CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

# An Analysis of Linespots and its Utility in Realistic Scenarios

Master's thesis in Computer science and engineering

Maximilian Scholz

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2019

Master's thesis 2019

# An Analysis of Linespots and its Utility in Realistic Scenarios

Maximilian Scholz

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2019

An Analysis of Linespots and its Utility in Realistic Scenarios
Maximilian Scholz



Supervisor: Richard Torkar, Computer Science and Engineering
Examiner: Jan-Philipp Steghöfer , Computer Science and Engineering

Master's Thesis 2019
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2019

An Analysis of Linespots and its Utility in Realistic Scenarios
Maximilian Scholz
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

# Abstract

Fault prediction is a promising technique that can potentially help developers build software with fewer faults. Bugspots is an algorithm developed by Google, that is used for its simplicity and short run times and is used as a baseline for other fault prediction algorithms. Linespots is a variant of Bugspots that works on lines instead of files, thus potentially improving performance through higher precision. In this thesis, we analyzed the effect different weighting-functions and age-calculations have on the performance of Linespots, investigated the possibility to turn Linespots into a classifier and compared the performance and results of Linespots to those of Bugspots. Based on the algorithms, weighting-functions and age-calculations, we used a full factorial experiment design where we evaluated a total of 65 revisions of 23 open source projects from GitHub and analyzed the resulting 780 samples using Bayesian data analysis. We found that none of the weighting-functions or age-calculation variants had any reliable effect on the performance of Linespots and that the classification performance of Linespots makes it unsuited for production use. Furthermore, we found that while the ranked result lists differ between Bugspots and Linespots, the averaged predictive performance is similar. However, Linespots tends to outperform Bugspots for the early parts of the result list. These findings implicate that Linespots could be a better baseline choice for fault prediction than Bugspots, but there is more work needed to identify the optimal parameters for Linespots. Moreover, additional investigations are needed into interactions between different parameters and both the weighting-function and age-calculations, as well as the methodology of using a pseudo future for evaluation.

Keywords: Bugspots, computer science, software engineering, project, thesis, fault prediction, Linespots, Bayesian data analysis, fault localization.

# Acknowledgements

I want to thank Richard Torkar for the support and guidance he has given me since the first time I we met. The work on this thesis essentially started with a question I had for him two years ago and contains countless hours of his time answering my never ending inquiries. You taught me how to be a better researcher and how to wield the obscure power of Bayes.
I would also like to thank Mareile Knudsen, Thorbjörn Junge, Dennis Wurm and Sebastian Schlaadt as well as my parents and brother for the support and feedback they offered throughout my time working on this thesis. I could not have done this without you.
Finally, I would like to thank the stan community [42] for their patience with my questions and the welcoming atmosphere they offered.

Maximilian Scholz, Gothenburg, August 2019

# Contents

**List of Figures** **xiii**

**List of Tables** **xvii**

**1 Introduction** **1**
1.1 Problem and Purpose . . . 2
1.1.1 Research Question 1: What kind of weighting function produces the best results for Linespots? . . . 2
1.1.2 Research Question 2: How does index-based age calculation influence the predictive performance of Linespots compared to time stamp based age calculation? . . . 3
1.1.3 Research Question 3: What is a good cut-off-point to turn Linespots into a classifier? . . . 3
1.1.4 Research Question 4: What is the prediction performance of Linespots compared to Bugspots? . . . 3
1.1.5 Research Question 5: Do Bugspots and Linespots predict faults in the same order? . . . 3

**2 Background** **5**
2.1 Fault Prediction . . . 5
2.1.1 Fault Localization . . . 5
2.1.2 Granularity . . . 6
2.1.3 Result Types . . . 6
2.1.4 Applicability . . . 6
2.2 Past Faults . . . 6
2.2.1 Identification of Faults . . . 7
2.3 Bugspots . . . 7
2.3.1 Weighting Functions . . . 8
2.3.2 Relative Commit Age . . . 9
2.4 Linespots . . . 10
2.4.1 Determining Faulty Elements . . . 11
2.4.2 Scoring Lines . . . 11
2.4.3 Implementation Changes . . . 11
2.4.4 Predictive Performance . . . 12
2.5 Relevance to Practice . . . 12

**3 Methods** **15**

3.1 Objectives . . . 15
3.1.1 Predictive Performance . . . 15
3.1.2 Optimal Cut-Off Point . . . 15
3.1.3 Ranking Comparison . . . 16
3.2 Experimental Design and Preparation . . . 16
3.3 Metrics . . . 16
3.3.1 Cost-Effectiveness . . . 17
3.3.2 Precision and Recall . . . 17
3.3.3 EXAM . . . 19
3.3.4 EXAMF . . . 19
3.3.5 $E_{inspect}$@n . . . 19
3.3.6 Hit Density . . . 19
3.3.7 Average Rank Difference . . . 20
3.3.8 Comparing Granularities . . . 20
3.3.9 Influence of Sorting on Metrics . . . 20
3.4 Dataset . . . 21
3.4.1 Sources . . . 21
3.4.2 Sample Size . . . 21
3.4.3 Building the Dataset . . . 22
3.5 Validation Data . . . 24
3.6 Parameters . . . 25
3.6.1 Fix Indicator . . . 25
3.6.2 Weighting Function and Time Version . . . 25
3.6.3 Origin . . . 25
3.6.4 Depth . . . 25
3.6.5 Future . . . 26
3.7 Procedure . . . 26
3.8 Analysis . . . 27
3.8.1 Exploration and Simulation . . . 27
3.9 Projection Predictive Feature Selection . . . 27
3.9.1 Prior Sensitivity Analysis . . . 29
3.10 Model Design . . . 31
3.10.1 Model Diagnostics . . . 31
3.10.2 Model Comparison . . . 31
3.10.3 Model Interpretation . . . 32

**4 Results** **33**
4.1 Exploration . . . 33
4.2 Research Question 1 . . . 36
4.2.1 EXAM Results . . . 36
4.2.2 AUCECEXAM Results . . . 38
4.3 Research Question 2 . . . 40
4.3.1 EXAM Results . . . 40
4.3.2 AUCECEXAM Results . . . 43
4.4 Research Question 3 . . . 45
4.5 Research Question 4 . . . 46

4.5.1 EXAM Results . . . 46
4.5.2 AUCECEXAM Results . . . 49
4.5.3 RQ4: EXAM25 Results . . . 51
4.5.4 RQ4: EInspect25EXAM Results . . . 53
4.6 Research Question 5 . . . 55

**5 Discussion** **57**
5.1 What kind of weighting function produces the best results for Linespots? 57
5.2 How does index-based age calculation influence the predictive performance of Linespots compared to time stamp based age calculation? . 58
5.3 What is a good cut-off-point to turn Linespots into a classifier? . . . 59
5.4 What is the prediction performance of Linespots compared to Bugspots? 60
5.5 Do Bugspots and Linespots predict faults in the same order? . . . 61
5.6 Other Observations . . . 62
5.6.1 Language and Domain Differences . . . 62
5.7 Threats to Validity and Limitations . . . 62
5.7.1 Faults in the Implementation . . . 62
5.7.2 Training on Evaluation Data . . . 63
5.7.3 Sourcing of Training and Validation Data . . . 63
5.7.4 Impact of Source and Choice . . . 64
5.7.5 GitHub as Data Source . . . 64
5.7.6 Bad Smells . . . 65
5.8 Future Work . . . 65
5.8.1 Standard Evaluation Suite . . . 65
5.8.2 Analyze Smaller Projects . . . 65
5.8.3 Linespots Performance . . . 66
5.8.4 Dataset Building . . . 66

**6 Conclusion** **69**

**Bibliography** **71**

**A Dataset** **I**
A.1 Past Work . . . I
A.2 Drawing Process . . . II

**B Projpred Results** **VII**

**C Models** **XIII**
C.1 Research Question 1 . . . XIII
C.1.1 EXAM . . . XIII
C.1.2 AUCECEXAM . . . XIII
C.2 Research Question 2 . . . XIV
C.2.1 EXAM . . . XIV
C.2.2 AUCECEXAM . . . XIV
C.3 Research Question 4 . . . XV
C.3.1 AUCECEXAM . . . XV

**D Model Diagnostics** **XVII**
D.1 RQ1 Diagnostics . . . XVII
D.1.1 EXAM Model 1 . . . XVII
D.1.2 EXAM Model 2 . . . XIX
D.1.3 AUCECEXAM Model 1 . . . XXI
D.1.4 AUCECEXAM Model 2 . . . XXIII
D.2 RQ2 Diagnostics . . . XXV
D.2.1 EXAM Model 1 . . . XXV
D.2.2 EXAM Model 2 . . . XXVII
D.2.3 AUCECEXAM Model 1 . . . XXIX
D.2.4 AUCECEXAM Model 2 . . . XXXI
D.3 RQ4 Diagnostics . . . XXXIII
D.3.1 EXAM Model 1 . . . XXXIII
D.3.2 EXAM Model 2 . . . XXXV
D.3.3 AUCECEXAM Model 1 . . . XXXVII
D.3.4 AUCECEXAM Model 2 . . . XXXIX
D.3.5 EXAM25 . . . XLI
D.3.6 EInspect25EXAM . . . XLIII

# List of Figures


2.1 Google Weighting Function . . . . . 8
2.2 Linear and Flat Weighting Function . . . . . 9
2.3 Difference between timestamp and index-based commit weighting . . 10

3.1 A High Level Overview of the Process used for this Thesis . . . . . 16
3.2 Cost-Effectiveness Curves for a Random and a More Optimal Algorithm 18
3.3 Origin, depth and future on the commit history . . . . . 24
3.4 varsel_plot and mcmc_areas plot for EXAM prediction in RQ1 and RQ2 . . . . . 28
3.5 Curves for a wide and a narrow prior . . . . . 30
3.6 Curves for Normal(0, 0.5) priors . . . . . 30

4.1 Densities of AUCECEXAM and AUCECDENSITY with 2 SD intervals 34
4.2 Density of EXAM and correlation of EXAMF and hdMaxLOCEXAM 34
4.3 EXAM and AUCECEXAM results for Linespots for different Languages and Domains . . . . . 35
4.4 Posteriors and marginal effects for the weighting functions in Model 4.1 37
4.5 Contrasts between the different weighting functions based on posterior predictions in Model 4.1 . . . . . 38
4.6 Posteriors and marginal effects for the weighting functions in Model 4.2 39
4.7 Contrasts between the different weighting functions based on posterior predictions in Model 4.2 . . . . . 40
4.8 Posteriors of the coefficients . . . . . 42
4.9 Posteriors and contrast between the different time versions based on posterior predictions in Model 4.3 . . . . . 42
4.10 Posteriors of the coefficients . . . . . 44
4.11 Posteriors and contrast between the different time versions based on posterior predictions in Model 4.4 . . . . . 44
4.12 Densities of hdMaxLOCEXAM and EXAMF for Linespots samples . 45
4.13 EXAM25, precision and recall densities for Linespots samples . . . . 46
4.14 Posteriors of Bugspots . . . . . 48
4.15 Marginal effects and contrast based on posterior predictions for both algorithms in Model 4.5 . . . . . 49
4.16 Posteriors of Bugspots . . . . . 50
4.17 Marginal effects and contrast based on posterior predictions for both algorithms in Model 4.6 . . . . . 51
4.18 Posteriors of Bugspots . . . . . 52

4.19 Marginal effects and contrast for both algorithms based on posterior predictions in in Model 4.7 . . . 53
4.20 Posteriors of Bugspots . . . 54
4.21 Marginal effects and contrast based on posterior predictions for both algorithms in Model 4.8 . . . 55
4.22 Mean absolute rank difference of faults between Bugspots and Linespots 56

5.1 Influence of Source and Choice on EXAM results for Bugspots and Linespots . . . 64

B.1 varsel_plot and mcmc_areas plot for AUCECEXAM prediction in RQ1 and RQ2 . . . VII
B.2 varsel_plot and mcmc_areas plot for EXAM prediction in RQ4 . . . VIII
B.3 varsel_plot and mcmc_areas plot for EXAM prediction in RQ4 . . . IX
B.4 varsel_plot and mcmc_areas plot for EXAM25 prediction in RQ4 . . X
B.5 varsel_plot and mcmc_areas plot for EInspect25EXAM prediction in RQ4 . . . XI

D.1 Rhat and n_eff for Model C.1 . . . XVII
D.2 Acceptance and divergent transitions for model C.1 . . . XVIII
D.3 Step size and tree depth for model C.1 . . . XVIII
D.4 Energy and trace plots for model C.1 . . . XIX
D.5 Rhat and n_eff for model 4.1 . . . XIX
D.6 Acceptance and divergent transitions for model 4.1 . . . XX
D.7 Step size and tree depth for model 4.1 . . . XX
D.8 Energy and trace plots for model 4.1 . . . XXI
D.9 Rhat and n_eff for Model C.2 . . . XXI
D.10 Acceptance and divergent transitions for Model C.2 . . . XXII
D.11 Step size and tree depth for Model C.2 . . . XXII
D.12 Energy and trace plots for Model C.2 . . . XXIII
D.13 Rhat and n_eff for Model 4.2 . . . XXIII
D.14 Acceptance and divergent transitions for Model 4.2 . . . XXIV
D.15 Step size and tree depth for Model 4.2 . . . XXIV
D.16 Energy and traec plots for Model 4.2 . . . XXV
D.17 Rhat and n_eff for Model C.3 . . . XXV
D.18 Acceptance and divergent transitions for model C.3 . . . XXVI
D.19 Step size and tree depth for model C.3 . . . XXVI
D.20 Energy and trace plots for model C.3 . . . XXVII
D.21 Rhat and n_eff for model 4.3 . . . XXVII
D.22 Acceptance and divergent transitions for model 4.3 . . . XXVIII
D.23 Step size and tree depth for model 4.3 . . . XXVIII
D.24 Energy and trace plots for model 4.3 . . . XXIX
D.25 Rhat and n_eff for model C.4 . . . XXIX
D.26 Acceptance and divergent transitions for model C.4 . . . XXX
D.27 Step size and tree depth for model C.4 . . . XXX
D.28 Energy and trace plots for model C.4 . . . XXXI
D.29 Rhat and n_eff for model 4.4 . . . XXXI

D.30 Acceptance and divergent transitions for model 4.4 . . . . . XXXII
D.31 Step size and tree depth for model 4.4 . . . . . XXXII
D.32 Energy and trace plots for model 4.4 . . . . . XXXIII
D.33 Rhat and n_eff for Model C.5 . . . . . XXXIII
D.34 Acceptance and divergent transitions for model C.5 . . . . . XXXIV
D.35 Step size and tree depth for model C.5 . . . . . XXXIV
D.36 Energy and trace plots for model C.5 . . . . . XXXV
D.37 Rhat and n_eff for model 4.5 . . . . . XXXV
D.38 Acceptance and divergent transitions for model 4.5 . . . . . XXXVI
D.39 Step size and tree depth for model 4.5 . . . . . XXXVI
D.40 Energy and trace plots for model 4.5 . . . . . XXXVII
D.41 Rhat and n_eff for model C.6 . . . . . XXXVII
D.42 Acceptance and divergent transitions for model C.6 . . . . . XXXVIII
D.43 Step size and tree depth for model C.6 . . . . . XXXVIII
D.44 Energy and trace plots for model C.6 . . . . . XXXIX
D.45 Rhat and n_eff for model 4.6 . . . . . XXXIX
D.46 Acceptance and divergent transitions for model 4.6 . . . . . XL
D.47 Step size and tree depth for model 4.6 . . . . . XL
D.48 Energy and trace plots for model 4.6 . . . . . XLI
D.49 Rhat and n_eff for model 4.7 . . . . . XLI
D.50 Acceptance and divergent transitions for model 4.7 . . . . . XLII
D.51 Step size and tree depth for model 4.7 . . . . . XLII
D.52 Energy and trace plots for model 4.7 . . . . . XLIII
D.53 Rhat and n_eff for model 4.8 . . . . . XLIII
D.54 Acceptance and divergent transitions for model 4.8 . . . . . XLIV
D.55 Step size and tree depth for model 4.8 . . . . . XLIV
D.56 Energy and trace plots for model 4.8 . . . . . XLV

# List of Tables


3.1 Project property overview. Src = Source, Ch = Choice, A = Author, P = Past, R = Random . . . . . 23
3.2 loo_compare result for research question 1 EXAM models . . . . . 32

4.1 Summary of exploration, rounded to 3 decimals or integer, 0 used as lower limit . . . . . 35
4.2 Fixed effects of models C.1 and 4.1 rounded to 3 significant digits . . 37
4.3 loo_compare output for models C.1 and 4.1 . . . . . 37
4.4 Contrast between weighting functions in Model 4.1 rounded to 3 significant digits . . . . . 38
4.5 Fixed effects of models C.2 and 4.2 rounded to 3 significant digits . . 38
4.6 loo_compare output for models C.2 and 4.2 . . . . . 39
4.7 Contrast between weighting functions in Model 4.2 rounded to 3 significant digits . . . . . 39
4.8 Fixed effects of models C.3 and 4.3 rounded to 3 significant digits . . 41
4.9 loo_compare output for models C.3 and 4.3 . . . . . 41
4.10 Contrast between time versions in Model 4.3 rounded to 3 significant digits . . . . . 41
4.11 Fixed effects of models Model C.4 and 4.4 rounded to 3 significant digits . . . . . 43
4.12 loo_compare output for models C.4 and 4.4 . . . . . 43
4.13 Contrast between time versions in Model 4.4 rounded to 3 significant digits . . . . . 44
4.14 Mean, Median, 2 and 4 standard deviation intervals for the Linespots sample . . . . . 45
4.15 Fixed effects of models C.5 and 4.5 rounded to 3 significant digits . . 47
4.16 loo_compare output for models C.5 and 4.5 . . . . . 47
4.17 Contrast between algorithms in Model 4.5 rounded to 3 significant digits . . . . . 48
4.18 Fixed effects of models C.6 and 4.6 rounded to 3 significant digits . . 50
4.19 loo_compare output for models C.6 and 4.6 . . . . . 50
4.20 Contrast between algorithms in Model 4.6 rounded to 3 significant digits . . . . . 50
4.21 Fixed effects of Model 4.7 rounded to 3 significant digits . . . . . 52
4.22 Contrast between algorithms in Model 4.7 rounded to 3 significant digits . . . . . 53

4.23 Fixed effects of Model 4.8 rounded to 3 significant digits . . . . . 54
4.24 . . . . . 55
4.25 Mean, Median, 2 and 4 SD Intervals rounded to 3 significant digits or integers . . . . . 56

A.1 Project List with Indices . . . . . IV
A.2 Project Drawing . . . . . V

# 1

# Introduction

Both individual software systems, as well as the field of software engineering, have become larger and more complex over the years and, according to Nosek and Palvia [1], maintenance costs were estimated to make up most of the overall cost of software systems. This makes the idea of fault prediction more relevant than ever. After all, an ideal fault prediction algorithm could point developers to all the code that contains faults and thus reduce the time needed to locate them immensely. One could also use it to focus on testing and auditing efforts on those areas that contain the most faults, thus making more efficient use of those efforts.

There have been a lot of approaches to predicting faults in software systems in the past. Ranging from simple source code based metrics like lines of code (LOC), class sizes for object-oriented (OO) languages and complexity measures to more sophisticated metrics that are based on the entropy of changes, process derived metrics or even socio-technical information. While some of these metrics perform better than others, most of them show poor performance on their own [8], [13], [26]. With the new wave of machine learning, some models for fault prediction combine multiple metrics to improve their predictions [7], [8], [14].

Linespots is a novel approach to the 'past-faults' fault prediction metric, developed by me in 2016 as part of my bachelor thesis [18]. It is based on Bugspots [9], [15], a fault prediction algorithm by Google that was inspired by the work of Rahman *et al.* [12] and improves upon it. Bugspots is based on a simple idea: If a file contained faults in the past, there is a higher probability that it will contain more faults in the future. This is based on the assumption that whatever caused the past faults might have lead to more faults that just have not surfaced. The algorithm can then simply rank files by the number of faults they contained in the past. To this the Google engineers added a weighting function, to increase the impact more recent faults have on the ranking. This prevents files that contained faults in the past but that the team went over and fixed from staying at the top of the ranking for too long. Linespots took this idea and applied it to individual lines in a project, instead of whole files. So, while Bugspots reports a list of files, ranked by the weighted faults in the past, Linespots reports a list of lines, using the same ranking. This offers a higher resolution of the results and leads to fewer false positives.

The integration of Linespots into the landscape of fault prediction metrics requires an analysis of its performance as well as a comparison of the predicted faults with other algorithms. For someone who is building a model for fault prediction, it is

valuable to know if a metric offers additional prediction power to the model. If the new metric predicts the same faults as another metric that is already in the model, there would be little gained by adding the new one.

## 1.1 Problem and Purpose

When building a model for fault prediction, one has to choose which metrics to include. There is a trade-off in this decision, as more metrics can mean more information to the model to make better predictions, but more metrics also means a more complex model, which takes more computational power, provides lower understandability, increases the risk of overfitting and requires more effort/cost to use and maintain.

The term feature selection is used in statistics and machine learning to describe the process of choosing only those features, e.g., metrics that are most valuable for our model. This is done to restrict model complexity. To make the initial decision about what metrics to include, an engineer needs information about the pros and cons of each metric. What information they offer, how expensive they are to compute, what data they need, the cost of collecting that data, and if they are complementing each other or merely present proxies for the same information.

As Linespots is a recently proposed version of the 'past-faults' metric family, some of these properties are not yet known or well understood as they might differ from other metrics in this family. This poses the problem that someone doing fault prediction cannot know if Linespots is the right metric for the given scenario or not. Furthermore, these questions are also relevant for other researchers, as it is unclear if Linespots is a metric worth investigating further at this point.

One purpose of this thesis is to answer some of the most common questions we got regarding Linespots in the past.

### 1.1.1 Research Question 1: What kind of weighting function produces the best results for Linespots?

The first question is related to the weighting function that Lewis *et al.* [15] proposed for Google. They mention that the exponential weighting function Google uses is tuned for their projects and that other weighting functions might work better for other projects. This leads to the first research question:

### 1.1.2 Research Question 2: How does index-based age calculation influence the predictive performance of Linespots compared to time stamp based age calculation?

Another question we received was regarding the way the age of commits is calculated for the weighting function. For Bugspots, the age of a commit is normalized based on the UNIX timestamps of the oldest and youngest commits that are analyzed. The proposed alternative to this is to normalize the age of commits based on their position in history, regardless of actual passed time. The underlying idea is that some projects might have slower or faster development cycles and that the time between commits might skew the impact even if they are consecutive. This leads to the second research question:

### 1.1.3 Research Question 3: What is a good cut-off-point to turn Linespots into a classifier?

While ranked results support some use cases, such as prioritizing efforts, tools like linters usually work on 'good or bad' dichotomies. The investigation of Linespots capabilities as a classifier requires an analysis of the performance for different cut-off points. All lines before that point are proposed as faulty, all lines after it as not-faulty. It might also be useful to find a procedure for how one could derive a cut-off-point for their project. Answering this question could also be useful to allow comparison between Linespots and other classifying metrics:

### 1.1.4 Research Question 4: What is the prediction performance of Linespots compared to Bugspots?

The first three research questions can be seen as improving the initial Linespots implementation. After identifying the parameters that give the best results, we want to investigate how Linespots performs in this improved state to update the comparison done in the bachelor thesis and allow for a better estimation of where it lies in the landscape of prediction metrics.

### 1.1.5 Research Question 5: Do Bugspots and Linespots predict faults in the same order?

Finally, we want to investigate the faults that Linespots predicts in comparison to the faults other metrics predict to see if they are similar or if we can find differences. This is interesting as too similar results could indicate that using both metrics combined does not offer additional benefits.

# 2

# Background

In 2011, Rahman *et al.* [12] discovered that a very simple ranking of source files based on the number of past-faults in those files was almost as useful for predicting future faults as the more complex BugCache algorithm they were developing. Based on this, Lewis and Ou [9] proposed their Bugspots algorithm that added a weight decay for older faults to that basic idea. Both of these algorithms are proposing whole files for inspection, something that comes with a lot of overhead of non-faulty lines.

The above was the basis for my bachelor's thesis [18] that applied the past-faults idea to individual lines of code and showed promising improvements in hit density.

## 2.1 Fault Prediction

Fault prediction is the process of using statistical techniques, including machine learning, to predict where faults might occur in the code of a project, usually before any failure. The process of prediction can classify parts of the code as faulty or not faulty, can flag fault-inducing modifications to the projects or rank code elements by their likelihood of containing faults. Common approaches for fault prediction are based on static code metrics like size and complexity and object-oriented metrics like coupling, cohesion and inheritance. More recently, process metrics that include development process information, are being developed. This includes the past faults class that Linespots is part of, as well as concepts like code churn and socio-technical networks [14].

### 2.1.1 Fault Localization

Fault localization is similar to fault prediction, in that both try to point developers to problematic code. The main difference is that fault localization is used after a failure, which means it can utilize information that can only be collected by running the program, like testing information or dynamic analysis. So fault localization tries to localize a fault that led to failure, while fault prediction is more of a preventive measure, trying to predict faults before they lead to failures. While the two are

considered as two different things, they do produce very similar, if not the same, results. Namely a proposal of code that is likely to contain faults.

### 2.1.2 Granularity

The granularity of an algorithm describes the type of code unit it proposes as more or less fault-prone. Common granularities are binaries, modules, files, methods, classes and statements [4], [26] There are advantages and drawbacks for different granularities. Files, for example, are easy to work with and language-agnostic but proposing whole files as faulty is not very precise. Methods or functions, classes and statements are closer to the way developers work with the code in most languages and more precise than files, but their handling has to be implemented individually for each language. Lines lack the context that language constructs offer but are agnostic of programming language. This makes them a compromise between the easy to handle and language agnostic file granularity and the higher precision that language constructs can offer. For that reason, Linespots works on the line granularity.

### 2.1.3 Result Types

The two common ways of presenting results are as a binary classification of faulty and not faulty, and as a ranked list with elements that are most likely to contain faults at the top. The faulty elements of classification can also be ranked or grouped by severity as well. While the classification of elements into faulty or not faulty paints a clear image of certainty, it lacks the ability to nuance the results. If no further prioritization is offered, the number of elements in the faulty class can be too high for practical purposes. A ranked list allows the user to set their thresholds as needed but requires more competency in interpreting the position of elements in the list.

### 2.1.4 Applicability

The scenario that we encountered most often for how to use fault prediction in a development workflow was for code inspection [12], [15] and code testing [26]. The results of the fault prediction would tell developers what elements have a higher risk to contain faults and thus prioritize their efforts into those elements. Lewis *et al.* [15] found that this type of information does not support developers during code inspection and instead proposed to use the results of fault prediction to focus testing efforts.

## 2.2 Past Faults

As mentioned before, the initial idea came from Rahman *et al.* [12] when they tried to compare their newly developed fault prediction algorithm, FixCache, to a

comparatively simple algorithm that ranked all files in a project by the number of faults the files contained in the past. The simple idea was that more faults in the past would lead to more faults in the future. The argument is that whatever lead to the past faults, such as complicated code, complex algorithms or a tired developer could lead to more faults in the same area. This algorithm performed surprisingly well and they concluded that "The naive model is actually about the same utility for inspections, under the $\text{aucec}_{20}$ measure, as the best possible setting for FixCache." Rahman *et al.* [12].

This meant that without further research into FixCache the naive model would be the better choice.

### 2.2.1 Identification of Faults

We are not aware of a technique that would allow us to know all existing faults in a software system, which makes testing predictions difficult. In the past, researchers have used the fixed faults, and sometimes the identified faults, as proxies. Fixed faults are easy to use, as they show what part of the code contained the fault and how to correct it. In addition to the fixed faults, information from a bug tracker can be used to identify faults that have been located but not yet fixed. This, however, requires information about the location of the fault inside of the bug tracker, which might not be available on the necessary granularity level. As Linespots works lines, we follow the approach of using fixed faults as an indicator of existing faults. Any information about fault location that might exist in a bug tracker, will most likely not be detailed enough to include it into the algorithm.

Linespots currently identifies fault fixing commits by parsing the commit message of each commit and searching for a given regular expression (regex). These regular expressions usually are indicators like 'Fix' or 'Bug'. As these indicators depend on commit message rules of each project, they have to be set on a per-project base. Many projects do not have clear rules for differentiating commit messages for fault fixing commits and other types of commits, which makes them unsuited for usage with Linespots. Another downside of using only the commit message for identification was shown by D'Ambros *et al.* [8], who found that simply doing this kind of string matching decreases the accuracy of the predictions and propose checking the matches with existing bug databases. This would be possible for most of the projects in this thesis as they use the GitHub issue tracker. However, this was not included due to time constraints, as we did not find a readily available implementation of such a lookup and had to pass on this due to time limitations.

## 2.3 Bugspots

Based on the findings of Rahman *et al.* [12], Lewis *et al.* [15] proposed an improved version of the initial idea. They found that the algorithm was ranking files too high that contained a lot of faults in the past but were fixed since then. So files would

never sink to the bottom of the ranking, even if they had been fixed for a long time. They introduced "Time-Weighted Risk"[15], a weight decay for older faults. This meant that more recent faults influence the ranking stronger than older ones. So files that contained a lot of faults in the past receive lower ranks if they don't contain faults in more recent time. Figure 2.1 shows the weighting function used by Google [9]. The x-axis shows the normalized age of a commit between 0 and 1. 1 being the newest commit and 0 being the oldest one. The y-axis shows by how much the score of a file increases if a commit that modified the file fixes a fault.

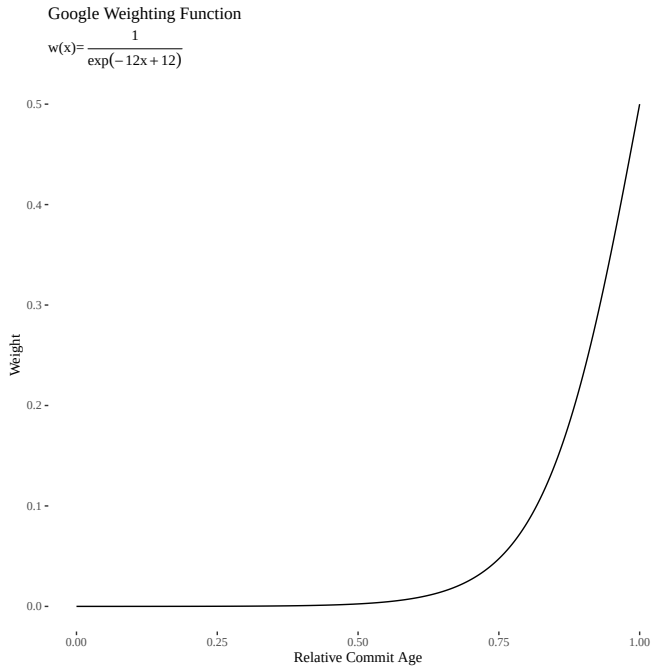
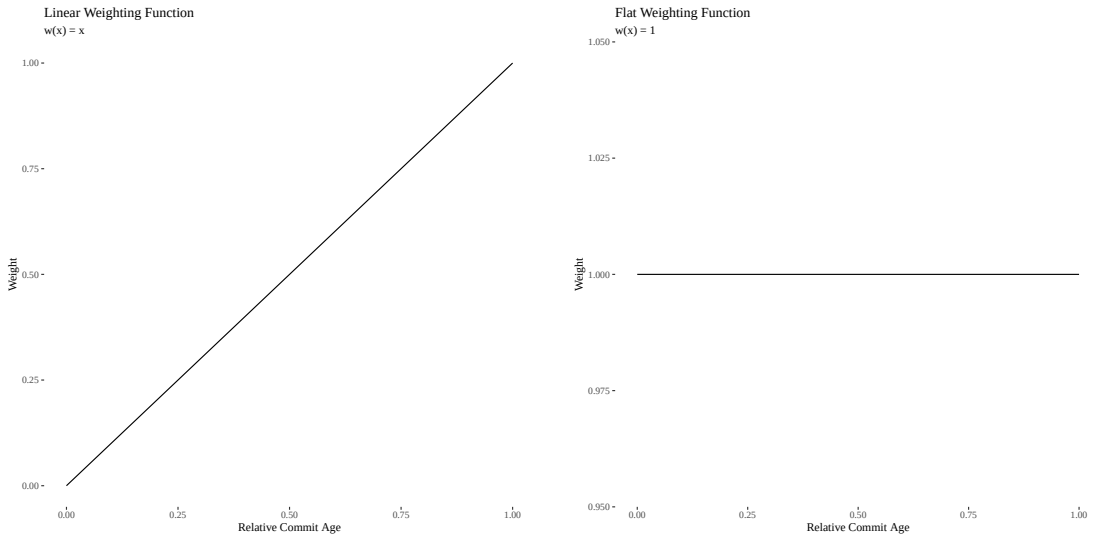



**Figure 2.1:** Google Weighting Function

### 2.3.1 Weighting Functions

Lewis *et al.* [15] argue that their weighting function should ideally be dynamically adapted to count commits in a 6 to 8 month window before they become inconsequential based on their domain knowledge and experience for their projects. This introduces the idea of finding the right weighting function for different projects. As Lewis *et al.* [15] did not share their process and intermediate results of coming up with a weighting function, we can only speculate about the influence on performance different weighting functions can have. For this reason, we compare the weighting function proposed by Lewis *et al.* [15] with a linear and a flat weighting function as shown in Figure 2.2.

While Lewis *et al.* [15] propose a more generic weighting function in their paper, we keep the w=12 they proposed. Changing w, however, seems to be a very powerful tool for tuning the performance of Bugspots to a certain project. This is something that might be interesting for future work. The flat weighting function in Figure 2.2b makes the Bugspots algorithm perform as if there is no weight decay at all.




**(a)** Linear **(b)** Flat

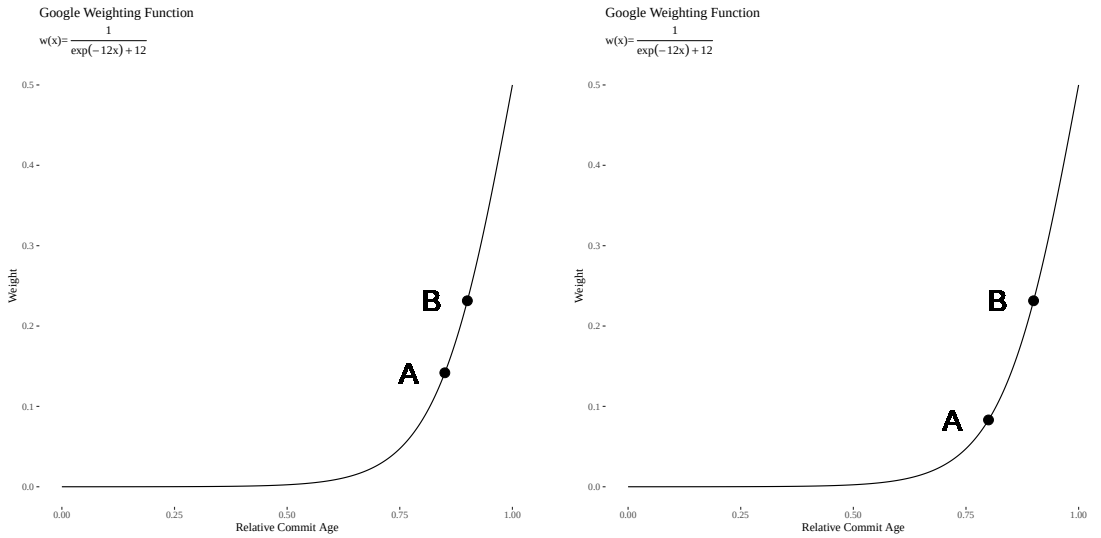
**Figure 2.2:** Linear and Flat Weighting Function

All commits have the same influence on the score of a file, regardless of their age. This is meant as a control for the claim of Lewis *et al.* [15] that the use of a weight decay improves performance. Finally, we test a linear weighting function shown in Figure 2.2a that works as a middle ground between the initial flat weighting function and Lewis *et al.* [15] exponential weighting function. As the linear weighting function does offer weight decay, it could give insight into why the weighting function improves performance. The linear function performing closer to the flat one could mean that it is the very steep decline in relevancy that the exponential function has that is responsible for the improved performance. It would also support the argument that tuning the weighting function to a project might improve performance, as not any kind of weight decay would help. If the linear function performs closer to the exponential one, it might be that just any weight decay is useful and tuning weighting functions could have less impact than claimed by Lewis *et al.* [15].

### 2.3.2 Relative Commit Age

Directly connected to the weighting function is the idea of the relative commit age. Lewis *et al.* [15] do not explain how they calculate the normalized commit age but existing implementations [28] base the age calculations of commits on the commit timestamps. While this seems straight forward initially, there may be some drawbacks to this.

The main downside that came up a lot during our past work [18] is that the physical time between two commits might not represent the time to develop something properly and might influence performance on projects that have faster or slower development or commit cycles. Another problem that came up during the development of the current Linespots reference implementation is that due to capability to




**(a)** Timestamp based commit weights **(b)** index-based commit weights

**Figure 2.3:** Difference between timestamp and index-based commit weighting

rewrite history in Git, timestamps might be unreliable and in some corner cases, the HEAD commit of a branch can be older than a commit preceding it in the branch structure.

The alternative to time stamp based age calculation is the index-based age calculation. For this, all commits are indexed in their order on the Git branch and the relative age is calculated based on the position of a commit relative to the highest index. Figure 2.3 shows an example of two commits on the Google-weighting-function, one with the timestamp-based age calculation and one with the index-based time calculation. As can be seen in the example, the weight of commit A changes depending on how the relative age is calculated. This can happen if the distance in time between the two commits is not the same as the distance in indices. As this difference can influence the performance, we analyze the difference in performance between the two versions and try to find any patterns that emerge, like certain domains working better with one or the other version.

## 2.4 Linespots

The Linespots algorithm is based on Bugspots but instead of scoring and predicting whole files, it works on a line level. This means that each line in a project has a score attached to it and that the output is the list of all lines in a project, ranked by highest line score.

This change increases the complexity of the implementation between Bugspots and Linespots. While Bugspots can simply list all files that were modified during a commit and increase their scores in case the commit was a fix, Linespots has to parse the diffs attached to the commit, track added and removed lines, and handle

a lot more corner cases.

### 2.4.1 Determining Faulty Elements

After identifying a fix-commit, it is necessary to identify the faulty lines from that commit. Pearson *et al.* [21] define faulty lines as those lines that get removed or changed during a fix-commit. If only new lines are added, they propose to tag the line immediately following the newly added lines as faulty. This corner case is not part of the implementation that was used for the evaluation but from our observation most fixing commits contain removed lines so the impact of this should be small. Although these findings are from the area of fault location, they apply to fault prediction as we want to locate the past faults to base our predictions on them.

### 2.4.2 Scoring Lines

With Bugspots all files start with a score of 0, as new files cannot contain faults before they exist. This concept does not work for lines however, as GIT only works with added and removed lines. To allow line scores to increase over time if they are part of fixing commits, new lines start with the average score of the hunk they are added to. Hunks are created by GIT when displaying diffs and show part of a file with the corresponding line changes. Hunks give some unchanged lines before and after the first and the last added or removed line as context, so changes to one file can be displayed through multiple hunks if far enough apart. The choice for using hunks as a measure of the locality was made out of convenience.

### 2.4.3 Implementation Changes

The initial implementation for Linespots was proposed in 2016 [18] and evolved since then. Going through two rewrites over the years, it is easier to work with today and can be used on more projects than the initial implementation. The change that is most relevant for this thesis is the support of merge commits. While the original implementation could not be used on projects that use merges in their workflow, due to being unable to parse merge commits, the current implementation has full support for merge commits with arbitrary numbers of parents. This change has increased run times for Linespots. While we did not thoroughly test for this, we assume it to take one to two orders of magnitudes longer. The run times, however, were not a concern for this thesis and we presume that it can be lowered by around at least an order of magnitude with an improved implementation.

Another important addition to the implementation is the support for different file encodings as this again allows Linespots to be run on a wider range of projects. File encodings are both relevant for parsing the diffs and for counting the number of lines in the whole project, which is needed for the evaluation.

Besides the two big changes, several Git corner cases and different file encodings and are now properly handled. Many of these stem from the fact that Linespots currently parses the diffs manually and as GIT is a complex system, there are many cases that we encountered only once during our work on Linespots. All these cases that are now cared for again allow Linespots to run on more projects without failure and increase the reliability of the results.

### 2.4.4 Predictive Performance

In our past work [18], we found that the inspection of the highest-ranked 5% LOC resulted in 3.91% faults found for Bugspots and 47.7% faults found for Linespots on average. This result showed the potential for big improvements in predictive performance but was lacking rigour in its methods. This is why this thesis will look further into the performance of Bugspots and Linespots.

