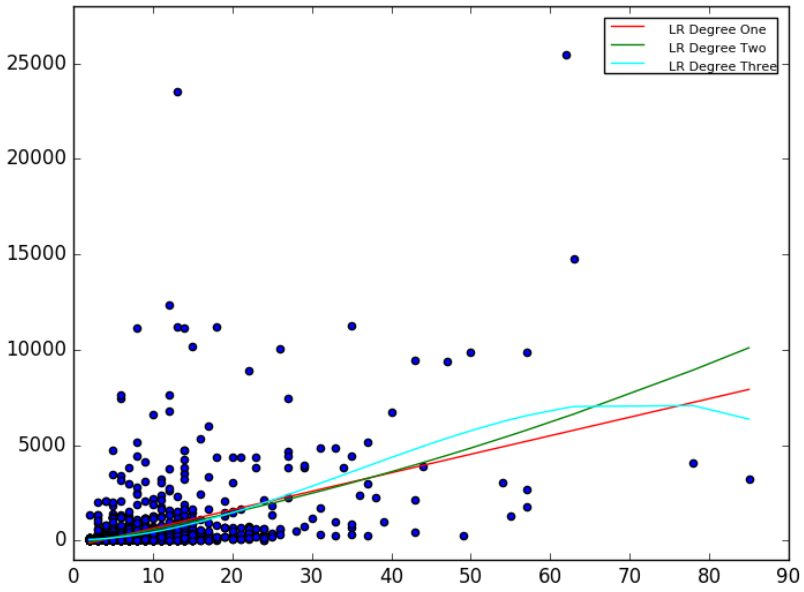



**Figure 4.1:** Actual duration ($D_a$) of code reviews for `JGit`, sorted by actual effort ($E_a$)

($E_h$) as the average amount of effort spent per hour. To calculate this value, I divide the sum of effort for every code review in the project and by the number of hours between the creation of the first code review and the last registered activity on any code review.

One might note the large differences of values between $D_a$ and both $E_a$ and $E_h$ for most projects. These differences are the result of how the values are calculated. While $D_a$ is based on the values of actual duration from the code reviews, $E_a$ and $E_h$ are derived from the time difference, in hours, between the creation of the first code review and the last registered activity. For this reason, $D_a$ is affected by the co-existence of multiple open code reviews at the same time, while the interval of time calculated is not.

Table 4.2 shows the value of these three metrics for each project in the simulation. The first two columns show the values for the average actual duration ($D_a$)

**Table 4.2:** Duration and effort in hours for the Eclipse dataset

| **Project Name** | Avg. $D_a$ | Avg. $E_a$ | $E_h$ |
|---|---|---|---|
| egit | 268.27 | 6.46 | 0.34 |
| org.eclipse.linuxtools | 138.58 | 7.41 | 0.81 |
| jgit | 358.32 | 6.99 | 0.41 |
| org.eclipse.sirius | 190.07 | 8.13 | 0.79 |
| org.eclipse.osee | 76.37 | 8.06 | 0.23 |
| org.eclipse.tracecompass | 303.08 | 9.68 | 1.33 |

and the average actual effort ($E_a$). The average actual duration varies greatly between projects and does not show any correlation with the project size (as reported in Table 4.1. Average effort ($E_a$) is in a much smaller range between 6 and 9, hinting that the effort per code review in terms of externalized actions is roughly the same between projects.

The third column of Table 4.1 shows the computed $E_h$ for each project in the dataset studied. All of these values are quite low compared to their respective $E_a$, meaning that it will be unlikely for a code review to be closed immediately even if it is the only open review.

## 4.5 Simulation and Estimated Duration

The simulation proceeds for a code review ordering algorithm by iterating over the data, in order of the project history, and estimating the amount of time that each code review would have taken if it was worked on in the order suggested.

Specifically, the simulation keeps a list of open code reviews, ordered increasingly by creation date, and has a clock, initialized at the creation date of the first code review. The clock ticks hour by hour and runs until the date of the last activity registered on Gerrit. At every iteration, the following actions are executed in order:

- The clock is moved forward by an hour.
- The list of open code reviews is updated by adding the code reviews opened in the last hour.

- The list of open code reviews is sorted following the current code review ordering algorithm being analyzed.
- The amount of effort available for the hour is assigned to the code reviews, starting from the top one. Each time a code review is assigned an effort at least as large as its $E_a$, the review is considered closed. If there are no code reviews left open, the amount of effort remaining is carried forward.

I repeated the process for every project with every code review ordering algorithm. The simulation enables the computation of an estimated duration ($D_e$) for each review. The estimated duration represents the estimated amount of time that would have taken to close a code review if the recommended order of code reviews was to be followed.

The code of the simulation is available at http://www.cs.ubc.ca/~vivianig/scribe.

# Chapter 5

# Results

I simulated each of the recommendation algorithms on the dataset. In table 5.1, I show the results of the simulation. Each project in the table has two rows, one for the algorithm version using the LOC[1] metric, the other one for the version using the edit actions. The columns are used to show the results for each ordering of the recommendation. For instance, $R_{locMin}$ is represented by LOC indicated in the row and Min represented by the column. For each project, I report the average estimated duration ($D_e$) computed by a particular algorithm, the difference between the average actual duration and the average estimated duration ($D_a - D_e$) and the standard deviation of the estimated duration.

The results show that the $R_{locMin}$ algorithm based on ordering the reviews from the least lines of code changes to the most changed, shown in the Min column, always produces an estimated duration ($D_e$) less than the actual duration, a desirable result. This result can be seen because the values of the difference in the actual and estimated duration (Avg $D_a - D_e$) are always positive (and large). With two exceptions, the $R_{editMin}$ also performs better than what occurred in reality. The two exceptions are `org.eclipse.linuxtools` and `orc.eclipse.osee`.

The algorithms based on ordering from most to least, whether line or edit action based, perform worse than reality for all but one projects. For `org.eclipse.osee`, the algorithm based on lines of code ($R_{locMax}$) performs better than reality. The standard deviation values are fairly large compared to the respective $D_e$ values.

[1] Lines of code

**Table 5.1:** Simulation results

| Project Name | Metric | Avg $D_e$ (Hrs) | | Avg $D_a - D_e$ (Hrs) | | Std. Dev. $D_e$ | |
|---|---|---|---|---|---|---|---|
| | | Min | Max | Min | Max | Min | Max |
| egit | LOC | 47 | 5248 | 221 | -4979 | 412 | 6141 |
| | Actions | 107 | 9810 | 161 | -9540 | 861 | 8593 |
| org.eclipse.linuxtools | LOC | 8 | 1060 | 131 | -921 | 175 | 1884 |
| | Actions | 197 | 10427 | -57 | -10287 | 1276 | 8459 |
| jgit | LOC | 23 | 2091 | 335 | -1731 | 282 | 3356 |
| | Actions | 15 | 374 | 343 | -16 | 146 | 1067 |
| org.eclipse.sirius | LOC | 11 | 425 | 179 | -235 | 101 | 720 |
| | Actions | 13 | 1549 | 177 | -1357 | 183 | 2356 |
| org.eclipse.osee | LOC | 61 | 9929 | 16 | 7961 | 545 | 7966 |
| | Actions | 156 | 14610 | -78 | -14533 | 1071 | 7967 |
| org.eclipse.tracecompass | LOC | 42 | 5422 | 261 | -5117 | 341 | 3413 |
| | Actions | 50 | 5734 | 253 | -5429 | 351 | 3439 |

These results suggest that either $R_{locMin}$ or $R_{editMin}$ may be useful to subject to human evaluation in the context of actual use. A decision on which algorithm to first subject to human evaluation requires a deeper investigation of the trends indicated in Table 5.1. I run an detailed analysis on two projects, `JGit` and `EGit`, focusing only on the algortithm sorting in ascending order. The complete set of results are available in Appendix A.

## 5.1 `JGit`

Figure 5.1 presents the distribution of actual duration of code reviews compared to their effort for the `JGit` project. The duration is on the y-axis, in hours, while the effort is on the x-axis. The figure shows most code reviews have a low value for both effort and duration, as the data points are clustered near the bottom left. There is no recognizable correlation between duration and effort. The three coloured lines represent the regression curves for degrees 1, 2 and 3.

Figures 5.2(a) and 5.2(b) show the the result of the simulations for $R_{locMin}$ and $R_{editMin}$ respectively by plotting estimated duration ($D_e$) (y-axis) against actual effort ($E_a$) (x-axis) sorted by effort. As before, regression lines are shown with degree 1 (red), degree 2 (green) and degree 3 (cyan). These figures show how, compared

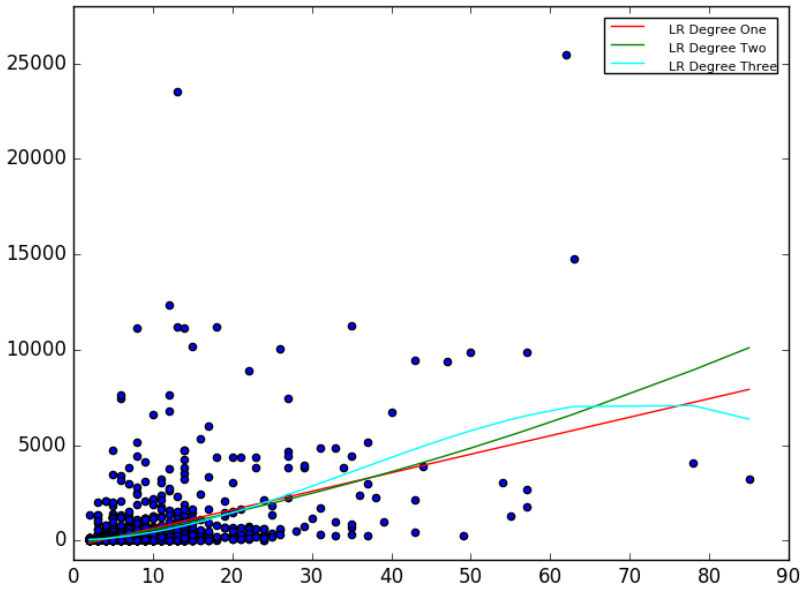
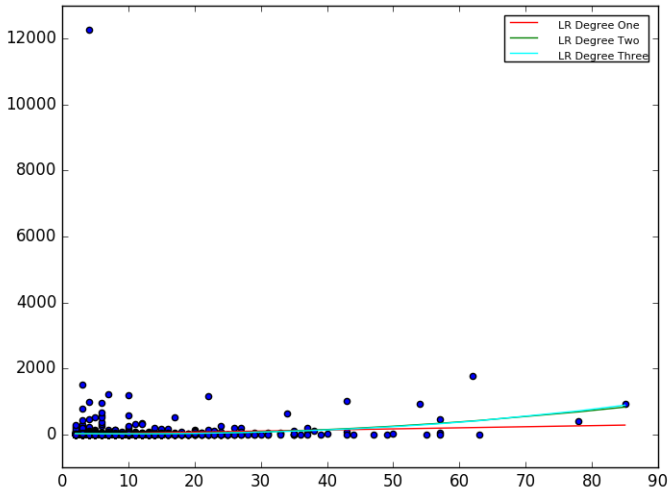


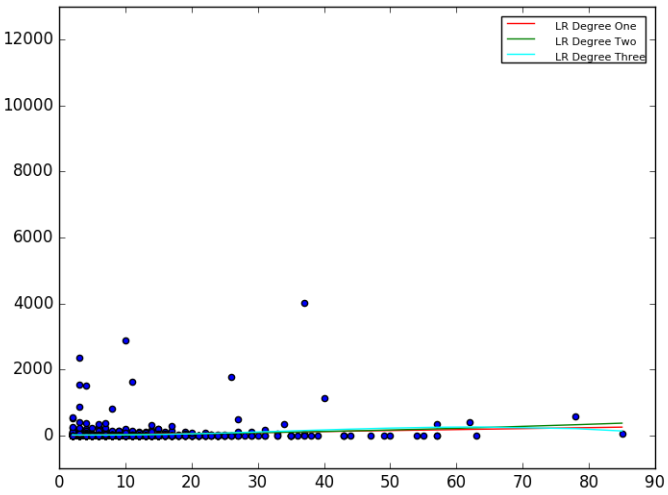

**Figure 5.1:** Actual duration ($D_a$) of code reviews for `JGit`, sorted by actual effort ($E_a$)

to the actual values shown in figure 5.1, the estimated durations of the code reviews have a largely smaller value for both $R_{locMin}$ and $R_{editMin}$. The figures also show how the number of outliers, representing stagnant reviews, has diminished considerably.

Most of the cases where code reviews had large values also disappeared, leading to a more compact graph. Looking at the regression curves, we see how the results obtained using $R_{editMin}$ are less scattered, since even the regression of degree one is able to predict it with the same error as higher degrees. In figure 5.2(a), a single code review (the dot that appears in the top left corner) received a large estimated duration when estimating using $R_{locMin}$, even if it has a fairly low effort value. This outlier is the result of my algorithm being nothing more than a greedy algorithm: because I am considering the ordering of reviews by least estimated effort to largest estimated effort, it can happen than certain code reviews with a low

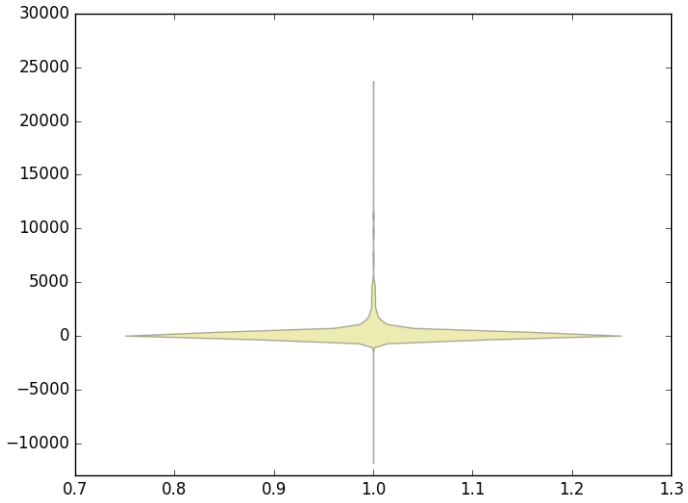



(a) Estimated Duration ($D_e$) computed using $R_{locMin}$v. Effort ($E_a$) for `JGit`

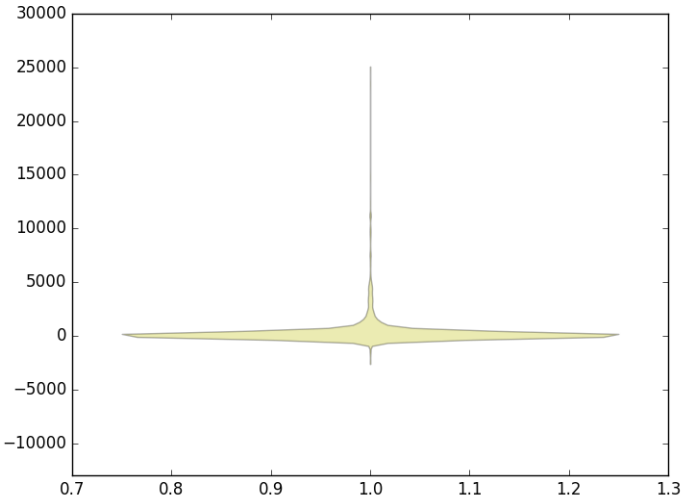



(b) Estimated Duration ($D_e$) computed using $R_{editMin}$v. Effort ($E_a$) for `JGit`

**Figure 5.2:** Simulation results for `JGit` using the $R_{locMin}$ and $R_{editMin}$ algorithms




(a) Difference between actual duration ($D_a$) and estimated Duration ($D_e$) computed using $R_{locMin}$ for `JGit`




(b) Difference between actual duration ($D_a$) and estimated Duration ($D_e$) computed using $R_{editMin}$ for `JGit`

**Figure 5.3:** Violin plots of the difference between the two estimates and the real durations for `JGit`

estimated effort are delayed. I describe some possible solution for this problem in Chapter 7.

Figure 5.3 shows the difference between the actual duration and the estimate from the $R_{locMin}$ and $R_{editMin}$ algorithms, using a violin plot. In the same way as the previous plots, the y-axis represents the duration in hours and the x-axis represents the effort. These plots are useful to understand the degree of improvement in my estimates. In figure 5.3(a), most of the differences are positive, indicating that the estimates produced by the algorithms are better than the actual durations in the majority of cases. There are few outliers, some of them positive and one of them negative. The negative outlier is the same code review described in the previous paragraph. The positive outliers indicate code reviews that either got resolved immediately, in the case of small effort values, or were introduced in a situation were all other code reviews had already been solved, in the case of a large effort value. Figure 5.3(b) presents a more scattered situation. There are no differences with a large negative result, but there are more differences with a negative result. The two plots are similar, suggesting that the two algorithms return a similar ordering in many cases.

## 5.2 `EGit`

Figure 5.4 shows the distribution of actual duration of code reviews compared to their effort for the `EGit` project. The duration is on the y-axis, while the effort is on the x-axis. Similarly to the `JGit` project in figure 5.1, most code reviews have a low value for both effort and duration, as can be seen by the dots being prevalently located in the bottom right corner. In confront, to figure 5.1 for the JGit project, this plot appears to be less scattered. As before, the three coloured lines represent the regression curves for degrees 1, 2 and 3.

