TUHH
Hamburg University of Technology

Bachelor Thesis

Maximilian Scholz

A Line Based Approach for Bugspots

October 4, 2016

supervised by:
Prof. Dr. Sibylle Schupp

Hamburg University of Technology (TUHH)
Technische Universität Hamburg-Harburg
Institute for Software Systems
21073 Hamburg

STS
Software
Technology
Systems

Abstract

Code review is an important aspect of modern software development but time consuming. In 2011, Google proposed the Bugspots algorithm to help reviewers to focus on files that are more bug prone. The algorithm ranks the files based on the number of bug-fixes they received in the past, weighted by the age of the corresponding commit. A higher score equals more bug-fixes in recent times and indicates that there will be more bugs in that file.

In this thesis we propose Linespots, a modified version of the Bugspots algorithm that ranks the individual lines of code instead of whole files. Linespots gathers information by scoring every line involved in a bug-fix commit. Using these scores, Linespots can either give a list of ranked lines as a result or instead project the individual line scores back to file scores, offering the same result format as Bugspots this way.

An evaluation process was setup, comparing both kinds of Linespots' results to the results of Bugspots using hit density and the area under the hit density curve (AUCEC) as metrics. Both were proposed by Rahman et al. [5] in their work that served as a foundation for Bugspots.

The evaluation finds that the projected results are less consistent than the original Bugspots algorithm and do not improve the hit density or AUCEC. The line-based results have worse AUCEC values by design but could improve the hit density across all tested projects and most parameter configurations.

The written code is maintained openly and can be inspected in the repository.[1]

[1] https://gitlab.com/sims1253/Linespots

Acknowledgements

I hereby want to thank Lasse Schuirmann, Tanya Braun, Hauke Nuestedt, Sebastian Schlaadt, Frauke Gassmann-Scholz, Stephanie Kitzing and Britta Hauk for taking the time to review this thesis and providing helpful feedback.

Special thanks go to my supervisor Prof. Sibylle Schupp for supporting and supervising my thesis.

Contents

List of Tables

List of Figures

1. Introduction

In modern software development, code review is an integral part of the workflow but time consuming. One common scenario involves the following question:

- Which files should be reviewed if there is not enough time to cover all files?
- Can reviewing fault prone files twice be worth ignoring more stable files, if the time suffices to review everything?

One way to improve the ratio of bugs found per line of code reviewed is to prioritize files that are more fault prone. There are different algorithms to guess which files contain more bugs than others, varying in precision and cost. Rahman et al. found that simply ranking the files by the number of bug-fixes they received in the past is almost as precise as the more expensive BugCache algorithm [5]. Using the simple algorithm as a foundation, Google proposed an algorithm called Bugspots, which weighs the fixes by their age, so that older fixes are less relevant [3].

This thesis uses Bugspots as a foundation to propose a modified version, named Linespots. The aim is to determine whether a line-based approach for the Bugspots algorithm can yield better results than the file-based approach Bugspots uses. The quality of the results will be measured using the hit density metric proposed by Nachiappan et al. [4] and the cost effectiveness metric proposed by Arisholm et al. [1]. The focus lies on the hit density as it reflects the challenge of code review the most, which is finding the most bugs in the given lines of code.

Research Questions:

1. Can the line-based approach improve the hit density of the Bugspots algorithm?
2. How does a line-based approach effect the area under the cost effectiveness curve?

To reach the stated goal and answer the research questions, the following steps were followed:

1. Implement a reference Bugspots algorithm with optional file change tracking.
2. Implement a modified Bugspots algorithm with the proposed changes.
3. Implement an evaluation procedure using the hit density and the cost effectiveness.
4. Use the evaluation suite to compare the implementations.

The following chapters show the design of the modifications and the results of the evaluation.

Code samples given in this thesis follow the python syntax.

2. Fundamentals

This chapter explains the basic terms used in the thesis. It offers a short introduction to the version control software Git, a description of what code reviews and audits are, as well as a reference Bugspots algorithm, which serves as a baseline for the proposed changes and evaluation.

2.1. Git Commits

Git[1] is a version control software which stores the history of a project. A project's history can be described with a series of commits, which are changes made to the project. Starting from the initial empty state, every state of a project can be reached by applying all commits up to that moment, as long as a commit exists for that exact point in the history.

A commit consists of a head and a body:

```
commit bd3798f6804acadf8847eb7eb5c371079b634797
Author: Maximilian Scholz <m0hawk@gmx.de>
Date:   Sat Sep 3 19:25:05 2016 +0200

    example.py: Fix divide by zero bug

diff --git a/example.py b/example.py
index f0c41e9..093b844 100644
--- a/example.py
+++ b/example.py
@@ -1,3 +1,4 @@

 def mydivision(a, b)
-    return a/b
+    if b != 0:
+        return a/b
```

The head is the first part. It holds a hash that identifies the commit, the author of the changes, a date and a commit message, which serves as a note, tagged on to the change to explain what was done and why.

The body starts with the first `diff` statement and hold a section for each file consisting of hunks. Each hunk represents the changes applied to a continuous block of code. Each file starts with information about file level changes, i.e., renaming or deletion. Following are the hunks, each beginning with a line, holding positional information, then the changes

[1] https://git-scm.com/

made in the commit follow. Lines with no prefix remain unchanged. A minus indicates a deleted line and a plus a newly inserted line. Changing parts of a line also yields a removed and newly inserted line.

2.2. Reviews and Audits

In software engineering, reviewing and auditing code before it is released into production are common tasks. During a review or audit, the code is inspected to find bugs and improve readability and to increase maintainability.

Although not strictly defined, the term review commonly describes the inspection of a proposed change to a code base. Other developers, usually team members, inspect the proposed changes to find bugs, create easier solutions for the problem and, help the team understanding the change. After everyone agrees that the proposed changes meet the set standards, they are applied to the code base.

The term audit describes an additional inspection of the code base, independent of the development process. A common use case for an audit is a so called security audit, in which a piece of software is examined to find security problems that could be exploited to get other users' information for example.

2.3. Bugspots

In 2011, Rahman et al. [5] used a simple algorithm to compare their newly proposed expensive algorithm against. They found that the simple algorithm could predict future bugs with similar precision to their much more complex algorithm by ranking the files based on the number of bug-fix commits they received in the past. The idea behind the simple algorithm is that if a higher number of bugs occurred in a file, this file must be complex. Thus, the probability rises that more bugs are present or introduced frequently to this file.

Later that year Google, used this idea as a basis for Bugspots [3]. Instead of ranking files based on the amount of fixes they received, Google proposed weighting the fixes based on the time they occurred. If a fix becomes older, it has less influence on the score. The weighting helped to move files that once where bug prone, but got fixed out of the top of the ranking.

Google declares the top 10% of ranked files as hot-spots. During reviews and audits the developers can use this information to spend more time on them.

2.3.1. Algorithm

The algorithm looks through a list of commits and for every commit it checks if it is a bug-fix or not, which is not easy and an own research problem. If so, it then proceeds to increase the involved files' scores depending on the commit's age. Finally it returns a list of all files ranked based on the score they received:

```
for commit in commits_to_crawl:
    if commit_is_fix(commit):
        for file in commit.diff:
            file.score +=
                1 / (1 + math.exp(
                    (-12 * normalized_timestamp(commit)) + 12))
```

2.3.2. Finding Fix Commits

Google determines if a commit fixed a bug by checking the commit message for an attached bug and then using their bug-tracking database to decide if it was a bug or a feature request. Existing open source implementations of the Bugspots algorithm use a regular expression (regex) on the commit messages to find indicators for bug-fixes, which is what the algorithm proposed in this thesis also uses.

The regex `fix(ing|e[s|d])?|bug` can find the words fix, fixing, fixes, fixed and bug. Using it on `Bug 771131 - Replying in plain text ignores quotation level` to identify the fix commits would recognize `Bug` and identify the commit message as a bug-fix.

To ensure the best precision in finding bugs, the regex has to be chosen cautiously to fit the project-specific commit message guidelines.

2.3.3. Weighted Scoring

The reasoning behind the weighting of commits is that without it, a buggy file that gets fixed still retains a high score. To account for fixed files, older bug-fixes weigh less than newer ones. Google proposed the following function as a way to weigh commits by age.

$$Score = \sum_{i=0}^{n} \frac{1}{1 + e^{-12t_i+12}}$$

"Where n is the number of bug-fixing commits, and t_i is the timestamp of the bug-fixing commit represented by i. The timestamp used in the equation is normalized from 0 to 1, where 0 is the earliest point in the code base, and 1 is now (where now is when the algorithm was run). Note that the score changes over time with this algorithm due to

the moving normalization; it's not meant to provide some objective score, only provide a means of comparison between one file and another at any one point in time."[3]

In figure 2.1 the function is displayed for the normalized time between 0 (oldest commit) and 1 (newest commit). It shows how only the newest commits have a high weight and older commits become less important fast.

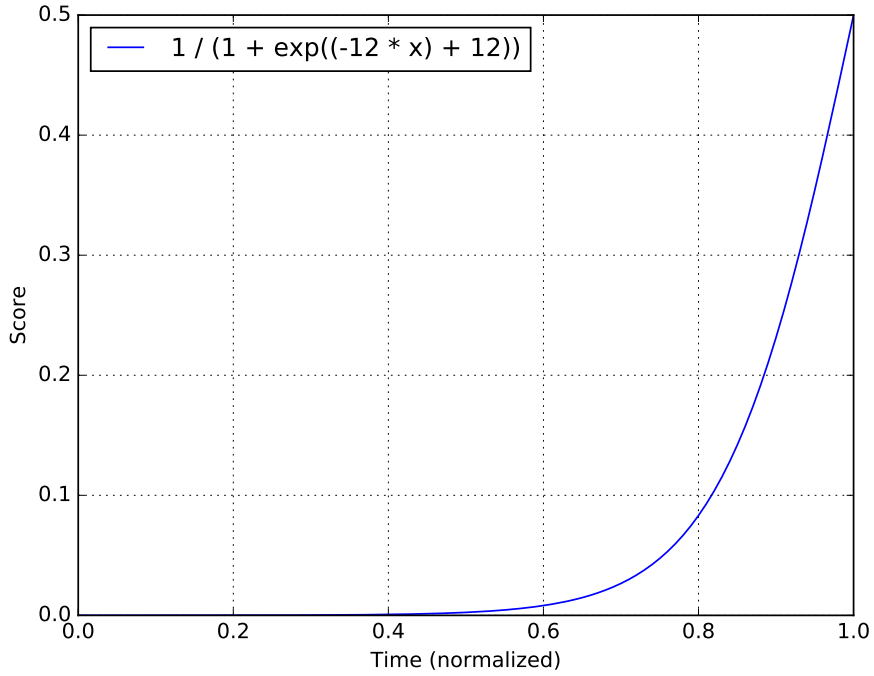


Figure 2.1.: Bugspots Weighting Function

3. Design

This chapter holds information about the design of the newly proposed Linespots algorithm. It describes the thought process and principles behind the single parts.