## 2.5 Relevance to Practice

To further the knowledge in a field, it is important to discuss how the results of this thesis might influence future research on fault prediction and the work of practitioners in software engineering.

Bugspots or the initial algorithm by Rahman *et al.* [12] are usually used as a baseline comparison in the field of fault prediction. The algorithm is rather simple and the run time is short as well, so any more complex algorithm for fault prediction should improve on the results that Bugspots delivers. If Linespots would come out to improve the predictive performance of Bugspots, it could serve as a new baseline for fault prediction in the future. As long as Linespots' performance is at least similar to Bugspots', it can also serve as a comparison algorithm for other algorithms that work on a finer granularity than files. This would remove the need to map Bugspots' results to finer granularities. Finally, Linespots might set the foundation for more specialized versions of the past faults algorithm family that track language constructs instead of lines. While these specialized versions would not be language agnostic any more, they could offer better performance for the cases they cover. If the performance of Linespots should prove to be below the performance of Bugspots, this thesis would add evidence to the idea that there is not much to gain from improving on the idea of Bugspots and efforts should be focused on other areas.

For practitioners, the existence of an open implementation might make the usage of Linespots easier than other algorithms. As it is language agnostic, there are no limitations on what projects can use it and the idea of the algorithm is simple so that it can be understood by developers without a background in information theory. All of these could put Linespots in the position of being an entry point to fault prediction for one's project if the performance ends up being sufficient. Anecdotally, when discussing the general idea of fault prediction and Linespots with visitors of developer conferences we received positive feedback for the idea of Linespots. We

talked to visitors of the Europython 2016 and 2017, PyConDE 2017, FOSSDEM2017 and Linuxwochen Wien 2016 and 2017 conferences and presented the results of our past work [18] in two talks at Europython 2017 and Linuxwochen Wien 2017. In particular, the fact that it is language-agnostic and has a free and open implementation available was well received. As most of the benefits of Linespots also apply to Bugspots, this enthusiasm could just be based on a lack of knowledge about the field of fault prediction or the fact that fault prediction over-promises as indicated by Lewis *et al.* [15].

# 3

# Methods

## 3.1 Objectives

There are five research questions that we want to investigate. To enable us to answer them, we first have to clarify what we want to measure and how it will help answer our questions. The proposed metrics are discussed in detail in the next section.

### 3.1.1 Predictive Performance

Three of our research questions are related to predictive performance. To recapitulate:

**RQ1:** What kind of weighting function produces the best results for Linespots? For this question, we have to gather results from Linespots using the three different weighting-functions and compare them against each other.

**RQ2:** How does index-based age calculation influence the predictive performance of Linespots compared to time stamp based age calculation? For this question, we have to gather results from Linespots using the two different time-versions and compare them against each other.

**RQ4:** What is the prediction performance of Linespots compared to Bugspots? For this question, we have to gather results from Linespots and Bugspots and compare them against each other.

### 3.1.2 Optimal Cut-Off Point

**RQ3: What is a good cut-off-point to turn Linespots into a classifier?** For research question 3, we want to investigate the ideal cut-off point for Linespots to turn it into a binary classifier. As this question does not have a unique correct answer and depends on the needs of the user, we will have to analyze the performance of Linespots at different possible cut-off points. This would allow users to make an informed decision knowing the tradeoffs of different cut-off points.

### 3.1.3 Ranking Comparison

**RQ5: Do Bugspots and Linespots predict faults in the same order?** For research question 5, we want to investigate, if Bugspots and Linespots rank the lines in a way that lead to the same ordering of predicted faults.

## 3.2 Experimental Design and Preparation

The easiest design to test the impact of different factors on a given outcome is a full factorial design [2]. The usual downside to this kind of design is the amount of data that is needed, which usually increases the needed time and cost of an experiment. As we expected no further cost besides our own time and electricity bill, cost was not a concern for this thesis. And we were hoping to decrease the amount of manual labour by building a fully automated evaluation suite.

The next step in our design was to find metrics, that are able to measure the things we want to measure for our objectives in Section 3.1. We describe all the metrics that we use and how they can help answer the research questions below in Section 3.3.

After deciding on the metrics, we needed a dataset to run the algorithms on and calculate the metrics from the results. we describe the process of building the dataset in Section 3.4. As part of the dataset, we also need validation data to calculate the metrics. Section 3.5 describes the process of how we construct the validation data.

Figure 3.1 shows a high level overview of the process we used for this thesis.

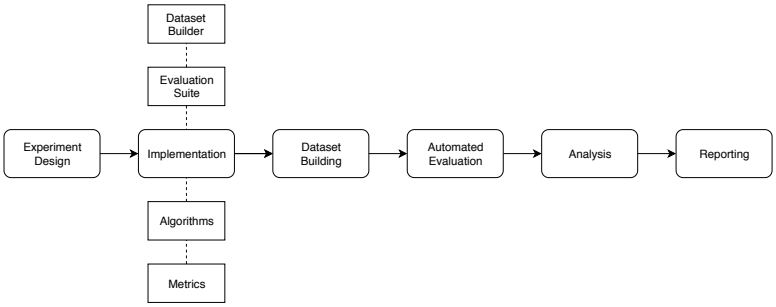



**Figure 3.1:** A High Level Overview of the Process used for this Thesis

## 3.3 Metrics

The evaluation of our research questions requires metrics to compare the performance of different algorithms or the influence of parameters on the performance of an algorithm. As with most things, there is no one and true metric that incorporates everything that we would like to measure. Instead, all available metrics have strengths and drawbacks. In this section, we will explore metrics that have been

used in prior work, discuss how they should be used and how to interpret them. We also argue for our choices of metrics and how they can help answer the questions we posed earlier.

### 3.3.1 Cost-Effectiveness

When evaluating the predictive performance of Linespots, we are interested in how good the ranking of lines is. A more traditional measure for this is the receiver operating characteristic (ROC) but it has received criticism for not taking cost-effectiveness into account [7] and assigning "different misclassification cost distributions for different classifiers." [6]. The cost-effectiveness is less of a concern with Linespots, as it works on a line granularity instead of a class one. However, we still prefer the cost-effectiveness measure that Arisholm *et al.* [7] propose over the ROC, as it is easier to interpret and can be adapted to include information about severity and risk of faults if known. While Hand [6] also proposes an alternative to the ROC, the H measure, we did not include it in this thesis due to time constraints.

The cost-effectiveness initially should solve the problem that simply proposing the largest files or classes could inflate the performance of an algorithm, as they did not take into account the size of the proposed elements. Inspecting the largest files in a project might result in some found faults but will also take a long time. The proposed solution was a curve that plots the spent effort on the x-axis and the received benefit on the y-axis. This is called the cost-effectiveness curve (CEC). If no further information about the complexity of the code and the severity of the faults is known, this can be done by having the percentage of LOC on the x-axis and the percentage of faults on the y-axis. This assumes that inspecting each line is approximately equally hard and all faults to be equally severe. The cost-effectiveness curve then results from plotting the percentage of faults found, if one were to inspect all proposed lines up to x.

Figure 3.2 shows an example of two CECs. A trivial algorithm that randomly ranks lines and a more optimal algorithm like Linespots. The higher the CEC is above the y=x line of the trivial algorithm, the better. This can be summarized by the area under the CEC (AUCEC). So the baseline performance of a random algorithm would be an AUCEC of 0.5 and a perfect algorithm would approach 1. It could never reach one, as that would require all faults to be found with 0 inspected lines.

### 3.3.2 Precision and Recall

Precision and recall are both derived from the confusion matrix and common metrics used to measure the performance of classifiers. We will use an example project where 5 out of 100 elements contain faults to explain the idea of both and give a baseline performance for a naive algorithm.

**Precision** describes the proportion of truly faulty elements out the ones that the

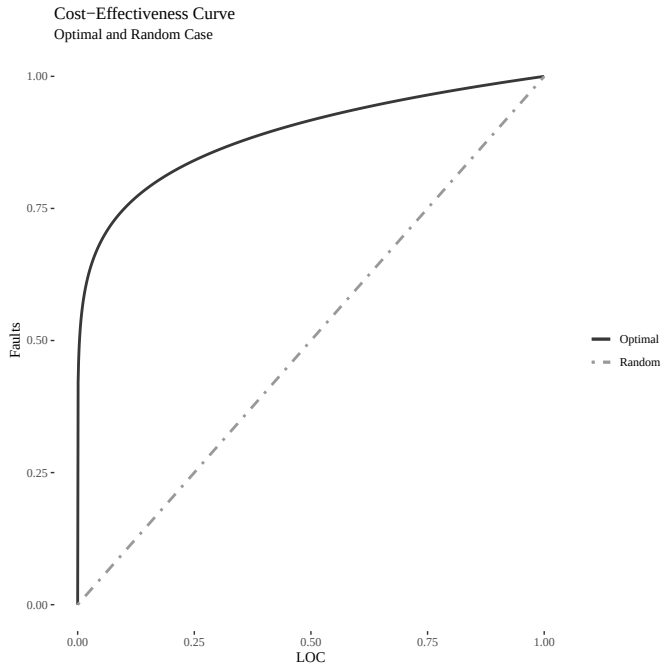



**Figure 3.2:** Cost-Effectiveness Curves for a Random and a More Optimal Algorithm

classifier classified as faulty:

$$\frac{\text{True Positive}}{\text{True Positive + False Positive}}$$

Using a trivial algorithm that classifies all elements of the sample project as faulty, it would have a precision of 5 / (5 + 95) = 0.05 as only five out of the 100 elements it claimed to be faulty truly are faulty.

**Recall** describes the proportion of correctly classified faulty elements out of all faulty elements:

$$\frac{\text{True Positive}}{\text{True Positive + Fase Negative}}$$

An algorithm that classifies all elements into one class can either achieve a recall of 0 or 1. An algorithm that randomly classifies elements into either class would achieve a mean recall of 0.5.

Precision and recall often have an inverse relationship, as increasing one can lead to decreasing the other. For this reason, they are usually discussed in combination and while ideally, both would be 1, one might be more important than the other in the context of the classification. For example, a high precision might be needed for developers to accept a tool, as it means that there are few false alarms. There might however be projects, where missing a fault is worse, in which case a higher number of false alarms might be acceptable to ensure a higher number of faults discovered. This type of tradeoff can be solved by setting one of the two metrics to a necessary value, for example, a minimal recall of 0.9, and then optimizing the other metric under this condition.

### 3.3.3 EXAM

Proposed by Wong *et al.* [5], it is the percentage of elements that has to be inspected until finding a faulty element, averaged across all elements. [26] A lower EXAM score would mean that the faulty elements are ranked higher and thus indicate a better performing algorithm.

We also propose EXAMN, where N denotes the percentile of faults found when inspecting the ranked list. EXAM25, for example, would give the average percentage of elements to inspect to find the first 25 per cent of faults. We use N= 25, 33, 50, 75, and 95 to indicate different points of threshold that might be useful when choosing cut-off points for classification.

### 3.3.4 EXAMF

Based on the idea of the EXAM score, we propose EXAMF as being the percentage of elements that have to be inspected until the first faulty element is found. When collected for different projects and revisions, it can serve as an indicator for a minimum of lines to inspect to get any benefit. When looking at the distribution of EXAMF scores over a sample set, it is possible to propose the upper 95 percentile as a cut-off point, for example, to find at least one fault in 95 per cent of the cases.

### 3.3.5 $\mathbf{E_{inspect}}$@n

Proposed by Zou *et al.* [26] and based on the acc@n metric by B. Le *et al.* [17], it counts the number of faults that were successfully localized within the top n positions of the resultant ranked list [26]. We use the Einspect@10 versions, as Parnin and Orso [10] found that developers will only inspect the first few entries of a ranked list. While Zou *et al.* [26] also use lower n in their study, their results indicate that Bugspots predictive performance is too low to give reasonable results when n is too small. For this reason we keep n=10 for comparability with Zou *et al.* [26].

### 3.3.6 Hit Density

Based on the defect density of Nagappan and Ball [3] and closed bug density of Rahman *et al.* [12] we use hit density [18] to describe the ratio of faults found per lines inspected. Hit Density = We will use the point of maximum hit density as another indicator for a cut-off point, as it presents the point of highest cost-effectiveness.

### 3.3.7 Average Rank Difference

When given two ranked lists containing the same items, but potentially different ranks per list, we can calculate the difference in ranks per item and use the average difference across all items as a measure of how similar the rankings are.

### 3.3.8 Comparing Granularities

As Bugspots and Linespots report their results on different granularity levels, we have to be careful when comparing them. One way to compare them is to transform the file-based results of Bugspots into line-based results, as proposed by Zou *et al.* [26]. We do this by setting each line's score to the corresponding file's score. This results in lists of ranked lines for both Bugspots and Linespots, so all metrics can be calculated equally for both of them.

Depending on the exact way of how metrics are calculated, this kind of transformation can impact the results for Bugspots. We argue that this does not put Bugspots at a disadvantage, as we found that past research had calculated results in a way that files could only be inspected as a whole. By transforming the results to the line granularity metrics like EXAM will show better performance for Bugspots due to the way blocks with the same score are handled. While we do not think this will influence the results a lot, Bugspots might seem to perform better with this kind of transformation than without.

### 3.3.9 Influence of Sorting on Metrics

During the implementation of the metric calculations for this thesis, we noticed that there is more than one way to calculate some of the metrics. In our past work [18], we have sorted the Bugspots and Linespots results first by their score, then by their path and finally their line number. This meant that groups of lines with the same score from the same file would be ordered in the results list, as they would be in their file. When calculating the cost-effectiveness and hit density, such a sorting method imitates a reader inspecting groups of code, following that sorting. So, if a reader were to inspect multiple lines with the same score, they would sort them by their paths and then inspect them top to bottom in their containing files. While this does seem reasonable, it adds assumptions to the reported results that should be communicated, namely the type of sorting that was done. If one were to sort lines with the same score in a different way, the resulting metrics might be different, even though the scores that Linespots calculated are the same.

As an alternative to the path and line number based sorting, we now also report the cost-effectiveness and hit density metrics based on the $E_{inspect}$ values of each fault. This represents a random sorting of all lines in a group with an equal score. While this probably does not represent the inspection behaviour of a developer, it removes the influence of the sorting that is done and can serve as a baseline.

It might be interesting to investigate the influence of different sorting strategies on the performance of Linespots and other ranking algorithms in the future.

## 3.4 Dataset

This section describes the process and choices that went into the creation of the dataset used for the experiment.

### 3.4.1 Sources

With the growth of services like GitHub, GitLab and BitBucket, access to open source projects is easy. This makes open source projects interesting for empirical research, as it is easier to get access to them compared to industry projects. We chose GitHub as our source, as it offers a large collection of open source projects, with over 100 million repositories [37]. There might be a bias in the results due to this, as some communities might choose platforms besides GitHub and the focus on open source projects. As there has not been a direct comparison of the performance of fault prediction algorithms for industry projects versus open source projects, this is a possibility that we cannot rule out. We, however, try to mitigate this risk by choosing a large sample size and projects from many different domains.

### 3.4.2 Sample Size

Li *et al.* [30] found that the average study is done with 11.3 project datasets. They don't mention if the studies drew more than one sample per project, which is commonly done by using different revisions of the same project. Based on this, we decided to use at least 12 projects in this experiment, to have an above-average sample size. The actual sample size would depend on the amount of time available and other factors as described in Section 3.4.3.

Besides these numbers, we considered doing a traditional or Bayesian power analysis to calculate the minimum sample size. This proved hard as we had limited prior information to base our effect sizes on. In addition to that, both are based on p-values or Bayes factors respectively, both of which were recently discouraged to be used by Wasserstein *et al.* [35]. Instead, we decided to start with a sample size that would be practical to work with under the given time constraints and report the uncertainties from the posteriors directly. In the case of inconclusive results, we could add more projects during the thesis time, if time allows for it, or leave it for future work to use this thesis' results as priors.

### 3.4.3 Building the Dataset

The first step of assembling the dataset was to build a library of possible projects that we could choose projects from until we would reach our final sample size. This library consisted of three sections:

**Prior Work Projects** as shown in Section A.1. We chose those projects, as they come from studies that have investigated Bugspots or algorithms similar to it. This will allow for an easier connection of our results to the existing body of research.

**Random Github Projects** from the top 1000 GitHub projects, ranked by stars. Section A.2 shows the complete drawing process and indicates why some of the projects that were drawn did not qualify for the library. The main reason was that we decided to require at least 3000 commits, to allow for larger pseudo futures.

**Authors Choice Projects** based on experience and knowledge of certain domains. This was primarily done to have a more even distribution of programming languages and domains in our library. This kind of convenience sampling might introduce bias in our results, so we marked projects that were added by the authors for later analysis of the influence.

After building the library, we randomly drew 15 projects from it, which is a nice number above 12 but simple convenience otherwise. Table A.1 shows the list of all potential projects and their index, while Table A.2 shows the drawing process itself. We then added another 6 projects from past studies, as we felt that they might be underrepresented and then another two projects from the random GitHub section of our library as we felt that they were underrepresented as well. The resulting project list is shown in Table 3.1. While there is some randomness to the way we chose the projects, there is convenience sampling involved again, so as with the sources, we mark if a project made it from the library into the sample by random chance or by author choice. An analysis of the influence of the source and decider for each project is presented in Section 5.7.4 The domain column is based on the Global Industry Classification Standard [38].

**Table 3.1:** Project property overview. Src = Source, Ch = Choice, A = Author, P = Past, R = Random

| Project | Src | Ch | fix indicator | commits | language | domain |
|---|---|---|---|---|---|---|
| PrestaShop | A | R | fix \|fixes | 53k | PHP | 25502020 |
| scikit-kearn | A | R | fix \|fixes | 24k | Python | 45102020 |
| BroadleafCommerce | A | R | fix \|fixes \|fixed | 16k | Java | 25502020 |
| ceylon-ide-eclipse | P | R | fix \|fixed | 8k | Java | 45103010 |
| WooCommerce | A | R | fix \|fixes | 31k | PHP | 25502020 |
| ffmpeg | R | R | fix \|fixed | 93k | C | 50202010 |
| Rails | A | R | fix \|fixes | 73k | Ruby | 45102030 |
| Lucene-Solr | P | R | fix | 31k | Java | 45102020 |
| MariaDB Server | A | R | fix \|fixed \|fixing | 185k | C++ | 45102020 |
| MySQL Server | A | R | fix | 148k | C++ | 45102020 |
| mpchc | A | R | fix | 10k | C | 50202010 |
| junit5 | P | R | fixes | 5k | Java | 45103020 |
| Equinox Framework | P | R | fix | 4k | Java | 45103020 |
| bootstrap | A | R | fix | 18k | JS | 45102030 |
| cpython | A | R | fix \|fixes | 100k | Python | 45103020 |
| discourse | R | A | FIX:\|fix | 33k | Ruby | 45103010 |
| mongoose | R | A | fix(\|fix:\|fix | 10k | JS | 45102020 |
| commons-math | P | A | fix\|fixed | 6k | Java | 45103020 |
| closure compiler | P | A | fix \|fixed | 14k | Java | 45103020 |
| jfreechart | P | A | fix \|fixed | 3k | Java | 45103020 |
| coala | P | A | Fixes | 4k | Python | 45103020 |
| evolution | P | A | fix \|fixes | 44k | C | 45103010 |
| httpd | R | A | fix | 31k | C | 45102030 |

## 3.5 Validation Data

To calculate the metrics as described in Section 3.3, we need the output of Bugspots and Linespots and have to compare them with some kind of validation data or reality. One problem with fault prediction is that by its nature, it tries to predict faults that have not led to a failure and thus might not have been detected in the present. One way to build the validation data, as described in our past work [18], is to build a pseudo future. This takes advantage of the capability of Git to essentially time travel. When checking out a commit from the past and then treating it as the present, we have all commits ahead of the commit as a future to test against. Figure 3.3 shows the concept of creating a pseudo future on the timeline of a Git branch.

The creation of a pseudo future allows us to use the same basic line tracking and fix inducing commit identification as Linespots uses to build the validation data. We start with all lines that exist at the pseudo present commit and then walk along the branch towards the future, tracking line changes and marking lines that are removed in a fix inducing commit as faulty. We keep track of lines that were added after the pseudo present, as we do not want to have faults in those lines skew our results. Only lines that exist at the pseudo present can be proposed to be faulty or not and only the ranking of those lines should be validated. The size of this pseudo future will impact the results of an evaluation, as larger pseudo futures will lead to finding faults that are fixed further from the pseudo present. As there currently is no empirical research on optimal pseudo future sizes known to us, we suggest using the largest sized pseudo future possible as it increases the validity of the resulting metrics. The only downside to using larger pseudo futures are higher run times for the evaluation and a barrier for smaller projects that might not have enough commits to allow for the desired size.

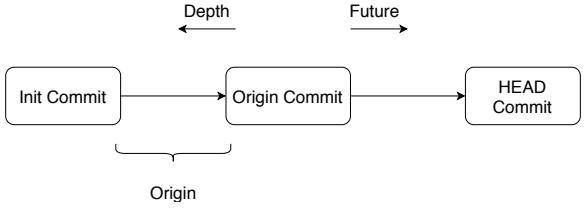



**Figure 3.3:** Origin, depth and future on the commit history

## 3.6 Parameters

There are parameters for both the Bugspots and Linespots algorithms, and the evaluation suite that must be set. In this section, we present and argue for the parameters we used in our evaluation.

### 3.6.1 Fix Indicator

The fix indicator is the string that identifies fix inducing commits. It is individual for each project and often not derived from a strict commit message rule. Table 3.1 shows the fix indicators for each project as proposed by us. As shown in the table, most of the identifiers were derived by us reading through the commit logs of the projects and deriving a search string by feel. This process is prone to error and ideally, clear commit message rules would give unambiguous indicators for fix inducing commits.

### 3.6.2 Weighting Function and Time Version

Both the Linespots and the Bugspots algorithms rely on a weighting function and a way to calculate the relative commit age. In Section 2.3.1 we presented three different weighting functions, and in section [Relative Commit Age] we presented two different ways to calculate the relative commit age, called time version in the algorithm. We will use all presented weighting functions and time versions in the evaluation to answer the respective research questions.

### 3.6.3 Origin

The origin parameter is the pseudo present commit as discussed in Section 3.5. It marks the entry point for the Linespots algorithm and the start of the pseudo future. We randomly chose origins throughout the entire commit history of the projects, with the condition that enough commits are left for the depth and the future as described in sections 3.6.4 and 3.6.5. This meant that each origin would combine a 3000 commit range into one sample. We also decided to not have an overlap in the overall analyzed commits per sample, as it gives us a larger variety of analyzed periods over the project histories.

### 3.6.4 Depth

The depth parameter is used by Bugspots and Linespots and sets the number of commits they analyze and base their predictions on. There has not been any empirical research done on optimal depth sizes that is known to us but the first implementation, written by a Google engineer that was published used 500 commits

as the default value [28], which is also what we used in our past work on Linespots [18]. Preliminary analysis showed that analyzing more commits results in better predictions. This follows our intuition, as more information should lead to better predictions. Based on these preliminary findings, we increase the depth parameter to 2000 for this experiment. While a depth of 500 leads to shorter run time, a depth of 2000 presents more of a best-case scenario for both Bugspots and Linespots but allows us to investigate the potential for both algorithms.

### 3.6.5 Future

The future parameter sets the size of the pseudo future used to build the validation data, as described in section [Validation Data]. For this experiment, we use a future size of 1000 for all samples. Although we have used smaller future sizes of 150 commits in our past work [18], a larger future size is desirable here. So, we chose 1000 commits as it still allowed for a reasonable run time of the evaluation.

## 3.7 Procedure

As we build a fully automated evaluation suite, the entire experimental procedure is handled by it. The evaluation suite uses a JSON configuration file that holds all the necessary information for each project. The linespots.utils.json_builder module contains the code to create the configuration for this experiment. The script for running the evaluation suite with the given configuration file is the full-evaluation.py file.

For each project, the evaluation suite runs each set of (origin, depth, future) multiple times to test the different algorithms, weighting functions and time versions. First, it creates the set of all future fixes for a given origin and future, then for each combination of algorithm, weighting function and time version it runs the algorithm with the given parameters and calculates the metrics for that run. For this experiment, this results in 12 runs for each set of (origin, depth, future).

The results are saved in individual CSV files named after the project and after completing the entire evaluation, they are combined into the full_evaluation.csv file for convenience.

This setup allows for future research to be based on the same evaluation suite, simply using a different configuration file. We plan to develop this approach further and build it into a library for fault prediction benchmarking, as we found a lack of standardized benchmarking in the field of fault prediction.

## 3.8 Analysis

For this thesis, we follow the process of Bayesian data analysis outlined by Schad *et al.* [31]. This chapter outlines the steps we took. It does, however, require the reader to have a basic understanding of Bayesian statistics. We recommend Statistical Rethinking by McElreath [23] as an introduction to the methods. We used stan [41] for statistical modeling through the brms R package [20].

### 3.8.1 Exploration and Simulation

Before we start building the actual models, we explore the data using descriptive statistics, histograms, density- and boxplots. This can serve as a first sanity check as unexpected results could be due to a problem in our process. For example, the first time we ran the exploration, we noticed that the EXAM scores we got for Bugspots were far lower than the results of Zou *et al.* [26]. We investigated and found a fault in the way we calculated the EXAM scores. This finding also led to the realization that there are multiple ways to calculate the AUCEC.

After the exploration, we tested our modelling ideas on simulated data first. This is one of the steps recommended by the stan developers [33] for model validation but we also found it useful as a starting point before starting the modelling process. As we know the exact parameters that went into simulating the data, we could make sure that the models we were planning to use would be able to retrieve the parameters we were looking for. This step can also help when interpreting the real models later on as the simulated models can work as a guide with known effects.

## 3.9 Projection Predictive Feature Selection

As the first step in our analysis, we identify the variables that are best suited to predict the outcomes that we want to analyze. This allows us to focus on building and comparing models with those predictors and ignore predictors that seem to have low explanatory power. Without this kind of feature selection, we would ideally try every possible combination of predictors in our models to find the simplest model with the best explanatory power. The process we use for this is projection predictive feature selection proposed by Piironen *et al.* [25]. Figure 3.4 shows the plotted result of the feature selection for a model predicting the EXAM for Linespots. It shows the performance of models after step-wise including the next best predictor. For this model, the proposed model size is 3 and the figure shows that adding more predictors does not increase the explanatory power of the model. The plots for the other outcome variables are in appendix Chapter B.

It is important to note that the current implementation of projpred does not support beta distribution likelihoods. Instead, we used a gaussian likelihood, which may not

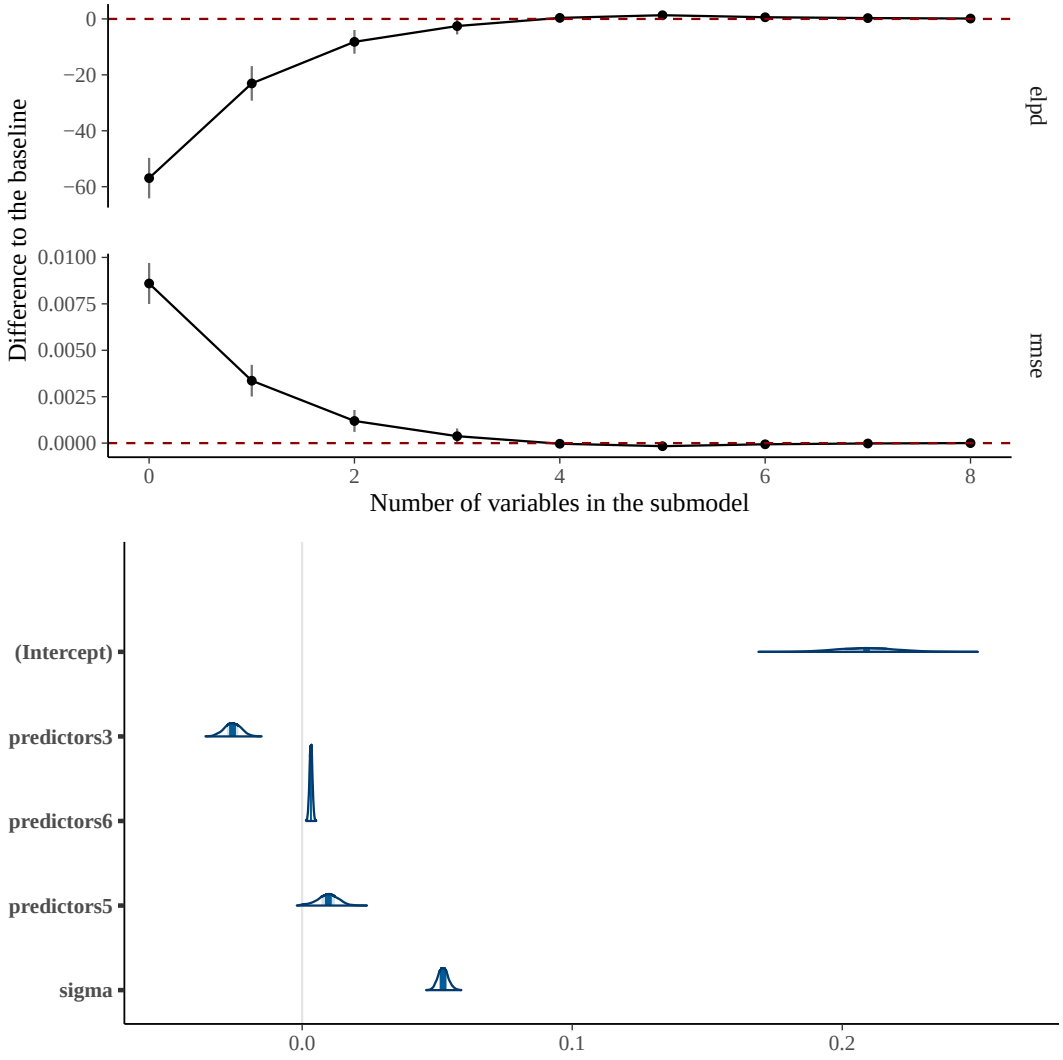



**Figure 3.4:** varsel_plot and mcmc_areas plot for EXAM prediction in RQ1 and RQ2

be ideal given the nature of our data. However, we still used the results as a starting point for our model building phase to reduce the time needed.

### 3.9.1 Prior Sensitivity Analysis

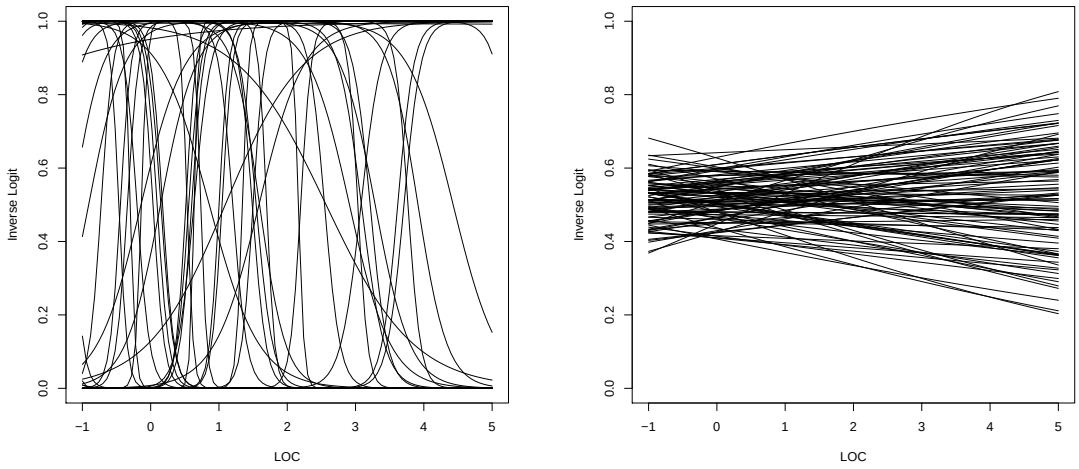
This is sometimes also called parameter sensitivity analysis.

The most common prior in Bayesian inference is the Gaussian. It is easy to work with and if all we know about a distribution are the mean and variance, it is the maximum entropy distribution. Maximum entropy means that it offers the highest number of possibilities for the data to follow the distribution. As we use priors as the starting point for our models, we don't wont to push the models in unlikely directions without good arguments to prevent unnecessary bias. In the case of both the AUCEC and EXAM scores however, we can say more about their distributions as we know their process of calculation. Both are limited between 0 and 1. We could either add this prior knowledge by limiting a normal prior to between 0 and 1, or we could instead use a beta distribution. The beta distribution is limited between 0 and 1 and often used to describe probabilities of probabilities or events which have limited values. While both the AUCEC and EXAM score are not strictly speaking probabilities, they both are connected to the idea of how likely it is to find faults when following a ranked list and intuitively can be understood as the probability that the highest-ranked lines contain all faults. For these reasons, we chose the beta distribution as the likelihood for both the AUCEC and EXAM scores.

As for the different predictors, we will go with Gaussian priors as the maximum entropy distributions for continuous variables. While the even wider student's t distribution has become a more popular default, primarily to allow for more extreme cases in financial modelling, we argue that this is not necessary for the domain of fault prediction at this point.

We then use prior sensitivity analysis to choose proper priors for our models. Figure 3.5a shows the result of very wide priors for our model. The steep curves are a sign that our model would put a lot of weight on large effects, as a small change in LOC would lead to a large difference in the EXAM score. Using this technique, we can iteratively narrow the priors until their behaviour seems to match our prior expectation. Figure 3.5b shows the impact of too narrow priors. As most curves are almost horizontal over the parameter space, the estimated prior effect sizes are extremely small.

Finally, Figure 3.6 shows how a good prior could look like. It allows for small and large effects equally and does not put too much weight on either of both. While there is a visible emphasis on the limits to the right side of the plot that is to be expected for such large LOC values and will be hard to model better with a Gaussian prior. As the idea of a prior is to include prior knowledge about the nature of the data, without looking at the data itself, this process is not entirely systematic and will depend on one's experience with the domain.




**(a)** Curves for Normal(0, 10) priors **(b)** Curves for Normal(0, 0.1) priors

**Figure 3.5:** Curves for a wide and a narrow prior

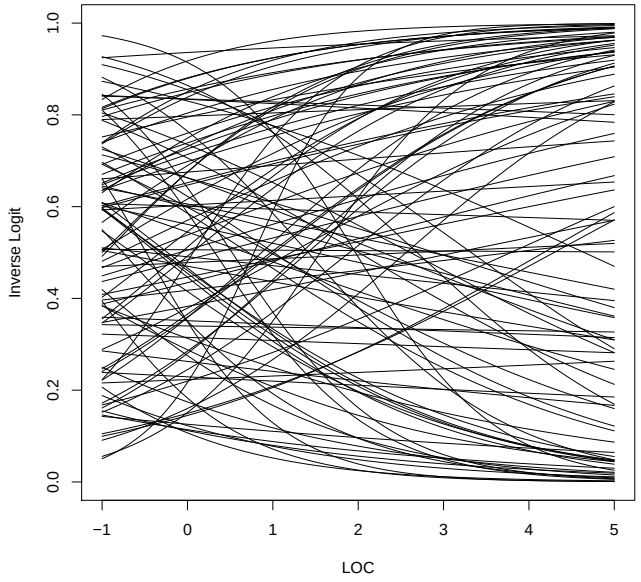



**Figure 3.6:** Curves for Normal(0, 0.5) priors

## 3.10 Model Design

Based on the proposed predictors we build several models of increased complexity and compared their performance using state of the research Pareto smoothed importance sampling leave one out cross-validation [34]. These ranged from models that simply included one predictor up to the models that include all the predictors that the feature selection proposed. We always start with the variable for which we want to measure the effect, regardless of the proposed features. We then used prior sensitivity analysis to evaluate the impact of different priors. These steps were iterated back and forth multiple times until we chose the best performing models for a final comparison and analysis.

### 3.10.1 Model Diagnostics

Before interpreting a model, one has to check if the Markov Chain Monte Carlo sampling did work properly. The three most important diagnostics according to the stan developers [33] are the number of effective samples (n_eff), the consistency of multiple Markov chains quantified as R-hat (Rhat) and not having divergent transitions. The Rhat should approach 1 from above and be no more than 1.01. The threshold for n_eff is more dependent on what the purpose of the model is. If one is only interested in mean values, very little samples are needed. McElreath [23] proposes around 200 samples in this case. If one, however, is interested in the behaviour of the posterior in the extremes, then many more samples are necessary for this. For an approximately Gaussian posterior, McElreath [23] proposes 2000 samples. In the past, 0.1 of post warmup samples was used as a threshold for n_eff. We will use this as it is more conservative in our case. Regardless of one's goal, a very low n_eff can be a sign for sampling problems and should be investigated. McElreath [23] also proposes the visual inspection of the trace plots of the chains to ensure they mix well and are neither stuck in one point nor swing too far out. The desired shape should look like a "Hairy Caterpillar".

### 3.10.2 Model Comparison

For our model comparison, we use the loo package (cite) and the loo() and loo_compare functions. Table 3.2 shows the comparison of the EXAM models for research question 1:

The three columns are the model name, models become more complex with a higher index, the elpd difference to the best performing model and the standard error of the difference. For a significant difference between two models, the difference should be at least twice, but preferably four times the standard error. In this case, this means that models 3 to 7 are almost indistinguishable in performance. Model 4 is right on the edge so we might rule it out to reduce the number of models. We decided to proceed with models 3, 5 and 7 as model 3 is the simplest model, model 7 is the

**Table 3.2:** loo_compare result for research question 1 EXAM models

| | elpd_diff | se_diff |
|---|---|---|
| m1.5 | 0.0 | 0.0 |
| m1.7 | -0.2 | 0.2 |
| m1.6 | -0.2 | 0.4 |
| m1.3 | -0.9 | 1.7 |
| m1.4 | -1.0 | 0.5 |
| m1.2 | -241.3 | 16.0 |
| m1.1 | -278.8 | 17.1 |

most complex one and model 5 is the highest-ranked by the loo. While one would usually go with the simplest model to prevent overfitting and sampling issues, we want to compare the three models again after a final more thorough sampling.

### 3.10.3 Model Interpretation

Model interpretation is complex and there is no single correct way of doing it. A lot of it depends on what one is interested in and wants to show. In our case, we are interested in the differences in outcomes between different algorithms, time versions or weighting functions. The brms package offers us to get posterior predictions from a model, which we can use to get a distribution of differences between the variables we want to analyze. To get the differences, we calculate the posterior predictions based on the subsamples of our data that were created with the desired variable, for example, all samples drawn from with the Linespots algorithm, using the google weighting function. We can then subtract the posterior predictions from each other to receive the differences. The result of the subtraction is an empirical distribution of the absolute effect of moving from one weighting function to another. We can then present the distribution itself together with descriptive statistics such as the mean, median and confidence intervals.

# 4

# Results

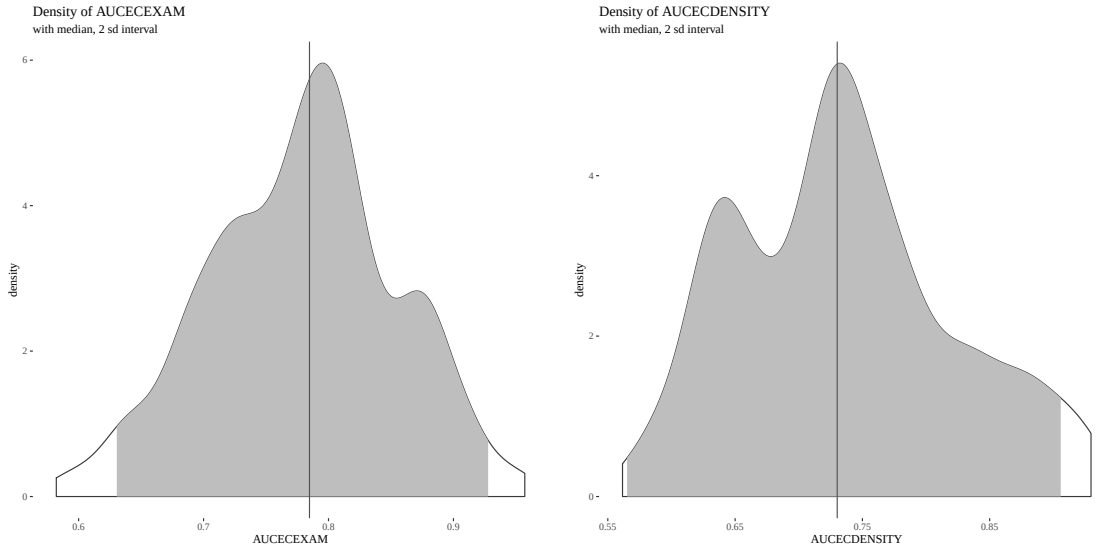
This chapter presents and describes the results of our experiments and analysis. For an interpretation and discussion, see chapter Discussion. The sections are split into the different Metrics we investigated for easier access. All model diagnostics are shown in appendix D.

