Probabilistic Graph Models for Debugging Software

Laura Dietz*
RG2: Machine Learning
Max Planck Institute for Computer Science
Saarbrücken, Germany
dietz@mpi-inf.mpg.de

Valentin Dallmeier
Dept. of Computer Science
Saarland University
Saarbrücken, Germany
dallmeier@st.cs.uni-sb.de

1 Introduction

Of all software development activities, debugging—locating the defective source code statements that cause a failure—can be by far the most time-consuming. We employ probabilistic modeling to support programmers in finding defective code. Most defects are identifiable in control flow graphs of software traces. A trace is represented by a sequence of code positions (line numbers in source filenames) that are executed when the software runs. The control flow graph represents the finite state machine of the program, in which states depict code positions and arcs indicate valid follow up code positions. In this work, we extend this definition towards an n-gram control flow graph, where a state represents a fragment of subsequent code positions, also referred to as an n-gram of code positions. We devise a probabilistic model for such graphs in order to infer code positions in which anomalous program behavior can be observed. This model is evaluated on real world data obtained from the open source AspectJ project and compared to the well known multinomial and multi-variate Bernoulli model [1].

Today’s best practice in software development suggests to develop two kinds of source code. Production code, which implements the functionality and will be shipped to customers; and test code, which consists of several self-contained programs (called test cases) that evaluate the correctness of routines in the production code. When a developer modifies production code (i.e. fixing defects or adding features) all test cases are executed. If any test cases fail, the production code contains a defect, which has to be resolved by the programmer before shipping the code. At this point a programmer should be supported by predicting which parts of the production code are likely to contain defects. User interface widgets might guide the user through a list of code positions that are likely to point him to the defect.

Current approaches towards the defect localization problem fall into three categories. Work of Elfeky et al. [2] relies on the fact that different programmers are likely to make the same errors when using some programming concepts, i.e. usages of complex concepts such as semaphores are likelier to be erroneous than simple operations like integer incrementation. The approach consists of learning a latent Dirichlet allocation on defective source code, treating it as a text document. The trained model can be employed to detect reoccurring defect patterns. On the downside, this approach relies on code that is manually labeled with defect topics, often created by intentionally introducing such mistakes in code.

The second category (which includes our approach) draws inference on the control flow of the program, i.e., the sequence of executed statements. The underlying assumptions is that code not covered by passing test cases is likely to contain defects. For that reason these techniques are also called code coverage methods. Tarantula [3] is a heuristic that yields rank scores for each statement, depending on the number of passing and failing test cases that executed this statement. So far, Tarantula is the best known algorithm to predicting defects. Our approach belongs to this category as well.

*Also affiliated with University of Potsdam, Germany.
In contrast to code coverage techniques which only examine the control flow of the program, the last category draws on the data space as well. The data space consists of assignments of local / global variables (int, float), member variables of objects (lists and aggregated objects), and the state of system devices and files. For instance, the IVMP approach [4] alters the data space to identify defective statements to outperform Tarantula. Liblit et al. [5] introduced an approach based on predicates that are associated to the program. Although some defects become only visible in the data space, the recording the data space over time results in huge amounts of data—taking much time to record and process—of which most parts are not instrumental in predicting defects. For that reason we study how well approaches can do without such additional effort.

This paper is outlined as follows. We start by giving details about the problem statement. In Section 3 we introduce a new probabilistic model, the Bernoulli graph model. Furthermore, we revise the multinomial and multi-variate Bernoulli models for the purpose of predicting anomalous code positions. Section 4 is about interpolating models of various context lengths and tuning hyperparameters by optimizing an normalized cumulative gain (NDCG) objective. The experimental evaluation is detailed in Section 5, after which we conclude.

2 Problem Statement

Control flow graphs can be reconstructed from a trace of the software execution path. Formally, a trace is represented by a sequence of statements where each statement $s$ is represented by its line number and the id $e$ of the containing entity, which might be a class, a method, or a package. For each test case, a trace $t$ is recorded. A trace from a passing (failing) test case is called a passing (failing) trace. Only few test cases—usually one—are failing in each revision.

