This is the author prepared accepted version. © 2016 IEEE. The published version is: https://doi.org/10.1109/QRS.2016.54
Leif Jonsson, David Broman, Måns Magnusson, Kristian Sandahl, Mattias Villani, and Sigrid Eldh. Automatic Localization of Bugs to Faulty Components in Large Scale Software Systems using Bayesian Classification. In Proceedings of IEEE International Conference on Software Quality, Reliability & Security (QRS), Vienna, Austria, IEEE, 2016.

Automatic Localization of Bugs to Faulty Components in Large Scale Software Systems using Bayesian Classification

Leif Jonsson*, David Broman‡, Måns Magnusson†, Kristian Sandahl†, Mattias Villani† and Sigrid Eldh*
*{leif.jonsson,sigrid.eldh}@ericsson.com, †{mans.magnusson,kristian.sandahl,mattias.villani}@liu.se, ‡dbro@kth.se
*Ericsson AB,†Linköping University, ‡KTH Royal Institute of Technology and UC Berkeley

Abstract

We suggest a Bayesian approach to the problem of reducing bug turn-around time in large software development organizations. Our approach is to use classification to predict where bugs are located in components. This classification is a form of automatic fault localization (AFL) at the component level. The approach only relies on historical bug reports and does not require detailed analysis of source code or detailed test runs. Our approach addresses two problems identified in user studies of AFL tools. The first problem concerns the trust in which the user can put in the results of the tool. The second problem concerns understanding how the results were computed. The proposed model quantifies the uncertainty in its predictions and all estimated model parameters. Additionally, the output of the model explains why a result was suggested. We evaluate the approach on more than 50000 bugs.

Keywords

Machine Learning, Fault Detection, Fault Location, Software Maintenance, Software Debugging, Software Engineering

1. Introduction

In large support organizations, it is common to have a multi-tiered organizational structure. The 1st line support acts as a filter and routing function. It filters out issues that are not bugs, and routes issues that need more analysis further down the organizational chain. Bug reports may have to traverse many organizational layers, geographically located in different time zones, before reaching the final development organization. By contrast, bug routing in *open source software* (OSS) development projects is typically more direct: bugs are assigned to developers directly and are not traversing through a large support organization. In this paper we study the general field of bug handling, and how to automate bug handling processes. The work is based on experiences from the multi-tiered support organization at Ericsson.

Efficiency in the bug handling process is essential, and previous research [1], [2], [3], [4], [5] has shown how the process can be partially automated using *machine learning* (ML). The field can be split into two areas: i) routing of bugs to human designers, and ii) localization of faults to specific parts of the software. The *routing of bugs* can be done either to an individual person [3], [4] or to a design team [1], [2], [5]. The *fault localization* can be done either at the component level [6], at the file level [7], [8], or at a finer grained level such as a method, a statement, or individual lines of code (see the survey by Wong et al. [9] for an overview of the area). When the localization is at a fine grained level, the area is usually called *automatic fault localization* (AFL). In this paper, we focus on automatic fault localization *at the component level*. In large support organizations with hundreds to thousands of developers, a more detailed localization is too fine grained in the earlier phases of the bug handling processes.

Very few studies in the AFL area are performed on large software systems where AFL is perhaps most needed. This is evident from a large survey [9] of more than 300 papers on fault localization, where only three papers study programs with more than one million lines of code. Moreover, the median number of lines of code for the projects in this survey is 2463. By contrast, our work focuses on large industrial systems: the main evaluation is on an industrial proprietary telecommunication system. For purposes of replication, we also evaluate the approach on three OSS projects: Mozilla, Eclipse, and Gnome.

In this paper we address two problems originally identified by Parnin and Orso [10]. In their rather unique user study on AFL techniques, they report ([10], p8):

> *"We also identified a need to improve the explanatory capabilities of automated debugging tools. Developers were quick to disregard the tool if they felt they could not trust the results or understand how such results were computed."*

The first problem concerns *trust* of the prediction. Similar to what was pointed out in the study above, users and managers at our organization question why they should trust a "black-box" ML system[1]. The problem concerning uncertainty in the results from the tool manifests itself when a user is employing a fault localization tool and is getting a recommendation, but the tool does not offer any measure of the confidence in the recommendation. The question is then how much the user should trust and put an effort in following the recommendation

1. In this paper, the word *system* can have two separate meanings. In cases where confusion can arise, we denote the machine learning system that implements our approach as *the ML system*, and the production system on which the bug reports are reported as *the target system*.

of the tool. If the user knew that the confidence in the recommendation was high, that would motivate the user to follow the recommendation. If the tool reported a low confidence in the recommendation, the user might consider other alternatives.

The second problem concerns the ability to *understand* how a recommendation was computed. A user that understands the model and the basis for the given prediction would be more inclined to follow the recommendation than if the model is complicated and hard to understand.