3.1. The Linespots Algorithm

The Bugspots algorithm by Google uses a single score per file. This should intuitively add a lot over overhead lines for every proposed file as the whole file has to be inspected even if just one line is fault prone. To reduce the number of inspected lines with low scores we propose to use one score per line of code instead. This should enable the algorithm to only propose the fault prone lines, leaving out the stable ones and thus increasing the number of bugs per proposed line. As result Linespots can either present a list of files as described in section 3.6 or a list of lines as described in section 3.7. An example of how the line-based result is displayed to the user is shown in figure 3.1. The overall structure of Linespots is the same as Bugspots, with the differences being in the ranking and tracking of files and lines.

```
for commit in commits_to_crawl:
    if commit_is_fix(commit):
        for file_diff in commit.diff:
            file.track(file_diff)
            for hunk in file_diff:
                file.lines.track(hunk)
                    file.lines.update_score(hunk, commit.date)
    else:
        for file_diff in commit.diff:
            file.track(file_diff)
            for hunk in file_diff:
                file.lines.track(hunk)
```

In the following sections, the separate components of the Linespots algorithm are described in more detail.

3.2. Recognizing Bug-Fix Commit

Rahman et al. use three different approaches to identifying bug-fixing commits. First, they search the commit message for keywords like ‘fix’ and ‘bug’ and second, they use a bug tracking database to find bugs that were fixed and commits that correspond to the fix. Last, they did a manual review of commits to remove false positives and find commits the previous methods missed.

```
[m0hawk@m0hawkPC coala]$ time python3 linespots.py -g "Fixes" -l 20 -s lines
################################################################
0.462670  coalib/misc/Compatibility.py line: 4
0.462670  coalib/misc/Compatibility.py line: 3
0.462670  coalib/misc/Compatibility.py line: 2
0.462670  coalib/misc/Compatibility.py line: 1
0.462670  coalib/misc/Compatibility.py line: 0
0.426189  tests/bearlib/spacing/SpacingHelperTest.py line: 89
0.426189  coalib/bearlib/spacing/SpacingHelper.py line: 12
0.426189  coalib/bearlib/spacing/SpacingHelper.py line: 10
0.426189  coalib/bearlib/spacing/SpacingHelper.py line: 9
0.426189  coalib/bearlib/spacing/SpacingHelper.py line: 8
0.426079  CONTRIBUTING.rst line: 53
0.426079  CONTRIBUTING.rst line: 15
0.303261  coalib/output/ConsoleInteraction.py line: 510
0.289872  coalib/coala_main.py line: 79
0.289021  tests/misc/CachingUtilitiesTest.py line: 71
0.289021  tests/misc/CachingUtilitiesTest.py line: 70
0.289021  tests/misc/CachingUtilitiesTest.py line: 69
0.289021  tests/misc/CachingUtilitiesTest.py line: 68
0.289021  tests/misc/CachingUtilitiesTest.py line: 67
0.289021  coalib/misc/CachingUtilities.py line: 149

real    0m0.361s
user    0m0.330s
sys     0m0.033s
```

Figure 3.1.: Line-Based Result for the coala Repository

This thesis only uses the approach of parsing the commit message with a regex. For each project an own regex is chosen and used to find project specific keywords in the commit messages. If the regex finds one of the given keywords, the commit is flagged as a fix commit.

Even though the use of bug tracker information and manual review of commits improves the precision of identifying the commits, those processes are not used in this thesis due to time constrains.

3.3. Tracking File Movement

The Google engineers did not address any implementation details so it is uncertain if they track file changes or not. The more popular implementations of the Bugspots algorithm do lack this feature though.[1][2]

Linespots uses file tracking as it preserves information for files that move or change their name. Without file tracking the information gathered before the change is lost. File level

[1] https://github.com/igrigorik/bugspots
[2] https://github.com/niedbalski/python-bugspots

changes can be gathered from the diff body. Every file section in a commit starts with information about the file level changes. If the commit added, renamed or deleted a file, the diff holds the old and new paths of this file. Linespots uses this information to move the line scores with a renamed file, delete them for a deleted file or initialize a new list for an added file.

3.4. Tracking Line Movement

As Git diffs only display line level changes as either unchanged, removed and added lines, all scores of changed lines could be lost with no proper line tracking.

For unchanged and deleted files the tracking is simple. If a line remains unchanged, Linespots does not change the score as well. For deleted lines, Linespots also deletes the scores. But for added lines it is hard to determine if a line is a modification of an old line or a newly added line. We make the assumption that every line that is part of a bug-fix commit was part of the bug inducing code and thus every line added with a bug-fix commit is treated as a modification. Every modified line needs to have a corresponding line-score from before the commit to serve as the basis before updating the scores in the next step. Identifying the lines from before the change is hard and an own research problem. Canfora et al. [2] found that similar location and immediate proximity are good indications for the identification of changed lines and their origin. As Git offers changes in hunks that hold the changed lines and minimal context overhead, we treat all lines in a hunk as being a common origin line for all lines added to that hunk with the commit. Lines added in a hunk use the average line-score of that hunk as a basis for the score update.

Suppose a hunk h has n lines before a commit. For all lines l with line-score $S(l)$ in a hunk, let l_a be the number of added lines, l_r be the number of removed lines, l_u be the number of unchanged lines, and $\sum S(l)$ be the sum of all line-scores for the lines l:

$$n = l_u + l_r$$

- If the line is untouched, the line-score $S(l)$ is updated
- If the line is deleted, the line-score $S(l)$ is deleted
- If the lines is added, first the line-score is determined by the average hunk line-score before the commit and then the line-score is updated:

$$S(l) = S_{\text{average}}(h) = \frac{1}{n} \cdot \left(\sum S(l_u) + \sum S(l_r) \right)$$

If there are no previous scores for the hunk because it is the first time the algorithm encounters this file, the line-score $S(l)$ is set to 0.

3.5. Updating Scores

After tracking the lines in a hunk, Linespots updates the remaining line-scores the same way Bugspots does, as described in section 2.3.3. Linespots also uses the same weighting function as Bugspots because, although using a different function or weighting process could yield better results, investigating the influence of the weighting function is not the focus of this thesis.

Calculating the normalized age of a commit uses the UNIX timestamps of the commits. Let $T(c_0)$ be the timestamp of the oldest commit to inspect, $T(c_m)$ be the timestamp of the most recent commit to inspect, then

$$T_n(c_i) = \frac{T(c_i) - T(c_0)}{T(c_m) - T(c_0)}$$

is the normalized timestamp of commit c_i between c_0 and c_m. This lets the normalized timestamp be 0 for c_0 and 1 for c_m.

We calculate the score of a line l after a commit c_i, $S(c_i, l)$, by adding the result of the weighting function at the normalized age of the commit, to the base score that is the result of the line tracking, $S(c_i - 1, l)$.

$$S(c_i, l) = S_l(c_i - 1, l) + \frac{1}{1 + e^{-12T_n(c_i)+12}}$$

For commits that don't fix bugs the update is not done.

3.6. File Level Results

The Bugspots algorithm from Google has one score per file. To be able to evaluate against it under similar conditions, the line scores of each file have to be combined to a file score.

We propose two algorithms for the projection of line-scores to a file score. One uses the average line-score for a file, the other uses the average line-score of the highest scored 10% lines per file. The next two sub sections hold details about the two algorithms.

3.6.1. Average Line-Score

The average line-score is a file score where we compute the average score over all lines in a file. Formally, let f be a file with n lines of code l_i, then:

$$S(f) = \frac{1}{n} \cdot \sum_{i=1}^{n} S(l_i)$$

The average line-score is a simple and thus easy to use approach but it may not yield a useful metric if large files that have a lot of untouched code and a small portion of highly bug prone code are part of bug-fixes. Those files could have a low score even if a significant amount of bug-fixes was applied to them.

3.6.2. Top Ten Percent Score

The top ten percent score is a file score where we compute the average score over the highest 10% line-scores in a file. Formally, let f be a file with n lines of code l_i sorted by their score $S(l_i)$ in descending order, then:

$$S(l_1) \geq S(l_2) \geq S(l_3) \geq \ldots \geq S(l_n)$$

$$S(f) = \frac{1}{n} \cdot \sum_{i=1}^{\lfloor \frac{n}{10} \rfloor} S(l_i)$$

To cover the weakness of the average line-score, the top ten percent score sorts the lines in each file based on their score, takes the highest scored ten percent lines, and calculates the average score of them. This way the focus lies on the more bug prone parts of files. The downside to this algorithm is that it does ignore the size of the file and could lead to big files being at the top of the ranking, even if there are smaller files with better bugs per line of code ratio.

3.7. Line Level Results

The second goal for this thesis is to use the line-based information gathered to further improve the review process. We propose simply ranking all lines by their score and to set a threshold score to mark all lines above the threshold as bug prone.

3.8. Summary of Changes

We proposed the following changes to the original Bugspots algorithm:

- File tracking: This topic is not addressed in the Google engineers' blog post. We assume, that Google uses something similar to the proposed tracking capabilities but some more popular implementations of the Bugspots algorithm lack this feature.
- Line tracking and scoring: The biggest point of interest for this thesis is to see the advantages of using one score per line instead of one score per file. Using line-scores requires tracking and scoring of individual lines. Linespots either can output the

sorted line-scores directly or project them to file-scores, similar to the output of Bugspots.