## 4.1 Exploration

The first thing we wanted to investigate is how similar the AUCECEXAM and AUCECDENSITY results were. As discussed in Section 3.3.9 we noticed that there are different ways to calculate the AUCEC metric which might differ. Figure 4.1 shows both versions side by side and it is obvious that they are not the same and AUCECEXAM seems to be higher on average. This is also supported by higher mean and median values for AUCECEXAM compared to AUCECDENSITY as shown in Table 4.1. The distribution of AUCECDENSITY also seems wider and with a stronger bimodal shape compared to AUCECEXAM that is closer to a Gaussian shape. This matches the higher standard deviation of AUCECDENSITY displayed in Table 4.1.

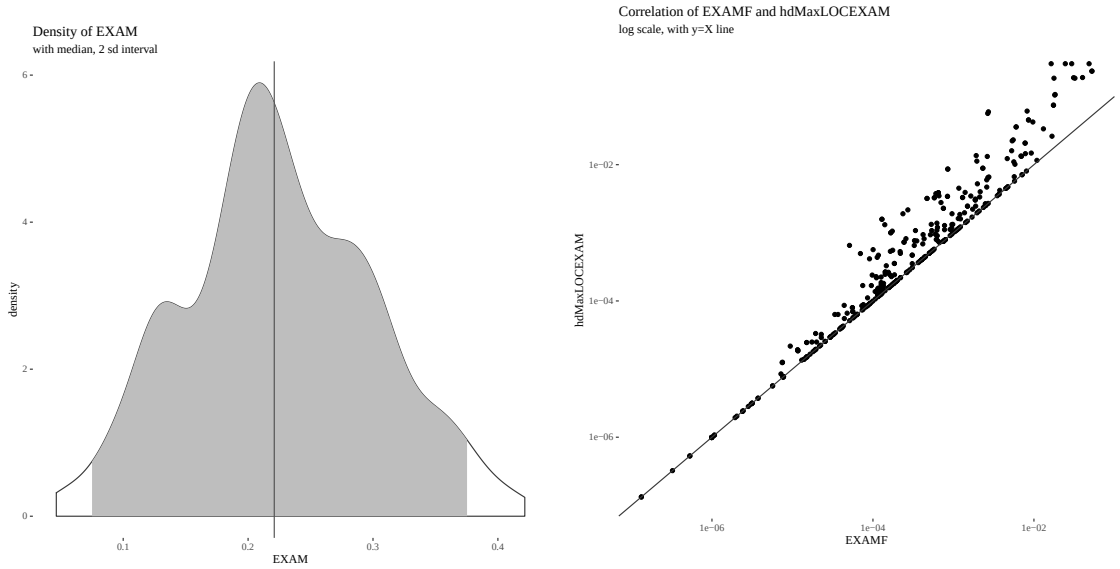
When comparing the AUCECEXAM and EXAM plots, they seem to be almost perfectly mirrored at the y-axis. Both density plots have the same shape and the with of the 95% intervals is roughly 0.3 for both. Another similarity is shown in Figure 4.2b that shows a plot of EXAMF against hdMaxLOCEXAM which shows a strong correlation between the two. This is expected, as the hit density will be at its highest at the first predicted fault. EXAMF also is the lower limit for hdMaxLOCEXAM, as hit density is 0 until at least the first fault is reached.

Finally, we compared results for Linespots between the different programming languages and domains in Figure 4.3. Starting with the languages, it seems that JavaScript and C++ behave different from the rest of the languages but similar to each other. The very small standard deviation of JavaScript is also noteworthy. The results for the domains are more similar to each other with only domain 45102020 (Data Processing & Outsourced Services) behaving notably better than the rest.




**(a)** AUCECEXAM **(b)** AUCECDENSITY

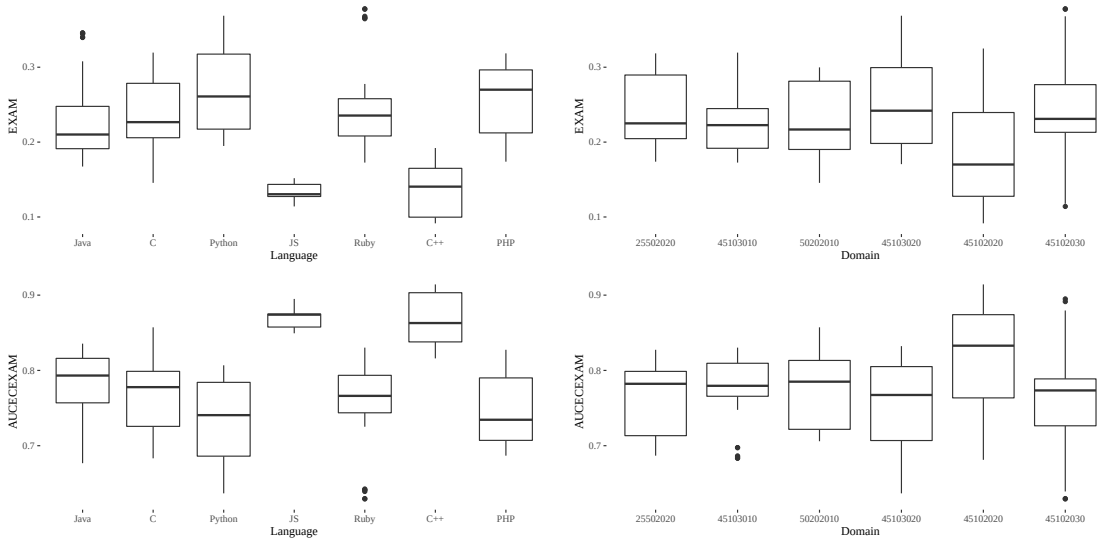
**Figure 4.1:** Densities of AUCECEXAM and AUCECDENSITY with 2 SD intervals




**(a)** EXAM with 2 SD intervals

**(b)** EXAMF and hdMaxLOCEXAM with diagonal on log scale

**Figure 4.2:** Density of EXAM and correlation of EXAMF and hdMaxLOCEXAM




**(a)** Language **(b)** Domain

**Figure 4.3:** EXAM and AUCECEXAM results for Linespots for different Languages and Domains

| | -4 SD | -2 SD | Mean | Median | +2 SD | +4 SD |
|---|---|---|---|---|---|---|
| EXAM | 0 | 0.104 | 0.226 | 0.221 | 0.348 | 0.471 |
| AUCECEXAM | 0.481 | 0.623 | 0.779 | 0.785 | 0.928 | 1.0774 |
| AUCECDENSITY | 0.394 | 0.565 | 0.735 | 0.73 | 0.906 | 1.077 |
| EXAM25 | 0 | 0 | 0.0278 | 0.0164 | 0.0891 | 0.15 |
| EInspect25EXAM | 0.394 | 0.565 | 0.735 | 0 | 0.906 | 1.077 |
| EXAMF | 0 | 0 | 0.00363 | 0.000472 | 0.0241 | 0.0445 |
| hdMaxLOCEXAM | 0 | 0 | 0.0154 | 0.000717 | 0.113 | 0.211 |

**Table 4.1:** Summary of exploration, rounded to 3 decimals or integer, 0 used as lower limit

## 4.2 Research Question 1

The first research question was: What kind of weighting function produces the best results for Linespots?

### 4.2.1 EXAM Results

After the model building phase, we kept two models. The simpler one, Model C.1, did not sample quite good enough, as shown in Figure D.1b. For this reason, we created a more complex model that uses an additional varying intercept with Model 4.1. While usually more complex models sample worse, the additional varying intercept probably breaks up the parameter space which helps with sampling.

$$
\begin{aligned}
\text{EXAM}_i &\sim \text{Beta}(\mu_i, \phi) \\
\text{logit}(\mu_i) &= \alpha + \beta_w W_i + \beta_l L_i + \gamma_{Project[i]} + \delta_{Language[i]} \\
\alpha &\sim \text{Normal}(0, 0.5) \\
\beta_w, \beta_l &\sim \text{Normal}(0, 0.5) \\
\gamma_j &\sim \text{Normal}(\bar{\gamma}, \sigma) \\
\delta_j &\sim \text{Normal}(\bar{\delta}, \epsilon) \\
\bar{\gamma}, \sigma, \bar{\delta}, \epsilon &\sim \text{HalfCauchy}(0, 0.1) \\
\phi &\sim \text{Gamma}(0.1, 0.1)
\end{aligned}
$$

**Model 4.1:** Final model for EXAM using Weighting

Both models have similar estimated values for the weighting functions as shown in Table 4.2 but they differ in their estimations for the intercept. Here the more complex model has more uncertainty. Table 4.3 shows the loo comparison between the two models and they seem equivalent in that regard. As the effect of the weighting functions depends partially on the intercept, due to the non-linear nature of the inverse logit transformation, we use the more complex model for our analysis to incorporate the higher uncertainty it has based on better sampling.

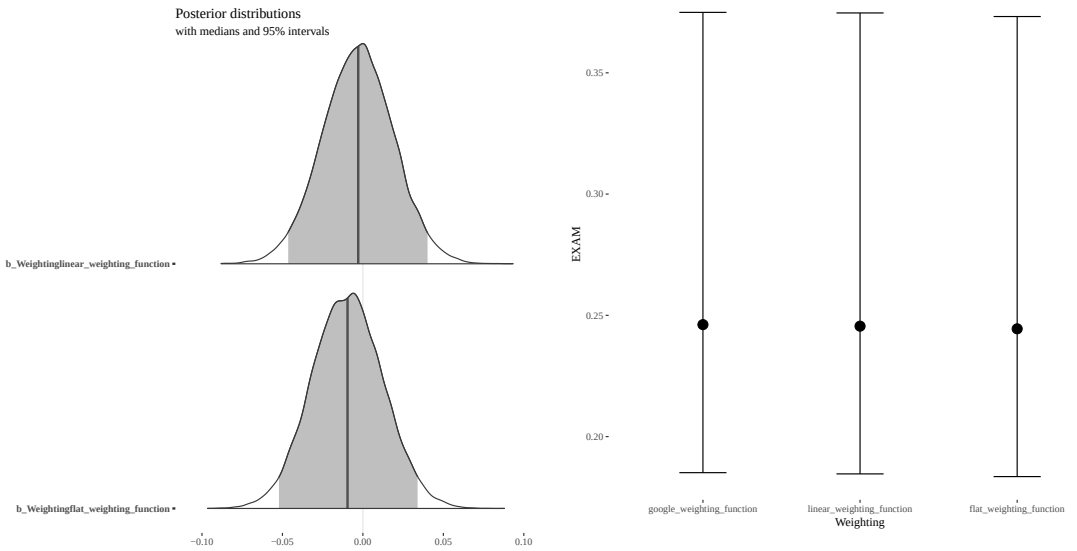
Moving to the results of Model 4.1, we start with the posterior distribution of the weighting functions coefficients in Figure 4.4a. Both the linear and the flat weighting functions have negative means but a high overlap with 0 in their posterior distributions. As the posterior distributions are on the logit scale, it is hard to interpret the effects on the outcome scale from the distributions alone. One way to approach this is to look at the marginal effects shown in Figure 4.4b. It shows the mean and 95% intervals for the EXAM scores predicted by Model 4.1 and split by weighting-function. However, there are no clear differences between the marginal effects of the weighting-functions.

| Model C.1 | Estimate | Est.Error | Q2.5 | Q97.5 |
|---|---|---|---|---|
| Intercept | -1.16 | 0.109 | -1.36 | -0.931 |
| linear_weighting | -0.00318 | 0.0221 | -0.0466 | 0.0396 |
| flat_weighting | -0.00925 | 0.0221 | -0.053 | 0.0341 |
| LOC | 0.218 | 0.0299 | 0.159 | 0.276 |
| Model 4.1 | Estimate | Est.Error | Q2.5 | Q97.5 |
| Intercept | -1.09 | 0.242 | -1.48 | -0.511 |
| linear_weighting | -0.00307 | 0.022 | -0.0464 | 0.0402 |
| flat_weighting | -0.0095 | 0.022 | -0.0522 | 0.034 |
| LOC | 0.227 | 0.0304 | 0.168 | 0.287 |

**Table 4.2:** Fixed effects of models C.1 and 4.1 rounded to 3 significant digits

| | elpd_diff | se_diff |
|---|---|---|
| Model 4.1 | 0.0 | 0.0 |
| Model C.1 | -0.5 | 0.3 |

**Table 4.3:** loo_compare output for models C.1 and 4.1

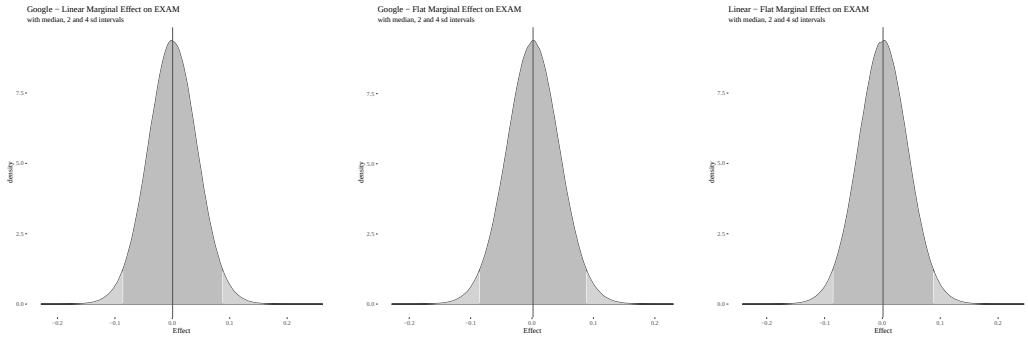



**(a)** Posteriors of the coefficients

**(b)** Marginal effects of the weighting functions with 95% intervals

**Figure 4.4:** Posteriors and marginal effects for the weighting functions in Model 4.1

Finally, we can take a look at the differences between predictions on the outcome scale as shown in Figure 4.5 and summarized in Table 4.4. These show the pairwise contrasts between all three weighting functions and their distributions. All three contrast distributions have medians close to 0 and the 95% intervals overlap 0 evenly.




**(a)** Google - Linear **(b)** Google - Flat **(c)** Linear - Flat

**Figure 4.5:** Contrasts between the different weighting functions based on posterior predictions in Model 4.1

| | -4 SD | -2 SD | Median | +2 SD | +4 SD |
|---|---|---|---|---|---|
| Google - Linear | -0.173 | -0.0863 | 0.000543 | 0.0874 | 0.174 |
| Google - Flat | -0.172 | -0.0851 | 0.00163 | 0.0884 | 0.175 |
| Linear - Flat | -0.172 | -0.0856 | 0.00106 | 0.0878 | 0.174 |

**Table 4.4:** Contrast between weighting functions in Model 4.1 rounded to 3 significant digits

### 4.2.2 AUCECEXAM Results

As indicated in Section 4.1 the results for AUCECEXAM are very similar to the ones using EXAM. For this reason, we only show the plots and tables here and refer to the explanations in Section 4.2.1.

| Model C.2 | Estimate | Est.Error | Q2.5 | Q97.5 |
|---|---|---|---|---|
| Intercept | 1.186 | 0.11 | 0.957 | 1.397 |
| linear_weighting | 0.00302 | 0.0223 | -0.0406 | 0.0469 |
| flat_weighting | 0.00974 | 0.0225 | -0.0344 | 0.0535 |
| LOC | -0.218 | 0.0304 | -0.278 | -0.159 |
| Model 4.2 | Estimate | Est.Error | Q2.5 | Q97.5 |
| Intercept | 1.097 | 0.252 | 0.489 | 1.5 |
| linear_weighting | 0.00283 | 0.0223 | -0.0404 | 0.0465 |
| flat_weighting | 0.00962 | 0.0223 | -0.0337 | 0.0535 |
| LOC | -0.228 | 0.0308 | -0.288 | -0.168 |

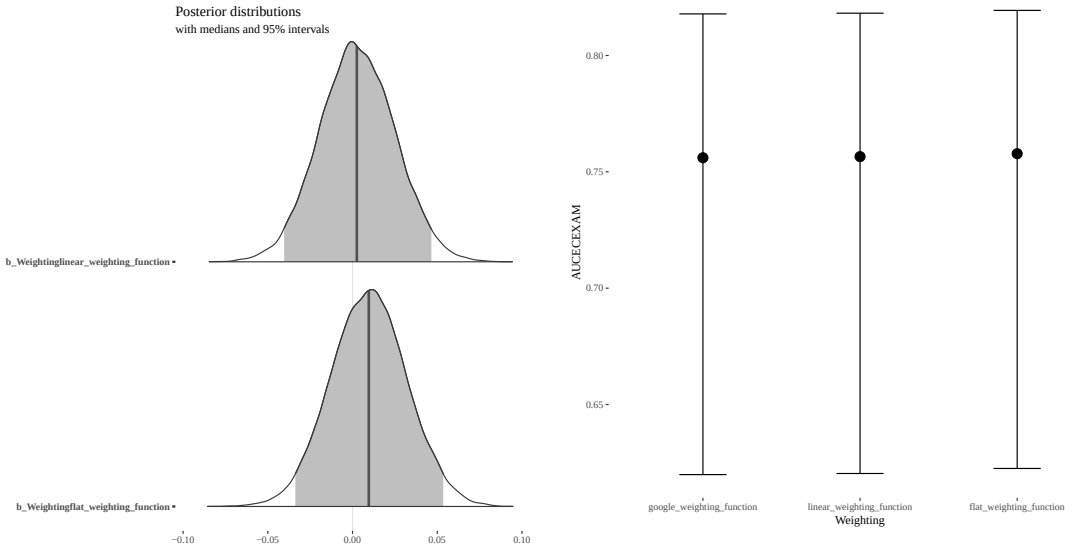
**Table 4.5:** Fixed effects of models C.2 and 4.2 rounded to 3 significant digits

$$
\begin{aligned}
\text{AUCECEXAM}_i &\sim \text{Beta}(\mu_i, \phi) \\
\text{logit}(\mu_i) &= \alpha + \beta_w W_i + \beta_l L_i + \gamma_{Project[i]} + \delta_{Language[i]} \\
\alpha &\sim \text{Normal}(0, 0.5) \\
\beta_w, \beta_l &\sim \text{Normal}(0, 0.5) \\
\gamma_j &\sim \text{Normal}(\bar{\gamma}, \sigma) \\
\delta_j &\sim \text{Normal}(\bar{\delta}, \epsilon) \\
\bar{\gamma}, \sigma, \bar{\delta}, \epsilon &\sim \text{HalfCauchy}(0, 0.1) \\
\phi &\sim \text{Gamma}(0.1, 0.1)
\end{aligned}
$$

**Model 4.2:** Final model for AUCECEXAM using Weighting

| | elpd_diff | se_diff |
|---|---|---|
| Model 4.2 | 0.0 | 0.0 |
| Model C.2 | -0.3 | 0.4 |

**Table 4.6:** loo_compare output for models C.2 and 4.2



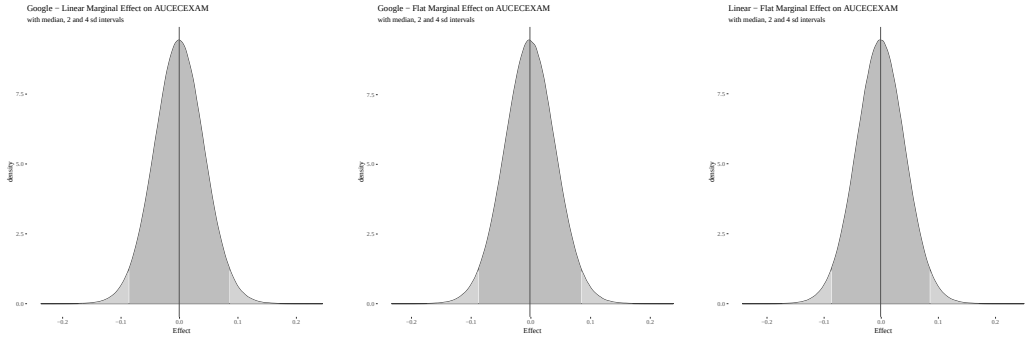

**(a)** Posteriors of the coefficients

**(b)** Marginal effects of the weighting functions

**Figure 4.6:** Posteriors and marginal effects for the weighting functions in Model 4.2

| | -4 SD | -2 SD | Median | +2 SD | +4 SD |
|---|---|---|---|---|---|
| Google - Linear | -0.173 | -0.0867 | -0.000512 | 0.0857 | 0.172 |
| Google - Flat | -0.174 | -0.0878 | -0.00164 | 0.0846 | 0.171 |
| Linear - Flat | -0.173 | -0.0873 | -0.00113 | 0.0849 | 0.171 |

**Table 4.7:** Contrast between weighting functions in Model 4.2 rounded to 3 significant digits




**(a)** Google - Linear **(b)** Google - Flat **(c)** Linear - Flat

**Figure 4.7:** Contrasts between the different weighting functions based on posterior predictions in Model 4.2

## 4.3 Research Question 2

The second research question was: How does index-based age calculation influence the predictive performance of Linespots compared to time stamp based age calculation?

### 4.3.1 EXAM Results

After the model building phase, we kept two models. The simpler one, Model C.3, did not sample quite good enough, as shown in Figure D.17b. For this reason, we created a more complex model that uses an additional varying intercept with Model 4.3. While usually more complex models sample worse, the additional varying intercept probably breaks up the parameter space which helps with sampling.

$$
\begin{aligned}
\text{EXAM}_i &\sim \text{Beta}(\mu_i, \phi) \\
\text{logit}(\mu_i) &= \alpha + \beta_t T_i + \beta_l L_i + \gamma_{Project[i]} + \delta_{Language[i]} \\
\alpha &\sim \text{Normal}(0, 0.5) \\
\beta_t, \beta_l &\sim \text{Normal}(0, 0.5) \\
\gamma_j &\sim \text{Normal}(\bar{\gamma}, \sigma) \\
\delta_j &\sim \text{Normal}(\bar{\delta}, \epsilon) \\
\bar{\gamma}, \sigma, \bar{\delta}, \epsilon &\sim \text{HalfCauchy}(0, 0.1) \\
\phi &\sim \text{Gamma}(0.1, 0.1)
\end{aligned}
$$

**Model 4.3:** More Complex model for EXAM using Time

Both models have similar estimated values for the time-versions as shown in Table 4.8 but they differ in their estimations for the intercept. Here the more complex model has more uncertainty. Table 4.9 shows the loo comparison between the two

models and they seem equivalent in that regard. As the effect of the time-versions depends partially on the intercept, due to the non-linear nature of the inverse logit transformation, we use the more complex model for our analysis to incorporate the higher uncertainty it has based on better sampling.

| Model C.3 | Estimate | Est.Error | Q2.5 | Q97.5 |
|---|---|---|---|---|
| Intercept | -1.2 | 0.11 | -1.42 | -0.983 |
| Timecommit | -0.000485 | 0.0182 | -0.0366 | 0.035 |
| LOC | 0.219 | 0.0304 | 0.159 | 0.278 |
| Model 4.3 | Estimate | Est.Error | Q2.5 | Q97.5 |
| Intercept | -1.09 | 0.243 | -1.48 | -0.502 |
| Timecommit | -0.000284 | 0.0181 | -0.0355 | 0.0354 |
| LOC | 0.227 | 0.0304 | 0.168 | 0.287 |

**Table 4.8:** Fixed effects of models C.3 and 4.3 rounded to 3 significant digits

| | elpd_diff | se_diff |
|---|---|---|
| Model 4.3 | 0.0 | 0.0 |
| Model C.3 | -0.2 | 0.3 |

**Table 4.9:** loo_compare output for models C.3 and 4.3

Moving to the results of Model 4.3, we start with the posterior distribution of the commit time-version coefficient in Figure 4.8. The distribution has a negative mean but almost indistinguishable from 0. The 95% intervals also evenly overlap with 0. To get a better picture for any effects on the outcome scale, we again use the marginal effects shown in Figure 4.9a. It shows the mean and 95% intervals for the EXAM scores predicted by Model 4.3 and split by weighting-function. Similar to the weighting-functions, there is no clear difference between the marginal effects of the time-versions.

Finally, we can take a look at the difference between predictions on the outcome scale as shown in Figure 4.9b and summarized in Table 4.10. Similar to the marginal effects, the contrast between both time-versions has a median close to 0 and an even overlap with 0.

| | -4 SD | -2 SD | Median | +2 SD | +4 SD |
|---|---|---|---|---|---|
| Time - Commit | -0.173 | -0.0864 | 0.0000736 | 0.0866 | 0.173 |

**Table 4.10:** Contrast between time versions in Model 4.3 rounded to 3 significant digits

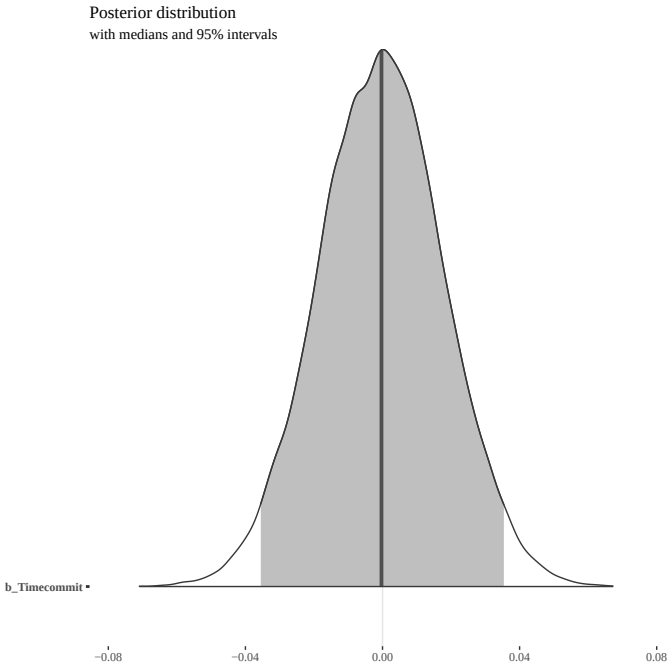
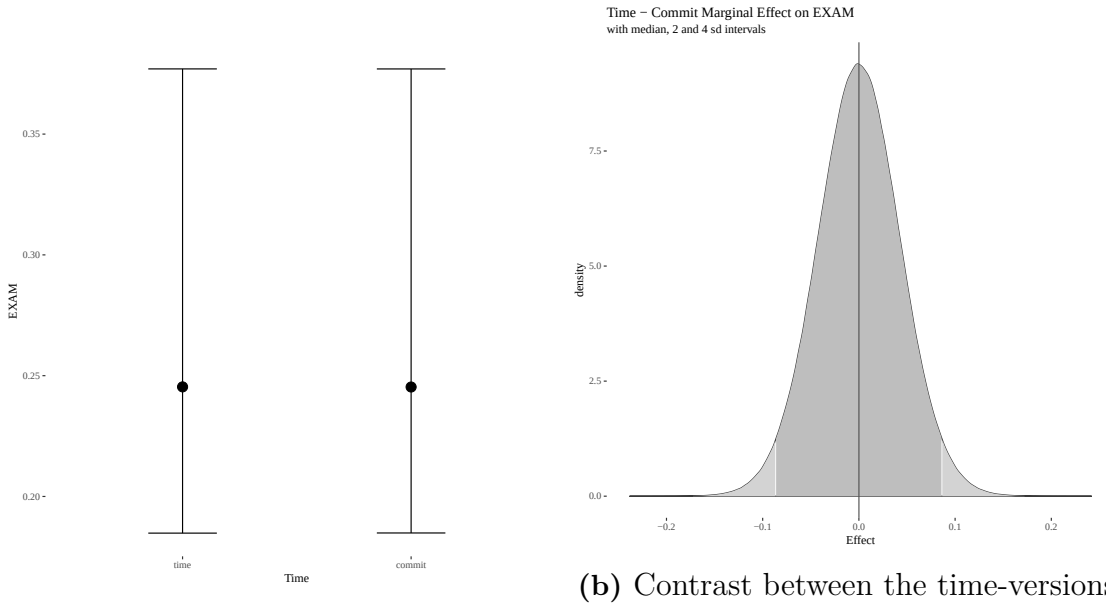



**Figure 4.8:** Posteriors of the coefficients




**(a)** Marginal effects of the time versions

**(b)** Contrast between the time-versions based on posterior predictions in Model 4.3

**Figure 4.9:** Posteriors and contrast between the different time versions based on posterior predictions in Model 4.3

### 4.3.2 AUCECEXAM Results

As indicated in Section 4.1 the results for AUCECEXAM are very similar to the ones using EXAM. For this reason, we only show the plots and tables here and refer to the explanations in Section 4.3.1.

$$
\begin{aligned}
\text{AUCECEXAM}_i &\sim \text{Beta}(\mu_i, \phi) \\
\text{logit}(\mu_i) &= \alpha + \beta_t T_i + \beta_l L_i + \gamma_{Project[i]} + \delta_{Language[i]} \\
\alpha &\sim \text{Normal}(0, 0.5) \\
\beta_t, \beta_l &\sim \text{Normal}(0, 0.5) \\
\gamma_j &\sim \text{Normal}(\bar{\gamma}, \sigma) \\
\delta_j &\sim \text{Normal}(\bar{\delta}, \epsilon) \\
\bar{\gamma}, \sigma, \bar{\delta}, \epsilon &\sim \text{HalfCauchy}(0, 0.1) \\
\phi &\sim \text{Gamma}(0.1, 0.1)
\end{aligned}
$$

**Model 4.4:** More Complex model for AUCECEXAM using Time

| Model C.4 | Estimate | Est.Error | Q2.5 | Q97.5 |
|---|---|---|---|---|
| Intercept | 1.19 | 0.11 | 0.971 | 1.4 |
| Timecommit | 0.000302 | 0.0182 | -0.0355 | 0.0363 |
| LOC | -0.218 | 0.0306 | -0.278 | -0.159 |
| Model 4.4 | Estimate | Est.Error | Q2.5 | Q97.5 |
| Intercept | 1.11 | 0.249 | 0.513 | 1.506 |
| Timecommit | 0.000309 | 0.0181 | -0.0354 | 0.0358 |
| LOC | -0.228 | 0.0307 | -0.288 | -0.168 |

**Table 4.11:** Fixed effects of models Model C.4 and 4.4 rounded to 3 significant digits

| | elpd_diff | se_diff |
|---|---|---|
| Model 4.4 | 0.0 | 0.0 |
| Model C.4 | -0.4 | 0.4 |

**Table 4.12:** loo_compare output for models C.4 and 4.4

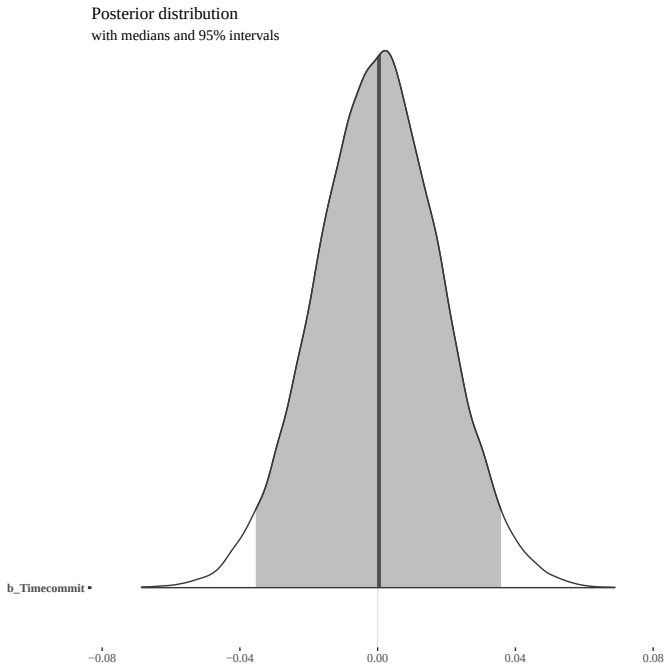
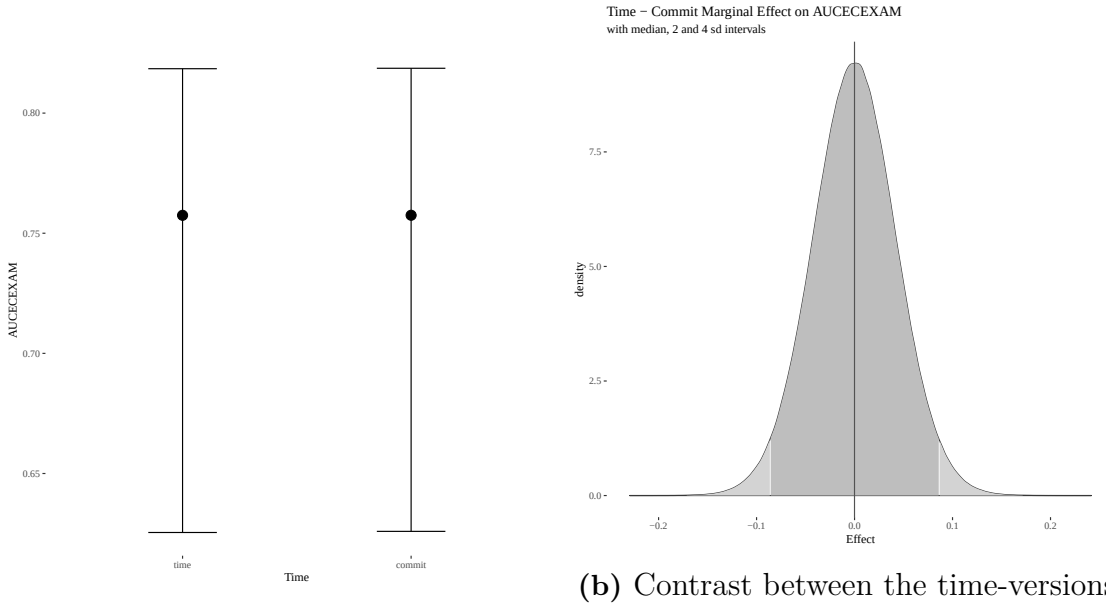



**Figure 4.10:** Posteriors of the coefficients




**(a)** Marginal effects of the time versions

**(b)** Contrast between the time-versions based on posterior predictions in Model 4.4

**Figure 4.11:** Posteriors and contrast between the different time versions based on posterior predictions in Model 4.4

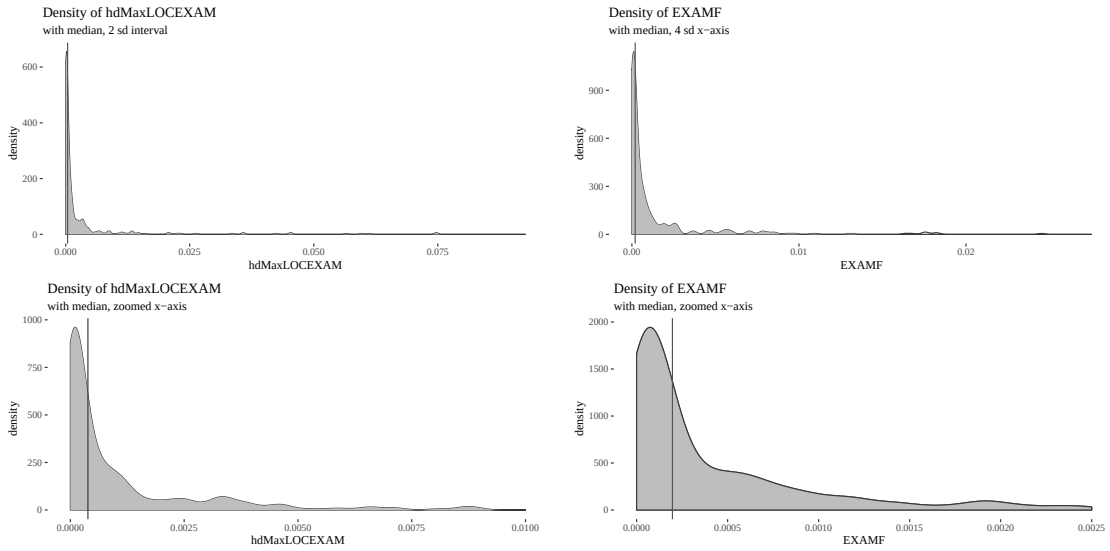
| | -4 SD | -2 SD | Median | +2 SD | +4 SD |
|---|---|---|---|---|---|
| Time - Commit | -0.172 | -0.086 | -0.0000546 | 0.0859 | 0.172 |

**Table 4.13:** Contrast between time versions in Model 4.4 rounded to 3 significant digits

## 4.4 Research Question 3

The third research question was: What is a good cut-off-point to turn Linespots into a classifier? For this question, we explore the behaviour of Linespots at different LOC percentages.

Starting with hdMaxLOCEXAM and EXAMF in Figure 4.12, both have a similar shape. As discussed in Section 4.1 they are strongly correlated and EXAMF represents a lower limit to hdMaxLOCEXAM, which is supported by their summary in Table 4.14.



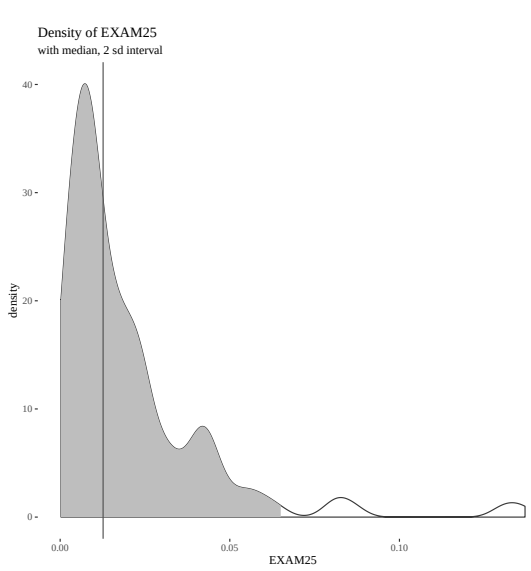

**(a)** hdMaxLOCEXAM **(b)** EXAMF

**Figure 4.12:** Densities of hdMaxLOCEXAM and EXAMF for Linespots samples

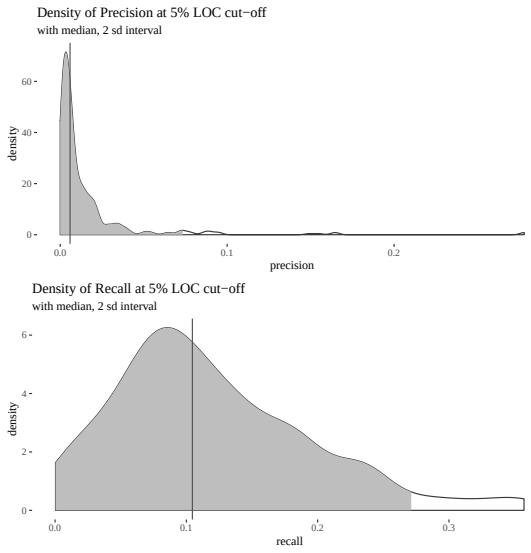
In addition to those lower limit measures, we also measured what proportion of lines had to be inspected to find the first 25% of faults with EXAM25 as shown in Figure 4.13a and the precision and recall when using a 5% LOC cut-off point as shown in Figure 4.13b.

| | -2 SD | Mean | Median | +2 SD | +4 SD |
|---|---|---|---|---|---|
| EXAMF | 0 | 0.00212 | 0.000197 | 0.0148 | 0.0275 |
| hdMaxLOCEXAM | 0 | 0.0101 | 0.000385 | 0.0926 | 0.175 |
| EXAM25 | 0 | 0.0202 | 0.0127 | 0.065 | 0.11 |
| Precision@5%LOC | 0 | 0.0149 | 0.00591 | 0.0739 | 0.133 |
| Recall@5%LOC | 0 | 0.12 | 0.105 | 0.271 | 0.423 |

**Table 4.14:** Mean, Median, 2 and 4 standard deviation intervals for the Linespots sample




**(a)** EXAM25




**(b)** Precision and Recall for 5% LOC cut-off

**Figure 4.13:** EXAM25, precision and recall densities for Linespots samples

## 4.5 Research Question 4

The fourth research question was: What is the prediction performance of Linespots compared to Bugspots?

### 4.5.1 EXAM Results

After the model building phase, we kept two models. The simpler one, Model C.5, did not sample quite good enough, as shown in Figure D.33b. For this reason, we created a more complex model that uses an additional varying intercept with Model 4.5. While usually more complex models sample worse, the additional varying intercept probably breaks up the parameter space which helps with sampling.