We suggest to use a supervised linear Bayesian model called DOLDA [11] for component-level AFL. The classification part of DOLDA is the Diagonal Orthant Probit Model (DO-probit), which is a Bayesian linear classifier that can handle many classes. DOLDA further includes a Latent Dirichlet Allocation (LDA) [12] component that is used to summarize and model unstructured text. Unlike standard TF-IDF text models, LDA finds higher level semantic *topics* or *themes* (See TABLE 1 for examples of five topics) in the text of the bug data that can be interpreted by humans. The Bayesian model gives a posterior distribution over the predictions that completely quantifies the uncertainty *both in the predictions and all other model parameters*. The prediction uncertainty includes both the noise in the data and the uncertainty in the parameters. In Bayesian analysis a prior is always used. We study how a new Bayesian prior called the Horseshoe by Carvalho et al. [13] helps to simplify interpretation of the model output.

Below is an outline of our main contributions:

- We are the first to describe an approach for solving the problem of component-level automatic fault localization by using a fully Bayesian supervised LDA model. The model caters for the basic needs of incorporating structured data and unstructured text in one coherent probabilistic model (Sections 2.2-2.3).
- We show how a linear Bayesian approach to component-level AFL gives many concrete benefits. Two of the main ones are the quantification of the uncertainty in the model parameters and predictions, and the linearity of the model which makes it easy to interpret and explain to end users (Sections 2.2-2.3).
- We study how the posterior distribution of the Bayesian model predictions can be used to balance accuracy and accept rate of automatic predictions (Section 4.1).
- We evaluate how the interpretation of the model is affected by using different Bayesian priors in the model. We show that using a Horseshoe prior leads to simpler model interpretation (Section 4.2).
- We compare the prediction accuracy of our supervised LDA model to other state-of-the-art models (both with and without LDA) for bug localization (Section 4.3).
- We have extracted several years of bug data from a large industrial software project, which is used in the evaluation. We also provide an analysis of three important OSS to make the results comparable to related studies (Section 3).

Topic	Topic label	Top 10 words in topic
11	HTTP	proxy server http network connection request connect error www host
27	Layout	div style px background color border css width height element html
28	Connection Headers	http cache accept en public localhost gmt max modified alive
55	Search	search google bar results box type find engine enter text
82	Scrolling	scroll page scrolling mouse scrollbar bar left bottom click content

TABLE 1: Top words for signal topics (Z11, Z27, Z28, Z55 and Z82) for the class Core.Networking from the Mozilla dataset. The topic label is manually assigned.

2. Deployment Scenario

In the following section, we first give some technical preliminaries. After that, we show how our approach can be deployed. The deployment is split into two main scenarios: *accepted* and *rejected* predictions.

In our approach, a bug report is encoded as a set of *variables*. Each variable represents an aspect of the bug report. For instance, one variable can represent in which version the software bug was reported. Another variable can represent at which site the bug was discovered.

We represent the text of a bug report by the LDA topics present in the bug report. LDA is a powerful probabilistic mixture model that, like the simpler TF-IDF model typically used to model text, uses a *bag-of-word* representation of the text. In *bag-of-word* models, the word order is ignored, only the frequency of the words are taken into consideration. TF-IDF is in its simplest form a frequency count of how many times a word occurs in a text (TF) relative its count in the whole corpus (IDF). When modeling the text with LDA, we use the so called the *document-topic distribution*. The distribution consists of a vector of topic-proportions. Each topic proportion tells how large proportion of that topic that exists in the bug report. Each proportion in the vector is encoded as one variable. If we have selected 40 topics, then the text in the bug report will be represented by 40 variables.

Each *component* that the system can classify to has a set of weights associated to it. The weights are called β *coefficients*. There is *one* β *coefficient per variable in the bug report encoding*. The *value* of the β coefficients of one component represents the importance of its corresponding variable when classifying a bug report to that specific component. The value of *the* β *coefficients, the topics and the topic distribution of the bug reports are learned by the system during the training phase*. After the training is finished, they can be used for classification of new bug reports. Figure 1 shows a plot of the β coefficients for the Mozilla component Core.Networking. The β coefficients for topics 11, 27, and 28 are annotated in the figure. The topic variables are encoded with a "Z" appended with the topic number. Descriptions of five example topics are listed in TABLE 1.

We call a variable a *signal variable* for a component if its

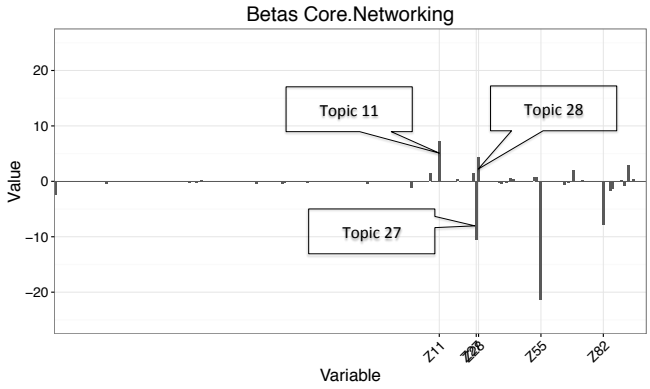


Fig. 1: β coefficients for the Core.Networking component. The Core.Networking component has five *signal variables*, Z11, Z27, Z28, Z55 and Z82 which represents topics 11, 27, 28, 55 and 82.

corresponding β coefficient has a value that lies more than two standard deviations (SD) away from the β coefficients mean. In figures, signal variables are marked with a vertical line and its name printed on the X-axis and its value on the Y-axis. Signal variables are particularly important and serve as characteristic variables for a component.