Figures 5.5(a) and 5.5(b) show the the result of the simulations for $R_{locMin}$ and $R_{editMin}$ respectively by plotting estimated duration ($D_e$) (y-axis) against actual effort ($E_a$) (x-axis) sorted by effort. As before, regression lines are shown with degree 1 (red), degree 2 (green) and degree 3 (cyan). These results differ from the simulation on `JGit`: in `JGit`, the simulation using $R_{editMin}$ performed better, for `EGit` the $R_{locMin}$ algorithm performs better. In `EGit`, the simulation using

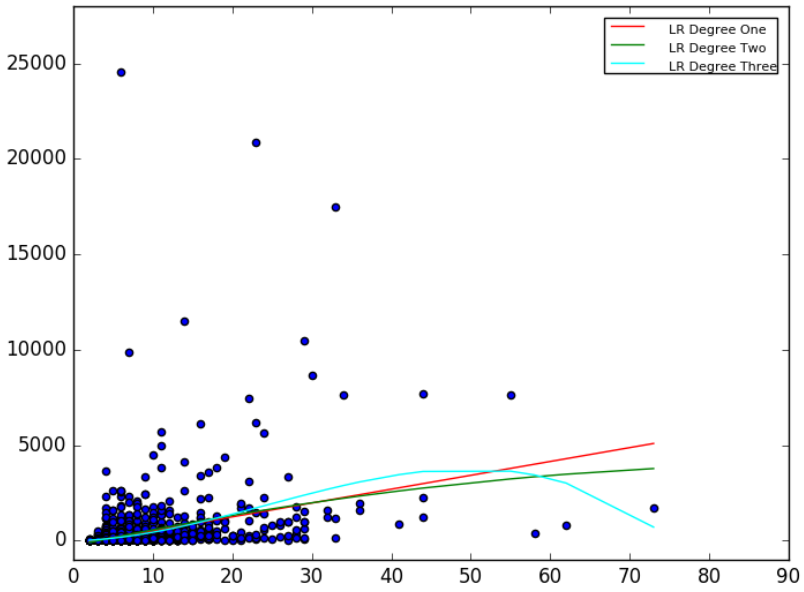
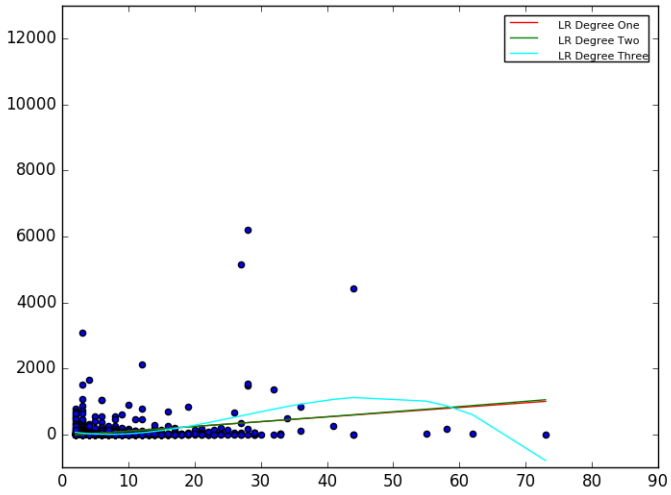


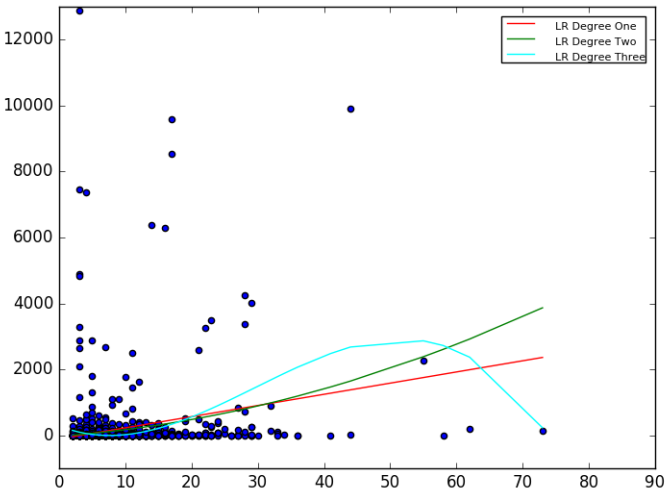

**Figure 5.4:** Actual duration ($D_a$) of code reviews for `EGit`, sorted by actual effort ($E_a$)

$R_{editMin}$, shown in figure 5.5(b) is more scattered and presents many outliers. On the other hand, the simulation using $R_{locMin}$, shown in figure 5.5(a) is more compact and presents few outliers. The regression curves echo this observation: in figure 5.5(a), the curves of first and second degree are nearly the same, and the curve of third degree is close; in figure 5.5(b), on the other hand, the curves differ a lot.

Figure 5.6 shows the difference between the actual duration and the estimate from the $R_{locMin}$ and $R_{editMin}$ algorithms, using a violin plot. In the same way as the previous plots, the y-axis represents the duration in hours and the x-axis the effort. These plots are useful to understand the degree of improvement in my estimates. The graph are fairly similar and positive, indicating that both algorithms perform well. Nonetheless, we can notice how figure 5.5(b) is more compact around the x-axis, while figure 5.5(a) is less scattered but tends upwards. This indicates that,

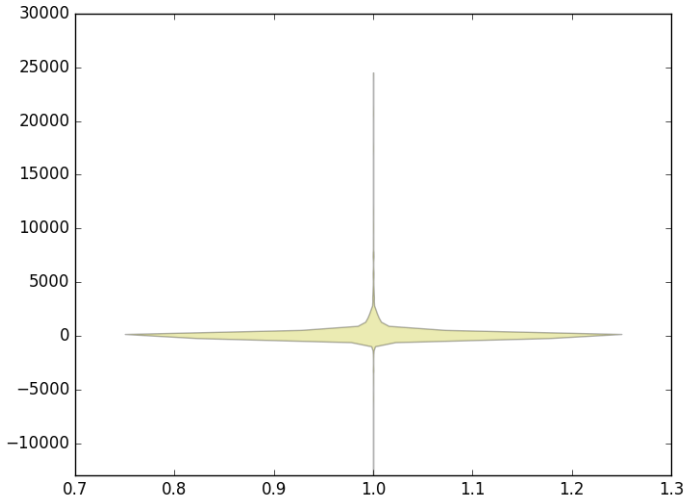



(a) Estimated Duration ($D_e$) computed using $R_{locMin}$v. Effort ($E_a$) for `EGit`

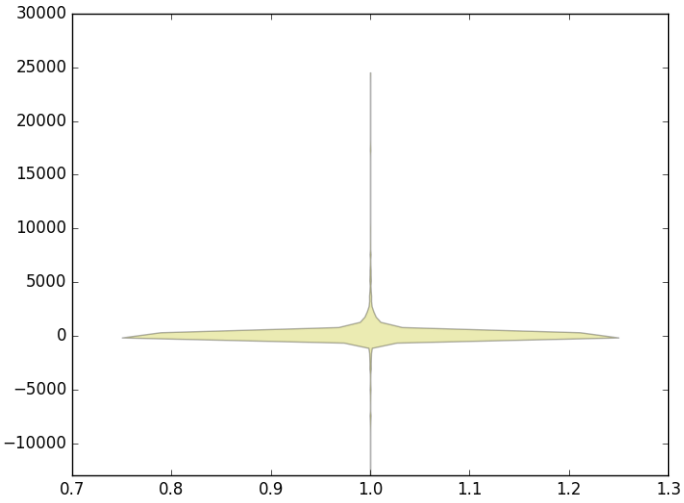



(b) Estimated Duration ($D_e$) computed using $R_{editMin}$v. Effort ($E_a$) for `EGit`

**Figure 5.5:** Simulation results for `EGit` using the $R_{locMin}$ and $R_{editMin}$ algorithms




(a) Difference between actual duration ($D_a$) and estimated Duration ($D_e$) computed using $R_{locMin}$ for `EGit`




(b) Difference between actual duration ($D_a$) and estimated Duration ($D_e$) computed using $R_{editMin}$ for `EGit`

**Figure 5.6:** Violin plots of the difference between the two estimates and the real durations for `EGit`

**Table 5.2:** Best estimation algoritm for each project

| **Project** | **Best Result** |
|---|---|
| egit | locMin |
| org.eclipse.linuxtools | locMin |
| jgit | editMin |
| org.eclipse.sirius | locMin |
| org.eclipse.osee | locMin |
| org.eclipse.tracecompass | locMin |

$R_{locMin}$ performs better than $R_{editMin}$.

## 5.3 Algorithm Choice

After analyzing in depth the simulation for two of my six projects, I can concluded that $R_{locMin}$ performs better. Table 5.2 shows the best algorithm for each project. Interestingly, in only one project out of six, the `JGit` project, $R_{editMin}$ performs better than $R_{locMin}$. In all other cases, $R_{locMin}$ performs better. Based on these results, and considering the higher computation of cost to compute edit actions compared to the cost to compute lines of code, I argue that $R_{locMin}$ should be subject to further human evaluation in real use.

# Chapter 6

# Threats to Validity

Every study must make choices that affect the validity of the results. I first discuss choices that affect the results I report. I then discuss choices that affect whether my results may hold when applied to order code reviews for other projects.

## 6.1 Internal Validity

In the simulation I used to evaluate the ordering algorithms, I assume that effort is spent on code reviews for the entire 24 hours of each day. This assumption does not reflect reality in which people work a limited number of hours a day. I believe my simulation is still valid because my calculation of the effort available per hour is averaged over the day. Thus, even though the effort is spread over 24 hours a day by the simulation, the total amount of effort is the same. For this reason, I argue that the estimate used by the simulation is valid.

The effort computed per review ($E_a$) may not accurately represent the amount of work actually required to complete the review. My computation for $E_a$ is able to measure only the amount of activity recorded in the review, such as in the online discussion; it is unable to measure the amount of work put into such actions as developing patches. My intent with my estimate of actual effort ($E_a$) is to capture the essence of work on code reviews.

Due to inconsistent metadata, as described in Chapter 4, I cleaned the dataset, removing a number of code reviews from my dataset and the simulation. Even with

the cleaning, the projects all contain a large number of code reviews, thus I believe the effect on the overall results is minimal.

A final threat comes from imprecision in the calculation of the metrics used in the algorithms. As described in Chapter 5, both lines of code and edit actions suffer from possible false results. In these cases, a single outlier can have a ripple effect over the entire simulation, inflating (or deflating) the estimated durations. I argue that the number of outliers is far smaller than the general trend: when the effect of the outlier is applied, it will affect the estimated duration for all the code reviews but only by a small amount.

## 6.2 Construct Validity

My recommendation approach aims to avoid stagnation in modern code review and to reduce overall code review resolution time. In order to evaluate it, I built the simulation to reproduce the original process and observe the effect of the algorithms. This approach might not be suited to claim an improvement in the modern code review process, but it allowed me to quickly understand which algorithms would perform the best before moving to a more structured user study.

I simulated the algorithms on projects from one ecosystem, Eclipse. The ecosystem may represent a limited number of software development processes that affect the data and results. By simulating over seven projects from the ecosystem, I believe I have captured a variety of teams with variations in their work practices. Also, since the simulation is relative to a given project, and not absolute, I believe there is more likelihood that similar results would be seen on projects from other ecosystem.

# Chapter 7

# Discussion and Future Work

I have made many assumptions in my investigation of recommenders to optimize code review processing. I discuss other metrics than those I studied that might be used for a code review ordering recommender, more personalized approaches for recommending which code review to work on and future work needed in evaluating recommenders to improve code review handling.

## 7.1 Additional Metrics

The algorithms for code review ordering I investigated were based on metrics available from the code reviews themselves. There are more metrics in this category that could be investigated. For example, instead of just using lines of code, a metric based on number of files or packages changed might provide better results. Or, path similarities of files modified in a change might be used: the more similar the paths, the less complex the code change may be. Other characteristics of the code reviews that are open and available to be ordered could also be considered, such as the length of time a review is open or the amount of activity that has occurred to date on the review.

## 7.2 Better Algorithms

The greedy approach returns promising results, but can suffer from particular cases. I described an example in chapter 5, in which a code review was delayed for a long

time, even though it should have been dealt with sooner. I argue that an algorithm that takes into account both the greedy order and the time that a code review has been waiting may be beneficial for creating a recommender.

## 7.3 Personalized Recommendations

The recommendation algorithms I have considered in this thesis all provide one ordering for all developers on a project. More complex recommendation algorithms could be investigated that produce a personalized recommendation for each developer available to work on a code review. For instance, the recommender could use knowledge of what code a developer knows (e.g., [9]) to rank code reviews where the developer has knowledge of the code higher in the list. A more complex recommendation algorithm could also take into account the time a developer has available to perform a code review: if the developer indicates only a short time is available, simpler code reviews might be prioritized over others.

## 7.4 Human Evaluation

My current evaluation of algorithms is based on a computer simulation. Although this approach can be useful to compare algorithms, it cannot take into account all of the factors that may affect the resolution of code reviews in practice. A human evaluation of algorithms that perform well in simulation is needed. A human evaluation of $R_{locMin}$ or $R_{editMin}$ could be performed as an A/B (or split) test. Since both of these algorithms order the list of open code reviews, the algorithms could be put into practice for different weeks of a year and the actual duration of reviews compared to the actual duration prior to the use of an ordered code review list. Care would need to be taken in the number of reviews subjected to each treatment and the complexity of the reviews as measured using some of the metrics I have introduced to ensure similar samples.

# Chapter 8

# Summary

Software developers expend significant human effort on code reviews. Yet, despite this effort, code reviews remain open for long periods of time on projects and a number of code reviews go stagnant.

As a means for reducing stagnation and the duration of time required to resolve reviews, I introduce the idea of a code review ordering recommender. An advantage of this approach is that it could be easily put in use in a project as the recommender can be integrated easily into existing code review tools, such as Gerrit.

I introduced four algorithms that might be used to recommend an order for open code reviews. I performed simulation studies on these algorithms on a dataset of six projects from the Eclipse Foundation. I found that the algorithm which ranked the open code reviews from least lines of code to the most lines of code involved in the change to be reviewed ($R_{locMin}$), and the algorithm which ranked the open code reviews from least amount of edit actions based on the syntax of the change to the most ($R_{editMin}$) performed the best of the four algorithms. A more detailed analysis indicates that the effectiveness of those two algorithms varies project by project.

Based on the results of the simulation, the algorithm based on edit actions ($R_{locMin}$) may be the most stable and the most suitable to subject, as a next step, to evaluation in use in a real project.

# Bibliography


[1] A. Bacchelli and C. Bird. Expectations, Outcomes, and Challenges of Modern Code Review. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 712–721, Piscataway, NJ, USA, 2013. IEEE Press. ISBN 978-1-4673-3076-3. URL http://dl.acm.org/citation.cfm?id=2486788.2486882. → pages 1, 5

[2] V. Balachandran. Reducing Human Effort and Improving Quality in Peer Code Reviews Using Automatic Static Analysis and Reviewer Recommendation. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 931–940, Piscataway, NJ, USA, 2013. IEEE Press. ISBN 978-1-4673-3076-3. URL http://dl.acm.org/citation.cfm?id=2486788.2486915. → pages 9

[3] O. Baysal, R. Holmes, and M. W. Godfrey. No Issue Left Behind: Reducing Information Overload in Issue Tracking. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 666–677, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3056-5. doi:10.1145/2635868.2635887. URL http://doi.acm.org/10.1145/2635868.2635887. → pages 9

[4] A. Bosu and J. C. Carver. Impact of Developer Reputation on Code Review Outcomes in OSS Projects: An Empirical Investigation. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '14, pages 33:1—-33:10, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2774-9. doi:10.1145/2652524.2652544. URL http://doi.acm.org/10.1145/2652524.2652544. → pages 8

[5] M. Fagan. *A History of Software Inspections*, pages 562–573. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002. ISBN 978-3-642-59412-0. doi:10.1007/978-3-642-59412-0_34. URL http://dx.doi.org/10.1007/978-3-642-59412-0_34. → pages 4

[6] M. E. Fagan. Design and Code Inspections to Reduce Errors in Program Development. *IBM Syst. J.*, 15(3):182–211, sep 1976. ISSN 0018-8670. doi:10.1147/sj.153.0182. URL http://dx.doi.org/10.1147/sj.153.0182. → pages 1

[7] M. E. Fagan. Advances in software inspections. *IEEE Transactions on Software Engineering*, SE-12(7):744–751, jul 1986. ISSN 0098-5589. doi:10.1109/TSE.1986.6312976. → pages 5

[8] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus. Fine-grained and Accurate Source Code Differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 313–324, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3013-8. doi:10.1145/2642937.2642982. URL http://doi.acm.org/10.1145/2642937.2642982. → pages 11

[9] T. Fritz, J. Ou, G. C. Murphy, and E. Murphy-Hill. A Degree-of-knowledge Model to Capture Source Code Familiarity. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 385–394, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-719-6. doi:10.1145/1806799.1806856. URL http://doi.acm.org/10.1145/1806799.1806856. → pages 33

[10] Y. Jiang, B. Adams, and D. M. German. Will My Patch Make It? And How Fast?: Case Study on the Linux Kernel. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 101–110, Piscataway, NJ, USA, 2013. IEEE Press. ISBN 978-1-4673-2936-1. URL http://dl.acm.org/citation.cfm?id=2487085.2487111. → pages 8

[11] S. Kollanus and J. Koskinen. Survey of software inspection research. *The Open Software Engineering Journal*, 3(1):15–34, 2009. → pages 5

[12] A. N. Meyer, T. Fritz, G. C. Murphy, and T. Zimmermann. Software Developers' Perceptions of Productivity. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 19–29, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3056-5. doi:10.1145/2635868.2635892. URL http://doi.acm.org/10.1145/2635868.2635892. → pages 2