3.9. Implementation

Every part of the algorithm described in the current chapter is implemented in its own function, which improves maintainability and the ability to test and evaluate the single parts.

One limitation of the implementation is that it does not parse diffs of merge commits. As every merge consists of multiple diffs and every merge commit has multiple parent commits, parsing them is complicated and was not implemented due to time constrains.

The implementation is written in python 3 and focuses on the precision of results as well as usability and maintainability. Some time was spent optimising the runtime and memory usage to make evaluation with larger sets of parameters possible.

4. Evaluation

This chapter describes the evaluation process used to analyse the changes proposed in earlier chapters and displays the results.

For the evaluation, three open source projects were analysed using different versions of the algorithms:

1. The Bugspots algorithm proposed by Google without file tracking
2. A modified version of 1. that includes file tracking
3. The newly proposed Linespots algorithm using file-based results
4. The newly proposed Linespots algorithm using line-based results

4.1. Metrics

We used three metrics for the evaluation: Hit density, area under the cost effectiveness curve and area under the cost effectiveness curve at 20 % lines of code.

4.1.1. Area Under the Cost Effectiveness Curve

Rahman et al. [5] use the cost effectiveness curve (CEC), which can be obtained by plotting the proportion of identified bugs (a number between 0 and 1) against the proportion of the lines of code (also between 0 and 1) inspected.

Suppose bugs are randomly distributed in the code. An algorithm randomly inspecting files should find one percent of bugs per one percent of lines. A useful algorithm thus should have a CEC that stays above the $f(x) = y$ graph.

Figure 4.1 shows a zoomed in CEC in blue and a hit density curve in dashed red. It demonstrates how hit density and the CEC are related, as the hit density peaks when the CEC has the biggest lead relative to the $y = x$ line.

By using the area under the curve as a metric, an algorithm can be compared to another algorithm broadly, as a bigger area comes from a faster rising graph and thus has a higher hit density in general.

Because it is not realistic to review most of a project, Rahman et al. proposed the area from 0% to 20% lines of code as a metric. The area under the CEC from 0% loc to 20% loc is referred to as AUCEC_{20}. The AUCEC_{20} represents the performance of an algorithm inspecting up to 20% lines of code.

Google uses the 10% highest rated files as hot spots, which follows the finding of Rahman et al. that 10% of top ranked files account for roughly 20% of the code.

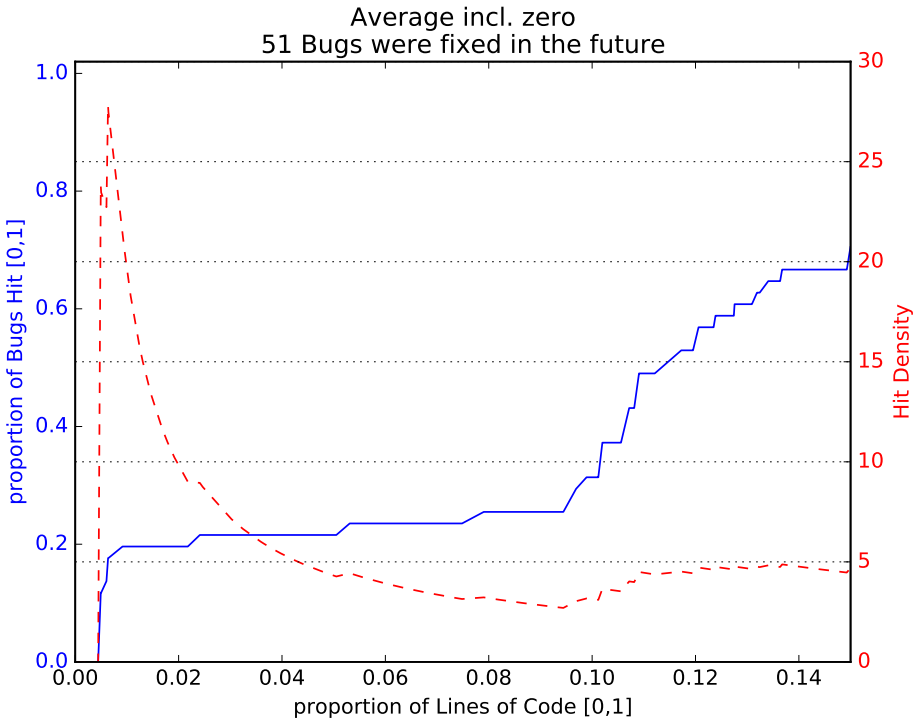


Figure 4.1.: Cost Effectiveness and Hit Density Curve

4.1.2. Hit Density

The hit density is similar to the defect density proposed by Nachiappan et al. [4] but instead of using the sum of all defective elements, this thesis only uses the sum of bugs correctly predicted, hit. Rahman et al. [5] use this idea called closed bug density. It reflects the proportion of closed bugs per proportion of proposed lines of code:

$$\text{Hit Density} = \frac{\%\text{ of bugs}}{\%\text{ of lines of code}}$$

We use the maximum of the hit density is an indicator of the peak performance of the algorithms and also inspect the proportion of loc at which the maximum hit density occurs.

We use hit density as a metric to compare different algorithms as it describes the effectiveness of the inspection of the proposed lines of code. This is interesting because the maximum hit density occurs early with few proposed loc and thus is useful for code review where only few loc, compared to the whole project, are looked at.

4.1.3. Recall and Precision

Two metrics often used in software analysis are recall and precision, defined by:

$$\text{Recall} = \frac{\text{True Positive}}{\text{Defective}}$$

$$\text{Precision} = \frac{\text{True Positive}}{\text{Marked as Defective}}$$

Recall represents how many of the truly defective elements were found. Precision represents how many of the elements marked as faulty were really defective.

We don't use Recall and Precision as it is unknown whether something is a true positive and what the overall amount of defective elements is.

4.2. Process

To decide if the proposed changes help to reach the goals set at the start of the thesis, we investigate the algorithms with the two main focuses being:

- Determine if a line-based approach of the Bugspots algorithm can yield better results than the file-based approach Bugspots uses.
- Determine if the additional information the line-based approach offers can further improve the review process.

Because the evaluation process is deterministic, all data can be reproduced with revision `d798da599f292191ad16fc144730330045075154` of the project.[1][2]

4.2.1. Creating a Pseudo Future

One way to evaluate the algorithms would be to let them make a prediction and either make a full code audit or wait for future commits to check if the prediction was right.

To simulate this kind of evaluation, the commit history of a repository is used to create a pseudo future. Instead of starting at the newest commit and letting it look into the past, we chose a commit in the past as the start point. From that start commit we run the algorithms and check the results against the commits that are younger than the start commit. Figure 4.2 shows a diagram visualizing the idea.

[1] https://gitlab.com/sims1253/Linespots

[2] Even though the design of the evaluation process is deterministic, values do differ between runs. As this only effects the less significant decimal places and the same trends can be observed over different runs. Due to time constrains this was not fixed before the deadline. For information about this issue go here: https://gitlab.com/sims1253/Linespots/issues/40

Using the same regex to identify bug-fix commits, we collect the bug-fixes in the pseudo future by running the absolute bug count algorithm by Rahman et al. [5]

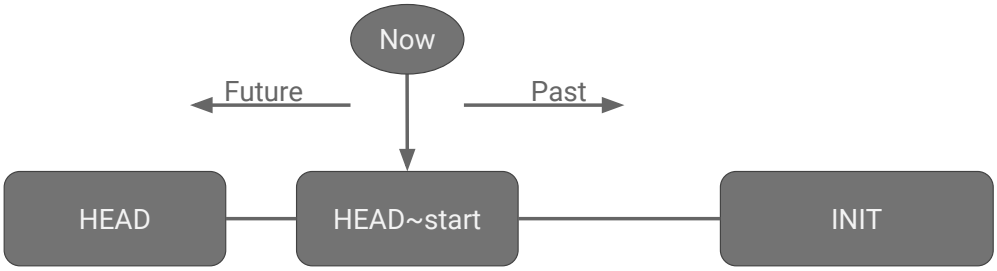


Figure 4.2.: Creating a Pseudo Future

4.2.2. Determining Hit Density and AUCEC

We use the result of the algorithms to build a set of proposed files. The set starts empty and files are added one after another, sorted by their scores in descending order. For each file put into the set, we add the number of fixes it received in the pseudo future to the number of fixes hit by the files in the set.

From the set of files and corresponding number of fixes received in the pseudo future, we calculate the relation between lines of code inspected and fixes found defined by:

$$\text{Hit Density} = \frac{\frac{\text{Bugs Hit by the Proposed Set}}{\text{Bugs Fixed in the Pseudo Future}}}{\frac{\text{LoC in the Proposed Files Set}}{\text{LoC in the Project}}}$$

We use the file set and fix counts to plot the CEC in the 2D plane. For every added file the proportion of loc of the project is the x coordinate. The proportion of future bugs hit is the y coordinate. Using the points, the area under the curve can be approximated and setting a threshold for x, we obtain the area at 20% lines of code covered.

For the line-based result we assign every bug-fixing commit a score that equals the highest line-score corresponding to a line that was part of the fix. We then build a set of proposed lines by going over the fixes sorted based on their assigned score in descending order. For each fix we count the number of lines with a higher or equal score. This is the number of lines that would have to be inspected to detect the bug. From these number we calculate the hit density and AUCEC the same way as for the file-based results.

For each run, the start, the depth, the AUCEC, the AUCEC_{20}, the maximum hit density, and the x coordinate for maximum hit density are recorded to allow comparison afterwards.

4.2.3. Choosing a Regular Expression

We only use the information from the commit message to determine if a commit is a bug-fix or not. Because we don't use further information, the ability to find bugs relies on the accuracy of the regex, i.e., how well it is able to separate commits including bug-fixes from commits without.

We use commit message guidelines and manual commit message review to determine a fitting regex for each project. The regular expressions used for evaluation and a reasoning for them can be found in section 5 for the respective repositories.

4.3. Test Repositories

Rahman et al. used five open source projects for their evaluation [5]. We use those projects as a basis for our evaluation data and in addition, we also considered the coala project. We chose the evaluation projects to offer diverse scenarios mostly based on the values shown in table 4.1.