2.1. Quantification of Uncertainty

The Bayesian approach gives us a so called posterior predictive distribution over the classes in the target system. This distribution quantifies the uncertainty in the model predictions. In our context, *the components represent the different classes of the system*. Figure 2 shows an example of a prediction with very low uncertainty. The model predicts, with high confidence, that component 12 contains the fault indicated by bug report HP19611. Sometimes, for convenience, we also talk about *precision* which is the inverse of uncertainty. Because of the low uncertainty (high precision) case in Figure 2, we would *accept* the prediction and automatically suggest that the bug is located in the predicted component.

By contrast, Figure 3 shows that for bug report HP32309,

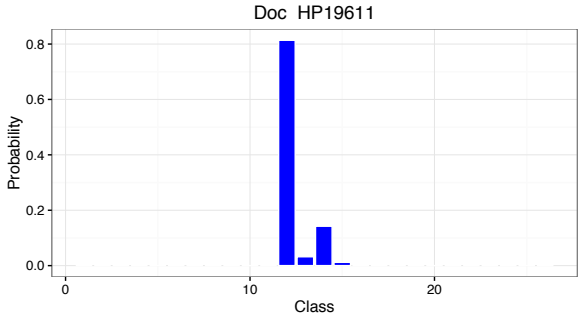


Fig. 2: Probability distribution over the classes with very low uncertainty.

the uncertainty is comparatively high (notice the range difference on the y-axis compared to Figure 2). If the system was tuned not to accept too high uncertainty in the prediction, it would *reject* this prediction and leave the decision up to a human.

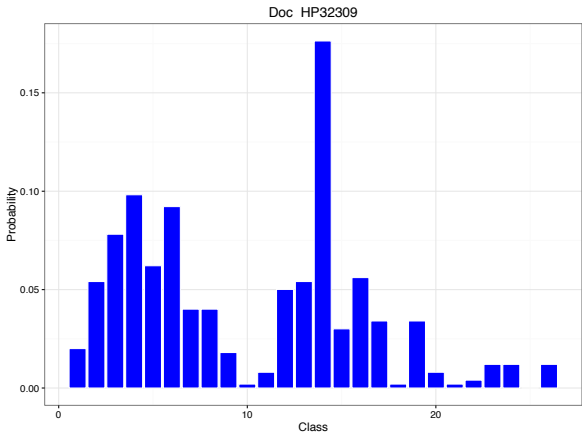


Fig. 3: Probability distribution over the classes with comparatively high uncertainty.

2.2. Accepted Predictions

Figure 4 shows a simplified view of the deployment of our approach. A new bug report arrives at the 1st line support organization and is registered in a bug tracking system such as Bugzilla, Jira or similar. In the extra *ML-Phase* of the process, the ML system is called with the new bug report. At this first stage, the ML system tries to predict where in the target system the bug is located.

A prediction is made by first calculating the so called *latent utility* of each component in the target system for the new bug report. The total latent utility for a component is the sum of all the β coefficients corresponding to that component, multiplied by their corresponding variables in the bug report. The component with the highest latent utility will be the component that the bug is classified to. This straightforward interpretation is an important property of the DOLDA model.

The uncertainty measure of the model output then is used to decide if the uncertainty in the prediction is too high. If the uncertainty is higher than a set uncertainty threshold, the bug report is rejected, otherwise it is accepted. The uncertainty threshold can be configured by a tunable system parameter (see Section 4.1).

If the prediction is accepted, the bug can be automatically classified as belonging to a specific component. The analysis can now be presented for the 1st line support personnel (if the system works well, this step could be completely automated to not have any human intervention). At this point, we face the same problem as presented in the Parnin and Orso paper. How much faith can the user put in the prediction? A visualization of the uncertainty measure similar to Figure 2 together with

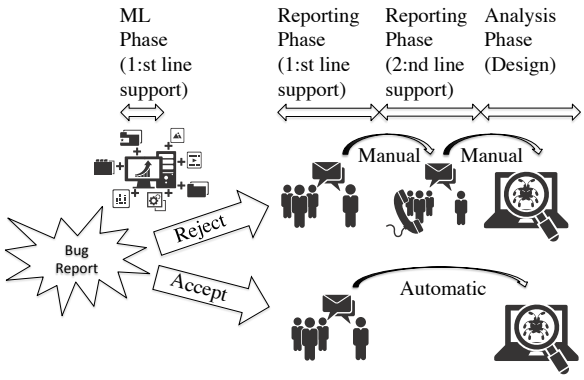


Fig. 4: Deployment use-case.

topics that summarize the bug report help to convince the user that the prediction is accurate.

