The Journal of Systems and Software 122 (2016) 72–82

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Software defect prediction using doubly stochastic Poisson processes driven by stochastic belief networks

Andreas S. Andreou, Sotirios P. Chatzis*

Department of Electrical Engineering, Computer Engineering and Informatics, Cyprus University of Technology, Limassol 3036, Cyprus

ARTICLE INFO

Article history:
Received 24 November 2015
Revised 29 August 2016
Accepted 1 September 2016
Available online 3 September 2016

Keywords:
Software defect prediction
Doubly stochastic Poisson process
Sampling importance resampling
Stochastic belief network

ABSTRACT

Accurate prediction of software defects is of crucial importance in software engineering. Software defect prediction comprises two major procedures: (i) Design of appropriate software metrics to represent characteristic software system properties; and (ii) development of effective regression models for count data, allowing for accurate prediction of the number of software defects. Although significant research effort has been devoted to software metrics design, research in count data regression has been rather limited. More specifically, most used methods have not been explicitly designed to tackle the problem of metrics-driven software defect counts prediction, thus postulating irrelevant assumptions, such as (log-)linearity of the modeled data. In addition, a lack of simple and efficient algorithms for posterior computation has made more elaborate hierarchical Bayesian approaches appear unattractive in the context of software defect prediction. To address these issues, in this paper we introduce a doubly stochastic Poisson process for count data regression, the failure log-rate of which is driven by a novel latent space stochastic feed-forward neural network. Our approach yields simple and efficient updates for its complicated conditional distributions by means of sampling importance resampling and error backpropagation. We exhibit the efficacy of our approach using publicly available and benchmark datasets.

1. Introduction

Software defect prediction has been a significant research topic in software engineering for the last 30 years, with studies concentrating on estimating how many defects remain in a system, identifying possible associations between defects, and revealing the defect proneness of software systems. Work in defect estimation involves statistical approaches, capture/recapture (CR), and detection profile (DP) methods (Ebrahimi, 1997; Briand et al., 2000; Wohlin and Runeson, 1998). Data mining is usually employed for revealing associations between defects (Song et al., 2006), while defect proneness is addressed by means of metric-based classification (Khoshgoftaar and Seliya, 2003; Menzies et al., 2007; Lessmann et al., 2008).

Currently, most research work in the field of software defect prediction has been devoted to software metrics design and extraction. In Nagappan and Ball (2005), relative code churn (i.e., the relative amount of change to the system) has been proposed as an effective software feature representation. Hassan (2009) introduced the entropy of changes, a measure of the complexity of code changes, and showed it yields better results than using the absolute amounts of code change. Chidamber and Kemerer (1994) introduced the Chidamber and Kemerer (CK) metrics suite, which postulates that the presence of future defects is mostly influenced by the current design and behavior of a program. As such, in contrast to the previous approaches, it only analyzes the current system state using a variety of metrics.

On the other hand, work on the development of effective regression models for count data has been rather limited, with most approaches resorting to the naive Poisson generalized linear model (pGLM) (Nelder and Wedderburn, 1972). The popularity of pGLM is clearly due to its simple learning algorithms and high computational efficiency. However, these advantages come with significant modeling limitations: pGLM is a log-linear model, not allowing for capturing *subtle* underlying patterns and *nonlinearities* in the modeled data, or modeling data with *multiple modes*, which are quite common in real-world datasets. In addition, its maximum-likelihood treatment, only providing a point estimate of the model, does not allow for the incorporation of prior information and lacks robustness, exhibiting high proneness to yielding *biased estimates* and *overfitting*, especially when the training sample size is small (McLachlan and Peel, 2000), which is usually the case in software reliability modeling. These issues result in *reduced predictive*

* Corresponding author.
E-mail addresses: andreas.andreou@cut.ac.cy (A.S. Andreou), Soteri0s@me.com (S.P. Chatzis).

http://dx.doi.org/10.1016/j.jss.2016.09.001
0164-1212/

accuracies due to *modeling inadequacies*, undermining the very effort put into developing representative software metrics.

Bayesian inference offers a solid solution towards better handling *uncertainty*. In regression analysis of counts, however, the lack of*simple and efficient* algorithms for posterior computation has seriously limited its application to software defect prediction. Notable exceptions to this rule include the method presented in Rinsaka et al. (2006), which introduces a proportional hazards type approach and the method presented in Wiper and Bernal (2001), which extends the seminal model of Jelinski–Moranda to the case of metrics-based reliability modeling. More recently, the state-of-the-art method introduced in Torrado et al. (2013) utilized a doubly stochastic homogeneous Poisson process, where the failure log-rate parameters are imposed a Gaussian process prior.

Under a different perspective, several researchers have considered using neural networks to perform metrics-based prediction of software defect counts (Su and Huang, 2007). The popularity of neural networks in the context of software defect prediction stems in part from their simple learning algorithms, combined with the fact that their non-linear nature allows for capturing more complicated underlying data distributions compared to simple (log-)linear models. Despite these merits though, neural network-based approaches continue to suffer from low effectiveness in handling *uncertainty* and modeling distributions with*multiple modes*.

One plausible way of resolving these issues consists in replacing the deterministic units of neural network architectures with *stochasticbinary* ones. Such architectures allow for different configurations leading to different network outputs conditional on some particular network input, and are widely known as sigmoid belief networks (SBNs) (Neal, 1992). SBNs can be viewed as directed graphical models where the sigmoid function is used to compute the degrees of "belief" of a child variable given the parent nodes. Due to this formulation, SBNs prove to be extremely efficient in capturing *multi-modality*, thus resolving the drawbacks of conventional neural network approaches. In addition, their formulation under the Bayesian inference paradigm allows for handling *uncertainty* much better than conventional approaches.

A drawback of SBNs concerns their inference algorithms: Similar to many models formulated under the Bayesian paradigm, inference for SBNs is analytically intractable. As such, one has to resort to some sort of Markov chain Monte Carlo (MCMC) sampling: In Neal (1992), a Gibbs sampler is proposed that cycles through the hidden units one at a time. Even though quite straightforward, this approach is extremely *slow* when learning large models, and suffers from slow mixing of the Gibbs chain, which typically leads to *biased* parameter estimates. In Saul et al. (1996), variational Bayesian inference based on the mean-field approximation is proposed, resolving the mixing issues of Gibbs sampling. However, variational Bayes still requires cycling through the hidden units one at a time, thus imposing high computational burden. Finally, Gaussian fields are used in Barber and Sollich (1999) for inference by making Gaussian approximations to the input of the units; however, the resulting learning algorithm does not enjoy the same convergence guarantees as previous approaches.

In this paper, we resolve these problems of existing techniques, by introducing a doubly stochastic homogeneous Poisson process model, where the failure log-rate parameters at each time point are modeled in a fashion inspired from SBN formulations. Our hierarchical Bayesian model allows for efficiently capturing complex *nonlinearities* in underlying data distributions with *multiple modes*, and with significant *agility* in handling *uncertainty*. Unlike existing SBN formulations, which can model strictly binary inputs, our model is designed to allow for modeling *continuous inputs* in $\mathbb{R}$ (software metrics). These merits are expected to yield better predictive accuracies in the context of software defect prediction compared to currently used techniques.

In contrast to existing approaches, inference for our model is performed by means of particle-filtered Markov Chain Monte Carlo Expectation Maximization (PF-EM) (Snijders, 2002): We utilize sampling importance resampling (SIR) (Gordon et al., 1993) to perform inference during the E-step of the algorithm, and standard error backpropagation to perform parameter optimization during the algorithm's M-step. This way, we yield an *elegant, simple*, and computationally *efficient* learning and prediction algorithm, that yields *unbiased* model estimates and predictions, with *relatively low variance*.

We dub our model as the stochastic belief Poisson regression network (SBPRN) for software defect prediction. We evaluate our model considering both open-source object-oriented software projects, as well as proprietary benchmark systems for metrics-based software defects prediction. These experiments allow for evaluating: (i) the capacity of our model to generalize across different classes of an object-oriented software system; (ii) generalization performance across different open-source projects; (iii) model performance when dealing with proprietary systems; and (iv) model performance evolution over time (i.e., subsequent releases).

The remainder of this paper is organized as follows: In Section 2, we introduce our method, and derive its inference and learning algorithms. In Section 3, we conduct our experimental evaluations. Finally, in the concluding section, we summarize and discuss our results.