[13] P. C. Rigby and C. Bird. Convergent Contemporary Software Peer Review Practices. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 202–212, New York, NY,

USA, 2013. ACM. ISBN 978-1-4503-2237-9. doi:10.1145/2491411.2491444. URL http://doi.acm.org/10.1145/2491411.2491444. → pages 1, 2, 8

[14] P. C. Rigby and M.-A. Storey. Understanding Broadcast Based Peer Review on Open Source Software Projects. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 541–550, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0445-0. doi:10.1145/1985793.1985867. URL http://doi.acm.org/10.1145/1985793.1985867. → pages 9

[15] P. C. Rigby, D. M. German, and M.-A. Storey. Open Source Software Peer Review Practices: A Case Study of the Apache Server. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 541–550, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-079-1. doi:10.1145/1368088.1368162. URL http://doi.acm.org/10.1145/1368088.1368162. → pages 8

[16] P. Thongtanunam, R. G. Kula, A. E. C. Cruz, N. Yoshida, and H. Iida. Improving Code Review Effectiveness Through Reviewer Recommendations. In *Proceedings of the 7th International Workshop on Cooperative and Human Aspects of Software Engineering*, CHASE 2014, pages 119–122, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2860-9. doi:10.1145/2593702.2593705. URL http://doi.acm.org/10.1145/2593702.2593705. → pages 9

[17] P. Thongtanunam, C. Tantithamthavorn, R. G. Kula, N. Yoshida, H. Iida, and K. i. Matsumoto. Who should review my code? A file location-based code-reviewer recommendation approach for Modern Code Review. In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, pages 141–150, mar 2015. doi:10.1109/SANER.2015.7081824. → pages 9

[18] R. J. Walker and R. Holmes. *Simulation*, chapter Simulation, pages 301–327. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014. ISBN 978-3-642-45135-5. doi:10.1007/978-3-642-45135-5_12. URL http://link.springer.com/10.1007/978-3-642-45135-5_12. → pages 14

[19] M. Zanjani, H. Kagdi, and C. Bird. Automatically Recommending Peer Reviewers in Modern Code Review. *IEEE Transactions on Software Engineering*, PP(99):1–1, jun 2015. ISSN 0098-5589. doi:10.1109/TSE.2015.2500238. URL

http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7328331.
→ pages 9

# Appendix A

# Simulation Results

In this Appendix we provide all the plots for each of the 6 projects used in the simulation described in chapter 4. For each project, we provide the following plots:

- a scatter plot of the distribution of actual duration of code review,
- a scatter plot for each of the estimated durations for the two Min algorithms,
- a scatter plot for the difference between each Min estimation and the actual duration,
- a violin plot for the difference between each Min estimation and the actual duration,
- a scatter plot for each of the estimated durations for the two Max algorithms,
- a scatter plot for the difference between each Max estimation and the actual duration,
- a violin plot for the difference between each Max estimation and the actual duration.

## A.1 EGit

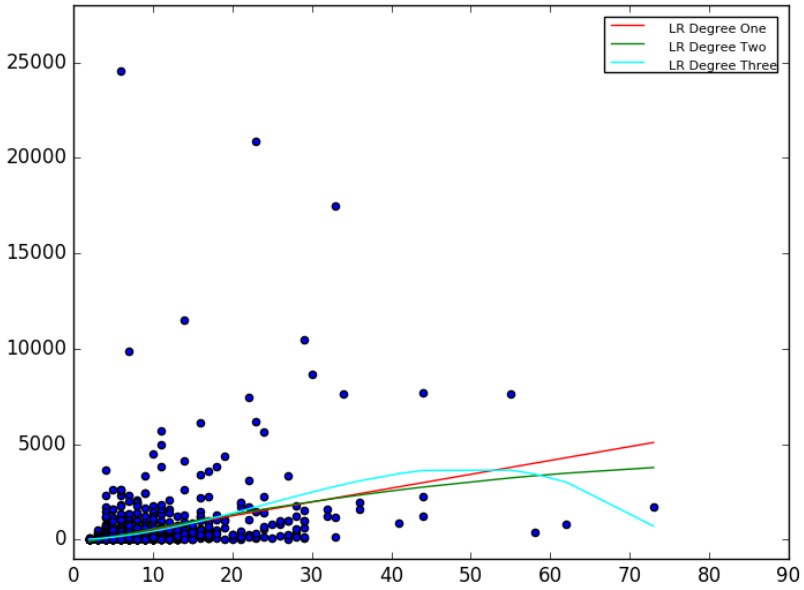
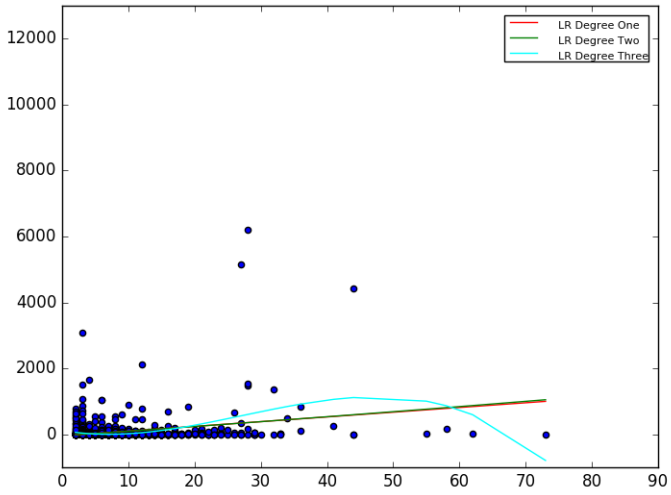
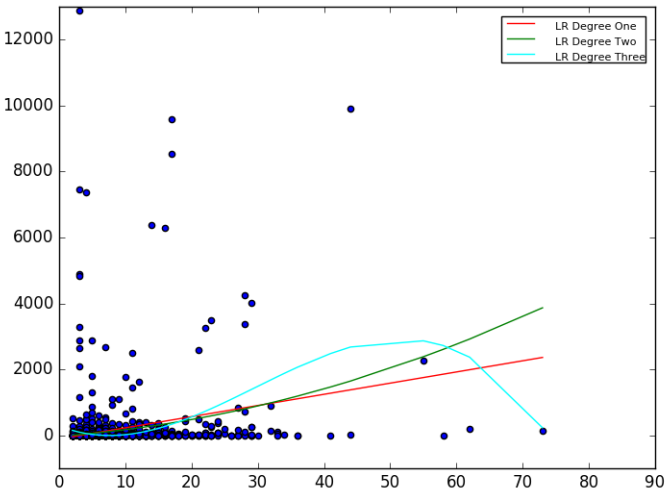



**Figure A.1:** Actual durations ($D_a$) of code reviews for `EGit`, sorted by actual effort ($E_a$)

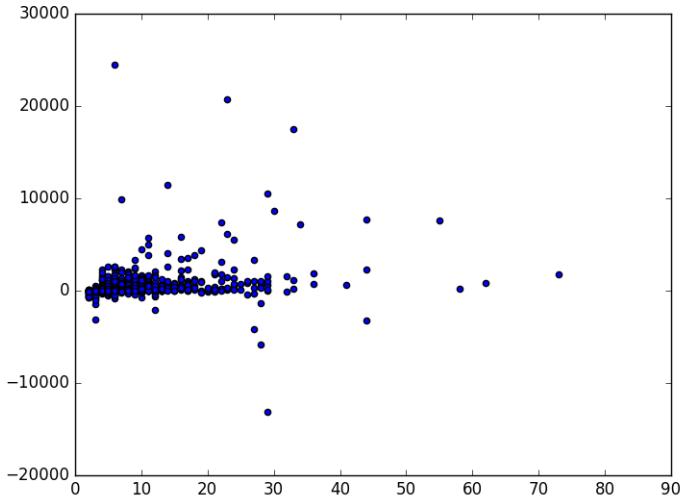



(a) Estimated Duration ($D_e$) computed using $R_{locMin}$ v. Effort ($E_a$) for `EGit`

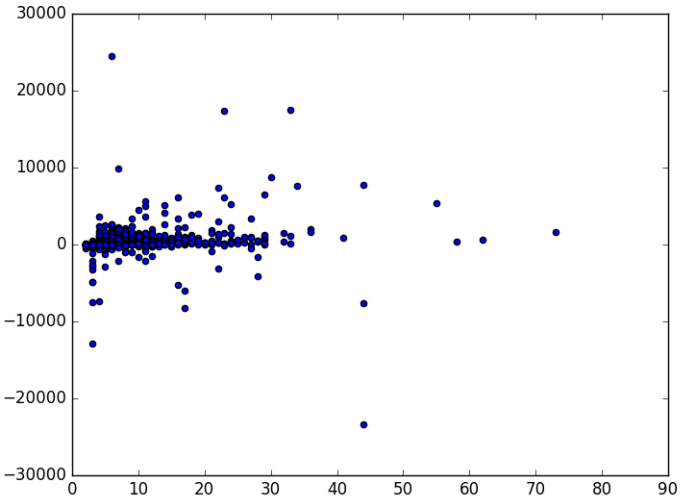



(b) Estimated Duration ($D_e$) computed using $R_{editMin}$ v. Effort ($E_a$) for `EGit`

**Figure A.2:** Scatter plots with regression lines of the estimated durations for `EGit` computed using the min algorithms

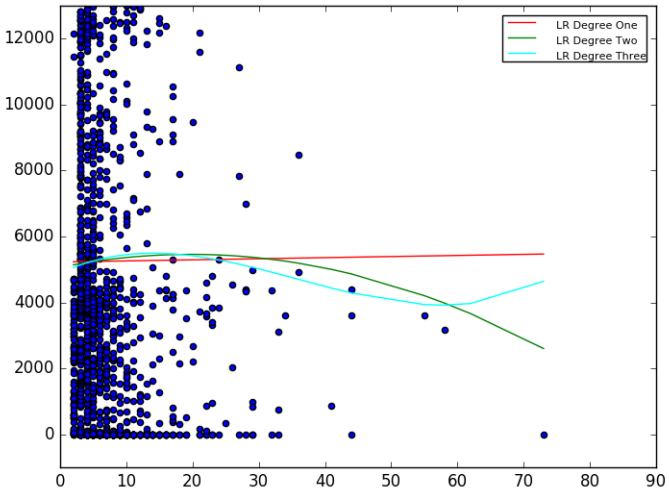



(a) Difference betweent the actual duration ($D_a$) and the estimated Duration ($D_e$) computed using $R_{locMin}$ v. Effort ($E_a$) for `EGit`

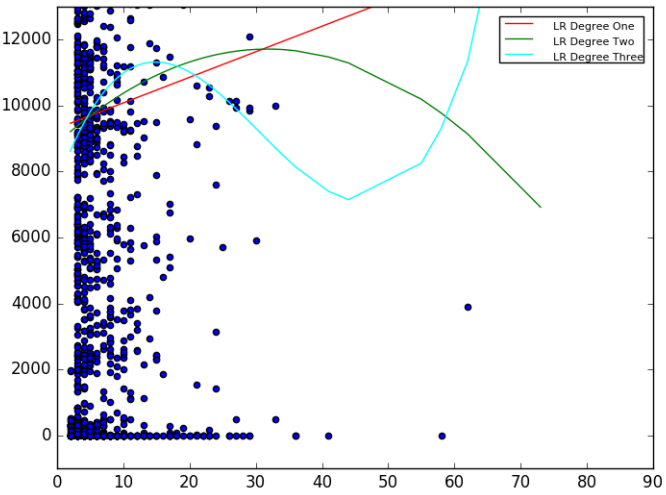



(b) Difference betweent the actual duration ($D_a$) and the estimated Duration ($D_e$) computed using $R_{editMin}$ v. Effort ($E_a$) for `EGit`

**Figure A.3:** Scatter plots with regression lines of the difference between the min estimates and the real durations for `EGit`

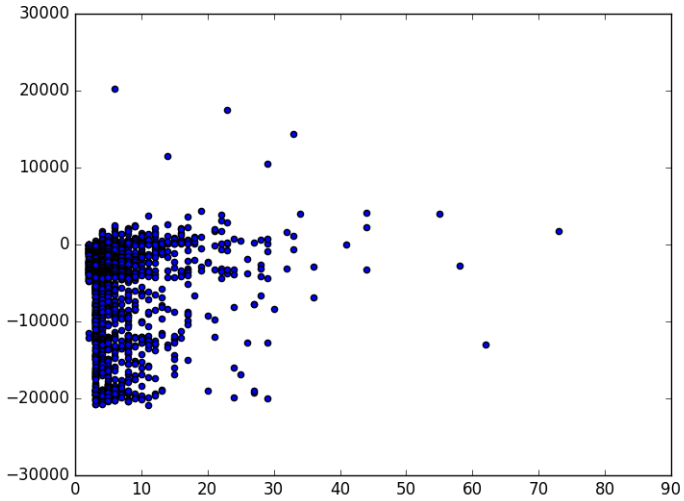



(a) Estimated Duration ($D_e$) computed using $R_{locMax}$ v. Effort ($E_a$) for `EGit`

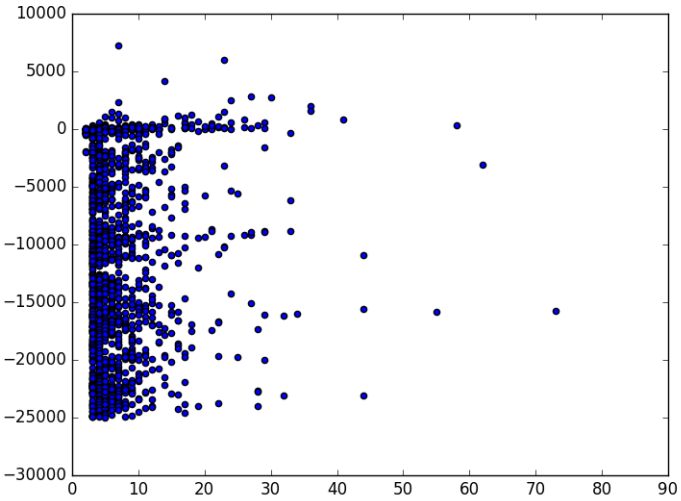



(b) Estimated Duration ($D_e$) computed using $R_{editMax}$ v. Effort ($E_a$) for `EGit`

**Figure A.4:** Scatter plots with regression lines of the estimated durations for `EGit` computed using the max algorithms

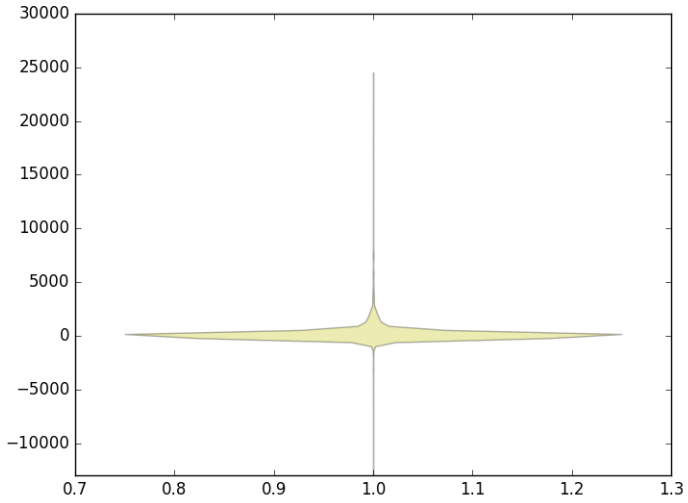



(a) Difference betweent the actual duration ($D_a$) and the estimated Duration ($D_e$) computed using $R_{locMax}$ v. Effort ($E_a$) for `EGit`

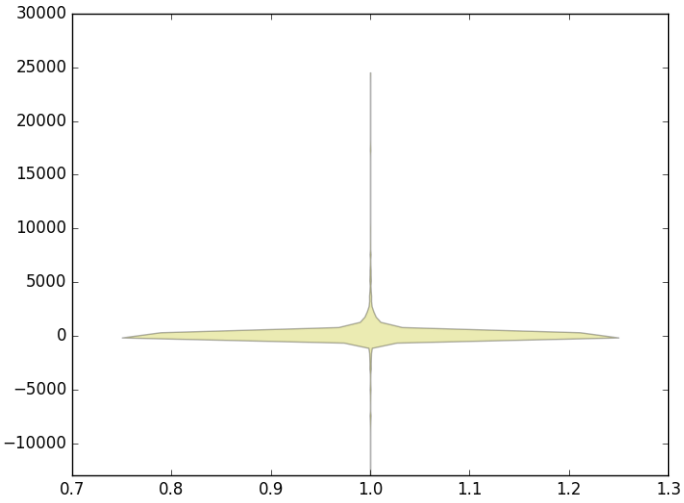



(b) Difference betweent the actual duration ($D_a$) and the estimated Duration ($D_e$) computed using $R_{editMax}$ v. Effort ($E_a$) for `EGit`

**Figure A.5:** Scatter plots with regression lines of the difference between the max estimates and the real durations for `EGit`