Evolution[3] is the default e-mail client for the Gnome desktop and it uses a well documented and modern workflow and has clear commit message guidelines. The table shows, that Evolution has the highest amount of loc as well as a medium file count, high number of commits and a long development history.

Httpd[4] is a web server developed by the Apache Software Foundation. It uses a more conservative workflow that relies mostly on the issue tracker for information, which makes it harder to retrieve information from the commit messages. The table shows, that Httpd has less loc than Evolution but still a lot. The number of files is medium too, the number of commits as well. Httpd is the oldest project we considered.

Nautilus[5] was not chosen due to the similarities to Httpd. The big difference of the two projects is the number of files. The similarities and time constrains led us to the decision to not use Nautilus for our evaluation.

GIMP[6] and Lucene-Solr[7] use merges in their workflows which are not handled by Linespots, thus we do not use them for evaluation.

Coala[8] is a framework for static code analysis. It is a young project that made changes to its workflow over time. It uses a workflow based on Gnome's but stricter in the way, how commits are labelled as bug-fixes or features. The clear labelling of commits as bug-fixes or non bug-fixes was introduced in mid 2016.

[3] https://github.com/GNOME/evolution
[4] https://github.com/apache/httpd
[5] https://github.com/GNOME/nautilus
[6] https://github.com/GNOME/gimp
[7] https://github.com/apache/lucene-solr
[8] https://github.com/coala/coala

The table shows, that coala is the smallest and youngest project in every aspect.

Both evolution and httpd are widely used, while coala's user base is small in comparison.

Using additional projects for the evaluation was considered but not done, due to time constrains for this thesis.

Name	Lines of Code	Files	Commits	First Commit
Evolution	3,320,062	2,480	42,527	1998.01.12
Httpd	947,606	3,609	29,121	1996.07.03
Nautilus	963,567	529	19,577	1997.01.02
GIMP	57,422	8,266	37,434	1997.01.01
Lucene-Solr	182,999	9,593	25,886	2001.09.11
coala	27,252	314	3,261	2014.07.18

Table 4.1.: Summary of Test Repositories

4.4. Choosing the Best Parameters

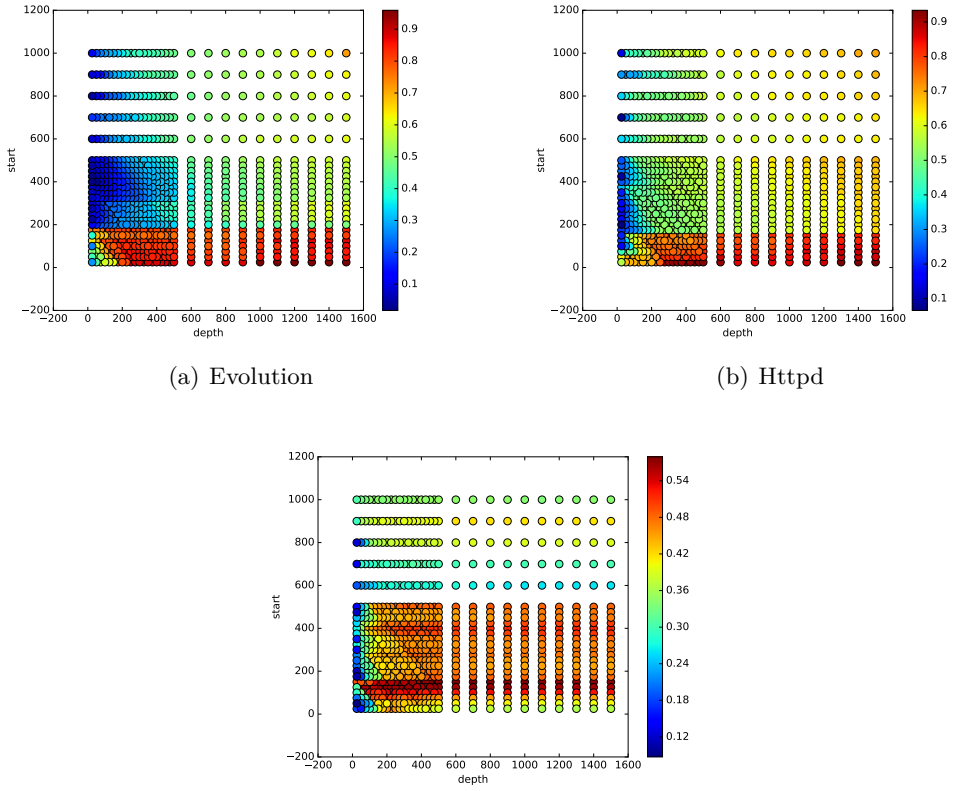
The two parameters that can be changed for the implemented algorithms are the number of commits to look at and the regular expression to identify bug-fixes.

As shown in the scatter plots 4.3(a), 4.3(b) and 4.3(c), a higher search depth almost always improves the AUCEC. Using the scatter plots as a foundation, we recommend a depth of at least 300 commits. This is based on the generally low AUCEC values before the 200-300 commit depth mark.

For the regular expression, the project documentation can give information about commit message conventions but verifying the commit messages manually should be done to ensure best results. A simple recommendation would be `(fix(ing|e[s|d])?)|bug`. It catches `fix, fixing, fixes, fixed, bug`. These words tend to be used most often as bug-fix identifiers.

As we only wanted to evaluate the algorithms in areas where they perform well in general, we used the scatter plots of the AUCEC shown in figure 4.3 to determine the pseudo future sizes and depth range. We chose pseudo futures up to 175 commits, as the scatter plots show a general loss of result quality between 150 and 200 commits upwards. For depths, we chose values between 150 and 1500. The scatter plots show that a depth of at least 300 commits is needed to ensure good results and a depth of 1500 commits still can be calculated with reasonable time and memory resources.

For each project we show one scatter plot exemplary, the remaining ones can be found in chapter C.



(a) Evolution

(b) Httpd

(c) coala

Figure 4.3.: AUCEC Scatter Plots for the Bugspots Implementation with File Tracking

5. Evaluation Results

For each tested project we present the median values for AUCEC, $AUCEC_{20}$, maximum hit density and loc for maximum hit density. The tables also include the respective standard deviations. For the maximum hit density we also show additional box plots that ignore outliers to improve readability.

5.0.1. Gnome/evolution:

We chose the regex, `Bug [1-9][0-9]*` based on the gnome commit guidelines and a manual review of commit messages, suggesting, that although there seem to be bug-fixes without attached bug labels and bug labels attached to fixes that would not qualify as a bug-fix, it is the most promising regex.

Table 5.1 shows that all file-based results have similar AUCEC and $AUCEC_{20}$ values and are consistent. The line-based values are lower and also less consistent.

Name	AUCEC Mean	AUCEC Dev.	$AUCEC_{20}$ Mean	$AUCEC_{20}$ Dev.
Bugspots	0.848134	0.058794	0.133341	0.009466
Bugspots with tracking	0.848144	0.058800	0.133297	0.009458
Average	0.860171	0.056994	0.138965	0.010949
Top ten percent	0.859782	0.058500	0.138111	0.012117
Line Ranking	0.462762	0.135338	0.086136	0.025323

Table 5.1.: AUCEC for Evolution

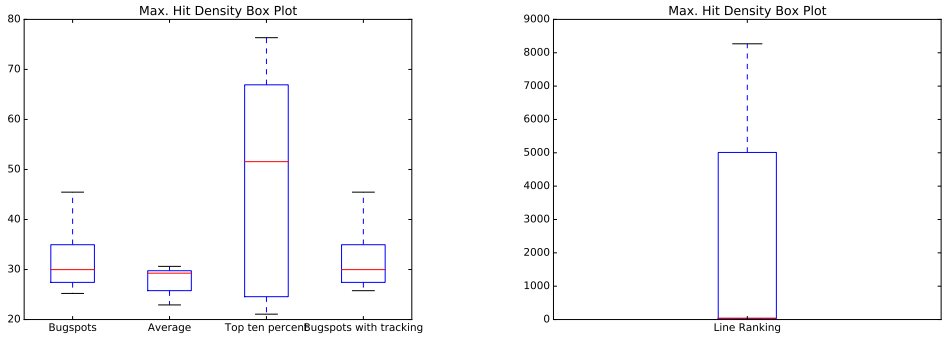
Table 5.2 shows that Linespots improves maximum hit density both with file-based results and with line-based results but the standard deviation is higher than for the Bugspots implementations. The loc for the maximum hit density are similar for all implementations besides the average line-score projection which is more than 1.5 times as high as the rest. Figure 5.1 shows that the hit density is usually higher for the line-based results than for the file-based results. The top ten percent projection is the best file-based solution.

Name	Max. HD Mean	Max. HD Dev.	Max. HD Index Mean	Max. HD Index Dev.
Bugspots	35.654	12.520	0.0080079910431794	0.0048496782430677
Bugspots with tracking	35.654	12.520	0.0080079910431794	0.0048496782430677
Average	45.229	72.736	0.0133191119281376	0.0071071095195690
Top ten percent	101.036	148.556	0.0069108714434092	0.0082834923878903
Line Ranking	1959.500	3073.980	0.0088131981416319	0.0085804463304783

Table 5.2.: Maximum Hit Density for Evolution

5.0.2. Apache/httpd:

We chose the regex `fix` based on a manual review of commit messages because the commit message guidelines do not give clear instructions on how to identify bug-fixes.



(a) File-Based Results

(b) Line-Based Result

Figure 5.1.: Evolution: Box Plots for the Maximum Hit Density

Almost every commit has a link to an issue in a bug tracker but without using a lookup via the tracker's api, the only identifier for bug-fixes available in commit messages was the word `fix`.

Table 5.3 shows similar AUCEC and $AUCEC_{20}$ values for all file-based results. Again the line-based result has worse AUCEC and $AUCEC_{20}$ than the file-based results but the gap in consistency is smaller than for Evolution.