With an accepted prediction, the user is presented with an option to directly send the bug report to the design organization that owns that particular component. With this approach, the manual analysis of the 1st and 2nd line support organization can be bypassed and substantial time can be saved. At the receiving development organization, the classification to a component helps the design teams pull the most suitable bug reports to handle. The teams pull reports classified to components in which they have expertise. The classification gives the team a quick pinpointing of which component to first start to examine and hence narrows down the problem search space. Here again, the uncertainty measure, the topic summary and signal variables present in the bug report can help the design teams to understand the automatic prediction. The topic summary can quickly show a team, without having to study the bug report in detail, that the bug report concerns for instance the topics *HTTP* and *Connection Headers* (See topics 11 and 28 in TABLE 1).

2.3. Rejected Predictions

Even when the bug classification is rejected due to too high uncertainty, the output of our approach is useful. The information can be used by the support personnel to make better decisions during the processing of the bug. The model output consists of:

- the probability distribution over the components (see Figures 2 and 3)
- the LDA document-topic distribution for the bug report
- the LDA topics that summarizes the text in the bug report (See TABLE 1)
- the β coefficients used for classifying to the different components (see Figure 1)

To understand why a classification was made by the model, we can inspect the signal variables of the class. As an example, we look at the β coefficients in Figure 1 for the Core.Networking component.

In Figure 1 we have 100 topics, encoded as Z0-Z99. We have five signal topics Z11, Z27, Z28, Z55, and Z82. In TABLE 1, we list the top words for the five signal topic variables for the class Core.Networking. We see that Core.Networking has positive β coefficients for topics 11 and 28 and negative for topic 27, 55, and 82. This means that bug reports that have high proportions of topics 11 and 28 tend to be classified to this component. While bug reports with high proportions of either topics 27, 55, and 28 tend not to. Since DOLDA is a linear model, all variables will affect the prediction, but the variables with the highest values (positive or negative) will matter the most. Variables with β coefficient zero will have no effect on the classification.

It makes intuitive sense that a *HTTP* topic (topic 11) and a *Connection Headers* topic (topic 28) have strong positive effect on the Core.Networking component since these are *networking* type topics. It also makes intuitive sense that topic 27, *Layout*, topic 55, *Search*, and topic 82, *Scrolling*, have a negative effect on Core.Networking since these are *not* typical *networking* concepts.

The human support operator can be presented with how much each topic is represented in the bug report and how the different topics affect predictions to the different components. A high proportion of a topic in a bug report and a high value of the β coefficient in a component for that topic means a high contribution to the latent utility for that component. When the bug report is routed through the 1st and 2nd line support organizations and the development organization, the bug topics can be used to summarize a bug report for easier overview by the support personnel.

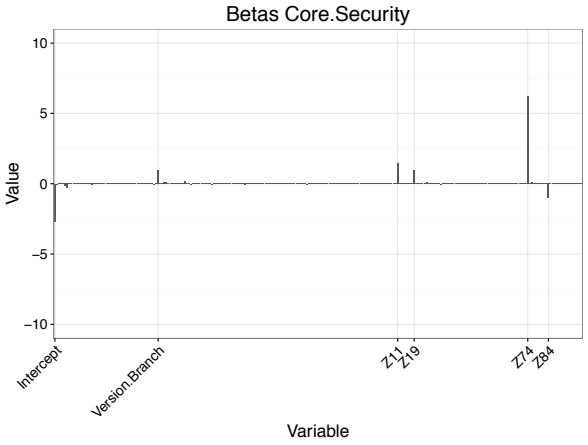


Fig. 5: β coefficients for the Core.Security component. The Core.Security component has five *signal variables*

In the model output, nominal variables are encoded with the variable name appended with a "." and the nominal value. In Figure 5 we see an example with a nominal signal variable called *Version*. This means that the version called *Branch* is important for classifying a bug to the Core.Security component. If a bug written on the version called *Branch* was classified as located in Core.Security, the high β coefficient on the *Version.Branch* for the component Core.Security raises the latent utility for that class and thus help to explain why

Dataset	No. Bug reports	No. Classes	Vocab. Size
Mozilla Core & Firefox & SeaMonkey	15000	118	3505
Eclipse	15000	49	3367
Gnome (gnome-core & gtk+)	15000	39	4242
Telecom	9778	26	5386

TABLE 2: Dataset statistics. Vocab. size is the size of the vocabulary of the unstructured data after stop word and rare word trimming.