Both models have similar estimated values for Bugspots as shown in Table 4.15 but they differ in their estimations for the intercept. Here the more complex model has more uncertainty. Table 4.16 shows the loo comparison between the two models and they seem equivalent in that regard. As the effect of Bugspots depends partially on the intercept, due to the non-linear nature of the inverse logit transformation, we use the more complex model for our analysis to incorporate the higher uncertainty it has based on better sampling.

Moving to the results of Model 4.5, we start with the posterior distribution of the Bugspots coefficient in Figure 4.14. The distribution has a clear negative mean but the 95% intervals still overlap with 0. This effect is again hard to interpret as it is on the logit scale.

$$
\begin{aligned}
\text{EXAM}_i &\sim \text{Beta}(\mu_i, \phi) \\
\text{logit}(\mu_i) &= \alpha + \beta_a A_i + \beta_l L_i + \gamma_{Project[i]} + \delta_{Language[i]} \\
\alpha &\sim \text{Normal}(0, 0.5) \\
\beta_a, \beta_l &\sim \text{Normal}(0, 0.5) \\
\gamma_j &\sim \text{Normal}(\bar{\gamma}, \sigma) \\
\delta_j &\sim \text{Normal}(\bar{\delta}, \epsilon) \\
\bar{\gamma}, \sigma, \bar{\delta}, \epsilon &\sim \text{HalfCauchy}(0, 0.1) \\
\phi &\sim \text{Gamma}(0.1, 0.1)
\end{aligned}
$$

**Model 4.5:** More Complex model for EXAM using Algorithm

| Model C.5 | Estimate | Est.Error | Q2.5 | Q97.5 |
|---|---|---|---|---|
| Intercept | -1.16 | 0.0951 | -1.35 | -0.969 |
| Bugspots | -0.0214 | 0.0199 | -0.0607 | 0.0175 |
| LOC | 0.043 | 0.0322 | -0.0203 | 0.106 |
| 4.5 | Estimate | Est.Error | Q2.5 | Q97.5 |
| Intercept | -1.16 | 0.157 | -1.44 | -0.809 |
| Bugspots | -0.0217 | 0.0198 | -0.0604 | 0.0173 |
| LOC | 0.0481 | 0.0325 | -0.0156 | 0.112 |

**Table 4.15:** Fixed effects of models C.5 and 4.5 rounded to 3 significant digits

| | elpd_diff | se_diff |
|---|---|---|
| Model 4.5 | 0.0 | 0.0 |
| Model C.5 | -0.2 | 0.3 |

**Table 4.16:** loo_compare output for models C.5 and 4.5

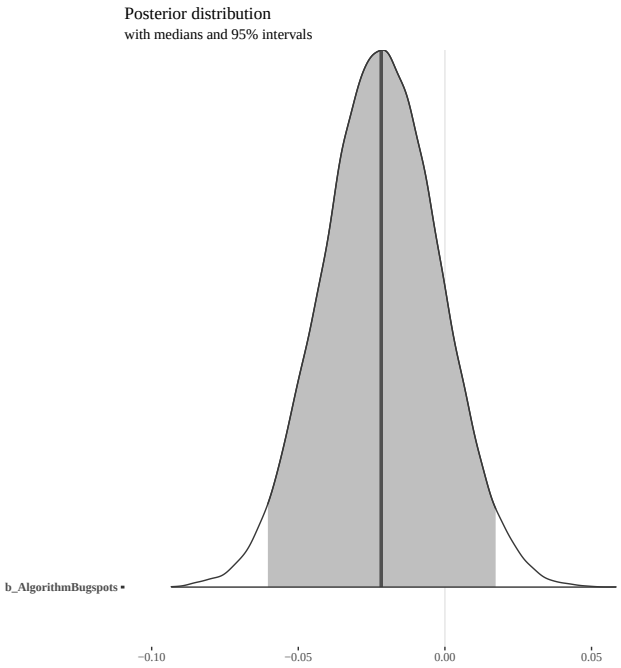
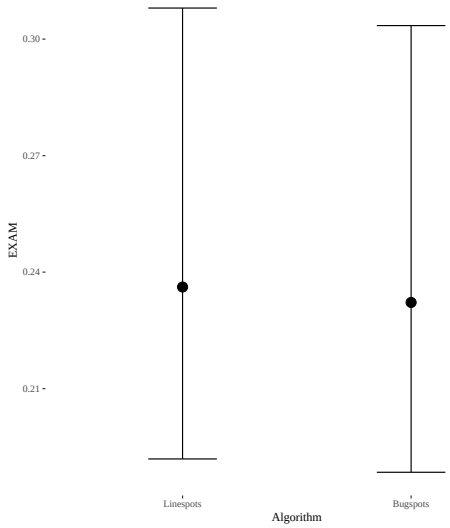



**Figure 4.14:** Posteriors of Bugspots

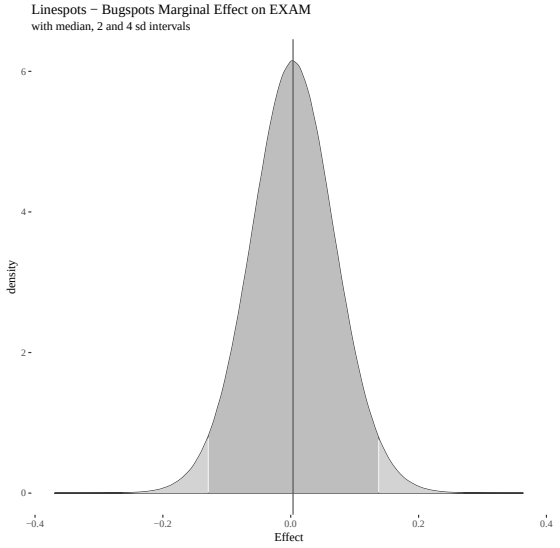
To get a better picture for any effects on the outcome scale, we again use the marginal effects shown in Figure 4.15a. It shows the mean and 95% intervals for the EXAM scores predicted by Model 4.5 and split by Algorithm. While the difference is not very big, both the mean and the 95% intervals are lower for Bugspots compared to Linespots. The y-axis scale, however, is very small, so any difference will be small as well. Finally, we can take a look at the difference between predictions on the outcome scale as shown in Figure 4.15b and summarized in Table 4.17. The contrast shows that the difference is very small and overlaps with 0.

| | -4 SD | -2 SD | Mean | +2 SD | +4 SD |
|---|---|---|---|---|---|
| Linespots - Bugspots | -0.263 | -0.13 | 0.00372 | 0.137 | 0.271 |

**Table 4.17:** Contrast between algorithms in Model 4.5 rounded to 3 significant digits




**(a)** Marginal effects of Bugspots and Linespots




**(b)** Linespots - Bugspots

**Figure 4.15:** Marginal effects and contrast based on posterior predictions for both algorithms in Model 4.5

### 4.5.2 AUCECEXAM Results

As indicated in Section 4.1 the results for AUCECEXAM are very similar to the ones using EXAM. For this reason, we only show the plots and tables here and refer to the explanations in Section 4.3.1.

$$
\begin{aligned}
\text{AUCECEXAM}_i &\sim \text{Beta}(\mu_i, \phi) \\
\text{logit}(\mu_i) &= \alpha + \beta_a A_i + \beta_l L_i + \gamma_{Project[i]} + \delta_{Language[i]} \\
\alpha &\sim \text{Normal}(0, 0.5) \\
\beta_a, \beta_l &\sim \text{Normal}(0, 0.5) \\
\gamma_j &\sim \text{Normal}(\bar{\gamma}, \sigma) \\
\delta_j &\sim \text{Normal}(\bar{\delta}, \epsilon) \\
\bar{\gamma}, \sigma, \bar{\delta}, \epsilon &\sim \text{HalfCauchy}(0, 0.1) \\
\phi &\sim \text{Gamma}(0.1, 0.1)
\end{aligned}
$$

**Model 4.6:** More Complex model for AUCECEXAM using Algorithm

| Model C.6 | Estimate | Est.Error | Q2.5 | Q97.5 |
|---|---|---|---|---|
| Intercept | 1.2 | 0.0955 | 1 | 1.38 |
| Bugspots | 0.0253 | 0.0202 | -0.0145 | 0.0651 |
| LOC | -0.0384 | 0.033 | -0.103 | 0.0266 |
| Model 4.6 | Estimate | Est.Error | Q2.5 | Q97.5 |
| Intercept | 1.19 | 0.166 | 0.802 | 1.47 |
| Bugspots | 0.0254 | 0.0201 | -0.014 | 0.0647 |
| LOC | -0.0434 | 0.0329 | -0.108 | 0.0206 |

**Table 4.18:** Fixed effects of models C.6 and 4.6 rounded to 3 significant digits

| | elpd_diff | se_diff |
|---|---|---|
| Model 4.6 | 0.0 | 0.0 |
| Model C.6 | -0.3 | 0.3 |

**Table 4.19:** loo_compare output for models C.6 and 4.6

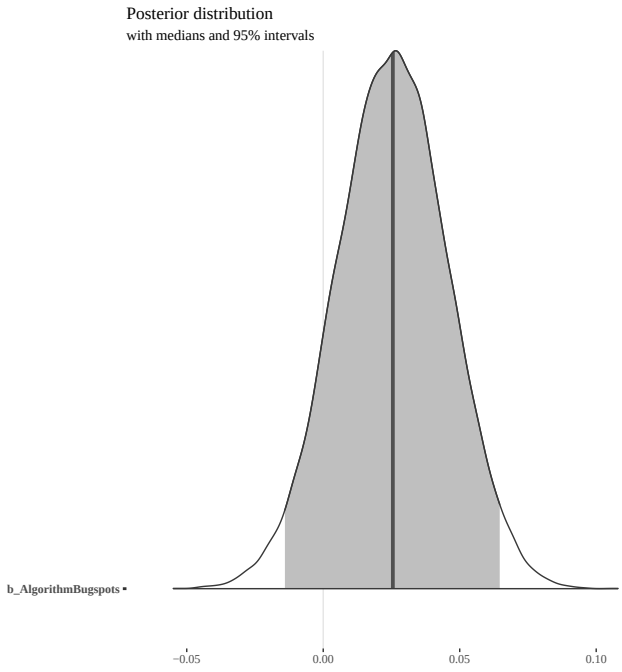
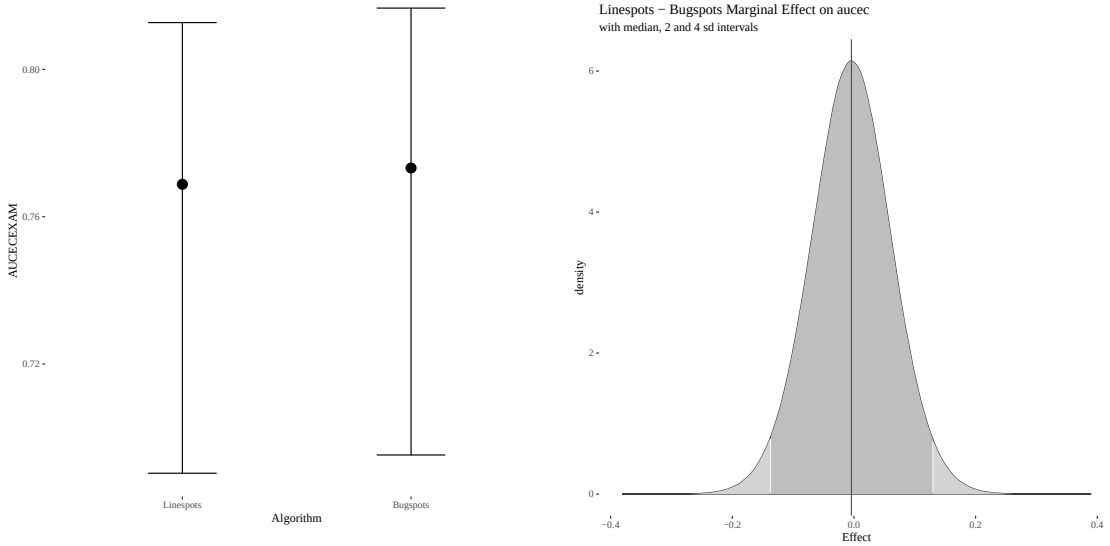



**Figure 4.16:** Posteriors of Bugspots

| | -4 SD | -2 SD | Median | +2 SD | +4 SD |
|---|---|---|---|---|---|
| Linespots - Bugspots | -0.271 | -0.138 | -0.00428 | 0.129 | 0.263 |

**Table 4.20:** Contrast between algorithms in Model 4.6 rounded to 3 significant digits




**(a)** Marginal effects of Bugspots and Linespots

**(b)** Linespots - Bugspots

**Figure 4.17:** Marginal effects and contrast based on posterior predictions for both algorithms in Model 4.6

### 4.5.3 RQ4: EXAM25 Results

The model building process for the EXAM25 models was done with less rigour compared to the EXAM and AUCECEXAM models so we only present the final model here. The full code is available together with the rest of the analysis [32]. We ended up using the same model structure as for EXAM and AUCECEXAM for sampling reasons as shown in Model 4.7.

$$
\begin{aligned}
\text{EXAM25}_i &\sim \text{Beta}(\mu_i, \phi) \\
\text{logit}(\mu_i) &= \alpha + \beta_a A_i + \beta_l L_i + \gamma_{Project[i]} \\
\alpha &\sim \text{Normal}(0, 0.5) \\
\beta_a, \beta_l &\sim \text{Normal}(0, 0.5) \\
\gamma_j &\sim \text{Normal}(\bar{\gamma}, \sigma) \\
\bar{\gamma}, \sigma &\sim \text{HalfCauchy}(0, 0.1) \\
\phi &\sim \text{Gamma}(0.1, 0.1)
\end{aligned}
$$

**Model 4.7:** Model for EXAM using Algorithm

We start with the posterior distribution summarized in Table 4.21. The posterior of the Bugspots coefficient in Figure 4.18 shows a positive effect on the logit scale as the entire probability mass is positive.

Moving to the outcome scale to better understand the effects, we again use the marginal effects shown in Figure 4.19a. It shows the mean and 95% intervals for

| Model 4.7 | Estimate | Est.Error | Q2.5 | Q97.5 |
|---|---|---|---|---|
| Intercept | -0.858 | 0.572 | -2 | 0.217 |
| Bugspots | 0.459 | 0.045 | 0.371 | 0.547 |
| LOC | -0.114 | 0.0812 | -0.273 | 0.0427 |

**Table 4.21:** Fixed effects of Model 4.7 rounded to 3 significant digits

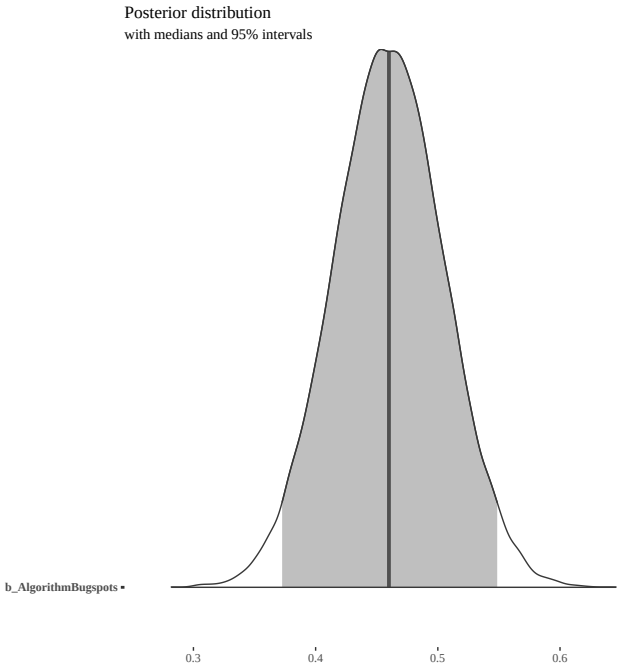
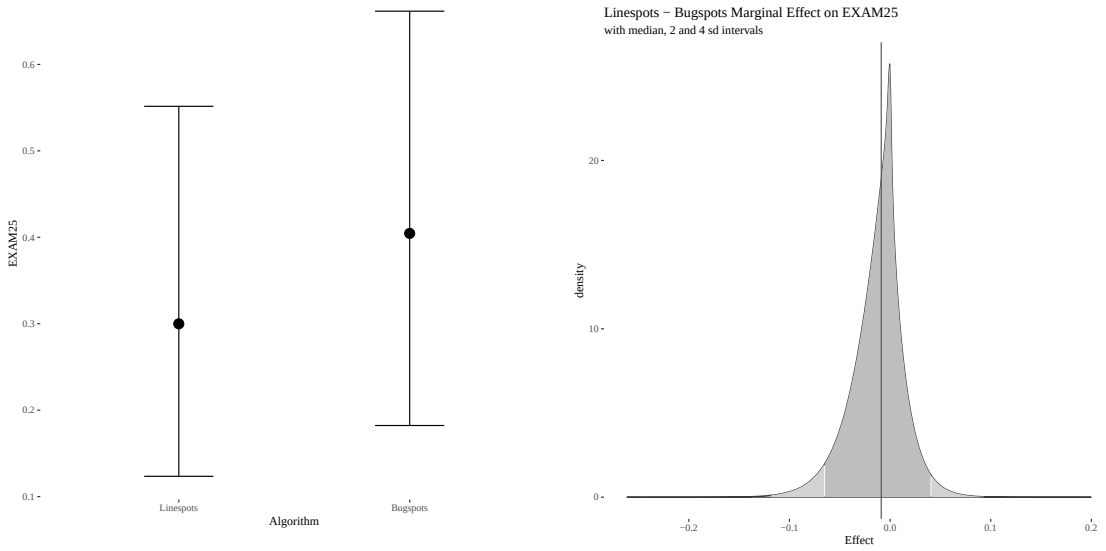



**Figure 4.18:** Posteriors of Bugspots

the EXAM25 scores predicted by Model 4.7 and split by Algorithm. This time the difference in means is around 0.1 which is quite a lot in terms of EXAM25, as it would indicate 10% fewer lines of code needed to find the first 25% of faults using Linespots. However, the 95% intervals are still overlapping a lot. Finally, we can take a look at the difference between predictions on the outcome scale as shown in Figure 4.19b and summarized in Table 4.22. While the contrast between Bugspots and Linespots is the biggest difference we have found so far, the 95% intervals still overlap with 0 so it is still reasonable to expect Bugspots to perform better here.




**(a)** Marginal effects of Bugspots and Linespots

**(b)** Linespots - Bugspots

**Figure 4.19:** Marginal effects and contrast for both algorithms based on posterior predictions in in Model 4.7

| | -4 SD | -2 SD | Mean | +2 SD | +4 SD |
|---|---|---|---|---|---|
| Linespots - Bugspots | -0.118 | -0.0652 | -0.0122 | 0.0407 | 0.0936 |

**Table 4.22:** Contrast between algorithms in Model 4.7 rounded to 3 significant digits

### 4.5.4 RQ4: EInspect25EXAM Results

The model building process for the EInspect25EXAM models was done with less rigour compared to the EXAM and AUCECEXAM models so we only present the final Model 4.8 here.

We start with the posterior distribution summarized in Table 4.23. The posterior of the Bugspots coefficient in Figure 4.18 shows a negative effect on the logit scale as the entire probability mass is positive.

$$
\begin{aligned}
\text{EInspect25EXAM}_i &\sim \text{Poisson}(\mu_i) \\
\log(\mu_i) &= \alpha + \beta_a A_i + \beta_l L_i + \gamma_{Project[i]} + \delta_{Language[i]} \\
\alpha &\sim \text{Normal}(0, 0.5) \\
\beta_a, \beta_l &\sim \text{Normal}(0, 0.5) \\
\gamma_j &\sim \text{Normal}(\bar{\gamma}, \sigma) \\
\delta_j &\sim \text{Normal}(\bar{\delta}, \epsilon) \\
\bar{\gamma}, \sigma, \bar{\delta}, \epsilon &\sim \text{HalfCauchy}(0, 0.1)
\end{aligned}
$$

**Model 4.8:** More Complex model for AUCECEXAM using Algorithm

| Model 4.8 | Estimate | Est.Error | Q2.5 | Q97.5 |
|---|---|---|---|---|
| Intercept | -0.215 | 0.536 | -1.26 | 0.847 |
| Bugspots | -1.02 | 0.194 | -1.41 | -0.644 |
| LOC | -0.805 | 0.236 | -1.3 | -0.371 |

**Table 4.23:** Fixed effects of Model 4.8 rounded to 3 significant digits

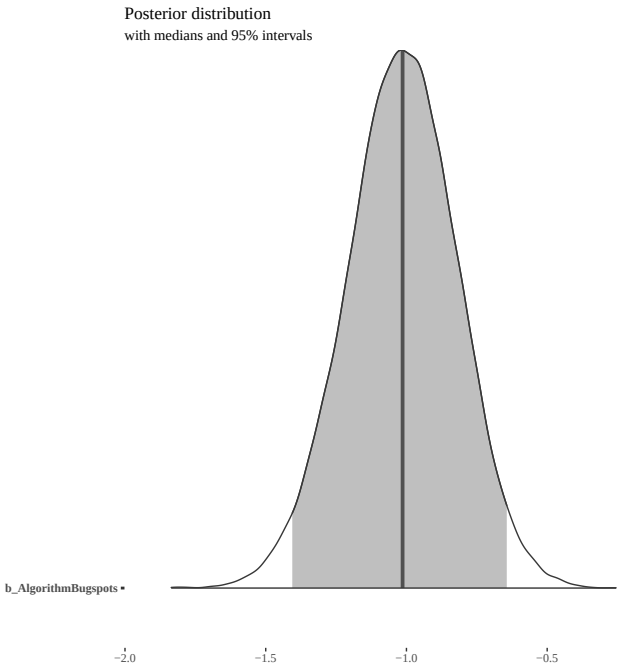
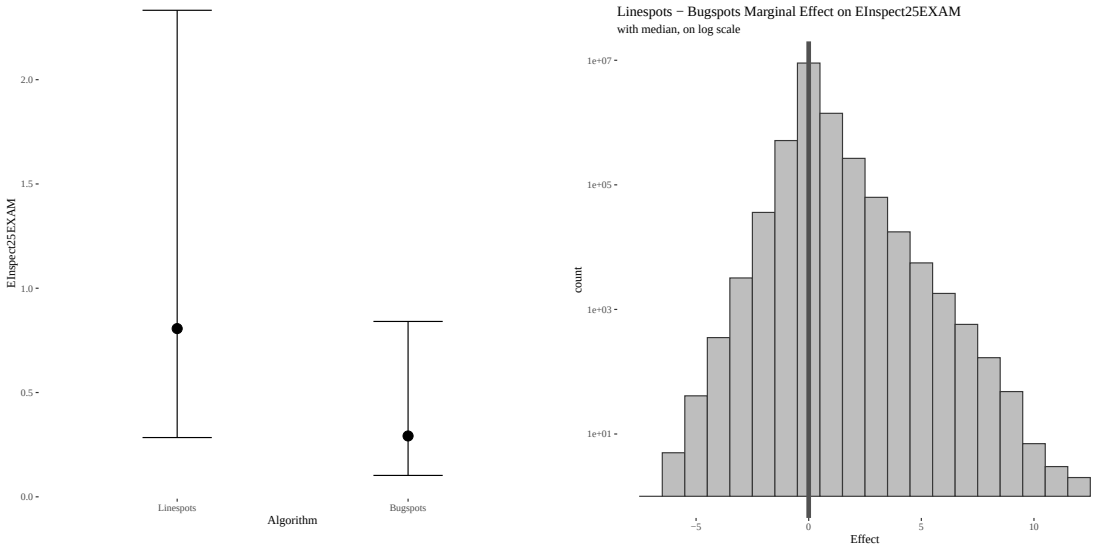



**Figure 4.20:** Posteriors of Bugspots

Moving to the outcome scale to better understand the effects, we again use the marginal effects shown in Figure 4.21a. It shows the mean and 95% intervals for the EInspect25EXAM scores predicted by Model 4.8 and split by Algorithm. For EInspect25EXAM both the means and 95% intervals differ a lot between Bugspots and Linespots. Both indicate a higher EInspect25EXAM score for Linespots. Finally, we can take a look at the difference between predictions on the outcome scale as shown in Figure 4.21b and summarized in Table 4.24. Here it is important to note that the y-axis is on the log scale, which might make a comparison hard on first glance. However, an easy way to read the plot would be to pairwise compare the height of the bars on the positive and negative side, which shows a higher number of positive contrasts than negative ones. The summary in Table 4.24 might be easier to read and that while again, the difference between Bugspots and Linespots is not reliable, the chance of Linespots outperforming Bugspots is higher compared to the reverse.




**(a)** Marginal effects of Bugspots and Linespots

**(b)** Linespots - Bugspots

**Figure 4.21:** Marginal effects and contrast based on posterior predictions for both algorithms in Model 4.8

| | -4 SD | -2 SD | Mean | +2 SD | +4 SD |
|---|---|---|---|---|---|
| Linespots - Bugspots | -2.24 | -1.05 | 0.145 | 1.34 | 2.53 |

**Table 4.24**

## 4.6 Research Question 5

The fifth research question was: Do Bugspots and Linespots predict faults in the same order?

For research question 5 we calculated the mean absolute rank difference of faults for each sample pair as shown in Figure 4.22. The results are summarized in Table 4.25. One way to interpret these numbers is that the mean distance between a fault's rank in the Bugspots and Linespots list is 27.5 slots. It is important to note here that a single fault that changes rank between the lists will automatically lead to subsequent faults to also be off by at least 1 rank which probably inflates this score.

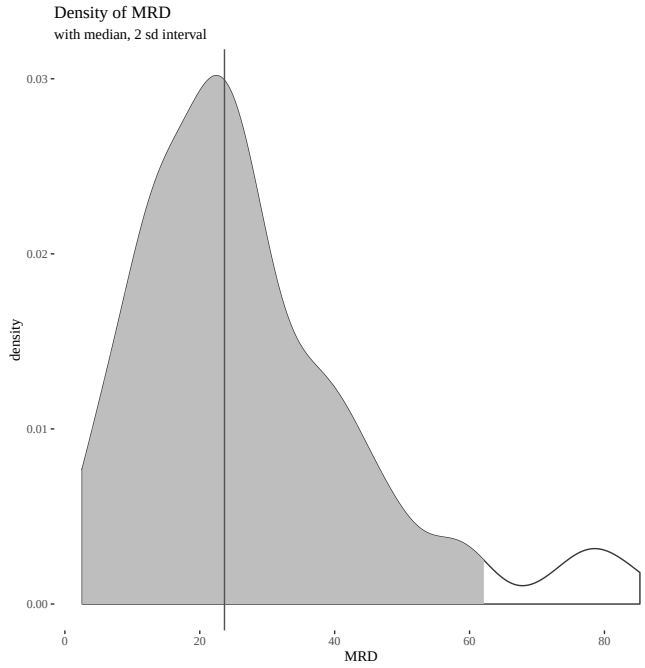



**Figure 4.22:** Mean absolute rank difference of faults between Bugspots and Linespots

| | -4 SD | -2 SD | Mean | Median | +2 SD | +4 SD |
|---|---|---|---|---|---|---|
| MRD | 0 | 0 | 27.5 | 23.7 | 62.2 | 97 |

**Table 4.25:** Mean, Median, 2 and 4 SD Intervals rounded to 3 significant digits or integers

# 5

# Discussion

In this chapter we will discuss the implications and possible causes of the findings from Chapter 4 as well as possible threats to validity and limitations and proposals for future work. The chapter is divided into the different research questions, threats and future work to allow easier navigation.

## 5.1 What kind of weighting function produces the best results for Linespots?

For research question 1 we compared the impact of the three different weighting functions proposed in Section 2.3.1 on the performance of Linespots.

Both the EXAM and the AUCECEXAM results in Section 4.2 show no clear positive or negative effect for any of the weighting functions, neither on the logit nor on the outcome scale. While the Google-weighting-function has the best mean and median results, we can expect either of the three weighting functions to outperform any other by around 0.086 for both EXAM and AUCECEXAM, based on the 2 standard deviation intervals in tables 4.4 and 4.7.

This finding does not match the experience described by Lewis *et al.* [15], however, this could be due to multiple reasons. One possible explanation for the lack of an effect is the depth we used in this experiment. It is possible that the effect of different weighting functions becomes more obvious when using a more commits for the analysis, eg, when using the entire commit history of a project. This would add older commits to the analysis that are less relevant to the present and could profit from having less impact on the results. Lewis *et al.* [15] did not specify what depth they used in their experiments and only gave a 6 to 8 month window until commits "become inconsequential" [15]. The Bugspots reference implementation [28], it only uses a depth of 500 commits in contrast to the 2000 commits we use. If we assumed this to be the depth that Lewis *et al.* [15] used the effect of weighting-functions should still be more pronounced for larger depth parameters.

Finally, there could be an interaction or masking effect at work that we have not controlled for in our models.

In summary, it can be stated that there is no reliable difference in EXAM and

AUCECEXAM performance between the different weighting-functions in our sample. And while we do recommend the usage of the Google-weighting-function, this is due to the experience of Lewis *et al.* [15] and a very small mean improvement over the other two, as well as our intuitive agreement with the arguments for an exponential weighting function.

> There is no reliable difference in EXAM and AUCECEXAM performance between the different weighting-functions for Linespots in our Sample.

## 5.2 How does index-based age calculation influence the predictive performance of Linespots compared to time stamp based age calculation?

For research question 2 we compared the impact of the two different time-versions proposed in Section 2.3.2 on the performance of Linespots.

Both the EXAM and the AUCECEXAM results in Section 4.3 show no clear positive or negative effect for both time-versions, neither on the logit nor on the outcome scale. While the commit based time-version has a slightly better mean and median marginal effects, the contrast is too small to be relevant. And as with the weighting-functions, we can expect either time-version to outperform the other by around 0.086 for both EXAM and AUCECEXAM, based on the 2 standard deviation intervals in tables 4.10 and 4.13.

After not finding an effect for different weighting-functions, this finding is almost expected. We would expect differences in relative commit age calculation to only have a small effect compared to the difference between a flat and an exponential weighting function. And as the time-versions mainly influence how the weighting-functions work, it follows our intuition that they would not influence the performance if the weighting-functions themselves do not influence the performance.

Besides the performance of Linespots, there is however an advantage to the index-based age calculation when implementing the algorithm, as it does not suffer from the problems that the non-linear history of git poses for the time-based age calculation. As the age of commits does not grow monotonically, one has to care for edge cases. And with our results pointing to no performance gains from any of the two, we do recommend the usage of the index-based age calculation in the future.

In summary, it can be said that there is no reliable difference between the two time-versions in regards to EXAM and AUCECEXAM performance, we do however recommend the usage of the index-based age calculation as it is easier to implement.

> There is no reliable difference in EXAM and AUCECEXAM performance between the two time-versions for Linespots in our Sample.

## 5.3 What is a good cut-off-point to turn Linespots into a classifier?

For research question 3 we analyzed Linespots' behaviour at different interesting points to find guidelines for how to choose a good cut-off point to turn Linespots into a classifier.

There is no clear answer to this question by nature, as it depends on the needs of the user of the classifier. Depending on the chosen cut-off point the resulting performance metrics will differ and it is up to the user to choose which ones are most important.

We split the choice for a cut-off point into two parts. The first part is the choice of a lower limit under which we expect Linespots not to offer any faults as part of the result list. The second part is an analysis of Linespots behaviour at different possible cut-off points.

Starting with the lower limit, we look at both the EXAMF and hdMaxLOCEXAM results in Figure 4.12. Using the standard deviation intervals as a guide, using 1.48% LOC to leave us with an approximate 97.5% chance to have at least one fault in our faulty class. This is probably a good starting point for most use cases and depending on how well a project works with Linespots, this could potentially be lowered by two orders of magnitude or more.

Moving from the lower limit to a more optimal point that gives a better ratio of value to effort, we can use both the hdMaxLOCEXAM as the highest ratio of found faults per predicted line and the EXAM25 as the percentage of LOC needed to find the first 25% of faults. The median of hdMaxLOCEXAM at 0.0385% LOC is not too far from the median of EXAMF and thus could lead to a high number of useless classifications without a single hit. Using the 2 standard deviation interval for hdMaxLOCEXAM would result in a cut-off at 9.26% LOC to include the point of highest efficiency in your classifier. This, however, does not seem useful as the point of highest efficiency is only useful if we can reliably get very close to it. The high standard deviation for hdMaxLOCEXAM makes that hard to achieve. Instead, we can look at EXAM25 and use the 2 standard deviation interval at 6.5% LOC to find 25% of faults in approximately 97.5% of classifier runs. This 6.5% cut-off point is also rather close to the 5% used by, among others, Arisholm *et al.* [7].

Finally, we present the precision and recall at 5% LOC in Figure 4.13b and Table 4.14. These numbers are not impressive and indicate a high number of false positives and false negatives.

Ultimately, the performance requirements of a classifier have to be estimated by each user and we did not expect to be able to give any definitive answer to this question. If one were to ask us for default cut-off point recommendation, we would suggest below 5%LOC as it lies in the range between EXAMF, EXAM25 and hdMaxLOCEXAM for most projects. However, we discourage users from using Linespots as a sole classifier in practice. It is rather suited as a baseline when comparing other fault

prediction or fault localization algorithms, or as part of an ensemble of algorithms that work together.

> We discourage users from using Linespots as a sole classifier in practice due to the bad performance.

## 5.4 What is the prediction performance of Linespots compared to Bugspots?

For the fourth research question, we compared the predictive performance of Bugspots and Linespots.

Starting with the EXAM and AUCECEXAM results, both of which are types of averaging metrics, we see stronger differences than we have seen for the weighting-functions and time-versions. While the posterior distributions for Bugspots in figures 4.14 and 4.16 put most of the weight on the negative side, there is still overlap with 0 and when moving to the outcome scale, there is only a very small difference left. While Bugspots' mean and median performance is slightly better than Linespots' the difference is not reliable and very small.

These findings are initially counter-intuitive as they both run against our preliminary findings in the past [18] as well as our argumentations for the theoretical benefits of Linespots over Bugspots. We do however have a theory for the cause of these results and why they seem to contradict past findings.

First, it is important to note that in our past work, we used the AUCECDENSITY version of cost-effectiveness instead of AUCECEXAM. While we expect them to behave similarly, they do not behave the same, as shown in Section 4.1. Second, we did not use the full AUCEC in the past, but instead the AUCEC at 5% LOC or AUCEC5. This emphasizes the early parts of the result list. While we do not expect the difference between the AUCECDENSITY and AUCECEXAM measures to result in such different results, the difference between the averaging style of the AUCECEXAM used here and the AUCEC5 used in our past work does have the potential to cause this.

The reason for the different behaviour between Bugspots and Linespots could stem from the difference in their granularity and how the EXAM score is calculated. Both algorithms produce a ranked list of elements for both of them, a large number of elements at the end of the list will have the score 0 as they either have never been part of a fix inducing commit or they never were modified during the length of the depth parameter used at all. Due to the finer granularity of Linespots, this part of the list with score 0 will be bigger. If we were to assume that Linespots performs better for the early parts of the result list than Bugspots, this could, in theory, be countered by a better performance of Bugspots in the later part, as both the AUCEC and EXAM metrics average across all faults. As the EXAM score, and thus also the AUCECEXAM, assumes random ordering of lines with the same score, faults with

only 0 line scores get rather high $E_{inspect}$ scores. This is because they are assumed to be randomly encountered anywhere in the block with score 0. As Bugspots has more lines with a score above 0 than Linespots, any fault that is encountered in those extra lines has a much lower $E_{inspect}$ value than if it were in the last block. To clarify the performance of both algorithms in the early parts of the result list, we also analyze their performance with the EXAM25 and the EInspect25EXAM metrics.

Starting with the EXAM25 results, we have a positive effect for Bugspots on the logit scale as shown in Figure 4.18. When moving to the outcome scale that effect is not as pronounced any more, probably due to the inverse logit transformation that includes the intercept of the model. And while the means and 95% intervals between the two algorithms are offset, the 95% intervals do overlap and the contrast does overlap 0 substantially. When choosing between the two, however, Linespots does produce better mean and median EXAM25 results. It is also important to note that the median EXAM25 score for both Bugspots and Linespots is 0.0164 so while the contrast might seem small, the mean of -0.0122 is rather big in relation. The EInspect25EXAM results follow right in those tracks and show a clear negative effect for Bugspots on the logit scale and a less reliable effect on the outcome scale. So while the 95% intervals overlap 0, Linespots does outperform Bugspots in both mean and median EInspect25EXAM.

Both the EXAM25 and the EInspect25EXAM results support the idea that Linespots performs better during the earlier parts of the result list than Bugspots. This also follows our line of reasoning that Linespots performs better for the early stages of the list but Bugspots performs better for the later stages due to a smaller 0 score block.

Combined we interpret the results such that Linespots does not offer improvements over Bugspots across all faults but does perform very similar to it on average. But when considering only the first parts of the result list, Linespots does seem to outperform Bugspots on average, which would be an improvement based on the finding of Parnin and Orso [11] that programmers only consider a small number of early entries in a ranked list.

> Linespots outperforms Bugspots for metrics that focus on early parts of the result list. For averaging metrics, both perform similar with a small lead for Bugspots.

## 5.5 Do Bugspots and Linespots predict faults in the same order?

For the fifth research question, we investigated if the order of the predicted faults is the same for Bugspots and Linespots.

Based on our results, the order is not the same for the two which was expected. We assume that this is due to the same reason that both algorithms perform differently

when compared with EXAM and EXAM25 as discussed in Section 5.4. As Bugspots has a lower granularity, it casts a wider net so to say and scores lines that Linespots does not score. This will inevitably lead to different rankings of lines in the project and thus to different rankings of predicted faults.

This difference might make it possible, to improve the performance of the simple past fault approach further by combining different granularities and utilizing their strengths.

| Bugspots and Linespots usualdo not predict faults in the same order. |
|---|

## 5.6 Other Observations

In this section, we collect other noteworthy observations that do not fit any research question directly.

### 5.6.1 Language and Domain Differences

While we did not do a thorough analysis of the differences between programming languages and domains in regards to Linespots' performance, the boxplots in Figure 4.3 show that there could be both programming languages and domains that work better with Linespots. It is important to note here that our sample size is far too small to draw any strong conclusions from this alone. It is, however, indicative that JavaScript and C++, as well as projects in the data processing & outsourced services domain, do for some reason work exceedingly well Linespots.

Looking at Table 3.1 we can see that there is some overlap between the projects using JavaScript and C++ and the project in the data processing & outsourced services domain. So the better performance of those languages and domain could just be due to the better performance of those few projects.

## 5.7 Threats to Validity and Limitations

### 5.7.1 Faults in the Implementation

It has happened before that a paper is retracted due to bugs in the implementation of some algorithm or analysis [16] so it is a valid critique of this thesis as well to ask if the results, and thus the drawn conclusions, are based on a faulty implementation. As our code has not been formally verified, nor has it undergone an exhaustive review and audit process, we can not say with certainty that it does not contain faults. The code is also not fully tested, which could be have been used as an argument for its reliability.

However, we are confident that our long experience with the algorithm and throughout the course of multiple implementations, each improving on the past ones, we have reduced the number of faults compared to past works [18]. We also have the work of Zou *et al.* [26] to compare our Bugspots results against. While Zou *et al.* [26] report an EXAM score of 0.465 for Bugspots, our results for the commons-math, closure-compiler and jfreechar projects have a mean of 0.295 and median of 0.274 with a standard deviation of 0.0496. While Zou *et al.* [26]'s result is outside of our 95% interval, we have used a depth of 2000 while Zou *et al.* [26] have used the default of 500. As we found both Bugspots and Linespots to perform better as argued in Section 3.6.4, this difference in depth could explain the difference in performance. We also only used the commit message identification as explained in Section 2.2.1 while Zou *et al.* [26] used a predefined set of known faults. As the identification through commit message is not a perfect process, this is prone to result in different sets of faults, which will influence the results. We can not say, however if this would increase or decrease our performance compared to Zou *et al.* [26]. Based on these arguments and our experience in the field, we argue that our approach is in line with the common process of developing implementations in the wider field of software engineering. And while not optimal, we think that this approach is reliable enough to trust the results in the given circumstances.