The contents of source files change over time as statements are inserted, deleted or modified. The development history of a software project is stored in a source code repository, in which a revision $\tau$ represents the set of source files at a point in time. Note, that because of changes in the source code, the semantics of a code position changes over time. For a revision $\tau$ the set of defective statements $D_\tau$ contains the statements which have to be altered in order to fix the defect. For revisions in the past, the defective statements are obtainable from the code repository by comparing two subsequent revisions.

Although many approaches claim to predict the location of defects, most of them end up predicting anomalies. The difference between the two is that the defective statement causes the defect and thus, need to be altered, where the anomalous statement is the code position in which the defect becomes observable. We make this difference explicit in predicting a ranking of anomalies. Since a ground truth for anomaly labels is not available for any software project, we evaluate the ranking according to its usefulness in finding the defective statement.

**Problem 1.** Given passing $\{t\}_\tau$ and failing traces $\{T\}_\tau$ for a revision $\tau$; given traces $\{t\}_{\tau'}$ and $\{T\}_{\tau'}$ as well as sets $D_{\tau'}$ of defective statements of past revisions $\tau' < \tau$; predict a ranking over anomalous statements, where statements ranked within the top $k$ ought to be within distance $k'$ to a defective statement $d \in D_{\tau}$.

Note that $k$ (importance of rank) and $k'$ (importance of short distance) are parameters that need to be specified by the application setting. In the software development domain, $k$ represents the maximal rank that a user might inspect, $k'$ the maximal range within the suggested statement, in which he will look for defective statements.

Since it is unrealistic to choose appropriate configurations of $k$ and $k'$ a priori we either plot over them or integrate them out by resorting to the normalized cumulative gain measure (NDCG). In NDCG, the importance of rank and distance is given by logarithmic decay function parametrized by $a$ and $b$ respectively.

**Remark.** Straightforward discriminative learning from defects of in past traces is not possible for two reasons. First, code positions change their semantics over time. For instance if subsequent execution of statements at line 17, 18, and 19 is normal in revision 1, it might be that in revision 5 a line is inserted before the block, so that the previously normal fragment 17,18,19 might become a very unusual in revision 6.
Figure 1: Process for generating outgoing edges. Left: source code containing a defect. Bottom left: trace for executing the code with input argument 1. Right: Trigram control flow graph for the trace where the generative process is expanding vertex \( u \) labeled 22,18. Coin tossing trials yield 1 for arcs towards vertices 18,19 and 18,24. These arc correspond to consecutively executing lines 22,18,19 and 22,18,19 which are depicted by arrows in the source code to the left. Although depicted otherwise for clarity, vertex 18,19 is already contained in the graph, thus only the arc is inserted and the vertex is reused.

Furthermore, in comparison to the vast amount of statements in one trace, only one or two statements are defective in each revision, yielding very little data for the negative class.

For that reason, one has to resort to approaches that rely on positive data only. This applies to the probabilistic models given in Section 3. In section 4 we describe a tuning scheme to exploit data from past revisions indirectly.

3 Probabilistic Models for n-Gram Control Flow Graphs

In order to predict anomalies in source execution traces we examine generative models that are trained from passing traces of one revision only. We introduce a new probabilistic model for graphs and revise how the multinomial and multi-variate Bernoulli model may be applied to the software domain. This section specifies their generative processes to yield corresponding anomaly scores. In this section we assume a fixed context length \( n \) and given hyperparameters.

3.1 Bernoulli Graph Model

The Bernoulli graph model generates an n-gram equivalent of a trace’s control flow graph. In an bigram control flow graph, each vertex \( v \) represents a statement; each arc \( (v_1, v_2) \) indicates that \( v_2 \)’s statement may be executed after the statement of \( v_1 \). In an n-gram control flow graph, each vertex \( v \) represents a fragment \( f \) of \( n-1 \) consecutively executed statements. An arc \( (v_1, v_2) \) for corresponding fragments \( f_1, f_2 \) represents a valid transition, meaning that after executing statements in \( f_1 \) there is a follow up statement \( s_2 \) that may be executed, with \( s_2 \) being the last statement of \( f_2 \). Such an arc may only exist if the fragments overlap in all but statement \( s_2 \). This is true if the fragments can be decomposed into \( f_1 = s_1 \circ f' \) and \( f_2 = f' \circ s_2 \). An example of a trigram control flow graph is given in Figure 1 (right).

