Automatic Recovery of Missing Issue Type Labels

Farida El Zanaty*, Christophe Rezk*, Sander Lijbrink†, Willem van Bergen†, Mark Côté†, Shane McIntosh‡
* McGill University, Canada; {farida.elzanaty, christohpe.rezk}@mail.mcgill.ca
† Shopify, Inc., Canada; {sander.lijbrink, willem, mark.cote}@shopify.com
‡ University of Waterloo, Canada; shane.mcintosh@uwaterloo.ca

***Abstract*—Today's agile software organizations aim to empower developers to make appropriate decisions rather than enforce adherence to a process. As a result, the data in software archives is more likely to be incomplete and noisy. Since software analytics techniques are trained using this data, automated techniques are required to recover such information.**

In this paper, we lay the foundation for the adoption of software analytics techniques at Shopify (a large software organization that develops commerce-related products and solutions) by recovering missing issue type labels. To do so, we train classifiers to label issue reports as defect-fixing or not using textual features from 951 manually-labelled issue reports. Our classifiers show promise in intra- and inter-project experimental settings: (1) outperforming baseline approaches in the intra-project setting like random guessing (AUC values of 0.5271–0.8070) and Zero-R (F1-scores that are 0.31–21.72 percentage points better); and (2) achieving inter-project performance scores that are on-par with intra-project classifiers when trained using pools of data that are drawn from multiple other projects. Interestingly, when the importance of precision is taken into consideration, standard model construction operations like rebalancing should be omitted to produce classifiers that are more suitable for deployment.

I. Introduction

Modern software development generates plenty of data. For example, commits record changes to codebases in Version Control Systems (VCSs). Issue Tracking Systems (ITSs) contain issue reports that describe product defects, enhancement requests, and other potential improvements. Review Tracking Systems (RTSs) are archives of peer code review discussions that take place during software development.

Software analytics techniques aim to discover and communicate insights from this software data. For example, by mining historical issue reports (ITS) and change records (VCS), researchers have proposed: (1) defect prediction [5], [10], [13], which helps stakeholders to allocate quality assurance resources to the modules or changes that are most likely to contain or introduce defects; and (2) bug localization [6], [12], [14], helps developers to locate the modules in a codebase that are likely to require changes to address a given issue report.

While software analytics techniques present opportunities for organizations, the value that they provide depends on the quality of the software process data that is available. In modern software organizations, the focus has shifted from strict adherence to process rules to more flexible Agile approaches. While providing several benefits for the organizations (e.g. faster reactions to field emergencies and edge cases), following less rigid processes may threaten the quality of software process data that is available for software analytics.

Indeed, one of the foundational pillars of the Agile Manifesto [1] suggests that software organizations should empower individuals rather than strictly adhere to a process. In following that principle, developers at Shopify (a large software organization that specializes in commerce solutions) are free to submit issue reports without specifying the issue type (i.e. bug, enhancement). This presents a barrier to the adoption of software analytics because identifying code change activity that is related to defect repair (i.e., changes associated with bug-type issues) is a critical step in several software analytics approaches. For example, defect prediction relies on a crisp delineation between bug-type issue reports (i.e., issue reports describing bugs in the system) and other issue reports to be able to train a classifier to predict in which modules future bug-type issue reports are likely to appear.

To tackle this problem, we present an approach to automatically recover issue types. Although we are not the first to study the problem [5], [10], to the best of our knowledge, this is the first attempt in an industrial setting. We (1) manually classify a random sample of issue reports; (2) train machine learning classifiers to automatically identify bug-type issue reports using features that we derive using Natural Language Processing (NLP); and (3) evaluate those classifiers in both intra- and inter-project configurations. Broadly speaking, through analysis of 951 issue reports from three repositories developed by Shopify, we address the following two research questions:

(RQ1) Can our classifiers accurately predict when an issue report describes a defect? Our classifiers outperform random guessing (AUC values of 0.5271–0.8070) and Zero-R baselines (F1-score improvements of 0.31–21.72 percentage points).

(RQ2) How well does classifier performance transfer to among projects? When data sets from two other projects are pooled to create a training sample, our models achieve performance that is on par with intra-project classifiers. More specifically, the performance is within five percentage points unless the more lightweight KNN and MNB classification techniques are applied. When the SVC and MLP classification techinques are applied, the F1-score and AUC measures improve with respect to intra-project baselines in four of six and two of six experiments, respectively.

Our results highlight the promise of combining NLP and ML techniques to recover missing issue types and lay the groundwork for adopting software analytics at Shopify.

TABLE I
AN OVERVIEW OF THE SUBJECT PROJECTS.

Project	Description	Entire Project		Manually-Classified Sample	
		#Issues	Time Frame	%Bugs	# Issues
SHOPIFY CORE	The core components of Shopify's e-commerce platform.	205,366	2011–2019	41	330
PARTNERS	Resources for other organizations that build on top of the Shopify platform.	17,363	2016–2019	37	325
STARSCREAM	Apache Spark-based data modelling tool.	34,540	2014–2019	37	296

II. EXPERIMENTAL DESIGN

In this section, we introduce the studied projects and present our approaches to data preparation and model construction.

A. Studied Projects

Shopify maintains a large collection of software systems hosted in several repositories. We evaluate our approach on SHOPIFY CORE—the largest and most critical component of the Shopify system. To improve the generalizability and replicability of our study, we evaluate our approach on PARTNERS and STARSCREAM—two other Shopify projects of more narrow domains. Table I provides an overview of the three studied Shopify projects. SHOPIFY CORE and PARTNERS are written in Ruby, while STARSCREAM is written in Python. SHOPIFY CORE, as the name implies, is the underlying core of the e-commerce system. PARTNERS is a component that supports partner organizations who are extending and/or using Shopify products/services. STARSCREAM is a machine learning and data modeling component.