Dataset	Dependent Variable	Independent Variables
Mozilla	Component	Classification, Version, Architecture, Priority, LDA Document Topic Means of first short and long description
Eclipse	Component	Classification, Version, OS, Priority, LDA Document Topic Means of first short and long description
Gnome	Component	OS, Severity, Priority, LDA Document Topic Means of first short and long description
Telecom	Component	Customer, Faulty revision, Siteid, Prio, Document Topic Means of Observation text

TABLE 3: Dependent and Independent variables per dataset.

the prediction was made. Another interesting and important fact to notice in Figure 5 is that, as for the Core.Networking component in Figure 1, topic 11 is important to classify to Core.Security. That is, the model does *not* map only one topic to one class, but the *full combination* of β coefficients affect the prediction. One variable can have high β coefficients for several classes.

To summarize; the probability distribution over the component classification shows which components that are most likely to contain a bug. The β coefficients explain which variables are important when classifying a bug report to a component. The β coefficients together with the bug report explain why a classification was made. The LDA topics quickly summarize a bug report and can be displayed to the human operator to quickly judge if a classification is reasonable.

3. Experimental Setup

In this section, we discuss the datasets and preparations we have carried out in order to perform the various experiments.

We evaluate our suggested approach on four datasets: three datasets are open source software (OSS) projects, and one dataset is proprietary and collected from the telecommunications industry. TABLE 2 summarizes some basic information about the datasets.

For all three OSS datasets, we randomly extracted 15000 bug reports. We have previously [2] shown that the time between bug reports is an important aspect when training a bug prediction system. Very old bug reports do not reliably predict new reports. To get a selection that was random, but from roughly the same time period, the selection was made sequentially from the start of the bug repository. We then iterate through each bug report giving it a 50% chance of inclusion until we had extracted 15000 reports. Bug reports marked with *error* in the OSS datasets were filtered out. For the OSS datasets, we also extracted additional variables. TABLE 3 summarizes which variables that were used for each of the datasets.

For the OSS datasets, we selected the *short_desc* and *first long_desc* and merged these as the unstructured text. The short_desc is a slogan and the first long_desc is a longer description of the observed problem entered by the filer of the bug at submission time. The *Component* variable was used as the dependent variable. The rest of the variables were used as features in the inference algorithm.

For the Telecom data set, we used our domain expertise to filter out outliers. Example of outliers are bug reports that consisted of highly temporary components that were created outside of the normal bug handling flow. Bug reports that were later deemed not to be bugs was also filtered out. These include, change requests and other types of issues not related to bugs. For this dataset, we selected the additional variables based on our earlier experiences [2] (see TABLE 3). The *heading* field was merged with the *observation* field. We additionally transformed the nominal variable faulty revision to cover only major revisions. For example, an R10B/25 revision would be transformed to R10.

For all datasets, we use standard stopword lists and normalization practices. We do lowercasing but not stemming. Furthermore, we remove rare words which occur less than 30 times in the data set.

In LDA models, the number of topics in the model must be selected. We have elected to run two sets of experiments with 40 and 100 topics. These were selected using a technique akin to the approach by Griffiths and Steyvers [14].

4. Results and Discussion

4.1. Tuning Prediction Accuracy and Acceptance Rate

In Section 4.3 we evaluate the quality of the predictions of classification models by looking at the *prediction accuracy* of the models. The prediction accuracy is the number of correctly classified bug reports divided by the total number of bug reports classified. In this section we show how we can use the model uncertainty our approach supplies to balance between the prediction accuracy and how many bug reports that can be automatically classified. We define *acceptance rate* as the number of bugs accepted as automatic classifications relative to the total number of classified bug reports.

We see from Figure 6 that when we choose to include only predictions with higher precision (i.e a lower variance and less uncertainty) we get an increase in the prediction accuracy. This

is the expected behavior, and if the model was perfect, zero uncertainty would *always* give a correct prediction. Due to noisy data and an imperfect model we still see a less than 100% correct prediction accuracy even with no uncertainty in the classification of the bug. In Figure 6 we plot the prediction accuracy at 20, 50, 80, 95, 99 and 100 % precision. That is, at 100 % precision we have no variance at all in the prediction and the posterior distribution peaks at one class only.

We can now tune the system to an acceptable level of accuracy relative to how many rejects an organization deems acceptable. As an example from Figure 6, if we select a prediction precision of around 80% we get a prediction accuracy of around 60% (top part) and acceptance rate of around 50% (lower part) of the bug reports. The higher prediction accuracy comes at the cost of a lower acceptance rate.

Using the mechanisms above, the Bayesian approach allows us very easily, without having to go outside the model, to implement something similar to the two-phased approach by Kim et al. [7]. An added benefit of our approach is that we can use the posterior for more advanced Bayesian decision approaches if desired. We, similar to Kim et al., see that if we restrict attention to the predictions with the lowest uncertainty, the prediction accuracy in general rises significantly with around 25 percentage points compared to the overall accuracy.