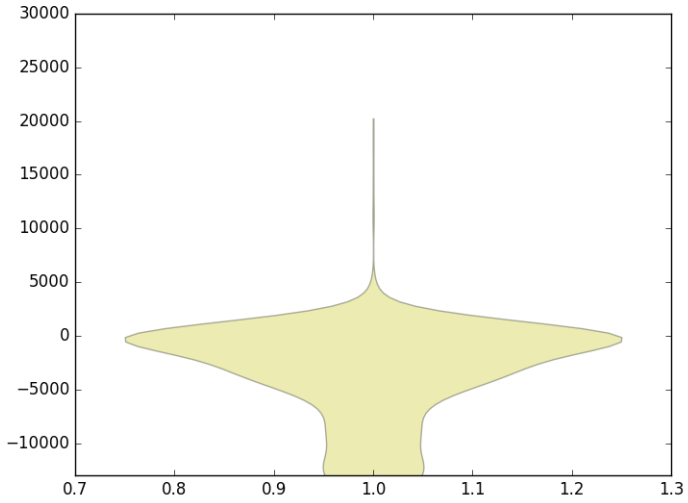



(a) Difference between actual duration ($D_a$) and estimated Duration ($D_e$) computed using $R_{locMin}$ for `EGit`

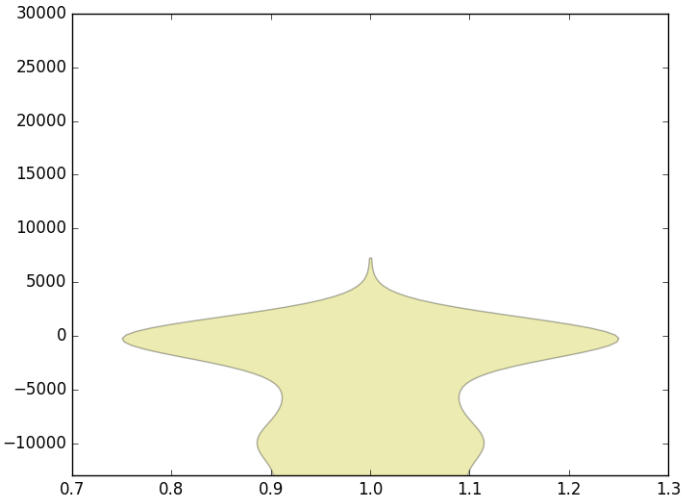



(b) Difference between actual duration ($D_a$) and estimated Duration ($D_e$) computed using $R_{editMin}$ for `EGit`

**Figure A.6:** Violin plots of the difference between the estimates and the real durations for `EGit` for the min algorithms




(a) Difference between actual duration ($D_a$) and estimated Duration ($D_e$) computed using $R_{locMax}$ for `EGit`




(b) Difference between actual duration ($D_a$) and estimated Duration ($D_e$) computed using $R_{editMax}$ for `EGit`

**Figure A.7:** Violin plots of the difference between the estimates and the real durations for `EGit` for the max algorithms

## A.2 Linuxtools

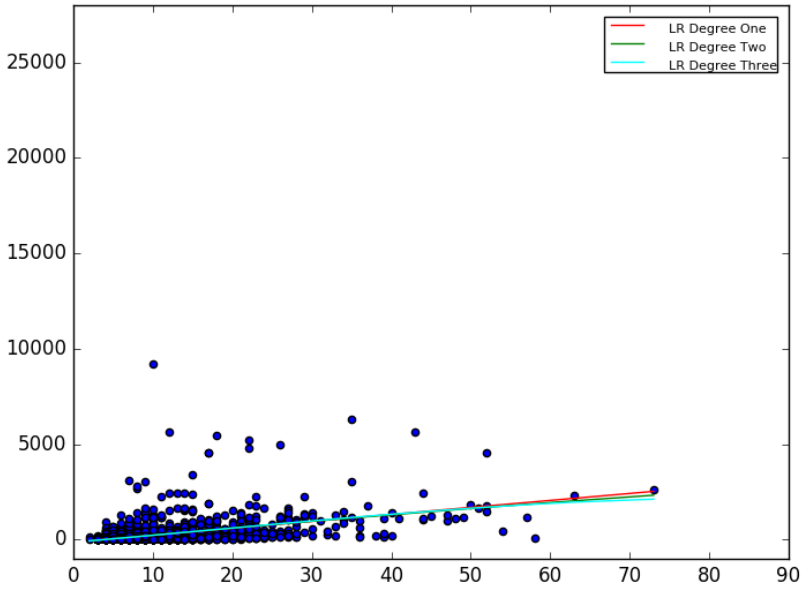
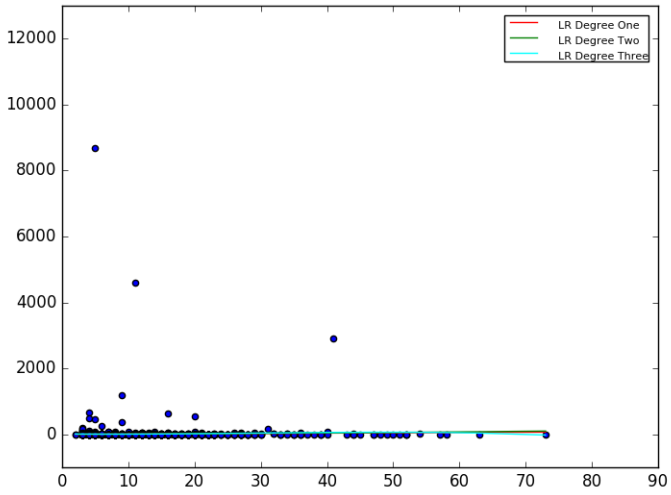
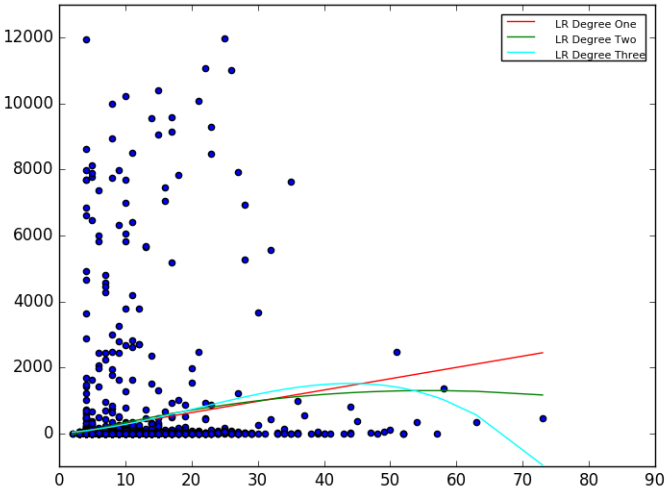



**Figure A.8:** Actual durations ($D_a$) of code reviews for `Linuxtools`, sorted by actual effort ($E_a$)

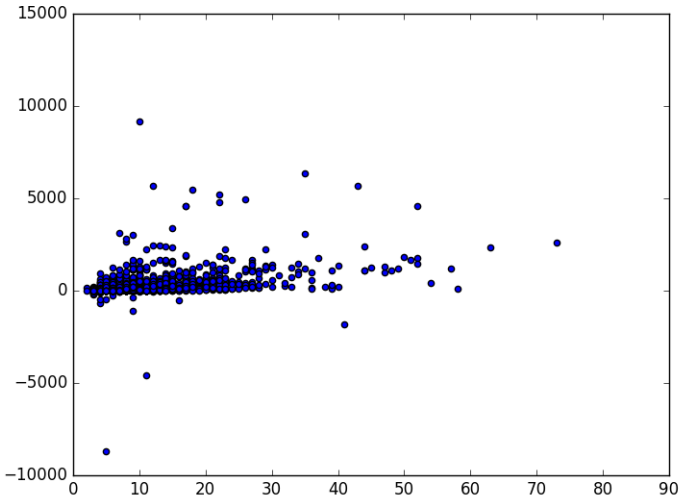



(a) Estimated Duration ($D_e$) computed using $R_{locMin}$ v. Effort ($E_a$) for `Linuxtools`

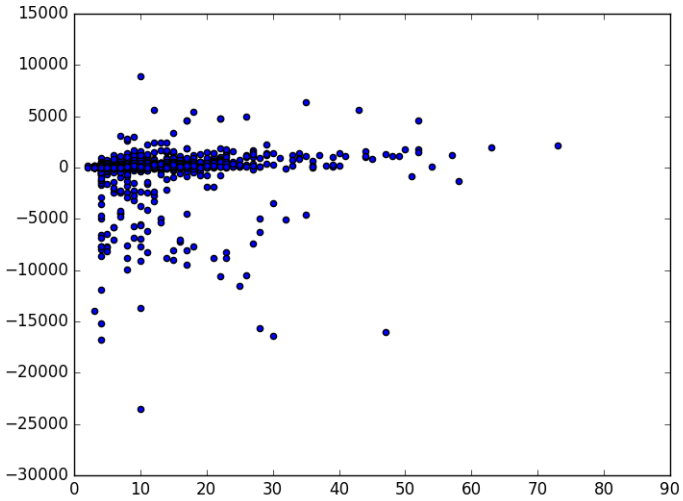



(b) Estimated Duration ($D_e$) computed using $R_{editMin}$ v. Effort ($E_a$) for `Linuxtools`

**Figure A.9:** Scatter plots with regression lines of the estimated durations for `Linuxtools` computed using the min algorithms

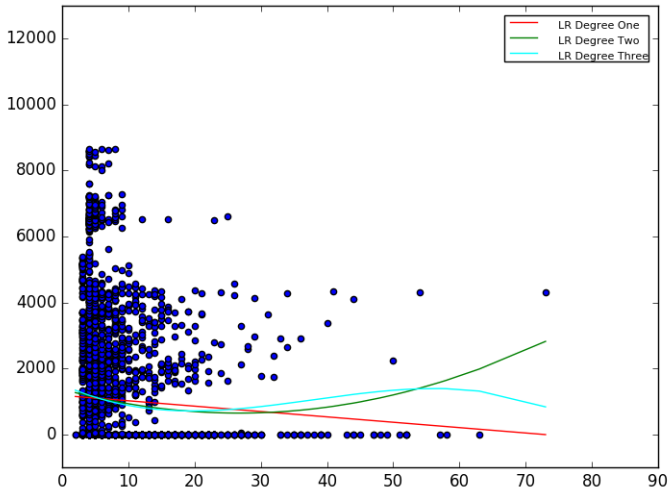



(a) Difference betweent the actual duration ($D_a$) and the estimated Duration ($D_e$) computed using $R_{locMin}$ v. Effort ($E_a$) for `Linuxtools`

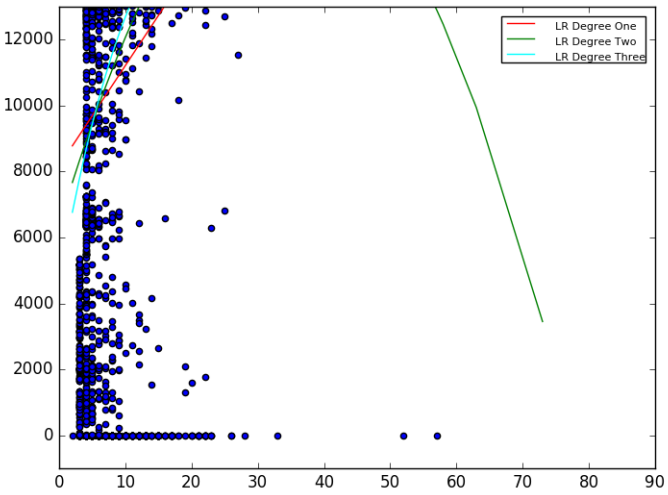



(b) Difference betweent the actual duration ($D_a$) and the estimated Duration ($D_e$) computed using $R_{editMin}$ v. Effort ($E_a$) for `Linuxtools`

**Figure A.10:** Scatter plots with regression lines of the difference between the min estimates and the real durations for `Linuxtools`

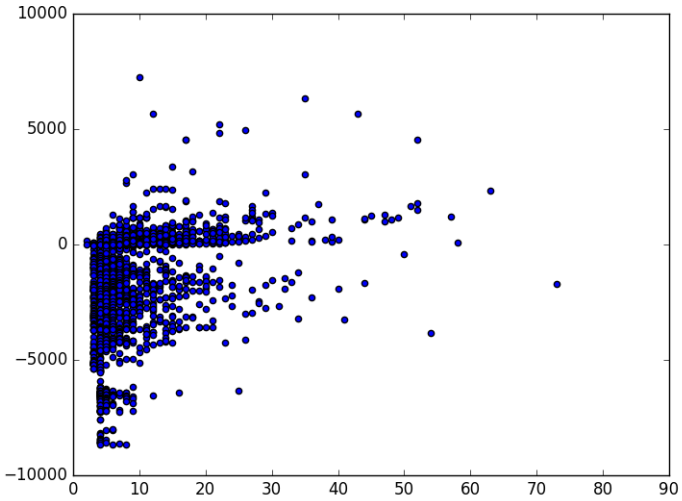



(a) Estimated Duration ($D_e$) computed using $R_{locMax}$ v. Effort ($E_a$) for `Linuxtools`

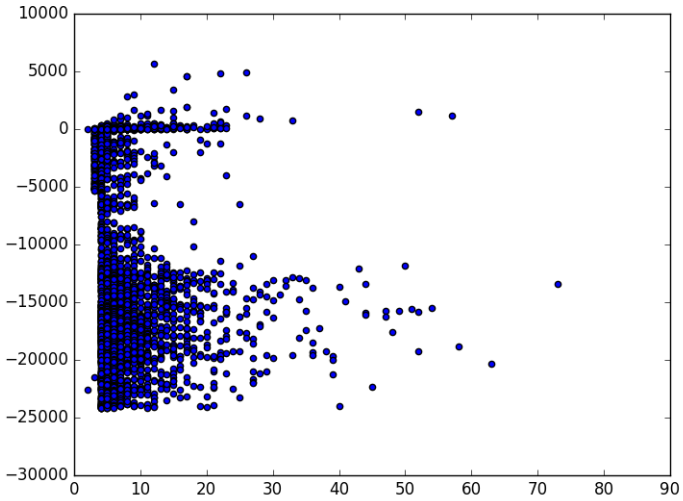



(b) Estimated Duration ($D_e$) computed using $R_{editMax}$ v. Effort ($E_a$) for `Linuxtools`

**Figure A.11:** Scatter plots with regression lines of the estimated durations for `Linuxtools` computed using the max algorithms

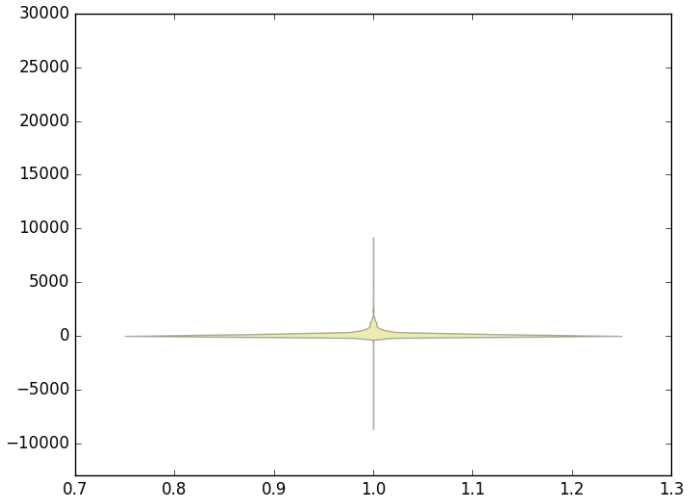



(a) Difference betweent the actual duration ($D_a$) and the estimated Duration ($D_e$) computed using $R_{locMax}$ v. Effort ($E_a$) for `Linuxtools`

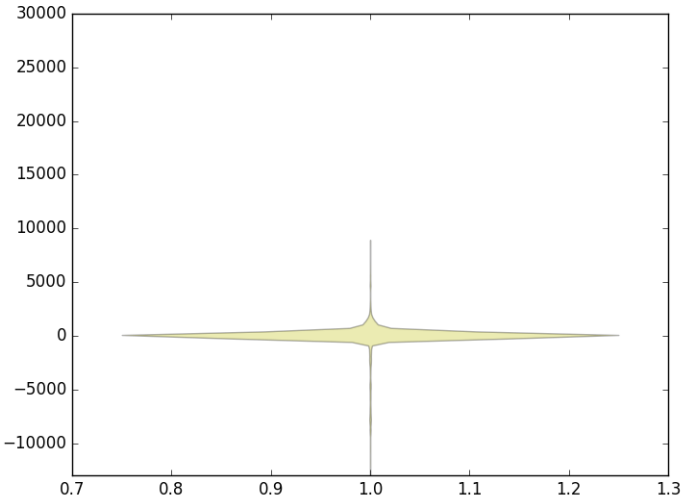



(b) Difference betweent the actual duration ($D_a$) and the estimated Duration ($D_e$) computed using $R_{editMax}$ v. Effort ($E_a$) for `Linuxtools`

**Figure A.12:** Scatter plots with regression lines of the difference between the max estimates and the real durations for `Linuxtools`

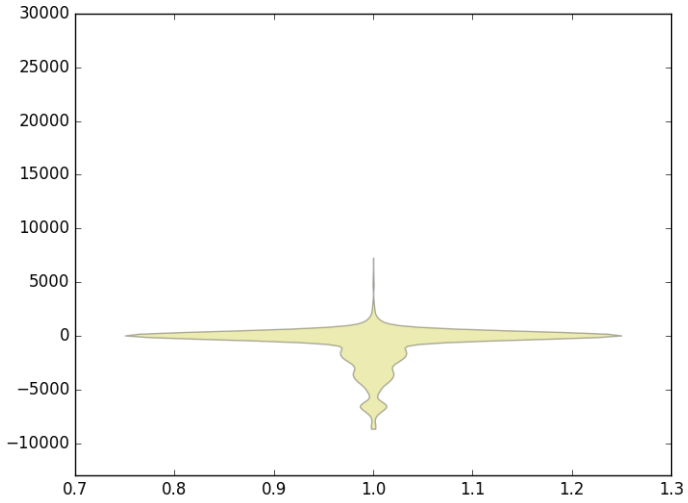



(a) Difference between actual duration ($D_a$) and estimated Duration ($D_e$) computed using $R_{locMin}$ for `Linuxtools`