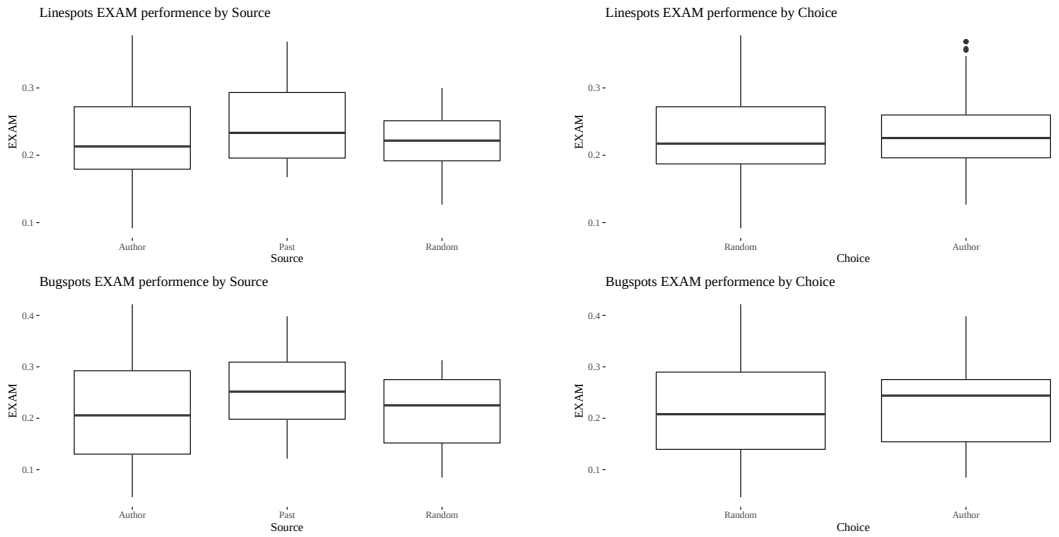
### 5.7.2 Training on Evaluation Data

While developing the implementation for Linespots we ran into multiple corner cases that we only encountered with a single project. One could argue that by running the algorithm on one such a project and improving it based on the result we received, we tuned the algorithm to perform better on said project and thus effectively trained the algorithm on our evaluation data.

As we had to implement Linespots ourselves and use Git, which is anything but simple, we were prone to encounter bugs and corner cases along the way. To reduce the risk of biasing our results by optimizing for the projects we had, we made sure not to analyze the performance of Linespots before the implementation did return valid results from all projects. We also chose the larger projects with more commits for the test runs, so that there was less of a chance to use the same commits in the final evaluation that we also used for testing during development. Finally, we chose the commits that were analyzed for the final evaluation randomly, to reduce the risk of bias by choosing the commits ourselves.

### 5.7.3 Sourcing of Training and Validation Data

Another possible critique for our process of data gathering is the fact that we use the same algorithm to identify fix inducing commits for the training data, the depth commits that Linespots analyses, as well as the validation data, the pseudo future we use to find future fixes. As mentioned in Section 2.2.1 the process of identifying fix inducing commits based on their commit message is not ideal. In addition to




**(a)** Source **(b)** Choice

**Figure 5.1:** Influence of Source and Choice on EXAM results for Bugspots and Linespots

that, the fact that we use the same process to gather both the predictions and the supposed truth might bias our results even further. We acknowledge this restriction and want to specify that our results only indicate the predictive power of those commits that were identified via their commit message and not to all possible faults. We propose some improvements to this situation in the future work section.

### 5.7.4 Impact of Source and Choice

Figures 5.1a and 5.1b show the EXAM results for the different sources and choices explained in Section 3.4.3. We argue that the differences between sources and choices across algorithms is small and thus does not weaken our conclusions.

### 5.7.5 GitHub as Data Source

The choice of GitHub as the source for all our project brings two limitations with it. First, all projects that we analyzed use git, which could skew the collected data. However, Git seems to be the most used version control system [36], [40] and thus the results of this thesis should apply to the majority of projects. Second, with other git hosting services like Gitlab or Bitbucket available, our choice of focusing on Github again could limit the generalizability of this thesis. We argue, that our sample size and spread across domains and programming languages minimizes this problem. Furthermore, our inclusion of randomly chosen projects reduces potential bias in our dataset.

### 5.7.6 Bad Smells

In this section we will shortly address some of the bad smells identified by Menzies and Shepperd [24].

**Not Using Related Work:** We based both the development of Linespots itself as well as the methods we used on past work in the field of fault detection as referenced throughout the background and method sections.

**Using Deprecated and Suspect Data:** We build a new dataset based on projects used in past work, random projects from Github and projects that are well known in their respective communities. We documented the process of how we build the dataset in Section 3.4 and analyzed the impact of different sources and choices in Section 5.7.4.

**Inadequate Reporting:** We report all gathered results that we saw as related to the research questions and otherwise noteworthy. In addition to that, both the Linespots and evaluation suite implementations are openly accessible [39] as well as the R code used for the analysis [32].

**Not Exploring Stability:** We conducted a prior sensitivity analysis for our models as explained in Section 3.9.1.

**Not Exploring Simplicity:** We compared models of different complexity in our analysis and argued for the choice of our models based on their loo performance and sampling behavior.

## 5.8 Future Work

### 5.8.1 Standard Evaluation Suite

Not only has the evaluation suite that we built for this thesis saved a lot of manual labour but it also allows for the reproduction of our results with one script. To prevent every researcher from having to build their own evaluation suite, an effort to build a standard evaluation suite for fault prediction algorithms could greatly improve the comparability between studies and reduce the time spent on repeatedly building similar programs. While we do recognize, that this kind of standard evaluation suite will be able to fit every use case [43] we sill believe that there could be great value in this. The evaluation suite that we built for this thesis could serve as a starting point, as the architecture has been set up in a way to allow for more projects, algorithms and metrics to be added without much work.

### 5.8.2 Analyze Smaller Projects

One of the limitations of this thesis was the restriction on projects with at least 3000 commits. The problems with smaller projects from an evaluation point of view is

that the possible size of the pseudo future becomes smaller which might lead to less reliable results. However many on GitHub have less than 3000 commits and could potentially profit from the usage of an algorithm like Bugspots or Linespots. To get a better understanding of how Linespots performs for smaller, less mature projects, it would be useful to also do an evaluation including them.

### 5.8.3 Linespots Performance

One of the big benefits of Bugspots is the short run time and the simplicity of the algorithm. While Linespots might be a worthwhile alternative in regards to performance, the current iteration of the implementation has suffered severe performance regressions compared to the first iteration and now has a lot longer run times. To make Linespots a viable alternative to Bugspots, the performance has to be improved to be able to compete with Bugspots again. One way to improve the performance that we have already identified is to use the pydriller library [27] for all git related work in the algorithm. This would also simplify the implementation and make it easier for others to contribute.

### 5.8.4 Dataset Building

One of the big challenges not only for Linespots but for the entire field of fault prediction is the lack of good datasets to work with. Currently, researchers have the option to either go with a pre-built set like Defects4J [29] that consists of a collection of elements and information about faults contained in those elements or to build their own dataset as done in this thesis. The first approach severely limits the sample size and thus generalizability of the studies done. All of the pre-build datasets we encountered during our research consisted solely of projects written in Java and the number of projects in the sets was low. The benefit of these sets usually is more complete information about faults and the time savings from not building a new dataset.

The second approach, building a new dataset, allows for arbitrary sample size and project choice, which improves the generalizability of the results, but comes with the drawback, that techniques for extracting fault information from version history are underdeveloped. The identification through commit message, as used in this thesis, does work if projects use unique identifiers in their commit messages for different commit types. This is however not the case for many projects which makes this approach unreliable and filters our many projects again.

We propose two measures to improve the current situation. First, we recommend all practitioners to adapt some kind of commit message convention and enforce it for their projects. While there are other benefits to it as well, it would greatly improve the ability to identify fix inducing commits through commit messages. The best example that we encountered during this research is used by Discourse [22]. In addition to that, we think that a lot of useful information could be pulled from issue

trackers but there are no tools to support researchers to achieve this. For this reason, we think there could be a lot of value in more tools to automate dataset generation, similar to how it is done in this thesis. This would allow for larger samples sizes and a better distribution of projects to analyze and thus improve the reliability of the entire field.

# 6

# Conclusion

This thesis set out to deepen the understanding of Linespots and its capabilities as a fault prediction algorithm. The main goal was to analyze the performance of Linespots with a better-designed experiment with a larger sample and stronger statistical method. The second aim for this thesis was to answer questions that repeatedly came up during discussions of the algorithm and investigate if some of the ideas brought up during those discussions could improve the performance of Linespots notably. Finally, this study was undertaken to further investigate the similarities and differences between Bugspots and Linespots and thus a file-level and a line-level granularity.

Our analysis shows that there are no consistent effects for both the time-versions and the weighting-functions, but there is a lot of uncertainty in the results so differences in performance between both time-versions and weighting-functions can be expected for individual projects. Since this thesis was limited to only analyzing the impact of the time-versions and weighting-functions, we have not identified the cause of this uncertainty. This thesis also shows that, while a cut-off point around 5% LOC might be a good default to use Linespots as a classifier, the performance of the classification is low. When comparing Bugspots to Linespots, we found that their performance for the averaging metrics was similar. However, when analyzing the EXAM25 and EInspect25EXAM Linespots performs better than Bugspots which indicates more predicted faults in early parts of the result list. Still, the results are uncertain enough that either algorihm could perform better depending on the project. Finally, we found that the order of predicted faults differs substantially between Bugspots and Linespots.

The practical implications of these findings are the usage of the index-based time-version, due to simpler implementation and small average performance lead, as well as the Google-weighting-function based on a small average performance lead and the results of Lewis *et al.* [15]. In addition to that, we discourage the use of Linespots as a classifier in production environments due to its low performance. Based on the findings of Lewis *et al.* [15] we also discourage the usage of Linespots for code inspection and instead agree with it being used to focus code testing efforts. Lastly, future research in the field of fault prediction should consider using Linespots as a baseline for fault prediction instead of Bugspots as it does show the potential to perform better for the more important early parts of the result list.

# Bibliography


[1] J. T. Nosek and P. Palvia, "Software maintenance management: Changes in the last decade", *Journal of Software Maintenance: Research and Practice*, vol. 2, no. 3, pp. 157–174, Sep. 1990, ISSN: 1040550X, 1096908X. DOI: 10.1002/smr.4360020303. [Online]. Available: http://doi.wiley.com/10.1002/smr.4360020303 (visited on 05/07/2019).

[2] N. Juristo and A. M. Moreno, *Basics of Software Engineering Experimentation*. Boston, MA: Springer US, 2001. DOI: 10.1007/978-1-4757-3304-4. [Online]. Available: http://link.springer.com/10.1007/978-1-4757-3304-4 (visited on 08/31/2019).

[3] N. Nagappan and T. Ball, "Use of Relative Code Churn Measures to Predict System Defect Density", in *Proceedings of the 27th International Conference on Software Engineering*, (St. Louis, MO, USA), ser. ICSE '05, New York, NY, USA: ACM, 2005, pp. 284–292, ISBN: 978-1-58113-963-1. DOI: 10.1145/1062455.1062514. [Online]. Available: http://doi.acm.org/10.1145/1062455.1062514 (visited on 07/16/2019).

[4] S. Kim, T. Zimmermann, E. J. Whitehead Jr., and A. Zeller, "Predicting Faults from Cached History", in *29th International Conference on Software Engineering (ICSE'07)*, Minneapolis, MN, USA: IEEE, May 2007, pp. 489–498, ISBN: 978-0-7695-2828-1. DOI: 10.1109/ICSE.2007.66. [Online]. Available: http://ieeexplore.ieee.org/document/4222610/ (visited on 05/06/2019).

[5] E. Wong, T. Wei, Y. Qi, and L. Zhao, "A Crosstab-based Statistical Method for Effective Fault Localization", in *And Validation 2008 1st International Conference on Software Testing, Verification*, Apr. 2008, pp. 42–51. DOI: 10.1109/ICST.2008.65.

[6] D. J. Hand, "Measuring classifier performance: A coherent alternative to the area under the ROC curve", *Machine Learning*, vol. 77, no. 1, pp. 103–123, Oct. 2009, ISSN: 0885-6125, 1573-0565. DOI: 10.1007/s10994-009-5119-5. [Online]. Available: http://link.springer.com/10.1007/s10994-009-5119-5 (visited on 07/15/2019).

[7] E. Arisholm, L. C. Briand, and E. B. Johannessen, "A systematic and comprehensive investigation of methods to build and evaluate fault prediction models", *Journal of Systems and Software*, vol. 83, no. 1, pp. 2–17, Jan. 2010, ISSN: 01641212. DOI: 10.1016/j.jss.2009.06.055. [Online]. Available:

http://linkinghub.elsevier.com/retrieve/pii/S0164121209001605 (visited on 11/17/2018).

[8] M. D'Ambros, M. Lanza, and R. Robbes, "An extensive comparison of bug prediction approaches", in *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, Cape Town, South Africa: IEEE, May 2010, pp. 31–41, ISBN: 978-1-4244-6802-7. DOI: 10.1109/MSR.2010.5463279. [Online]. Available: http://ieeexplore.ieee.org/document/5463279/ (visited on 11/17/2018).

[9] C. Lewis and R. Ou. (Dec. 14, 2011). Bug Prediction at Google, [Online]. Available: http://google-engtools.blogspot.com/2011/12/bug-prediction-at-google.html (visited on 05/06/2019).

[10] C. Parnin and A. Orso, "Are Automated Debugging Techniques Actually Helping Programmers?", in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, (Toronto, Ontario, Canada), ser. ISSTA '11, New York, NY, USA: ACM, 2011, pp. 199–209, ISBN: 978-1-4503-0562-4. DOI: 10.1145/2001420.2001445. [Online]. Available: http://doi.acm.org/10.1145/2001420.2001445 (visited on 07/17/2019).

[11] ——, "Are automated debugging techniques actually helping programmers?", in *Proceedings of the 2011 International Symposium on Software Testing and Analysis - ISSTA '11*, Toronto, Ontario, Canada: ACM Press, 2011, p. 199, ISBN: 978-1-4503-0562-4. DOI: 10.1145/2001420.2001445. [Online]. Available: http://portal.acm.org/citation.cfm?doid=2001420.2001445 (visited on 08/15/2019).

[12] F. Rahman, D. Posnett, A. Hindle, E. Barr, and P. Devanbu, "BugCache for Inspections: Hit or Miss?", in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, (Szeged, Hungary), ser. ESEC/FSE '11, New York, NY, USA: ACM, 2011, pp. 322–331, ISBN: 978-1-4503-0443-6. DOI: 10.1145/2025113.2025157. [Online]. Available: http://doi.acm.org/10.1145/2025113.2025157 (visited on 04/26/2019).

[13] M. DAmbros, M. Lanza, and R. Robbes, "Evaluating defect prediction approaches: A benchmark and an extensive comparison", *Empirical Software Engineering*, vol. 17, no. 4-5, pp. 531–577, Aug. 2012, ISSN: 1382-3256, 1573-7616. DOI: 10.1007/s10664-011-9173-9. [Online]. Available: http://link.springer.com/10.1007/s10664-011-9173-9 (visited on 02/08/2019).

[14] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, "A Systematic Literature Review on Fault Prediction Performance in Software Engineering", *IEEE Transactions on Software Engineering*, vol. 38, no. 6, pp. 1276–1304, Nov. 2012, ISSN: 0098-5589, 1939-3520. DOI: 10.1109/TSE.2011.103. [Online]. Available: http://ieeexplore.ieee.org/document/6035727/ (visited on 11/17/2018).

[15] C. Lewis, Z. Lin, C. Sadowski, X. Zhu, R. Ou, and E. J. Whitehead, "Does bug prediction support human developers? Findings from a Google case study", in *2013 35th International Conference on Software Engineering (ICSE)*, May 2013, pp. 372–381. DOI: 10.1109/ICSE.2013.6606583.

[16] S. for Neuroscience, "Author-Initiated Retraction: Anderson et al, Induced Alpha Rhythms Track the Content and Quality of Visual Working Memory Representations with High Temporal Precision", *Journal of Neuroscience*, vol. 35, no. 6, pp. 2838–2838, Feb. 11, 2015, ISSN: 0270-6474, 1529-2401. DOI: 10.1523/JNEUROSCI.0074-15.2015. pmid: 25673870. [Online]. Available: https://www.jneurosci.org/content/35/6/2838 (visited on 07/24/2019).

[17] T.-D. B. Le, D. Lo, C. Le Goues, and L. Grunske, "A Learning-to-rank Based Fault Localization Approach Using Likely Invariants", in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, (Saarbrücken, Germany), ser. ISSTA 2016, New York, NY, USA: ACM, 2016, pp. 177–188, ISBN: 978-1-4503-4390-9. DOI: 10.1145/2931037.2931049. [Online]. Available: http://doi.acm.org/10.1145/2931037.2931049 (visited on 07/17/2019).

[18] M. Scholz, "A Line Based Approach for Bugspots", p. 61, Oct. 2016. [Online]. Available: http://www.sts.tuhh.de/pw-and-m-theses/2016/scholz16.pdf (visited on 02/04/2019).

[19] Z. Tóth, P. Gyimesi, and R. Ferenc, "A Public Bug Database of GitHub Projects and Its Application in Bug Prediction", in *Computational Science and Its Applications – ICCSA 2016*, O. Gervasi, B. Murgante, S. Misra, A. M. A. Rocha, C. M. Torre, D. Taniar, B. O. Apduhan, E. Stankova, and S. Wang, Eds., vol. 9789, Cham: Springer International Publishing, 2016, pp. 625–638. DOI: 10.1007/978-3-319-42089-9_44. [Online]. Available: http://link.springer.com/10.1007/978-3-319-42089-9_44 (visited on 04/26/2019).

[20] P.-C. Bürkner, "Brms: An R Package for Bayesian Multilevel Models Using Stan", *Journal of Statistical Software*, vol. 80, no. 1, pp. 1–28, Aug. 29, 2017, ISSN: 1548-7660. DOI: 10.18637/jss.v080.i01. [Online]. Available: https://www.jstatsoft.org/index.php/jss/article/view/v080i01 (visited on 08/31/2019).

[21] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller, "Evaluating and Improving Fault Localization", in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, Buenos Aires: IEEE, May 2017, pp. 609–620, ISBN: 978-1-5386-3868-2. DOI: 10.1109/ICSE.2017.62. [Online]. Available: http://ieeexplore.ieee.org/document/7985698/ (visited on 05/08/2019).

[22] (Nov. 8, 2018). Discourse Development Contribution Guidelines, [Online]. Available: https://meta.discourse.org/t/discourse-development-contribution-guidelines/3823 (visited on 08/18/2019).

[23] R. McElreath, *Statistical Rethinking : A Bayesian Course with Examples in R and Stan*. Chapman and Hall/CRC, Jan. 3, 2018, ISBN: 978-1-315-37249-5. DOI: 10.1201/9781315372495. [Online]. Available: https://www.taylorfrancis.com/books/9781315372495 (visited on 08/14/2019).

[24] T. Menzies and M. Shepperd, "Bad Smells in Software Analytics Papers", Mar. 14, 2018. arXiv: 1803.05518 [cs]. [Online]. Available: http://arxiv.org/abs/1803.05518 (visited on 04/26/2019).

[25] J. Piironen, M. Paasiniemi, and A. Vehtari, "Projective Inference in High-dimensional Problems: Prediction and Feature Selection", Oct. 4, 2018. arXiv: 1810.02406 [cs, stat]. [Online]. Available: http://arxiv.org/abs/1810.02406 (visited on 07/24/2019).

[26] D. Zou, J. Liang, Y. Xiong, M. D. Ernst, and L. Zhang, "An Empirical Study of Fault Localization Families and Their Combinations", Mar. 27, 2018. arXiv: 1803.09939 [cs]. [Online]. Available: http://arxiv.org/abs/1803.09939 (visited on 05/06/2019).

[27] S. Davide, *Python Framework to analyse Git repositories. Contribute to ishepard/pydriller development by creating an account on GitHub*, Aug. 16, 2019. [Online]. Available: https://github.com/ishepard/pydriller (visited on 08/18/2019).

[28] I. Grigorik, *Implementation of simple bug prediction hotspot heuristic: Igrigorik/bugspots*, Aug. 15, 2019. [Online]. Available: https://github.com/igrigorik/bugspots (visited on 08/17/2019).

[29] R. Just, *A Database of Real Faults and an Experimental Infrastructure to Enable Controlled Experiments in Software Engineering Research: Rjust/defects4j*, Aug. 4, 2019. [Online]. Available: https://github.com/rjust/defects4j (visited on 08/18/2019).

[30] L. Li, S. Lessmann, and B. Baesens, "Evaluating Software Defect Prediction Performance: An Updated Benchmarking Study", *SSRN Electronic Journal*, 2019, ISSN: 1556-5068. DOI: 10.2139/ssrn.3312070. [Online]. Available: https://www.ssrn.com/abstract=3312070 (visited on 02/08/2019).

[31] D. J. Schad, M. Betancourt, and S. Vasishth, "Toward a principled Bayesian workflow in cognitive science", Apr. 29, 2019. arXiv: 1904.12765 [stat]. [Online]. Available: http://arxiv.org/abs/1904.12765 (visited on 08/06/2019).

[32] M. Scholz, *This repository contains the evaluation data and analysis scripts and results for my master thesis.: Sims1253/linespots-analysis*, Aug. 11, 2019. [Online]. Available: https://github.com/sims1253/linespots-analysis (visited on 08/17/2019).

[33] *Stan development repository (home page is linked below). The master branch contains the current release. The develop branch contains the latest stable development. See the Developer Process Wiki f.*. Stan, Jul. 31, 2019. [Online]. Available: https://github.com/stan-dev/stan (visited on 08/01/2019).

[34] A. Vehtari, A. Gelman, J. Gabry, Y. Yao, P.-C. Bürkner, B. Goodrich, J. Piironen, and M. Magnusson, *Loo: Efficient Leave-One-Out Cross-Validation and WAIC for Bayesian Models*, version 2.1.0, Mar. 13, 2019. [Online]. Available: https://CRAN.R-project.org/package=loo (visited on 08/31/2019).

[35] R. L. Wasserstein, A. L. Schirm, and N. A. Lazar, "Moving to a World Beyond p < 0.05", *The American Statistician*, vol. 73, pp. 1–19, sup1 Mar. 29, 2019, ISSN: 0003-1305. DOI: 10.1080/00031305.2019.1583913. [Online]. Available: https://doi.org/10.1080/00031305.2019.1583913 (visited on 05/08/2019).

[36] (). Are there any statistics that show the popularity of Git versus SVN?, [Online]. Available: https://softwareengineering.stackexchange.com/questions/136079/are-there-any-statistics-that-show-the-popularity-of-git-versus-svn (visited on 05/08/2019).

[37] (). Build software better, together, [Online]. Available: https://github.com (visited on 08/17/2019).

[38] (). GICS - Global Industry Classification Standard - MSCI, [Online]. Available: https://www.msci.com/gics (visited on 05/08/2019).

[39] (). Max Scholz / linespots-lib, [Online]. Available: https://gitlab.com/sims1253/linespots-lib (visited on 05/08/2019).

[40] (). Stack Overflow Developer Survey 2018, [Online]. Available: https://insights.stackoverflow.com/survey/2018/?utm_source=so-owned&utm_medium=social&utm_campaign=dev-survey-2018&utm_content=social-share (visited on 05/08/2019).

[41] (). Stan, [Online]. Available: //mc-stan.org/ (visited on 08/31/2019).

[42] (). Stan Discourse, [Online]. Available: https://discourse.mc-stan.org/ (visited on 08/20/2019).

[43] (). Standards, [Online]. Available: https://xkcd.com/927/ (visited on 08/18/2019).

# A

# Dataset

## A.1 Past Work

Rahman *et al.* [12]

- Apache Lucene: https://github.com/apache/lucene-solr, 31000 commits, Java
- httpd: https://github.com/apache/httpd, 31000 commits, C
- gimp: https://github.com/GNOME/gimp, 43000 commits, C
- nautilus: https://github.com/GNOME/nautilus, 21000 commits, C
- evolution: https://github.com/GNOME/evolution, 44000 commits, C

D'Ambros *et al.* [8]

- Eclipse JDT Core: https://github.com/eclipse/eclipse.jdt.core, 24000 commits, Java
- Eclipse PDE UI: https://github.com/eclipse/eclipse.pde.ui, 13000 commits, Java
- Equinox Framework: https://github.com/eclipse/rt.equinox.framework, 4700 commits, Java
- Mylyn: https://github.com/eclipse/mylyn, 1200 commits, Java, Shell
- Apache Lucene: https://github.com/apache/lucene-solr, 31000 commits, Java

Tóth *et al.* [19] and later Li *et al.* [30]

- Android-Universal-Image-Loader: https://github.com/nostra13/Android-Universal-Image-Loader, 1000 commits, Java
- BroadleafCommerce: https://github.com/BroadleafCommerce/BroadleafCommerce, 16000 commits, Java
- MapDB: https://github.com/jankotek/mapdb, 2100 commits, Java
- antlr4: https://github.com/antlr/antlr4, 7000 commits, Java, Python, C#, C++, Swift

- ceylon-ide-eclipse: https://github.com/eclipse/ceylon-ide-eclipse/tree/_old/master, 8000 commits, Java, **old/master branch!**
- Elasticsearch, https://github.com/elastic/elasticsearch, 45700 commits, Java
- hazelcast: https://github.com/hazelcast/hazelcast, 28000 commits, Java
- junit: https://github.com/junit-team/junit5, 5600 commits, Java
- mcMMO: https://github.com/mcMMO-Dev/mcMMO, 5400 commits, Java
- mct: https://github.com/nasa/mct, 1000 commits, Java
- neo4j: https://github.com/neo4j/neo4j, 60000 commits, Java
- netty: https://github.com/search?q=netty, 9000 commits, Java
- orientdb: https://github.com/orientechnologies/orientdb, 18000 commits, Java
- oryx: https://github.com/OryxProject/oryx, 1100 commits, Java
- titan: https://github.com/thinkaurelius/titan, 4400 commits, Java Projects were chosen based on being big, maintained java projects.

Zou *et al.* [26]

- Apache Commons Math: https://github.com/apache/commons-math, 6400 commits, Java
- Apache Commons Lang: https://github.com/apache/commons-lang, 5400 commits, Java
- Joda-Time: https://github.com/JodaOrg/joda-time, 2000 commits, Java
- JFreeChart: https://github.com/jfree/jfreechart, 3600 commits, Java
- Google Closure compuler: https://github.com/google/closure-compiler/, 14400 commits, Java

## A.2 Drawing Process

These projects were drawn from the top 1000 GitHub projects, ranked by stars. Only projects with over 3000 commits containing source code were chosen. The numbers starting each entry are the random numbers that pointed to the projects in the top 1000 list at the time of building the dataset.

- ☐ 14: https://github.com/justjavac/free-programming-books-zh_CN, not code
- ☒ 83: https://github.com/GoogleChrome/lighthouse, 3300 commits, JS
- ☐ 178: https://github.com/square/leakcanary, 800 commits, Kotlin
- ☐ 311: https://github.com/balena-io/etcher, 2100 commits, JS
- ☐ 591: https://github.com/eczarny/spectacle, 700 Commits, Objective-C

☒ 626: https://github.com/prisma/prisma, 10600 commits, Scala

☒ 647: https://github.com/ncw/rclone, 3100 commits, Go

☐ 699: https://github.com/nswbmw/N-blog, 172 commits, JS

☐ 731: https://github.com/Wox-launcher/Wox, 1616 commits, C#

☒ 765: https://github.com/typeorm/typeorm, 3800 commits, TypeScript

☐ 437: https://github.com/byoungd/English-level-up-tips-for-Chinese, not code

☐ 11: https://github.com/airbnb/javascript, 1700 commits, JS

☐ 857: https://github.com/facebook/yoga, 1700 commits, C++, JS, Java, C#

☐ 327: https://github.com/jiahaog/nativefier, 780 commits, JS

☒ 460: https://github.com/guzzle/guzzle, 3200 commits, PHP

☐ 703: https://github.com/shuzheng/zheng, 1200 commits, Java

☐ 510: https://github.com/feathericons/feather, 600 commits, JS

☐ 51: https://github.com/hakimel/reveal.js, 2341 commits, JS

☐ 451: https://github.com/Solido/awesome-flutter, 1000 commits, Dart

☐ 606: https://github.com/JacksonTian/fks, 200 commits, JS

☐ 597: https://github.com/enyo/dropzone, 800 commits, JS

☒ 67: https://github.com/moment/moment, 3700 commits, JS

☒ 463: https://github.com/vim/vim, 9700 commits, Vim Script, C

☐ 458: https://github.com/angular/material, 4700 commits, JS

☒ 353: https://github.com/Automattic/mongoose, 10600 commits, JS

☒ 560: https://github.com/FFmpeg/FFmpeg, 93600 commits, C

☐ 535: https://github.com/jdg/MBProgressHUD, 600 commits, Objective C

☐ 887: https://github.com/winterbe/java8-tutorial, 150 commits, Java

☐ 29: https://github.com/axios/axios, 800 commits, JS

☒ 163: https://github.com/discourse/discourse, 32600 commits, Ruby

Table A.1 shows all the projects from GitHub, past research and the authors choice together with an index. This was then used to draw the final projects for this thesis. Table A.2 shows the drawing process with the projects that were drawn and in case of a rejection, the reason why we did not include the project.

**Table A.1:** Project List with Indices

| Index | Project Name | Index | Project Name |
|---|---|---|---|
| 1 | Android-Universal-Image-Loader | 39 | moment |
| 2 | angular | 40 | Money Manager Ex |
| 3 | antlr4 | 41 | MongoDB |
| 4 | Apache Lucene | 42 | Mongoose |
| 5 | Atom | 43 | mpv media player |
| 6 | bootstrap | 44 | Mylyn |
| 7 | BroadleafCommerce | 45 | MySQL |
| 8 | ceylon-ide-eclipse | 46 | nautilus |
| 9 | Discourse | 47 | neo4j |
| 10 | Django | 48 | netty |
| 11 | Eclipse Che | 49 | OpenBSD |
| 12 | Eclipse JDT Core | 50 | opencart |
| 13 | Eclipse PDE UI | 51 | OpenJDK |
| 14 | Elasticsearch | 52 | orientdb |
| 15 | Emacs | 53 | oryx |
| 16 | Equinox Framework | 54 | PostgreSQL |
| 17 | evolution | 55 | prestashop |
| 18 | FFmpeg | 56 | Prisma |
| 19 | Flask | 57 | Python |
| 20 | FreeBSD | 58 | Pytorch |
| 21 | gimp | 59 | Rails |
| 22 | GnuCash | 60 | Rclone |
| 23 | Go | 61 | react |
| 24 | guzzle | 62 | Redis |
| 25 | hazelcast | 63 | Ruby |
| 26 | httpd | 64 | Rust |
| 27 | junit | 65 | scikit-learn |
| 28 | Keras | 66 | Swift |
| 29 | KMyMoney | 67 | Tensorflow |
| 30 | kodi | 68 | Theano |
| 31 | lighthouse | 69 | titan |
| 32 | Linux | 70 | typeorm |
| 33 | Magento2 | 71 | Typescript |
| 34 | MapDB | 72 | vim |
| 35 | MariaDB | 73 | vlc |
| 36 | mcMMO | 74 | VSCode |
| 37 | mct | 75 | Vue |
| 38 | media player classic home cinema | 76 | WooCommerce |
| | | 77 | zencart |

**Table A.2:** Project Drawing

| Index | Project | Usable |
|---|---|---|
| 9 | Discourse | yes |
| 40 | Money Manager Ex | No fix indicator |
| 23 | Go | No fix indicator |
| 55 | PrestaShop | yes |
| 65 | scikit-learn | yes |
| 7 | BroadleafCommerce | yes |
| 48 | netty | No fix indicator |
| 8 | Ceylon IDE | yes |
| 76 | WooCommerce | yes |
| 60 | rclone | No fix indicator |
| 18 | ffmpeg | yes |
| 59 | Rails | yes |
| 4 | Apache Lucene-Solr | yes |
| 35 | MariaDB | yes |
| 45 | MySQL | yes |
| 72 | vim | No fix indicator |
| 38 | media player classic home cinema | yes |
| 27 | junit | yes |
| 16 | Equinox Framework | yes |
| 6 | bootstrap | yes |
| 57 | Python | yes |
| manual | Commons-math | yes |
| manual | Closure compiler | yes |
| manual | Jfreechart | yes |
| manual | coala | yes |
| manual | evolution | yes |
| manual | httpd | yes |

# B

# Projpred Results

This chapter holds the projpred results for the last 5 tested cases. The first case is shown in figure B.1.