2. Proposed approach

2.1. Problem definition

In our work, we focus on the problem of modeling and predicting the recorded numbers of *software defects* y_i over *fixed time periods* of length t_i. Under such a modeling setup, it is most appropriate and computationally convenient to assume that software correction takes place only*at the end of each time period*. This assumption is particularly relevant in the case of beta software releases, where beta users report defect types and a new release is produced after some time (Torrado et al., 2013). In addition, we assume that, with the *i*th release of the software, software metrics $\boldsymbol{x}_i$ are also made available. Such metrics may reflect characteristics of the code (e.g., number of lines), measures of the amount of the correction work (e.g., number of hours), or the numbers of defects discovered in previous releases. Indeed, most modern software defect prediction methods suppose that changes in the quality of the code will be reflected in changes in the values of appropriate software metrics; thus, software metrics are used to base defect count prediction upon.

As we have discussed, while design of software metrics has been extensively studied in the literature, developing models explicitly designed for prediction of*defect counts* based on appropriate *software metrics*is a substantial component of the software *defect prediction* pipeline rather neglected by the research community. However, design of machine learning models tailored to the characteristics of *software metrics*data used for *defect count regression* is as significant for the effectiveness of a *software defect prediction* system, as it is to design representative software metrics. Therefore, the lack of advanced count regression models explicitly designed to handle the software defect prediction problem is a strong motivation for our research, with the aim to offer new innovative tools and solutions to the software engineering community.

As extensively discussed in the literature (Ruggeri and Soyer, 2008; Rinsaka et al., 2006; Wiper and Bernal, 2001; Torrado et al., 2013), observed defect counts over fixed time periods, at the end of which (imperfect) debugging may take place, can be considered to be distributed according to a piecewise homogeneous Poisson

process, i.e.

$$y_i \overset{\text{i.i.d.}}{\sim} \mathcal{P}(t_i, \lambda_i) \tag{1}$$

where λ_i is the failure rate for the ith release of the software, and is taken as a function of the observed metrics data $\boldsymbol{x}_i$. This is a simple and quite realistic assumption from the modeling point of view. On the other hand, by properly selecting the learning model used to infer the failure rates λ_i, we can obtain quite diverse methods with properties tailored to the requirements and the characteristics of the application considered each time.

In the field of *software defect prediction,*most existing approaches adopt simple log-linear failure rate models that have been also applied to a multitude of different application areas. However, it is clear to us that software defect count data and associated metrics *cannot* be adequately modeled using such simplistic assumptions. Indeed, in our setting the captured data are highly non(-log)-linear, with multiple modes, and local pattern shifts. Such artifacts *cannot* be captured by postulating (log-)linear models. Recently, Torrado et al. (2013) tried to address this problem by introducing a non-linear modeling approach based on Gaussian processes. Gaussian processes (Vanhatalo et al., 2013) are one of the most popular non-linear regression models formulated under the Bayesian inference paradigm. Despite their modeling effectiveness though, they also suffer from high computational costs, which are cubic to the number of training data points.

In this work, we tackle the considered problems by introducing a hierarchical Bayesian model for the failure rates λ_i. Our method is designed for dealing with data with multiple modes and non-linearities, as is the case with software defect prediction applications. In addition, contrary to existing Bayesian approaches, we seek a learning algorithm with simple and elegant updating equations and high computational efficiency. Finally, we aim to devise learning and inference algorithms yielding unbiased estimates with relatively low variance. Such theoretical merits are important for obtaining defect count prediction systems with high predictive accuracy.

2.2. Model formulation

Let us consider a set of input/output observation pairs $\{\boldsymbol{x}_i, y_i\}_{i=1}^N$, comprising samples from N *releases*, where $\boldsymbol{x}_i$ are the software metrics pertaining to the ith release, and y_i are the corresponding defect counts, with corresponding time durations $\{t_i\}_{i=1}^N$. We denote as λ_i the failure rate for the ith release of the software. Following the principles of hierarchical (graphical) Bayesian modeling (Jordan et al., 1998), to allow for capturing *nonlinearities*in an elegant way, we consider that the rates λ_i can be expressed as a (log-)linear function of some *latent (hidden)* variables $\boldsymbol{h}_i$; these latent variables are, in turn, driven by a non-linear model of the observed inputs (software metrics) $\boldsymbol{x}_i$. For computational efficiency, we postulate *binary*hidden variables (units), yielding

$$p(Y) = \prod_{i=1}^{N} \mathcal{P}(y_i|t_i, \lambda_i) = \prod_{i=1}^{N} \frac{\exp(-t_i\lambda_i)(t_i\lambda_i)^{y_i}}{y_i!} \tag{2}$$

where we use the notation

$$\lambda_i \triangleq \exp\left(\boldsymbol{w}^T \boldsymbol{h}_i\right) \tag{3}$$

$Y = \{y_i\}_{i=1}^N$, and we define a set of latent variables with

$$p(\boldsymbol{h}_i|\boldsymbol{x}_i) = \text{Bernoulli}(\boldsymbol{h}_i|\sigma(\boldsymbol{\Phi}\boldsymbol{x}_i + \boldsymbol{b})) \tag{4}$$

where $\sigma(\cdot)$ is the employed sigmoid function, $\boldsymbol{\Phi}$ is the synaptic weights matrix, $\boldsymbol{b}$ the bias vector, and Bernoulli$(\boldsymbol{h}_i|\boldsymbol{\xi}_i)$ denotes the Bernoulli distribution of a binary (latent) vector $\boldsymbol{h}_i$ with parameters vector $\boldsymbol{\xi}_i = \{\xi_{ij}\}_{j=1}^M$, where ξ_{ij} is the probability of the jth variable h_{ij} being equal to 1. The postulated *binary* latent vectors $\boldsymbol{h}_i = \{h_{ij}\}_{j=1}^M$ of our model can encode a large number of different values (latent states), exponential to their number M. This way, our model essentially learns to extract a high-dimensional, sparse probabilistic encoding of the extracted metrics vectors. Such encodings have been shown to be capable of inferring salient latent features of the observed data, that are more powerful for the purpose of driving a predictive model (here, a piecewise homogeneous Poisson process) compared to the original observations (extracted metrics) themselves (Saul et al., 1996; Neal, 1992; Barber and Sollich, 1999; Salakhutdinov et al., 2013).

Clearly, our devised model is based on a piecewise homogeneous Poisson process likelihood function. The input to the latter function, though, is not the observed feature vectors $\boldsymbol{x}_i$ but, rather, the stochastic latent variables of an underlying binary hidden unit stochastic neural network. Indeed, due to these characteristics, our model is reminiscent of SBNs and restricted Boltzmann machines (RBMs) (Salakhutdinov et al., 2013; Neal, 1992; Saul et al., 1996; Barber and Sollich, 1999). Two major differences between our approach and existing techniques are that: (i) Our model is explicitly formulated for performing *count regression* (while existing SBN and RBM formulations are mostly tailored to binary data and classificationtasks). (ii) Existing stochastic neural networks require expensive MCMC techniques to perform inference and model training, contrary to our approach, as we shall discuss next.

2.3. Learning algorithm

To perform learning in the context of our model, we use expectation-maximization (EM), resorting to the familiar MCMC-EM learning paradigm. MCMC-EM allows for learning the parameters of hierarchical statistical models the latent variables of which yield posterior expectations that cannot be computed in an analytical fashion (Handcock et al., 2008). MCMC-EM proceeds in an iterative fashion, with each iteration comprising an E-step and an M-step.

M-step consists in using backpropagation to optimize the objective function of the MCMC-EM algorithm over the parameters of our model. On the E-step of the algorithm, inference is performed for the model latent variables by means of some appropriate sampling algorithm. Typically, for this purpose, importance sampling (IS) is used; IS consists in drawing samples from the simple prior distribution of the latent variables, instead of their complex and analytically intractable posterior, and assigning to each sample an *importance weight* which reflects how "relevant" it is w.r.t. to the actual posterior distribution.

A caveat of this approach is that the quality of the outcome of IS heavily depends on the distance between the initialization of the model and the actual (target) distributions. If they are far apart, the samples from the initial distribution may not be able to adequately cover the target space, leading to unreliable estimates (Bartz et al., 2008). To ameliorate this problem, in this work instead of IS we resort to SIR (Gordon et al., 1993), yielding a particle filtered MCMC-EM (PF-EM) algorithm, as suggested in Asuncion et al. (2010). This way, our learning algorithm can perform rejuvenating sampling steps so as to maintain diversity within the set of drawn samples (Gilks and Berzuini, 2001).