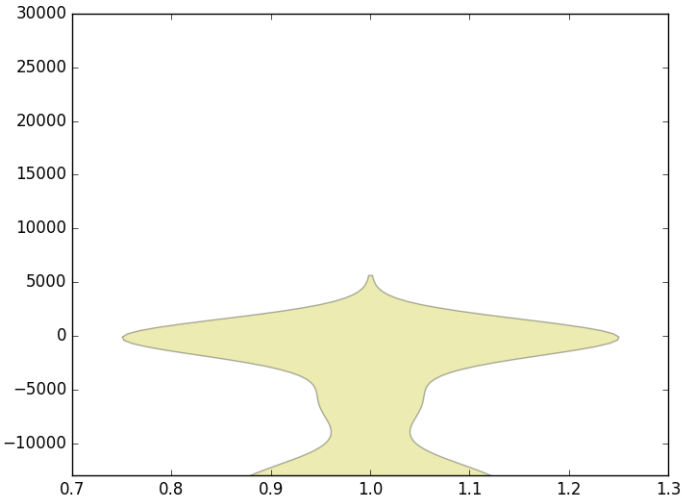



(b) Difference between actual duration ($D_a$) and estimated Duration ($D_e$) computed using $R_{editMin}$ for `Linuxtools`

**Figure A.13:** Violin plots of the difference between the estimates and the real durations for `Linuxtools` for the min algorithms




(a) Difference between actual duration ($D_a$) and estimated Duration ($D_e$) computed using $R_{locMax}$ for `Linuxtools`




(b) Difference between actual duration ($D_a$) and estimated Duration ($D_e$) computed using $R_{editMax}$ for `Linuxtools`

**Figure A.14:** Violin plots of the difference between the estimates and the real durations for `Linuxtools` for the max algorithms

## A.3 JGit

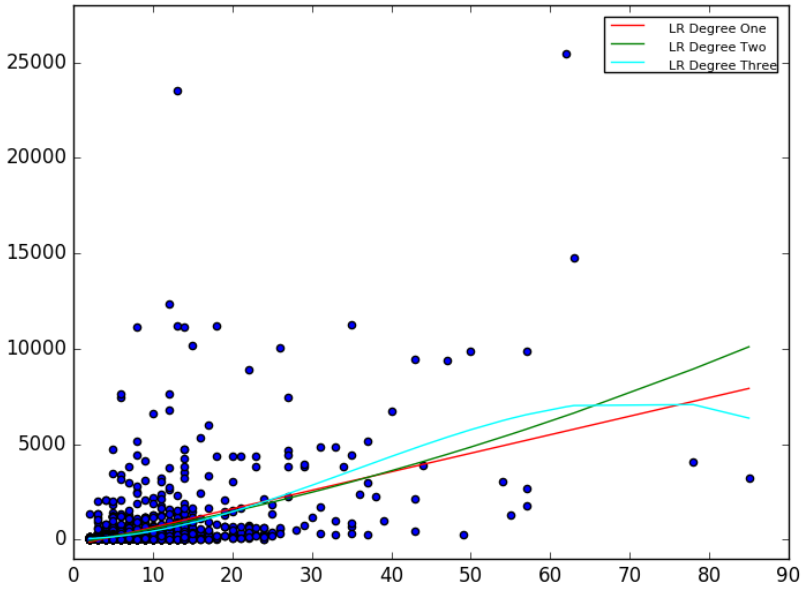
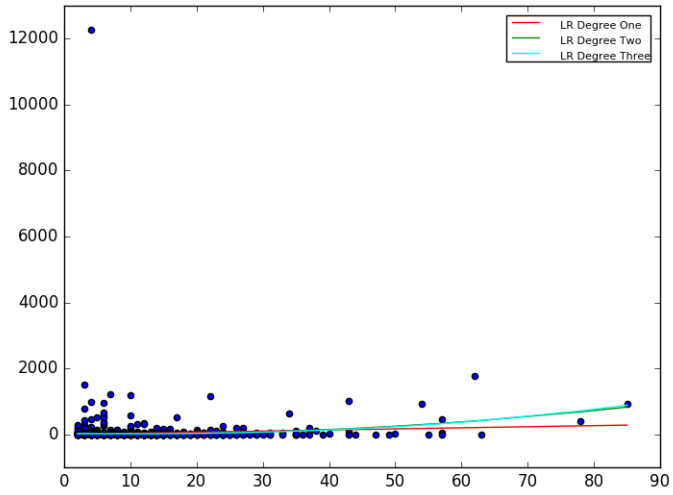
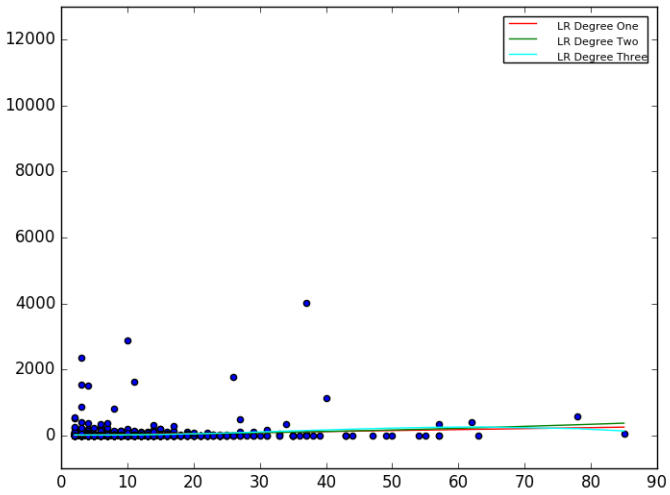



**Figure A.15:** Actual durations ($D_a$) of code reviews for JGit, sorted by actual effort ($E_a$)

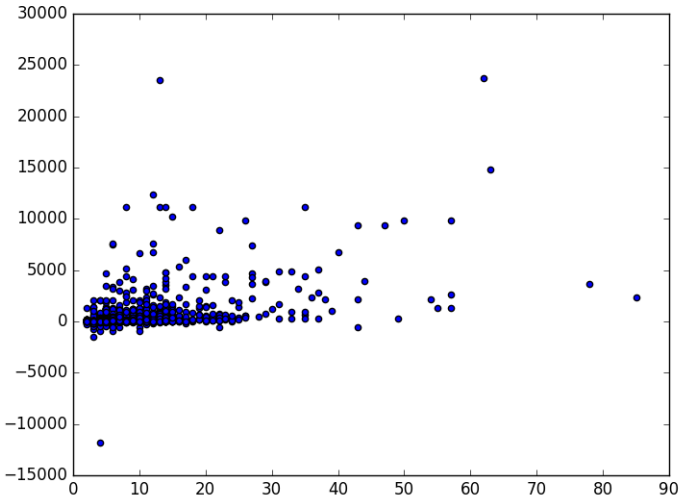



(a) Estimated Duration ($D_e$) computed using $R_{locMin}$ v. Effort ($E_a$) for `JGit`

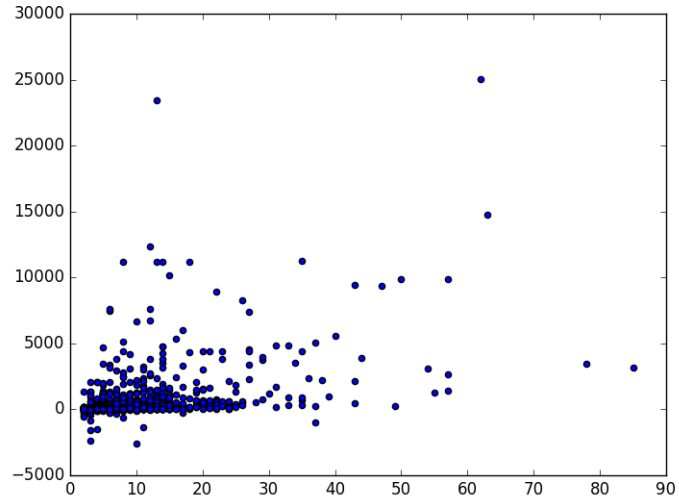



(b) Estimated Duration ($D_e$) computed using $R_{editMin}$ v. Effort ($E_a$) for `JGit`

**Figure A.16:** Scatter plots with regression lines of the estimated durations for `JGit` computed using the min algorithms

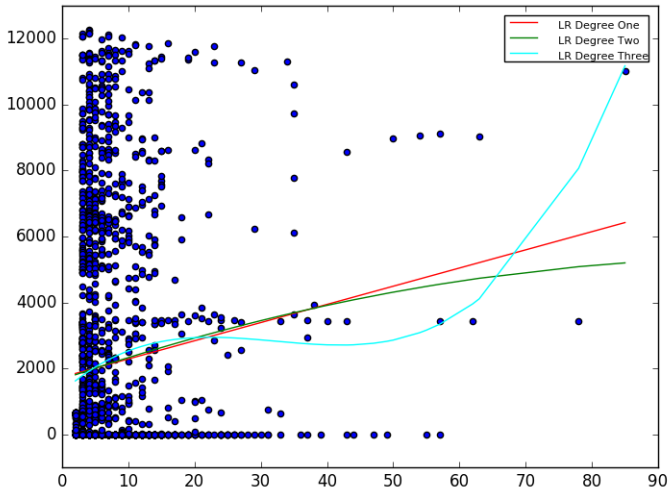



(a) Difference betweent the actual duration ($D_a$) and the estimated Duration ($D_e$) computed using $R_{locMin}$ v. Effort ($E_a$) for `JGit`

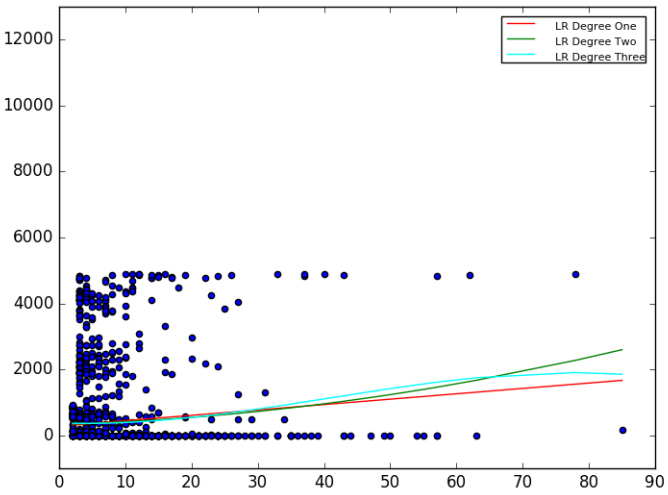



(b) Difference betweent the actual duration ($D_a$) and the estimated Duration ($D_e$) computed using $R_{editMin}$ v. Effort ($E_a$) for `JGit`

**Figure A.17:** Scatter plots with regression lines of the difference between the min estimates and the real durations for `JGit`

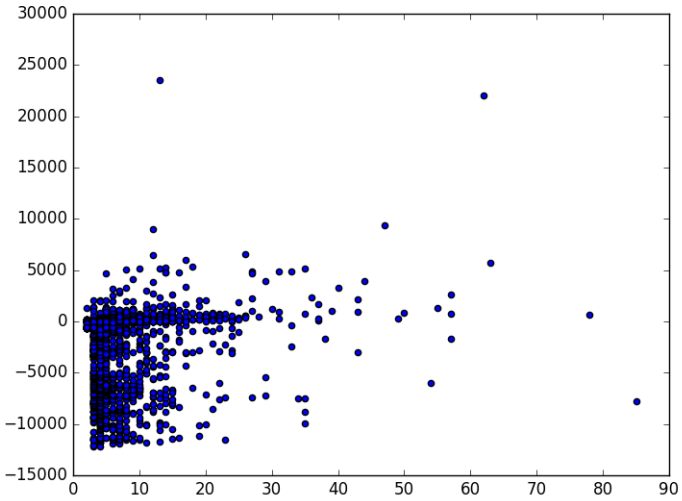



(a) Estimated Duration ($D_e$) computed using $R_{locMax}$ v. Effort ($E_a$) for `JGit`

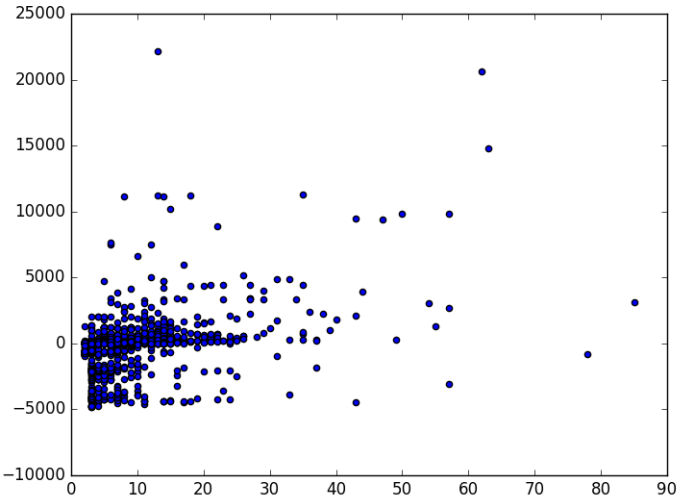



(b) Estimated Duration ($D_e$) computed using $R_{editMax}$ v. Effort ($E_a$) for `JGit`

**Figure A.18:** Scatter plots with regression lines of the estimated durations for `JGit` computed using the max algorithms

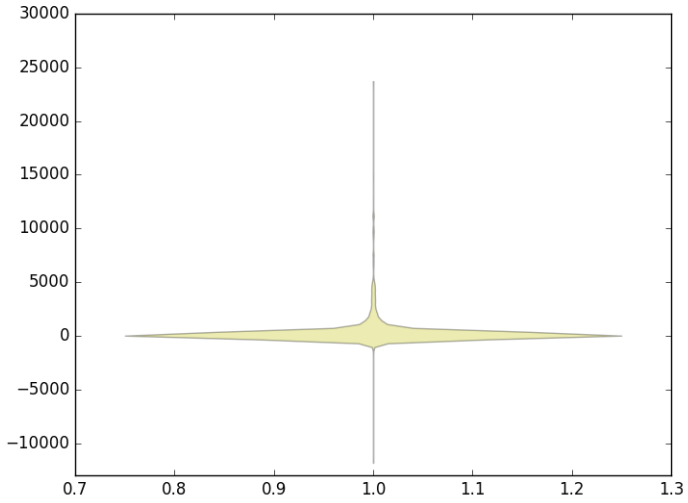



(a) Difference betweent the actual duration ($D_a$) and the estimated Duration ($D_e$) computed using $R_{locMax}$ v. Effort ($E_a$) for `JGit`

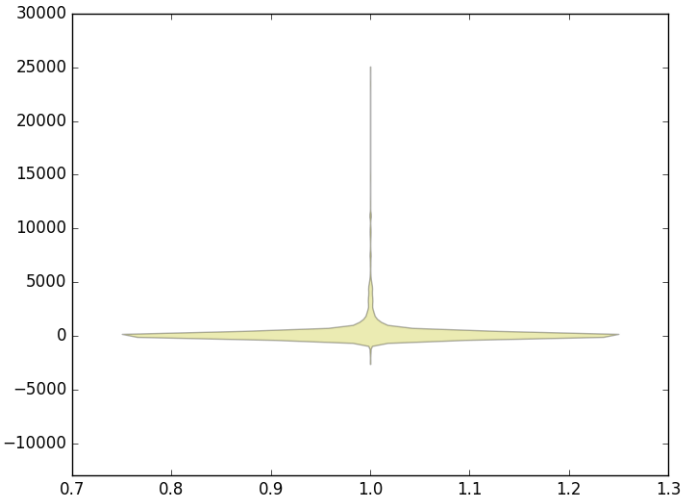



(b) Difference betweent the actual duration ($D_a$) and the estimated Duration ($D_e$) computed using $R_{editMax}$ v. Effort ($E_a$) for `JGit`

**Figure A.19:** Scatter plots with regression lines of the difference between the max estimates and the real durations for `JGit`

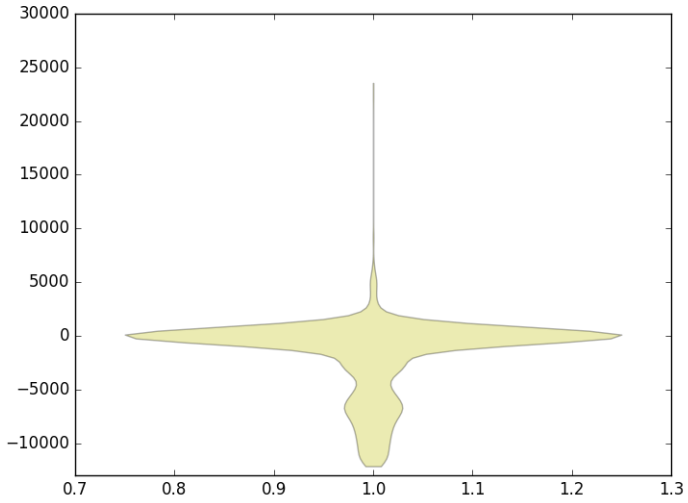



(a) Difference between actual duration ($D_a$) and estimated Duration ($D_e$) computed using $R_{locMin}$ for `JGit`