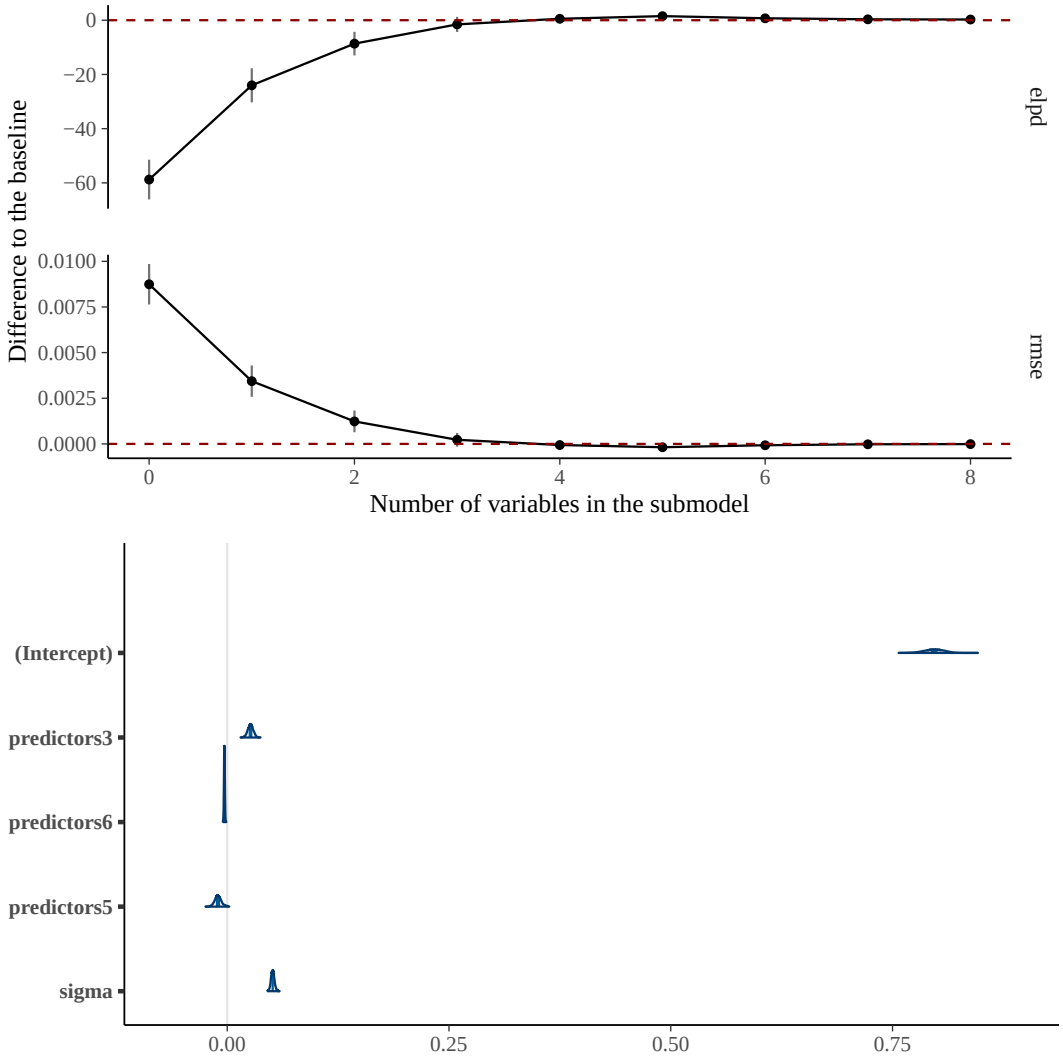



**Figure B.1:** varsel_plot and mcmc_areas plot for AUCECEXAM prediction in RQ1 and RQ2

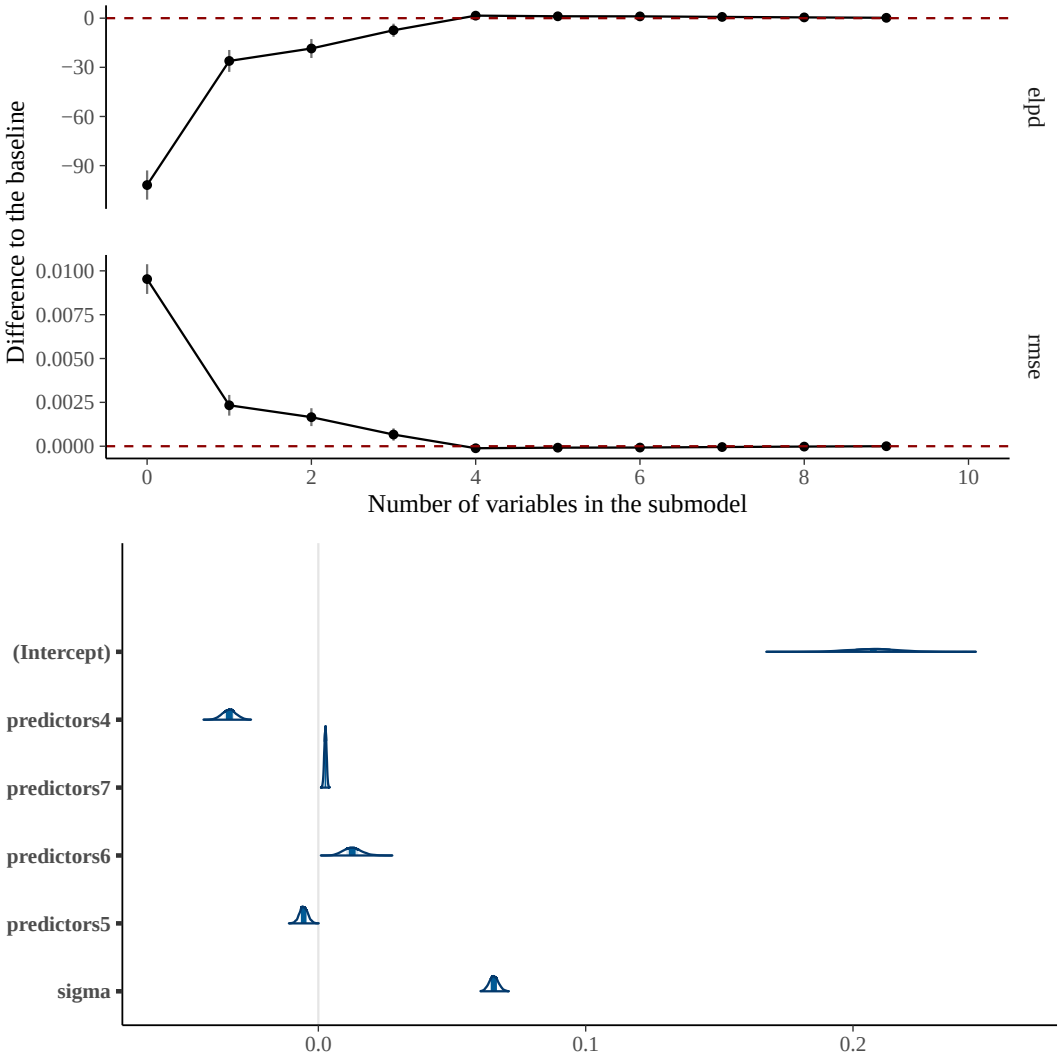



**Figure B.2:** varsel_plot and mcmc_areas plot for EXAM prediction in RQ4

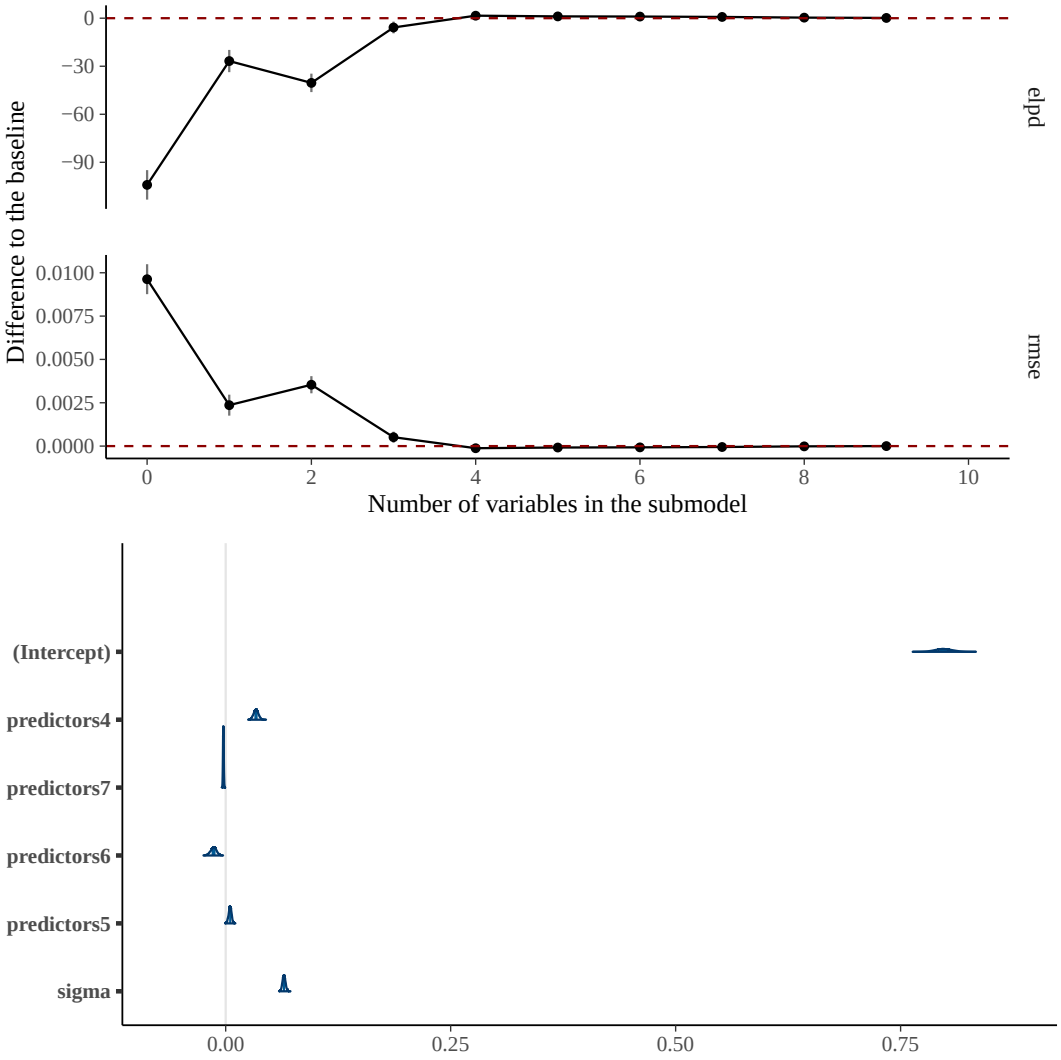



**Figure B.3:** varsel_plot and mcmc_areas plot for EXAM prediction in RQ4

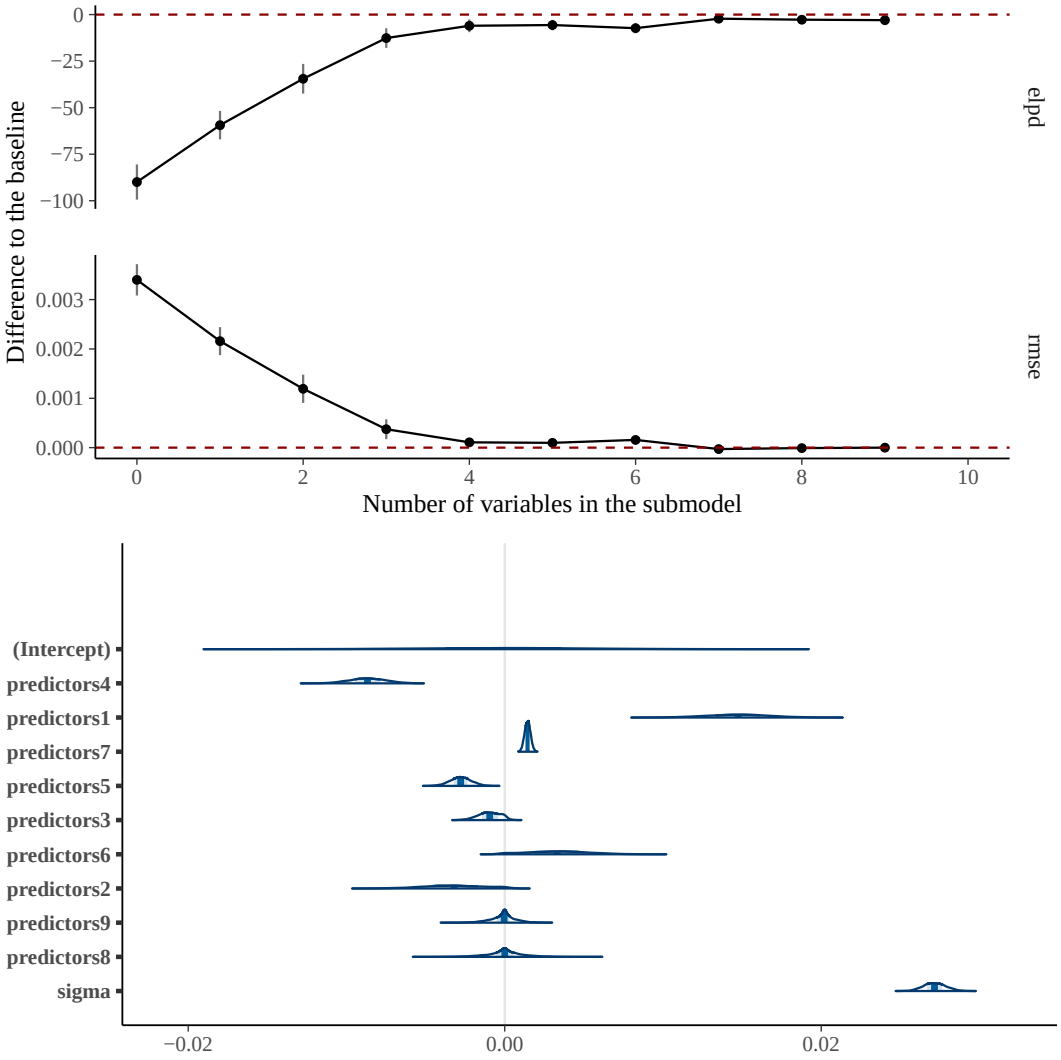



**Figure B.4:** varsel_plot and mcmc_areas plot for EXAM25 prediction in RQ4

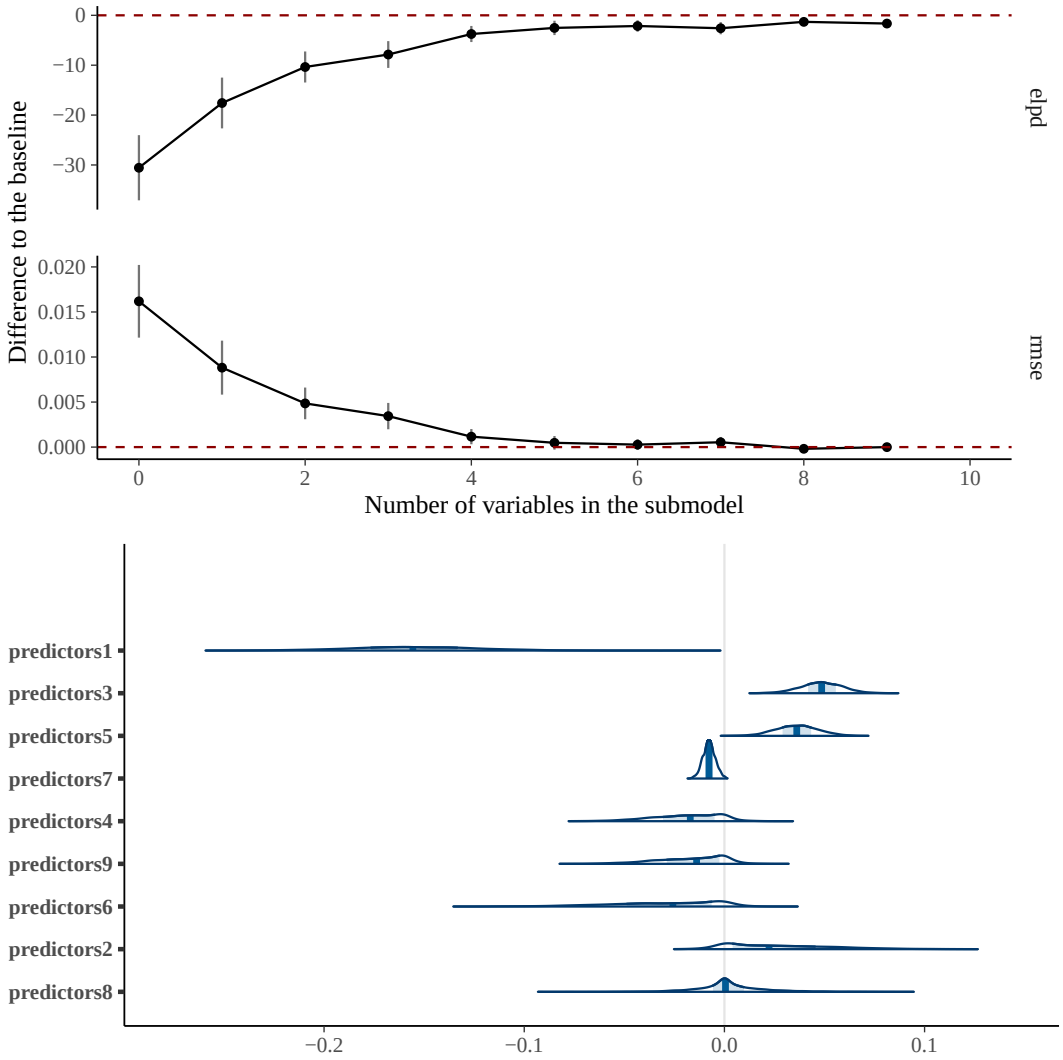



**Figure B.5:** varsel_plot and mcmc_areas plot for EInspect25EXAM prediction in RQ4

# C

# Models

## C.1 Research Question 1

### C.1.1 EXAM

### C.1.2 AUCECEXAM

$$
\begin{aligned}
\text{EXAM}_i &\sim \text{Beta}(\mu_i, \phi) \\
\text{logit}(\mu_i) &= \alpha + \beta_w W_i + \beta_l L_i + \gamma_{Project[i]} \\
\alpha &\sim \text{Normal}(0, 0.5) \\
\beta_w, \beta_l &\sim \text{Normal}(0, 0.5) \\
\gamma_j &\sim \text{Normal}(\bar{\gamma}, \sigma) \\
\bar{\gamma}, \sigma &\sim \text{HalfCauchy}(0, 0.1) \\
\phi &\sim \text{Gamma}(0.1, 0.1)
\end{aligned}
$$

**Model C.1:** Simple model for EXAM using Weighting

$$
\begin{aligned}
\text{AUCECEXAM}_i &\sim \text{Beta}(\mu_i, \phi) \\
\text{logit}(\mu_i) &= \alpha + \beta_w W_i + \beta_l L_i + \gamma_{Project[i]} \\
\alpha &\sim \text{Normal}(0, 0.5) \\
\beta_w, \beta_l &\sim \text{Normal}(0, 0.5) \\
\gamma_j &\sim \text{Normal}(\bar{\gamma}, \sigma) \\
\bar{\gamma}, \sigma &\sim \text{HalfCauchy}(0, 0.1) \\
\phi &\sim \text{Gamma}(0.1, 0.1)
\end{aligned}
$$

**Model C.2:** Simple model for AUCECEXAM using Weighting

## C.2 Research Question 2

### C.2.1 EXAM

### C.2.2 AUCECEXAM

$$
\begin{aligned}
\text{EXAM}_i &\sim \text{Beta}(\mu_i, \phi) \\
\text{logit}(\mu_i) &= \alpha + \beta_t T_i + \beta_l L_i + \gamma_{Project[i]} \\
\alpha &\sim \text{Normal}(0, 0.5) \\
\beta_t, \beta_l &\sim \text{Normal}(0, 0.5) \\
\gamma_j &\sim \text{Normal}(\bar{\gamma}, \sigma) \\
\bar{\gamma}, \sigma &\sim \text{HalfCauchy}(0, 0.1) \\
\phi &\sim \text{Gamma}(0.1, 0.1)
\end{aligned}
$$

**Model C.3:** Simple model for EXAM using Algorithm

$$
\begin{aligned}
\text{AUCECEXAM}_i &\sim \text{Beta}(\mu_i, \phi) \\
\text{logit}(\mu_i) &= \alpha + \beta_t T_i + \beta_l L_i + \gamma_{Project[i]} \\
\alpha &\sim \text{Normal}(0, 0.5) \\
\beta_t, \beta_l &\sim \text{Normal}(0, 0.5) \\
\gamma_j &\sim \text{Normal}(\bar{\gamma}, \sigma) \\
\bar{\gamma}, \sigma &\sim \text{HalfCauchy}(0, 0.1) \\
\phi &\sim \text{Gamma}(0.1, 0.1)
\end{aligned}
$$

**Model C.4:** Simple model for EXAM using Algorithm

## C.3 Research Question 4

### C.3.1 AUCECEXAM

$$
\begin{aligned}
\text{EXAM}_i &\sim \text{Beta}(\mu_i, \phi) \\
\text{logit}(\mu_i) &= \alpha + \beta_a A_i + \beta_l L_i + \gamma_{Project[i]} \\
\alpha &\sim \text{Normal}(0, 0.5) \\
\beta_a, \beta_l &\sim \text{Normal}(0, 0.5) \\
\gamma_j &\sim \text{Normal}(\bar{\gamma}, \sigma) \\
\bar{\gamma}, \sigma &\sim \text{HalfCauchy}(0, 0.1) \\
\phi &\sim \text{Gamma}(0.1, 0.1)
\end{aligned}
$$

**Model C.5:** Simple model for EXAM using Algorithm

$$
\begin{aligned}
\text{AUCECEXAM}_i &\sim \text{Beta}(\mu_i, \phi) \\
\text{logit}(\mu_i) &= \alpha + \beta_a A_i + \beta_l L_i + \gamma_{Project[i]} \\
\alpha &\sim \text{Normal}(0, 0.5) \\
\beta_a, \beta_l &\sim \text{Normal}(0, 0.5) \\
\gamma_j &\sim \text{Normal}(\bar{\gamma}, \sigma) \\
\bar{\gamma}, \sigma &\sim \text{HalfCauchy}(0, 0.1) \\
\phi &\sim \text{Gamma}(0.1, 0.1)
\end{aligned}
$$

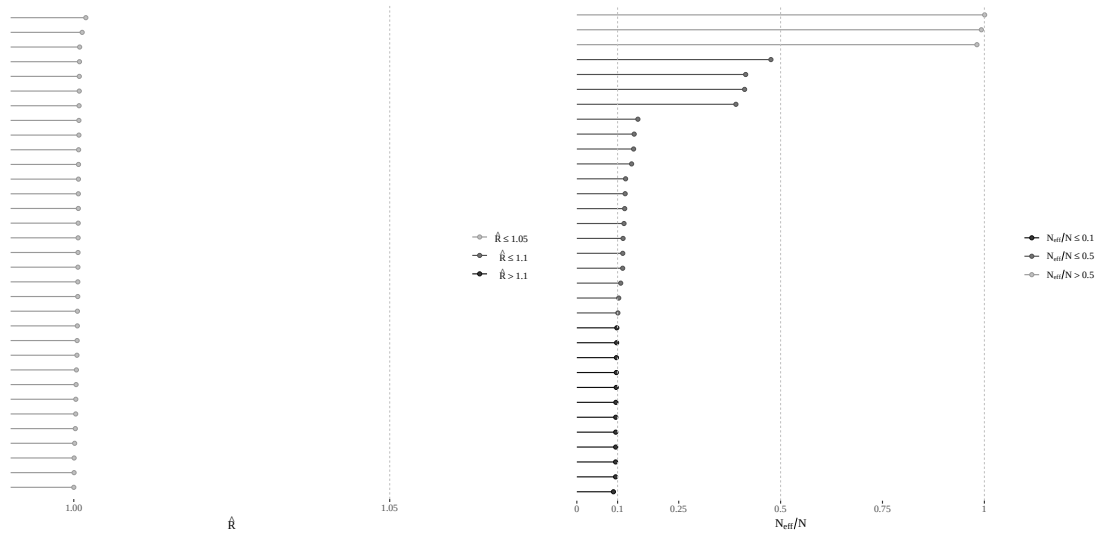
**Model C.6:** Simple model for EXAM using Algorithm

# D

# Model Diagnostics

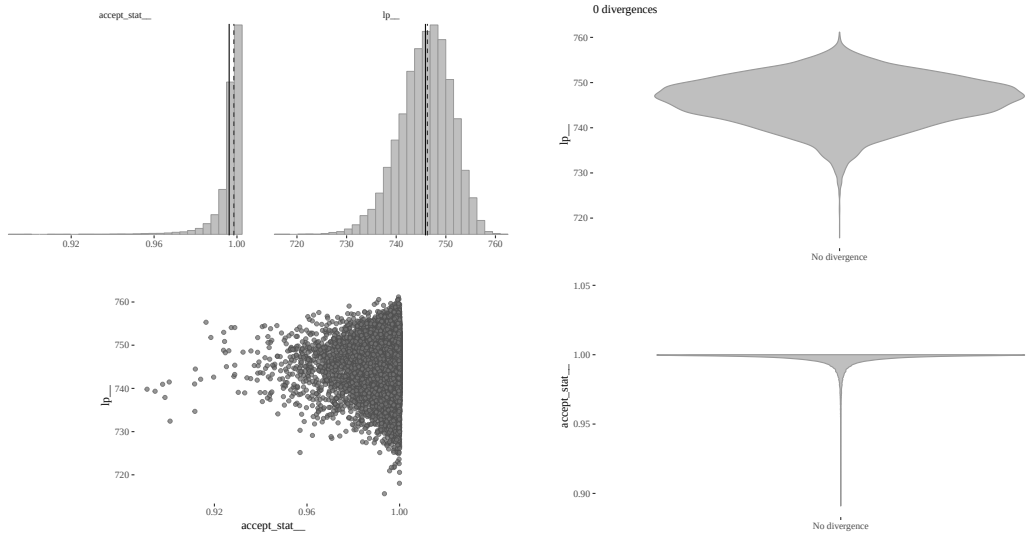
## D.1 RQ1 Diagnostics

### D.1.1 EXAM Model 1



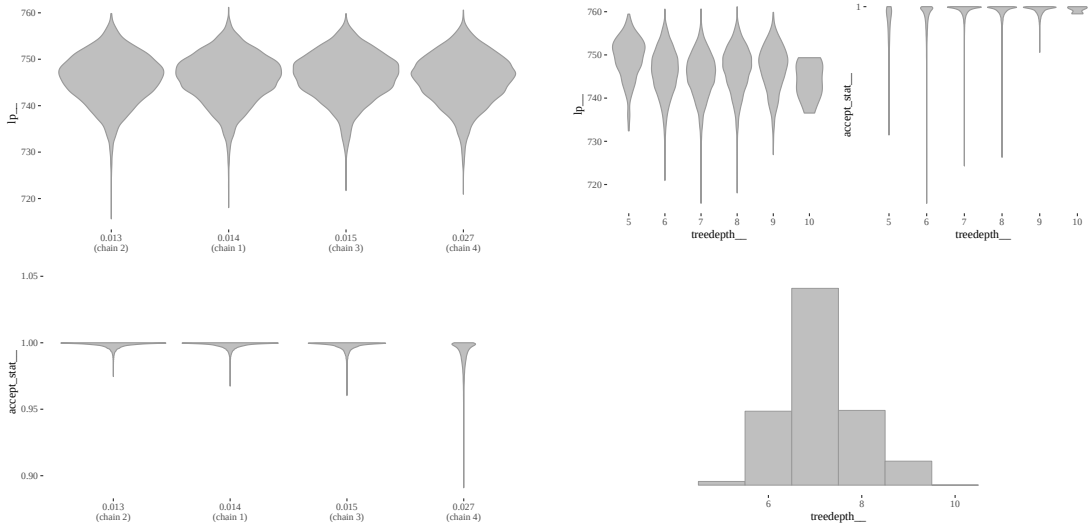

**(a)** Rhat **(b)** n_eff

**Figure D.1:** Rhat and n_eff for Model C.1



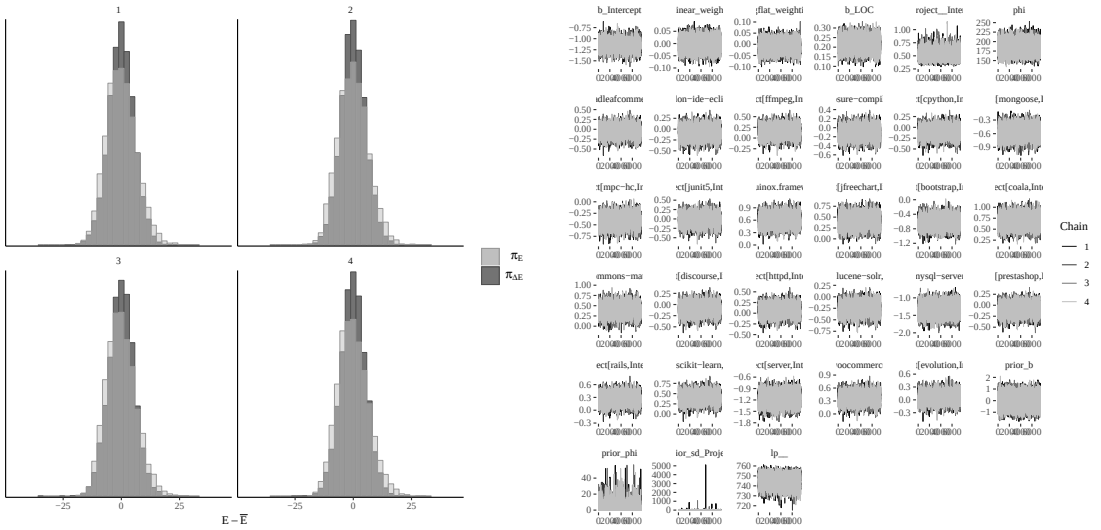

**(a)** Acceptance **(b)** Divergent transitions

**Figure D.2:** Acceptance and divergent transitions for model C.1




**(a)** Step size **(b)** Tree depth

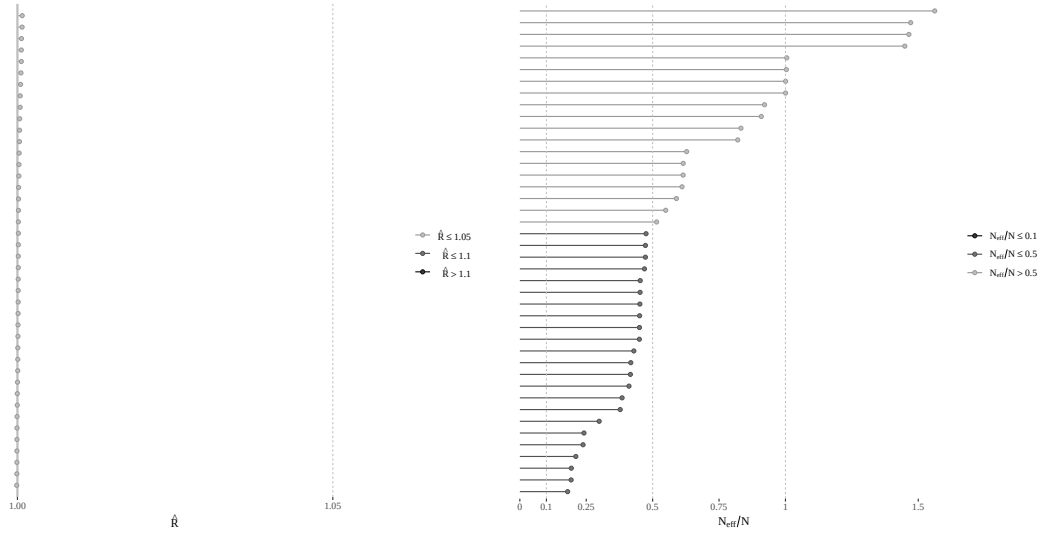
**Figure D.3:** Step size and tree depth for model C.1




**(a)** Energy **(b)** Trace plots

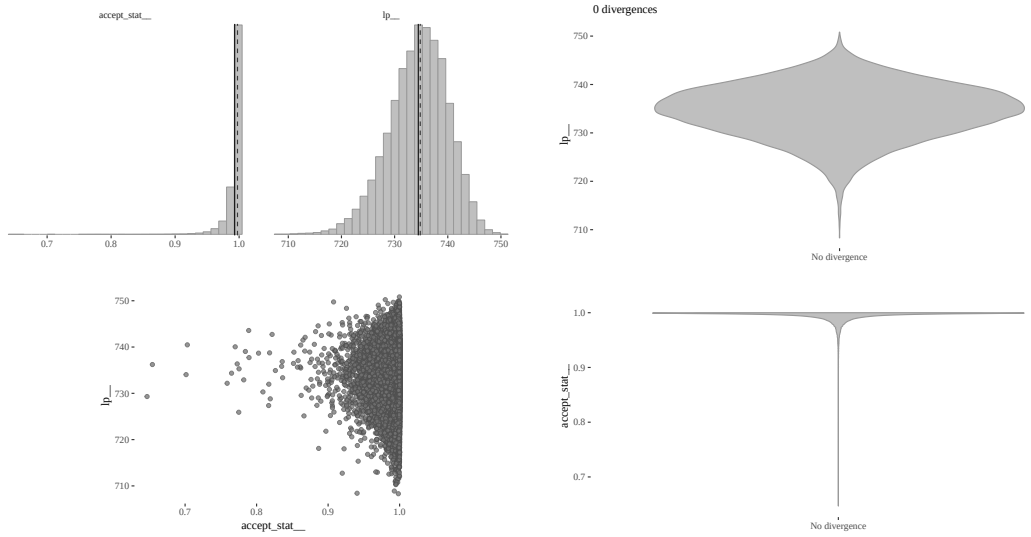
**Figure D.4:** Energy and trace plots for model C.1

### D.1.2 EXAM Model 2



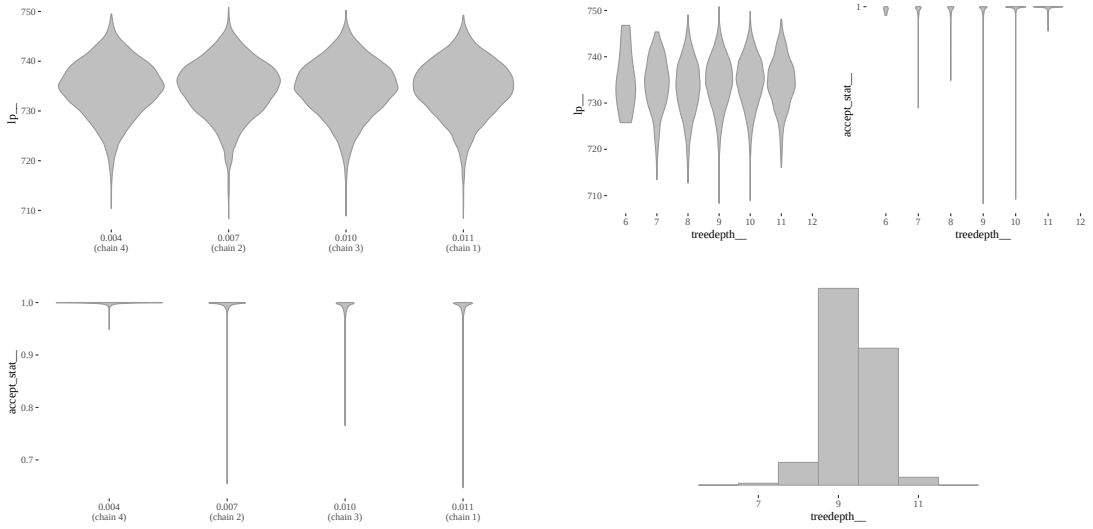

**(a)** Rhat **(b)** n_eff

**Figure D.5:** Rhat and n_eff for model 4.1



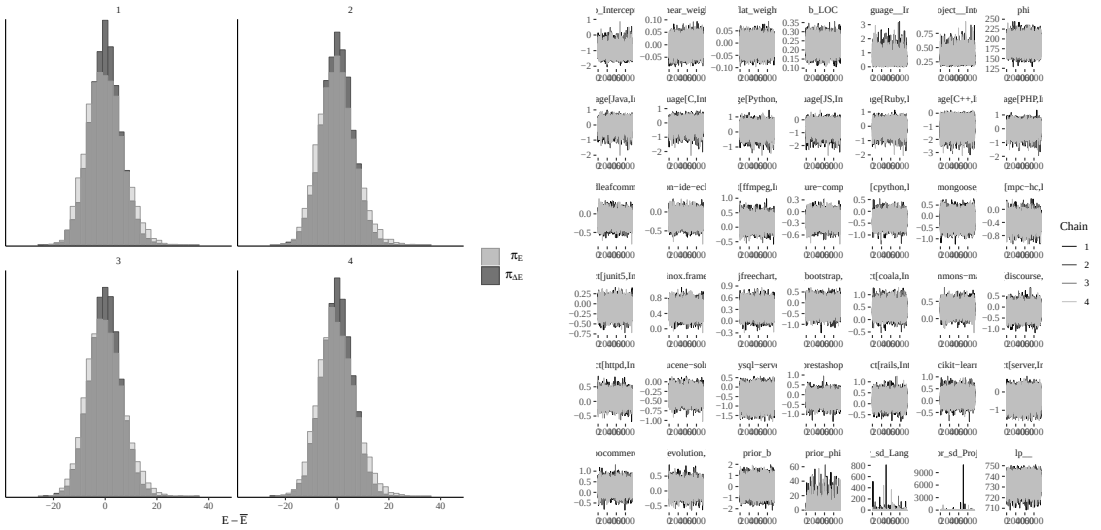

**(a)** Acceptance **(b)** Divergent Transitions

**Figure D.6:** Acceptance and divergent transitions for model 4.1




**(a)** Step size **(b)** Tree depth

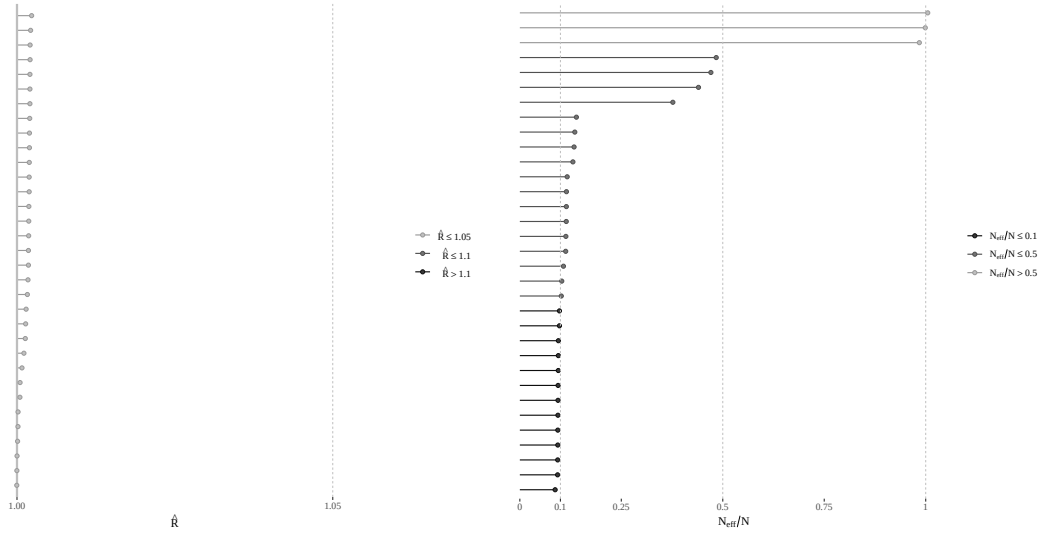
**Figure D.7:** Step size and tree depth for model 4.1




**(a)** Energy **(b)** Trace plots

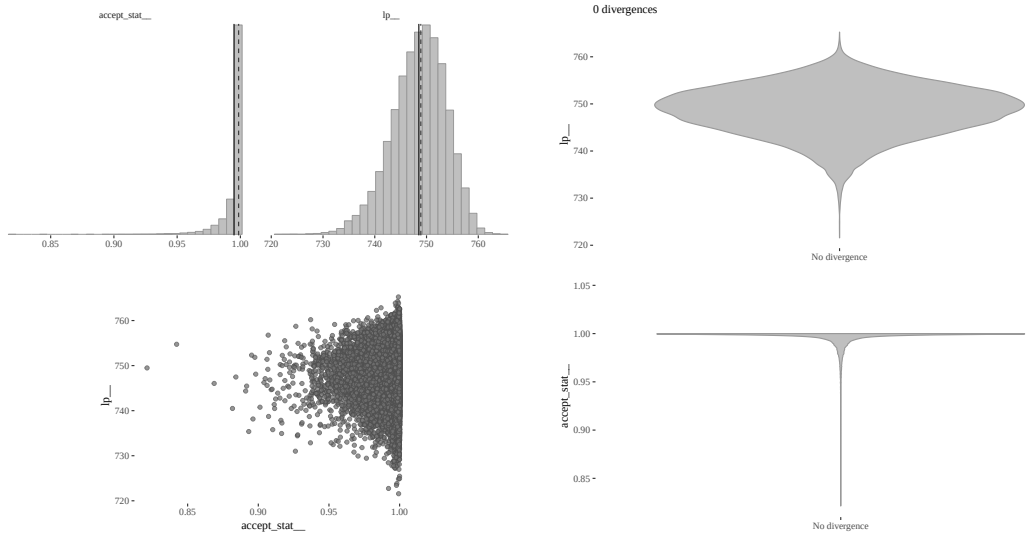
**Figure D.8:** Energy and trace plots for model 4.1

### D.1.3 AUCECEXAM Model 1



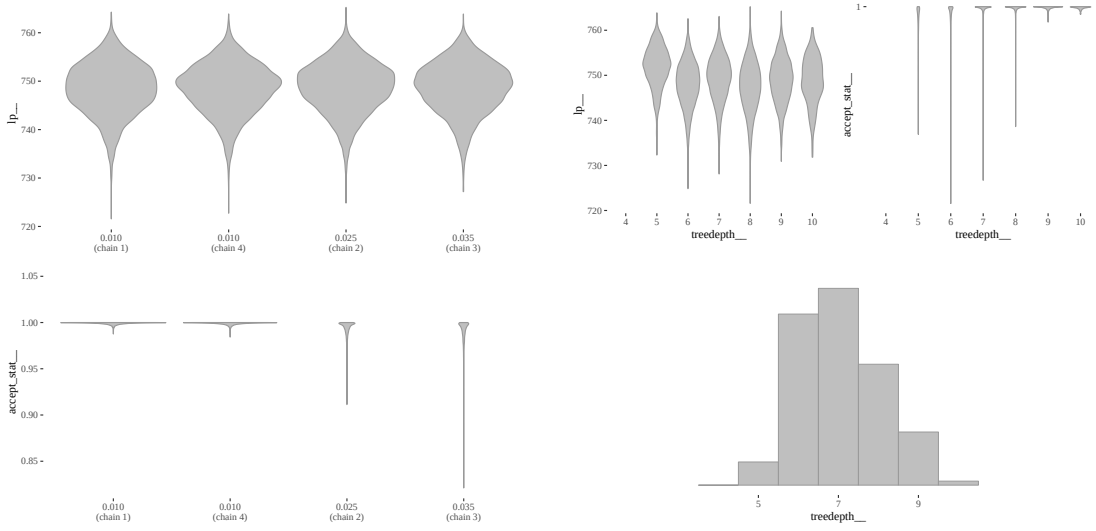

**(a)** Rhat **(b)** n_eff

**Figure D.9:** Rhat and n_eff for Model C.2



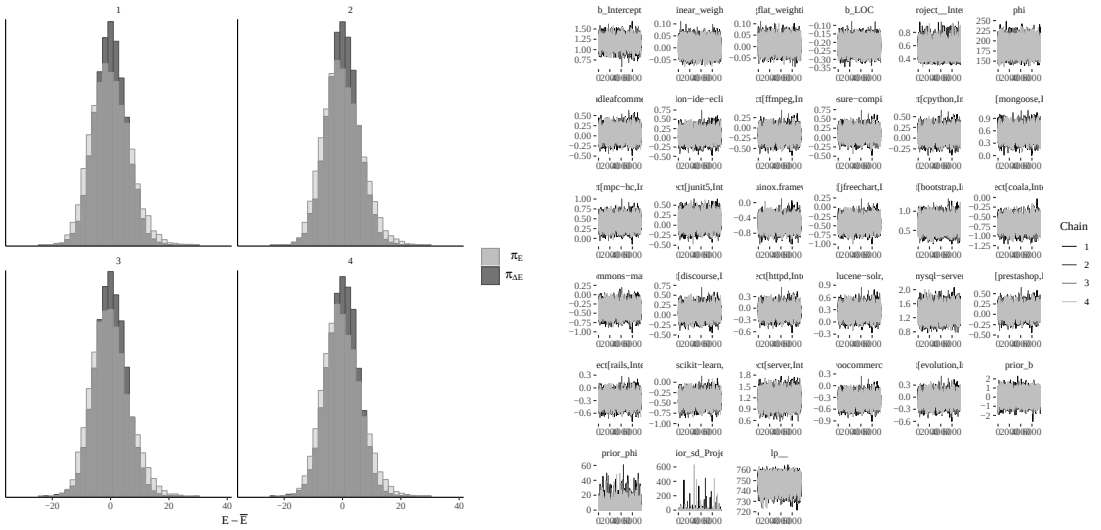

**(a)** Acceptance **(b)** Divergent Transitions

**Figure D.10:** Acceptance and divergent transitions for Model C.2




**(a)** Step size **(b)** Tree depth

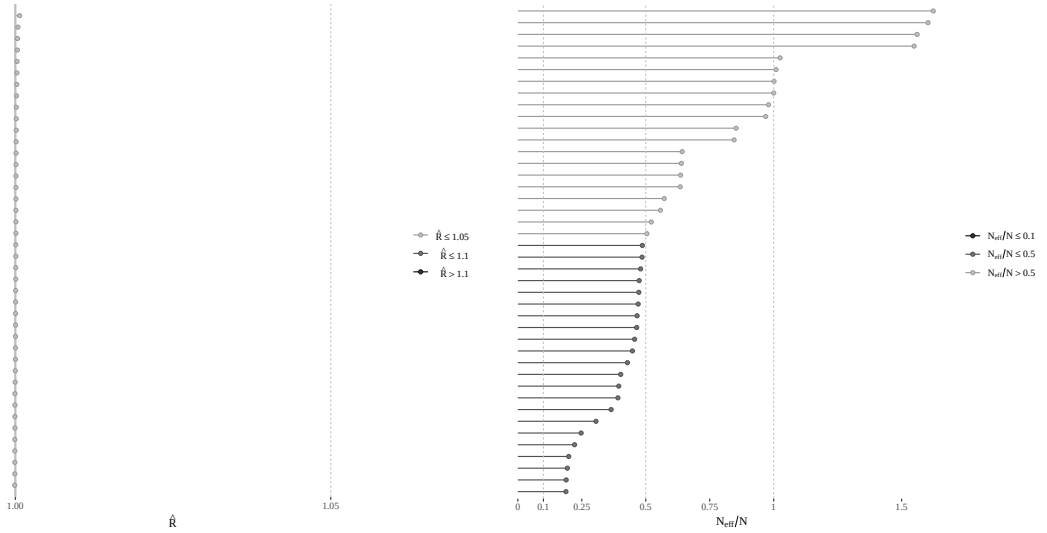
**Figure D.11:** Step size and tree depth for Model C.2




**(a)** Energy **(b)** Trace plots

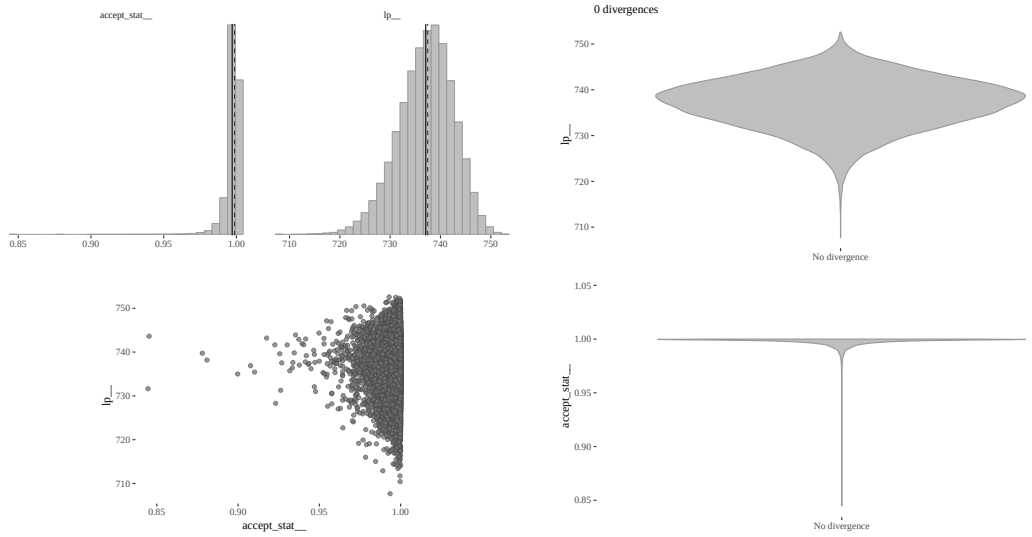
**Figure D.12:** Energy and trace plots for Model C.2