2.3.1. E-Step

As already explained, on the E-step we compute the value of the PF-EM algorithm's objective function, which reads (Asuncion et al., 2010):

$$Q(\boldsymbol{\theta}|\boldsymbol{\theta}_{old}) \approx \frac{1}{S}\sum_{s=1}^{S}\sum_{i=1}^{N} \omega_i^{(s)} \log\, p(y_i, \boldsymbol{h}_i^{(s)}, t_i, \boldsymbol{x}_i|\boldsymbol{\theta}) \tag{5}$$

In Eq. (5), we denote as $\boldsymbol{\theta}$ the set of *trainable* model parameters, i.e. $\boldsymbol{\theta} \triangleq \{\boldsymbol{b}, \boldsymbol{\Phi}, \boldsymbol{w}\}$, S is the number of samples $\boldsymbol{h}_i^{(s)}$ drawn from the

prior distribution over the latent variables $\boldsymbol{h}_i$, i.e.

$$\boldsymbol{h}_i^{(s)} \sim \text{Bernoulli}(\boldsymbol{h}_i|\sigma(\boldsymbol{\Phi}_{old}\boldsymbol{x}_i + \boldsymbol{b}_{old})) \tag{6}$$

and $\omega_i^{(s)}$ are the*importance weights* of the drawn samples, yielding

$$\omega_i^{(s)} = \frac{p(y_i|\boldsymbol{h}_i^{(s)}, t_i, \boldsymbol{x}_i; \boldsymbol{\theta}_{old})}{p(y_i|\boldsymbol{x}_i, t_i; \boldsymbol{\theta}_{old})} \tag{7}$$

which can be approximated by:

$$\omega_i^{(s)} \approx \frac{p(y_i|\boldsymbol{h}_i^{(s)}, t_i, \boldsymbol{x}_i; \boldsymbol{\theta}_{old})}{\frac{1}{S}\sum_{s=1}^{S} p(y_i|\boldsymbol{h}_i^{(s)}, t_i, \boldsymbol{x}_i; \boldsymbol{\theta}_{old})} \tag{8}$$

where

$$p(y_i|\boldsymbol{h}_i, t_i, \boldsymbol{x}_i; \boldsymbol{\theta}) = p(y_i|\boldsymbol{h}_i, t_i; \boldsymbol{\theta})\ p(\boldsymbol{h}_i|\boldsymbol{x}_i; \boldsymbol{\theta}) \tag{9}$$

$p(y_i|\boldsymbol{h}_i,\ t_i;\ \boldsymbol{\theta})$ is given by (2) and (3), and $p(\boldsymbol{h}_i|\boldsymbol{x}_i;\ \boldsymbol{\theta})$ is given by (4). The dimensionality of $\boldsymbol{h}_i$, i.e. the number of hidden units M, is *heuristically* determined, e.g. by means of cross-validation, as usual in the literature of neural networks.

Based on the above presentation, we observe that the E-step of our algorithm eventually boils down to computing the updates of the importance weights $\omega_i^{(s)}$ of the drawn samples (*particles*) of the latent variables, $\boldsymbol{h}_i^{(s)}$. Following the relevant derivations of Asuncion et al. (2010), to perform this procedure, we initially draw samples of the $\boldsymbol{h}_i$ from their prior (6), on the first iteration of our PF-EM algorithm, and compute their importance weights using (8). Further, on subsequent algorithm iterations, each particle weight $\omega_i^{(s)}$ is updated under the standard SIR scheme; for instance, on the kth iteration, we have

$$\omega_i^{(s)} \leftarrow \omega_i^{(s)} \frac{p(y_i|\boldsymbol{h}_i^{(s)}, t_i, \boldsymbol{x}_i; \boldsymbol{\theta}_k)}{p(y_i|\boldsymbol{h}_i^{(s)}, t_i, \boldsymbol{x}_i; \boldsymbol{\theta}_{k-1})} \tag{10}$$

where $\boldsymbol{\theta}_k$ are the model parameter values on the kth iteration.

In addition, an important aspect of the SIR scheme consists in monitoring the quality of the particle set. For this purpose, after updating the importance weights, the E-step of our algorithm proceeds to compute the effective sample size (ESS), which reveals the number of random samples needed to match the Monte Carlo variation of the particle set. It is given by Kong et al. (1994)

$$\text{ESS}\left(\{\omega_i^{(s)}\}_{s=1}^{S}\right) = \frac{\left(\sum_s \omega_i^{(s)}\right)^2}{\sum_s \left(\omega_i^{(s)}\right)^2} \tag{11}$$

ESS values below a predetermined threshold, M_{th}, suggest that the importance weights have become degenerate; in such cases, resampling and rejuvenation are necessary to replenish the diversity of the particles. Resampling is performed by sampling S particles from the set $\{\boldsymbol{h}_i^{(s)}\}_{s=1}^{S}$ with replacement, with probabilities proportional to $\{\omega_i^{(s)}\}_{s=1}^{S}$. After resampling, the importance weights are reset to 1. Rejuvenation is performed by sampling new particles from the prior distribution (6) using the current updates of the model parameters $\boldsymbol{\theta}$. Rejuvenation increases the diversity of samples but can be computationally expensive. To mitigate this cost, we restrict ourselves to conducting rejuvenation only every M_{iter} PF-EM algorithm iterations. In our experiments, we set $M_{\text{iter}} = 100$ and $M_{\text{th}} = 0.9 \times S$, as suggested in Asuncion et al. (2010).

2.3.2. M-Step

Once the E-step is completed, the learning algorithm of our model proceeds to its M-step. On each iteration, the M-step of the devised PF-EM algorithm comprises maximization of $Q(\boldsymbol{\theta}|\boldsymbol{\theta}_{old})$ over the parameters $\boldsymbol{\theta}$ by means of gradient ascent. We have

$$\frac{\partial Q(\boldsymbol{\theta}|\boldsymbol{\theta}_{old})}{\partial \boldsymbol{\theta}} \approx \frac{1}{S}\sum_{s=1}^{S}\sum_{i=1}^{N} \omega_i^{(s)} \frac{\partial}{\partial \boldsymbol{\theta}}\left\{\log p(y_i, \boldsymbol{h}_i^{(s)}, t_i, \boldsymbol{x}_i|\boldsymbol{\theta})\right\} \tag{12}$$

where

$$\begin{aligned}&\frac{\partial}{\partial \boldsymbol{\theta}}\left\{\log p(y_i, \boldsymbol{h}_i^{(s)}, t_i, \boldsymbol{x}_i|\boldsymbol{\theta})\right\} \\ &= \frac{\partial}{\partial \boldsymbol{\theta}}\left\{\log p(y_i|\boldsymbol{h}_i^{(s)}, t_i; \boldsymbol{\theta}) + \log p(\boldsymbol{h}_i^{(s)}|\boldsymbol{x}_i; \boldsymbol{\theta})\right\}\end{aligned} \tag{13}$$

As we observe, once the above mentioned gradients are computed for each one of the drawn samples $\boldsymbol{h}_i^{(s)}\ \forall s$, a weighted summation over the drawn samples is performed, where the weighting comes from the importance weights $\omega_i^{(s)}$. Interestingly, this derived optimization scheme matches our intuition, since learning adjusts the contribution of the drawn samples on the basis of the corresponding importance weights, which provide a measure of model fit to the training data. Maximization of this expression can be performed in a straightforward manner, by application of backpropagation.

A summary of the learning algorithm of our model is provided in Algorithm 1. A drawback of our model is the need of drawing S samples from the stochastic units. In fact, the requirement of sampling is common among hierarchical Bayesian models capable of structured learning. Nevertheless, we have observed that our method requires a very *limited* number of samples to perform well (see also Section 3); as such, the computational costs of sampling are greatly reduced. In addition, we underline that our method relies on SIR, thus yielding *unbiased* parameter estimates with *low variance* (Gordon et al., 1993). This is considerably advantageous for the learning algorithm, and allows for yielding models with better generalization accuracies.

Algorithm 1 PF-EM training algorithm for SBPRN.