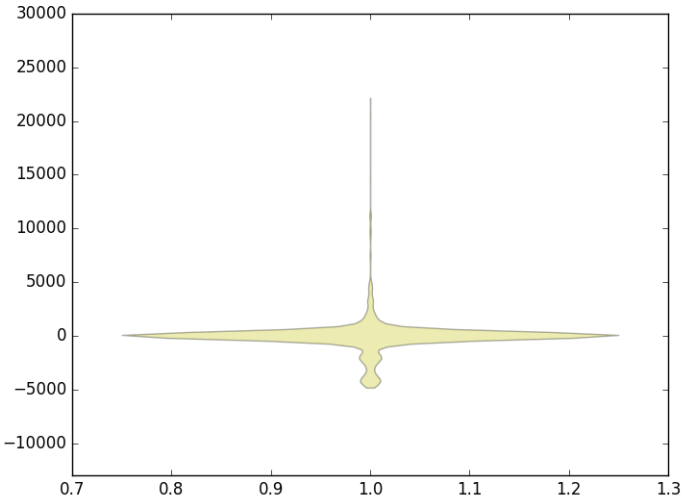



(b) Difference between actual duration ($D_a$) and estimated Duration ($D_e$) computed using $R_{editMin}$ for `JGit`

**Figure A.20:** Violin plots of the difference between the estimates and the real durations for `JGit` for the min algorithms




(a) Difference between actual duration ($D_a$) and estimated Duration ($D_e$) computed using $R_{locMax}$ for `JGit`




(b) Difference between actual duration ($D_a$) and estimated Duration ($D_e$) computed using $R_{editMax}$ for `JGit`

**Figure A.21:** Violin plots of the difference between the estimates and the real durations for `JGit` for the max algorithms

## A.4 Sirius

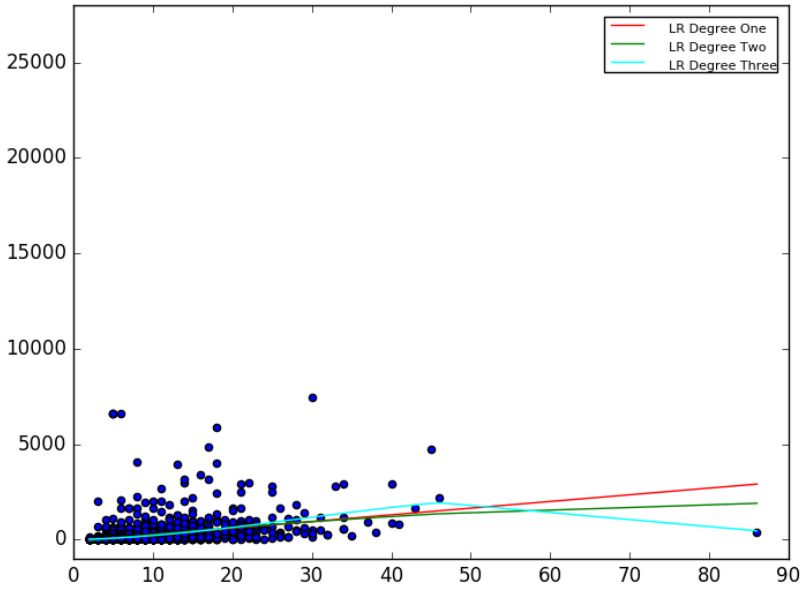
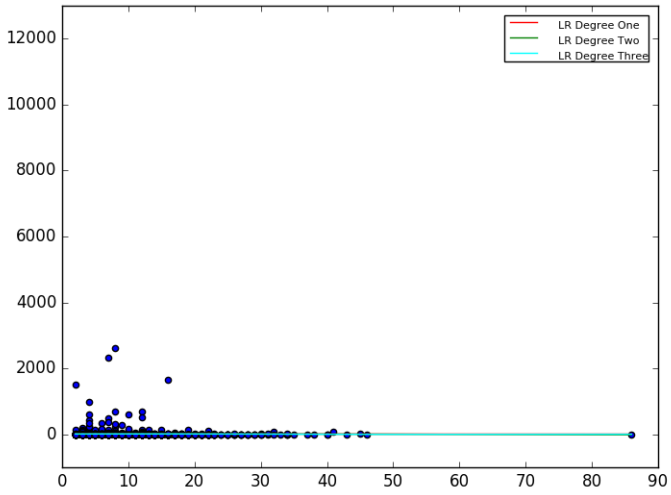
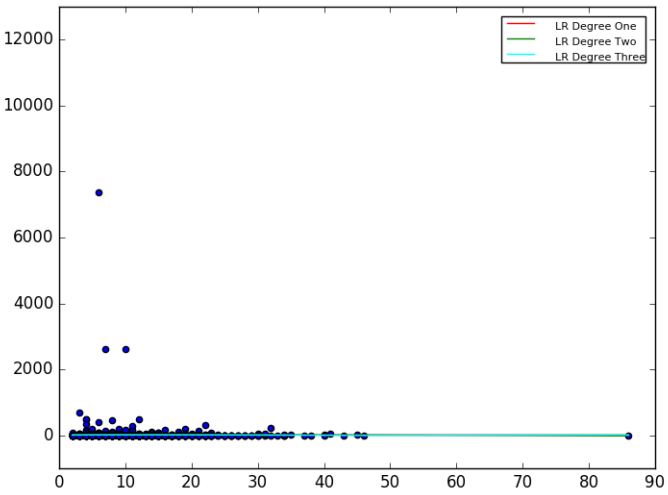



**Figure A.22:** Actual durations ($D_a$) of code reviews for `Sirius`, sorted by actual effort ($E_a$)

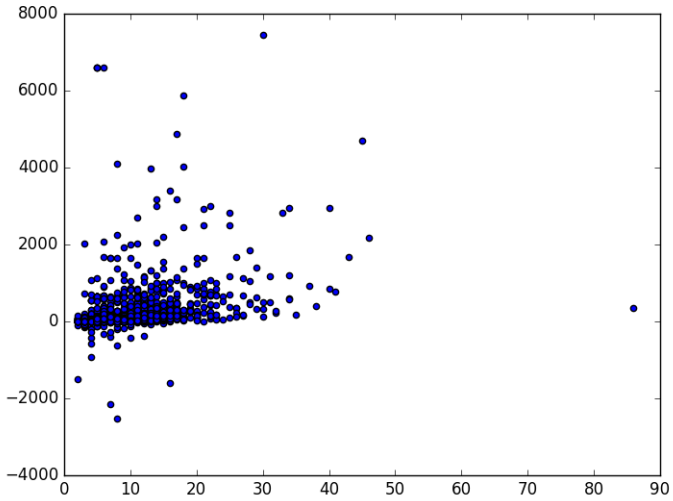



(a) Estimated Duration ($D_e$) computed using $R_{locMin}$ v. Effort ($E_a$) for `Sirius`

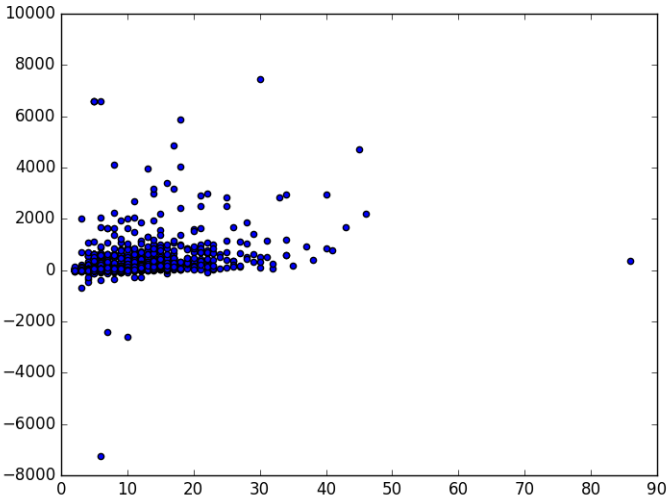



(b) Estimated Duration ($D_e$) computed using $R_{editMin}$ v. Effort ($E_a$) for `Sirius`

**Figure A.23:** Scatter plots with regression lines of the estimated durations for `Sirius` computed using the min algorithms

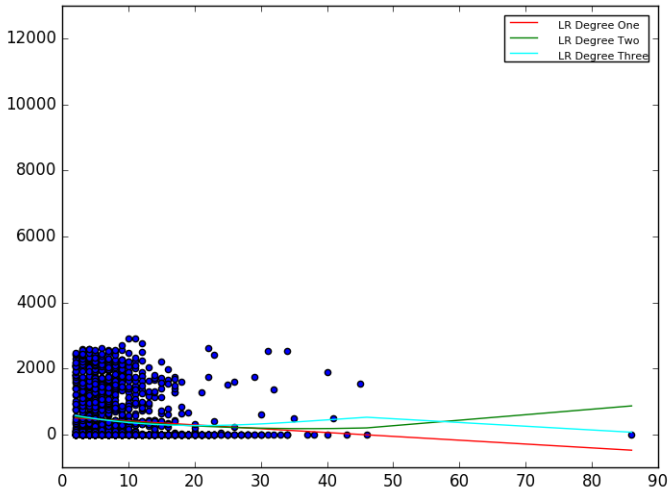



(a) Difference betweent the actual duration ($D_a$) and the estimated Duration ($D_e$) computed using $R_{locMin}$ v. Effort ($E_a$) for `Sirius`

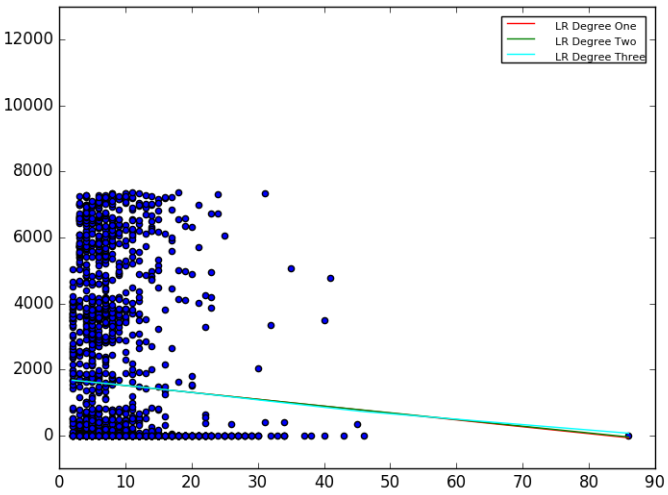



(b) Difference betweent the actual duration ($D_a$) and the estimated Duration ($D_e$) computed using $R_{editMin}$ v. Effort ($E_a$) for `Sirius`

**Figure A.24:** Scatter plots with regression lines of the difference between the min estimates and the real durations for `Sirius`

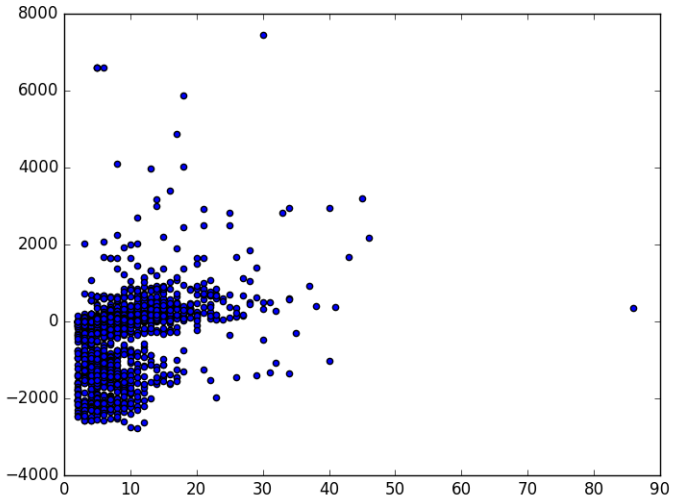



(a) Estimated Duration ($D_e$) computed using $R_{locMax}$ v. Effort ($E_a$) for `Sirius`

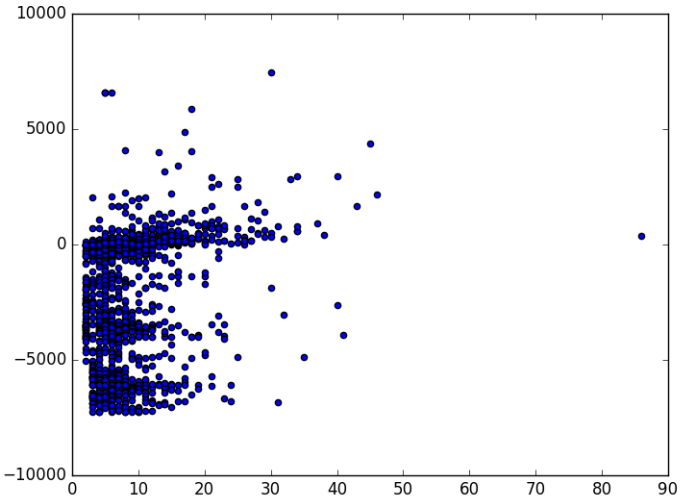



(b) Estimated Duration ($D_e$) computed using $R_{editMax}$ v. Effort ($E_a$) for `Sirius`

**Figure A.25:** Scatter plots with regression lines of the estimated durations for `Sirius` computed using the max algorithms

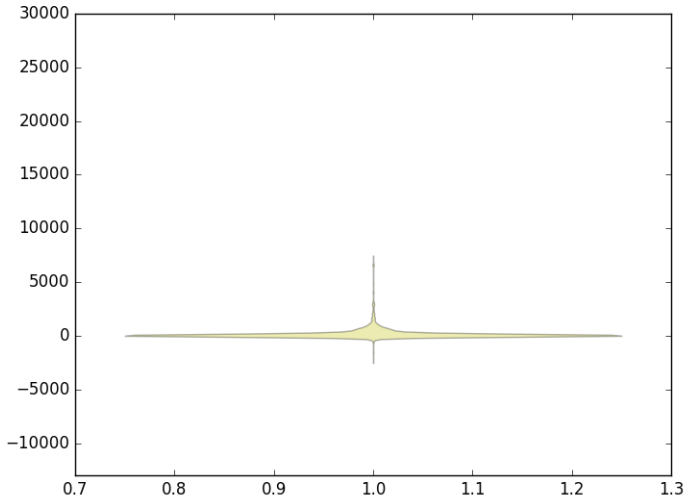



(a) Difference betweent the actual duration ($D_a$) and the estimated Duration ($D_e$) computed using $R_{locMax}$ v. Effort ($E_a$) for `Sirius`

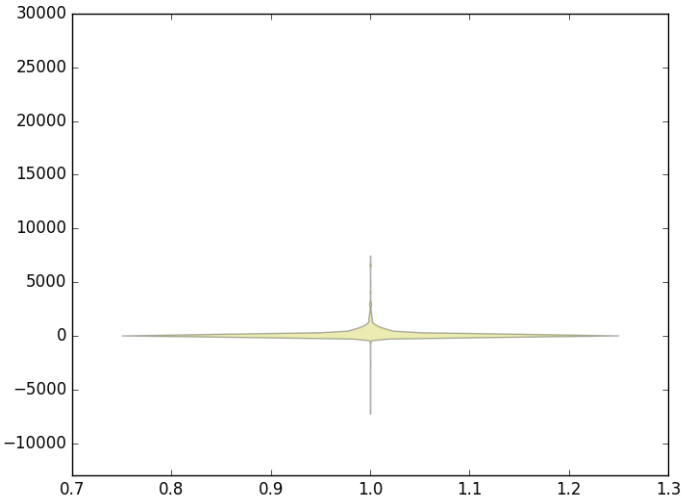



(b) Difference betweent the actual duration ($D_a$) and the estimated Duration ($D_e$) computed using $R_{editMax}$ v. Effort ($E_a$) for `Sirius`

**Figure A.26:** Scatter plots with regression lines of the difference between the max estimates and the real durations for `Sirius`

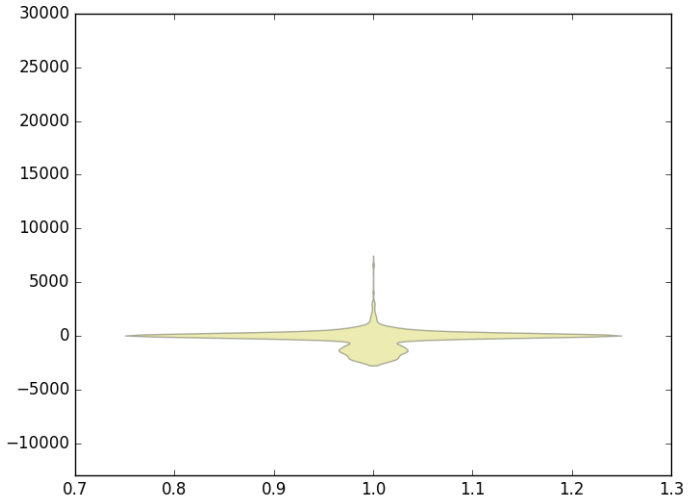



(a) Difference between actual duration ($D_a$) and estimated Duration ($D_e$) computed using $R_{locMin}$ for `Sirius`

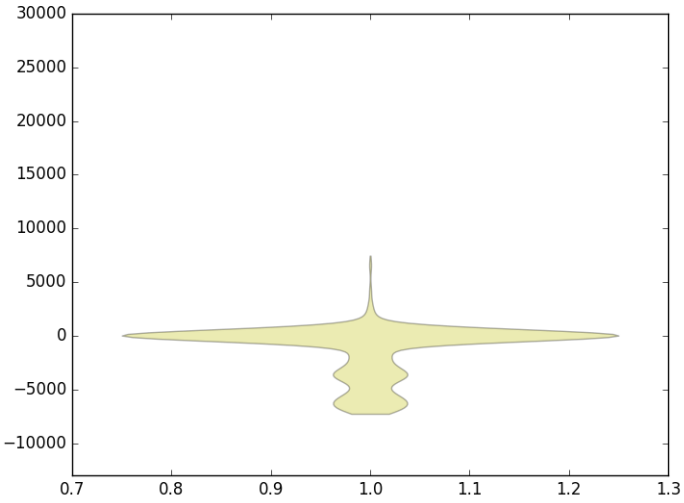



(b) Difference between actual duration ($D_a$) and estimated Duration ($D_e$) computed using $R_{editMin}$ for `Sirius`