### D.1.4 AUCECEXAM Model 2



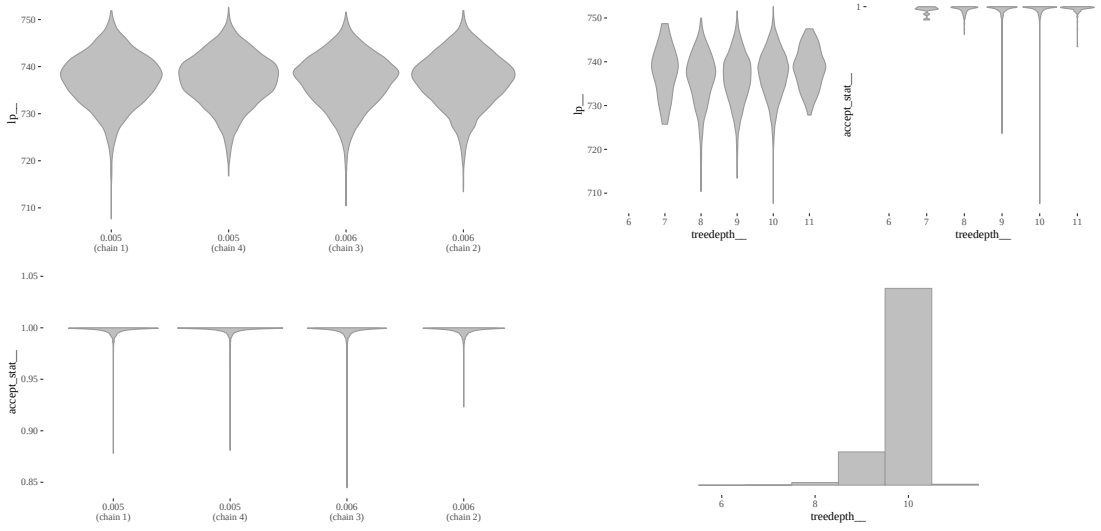

**(a)** Rhat **(b)** n_eff

**Figure D.13:** Rhat and n_eff for Model 4.2



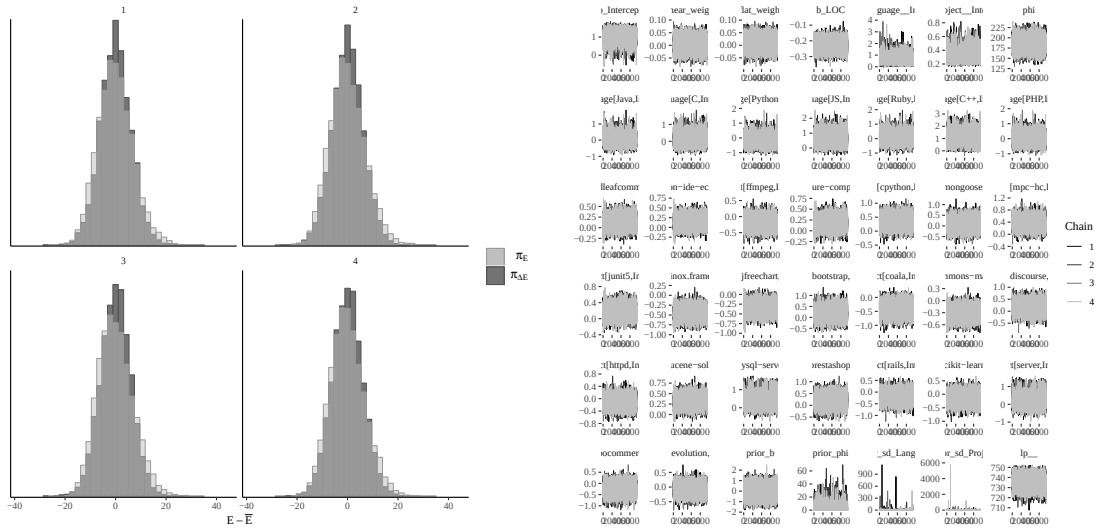

**(a)** Acceptance **(b)** Divergent Transitions

**Figure D.14:** Acceptance and divergent transitions for Model 4.2




**(a)** Step size **(b)** Tree depth

**Figure D.15:** Step size and tree depth for Model 4.2

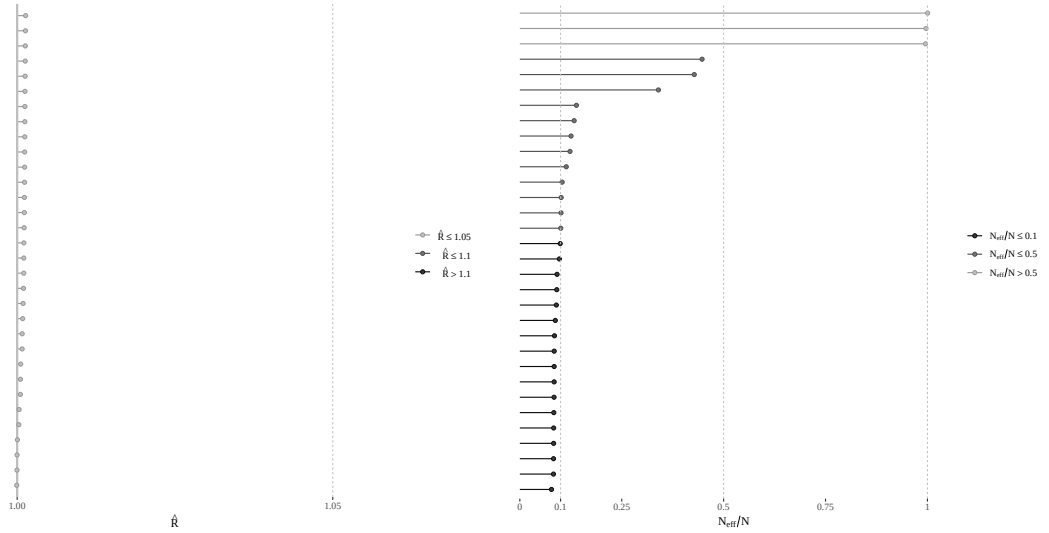



**(a)** Energy **(b)** Trace plots

**Figure D.16:** Energy and traec plots for Model 4.2

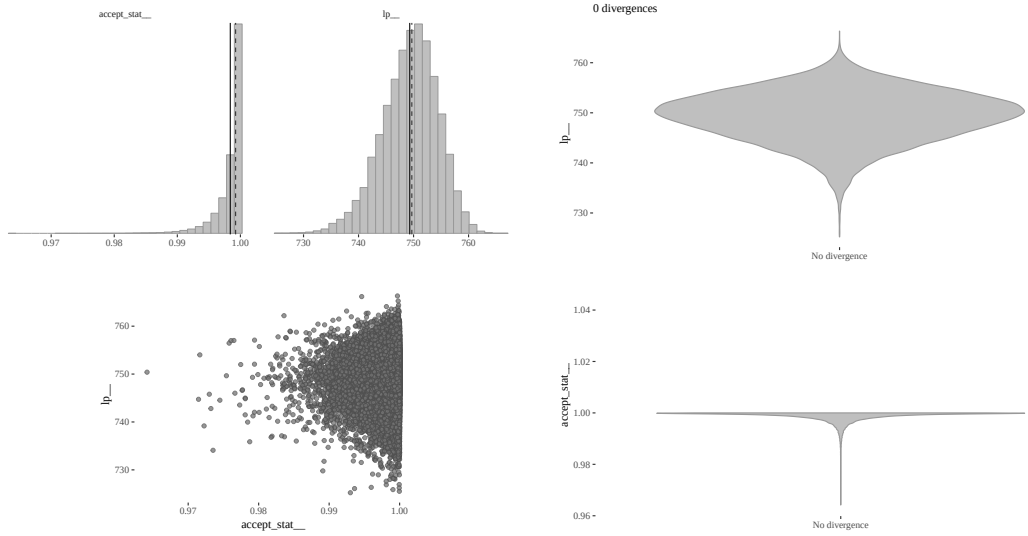
## D.2 RQ2 Diagnostics

### D.2.1 EXAM Model 1



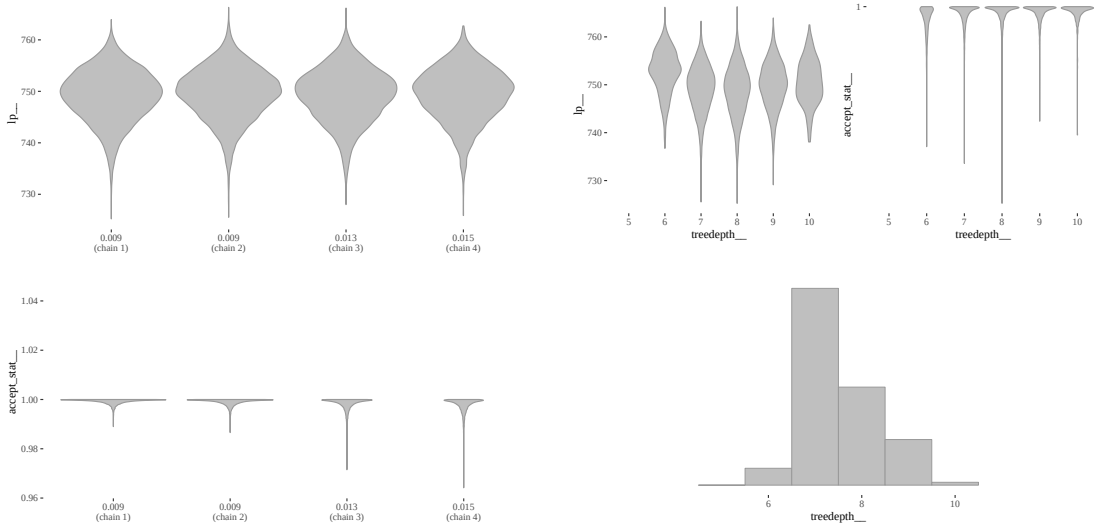

**(a)** Rhat **(b)** n_eff

**Figure D.17:** Rhat and n_eff for Model C.3



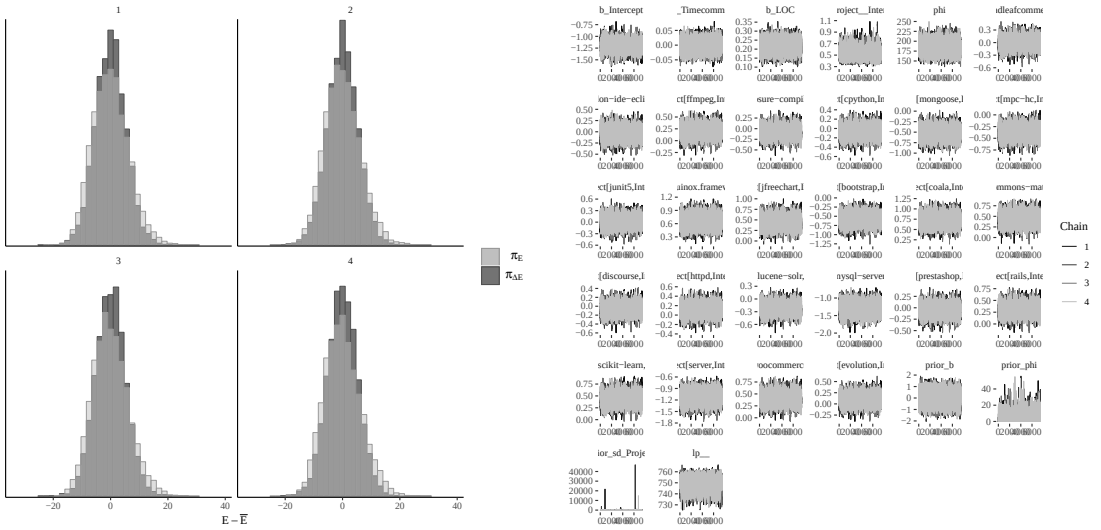

**(a)** Acceptance **(b)** Divergent Transitions

**Figure D.18:** Acceptance and divergent transitions for model C.3




**(a)** Step size **(b)** Tree depth

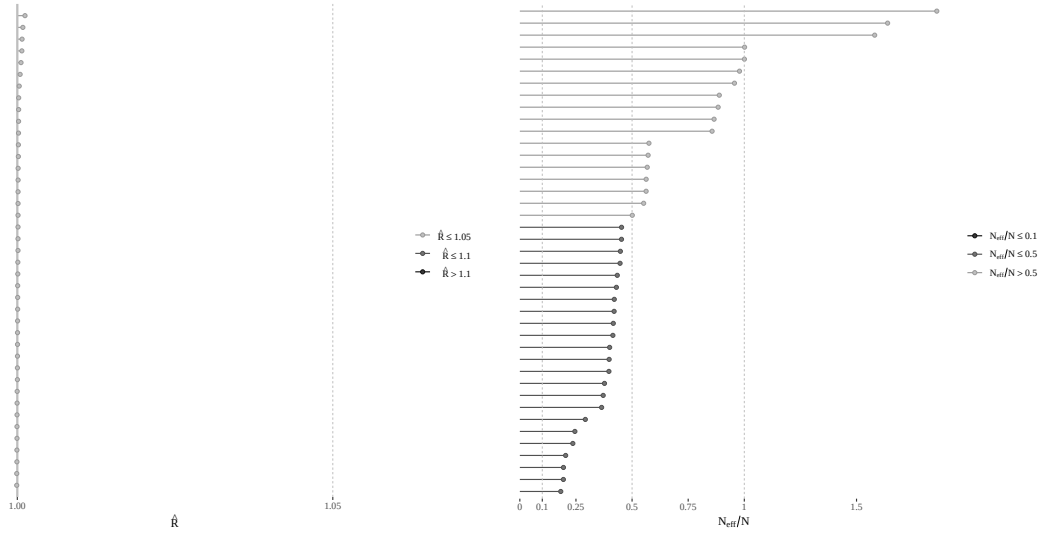
**Figure D.19:** Step size and tree depth for model C.3




**(a)** Energy **(b)** Trace plots

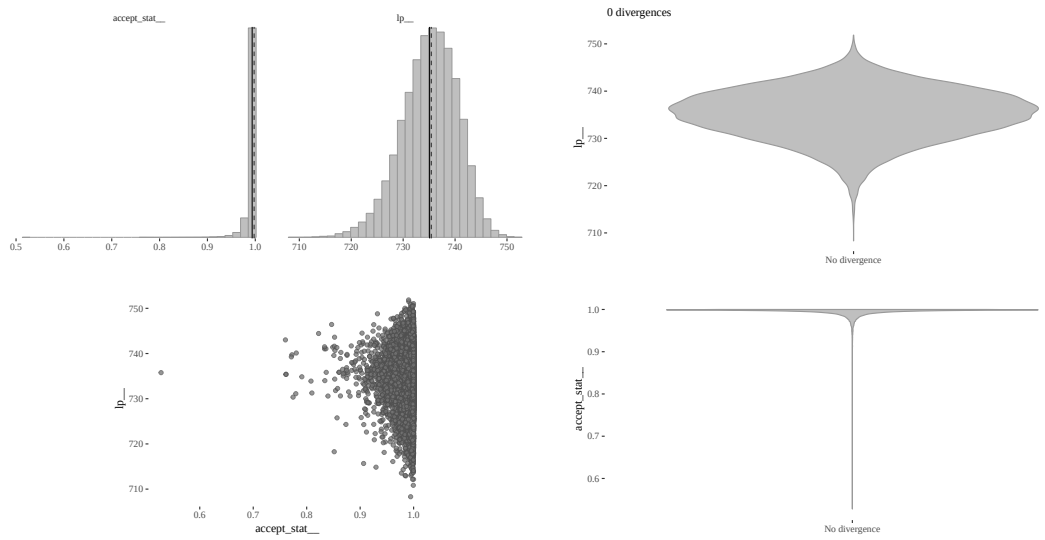
**Figure D.20:** Energy and trace plots for model C.3

### D.2.2 EXAM Model 2



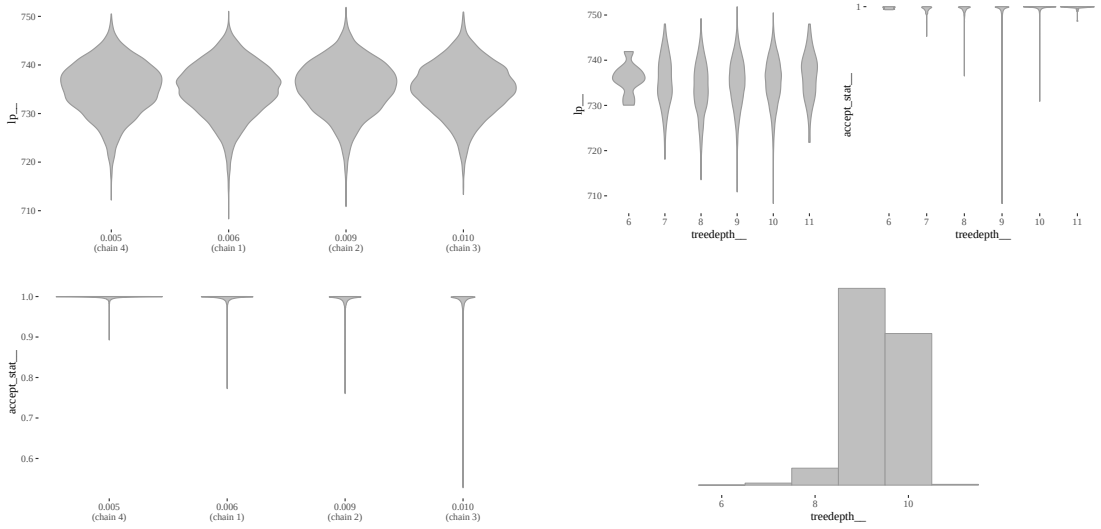

**(a)** Rhat **(b)** n_eff

**Figure D.21:** Rhat and n_eff for model 4.3



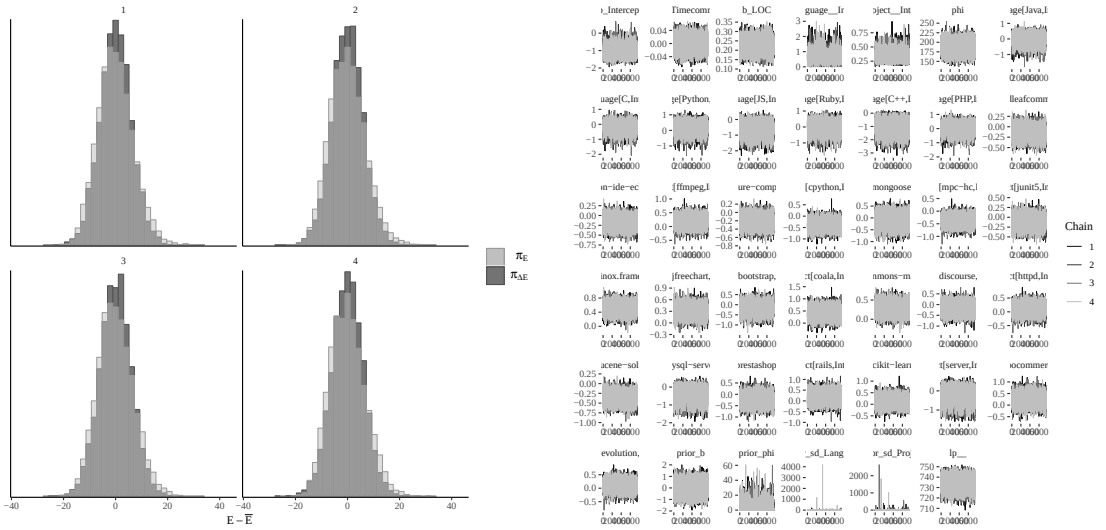

**(a)** Acceptance **(b)** Divergent Transitions

**Figure D.22:** Acceptance and divergent transitions for model 4.3




**(a)** Step size **(b)** Tree depth

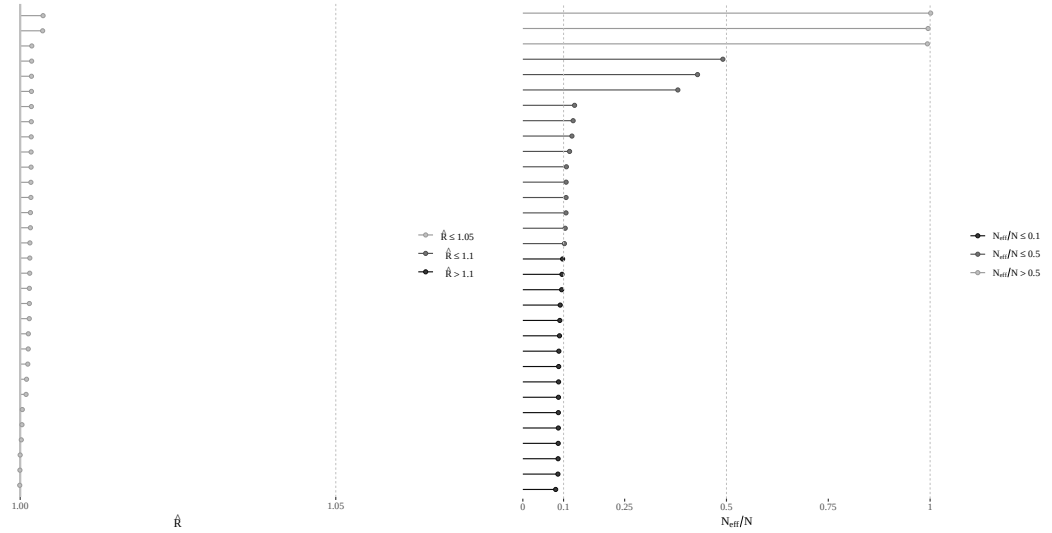
**Figure D.23:** Step size and tree depth for model 4.3




**(a)** Energy **(b)** Trace plots

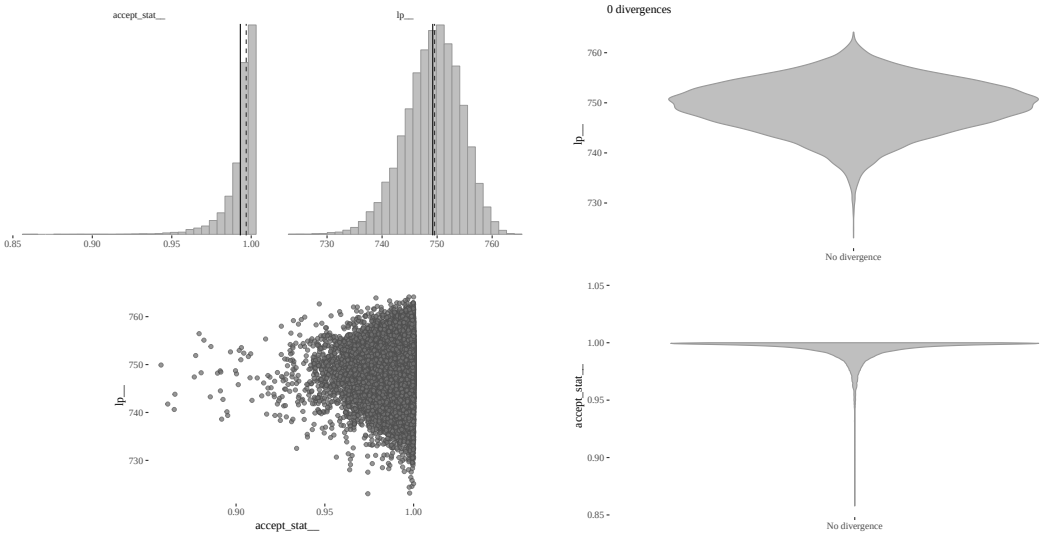
**Figure D.24:** Energy and trace plots for model 4.3

### D.2.3 AUCECEXAM Model 1



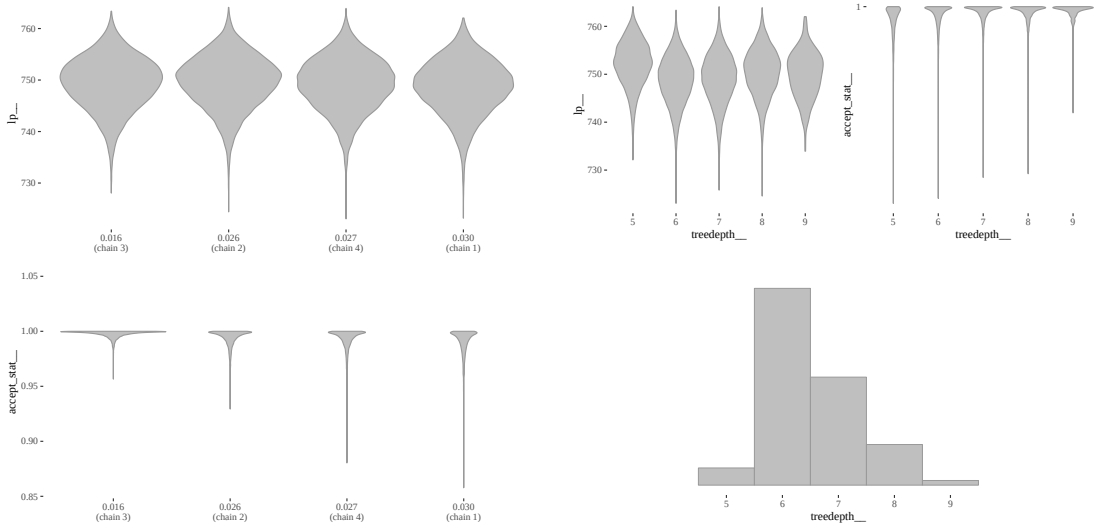

**(a)** Rhat **(b)** n_eff

**Figure D.25:** Rhat and n_eff for model C.4



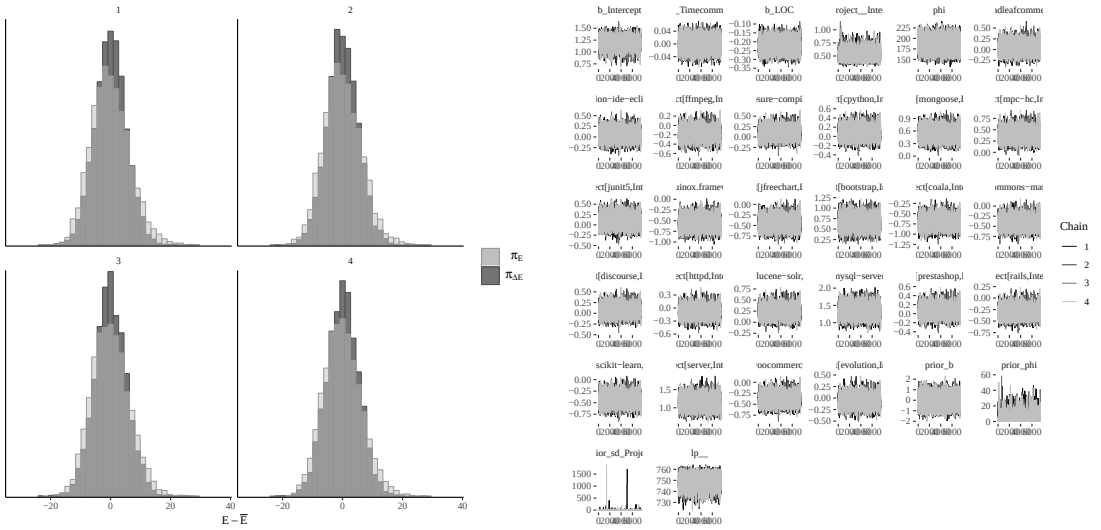

**(a)** Acceptance **(b)** Divergent Transitions

**Figure D.26:** Acceptance and divergent transitions for model C.4




**(a)** Step size **(b)** Tree depth

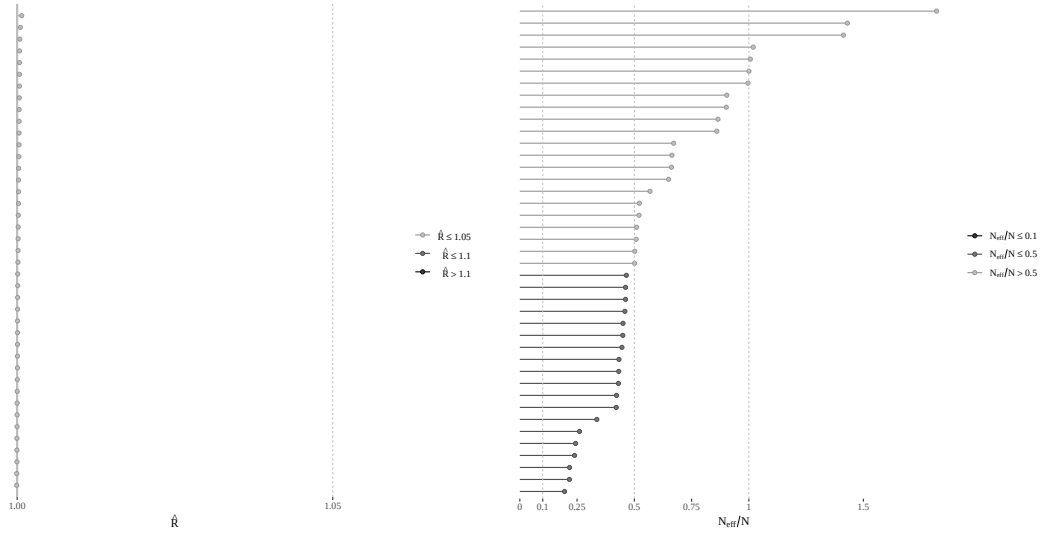
**Figure D.27:** Step size and tree depth for model C.4




**(a)** Energy **(b)** Trace plots

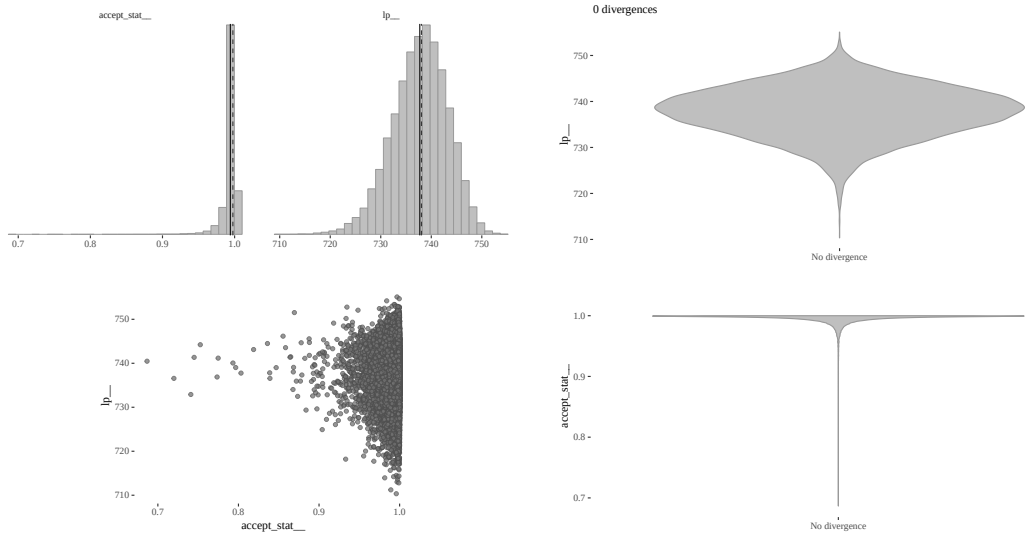
**Figure D.28:** Energy and trace plots for model C.4

### D.2.4 AUCECEXAM Model 2



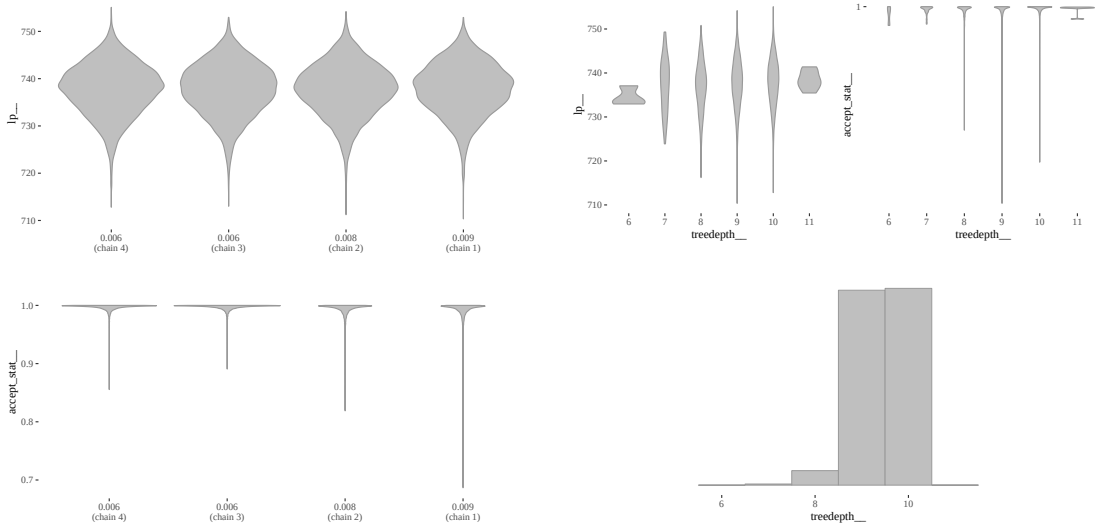

**(a)** Rhat **(b)** n_eff

**Figure D.29:** Rhat and n_eff for model 4.4



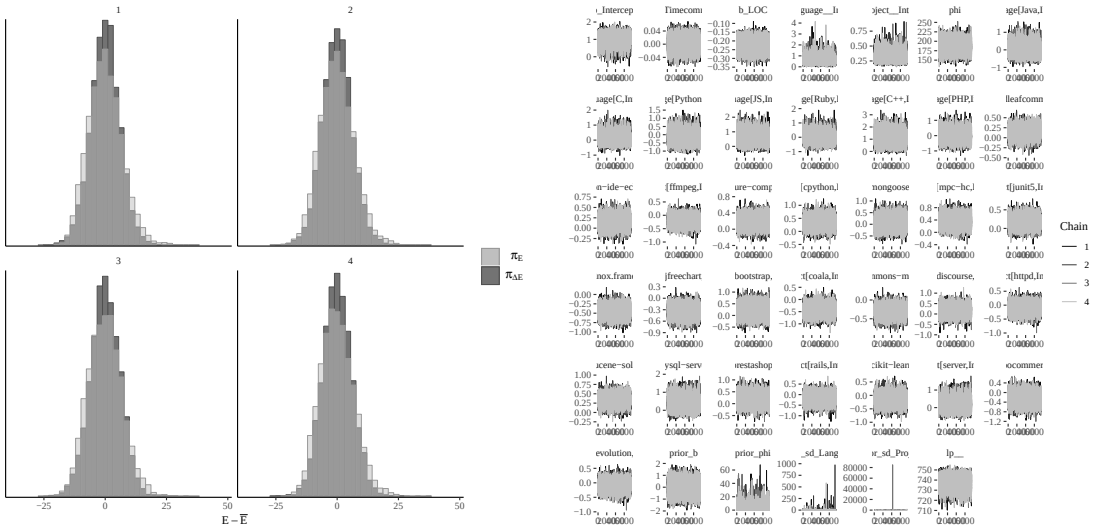

**(a)** Acceptance **(b)** Divergent Transitions

**Figure D.30:** Acceptance and divergent transitions for model 4.4




**(a)** Step size **(b)** Tree depth

**Figure D.31:** Step size and tree depth for model 4.4

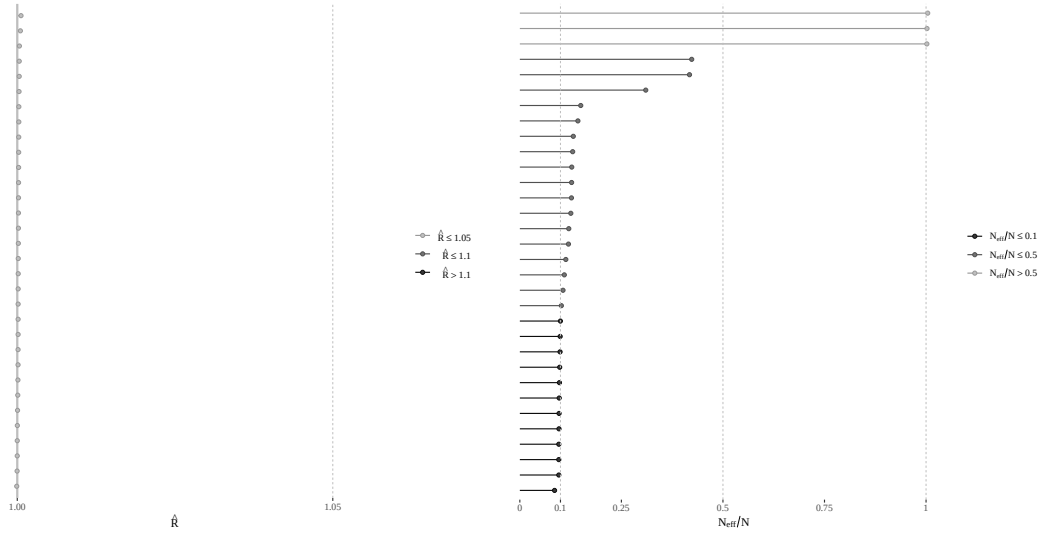



**(a)** Energy **(b)** Trace plots

**Figure D.32:** Energy and trace plots for model 4.4

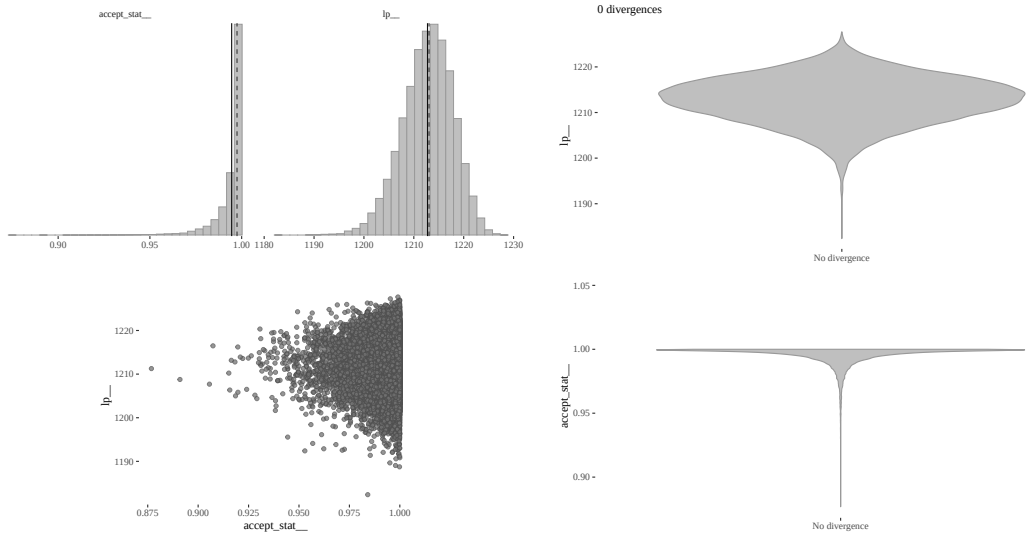
## D.3 RQ4 Diagnostics

### D.3.1 EXAM Model 1



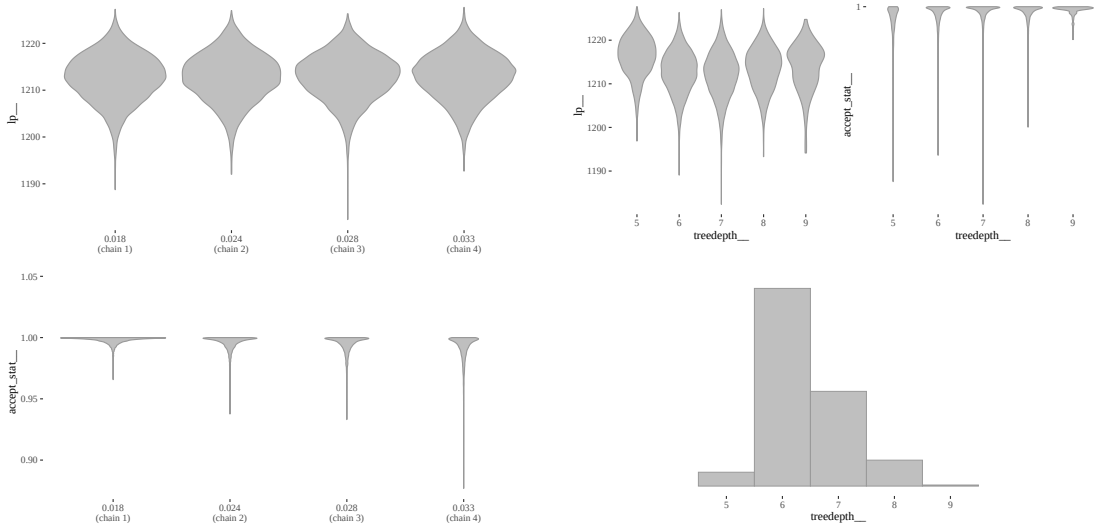

**(a)** Rhat **(b)** n_eff

**Figure D.33:** Rhat and n_eff for Model C.5



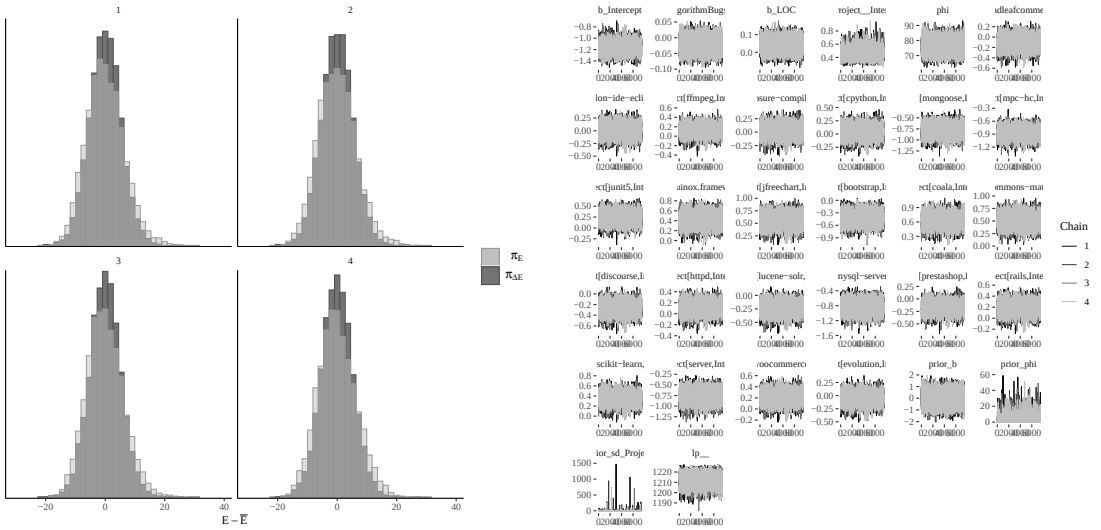

**(a)** Acceptance **(b)** Divergent Transitions

**Figure D.34:** Acceptance and divergent transitions for model C.5




**(a)** Step Size **(b)** Tree depth

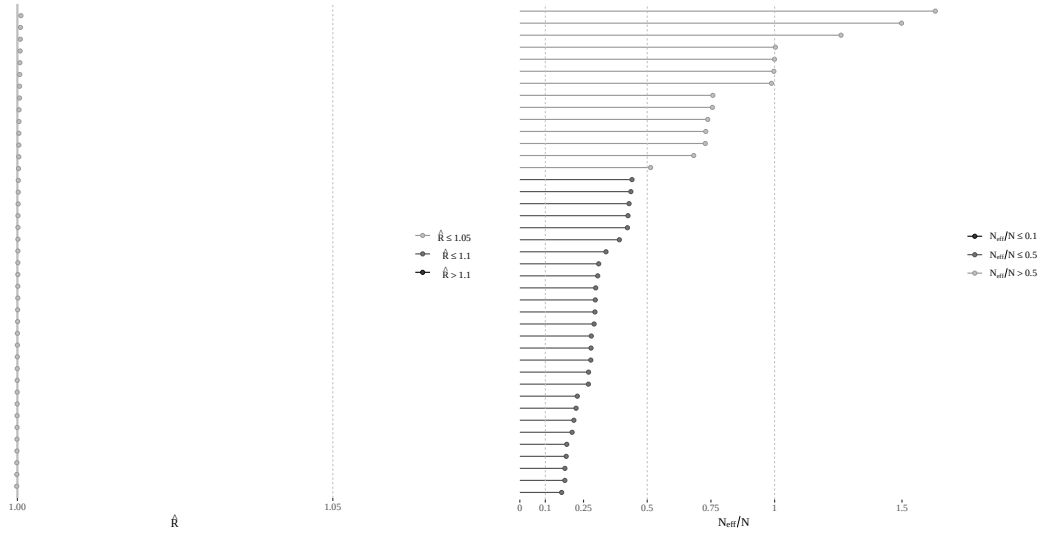
**Figure D.35:** Step size and tree depth for model C.5