- Select the number of hidden units M.
- Set the threshold values to $M_{\text{iter}} = 100$ and $M_{\text{th}} = 0.9 \times S$.
- Initialize the model parameters $\boldsymbol{\theta}$.
- Draw S samples of the hidden unit values $\boldsymbol{h}_i$ (corresponding to each one of the available training data points, $\boldsymbol{x}_i$), by using (6).
- Initialize the importance weights of these samples, setting $\omega_i^{(s)} = 1, \forall i, s$.
- For *MAXITER* number of iterations, or until convergence of $Q(\boldsymbol{\theta}|\boldsymbol{\theta}_{old})$, **do:**

1. *M-Step:* Update the model parameters $\boldsymbol{\theta}$ by means of backpropagation, based on (12)-(13).
2. *E-Step:* Update the importance weights $\omega_i^{(s)} = 1, \forall i, s$, using (10) and (9).
3. *E-Step:* Compute the ESS of the drawn $\boldsymbol{h}_i$ samples. If it is below threshold M_{th}, **then**:

- If M_{iter} algorithm iterations have elapsed fromthe previous sample *rejuvenation*, draw a new set of S samples from the prior distributions (6), and discard the old ones.
- Otherwise, *resample* S particles from the set $\{\boldsymbol{h}_i^{(s)}\}_{s=1}^{S}, \forall i$, with replacement, with probabilities proportional to $\{\omega_i^{(s)}\}_{s=1}^{S}$.
- Reset the importance weights of the samples, setting $\omega_i^{(s)} = 1, \forall i, s$.

2.4. Model augmentation with deterministic units

Despite allowing for computational efficiency, a drawback of building our *defect count regression* model with *binary* stochastic units concerns its capability of learning smooth predictive distributions $p(y|\boldsymbol{x}, t)$. Specifically, according to Neal (1992), the capacity of a model employing *only binary* stochastic units to predict output variables y taking *unbounded* values in $\mathbb{R}$ may be rather limited. To

resolve these issues, in addition to the stochastic units $\boldsymbol{h}$ of our model, we also introduce a *limited number* of auxiliary *deterministic* units $\boldsymbol{h}^{\text{det}}$. This is a novel modeling assumption that sets our work apart from the related literature, and is designed to specifically account for the requirements of our *count regression*problem of defect prediction in software systems. We have:

$$\boldsymbol{h}_i^{\text{det}} = \sigma\left(\boldsymbol{\Phi}^{\text{det}}\boldsymbol{x}_i + \boldsymbol{b}^{\text{det}}\right) \tag{14}$$

Learning under such a setup proceeds similar to the algorithm prescribed above, with $Q(\boldsymbol{\theta}|\boldsymbol{\theta}_{old})$ now reading

$$\frac{\partial Q(\boldsymbol{\theta}|\boldsymbol{\theta}_{old})}{\partial \boldsymbol{\theta}} \approx \frac{1}{S}\sum_{s=1}^{S}\sum_{i=1}^{N}\omega_i^{(s)}\frac{\partial}{\partial \boldsymbol{\theta}}\Big\{\log p(y_i|[\boldsymbol{h}_i^{(s)};\boldsymbol{h}_i^{\text{det}}], t_i;\boldsymbol{\theta}) + \log p([\boldsymbol{h}_i^{(s)};\boldsymbol{h}_i^{\text{det}}]|\boldsymbol{x}_i;\boldsymbol{\theta})\Big\} \tag{15}$$

where, we now have $\boldsymbol{\theta} \triangleq \{\boldsymbol{b}, \boldsymbol{\Phi}, \boldsymbol{b}^{\text{det}}, \boldsymbol{\Phi}^{\text{det}}, \boldsymbol{w}\}$, it holds

$$\frac{\partial}{\partial \boldsymbol{\theta}}\Big\{\log p(y_i, [\boldsymbol{h}_i^{(s)};\boldsymbol{h}_i^{\text{det}}], t_i, \boldsymbol{x}_i|\boldsymbol{\theta})\Big\} = \frac{\partial}{\partial \boldsymbol{\theta}}\Big\{\log p(y_i|[\boldsymbol{h}_i^{(s)};\boldsymbol{h}_i^{\text{det}}], t_i;\boldsymbol{\theta}) + \log p([\boldsymbol{h}_i^{(s)};\boldsymbol{h}_i^{\text{det}}]|\boldsymbol{x}_i;\boldsymbol{\theta})\Big\} \tag{16}$$

and $[\boldsymbol{\alpha};\boldsymbol{\gamma}]$ denotes the concatenation of vectors $\boldsymbol{\alpha}$ and $\boldsymbol{\gamma}$. Parameter optimization is performed by means of backpropagation, similar to previously (Section 2.3).

2.5. Prediction generation

Once the PF-EM learning algorithm of our SBPRN model converges, we can use it to perform defect count prediction for future releases, based on some software metrics measurements $\boldsymbol{x}$. For this to happen, we need to compute the predictive distribution

$$p(y|\boldsymbol{x}, t) \approx \frac{1}{S}\sum_{s=1}^{S} p(y|[\boldsymbol{h}^{(s)};\boldsymbol{h}^{\text{det}}], t) \tag{17}$$

where *the samples of the latent units* $\boldsymbol{h}^{(s)}$ are drawn from (6), and we have

$$\boldsymbol{h}^{\text{det}} = \sigma\left(\boldsymbol{\Phi}^{\text{det}}\boldsymbol{x} + \boldsymbol{b}^{\text{det}}\right) \tag{18}$$

Note that our Monte Carlo estimate of the predictive distribution in (17) is *unbiased*, similar to the case of our learning algorithm. In addition, it has *relatively low variance*, since it does not depend on the dimensionality of the latent units vector $\boldsymbol{h}$ (i.e., the selected number of hidden units). These *theoretical properties of our model offer significant advantages to the software practitioners,* as they are expected to result in better predictive accuracies.

Having obtained an (approximate) expression for the predictive distribution of our model, we can now devise a simple approach for generating defect count *predictions*. Specifically, in our work we use the expectation of the predictive distribution of our model. This way, we yield predictions under the following approximate expression

$$\hat{y} = \mathbb{E}[p(y|\boldsymbol{x}, t)] \approx t\,\exp\left(\frac{1}{S}\sum_{s=1}^{S}\boldsymbol{w}^T[\boldsymbol{h}^{(s)};\boldsymbol{h}^{\text{det}}]\right) \tag{19}$$

3. Experiments

3.1. Evaluation with open-source object-oriented software systems

We evaluate our approach considering three *open-source object-oriented* software systems, namely Eclipse JDT Core, Equinox framework, and Mylyn. To this end, we utilize data from the *Bug Prediction Dataset (BPD)*[1] (D'Ambros et al., 2010), which is a collection of models and metrics of software systems and their histories. BPD is explicitly designed for facilitating the purpose of comparing different bug prediction approaches, and evaluating whether a new technique is an improvement over existing ones. It provides a variety of source code metrics and/or historical measures pertaining to system releases of the considered software systems, as well as the related numbers of post-release defects reported in bug tracking systems. BPD is designed for performing bug prediction at the class level, driven by the extracted software metrics; hence, it is ideal for the purposes of our experimental evaluations.

An outline of the characteristics of the software systems included in BPD that we use in our experiments is provided in Table 1. Specifically, the software metrics provided by the authors of D'Ambros et al. (2010) have been computed using statistics extracted from the considered software projects *up to the last system release* in the examined release periods. On this basis, defect prediction concerns the *last project release*, and is performed on the *class-level*, rather than package- or subsystem-level defect prediction. We use post-release defects for validation to emulate a real-world scenario. Following D'Ambros et al. (2010), we consider different combinations of the software metrics provided therein as the used *independent variables* (regressors $\boldsymbol{x}$) of the evaluated models, which comprise the following:

Change metrics. These are sets of file-level change metrics, extracted as suggested by Moser et al. (2008). Specifically, we initially consider the case of using *all the metrics* suggested in Moser et al. (2008), in the form of a single observed vector, henceforth referred to as the **MOSER** setup *(this corresponds to all columns containing metrics in files "change-metrics.csv" of the BPD dataset)*. In addition, we consider using only the cumulative number of defects fixed until release, henceforth referred to as **NFIX**, the number of revisions (versions until release), henceforth referred to as **NR**, as well as the combination of NFIX and NR into one observed vector, henceforth referred to as **NFIX+NR**. In addition, we also consider the **BUG-CATEGORIES** setup, which uses as metrics the numbers of previously reported bugs belonging to five categories, namely all bugs, non-trivial bugs, major bugs, critical bugs (critical or blocker severity), and high priority bugs *(these metrics correspond to similarly named columns of the "bug-metrics.csv" files in the BPD dataset)*.

Source code metrics. These features are based on the CK metrics suite (Chidamber and Kemerer, 1994), with the optional addition of some further object-oriented (OO) metrics. The latter metrics comprise: number of other classes that reference the class, number of other classes referenced by the class, number of attributes, number of public attributes, number of private attributes, number of attributes inherited, number of lines of code, number of methods, number of public methods, number of private methods, and number of methods inherited. In our experiments, we consider using only the CK metrics suite, henceforth referred to as the **CK** setup, using only our object-oriented metrics, henceforth referred to as **OO**, using both of them, henceforth referred to as **CK+OO**, as well as using only the number of code lines as our metric, henceforth referred to as **LOC** (*the metrics used in all these setups correspond to similarly named columns of the "single-version-ck-oo.csv" files in the BPD dataset*).