Name	AUCEC Mean	AUCEC Dev.	$AUCEC_{20}$ Mean	$AUCEC_{20}$ Dev.
Bugspots	0.784571	0.110193	0.094818	0.024540
Bugspots with tracking	0.784521	0.110005	0.094764	0.024500
Average	0.772814	0.106025	0.092620	0.022623
Top ten percent	0.776976	0.113739	0.099360	0.026808
Line Ranking	0.193964	0.168412	0.038635	0.033627

Table 5.3.: AUCEC for Httpd

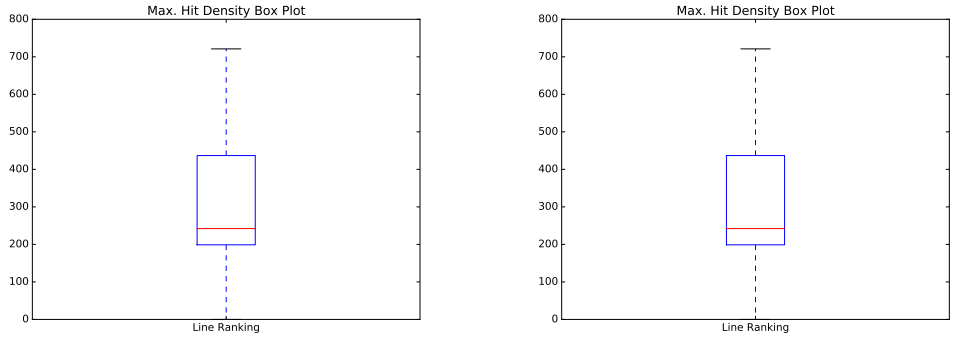
Table 5.4 shows that the average line-score projection has the highest mean maximum hit density but also a high standard deviation. The top ten percent projection has the worst maximum hit density while the line-based result is second only to the average projection. The line-based result has the lowest loc for maximum hit density. Figure 5.2 shows that the top ten percent projection has the lowest maximum hit density in general. Without the outliers the average projection looks similar to the Bugspots implementations and the line-based result has the highest maximum hit density.

5.0.3. coala:

We chose the regex `Fixes https:` based on the commit guidelines and a manual review of commit messages. The strict rule of using the `Fixes` keyword combined with a link to

Name	Max. HD Mean	Max. HD Dev.	Max. HD Index Mean	Max. HD Index Dev.
Bugspots	124.510	99.600	0.0041537353121725	0.0119488852162963
Bugspots with tracking	124.510	99.600	0.0041537353121725	0.0119488852162963
Average	4632.470	11091.675	0.0018729770731374	0.0024959862626980
Top ten percent	77.653	82.048	0.0092016154153657	0.0162686836814913
Line Ranking	307.531	212.493	0.0003983291328505	0.0002665438732935

Table 5.4.: Maximum Hit Density for Httpd



(a) File-Based Results

(b) Line-Based Result

Figure 5.2.: Httpd: Box Plots for the Maximum Hit Density

an issue on Github wasn't introduced until earlier this year which leads to worse results for higher depths as can be seen in figure 4.3(c). In general this repository does not work well with either Bugspots nor Linespots.

Table 5.5 shows similar AUCEC values for the file-based results and lower values for the line-based result. The mean $AUCEC_{20}$ is highest for the line-based result and worst for the projected Linespots results.

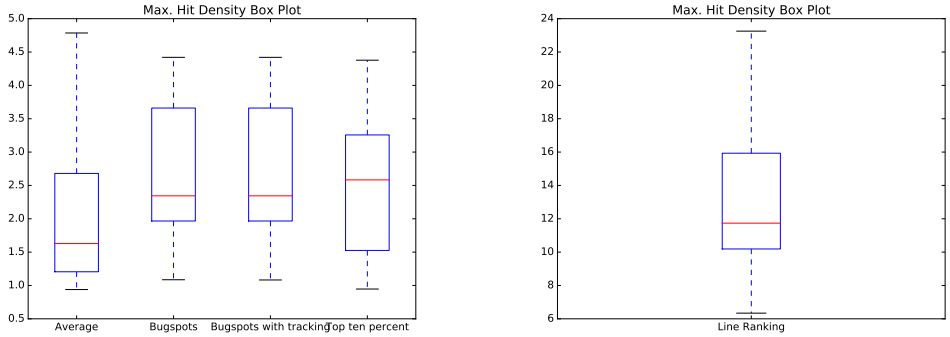
Name	AUCEC Mean	AUCEC Dev.	$AUCEC_{20}$ Mean	$AUCEC_{20}$ Dev.
Bugspots	0.482468	0.070940	0.027325	0.009118
Bugspots with tracking	0.486324	0.070041	0.027401	0.009019
Average	0.469454	0.065403	0.020175	0.007529
Top ten percent	0.475325	0.069582	0.025270	0.008440
Line Ranking	0.280775	0.146721	0.044863	0.022333

Table 5.5.: AUCEC for coala

Table 5.6 shows that the line-based result has the best maximum hit density and lowest loc at maximum hit density. The Bugspots implementations are the most consistent ones. The projected results are less consistent than the Bugspots results overall. Figure 5.2 confirms that the line-based result has the highest maximum hit density and the top ten percent projection has a better median than the other file-based results.

Name	Max. HD Mean	Max. HD Dev.	Max. HD Index Mean	Max. HD Index Dev.
Bugspots	2.798	0.967	0.0480461556354995	0.0915444751121636
Bugspots with tracking	2.798	0.967	0.0480601049342504	0.0916091067412921
Average	2.374	2.691	0.1105846311615345	0.1763613845545503
Top ten percent	3.145	3.667	0.0619731737428163	0.1018196120878422
Line Ranking	13.211	5.393	0.0137126906718898	0.0107461301686817

Table 5.6.: Maximum Hit Density for



(a) File-Based Results (b) Line-Based Result

Figure 5.3.: coala: Box Plots for the Maximum Hit Density

6. Discussion

This chapter holds the discussion of evaluation results. It offers interpretations for the shown data and draws a conclusion based on the trends that show in it.

6.1. Project Differences

We chose the three projects to cover a broad spectrum of project types. Here we will offer a comparison of how the algorithms performed on the projects and hypothesis for why they do so.

The first observation is that Evolution and Httpd show much better results overall than the coala repository with Evolution having the best AUCEC and $AUCEC_{20}$ values. For both projects, all four file-based results show much better AUCEC and $AUCEC_{20}$ values than the line-based version. The hit density on the other hand is best for the line-based version. For coala, the file-based results are much worse than for the other two projects. The line-based result still has lower AUCEC than the file-based ones but the gap is smaller and the $AUCEC_{20}$ values are better for the line-based result. Most noteworthy is the very good maximum hit density the line-based result offers compared to the file-based ones.

One hypothesis for the different behaviours could be the differences in commit message guidelines used in the projects. While coala's current workflow includes identifying commits as bug-fixes explicitly in the commit message, this rule was introduced around 500 commits ago, so going further into the past will not help as much as one would expect. The changed workflow shows in the sub par scaling coala has with the higher depth. For Evolution and Httpd, the workflows are more stable and did not change recently but identification of bug-fix commits is not done clearly. This uncertainty could lead to a smaller fraction of bug-fix commits that are correctly identified and to a higher portion of non fix commits that are falsely identified as fixes. As the evaluation uses the same process, problems with misidentification should not show in the evaluation data.

Another hypothesis is the different amount of file level changes that could interfere with algorithm performance. Large amounts of added and deleted files could make past information less useful, as gathered information is lost for deleted files and new files have less time to face fixes. Table 6.1 shows, that the number of file level changes indeed is much higher for coala, than for Evolution and Httpd, which supports the idea that for coala bugs might happen in files that are added in the future and thus can not be proposed for inspection in the present.

Name	Depth	Added Files	Deleted Files	Renamed or Moved Files
Evolution	500	57	140	1
	1500	102	126	9
Httpd	500	18	0	0
	1500	68	20	2
coala	500	65	62	6
	1500	309	409	294

Table 6.1.: File Changes

6.2. File Tracking

Both the Bugspots implementation, with and without file tracking, performed well in most scenarios. Comparing the results of both, file tracking has minimal influence on the metrics.

The reasoning for why file tracking should improve results is the preservation of information. If file level changes are not tracked, they will result in loss of all information about that file from before the change. Moving the information with the file should in theory yield better results, since more information is available. If a file that is bug prone is renamed or moved and the score is not moved with it, the ranking for that file would change but within our experiments, we did not observe a lot of file deletions or movements that led to changing scores.

Table 6.1 shows file level changes for the projects for the last 500 and 1500 commits. It shows that coala has by far the highest number of moved files, which supports the observation that coala shows the biggest difference between the two versions, although the difference is still minimal. As more file level changes happen, the possibility for a relevant information loss rises so that preserving that information yields better results. But even though a lot of file-level changes happened for coala in the last 1500 commits, the influence on the metrics is minimal. The small impact could be because the changes happen in the older part of the commits that, due to the weighting function, has less influence anyway. Also, to effect the ranking of files a change would have to hit a file with a high score to have a big influence.

In conclusion, even though the impact of file tracking is minimal, we suggest to use it, since projects with more file level changes might profit from it and those with less do not experience drawbacks.

6.3. Linespots File Scores

The Linespots algorithm is able to gather information with a higher resolution than the Bugspots algorithm. This thesis proposed a way to project these high resolution line

scores back on to file scores and compare the result with the reference implementations.

In this section we discuss the two projected result variants and compare them against each other and the Bugspots implementations.

6.3.1. Comparison of Average and Top Ten

Both versions are less consistent than the reference implementations. Only the maximum hit density shows real differences for Linespots' file-based results but it is the least consistent value for both, with it being high for one projection but low for the other.

In conclusion, there is no clear favourite between the average and top ten projection but a trend becomes apparent, favouring the top ten projection.

6.3.2. Comparison of Projected Scores and Reference Implementations

Comparing the Bugspots reference implementations to the average and top ten results gives a clear favourite roll to the reference implementations. They are more consistent and yield better results most of the time. The only metric, in which the average and top ten results can outperform the reference implementations regularly, is maximum hit density. But due to the inconsistency of the projected results, neither of the two can exceed the Bugspots implementations sufficiently to be able to propose it for that metric. Overall, the values are often similar but in general, the reference implementations give better results.

6.4. Linespots Line Scores

In this section we discuss the line-based results of the Linespots algorithm.

6.4.1. Behaviour

From all projects tested, the line-based version has lower AUCEC and $AUCEC_{20}$ values than the other implementations. Only for the coala project, it can achieve higher $AUCEC_{20}$ values than the other implementations.