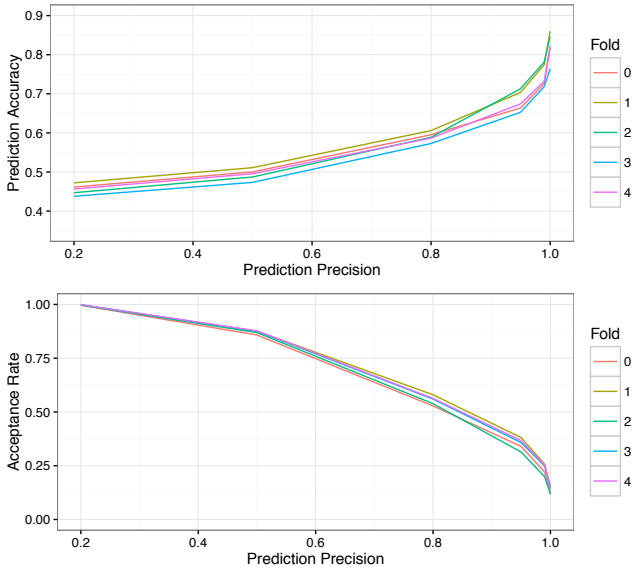
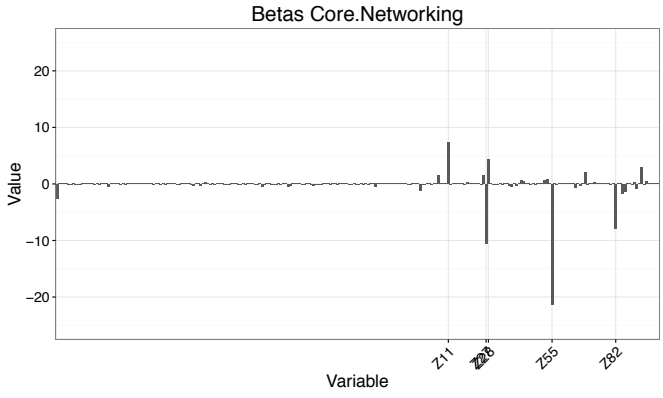


Fig. 6: Precision vs. accuracy and precision vs. acceptance rate plots from five experimental runs (folds) on the Mozilla dataset. The Horseshoe prior and 100 topics are used. The top graph shows that as the uncertainty in the prediction decreases (prediction precision increases) the prediction accuracy increases. Bottom graph shows that as we increase the required precision in the classification, more classifications are rejected and the ratio of accepted classifications decreases.

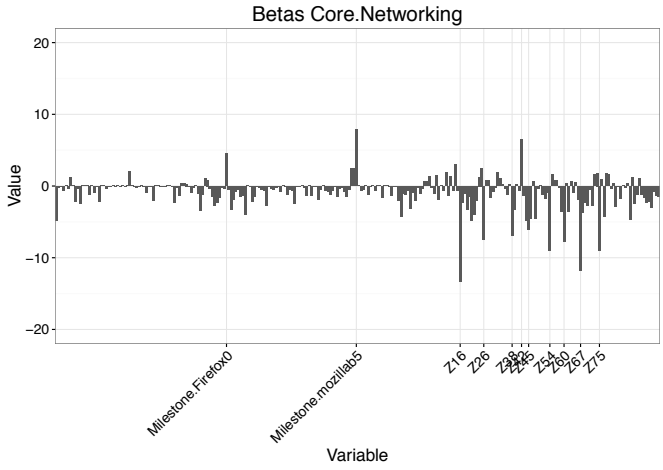
4.2. Model Interpretation

To understand a prediction, all signal variables need to be studied. It is therefore desirable that the number of signal variables is small. To this end, we compare the model behavior using two different priors; the Horseshoe [13] and the normal prior. The Horseshoe prior causes *most of the β coefficients to be zero, but a few will get high values*. By contrast, the normal prior results in small β coefficient values around zero. The property of compressing the values of the β coefficient towards zero is called *regularization*. The Horseshoe induces a strong regularization effect on the β coefficients similar to the Lasso [15].

Figure 7 shows a typical example of the difference in the beta coefficients between using the Horseshoe prior and the normal prior. We observe (by visual inspection of plots as those in Figure 7) similar behavior in the vast major-



(a) Horseshoe Prior



(b) Normal Prior

Fig. 7: Comparison of the effect on the β coefficients when using the Horseshoe (a) prior vs the normal (b) prior for the Mozilla class (component) Core.Networking. On the X-axis is the variable of the corresponding β coefficient. The value of the β coefficient is on the Y-axis.

Dataset	No. Topics	DOLDA Normal %	DOLDA Horseshoe %	Stacking TF-IDF %	LDA+Stacking %	LDA+KL %
Mozilla	100	**46** (79)	45 (71)	39 (*)	39	26
Eclipse	100	**61** (90)	**61** (87)	50 (*)	55	37
Gnome	100	**63** (85)	61 (81)	60 (*)	60	35
Telecom	100	72 (93)	71 (92)	**80** (*)	75	41
Mozilla	40	**43** (70)	42 (66)	39 (*)	39	19
Eclipse	40	**57** (85)	**57** (83)	50 (*)	54	30
Gnome	40	54 (69)	54 (68)	**60** (*)	55	33
Telecom	40	69 (91)	68 (89)	**80** (*)	73	36