Entropy of changes metrics. These features are based on measuring how distributed changes are in a system over a time interval. This is effected by computing the Shannon entropy of code change, following the approach presented in Hassan (2009). The more spread the changes are, the higher is the complexity. The intuition is that one change affecting one file only is simpler than one affecting many different files, as the developer who has to perform the change has to keep track of all of them. We refer to this experimental setup as **HCM**. We also consider the variants of HCM suggested by Hassan (2009), namely **WHCM**, which weighs

[1] http://bug.inf.usi.ch.

Table 1
Summary of characteristics of the considered benchmark systems.

System	Eclipse JDT core	Equinox framework	Mylyn
Prediction release	3.4	3.4	3.4.1
Time period	1.1.2005–6.17.2008	1.1.2005–6.25.2008	1.17.2005–3.17.2009
#Classes	997	439	2,196
#Versions	91	91	98
#Transactions	9135	1616	9,189
#Post-release defects	463	279	677

the entropy of the system with the probability of each file being modified, **EDHCM**, which exponentially reduces the contribution of the entropies for earlier periods of time, **LDHCM**, which linearly reduces the contribution of the entropies for earlier periods of time, and **LGDHCM**, which logarithmically reduces the contribution of the entropies for earlier periods of time *(the sets of metrics pertaining to each one of these setups comprise the full set of columns of the corresponding .csv files inside the folders "entropy" of the BPD dataset)*.

To perform model training, we randomly select 90% of the classes of each project, and use the above described software metrics combinations and the corresponding number of defects (pertaining to the last project release) to perform training of our model. We then evaluate our trained model on the remaining 10% of the classes of each system, thus allowing for evaluating model *generalization* capacities *across classes*. We repeat this experiment 10 times, with different random splits of the classes into training and test sets, to ameliorate the effect of random sample selection on the observed model performances. Similar to D'Ambros et al. (2010), performance evaluation is conducted on the basis of the obtained Spearman' s correlation coefficients between the predicted and the actual numbers of defects of the classes in our test sets. The Spearman correlation is computed on two lists (classes ordered by actual number of bugs and classes ordered by number of predicted bugs) and is an indicator of the similarity of their order. We measure the correlation with the Spearman coefficient (instead of, for example, the Pearson coefficient), as it is recommended with skewed data (which is the case here, as most classes have no bugs).

To obtain some comparative results, we also evaluate the baseline pGLM model (Nelder and Wedderburn, 1972), which is used as a key-component of a number of classical software defect prediction approaches, a recently proposed, state-of-the-art Bayesian inference approach, namely the Gaussian process-based homogeneous Poisson process model (Torrado et al., 2013) (henceforth denoted as GP), and a standard feedforward neural network-based modeling approach (henceforth referred to as NN (Su and Huang, 2007)). For both our approach and the NN method, the number of employed units ("neurons") is determined heuristically, by means of cross-validation. Turning to the sampling schemes of our method, we have experimented with different selections of the number of drawn samples S used to perform model training. We have observed that values of S greater than 30 offer negligible performance improvement compared to the incurred increase in computational costs. For this reason, in the following, we report results with the number of samples set to $S = 30$.

3.1.1. Evaluation using change metrics

In Table 2a–e, we depict the obtained performance of the evaluated algorithms when using *change metrics* as the independent variables for model training and prediction generation. These results are means and standard deviations of the obtained Spearman's correlation values over the conducted 10 experiment repetitions. In these tables, we type in *emphasis* the performances of competitors of our model that are deemed statistically comparable to our approach, based on the Wilcoxon signed-rank test. As we

Table 2
Change metrics-based evaluation: Obtained performance of the evaluated methods: (a) MOSER setup. (b) NFIX setup. (c) NR setup. (d) NFIX+NR setup. (e) BUG-CATEGORIES setup.

(a)

System	pGLM	GP	NN	SBPRN
Eclipse JDT Core	0.36 (0.07)	0.41 (0.06)	0.32 (0.07)	**0.54 (0.07)**
Equinox framework	0.56 (0.08)	0.60 (0.06)	0.58 (0.12)	**0.71 (0.04)**
Mylyn	0.35 (0.02)	0.37 (0.03)	0.31 (0.06)	**0.40 (0.02)**

(b)

System	pGLM	GP	NN	SBPRN
Eclipse JDT Core	0.24 (0.05)	0.24 (0.05)	0.16 (0.10)	**0.34 (0.09)**
Equinox framework	0.35 (0.11)	0.36 (0.11)	0.29 (0.12)	**0.42 (0.10)**
Mylyn	0.13 (0.07)	0.13 (0.07)	0.10 (0.06)	**0.16 (0.04)**

(c)

System	pGLM	GP	NN	SBPRN
Eclipse JDT Core	0.35 (0.17)	0.40 (0.17)	0.37 (0.13)	**0.48 (0.10)**
Equinox framework	0.56 (0.05)	0.57 (0.04)	0.49 (0.12)	**0.66 (0.10)**
Mylyn	0.09 (0.03)	0.19 (0.05)	0.16 (0.05)	**0.24 (0.04)**

(d)

System	pGLM	GP	NN	SBPRN
Eclipse JDT Core	0.35 (0.13)	0.35 (0.15)	0.35 (0.18)	**0.42 (0.10)**
Equinox framework	0.58 (0.18)	0.64 (0.17)	0.62 (0.17)	**0.68 (0.14)**
Mylyn	0.14 (0.05)	0.18 (0.08)	0.16 (0.06)	**0.26 (0.03)**

(e)

System	pGLM	GP	NN	SBPRN
Eclipse JDT Core	0.41 (0.06)	0.42 (0.06)	0.36 (0.09)	**0.47 (0.08)**
Equinox framework	*0.61 (0.11)*	*0.62 (0.11)*	0.54 (0.12)	**0.68 (0.10)**
Mylyn	*0.14 (0.04)*	*0.15 (0.04)*	0.08 (0.11)	**0.17 (0.04)**

Note: Best obtained performance is typed in bold.

observe, performance of all methods fluctuates considerably when trained with different metrics combinations. For example, it appears that MOSER consistently yields (one of) the top results for all methods and datasets, while NFIX turns out to be the worst performer.

In addition, we observe that all methods seem to yield consistently higher performances in the case of the Equinox project, while the lower average performance for all methods is observed in the case of the Mylyn project. We believe this consistent behavior across all the considered metrics combinations is related to the difficulty of these projects. Indeed, Mylyn is the biggest project of all, comprising more than a thousand classes; as such, the across classes generalization problem becomes much harder, resulting in lower performance for all methods. On the other hand, Equinox is the smallest, thus easiest, of the considered projects, and should be expected to allow for obtaining higher performances (as it actually happens).

Finally, we observe that our method consistently yields better performance than the competition. Note also that GP performance is only slightly better than pGLM; this finding indicates that our approach is much more competent in modeling the peculiarities of software metrics data compared to existing state-of-the-art approaches (as *theoretically* expected).

3.1.2. Evaluation using source code metrics

In Table 3a–d, we depict the obtained performance of the evaluated algorithms (Spearman's correlation means and standard deviations over the conducted 10 folds) when using *source code*

Table 3
Source code metrics-based evaluation: Obtained performance of the evaluated methods: (a) CK setup. (b) OO setup. (c) CK+OO setup. (d) LOC setup.

System	pGLM	GP	NN	SBPRN
Eclipse JDT Core	0.38 (0.06)	0.40 (0.09)	0.38 (0.09)	**0.48 (0.04)**
Equinox framework	0.53 (0.19)	0.53 (0.18)	0.46 (0.23)	**0.60 (0.04)**
Mylyn	*0.25 (0.10)*	*0.26 (0.11)*	0.16 (0.10)	**0.30 (0.08)**
		(a)		
System	pGLM	GP	NN	SBPRN
Eclipse JDT Core	0.34 (0.06)	0.47 (0.07)	0.34 (0.04)	**0.52 (0.06)**
Equinox framework	0.60 (0.15)	0.60 (0.12)	0.58 (0.09)	**0.67 (0.06)**
Mylyn	0.26 (0.03)	*0.33 (0.05)*	0.28 (0.07)	**0.37 (0.05)**
		(b)		
System	pGLM	GP	NN	SBPRN
Eclipse JDT Core	0.41 (0.05)	0.52 (0.04)	0.36 (0.09)	**0.60 (0.04)**
Equinox framework	0.46 (0.16)	0.49 (0.16)	0.45 (0.22)	**0.58 (0.07)**
Mylyn	0.25 (0.02)	*0.31 (0.05)*	0.23 (0.08)	**0.33 (0.05)**
		(c)		
System	pGLM	GP	NN	SBPRN
Eclipse JDT Core	0.39 (0.08)	0.40 (0.06)	0.35 (0.08)	**0.49 (0.03)**
Equinox framework	0.53 (0.11)	0.55 (0.13)	0.47 (0.16)	**0.60 (0.09)**
Mylyn	*0.20 (0.07)*	*0.20 (0.05)*	0.15 (0.07)	**0.24 (0.05)**
		(d)		

Note: Best obtained performance is typed in bold.