**Figure A.27:** Violin plots of the difference between the estimates and the real durations for `Sirius` for the min algorithms




(a) Difference between actual duration ($D_a$) and estimated Duration ($D_e$) computed using $R_{locMax}$ for `Sirius`




(b) Difference between actual duration ($D_a$) and estimated Duration ($D_e$) computed using $R_{editMax}$ for `Sirius`

**Figure A.28:** Violin plots of the difference between the estimates and the real durations for `Sirius` for the max algorithms

## A.5 Osee

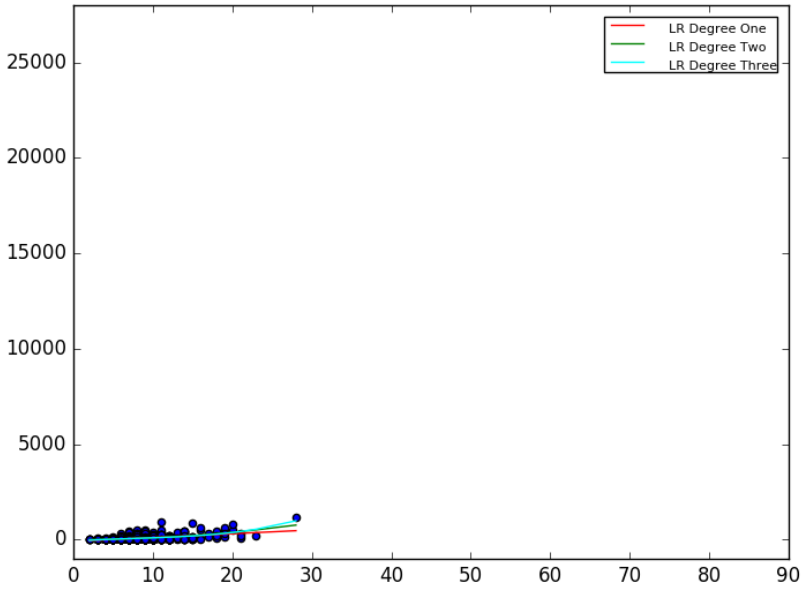
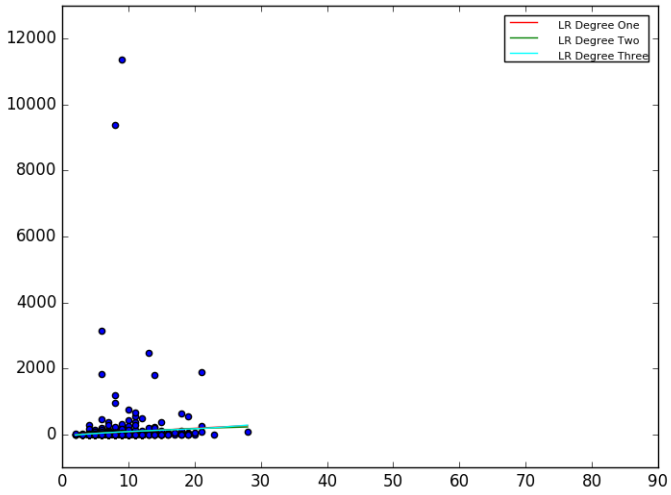
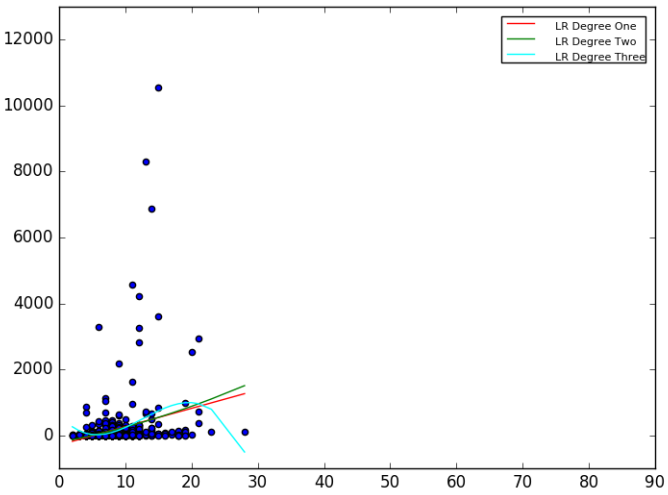



**Figure A.29:** Actual durations ($D_a$) of code reviews for Osee, sorted by actual effort ($E_a$)

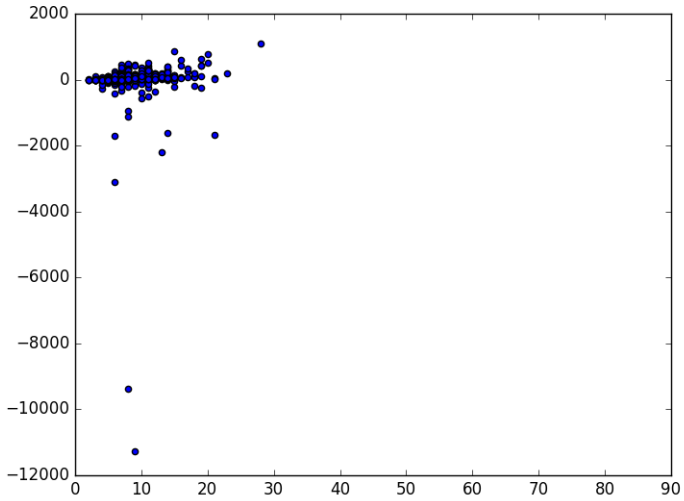



(a) Estimated Duration ($D_e$) computed using $R_{locMin}$ v. Effort ($E_a$) for `Osee`

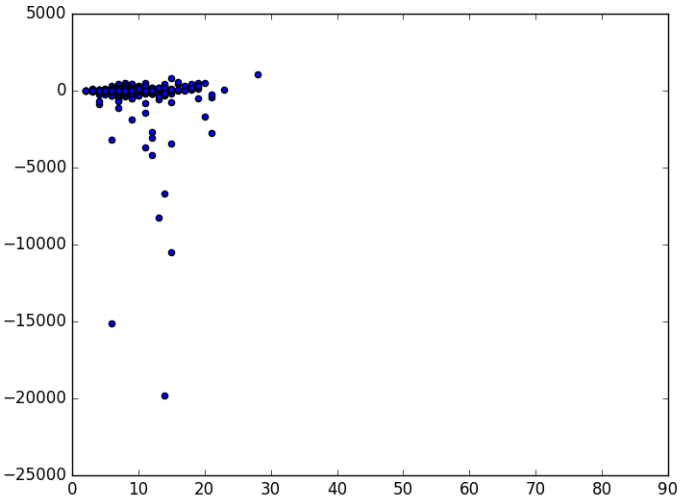



(b) Estimated Duration ($D_e$) computed using $R_{editMin}$ v. Effort ($E_a$) for `Osee`

**Figure A.30:** Scatter plots with regression lines of the estimated durations for `Osee` computed using the min algorithms

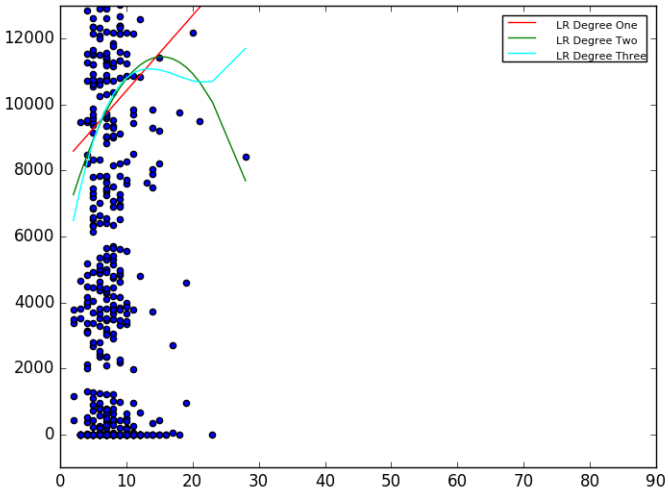



(a) Difference betweent the actual duration ($D_a$) and the estimated Duration ($D_e$) computed using $R_{locMin}$ v. Effort ($E_a$) for `Osee`

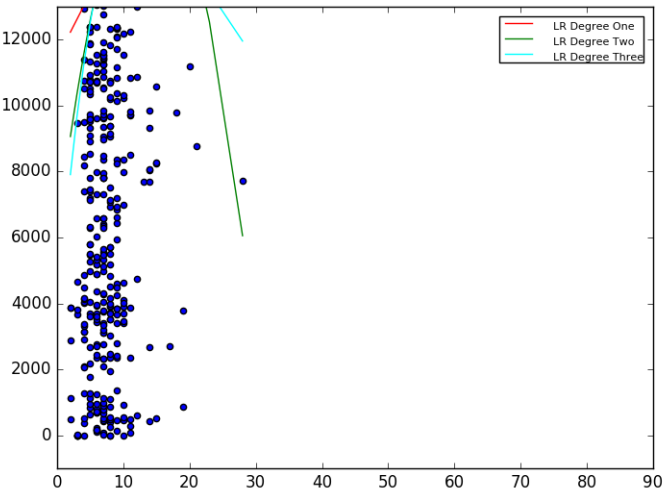



(b) Difference betweent the actual duration ($D_a$) and the estimated Duration ($D_e$) computed using $R_{editMin}$ v. Effort ($E_a$) for `Osee`

**Figure A.31:** Scatter plots with regression lines of the difference between the min estimates and the real durations for `Osee`

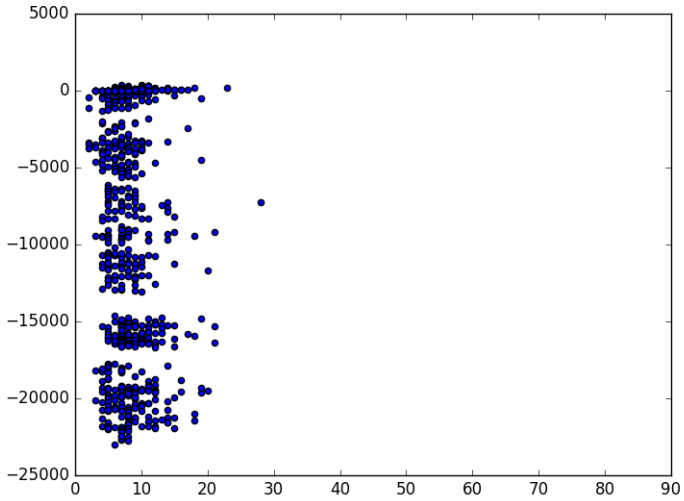



(a) Estimated Duration ($D_e$) computed using $R_{locMax}$ v. Effort ($E_a$) for `Osee`

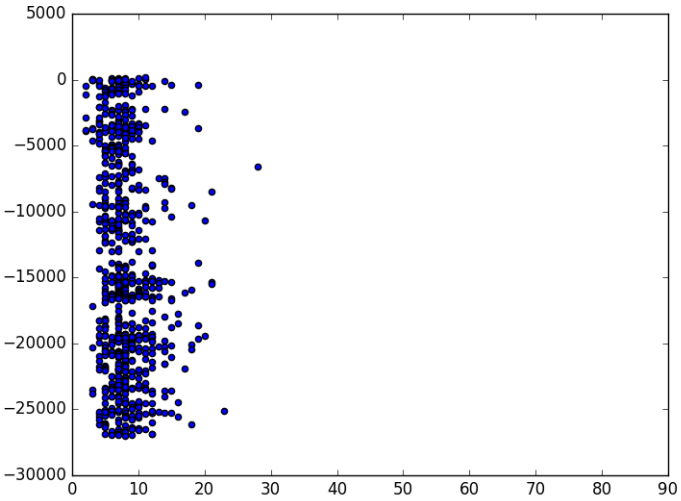



(b) Estimated Duration ($D_e$) computed using $R_{editMax}$ v. Effort ($E_a$) for `Osee`

**Figure A.32:** Scatter plots with regression lines of the estimated durations for `Osee` computed using the max algorithms

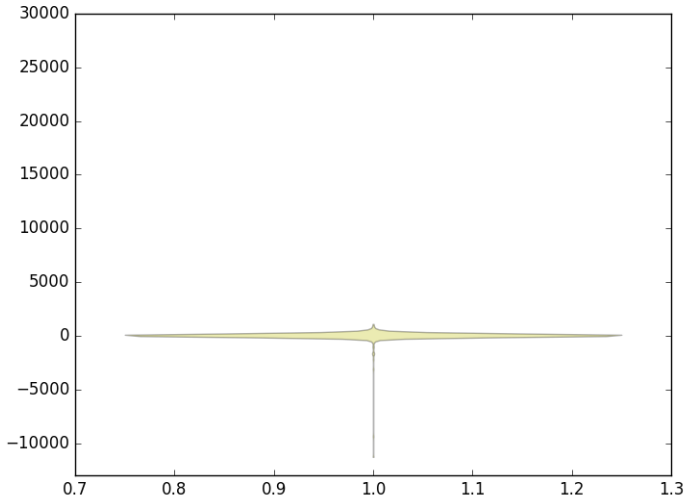



(a) Difference betweent the actual duration ($D_a$) and the estimated Duration ($D_e$) computed using $R_{locMax}$ v. Effort ($E_a$) for `Osee`

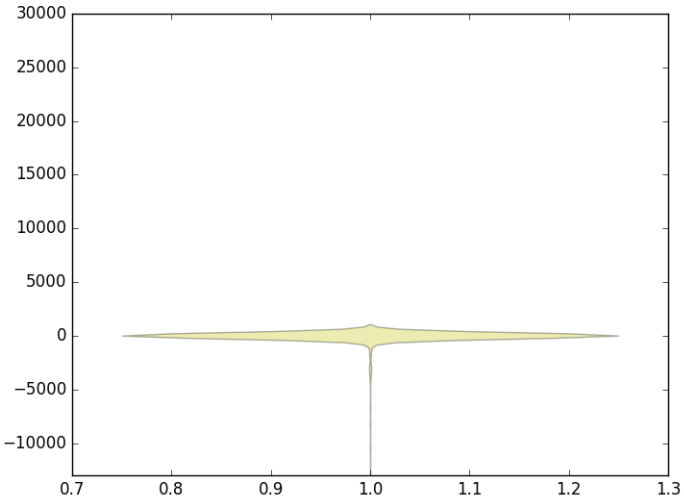



(b) Difference betweent the actual duration ($D_a$) and the estimated Duration ($D_e$) computed using $R_{editMax}$ v. Effort ($E_a$) for `Osee`

**Figure A.33:** Scatter plots with regression lines of the difference between the max estimates and the real durations for `Osee`

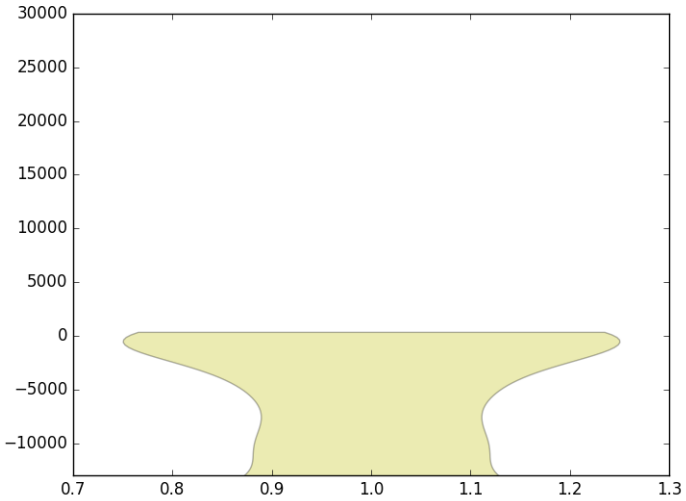



(a) Difference between actual duration ($D_a$) and estimated Duration ($D_e$) computed using $R_{locMin}$ for `Osee`

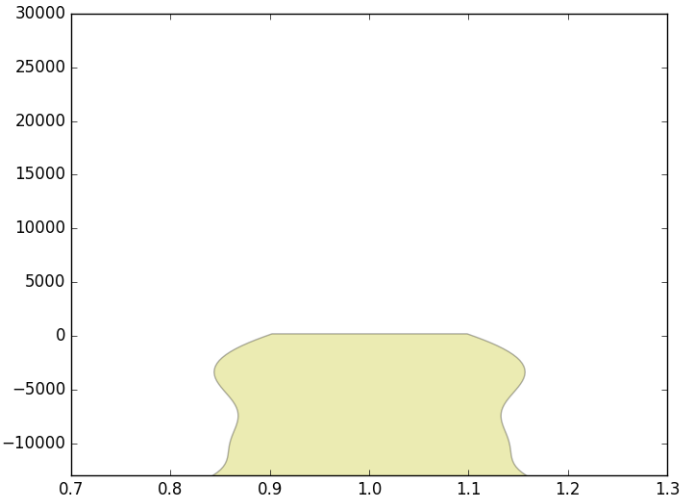



(b) Difference between actual duration ($D_a$) and estimated Duration ($D_e$) computed using $R_{editMin}$ for `Osee`

**Figure A.34:** Violin plots of the difference between the estimates and the real durations for `Osee` for the min algorithms