TABLE 4: Summary of prediction accuracy experiments. We present the same figures for 40 and 100 topics for Stacking+TF-IDF because there is no notion of topics in that method, the (*) is a reminder of this. Figures in parentheses are the accuracies at minimum uncertainty in the prediction (which means a low acceptance rate, see Figure 6).

ity of cases. We see that the Horseshoe model prior gives substantially less noisy β coefficients. Figure 7a shows that with the Horseshoe we get five signal variables, Z11, Z27, Z28, Z55, and Z82, while the normal prior gives 11 signal variables. We remind the reader that the signal variables of a class are those variables most significant for predicting to a component. This shows that the Horseshoe prior leads to a much easier interpretation compared to the normal prior. We see this behavior in *all experiments* and *all classes*.

4.3. Prediction Accuracy

We evaluate the prediction accuracy of our approach on the selected datasets and compare it with three approaches that are similar to ours. The reason for evaluating precision accuracy is to show that the Bayesian approach using DOLDA gives at least as good prediction accuracy as state-of-the-art classification methods. While previous approaches [6], [16] have used support vector machines (SVM) for AFL, we have opted to use an ensemble technique called Stacked Generalization (or Stacking) by Wolpert [17] instead. This choice is motivated by our previous experiences in Jonsson et al. [2]. The first method we compare with is a Stacking model (including an SVM) which represents the text using TF-IDF. In the second model we replace the TF-IDF representation with an LDA text representation using the exact PC-LDA method of Magnusson et al. [18]. Both models include text and other variables. The third approach learns an LDA model from the training data and then extracts C (the number of classes in the data) number of class centroids from the document-topic distributions of the training bug reports. The classification is done by sampling a test bug report using the learned LDA model and comparing its document-topic distribution to the C centroids. The bug report is then classified to the class which has the smallest symmetrizied Kullback-Lebiler divergence between it and the corresponding class centroid. Additional variables are *not* included in this approach.

As can be seen from TABLE 4, in five out of eight experiments, the DOLDA based methods get slightly higher prediction accuracy compared to the other models (Stacking and TF-IDF, LDA + Stacking and LDA+KL), though more experiments are needed to decide if this holds in the general case. In our experiments, the DOLDA based methods are at least on par with the other methods we have compared. We recall that the LDA+KL approach does not use the additional variables, which may explain its weaker performance. Our results on the Telecom data are similar to Di Lucca's [5], despite substantially more classes (26 compared to eight).

Section 4.2 shows that DOLDA with the sparse horseshoe prior produces class probabilities that depend on a very small set of interpretable topics. The results in TABLE 4 is evidence that the sparsity in DOLDA comes at no reduction in prediction accuracy. Except for the Gnome dataset, the predictive performance of the model with 40 topics performs only marginally (around 3%) worse than the model with 100 topics.

5. Related Work

According to Chen et al. [19], there are more than 100 articles applying topic models such as LDA to software repositories. Unlike our approach, no previous research in AFL or bug routing have used one coherent Bayesian supervised LDA model for handling the text in combination with the other structured data in a bug tracking system.

Di Lucca et al. [5] study a problem very similar to ours in a large industry context. They evaluate five ML approaches to classifying software maintenance requests on a distributed telecommunication system. Unlike our approach none of the approaches are fully Bayesian or LDA-based.

Previous research [2], [6], [20], [21] in AFL and bug routing have examined various ML or statistics based approaches for solving the problem of locating a bug or finding the team which should handle the bug. Much previous research has focused on one aspect of the content in the bug tracking system, either the textual or the nominal attributes. However, recent research [2], [7] have shown that combining both the unstructured text and the additional attributes can give a better prediction accuracy than just using one of them.

Kim et al. [7] suggest an innovative *two-phase* model which first determines if the bug report contains enough information to make a good prediction, and only if so, the system actually performs the prediction. They show that the system can make better predictions by only trying to classify what they call *predictable* bug reports. We can easily implement a reject option or something akin to the two-phase approach using

the uncertainty over the predictions from our model. The difference compared to the two-phase approach of Kim et al. is that with our method we automatically get the uncertainty for *all* bug reports. For the bug reports where the uncertainty is too high, we do a reject. This also leads to an architectural difference compared to the two-phase approach; *we only need one training step and one model* while in the two-phase approach there are two training steps, *Phase 1* and *Phase 2* with two different models that need to be trained. Kim et al. do not report if Phase 1 produces an uncertainty measure.

Somasundaram et al. [6] compare an LDA plus Kullback Leibler (LDA-KL) approach with an SVM-LDA and SVM-TF-IDF approach. They found that the LDA-KL approach was more consistent (i.e having the least variance in the precision/recall measures for the different number of components) in its performance. In terms of average recall, the three methods produced similar results with SVM-LDA consistently having the lowest score. In both LDA-KL and SVM-LDA it is hard to interpret the output from the model. There is no obvious way to interpret how one topic affects the classification. By contrast, our approach has none of these problems. DOLDA learns one single coherent Bayesian linear classification model for all of the training data based on the combination of the additional variables and the LDA topic distribution of the training documents. In our view, this makes for a simpler interpretation of the model. We can study the weights of the linear model and directly interpret how much one particular topic or variable affects a particular class. This is much harder with the LDA-KL or SVM-LDA approach. Furthermore, when a user receives the result of the prediction, we can highlight which words in the bug report that are particularly relevant for the classification.