In all projects, and especially the coala project, in which the other algorithms performed poorly, the line-based version has the best maximum hit density values. There are some occasions, like the extreme high outliers for the average projection with the Httpd project, where another algorithm tops the line-based value but it never has bad hit density values.

6.4.2. Comparison of Line Based Results and File-Based Results

Looking at the data, the first thing to note are the low values for AUCEC and $AUCEC_{20}$ for the line-based results. Two hypothesis for the low values are the smaller amount of scored code after running the algorithm and the inability to find bugs in new code.

Contrary to the file-based results, the line-based variant holds a far smaller part of the project's code in its scored-code set. Instead of adding each file that has been part of a bug-fix, only the corresponding lines are added. The line-based results miss future bug-fixes that happen in files that received fixes before but on lines that did not receive fixes. The file-based results hit those bugs. This difference in bug hitting ability leads to smaller AUCEC values for the line-based results as they hit less future bugs.

Another reason for this behaviour could be the inability to predict bugs in lines that are written in the pseudo future. While the file-based approach proposes whole files for inspection, the line-based approach only proposes lines. Bugs that are fixed in code that was introduced in the future, can occur in files from the past but never in lines from the past. Thus, the line-based-results are not able to find bugs in lines of code that did not exist in the past. The difference stems from the design of the evaluation method and is not existent due to the algorithm. An evaluation could be designed so that it ignores code introduced in the future for file-based results too.

Due to these design limitations, the line-based version is expected to achieve lower AUCEC and $AUCEC_{20}$ values as it can not hit the same absolute amount of bugs as the file-based versions with a much smaller set of code.

As stated before, the maximum hit density for the line-based version is the highest in most cases. This advantage probably is due to the design decision, to not hold whole files but only lines of code instead. The same reason that might lead to smaller AUCEC values probably also results in higher hit density values. Where the file-based approach proposes every file that was part of a bug-fix, the line-based version only proposes lines that were part of a fix and are still in the code base when the algorithm is run.

When a bug prone line is deleted from a file, the file-based scores do not change while the line-based version drops that line and thus will not propose it for inspection. This preservation of old information could result in files being proposed that had bugs fixed in them, but in which the bug prone lines were removed later. Another potential reason for the higher hit density values for the line-based version could be that it proposes less overhead lines per bug. Because the line-based version only adds the lines that are part of a bug-fix to the proposal set instead of the whole file, fewer lines with low scores are proposed upfront and either are proposed later or not at all.

The reason why the hit density is not a lot higher all the time might be, that in the projects chosen, one file often contains a lot of the future bugs. If the file-based results propose this file first, they could achieve a high hit density where if they miss it, the hit density is low. The line-based version can achieve a high hit density more consistently as

it has the information it holds in more detail than the file-based approaches.

6.5. Summary

The results of the evaluation leads to the conclusion that all tested algorithms work best with older more stable repositories that introduce fewer file level changes. Taking a look at individual performance, the Bugspots algorithm is the most consistent over repositories and parameters, having the best AUCEC and $AUCEC_{20}$ values. File tracking does improve the results in theory but we could only observe minimal effects.

The file-based results of the Linespots algorithm are less consistent than the Bugspots results and even though they showed promising improvements for the maximum hit density, overall the inconsistency makes them unreliable and they should not be used instead of the reference algorithm.

The line-based results of the Linespots algorithm show lower AUCEC and $AUCEC_{20}$ values than the file-based results, as was expected, but could improve the hit density in most cases.

With the

How does a line-based approach effect the area under the hit density curve?

The evaluation showed that the AUCEC and $AUCEC_{20}$ both suffer from the usage of line scores instead of file scores. The lower values probably are due to more precise information gathering, which leads to a smaller set of ranked code so the amount of bugs that can be correctly predicted is smaller. Another factor might be that the evaluation enables the file-based versions to predict bugs in code that was introduced in the future which again increases the amount of bugs the file-based results can predict compared to the line-based results.

With these conclusions, we answer the research questions:

Can the line-based approach improve the hit density of the Bugspots algorithm?

The evaluation found that using file-based results of the line-based approach did not show improved hit density compared to the Bugspots implementations consistently. Using the line-based results instead, does show a trend of improved hit density in most cases, even for projects that do not work well with file-based results.

7. Recommendations

This chapter holds recomendations for future work on this topic.

7.1. Algorithmic Improvements

There still are a lot of possible modifications that could improve the result quality of the algorithm. The proposed changes are non exhaustive:

As stated earlier, the precision with which a bug-fix commit can be correctly identified is very important as it gives the data base for the algorithm. Missing bug-fix commits or identifying non fix commits wrongly leads to missed or false information in the data set used for the ranking. This probably leads to a sub par precision for the prediction of future bugs. To improve the precision, additional information can be used to identify the commits. Rahman et al. [5] propose the use of bug tracking databases as a source for additional information. If a commit has a link to a bug tracker entry attached, it can be possible to obtain information about the nature of the issue that was worked on with the commit. Additionally, extensive manual classification can be done to ensure maximum precision of the input data for the algorithms.

Another way of improving identification of commits could be to train neural networks with pre classified commits and use them as additional information in the identification process.

In addition to the improved precision of the identification, another improvement could be gained by using a more detailed set of data than just the closed bugs. The total number of bugs consists of the closed bugs, the reported but unfixed open bugs, and the unreported bugs. As it is not possible to get information about unreported bugs without reviewing the code, these bugs can not be used by any prediction algorithm. The open bugs however can be used as long as they have files associated with them. This information again can be obtained by searching a bug tracking database.

7.2. Future Work

Due to the time constrains set by the format of this thesis, the evaluation process was far less in-depth and exacting as needed for a clear definite answer of the research question. For future research work on this topic we propose multiple changes and additions to the evaluation process.

To broaden the scope of the evaluation a higher number of projects should be tested and compared during the evaluation process. The small sample size used in this thesis is prone to individual cases of extreme behaviour as could be observed with the hit density

values for Httpd. A larger dataset would give more capability to compare and even out such spikes in the results.

Together with the bigger dataset a number of statistical methods could be used to find trends in large numbers of results. This would provide additional arguments to support or refute trends observed in case studies like the one performed in this thesis. In addition, this could give a better foundation for good default parameters and factors that help projects to work well with the algorithm.

To improve the comparability of the line-based and file-based results the way of identifying if a future bug was fixed in a proposed file should be changed to only include the same set of fixes as the file-based version. This can be done by tracking the lines and only counting bug-fixes that involve code that was introduced in the past. If instead this is ignored as was in this thesis, the file-based results have the advantage of being able to predict future bugs in code that also was introduced in the future.

Finally, the same improvements for identifying bug-fix commits for the algorithm should be applied to the evaluation process so that the rules for past and future bugs are the same.

A. Linespots Usage

This section holds information about how to reproduce the results used in the evaluation process and how to use the Linespots algorithm on any repository.

A.1. Reproduction

To reproduce the results from the evaluation process the following commands can be used. If you are only interested in the results and not actually want to reproduce them, you can get them in the result repository[1]

First clone the repository of the project for which the results should be reproduced:

- Evolution: https://github.com/GNOME/evolution
- Httpd: https://github.com/apache/httpd
- coala: https://github.com/coala/coala

```
git clone project
git clone https://gitlab.com/sims1253/Linespots.git
cd Linespots
git checkout d798da599f292191ad16fc144730330045075154
cp linespots.py /path/to/test/repository/linespots.py
cp evaluate.py /path/to/test/repository/evaluate.py
cd ~/path/to/test/repository
```

Then run the corresponding command for the project once as given in section A.1.2 and once with the –line True flag addition. The results will be stored in the linespots folder. You have to move the results after each run as the folder is overwritten otherwise.

A documentation for the command line options can be found in the readme of the repository or via the `-h` option.

A.1.1. Example

As an example, this would be the complete procedure for Evolution:

```
git clone https://github.com/GNOME/evolution
git clone https://gitlab.com/sims1253/Linespots.git
cd Linespots
git checkout d798da599f292191ad16fc144730330045075154
cp linespots.py ../evolution/linespots.py
cp evaluate.py ../evolution/evaluate.py
```

[1] https://gitlab.com/sims1253/Linespots-Results

```
cd ../evolution
python3 evaluate.py -o 8abd509a8fd1bdc4c58c08c2a35d1cf11f7ab67f
-d 150 200 300 400 500 600 700 800 900 1000 1100 1200 1300 1400
1500 -f 25 50 75 100 125 150 175 -g "Bug [1-9][0-9]*"

mv linespots linespotsFiles

python3 evaluate.py -o 8abd509a8fd1bdc4c58c08c2a35d1cf11f7ab67f
-d 150 200 300 400 500 600 700 800 900 1000 1100 1200 1300 1400
1500 -f 25 50 75 100 125 150 175 -g "Bug [1-9][0-9]*"--Line True
```

A.1.2. Project Specific Commands

Gnome Evolution

For the scatter plots:

```
python3 evaluate.py -o 8abd509a8fd1bdc4c58c08c2a35d1cf11f7ab67f -d 25 50
75 100 125 150 175 200 225 250 275 300 325 350 375 400 425 450 475 500
600 700 800 900 1000 1100 1200 1300 1400 1500 -f 25 50 75 100 125 150
175 200 225 250 275 300 325 350 375 400 425 450 475 500 600 700 800 900
1000 -g "Bug [1-9][0-9]*"
```

For the evaluaton data:

```
python3 evaluate.py -o 8abd509a8fd1bdc4c58c08c2a35d1cf11f7ab67f -d 150
200 300 400 500 600 700 800 900 1000 1100 1200 1300 1400 1500 -f 25 50
75 100 125 150 175 -g "Bug [1-9][0-9]*"
```

Apache Httpd

For the scatter plots:

```
python3 evaluate.py -o b7a493d92fa60c339244bf6f7ff0dca29c95e562 -d 25 50
75 100 125 150 175 200 225 250 275 300 325 350 375 400 425 450 475 500
600 700 800 900 1000 1100 1200 1300 1400 1500 -f 25 50 75 100 125 150
175 200 225 250 275 300 325 350 375 400 425 450 475 500 600 700 800 900
1000 -g fix
```

For the evaluaton data:

```
python3 evaluate.py -o b7a493d92fa60c339244bf6f7ff0dca29c95e562 -d 150
200 300 400 500 600 700 800 900 1000 1100 1200 1300 1400 1500 -f 25 50
75 100 125 150 175 -g fix
```

coala

For the scatter plots:

```
python3 evaluate.py -o 3448554d48423e5082cd86b3a1a9b708b147a7de -d 25 50
75 100 125 150 175 200 225 250 275 300 325 350 375 400 425 450 475 500
600 700 800 900 1000 1100 1200 1300 1400 1500 -f 25 50 75 100 125 150
175 200 225 250 275 300 325 350 375 400 425 450 475 500 600 700 800 900
1000 -g "Fixes https:"
```

For the evaluaton data:

```
python3 evaluate.py -o 3448554d48423e5082cd86b3a1a9b708b147a7de -d 150
200 300 400 500 600 700 800 900 1000 1100 1200 1300 1400 1500 -f 25 50
75 100 125 150 175 -g "Fixes https:"
```

A.2. Algorithm Usage

To apply the Linespots Algorithm to any repository clone the Linespots repository, copy the linespots.py file into the repository you want to analyse and run the following command:

```
python3 linespots.py
```

Additional parameters are available. More information can be obtained with the `-h` flag.