**Generative process.** The Bernoulli graph model generates one graph per trace and entity, yielding a stochastic model for edge occurrences in the entity’s control flow.

The model begins the generation for each graph with an initial vertex representing an fragment of empty statements \( \varepsilon \). It subsequently keeps adding outgoing arcs for one vertex after the other and (if necessary) adds the end points of the arc as new vertices. It maintains a queue of vertices for which outgoing edges have not yet been generated.
Initializing the queue with the initial vertex, the model iterates the following steps until the queue is empty. A vertex is dequeued. Assuming is labeled with fragment , for all statements the sequence is generated by drawing entities the model can be applied to our prediction task. For each entity a set of entity-wise multinomial distributions be derived. Because of the beta-Bernoulli conjugacy, a point estimator for arc parameters as in Equation 1 can be derived.

\[
\hat{\psi}_{e,(u,v)}(b = 1) = \frac{\# \{ G_e | (u, v) \in G_e \} + \alpha_u}{\# \{ G_e | u \in G_e \} + \alpha_v + \beta_v} \quad (1)
\]

\[
\hat{\psi}_{e,fo}(b = 1) = \frac{\# \{ t | f \circ s \in t \} + \alpha_s}{\# \{ t | f \in t \} + \alpha_s + \beta_s} \quad (2)
\]

There is a duality between graph elements and fragments, as arc iff the fragment is contained in the corresponding trace , and vertex exists iff 's label is contained in the trace. We can exploit this duality for estimating without making the graph representation of traces explicit.

**Predicting anomalous statements.** Once is estimated from data, allows to estimate the likelihood of statements in a new trace. Since is very close to 1 if an occurrence of statement after has been anticipated in many passing traces, it represents a measure for normality. In contrast to this, we are interested code positions where anomalous behavior of the programs becomes observable. In such positions, the failing trace deviates from the language model that is learned from passing traces. Quantifying the deviation from the language model introduces a ranking of statements by defectiveness as given in Equation 3.

\[
\text{score}(s, e) = 1 - \min_{f \in t_e} \left( \hat{\psi}_{e,fo}(1) \right) \quad (3)
\]

A statement may have various preceding fragments within the failing trace. Many of these combinations may be in line with the passing test cases, but just one anomalous combination is sufficient to give rise to an error. For that reason we score each statement according to the highest deviation achieved by any of its preceding fragments, represented by a minimum in Equation 3.

### 3.2 Multinomial Model

The multinomial n-gram model [1] is well known in text domains. By replacing words with statements, the model can be applied to our prediction task. For each entity in a trace the multinomial generates a sequence of statements, each depending on the preceding fragment.

A set of entity-wise multinomial distributions (ranging over statements) and one additional multinomial (ranging over entities) are drawn from a Dirichlet distribution with parameters , respectively. The sequence is generated by drawing entities and statements of its multinomial conditioned on the fragment of n-1 preceding generated statements.

Due to multinomial-Dirichlet conjugacy, point estimates and can be inferred exactly. The probability for a statement in entity given fragment is given by

\[
p(s, e | f; \hat{\gamma}, \hat{\phi}) = \frac{\text{\#}(s | f) + \alpha_s}{\#(f | t) + \alpha_s} . \quad \text{Anomaly scores are derived in analogy to the Bernoulli graph model.}
\]

### 3.3 Bernoulli Fragment Model

Like the multinomial, the multi-variate Bernoulli model [1] has successfully been applied to text domains. The model generates a set of all fragments of length which occur in a trace. The set
is generated by tossing a coin with parameter $\zeta_{e,f}$ for each fragment $f$. If the coin $\zeta_{e,f}$ yields 1, fragment $f$ is added to the observed set. All coin parameters are drawn from a Beta distribution with hyperparameters $\alpha_e$ and $\beta_e$. Thus, $p(s, e|f; \zeta) = \frac{\#\{|f\in\ell\}+\alpha_e}{\#\ell+\alpha_e+\beta_e}$ gives the likelihood for new statements and anomaly scores are obtained as before.

Note, that in the unigram case Bernoulli graph model generates a set of unconnected vertices and thus yields the same scores as Bernoulli fragment model.