Table 4
Change metrics entropy-based evaluation: Obtained performance of the evaluated methods: (a) HCM setup. (b) WHCM setup. (c) EDHCM setup. (d) LDHCM setup. (e) LGDHCM setup.

(a)				
System	pGLM	GP	NN	SBPRN
Eclipse JDT Core	0.41 (0.10)	0.44 (0.10)	0.31 (0.10)	**0.51 (0.08)**
Equinox framework	0.56 (0.08)	0.56 (0.08)	0.45 (0.12)	**0.60 (0.06)**
Mylyn	0.06 (0.07)	*0.21 (0.04)*	0.13 (0.05)	**0.28 (0.06)**
		(b)		
System	pGLM	GP	NN	SBPRN
Eclipse JDT Core	0.45 (0.08)	0.45 (0.08)	0.43 (0.07)	**0.48 (0.08)**
Equinox framework	*0.66 (0.04)*	**0.67 (0.02)**	0.56 (0.12)	**0.75 (0.08)**
Mylyn	0.06 (0.04)	0.16 (0.06)	0.11 (0.06)	**0.22 (0.07)**
		(c)		
System	pGLM	GP	NN	SBPRN
Eclipse JDT Core	0.39 (0.10)	***0.47 (0.09)***	0.38 (0.10)	**0.47 (0.06)**
Equinox framework	0.55 (0.12)	0.56 (0.11)	0.41 (0.18)	**0.67 (0.03)**
Mylyn	*0.19 (0.07)*	*0.24 (0.08)*	0.08 (0.04)	**0.26 (0.04)**
		(d)		
System	pGLM	GP	NN	SBPRN
Eclipse JDT Core	0.46 (0.09)	0.51 (0.08)	0.37 (0.07)	**0.54 (0.06)**
Equinox framework	0.56 (0.15)	0.58 (0.15)	0.41 (0.13)	**0.63 (0.07)**
Mylyn	0.08 (0.10)	***0.25 (0.04)***	0.11 (0.05)	**0.25 (0.03)**
		(e)		
System	pGLM	GP	NN	SBPRN
Eclipse JDT Core	0.45 (0.09)	***0.50 (0.09)***	0.43 (0.09)	**0.50 (0.08)**
Equinox framework	0.55 (0.08)	0.56 (0.08)	0.54 (0.08)	**0.68 (0.06)**
Mylyn	0.19 (0.06)	*0.25 (0.11)*	0.17 (0.10)	**0.30 (0.08)**

Note: Best obtained performance is typed in bold.

metrics as the independent variables for model training and prediction generation. We type in *emphasis* the performances of competitors of our model that are deemed statistically comparable to our approach, based on the Wilcoxon signed-rank test. We observe that LOC is the worst performing metrics selection in all cases. This is expectable, since LOC utilizes only one software metric, while all the rest are combinations of several metrics.

Another characteristic finding is that lower average performance over all methods for some metrics/project combination is typically combined with lower differences between the performance of our approach and the considered alternatives. This observation is quite interesting, since it matches our intuition that the obtainable performance gains stemming from devising better modeling principles are much more limited when dealing with harder problems than in cases of easier problems. For example, observe how performance differences between our method and the competition vary among the Equinox and Mylyn projects: Equinox is the smallest, and, thus, easiest project, where the performance differences between our method and the competition are quite high, while Mylyn is the largest and hardest one, resulting in smaller differences between our method and the competition. Finally, we observe that our method consistently yields better performance than the competition, while GP usually performs similar to pGLM.

3.1.3. Evaluation using entropy of changes metrics

In Table 4a–e, we depict the obtained performance of the evaluated algorithms (Spearman' s correlation means and standard deviations over the conducted 10 folds) when using *entropy of changes metrics* as the independent variables for model training and prediction generation. A major difference between these results and our previous empirical observations is that GP now yields significantly better comparative performance results w.r.t. pGLM. It is interesting though that this behavior is mostly due to a significant drop in pGLM performance rather than a GP performance improvement. We believe these results are due to the highly non-(log-)linear nature of the used metrics data (e.g., cumulative changes data and some non-linear functions thereof), which render the (log-)linear assumptions of pGLM highly irrelevant.

3.2. Generalization across open-source object-oriented software systems

In a further set of experiments, we evaluate the generalization capacity of our model *across* the previous *projects*: In other words, we train our model using samples (selected as in the previous experiments) from *one*of the available software systems, and we evaluate its performance on the available data pertaining to the classes of the *remainder* of the considered software systems. For comparative purposes, we also evaluate pGLM, GP, and NN under the same experimental setup. Model size selection is performed similar to the previous experiments; the same holds for the number of drawn samples S, setting $S = 30$. To allow for examining how model generalization performance varies with the number of classes of the target software systems, we opt for conducting training on the Eclipse project, and using the obtained models to perform predictions on the Equinox project, which comprises a relatively low number of classes, and on the Mylyn project, which comprises a relatively high number of classes. The obtained results (Spearman' s correlation values) are reported in Table 5a and b.

These results lead to some interesting observations: (i) It appears that generalization performance for all methods is significantly better when the target project contains a low number of classes compared to the cases dealing with large target projects. (ii) In both the cases of the Equinox and Mylyn projects, it appears that some metrics combinations consistently yield inferior generalization performance for all the evaluated methods, while some other combinations consistently yield optimal generalization performances for all the evaluated methods. Characteristic such cases are the NFIX, HCM, EDHCM, and LGDHCM metrics combinations, and the CK and OO combinations, respectively. (iii) Turning to the comparative results between our method and the competition, we observe that in cases of lower performance metrics combinations our method yields a clear generalization performance advantage over the competition, retaining performance similar to the best-performing metrics; for example, characteristic such cases are the EDHCM and LGDHCM metrics combinations for both projects. On the other hand, in the cases of the best-performing metrics, it

Table 5
Generalization performance on the target open-source object-oriented software systems:(a) Equinox. (b) Mylyn.

(a)

Metrics	pGLM	GP	NN	SBPRN
MOSER	0.50	0.51	0.48	**0.52**
NFIX	**0.46**	**0.46**	0.43	**0.46**
NR	0.52	**0.54**	0.45	**0.54**
NFIX+NR	0.53	**0.54**	0.47	**0.54**
BUG-CATEGORIES	0.54	**0.55**	0.39	**0.55**
CK	0.51	**0.52**	0.48	**0.52**
OO	0.50	**0.52**	0.49	**0.52**
CK+OO	0.44	**0.45**	0.42	**0.45**
LOC	0.45	**0.48**	0.44	**0.48**
HCM	0.24	0.27	0.32	**0.34**
WHCM	**0.55**	**0.55**	0.48	**0.55**
EDHCM	0.40	0.40	0.36	**0.48**
LDHCM	0.44	0.48	0.46	**0.49**
LGDHCM	0.45	0.57	0.44	**0.58**

(b)

Metrics	pGLM	GP	NN	SBPRN
MOSER	0.15	**0.16**	0.15	**0.16**
NFIX	0.14	**0.14**	0.13	**0.14**
NR	0.10	0.17	0.13	**0.18**
NFIX+NR	0.13	**0.18**	0.15	**0.18**
BUG-CATEGORIES	0.12	0.12	0.09	**0.16**
CK	0.21	**0.21**	0.21	**0.21**
OO	0.21	**0.22**	0.21	**0.22**
CK+OO	0.18	**0.20**	0.19	**0.20**
LOC	0.21	0.22	0.15	**0.24**
HCM	0.10	0.11	0.07	**0.14**
WHCM	0.07	**0.14**	0.07	**0.14**
EDHCM	0.12	0.13	0.09	**0.19**
LDHCM	0.06	0.10	0.06	**0.15**
LGDHCM	0.12	0.14	0.10	**0.19**

Note: Best obtained performance is typed in bold.

appears that all approaches yield similar results (see, e.g. the cases of the CK and OO metrics combinations); this is especially true in the case of the Mylyn project.