B. Abbreviations

Abbreviation	Long Form
CEC	cost effectiveness curve
loc	lines of code
regex	regular expression
AUCEC	area under cost effectiveness curve
$AUCEC_{20}$	area under cost effectiveness curve at 20% loc

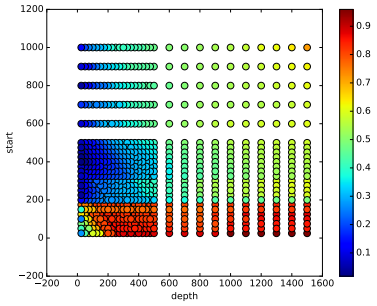
Table B.1.: Abbreviations

C. Results

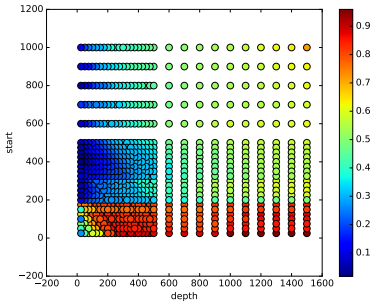
As the amount of Plots generated for a single project is too high to be displayed here, only the scatter plots and box plots will be shown. The same applies for the metrics for individual parameter combinations.

To see the full results either use the result repository[1] or reproduce them yourself as described in the previous chapter.

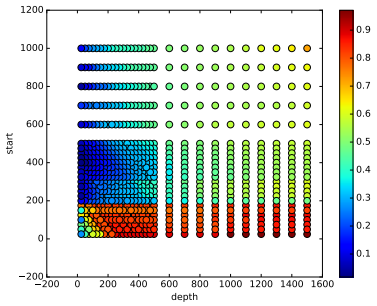
[1] https://gitlab.com/sims1253/Linespots-Results



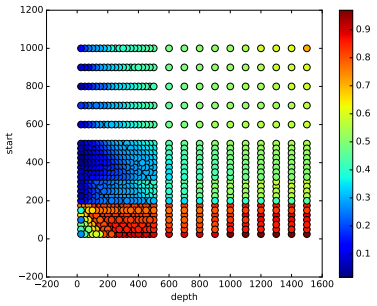
(a) Bugspots without File Tracking



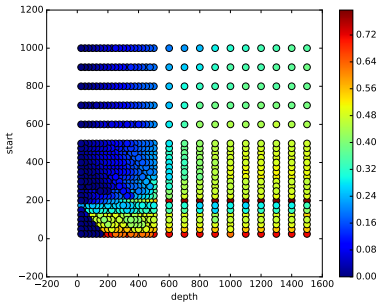
(b) Bugspots with File Tracking



(c) Average Line-Score

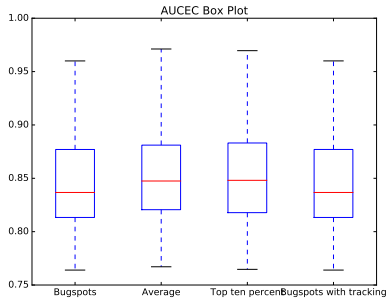


(d) Top Ten Percent

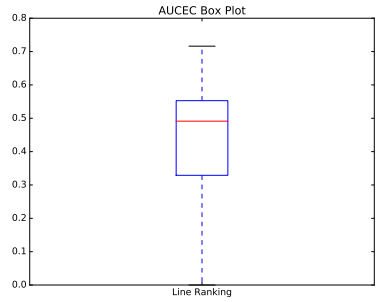


(e) Line-Based

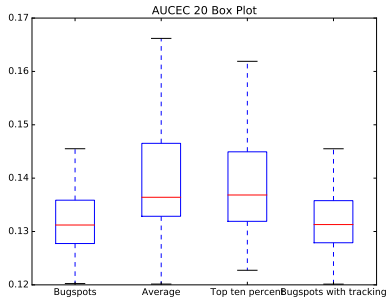
Figure C.1.: coala Scatter Plots for the AUCEC



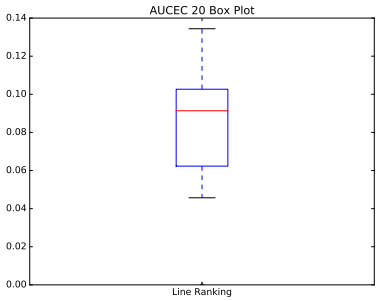
(a) File-Based AUCEC



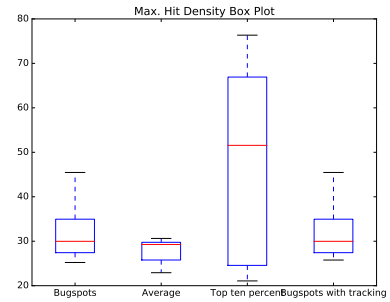
(b) Line-Based AUCEC



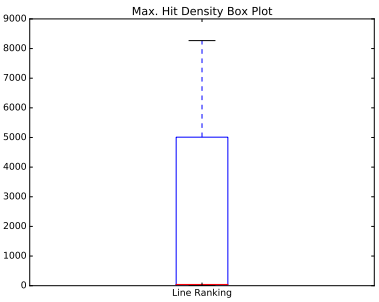
(c) File-Based $AUCEC_{20}$



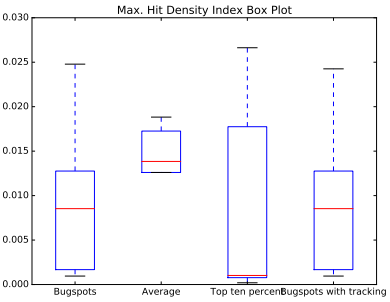
(d) Line-Based $AUCEC_{20}$



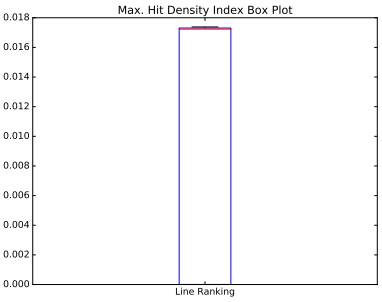
(e) File-Based Maximum Hit Density



(f) Line-Based Maximum Hit Density

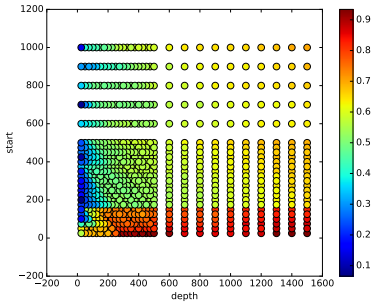


(g) File-Based Maximum Hit Density LoC

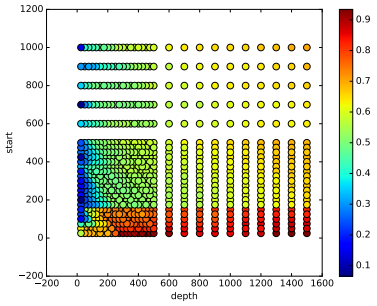


(h) Line-Based Maximum Hit Density LoC

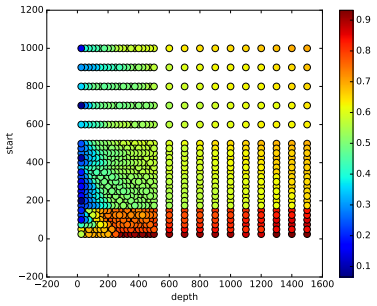
Figure C.2.: Evolution Box Plots



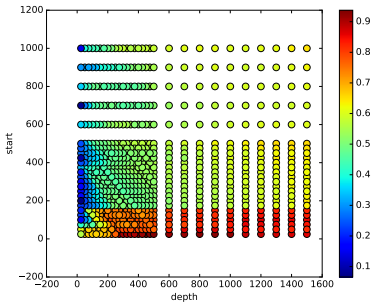
(a) Bugspots without File Tracking



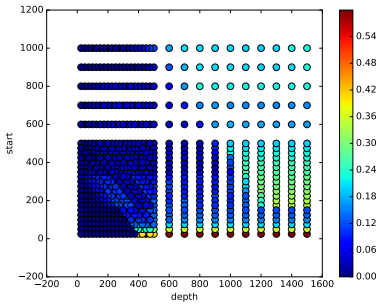
(b) Bugspots with File Tracking



(c) Average Line-Score

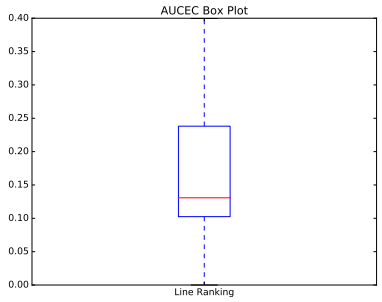


(d) Top Ten Percent

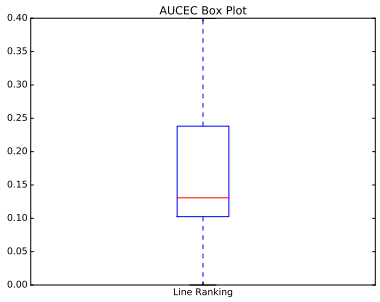


(e) Line-Based

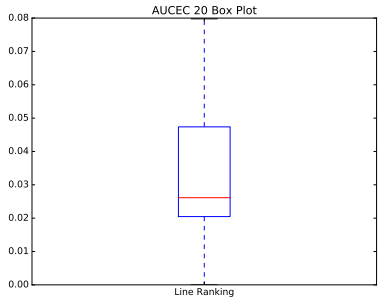
Figure C.3.: coala Scatter Plots for the AUCEC