**Relation to Tarantula.** A connection to the Tarantula approach exists for the unigram case where $n = 1$. Originally, Tarantula was an HCI project to research how results of unit tests could be best presented to the user. As one result, the authors proposed to rank statements by score $s = \frac{\%\text{fail}(s)+\%\text{pass}(s)}{\%\text{fail}(s)+\%\text{pass}(s)}$ where $\%\text{fail}(\%\text{pass})$ refers to the fraction of failing (passing) test cases that execute statement $s$.

In the special case of having only one failing test case at a time (which is the case in most of our revisions) $\%\text{fail}(s) = 1$ and $\%\text{pass}(s) = \frac{\#\{|t|e\ell\}}{\#\ell}$, thus Tarantula’s score boils down to be equivalent to ranking by score $s = \frac{1}{1+\%\text{pass}(s)}$. Any statement $s_1$ is ranked above $s_2$ by Tarantula iff $\frac{\#\{|t|e\ell\}}{\#\ell+\%\text{pass}(s_1)} > \frac{\#\{|t|e\ell\}}{\#\ell+\%\text{pass}(s_2)}$ which is equivalent to $\frac{\#\{|t|e\ell\}}{\#\ell} + \alpha > \frac{\#\{|t|e\ell\}}{\#\ell} + \alpha + \beta$ for constants $\alpha$ and $\beta$. Thus Tarantula’s score yields the same ranking as a fragment Bernoulli model with $n = 1$, as can be seen when inserting $p(s, e|f; \zeta)$ into the score function. Moreover we see that constant Beta parameters do not influence the ranking.

## 4 Model Smoothing and Tuning by NDCG

All three generative models take a fixed fragment length $n$ as input. The correct choice will influence the prediction performance. In order to study the assets of each model, we smooth the prediction by a set of models for various values of $n \in \{1, 2, \ldots, N\}$ by interpolation with mixing coefficients $\lambda$. In the following, the model parameters of either multinomial ($\gamma, \phi$), Bernoulli fragment ($\zeta$), or Bernoulli graph ($\psi$) for a given $n$ are represented by $\theta_n$. Each of the $N$ models gets its own set of hyperparameters denoted by $\alpha_n$.

If coefficients and hyperparameters are given, the ranking scores of the smoothed model are aggregated by score $s(e, s|\lambda, \alpha_n) = 1 - \sum_{n=1}^{N} \lambda_n \cdot \min_{f \in \ell, \theta_n(s|f; \alpha_n)}$.

As noted in Section 2, a statement at rank $i$ is a successful prediction, if $i < k$ and the statement $s_i$ has at most distance $k'$ from any defective statement. In the following, we give a scheme for tuning $\lambda$ and $\alpha$ so that the number of successful predictions is optimized, using prediction problems of past timepoints $\tau$ of which defect labels are available. Fixed values for $k$ and $k'$ do not pay respect to the fact that higher ranks (shorter distances) are more valuable than lower ones. Thus, we integrate over diverse configurations by evaluating the gain of showing an anomalous statement $s$ to the developer. The gain of $s$ is the higher, the closer $s$ is located to a defective statement $d$. The gain is maximal when $s$ is a defective statement itself.

We measure the gain of defect vicinity by a logarithmic decay function $\text{gain}(s, D) = \frac{1}{\log_{\alpha}(\text{dist}(s, D)) + b}$ that yields 1 if $s \in D$. Distance is measured by the number of lines between $s$ and the next defective statement in the source code, with infinite distance if the defect is in a different source file. Analogously, the gain of a high rank is modeled by another decay $\frac{1}{\log_{\beta}(\text{rank} + \alpha)}$ with highest gain at the top. Multiplying both gains and summing over the top $K$ statements, yields the discounted cumulative gain measure (DCG). DCG can be normalized by the DCG of an optimal ranking to obtain the NDCG measure as given in Equation 4. NDCG yields values between 1 and 0, where higher values represent better rankings.

$$NDCG(D, T, \ell, \lambda, \alpha) = \frac{1}{\text{DCG}^*(D)} \cdot \sum_{\text{ranks} \leq K} \left( \frac{1}{\log_{\alpha}(i + \alpha)} \cdot \frac{1}{\log_{\beta}(\text{dist}(s_i|\lambda, \alpha, D) + b)} \right)$$

Parameters $a$ and $b$ express the trade-off between rank and distance, which depends on the application domain.
In order to learn rankings by optimizing an NDCG based criterion, a couple of approaches exist, by constructing a convex upper bound [6] or using a Gaussian process [7]. Since choices of $\lambda$ and $\alpha$ influence the ranking indirectly via the generative models, an adoption of the approaches is not straightforward. For the sake of simplicity, we use a brute-force optimization approach in this work.