These results corroborate that design of appropriate count regression algorithms, tailored to the characteristics of the defect prediction problem, is of major significance for the successful performance of the whole defect prediction pipeline. It also appears that for each target project there must be some kind of generalization performance plateau for all methods, especially in the case of the Mylyn project. We believe this phenomenon has mostly to do with the intrinsic difficulty of the generalization problem at hand, rather than the properties of the evaluated methods and their inadequacies thereof. For example, this plateau is much lower in the case of the Mylyn project, which comprises a large number of classes, than in the case of the much smaller Equinox project. Even more characteristically, we observe that in the case of the Mylyn project this plateau is similar for all the evaluated methods, while in the case of the Equinox project, methods based on Bayesian inference and capable of capturing multiple modes in the modeled data (namely, GP and our method) yield better performance suprema compared to methods not possessing such capacities (namely, NN and pGLM). Finally, a comparison of the results in Tables 1–4 and 5(a) and (b) seems to support Nagappan et al. (2006), who have concluded that models are in general more accurate when trained on the same or similar systems.

3.3. Performance evolution evaluation with proprietary benchmark systems

Finally, here we evaluate our method using a dataset pertaining to a proprietary benchmark system, with the aim to examine how model performance changes over time, i.e. over consecutive time periods. For this purpose, we use the dataset presented in Dalal and McIntosh (1994). This dataset records the values of the new or changed non-commentary source lines (NCNCSL), the staff time spent testing, and the number of detected faults over a set of consecutive releases of a proprietary system. The total number of observed failures was 870.

Table 6
Performance evolution evaluation.

Iteration	pGLM	GP	NN	SBPRN
Last 50 - #1	0.41	0.45	0.43	**0.46**
Last 50 - #2	0.46	**0.46**	0.44	**0.46**
Last 50 - #3	0.40	0.44	0.44	**0.47**
Last 50 - #4	0.38	0.41	0.47	**0.53**
All history - #1	0.42	**0.48**	0.40	**0.48**
All history - #2	0.44	**0.46**	0.39	**0.46**
All history - #3	0.40	0.40	0.39	**0.42**
All history - #4	0.42	0.48	0.46	**0.54**

Note: Best obtained performance is typed in bold.

We examine two different evaluation scenarios. In the first scenario, our model is trained with data from the last 50 releases and is evaluated on the rest of the available (future) data. We retrain the evaluated models every 20 releases, and report the so-obtained Spearman' s correlation coefficients. Under the second scenario, model retraining is performed every 20 releases, using *all*the available past data (starting from the 50th release).

In Table 6, we illustrate our results. It is clear that pGLM yields the worst results, with GP and NN yielding comparably good results, and our method appearing to enjoy noticeable performance advantages over the competition. We also notice that retraining using *all* the past samples does not seem to yield significantly improved performance compared to retaining only the last 50 samples. This finding suggests that there are no discernible repeating failure patterns in the dataset to permit capturing defect behavior when all past history is taken into account.

3.4. Discussion

This research was initiated with the aim to better address the challenges related with software defect prediction. Defect counts prediction is one of the most significant problems in the field of software engineering, especially when it comes to the development of safety-critical systems, or systems with limited defect resilience. In our experimental investigations, we evaluated our method from multiple points of view; specifically, we examined: (i) the capacity of our model to generalize across different classes of an object-oriented software system; (ii) generalization performance across different open-source projects; (iii) model performance when dealing with proprietary systems; and (iv) model performance evolution over time (i.e., subsequent releases). Some of the major observations of our experiments are the following: (i) pGLMs and simple NNs (Su and Huang, 2007) seem to yield the lowest performance for all projects, when it comes to generalization across classes of object-oriented projects, generalization across different projects, as well as performance evolution evaluation over consecutive releases. (ii) All methods yield better performances when dealing with smaller object-oriented projects than in cases of larger ones. (iii) The performance improvements obtained by our method increase with the mean performance of all the evaluated methods. This indicates that some systems are inherently harder than others to perform prediction, irrespectively of the used count regression method or software metrics. (iv) In all the considered experimental scenarios, our method yielded the best observed performance among all the evaluated methods.

To contemplate the contribution of our findings to the general area of software engineering, we resort to the systematic

literature review on fault prediction performance in software engineering proposed by Hall et al. (2012). Therein, the authors propose the use of several research questions for assessing and analyzing models used to predict faults in the source code. For our purposes, we specifically focus on two of these proposed questions.

The first one investigates how context affects fault prediction, focusing on the source of the data and the maturity of the systems used for experimentation, the application domain and the size. Although Hall et al. (2012) pointed out that, in general, the effect of these parameters on the efficiency of a fault prediction model is yet not very clear, the results of our experiments are in agreement with studies suggesting that larger systems in terms, e.g. of classes and/or post release defects, present a less predictable failure behavior compared to smaller systems: The Equinox system, which is the smallest among the three systems used during the experimental process, presents consistently better fault prediction with the proposed model in nearly all of the metrics and methods evaluated, while Mylyn, which is the largest system, is consistently the hardest to predict. Therefore, these results are in perfect alignment with studies that support that size influences predictive performance more than system maturity, the latter being roughly measured by the number of released versions (Hall et al., 2012). The significance of the maturity of the systems under analysis is yet to be proved as various papers argue in favor (Weyuker et al., 2008) or against (Shatnawi and Li, 2008) the transferability of models between releases, preserving at the same time good performance. Our results indicate that the there is no evident sensitivity to the partition of the data used for learning with the proposed model, which manages to retain on average good levels of performance; nevertheless, its performance is slightly improved when using the latest releases.

The second considered research question, suggested by Hall et al. (2012), is related to the definition of the most appropriate independent variables that should be included in a model to achieve high fault prediction accuracy. Our experimental results show that change metrics seem more appropriate for defect prediction on average for all the modeled benchmark systems; among these, the MOSER setup stands out, yielding the highest performance. It should also be pointed out that results obtained from the MOSER setup are closely followed by those of source code metrics, and more specifically of metrics under the OO setup. The above findings underline the appropriateness of variables describing the degree and type of changes performed when debugging software, but also point out that descriptors of the object-oriented development scenery are quite useful for describing the evolution of systems' defect behavior.

4. Conclusions

In this paper, we addressed the problem of metrics-based software defect prediction by introducing a hierarchical Bayesian model for regression modeling of counts. Formulation of our method is based on a doubly stochastic homogeneous Poisson process formulation, where the failure rate parameter at each time point is modeled as a latent space parameter expressed as the output variable of an SBN-variant driven by the metrics data. This way, our model allows for better capturing nonlinearities compared to conventional (log-)linear approaches, and modeling data with multiple modes in their underlying distributions. We derived efficient model training and prediction algorithms, where inference is performed using MCMC, specifically an SIR scheme, and standard backpropagation is utilized to conduct parameter estimation. Training algorithm initialization was performed by means of the Glorot initialization scheme (Glorot and Bengio, 2010). This initialization consists in drawing the initial values of the parameters of a network layer from a Gaussian distribution with zero mean and variance equal to $2/(d_{in} + d_{out})$, where d_{in} is the dimensionality of the input, and d_{out} is the dimensionality of the output of the layer. This random initialization scheme is guaranteed to yield good estimators at every random start; we verified the latter findings in our experimental evaluations. Our empirical performance results, obtained by considering a number of benchmark datasets, showed that our approach yields a much better predictive performance than some state-of-the-art alternatives, without compromises in terms of computational costs.

An issue this work did not address concerns coming up with a way that allows for inferring a posterior distribution over the number of latent units. Such a capability would both alleviate the burden of heuristically determining this number on the grounds of obtained performance, as well as for regularizing the model in a manner robust to overfitting. To this end, one could consider imposition of appropriate nonparametric process-driven priors over the latent variables of the model, e.g. the Indian Buffet Process (IBP) prior (Griffiths and Ghahramani, 2005). Further, it would be interesting to examine whether our approach retains its desirable performance characteristics when dealing with proprietary software systems; however, this would require access to a variety of proprietary systems, which is a rather elusive goal. Hence, both these problems remain in the list of research directions we aim to pursue in the future.

References

Asuncion, A.U., Liu, Q., Ihler, A.T., Smyth, P., 2010. Particle filtered MCMC-MLE with connections to contrastive divergence. Proceedings of ICML.

Barber, D., Sollich, P., 1999. Gaussian fields for approximate inference in layered sigmoid belief networks. In: Proceedings of NIPS. MIT Press, pp. 393–399.