6. Conclusion

Three major aspects of AFL stand out as lacking in the current literature. First, little research in AFL is performed on large scale industrial systems. Second, research shows that we need AFL models that can reliably quantify the uncertainty in the output from the model so that users know how much they can trust the output from the model. Third, the model and its results need to be easy to understand. To alleviate these deficiencies, we suggest using a fully Bayesian supervised LDA model for component-level AFL. We show that our approach can tackle large systems and have competitive performance compared to other approaches. This is done by comparing the predictive performance of the proposed approach on four large datasets. Furthermore, we have shown that our proposed model gives clear answers to both questions of uncertainty and interpretability. In addition to this, the model handles both unstructured text together with structured variables.

Acknowledgments

We acknowledge the gracious support from Ericsson AB.

References

[1] L. Jonsson, D. Broman, K. Sandahl, and S. Eldh, "Towards automated anomaly report assignment in large complex systems using stacked generalization," in *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*. IEEE, 2012, pp. 437–446.

[2] L. Jonsson, M. Borg, D. Broman, K. Sandahl, S. Eldh, and P. Runeson, "Automated bug assignment: Ensemble-based machine learning in large scale industrial contexts," *Empirical Software Engineering*, pp. 1–46, 2015.

[3] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?" in *Proceedings of the 28th international conference on Software engineering*. ACM, 2006, pp. 361–370.

[4] D. Čubranić and G. Murphy, "Automatic bug triage using text classification," *Proceedings of Software Engineering and Knowledge Engineering*, pp. 92–97, 2004.

[5] G. d. Lucca, M. D. Penta, and S. Gradara, "An approach to classify software maintenance requests," in *Software Maintenance, 2002. Proceedings. International Conference on*. IEEE, 2002, pp. 93–102.

[6] K. Somasundaram and G. C. Murphy, "Automatic categorization of bug reports using latent Dirichlet allocation," *Proceedings of the 5th India Software Engineering Conference ISEC12*, pp. 125–130, 2012.

[7] D. Kim, Y. Tao, S. Kim, and A. Zeller, "Where should we fix this bug? a two-phase recommendation model," *Software Engineering, IEEE Transactions on*, vol. 39, no. 11, pp. 1597–1610, 2013.

[8] J. Zhou, H. Zhang, and D. Lo, "Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports," in *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 2012, pp. 14–24.

[9] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Transactions on Software Engineering*, vol. PP, no. 99, pp. 1–1, 2016.

[10] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?" *Proceedings of the 2011 International Symposium on Software Testing and Analysis ISSTA 11*, p. 199, 2011.

[11] M. Magnusson, L. Jonsson, and M. Villani, "DOLDA - a regularized supervised topic model for high-dimensional multi-class regression," *ArXiv e-prints*, Jan. 2016.

[12] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent Dirichlet Allocation," *Journal of Machine Learning Research*, vol. 3, no. 4-5, pp. 993–1022, 2003.

[13] C. M. Carvalho, N. G. Polson, and J. G. Scott, "The horseshoe estimator for sparse signals," *Biometrika*, vol. 97, no. 2, pp. 465–480, 2010.

[14] T. L. Griffiths and M. Steyvers, "Finding scientific topics," *Proceedings of the National Academy of Sciences*, vol. 101, no. suppl 1, pp. 5228–5235, 2004.

[15] R. Tibshirani, "Regression shrinkage and selection via the lasso: a retrospective," *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, vol. 73, no. 3, pp. 273–282, 2011.

[16] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, "A systematic literature review on fault prediction performance in software engineering," *Software Engineering, IEEE Transactions on*, vol. 38, no. 6, pp. 1276–1304, 2012.

[17] D. H. Wolpert, "Stacked generalization," *Neural networks*, vol. 5, no. 2, pp. 241–259, 1992.

[18] M. Magnusson, L. Jonsson, M. Villani, and D. Broman, "Parallelizing LDA using Partially Collapsed Gibbs Sampling," *ArXiv e-prints*, Jun. 2015.

[19] T.-H. Chen, S. W. Thomas, and A. E. Hassan, *A survey on the use of topic models when mining software repositories*. Empirical Software Engineering, 2015.

[20] X. Xia, D. Lo, X. Wang, and B. Zhou, "Dual analysis for recommending developers to resolve bugs," *Journal of Software: Evolution and Process*, vol. 27, no. 3, pp. 195–220, 2015.

[21] S. Lukins, N. Kraft, and L. Etzkorn, "Bug localization using latent Dirichlet allocation," *Information and Software Technology*, 2010.