(a) Difference between actual duration ($D_a$) and estimated Duration ($D_e$) computed using $R_{locMax}$ for `Osee`




(b) Difference between actual duration ($D_a$) and estimated Duration ($D_e$) computed using $R_{editMax}$ for `Osee`

**Figure A.35:** Violin plots of the difference between the estimates and the real durations for `Osee` for the max algorithms

## A.6 Tracecompass

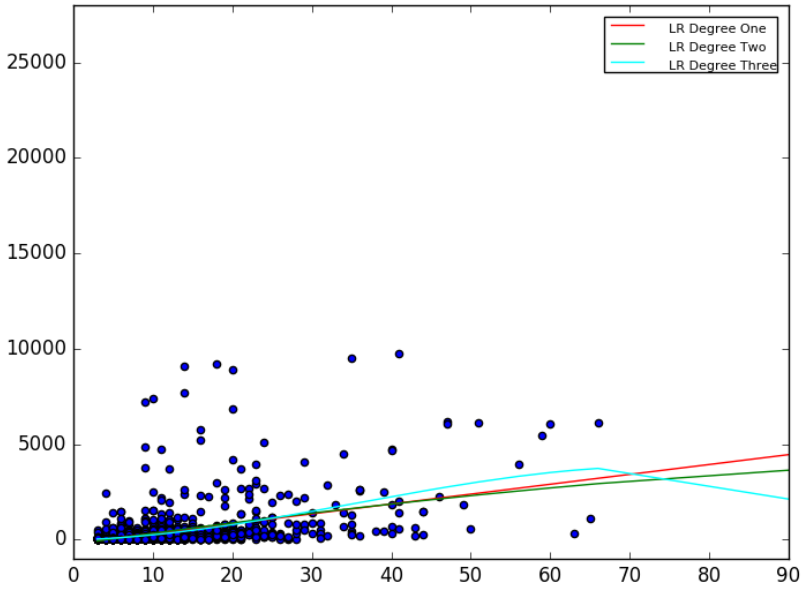
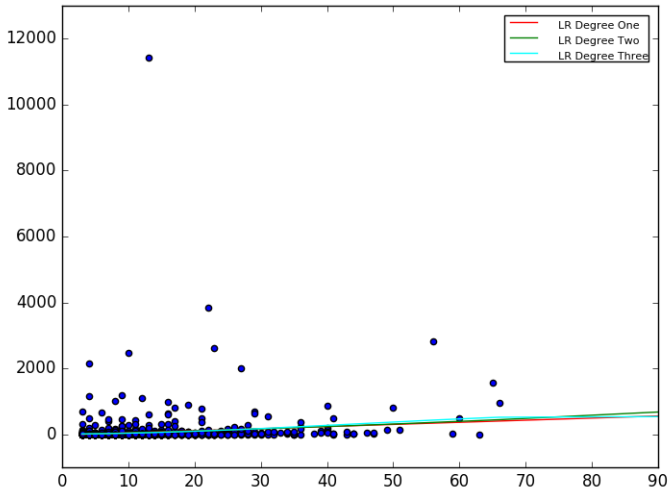
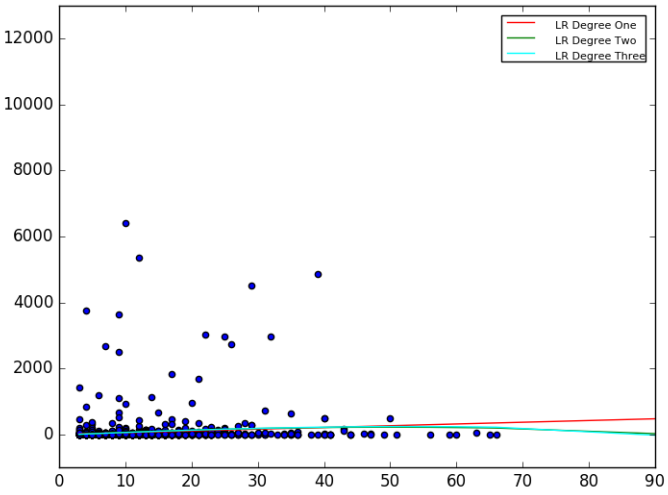



**Figure A.36:** Actual durations ($D_a$) of code reviews for `Tracecompass`, sorted by actual effort ($E_a$)

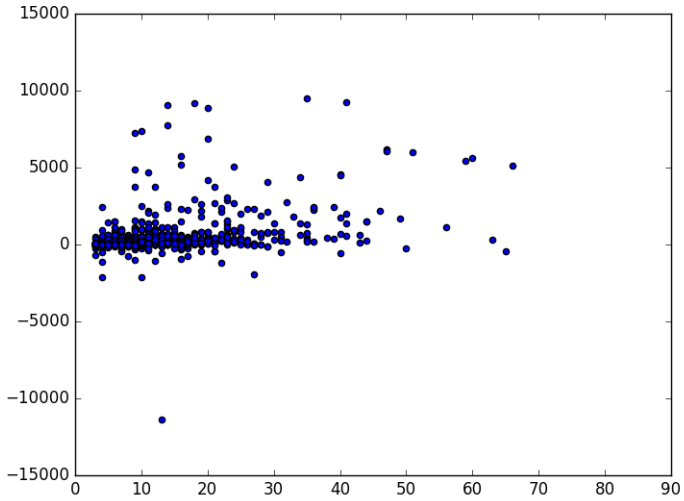



(a) Estimated Duration ($D_e$) computed using $R_{locMin}$ v. Effort ($E_a$) for `Tracecompass`

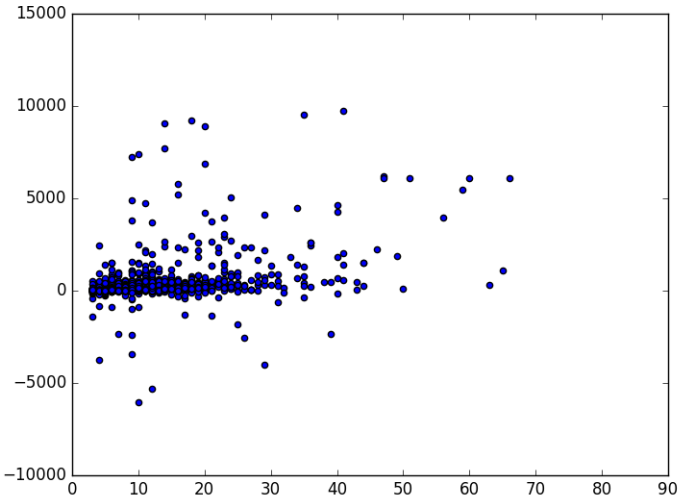



(b) Estimated Duration ($D_e$) computed using $R_{editMin}$ v. Effort ($E_a$) for `Tracecompass`

**Figure A.37:** Scatter plots with regression lines of the estimated durations for `Tracecompass` computed using the min algorithms

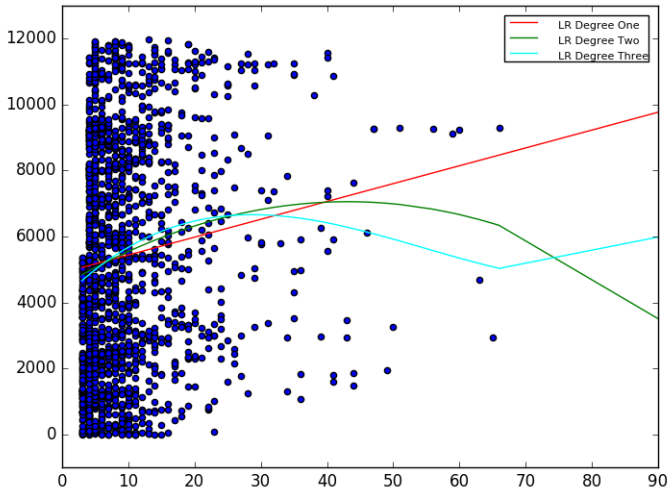



(a) Difference betweent the actual duration ($D_a$) and the estimated Duration ($D_e$) computed using $R_{locMin}$ v. Effort ($E_a$) for `Tracecompass`

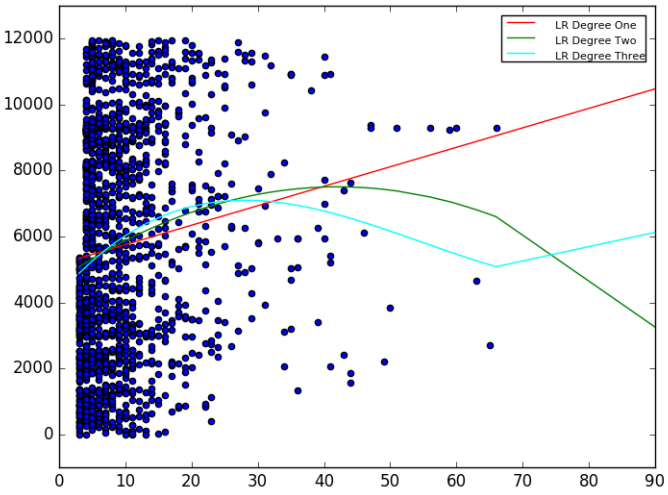



(b) Difference betweent the actual duration ($D_a$) and the estimated Duration ($D_e$) computed using $R_{editMin}$ v. Effort ($E_a$) for `Tracecompass`

**Figure A.38:** Scatter plots with regression lines of the difference between the min estimates and the real durations for `Tracecompass`

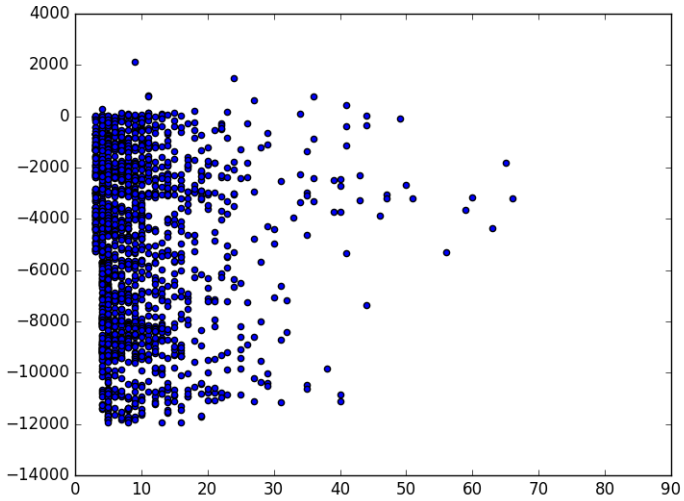



(a) Estimated Duration ($D_e$) computed using $R_{locMax}$ v. Effort ($E_a$) for `Tracecompass`

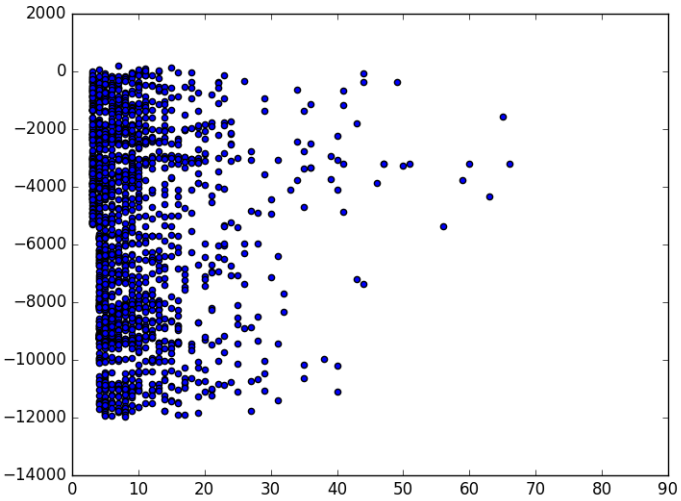



(b) Estimated Duration ($D_e$) computed using $R_{editMax}$ v. Effort ($E_a$) for `Tracecompass`

**Figure A.39:** Scatter plots with regression lines of the estimated durations for `Tracecompass` computed using the max algorithms

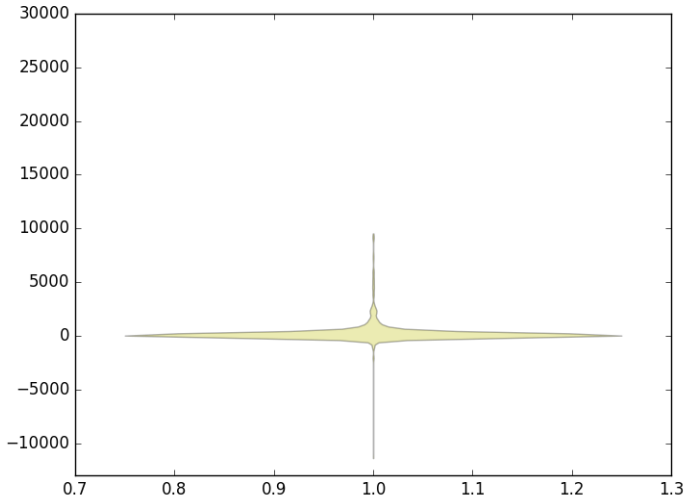



(a) Difference betweent the actual duration ($D_a$) and the estimated Duration ($D_e$) computed using $R_{locMax}$ v. Effort ($E_a$) for `Tracecompass`

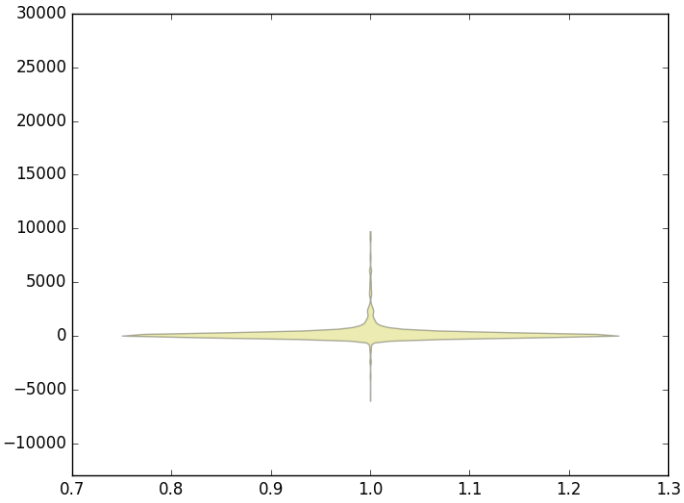



(b) Difference betweent the actual duration ($D_a$) and the estimated Duration ($D_e$) computed using $R_{editMax}$ v. Effort ($E_a$) for `Tracecompass`

**Figure A.40:** Scatter plots with regression lines of the difference between the max estimates and the real durations for `Tracecompass`

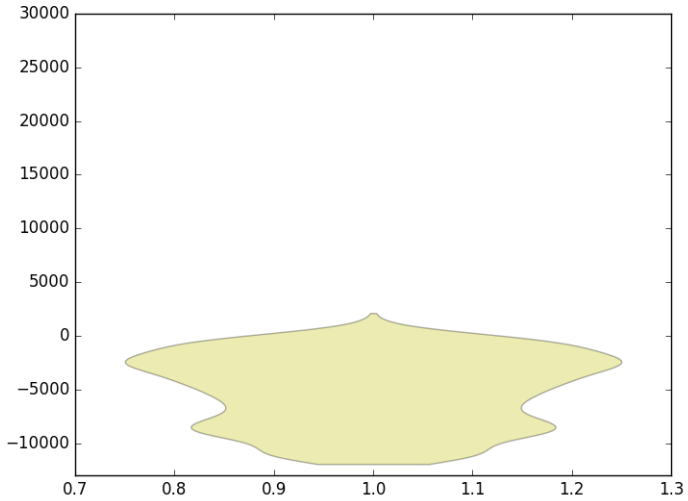



(a) Difference between actual duration ($D_a$) and estimated Duration ($D_e$) computed using $R_{locMin}$ for `Tracecompass`

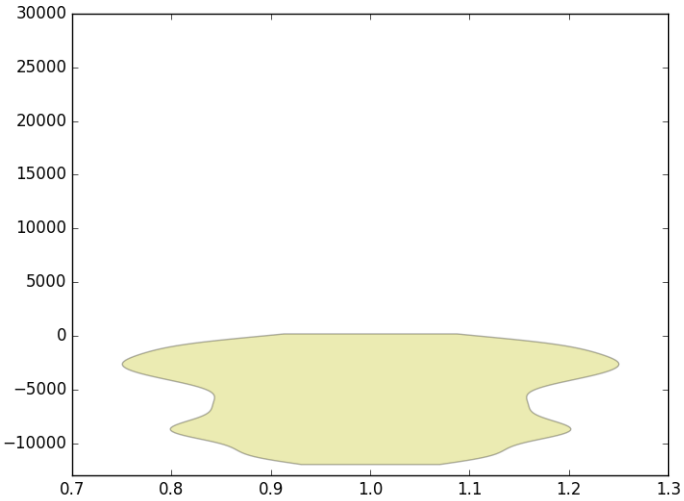



(b) Difference between actual duration ($D_a$) and estimated Duration ($D_e$) computed using $R_{editMin}$ for `Tracecompass`

**Figure A.41:** Violin plots of the difference between the estimates and the real durations for `Tracecompass` for the min algorithms




(a) Difference between actual duration ($D_a$) and estimated Duration ($D_e$) computed using $R_{locMax}$ for `Tracecompass`




(b) Difference between actual duration ($D_a$) and estimated Duration ($D_e$) computed using $R_{editMax}$ for `Tracecompass`

**Figure A.42:** Violin plots of the difference between the estimates and the real durations for `Tracecompass` for the max algorithms