Bartz, K., Blitzsten, J., Liu, J., 2008. Monte Carlo Maximum Likelihood for Exponential Random Graph Models: From Snowballs to Umbrella Densities. Technical report. Harvard University.

Briand, L., Emam, K.E., Freimut, B., Laitenberger, O., 2000. A comprehensive evaluation of capture-recapture models for estimating software defect content. IEEE Trans. Software Eng. 26 (6), 518–540.

Chidamber, S.R., Kemerer, C.F., 1994. A metrics suite for object oriented design. IEEE Trans. Softw. Eng. 20 (6), 476–493.

Dalal, S., McIntosh, A., 1994. When to stop testing for large software systems with changing code. IEEE Trans. Softw. Eng. 20 (4), 318–323.

D'Ambros, M., Lanza, M., Robbes, R., 2010. An extensive comparison of bug prediction approaches. In: Proceedings of MSR 2010 (7th IEEE Working Conference on Mining Software Repositories). IEEE CS Press, pp. 31–41.

Ebrahimi, N., 1997. On the statistical analysis of the number of errors remaining in a software design document after inspection. IEEE Trans. Software Eng. 23 (8), 529–532.

Gilks, W., Berzuini, C., 2001. Following a moving target-monte carlo inference for dynamic bayesian models. J. R. Stats. Soc. Series B 63 (1), 127–146.

Glorot, X., Bengio, Y., 2010. Understanding the difficulty of training deep feedforward neural networks. Proceedings of AISTATS.

Gordon, N.J., Salmond, D.J., Smith, A.F.M., 1993. Novel approach to nonlinear/non-Gaussian Bayesian state estimation. IEE Proc. Radar Signal Process. 140 (2), 107–113.

Griffiths, T., Ghahramani, Z., 2005. Infinite Latent Feature Models and the Indian Buffet Process. Technical report TR 2005-001. Gatsby Computational Neuroscience Unit.

Hall, T., Beecham, S., Bowes, D., Gray, D., Counsell, S., 2012. A systematic literature review on fault prediction performance in software engineering. Softw. Eng. IEEE Trans. 38 (6), 1276–1304.

Handcock, M., Hunter, D., Butts, C., Goodreau, S., Morris, M., 2008. Statnet: software tools for the representation, visualization, analysis and simulation of network data. J. Stat. Softw. 24 (1), 15–48.

Hassan, A.E., 2009. Predicting faults using the complexity of code changes. In: Proceedings of ICSE., pp. 78–88.

Jordan, M., Ghahramani, Z., Jaakkola, T., Saul, L., 1998. An introduction to variational methods for graphical models. In: Jordan, M. (Ed.), Learning in Graphical Models. Kluwer, Dordrecht, pp. 105–162.

Khoshgoftaar, T., Seliya, N., 2003. Analogy-based practical classification rules for software quality estimation. Emp. Softw. Eng. 8 (4), 325–350.

Kong, A., Liu, J., Wong, W., 1994. Sequential imputations and bayesian missing data problems. JASA 89, 278–288.

Lessmann, S., Baesens, B., Mues, C., Pietsch, S., 2008. Benchmarking classification models for software defect prediction: a proposed framework and novel findings. IEEE Trans. Softw. Eng. 34 (4), 485–496.

McLachlan, G., Peel, D., 2000. Finite mixture models. Wiley Series in Probability and Statistics. New York.

Menzies, T., Greenwald, J., Frank, A., 2007. Data mining static code attributes to learn defect predictors. IEEE Trans. Softw. Eng. 33 (1), 2–13.
Moser, R., Pedrycz, W., Succi, G., 2008. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In: Proceedings of ICSE., pp. 181–190.
Nagappan, N., Ball, T., 2005. Use of relative code churn measures to predict system defect density. In: Proceedings of ICSE., pp. 284–292.
Nagappan, N., Ball, T., Zeller, A., 2006. Mining metrics to predict component failures. In: Proceedings of ICSE., pp. 452–461.
Neal, R.M., 1992. Connectionist learning of belief networks. Neural Comput. 56, 71–113.
Nelder, J., Wedderburn, R., 1972. Generalized linear models. J. R. Stats. Soc. Series A (General) 135 (3), 370–384.
Rinsaka, K., Shibata, K., Dohi, T., 2006. Proportional intensity-based software reliability modeling with time-dependent metrics. Proceedings of 30th IEEE Annual International Computer Software and Applications Conference.
Ruggeri, F., Soyer, R., et al., 2008. Advances in bayesian software reliability modelling. In: T.B., et al. (Eds.), Advances in Mathematical Modelling for Reliability. IOS Press, pp. 149–157.
Salakhutdinov, R., Tenenbaum, J.B., Torralba, A., 2013. Learning with hierarchical-deep models. IEEE Trans. Pattern Anal. Mach. Intel. 35 (8), 1958–1971.
Saul, L.K., Jaakkola, T., Jordan, M.I., 1996. Mean field theory for sigmoid belief networks. J. Artif. Intell. Res. 4, 61–76.
Shatnawi, R., Li, W., 2008. The effectiveness of software metrics in identifying error-prone classes in post- release software evolution process. J. Syst. Softw. 81 (11), 1868–1882.
Snijders, T., 2002. Markov chain monte carlo estimation of exponential random graph models. J. Soc. Str. 3 (2), 1–40.
Song, Q., Shepperd, M., Cartwright, M., Mair, C., 2006. Software defect association mining and defect correction effort prediction. IEEE Trans. Softw. Eng. 32 (2), 69–82.
Su, Y.-S., Huang, C.-Y., 2007. Neural-network-based approaches for software reliability estimation using dynamic weighted combinational models. J. Syst. Softw. 80, 606–615.
Torrado, N., Wiper, M.P., Lillo, R.E., 2013. Software reliability modeling with software metrics data via gaussian processes. IEEE Trans. Softw. Eng. 39 (8), 1179–1186.
Vanhatalo, J., Riihimäki, J., Hartikainen, J., Jylänki, P., Tolvanen, V., Vehtari, A., 2013. Bayesian modeling with gaussian processes using the GPstuff toolbox. J. Mach. Learn. Res. 14, 1175–1179.
Weyuker, E.J., Ostrand, T.J., Bell, R.M., 2008. Do too many cooks spoil the broth? using the number of developers to enhance defect prediction models. Emp. Softw. Eng. 13 (5), 539–559.
Wiper, M., Bernal, M.R., 2001. Bayesian inference for a software reliability model using metrics information. Proceedings of Safety and Reliability: Towards a Safer World.
Wohlin, C., Runeson, P., 1998. Defect content estimations from review data. In: Proceedings ICSE., pp. 400–409.

Andreas S. Andreou is an Associate Professor of the Department of Electrical Engineering/Computer Engineering and Informatics of the Cyprus University of Technology (CUT). Prior to this appointment he was elected Lecturer and Assistant Professor at the Dept. of Computer Science of the University of Cyprus. He is also the director of the Software Engineering and Intelligent Information Systems Research Lab at CUT. He studied Computer Engineering and Informatics at the University of Patras, Greece (Diploma, 1993, Ph.D., 2000). Prior to joining the academia he worked in the industry at the posts of Programmer-Analyst, Director of Requirements Analysis, and IT Consultant in Banking Systems. He also served as Software Engineering and IT consultant in several major software projects in Cyprus, including the Integrated Software System for the New Nicosia General Hospital. His research interests include Software Engineering, Web Engineering, Electronic and Mobile Commerce, and Intelligent Information Systems.

Sotirios P. Chatzis received the M.Eng. (Hons.) degree in electrical and computer engineering and the Ph.D. degree in machine learning from the National Technical University of Athens, Athens, Greece, in 2005 and 2008, respectively. He was a Post-Doctoral Fellow with the University of Miami, Coral Gables, FL, USA, from 2008 to 2010. He was a Post-Doctoral Researcher with the Department of Electrical and Electronic Engineering, Imperial College London, London, U.K., from 2010 to 2012. He is currently an Assistant Professor with the Department of Electrical Engineering, Computer Engineering and Informatics, Cyprus University of Technology, Limassol, Cyprus. He has authored more than 60 papers in the most prestigious journals and conferences of the research field. His current research interests include machine learning theory and methodologies, specifically hierarchical Bayesian models, Bayesian nonparametrics, and deep hierarchical feature extractors, with a focus on modeling data with temporal dynamics. His Ph.D. research was supported by the Bodossaki Foundation, Greece, and the Greek Ministry for Economic Development. Dr. Chatzis was a recipient of the Dean's scholarship for Ph.D. studies, being the best performing Ph.D. student of the class.