**(a)** Energy **(b)** Trace plots

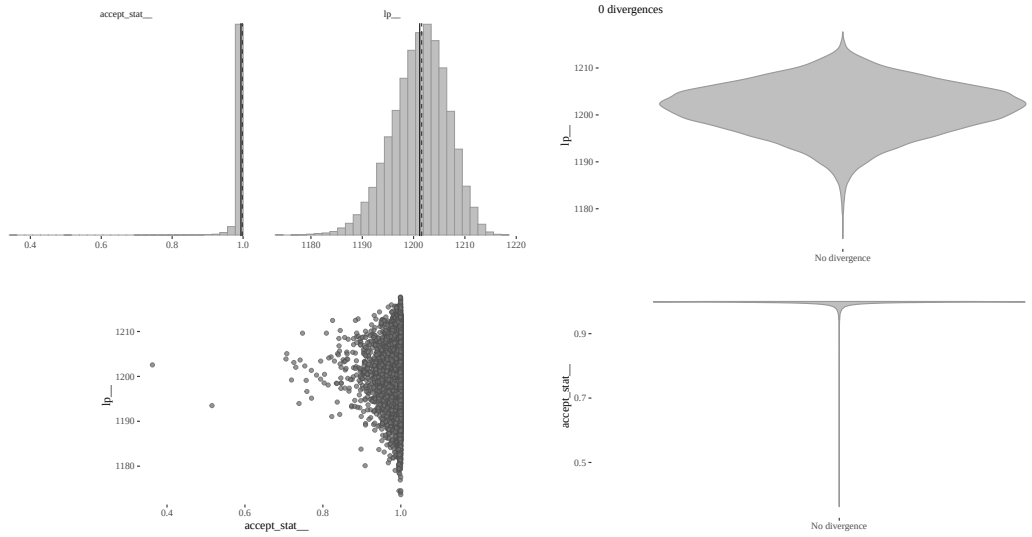
**Figure D.36:** Energy and trace plots for model C.5

### D.3.2 EXAM Model 2



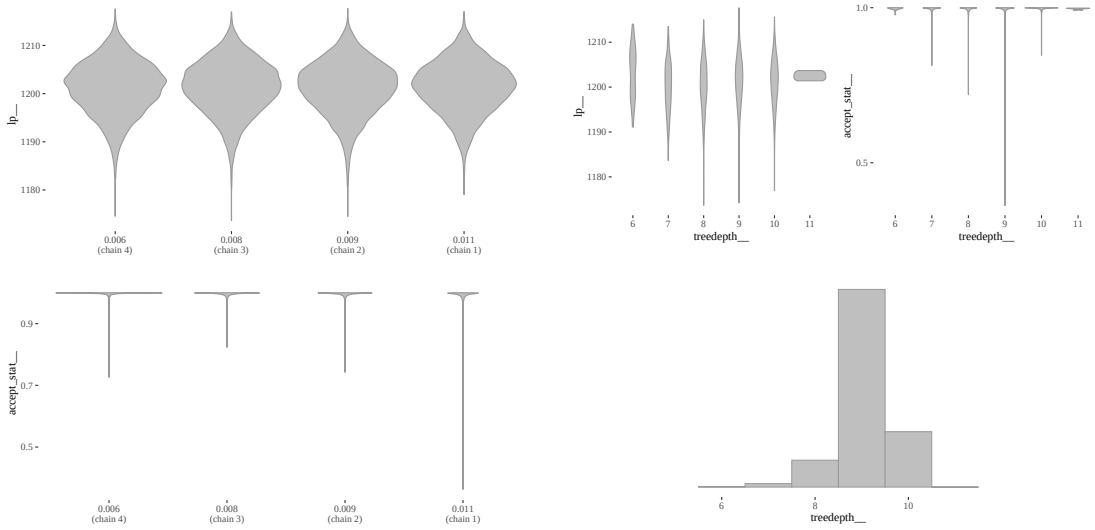

**(a)** Rhat **(b)** n_eff

**Figure D.37:** Rhat and n_eff for model 4.5



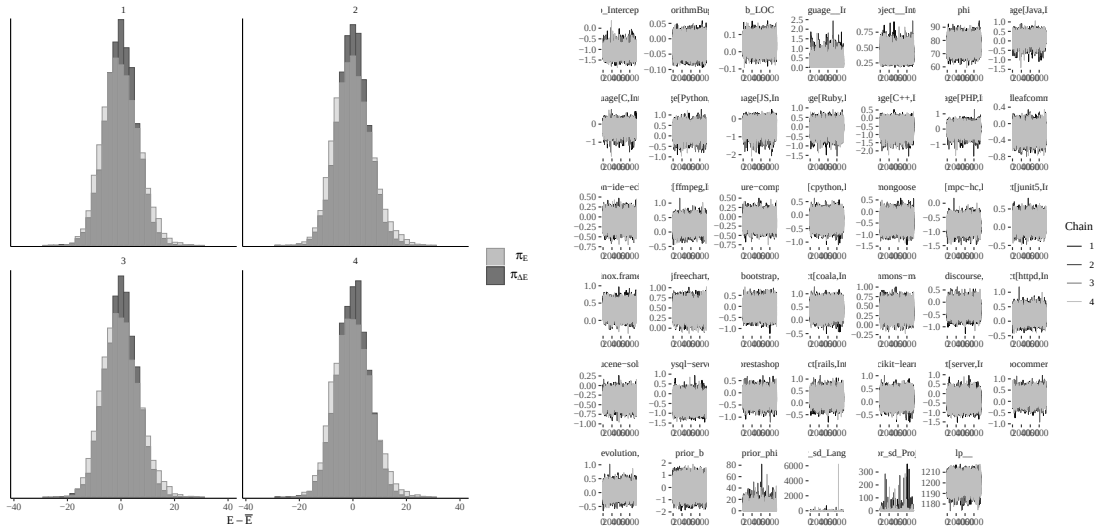

**(a)** Acceptance **(b)** Divergent Transitions

**Figure D.38:** Acceptance and divergent transitions for model 4.5




**(a)** Step Size **(b)** Tree depth

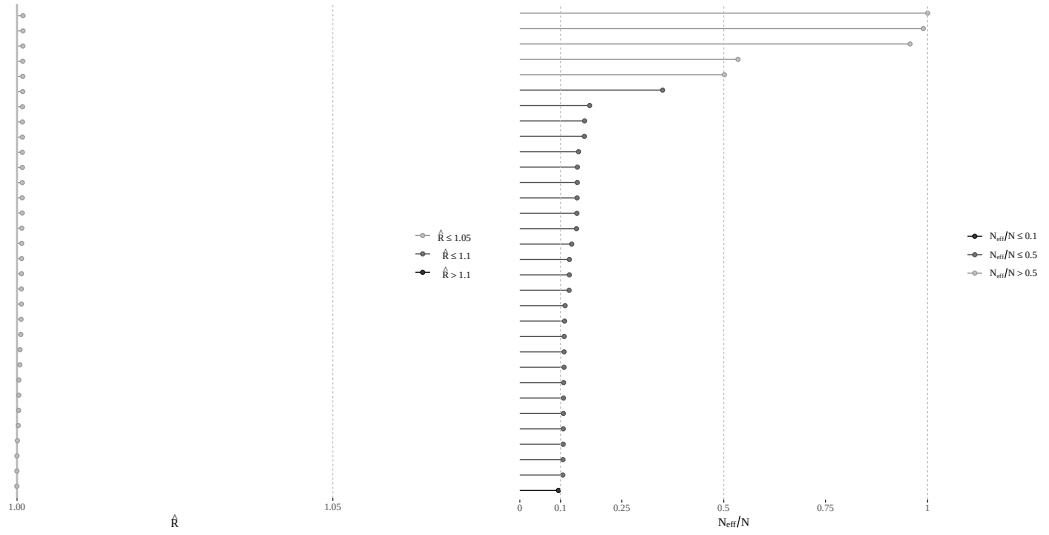
**Figure D.39:** Step size and tree depth for model 4.5




**(a)** Energy **(b)** Trace plots

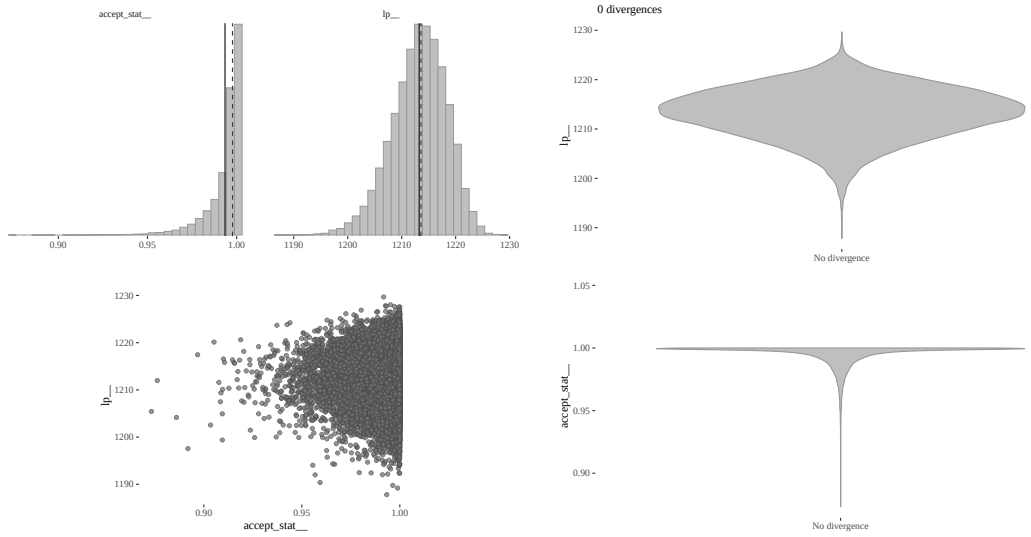
**Figure D.40:** Energy and trace plots for model 4.5

### D.3.3 AUCECEXAM Model 1



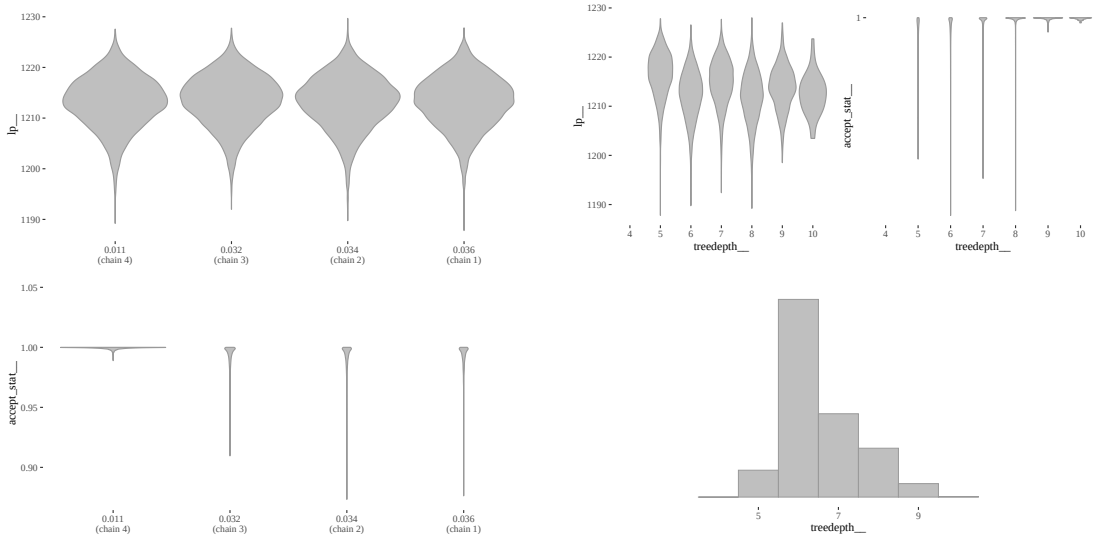

**(a)** Rhat **(b)** n_eff

**Figure D.41:** Rhat and n_eff for model C.6



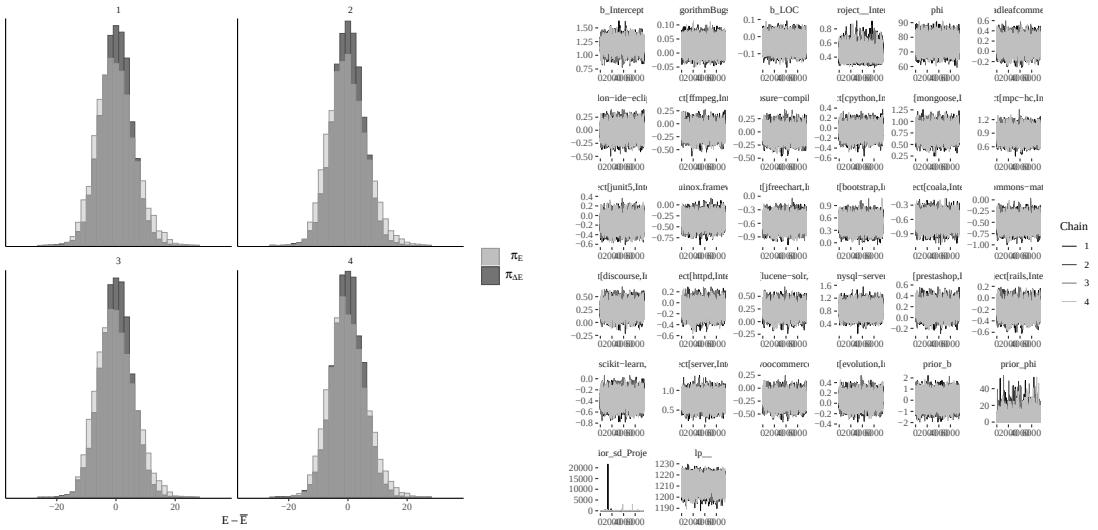

**(a)** Acceptance **(b)** Divergent Transitions

**Figure D.42:** Acceptance and divergent transitions for model C.6




**(a)** Step Size **(b)** Tree depth

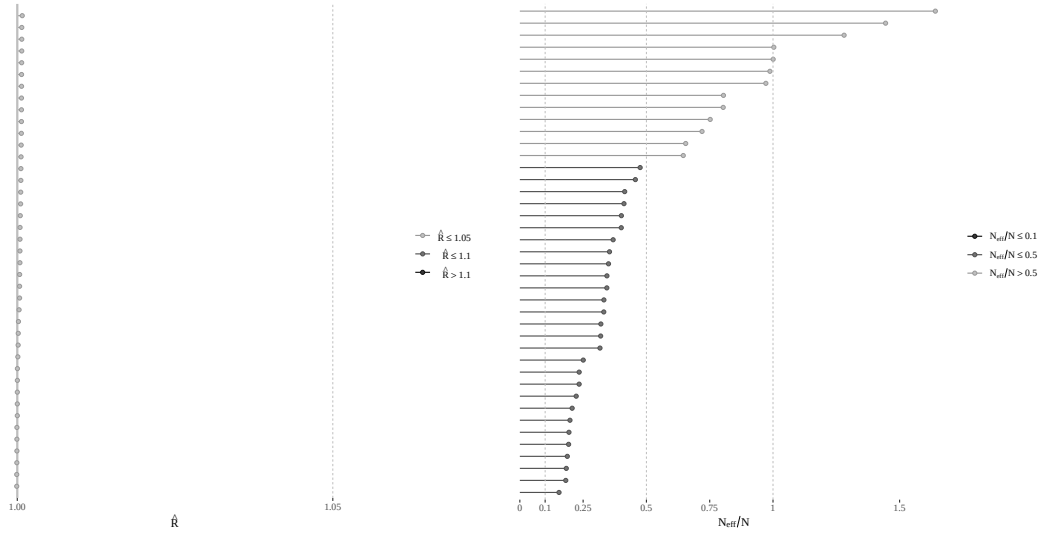
**Figure D.43:** Step size and tree depth for model C.6




**(a)** Energy **(b)** Trace plots

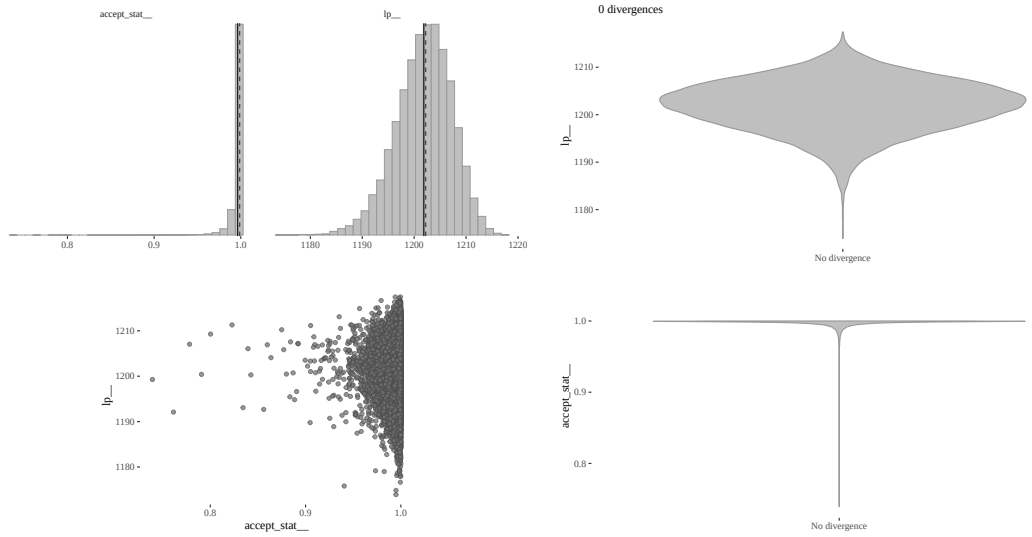
**Figure D.44:** Energy and trace plots for model C.6

### D.3.4 AUCECEXAM Model 2



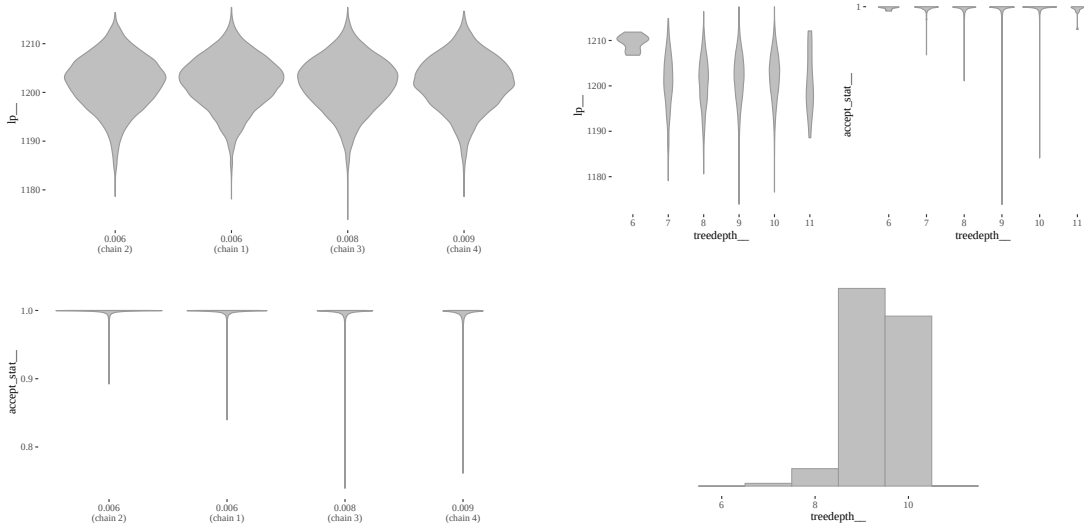

**(a)** Rhat **(b)** n_eff

**Figure D.45:** Rhat and n_eff for model 4.6



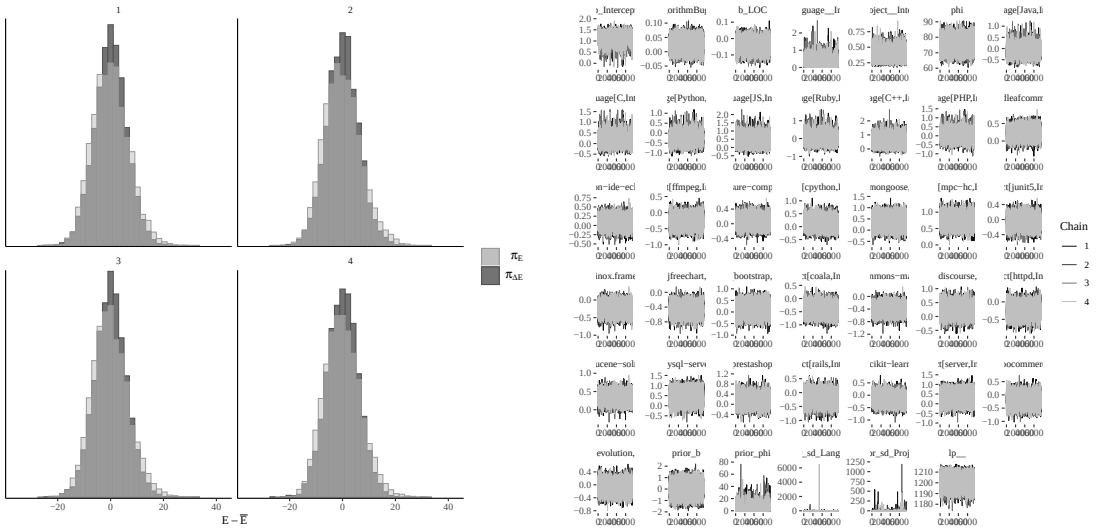

**(a)** Acceptance **(b)** Divergent Transitions

**Figure D.46:** Acceptance and divergent transitions for model 4.6




**(a)** Step Size **(b)** Tree depth

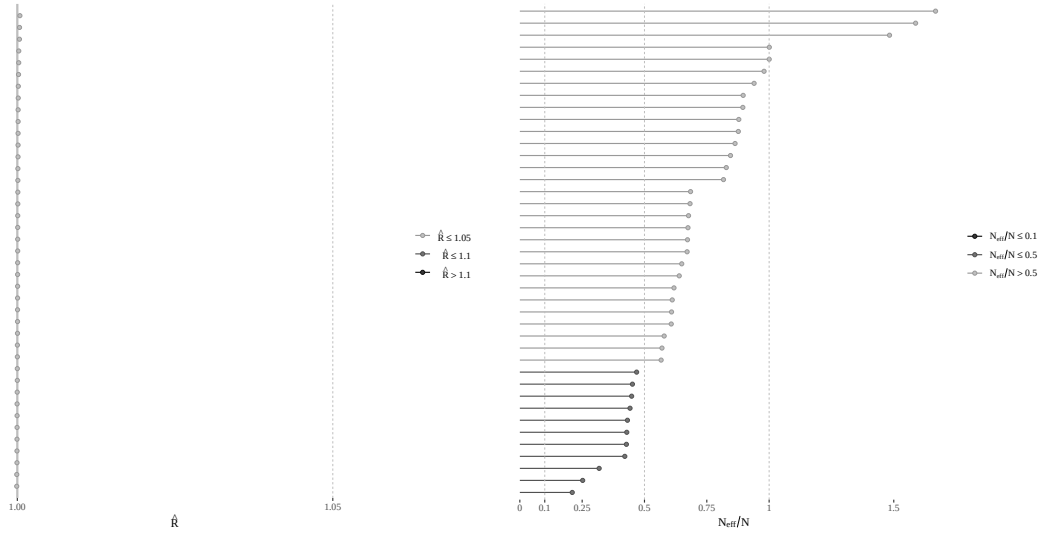
**Figure D.47:** Step size and tree depth for model 4.6




**(a)** Energy **(b)** Trace plots

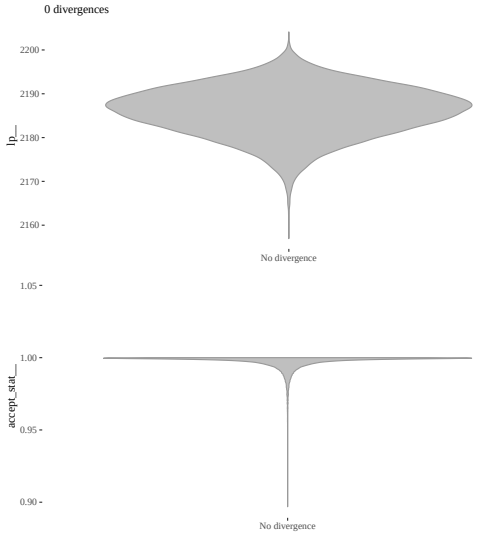
**Figure D.48:** Energy and trace plots for model 4.6

## D.3.5 EXAM25



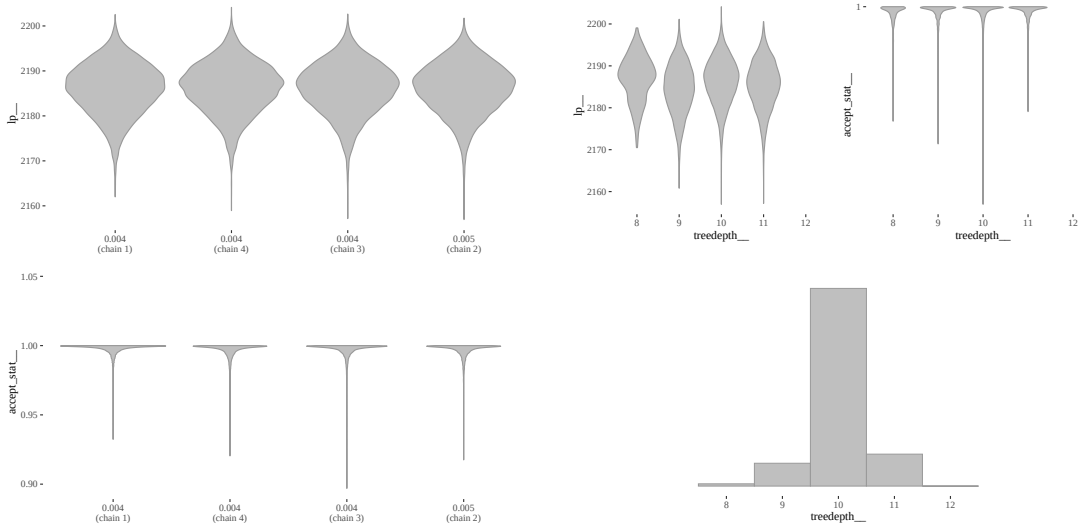

**(a)** Rhat **(b)** n_eff

**Figure D.49:** Rhat and n_eff for model 4.7



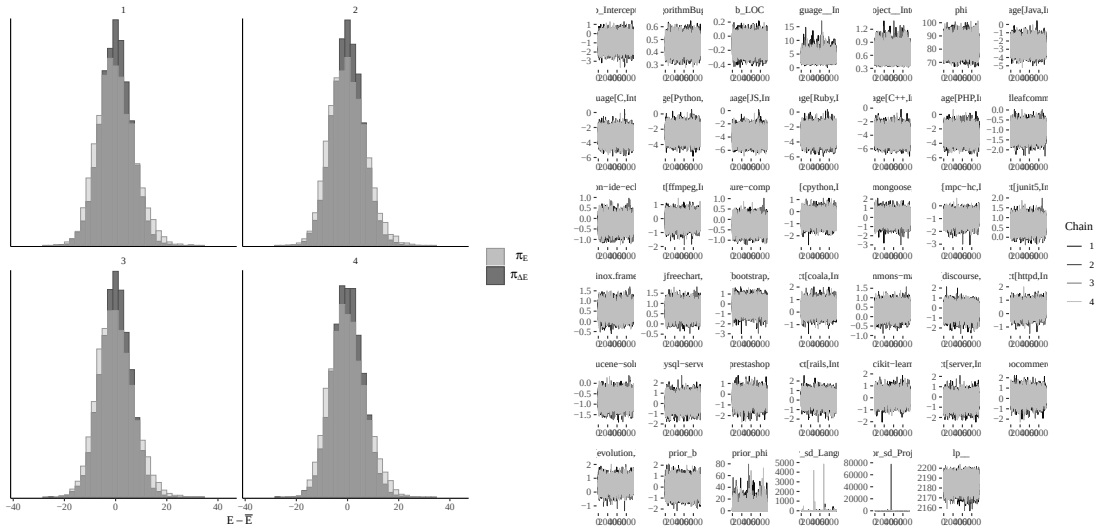

**(a)** Acceptance **(b)** Divergent Transitions

**Figure D.50:** Acceptance and divergent transitions for model 4.7




**(a)** Step Size **(b)** Tree depth

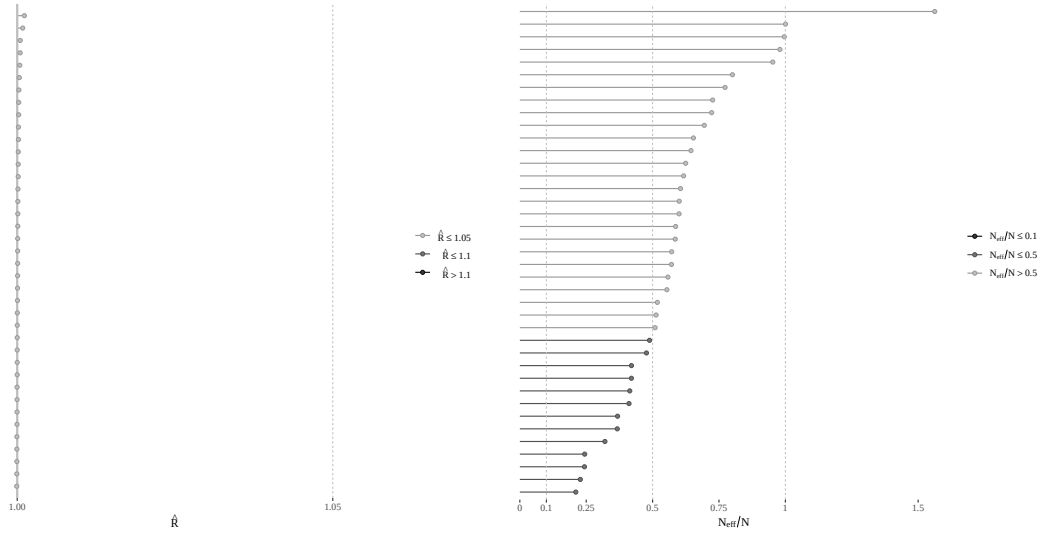
**Figure D.51:** Step size and tree depth for model 4.7




**(a)** Energy **(b)** Trace plots

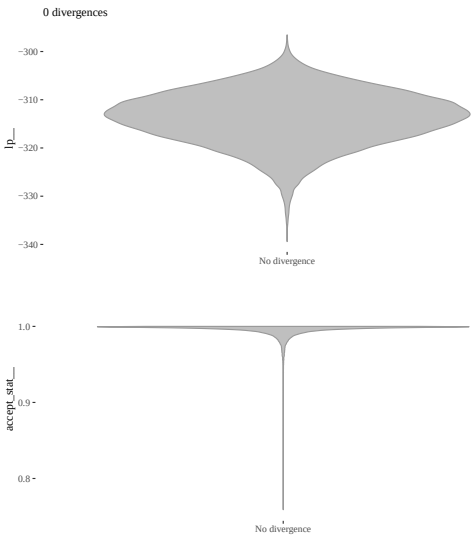
**Figure D.52:** Energy and trace plots for model 4.7

### D.3.6 EInspect25EXAM



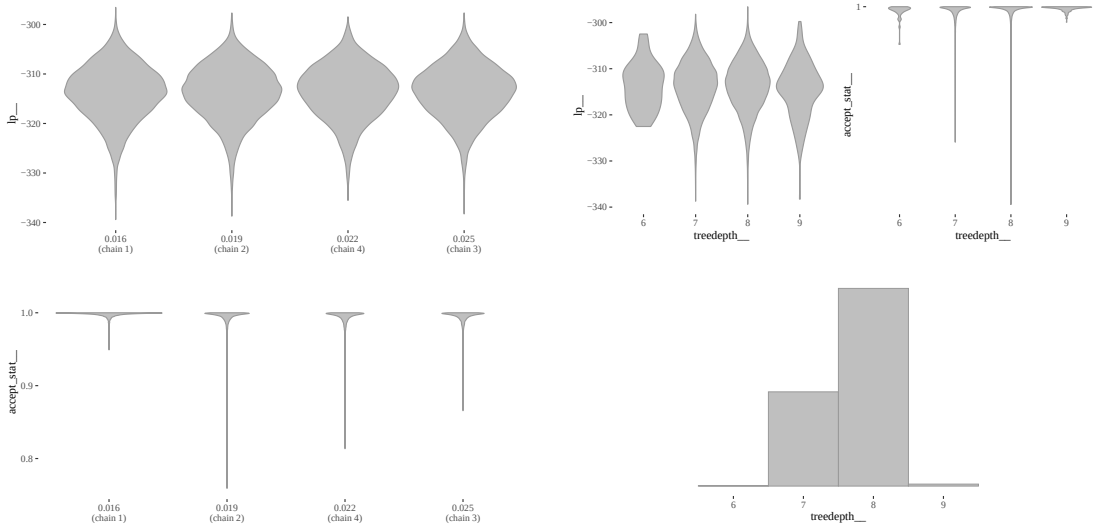

**(a)** Rhat **(b)** n_eff

**Figure D.53:** Rhat and n_eff for model 4.8



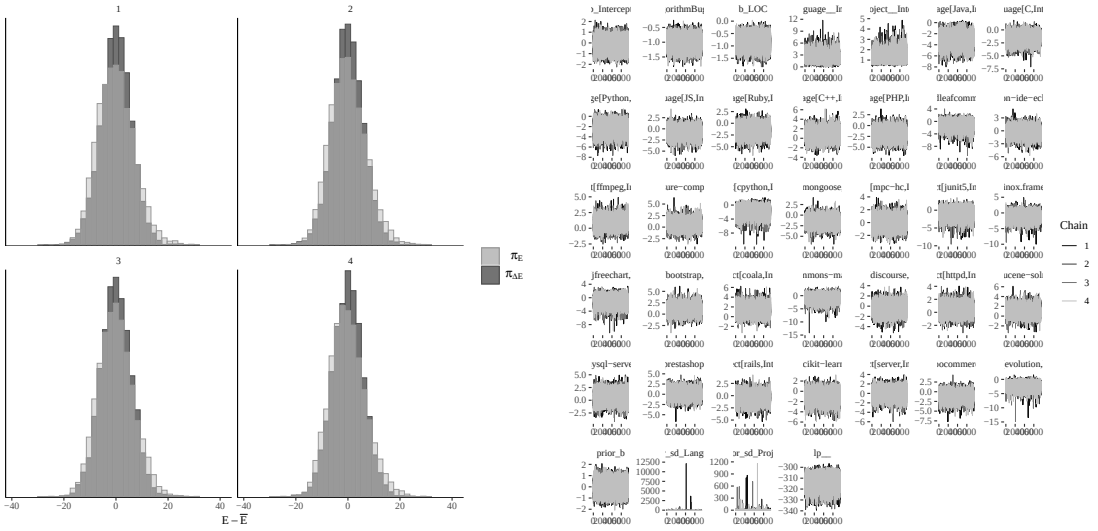

**(a)** Acceptance **(b)** Divergent Transitions

**Figure D.54:** Acceptance and divergent transitions for model 4.8




**(a)** Step Size **(b)** Tree depth

**Figure D.55:** Step size and tree depth for model 4.8




**(a)** Energy **(b)** Trace plots

**Figure D.56:** Energy and trace plots for model 4.8