We regularize the NDCG objective by vague priors. Each model specific hyperparameter $\alpha_{n,m}$ is regularized by a Gamma distribution centered around 1.0 with parameters $\eta_{\text{scale}}$ and $\eta_{\text{shape}}$. $\lambda$ gets a symmetric Dirichlet prior that prefers dense mixtures with parameter $\eta_\lambda$. The third term of the equation is the arithmetic mean of the NDCG values (Equation 4) over rankings for past revisions $\tau'$ with passing traces $T_{\tau'}$ and defective statements $D_{\tau'}$. The regularized optimization criterion is given in Equation 5.

$$
\hat{\lambda}_\tau, \hat{\alpha}_\tau = \arg \min_{\lambda, \alpha} \left\{ -C_1 \frac{1}{MN} \left( \sum_{n=1}^{N} \sum_{m=1}^{M} \log \text{Gamma}(\alpha_{n,m}|\eta_{\text{scale}}, \eta_{\text{shape}}) \right) - C_2 \log \text{Dirichlet}(\lambda|\eta_\lambda) - \frac{1}{(\tau - 1)} \sum_{\tau'=1}^{\tau-1} \text{NDCG}(D_{\tau'}, T_{\tau'}, \hat{t}_{\tau'}|\lambda, \alpha) \right\}
$$

### 5 Experimental Evaluation

**Data set.** Our data set is obtained from product and test code of the AspectJ project. From the code repository we select 50 source code revisions $\tau$ that contain defects. Only such revisions are selected for which a test case fails in revision $\tau$, but passes in the follow up revision. For each such revision $\tau$, execution traces $T_{\tau}$ of passing test cases are recorded on a statement basis. In the same manner, the failing trace $\hat{t}_{\tau}$ (in which the defective statement is to be identified) is recorded. The ground truth for the defective statements $D_{\tau}$ is taken from the source code repository as the line numbers that were modified between revision $\tau$ and its subsequent revision.

Because of time limitations the experiments were conducted on a smaller data set which contained all statements of the defective class plus statements of at least 100 further entities that are executed
Table 1: NDCG results with $a = 1.5$, $b = 10$ for the evaluated approaches. Columns to the right mark significant improvements towards compared approaches using a single sided pair-t-test with significance niveau 1%.

<table>
<thead>
<tr>
<th>Model</th>
<th>NDCG</th>
<th>m</th>
<th>f</th>
<th>g</th>
<th>t</th>
<th>u</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multinomial model (m)</td>
<td>0.46 ± 0.06</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Bernoulli fragment model (f)</td>
<td>0.47 ± 0.06</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Bernoulli graph model (g)</td>
<td>0.51 ± 0.07</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tarantula (t)</td>
<td>0.45 ± 0.06</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unigram multinomial model (u)</td>
<td>0.40 ± 0.05</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

by the failing trace. In this data set each trace contains approx. 100,000 executed statements, 25 classes and 110 methods. We report results on method entities, but preliminary experiments yielded similar results for class entities.

Some revisions were also evaluated on the complete data set, yielding lower absolute ranks, higher relative\(^1\) ranks and same order of approaches according to prediction performance.

**Compared approaches.** In order to study the helpfulness of each generative model, we evaluate interpolated models with $n \in \{1, 2, 3, 4, 5\}$ for each the multinomial, Bernoulli fragment and Bernoulli graph model. We compare those towards two base lines: The unigram multinomial model and Tarantula (which amounts to be an unigram Bernoulli fragment model).

**Tuning grid and regularization parameters.** For the interpolated models, $\lambda$ and $\alpha$ are tuned to optimize Equation 5 on a grid. The grid contains about 30 values for each hyperparameter ranging from 0.0001 to 5.0 for Dirichlet parameter $\alpha_\phi$; from 0.01 to 20.0 for Dirichlet parameter $\alpha_\gamma$ and from 0.01 to 10.0 for each of the two Beta parameters $\alpha_\psi$ and $\beta_\psi$. The grid for $\lambda$ includes all possibilities to distribute 100 units on the 5 mixing dimensions, with each mixture having at least\(^2\) one unit.