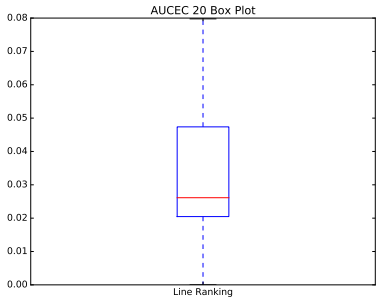
(a) File-Based AUCEC



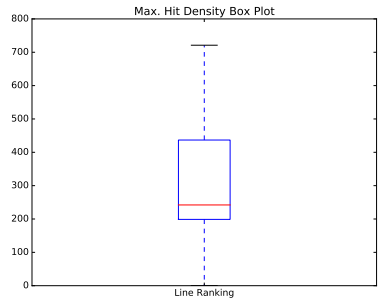
(b) Line-Based AUCEC



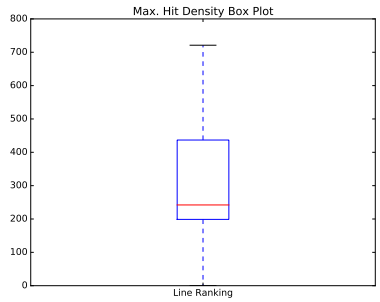
(c) File-Based $AUCEC_{20}$



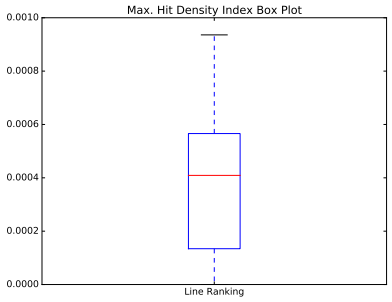
(d) Line-Based $AUCEC_{20}$



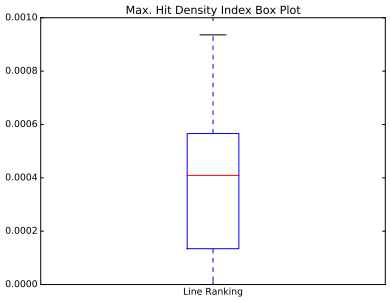
(e) File-Based Maximum Hit Density



(f) Line-Based Maximum Hit Density

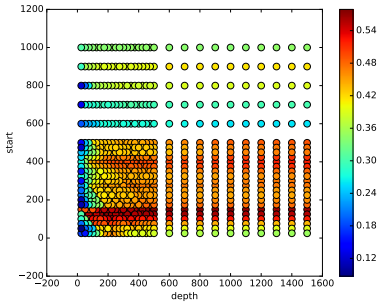


(g) File-Based Maximum Hit Density LoC

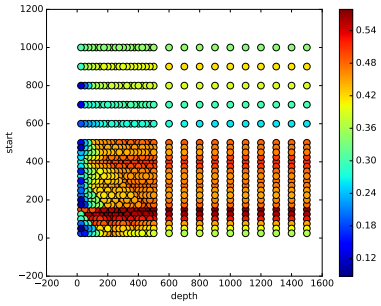


(h) Line-Based Maximum Hit Density LoC

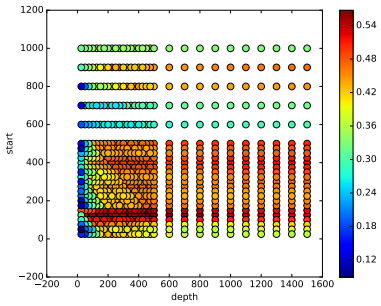
Figure C.4.: Httpd Box Plots



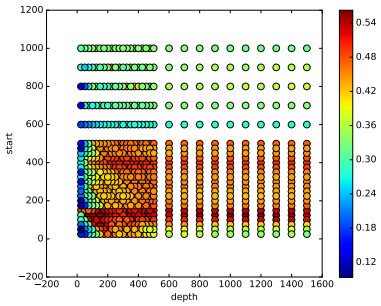
(a) Bugspots without File Tracking



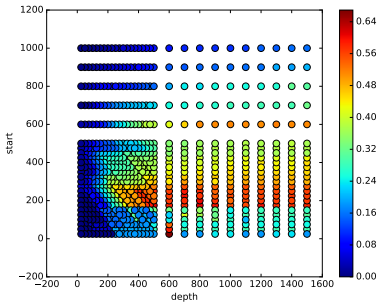
(b) Bugspots with File Tracking



(c) Average Line-Score

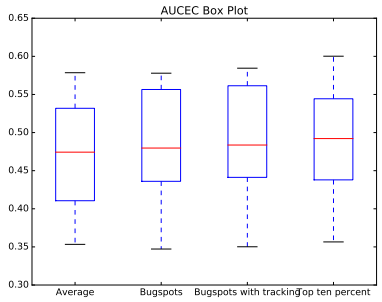


(d) Top Ten Percent

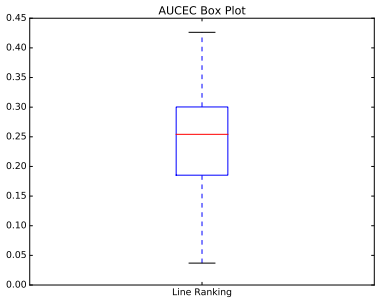


(e) Line-Based

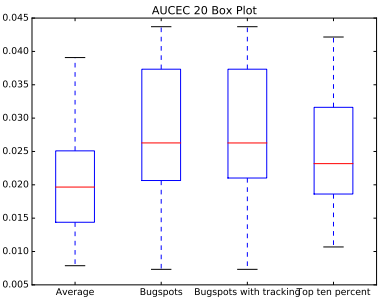
Figure C.5.: coala Scatter Plots for the AUCEC



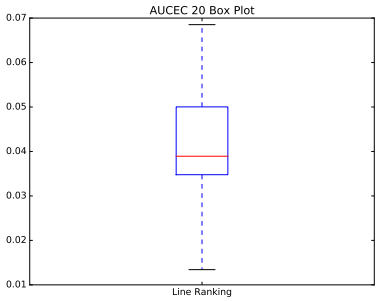
(a) File-Based AUCEC



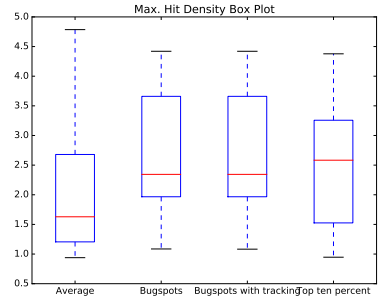
(b) Line-Based AUCEC



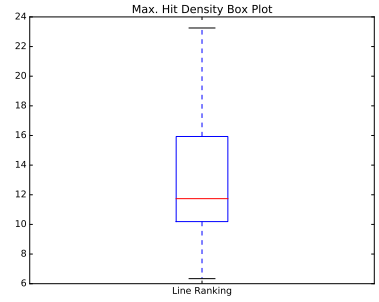
(c) File-Based $AUCEC_{20}$



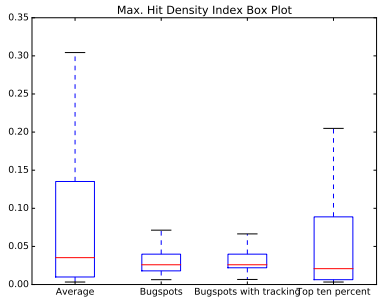
(d) Line-Based $AUCEC_{20}$



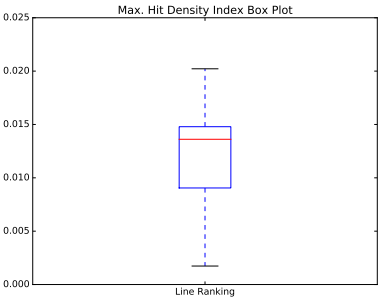
(e) File-Based Maximum Hit Density



(f) Line-Based Maximum Hit Density



(g) File-Based Maximum Hit Density LoC



(h) Line-Based Maximum Hit Density LoC

Figure C.6.: coala Box Plots

D. Affidavit

I, Maximilian Scholz, hereby confirm my thesis entitled **A Line Based Approach for Bugspots** is the result of my own work. I did not receive any help or support from commercial consultants. All sources and/or materials applied are listed and specified in the thesis.

Furthermore, I confirm that this thesis has not yet been submitted as part of another examination process neither in identical nor in similar form.

References

[1] Arisholm, E., Briand, L.C., and Fuglerud, M. Data Mining Techniques for Building Fault-Proneness Models in Telecom Java Software. *Proceedings of the 18th IEEE International Symposium on Software Reliability*, IEEE Computer Society (2007), 215–224.

[2] Canfora, G., Cerulo, L., and Penta, M.D. Identifying Changed Source Code Lines from Version Repositories. *Proceedings of the Fourth International Workshop on Mining Software Repositories*, IEEE Computer Society (2007), 14.

[3] Lewis, C. and Ou, R. "Bug Prediction at Google". 2011. https://google-engtools.blogspot.de/2011/12/bug-prediction-at-google.html.

[4] Nagappan, N. and Ball, T. Use of Relative Code Churn Measures to Predict System Defect Density. *Proceedings of the 27th International Conference on Software Engineering*, ACM (2005), 284–292.

[5] Rahman, F., Posnett, D., Hindle, A., Barr, E., and Devanbu, P. BugCache for Inspections: Hit or Miss? *Proceedings of the 19th ACM Sigsoft Symposium and the 13th European Conference on Foundations of Software Engineering*, ACM (2011), 322–331.