For the unigram multinomial model hyperparameters $\alpha$ are tuned on the same hyperparameter grid. As pointed out before, the hyperparameter of the unigram Bernoulli fragment model have no effect on the ranking and thus do not need to be tuned.

We chose the parameters of the regularizer to have up to 0.1% effect on interpolation and hyperparameters. This condition is met for $C_1 = 0.0012$, $\eta^\text{shape}_\alpha = 1.1$, $\eta^\text{scale}_\alpha = 10$, $C_2 = 0.03$, $\eta_\lambda = 1.01$.

**Evaluation measures.** The models are evaluated by NDCG of the predicted ranking given in Equation 4 and by plotting the successful predictions at given ranks $k$.

For the purpose of visualizing the anomalous statements within an integrated development environment, a high rank is more important than short distance. If developers are assumed to have a 21” display, at least 60 lines of code fit on a screen, thus distances up to 30 lines of code have still the potential to be helpful. In contrast to this, a developer may examine only anomalies within a few top ranks. For instance, this prior belief is met when setting the NDCG parameters to be $a = 1.5$, $b = 10$, and $K = 100$. Table 1 gives the prediction performance measured with NDCG. Significance of improvement is evaluated by a paired-t-test with significance niveau 1%.

Likewise, one may evaluate the fraction of predictions within the top $k$ being within max distance $k'$ of a true defect. This fraction can be plotted either over ranks $k$ for fixed distance, or over distances $k'$ for a fixed rank. The former plot for $k' \in \{0, 10\}$ is displayed in the top row of Figure 2, the latter in the bottom row for $k = 5$.

**Results.** Evaluation by NDCG (Table 1) demonstrates that the Bernoulli graph model suits developers’ needs significantly better than the other generative models. Given that optimal ranking would

\(^1\)Relative to the number of statements to be ranked.

\(^2\)Mixtures with empty components will be penalized towards infinity by the Dirichlet regularizer $\log \text{Dirichlet}(\lambda | \eta_\lambda)$. 

7
yield NDCG values of 1, one might object that benefits seem to be limited. The issue is that in order to obtain NDCG values of 1, all statements in the top $k$ must be located in the vicinity of a defect. Since debugging is an iterative activity, the user requires only one of the top ranked statements to be indicative of the defect. As can be seen from the fraction of successful predictions shown in Figure 2 (top right), the Bernoulli graph model yields 40% of successful predictions if only the top 5 ranks were inspected, with a prediction being successful if within distance of 10 code lines to a defective statement. Thus, with adequate visualization in an integrated development environment, five mouse clicks are sufficient to find the defect in nearly every second case, in contrast to spending hours to locate the defect cause manually.

From the top left plot in Figure 2 we see that Tarantula and the corresponding Bernoulli fragment model are better in identifying the actual defective statement, but multinomial and Bernoulli graph models catch up when longer distances are still helpful for the application. The bottom plot in Figure 2 demonstrates that Tarantula tends to find only such “low hanging fruit”, where Bernoulli graph model is constantly improving when relaxing the distance constraint.

6 Conclusions

Executions of a software program have quite rich graph based representations, which can be deflated from a trace. One may examine graphs about calling and called classes, static object aggregation graphs and UML sequence graphs. From the various options we chose to study control flow graphs by probabilistic models. We introduced the Bernoulli graph model which is a new generative model for inferring defective statements in source code and studied its relation to the state-of-the-art scoring heuristic Tarantula.

Experimentally, we compared the Bernoulli graph model to previously successfully applied generative models, namely the multinomial and the multi-variate Bernoulli model and their n-gram counterparts. Although being of similar performance in predicting defective statements, the strengths of the Bernoulli graph model come into play when predicting statements in a defect’s vicinity. If properly integrated into an development environment, it has the potential to replace tedious debugging sessions with a few mouse clicks.

Acknowledgments

Laura Dietz is funded by a scholarship from Microsoft Research Cambridge.

References