B. Data Preparation

For each studied system, we need to prepare the data for our analysis. Figure 1 provides an overview of our data preparation process, which is comprised of the following three steps.

(DP1) Representative Sample Selection. In order to study the bug-type issues of each project, we randomly select a representative sample of the available issues by extracting the list of Issue IDs from the GitHub repositories. We then select a random ID number, and download the corresponding issue using the GitHub API. Next, we manually classify it as bug-type or not (more details are provided under DP2 below). Issues that are closed as duplicates are discarded before the manual classification step. We repeat this process until we achieve saturation [8], i.e., adding and classifying new data does not add meaning to our categories. In our study, we operationalize saturation by checking for when the performance of preliminary prediction models plateau (see Section II-C for model construction and analysis details). We train our models using a growing sample size to display the number of issue reports necessary for the performance of prediction models to plateau.[1] From this analysis, we observe that our samples saturate after 330 (SHOPIFY CORE), 325 (PARTNERS), and 296 (STARSCREAM) issue reports, respectively.

[1]https://figshare.com/s/9984ef64e3cfd8b11dc0

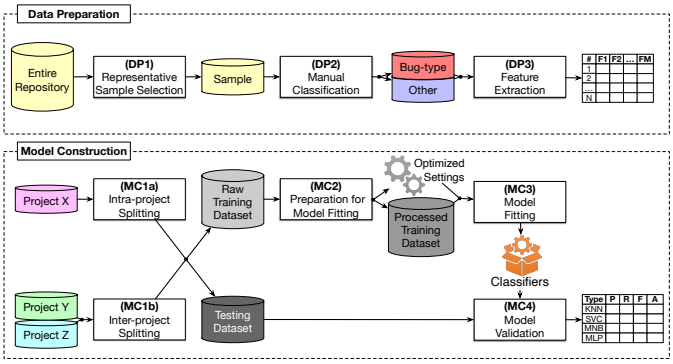

Fig. 1. An overview of our study design

(DP2) Manual Classification. To identify bug-type issue reports, we manually classify each sampled issue report. The first two authors participated in the manual classification process. Each author independently classified each issue as bug-type or not. In the rare cases where the authors disagreed about an issue report, a discussion took place until a consensus is reached. We eventually reached a consensus for each issue report; however, the pre-discussion inter-rater agreement score is *Near Perfect* for SHOPIFY CORE and *Substantial* for PARTNERS, and STARSCREAM (Cohen's Kappa values of 0.96, 0.80 and 0.74, respectively).

(DP3) Feature Extraction. Our models will make decisions about whether an issue report is of type-bug or not using textually extracted features. These textual features are extracted from the title and body of each sampled issue report.

As is common practice when analyzing natural language, stopwords (i.e., common words that add little value to a sentence) are removed, all letters are converted to lowercase, and each surviving term is stemmed (using the Porter Stemmer).

For each issue, we calculate the Term Frequency - Inverse Document Frequency (TF-IDF) weights, corresponding to the issue text. Consequently, each issue is represented by a feature vector (TF-IDF weights) and a field indicating whether the issue is bug-type or not.

C. Model Construction

After preparing our sample of labeled issue reports, we then use that sample to train our issue report classifiers, as shown in Figure 1. This model construction process is comprised of four key steps, which are described below.

(MC1a) Intra-project splitting. In our intra-project setting, we apply a 10x10 fold cross-validation approach to intra-project performance evaluation. As such, the data is split into ten folds of equal size, nine of which (90%) are used for training our models, while the remaining fold (10%) is used for testing. The process is repeated ten times—with each fold serving as the testing fold once. Since performance reported may be sensitive to the fold generation process, we repeat the entire cross-validation process ten times. The mean performance scores of the 100 iterations are reported.

(MC1b) Inter-project splitting. Since the manual creation of training data is expensive (see DP2), we set out to check the inter-project transferability of our models. In this setting, our models are trained using the data of a given project, and tested using the other two (unseen) projects.

(MC2) Preparation for Model Fitting. Prior to fitting our models, we must handle imbalanced categories and select hyperparameter settings.

Treating Imbalance: Our dataset is imbalanced—only 40% of SHOPIFY CORE issues and 37% of PARTNERS and STARSCREAM issues are classified as bug-type. To reduce bias towards the majority class (non-bug-type), the training data can be re-sampled. Our re-sampling process is implemented using a combination of oversampling of the minority class and undersampling of the majority class. We do so because prior work has shown that such a combination performs better than either oversampling or undersampling in isolation [2].

Hyperparameter Optimization: The classification techniques that we use have configurable parameters that impact their performance. To tune these settings, we apply a grid search, which examines all combinations of a specified set of candidate settings to find the best one. The search is applied only to training folds and is repeated for each experimental iteration.

(MC3) Model Fitting. We compare classifiers trained using classification techniques from four popular algorithmic families. Using the Euclidean distance between issues, K Nearest Neighbours (KNN) selects the K most similar training examples to classify an instance. Support Vector Classifiers (SVC) use a hyperplane to classify issues, which are mapped to a multidimensional feature space. Multinomial Naïve Bayes (MNB) uses estimates of conditional probabilities to classify issues. Multi-Layer Perceptron (MLP) uses neurons in sequential layers as an ensemble to classify issues.

(MC4) Model Validation. We use the common classifier performance measures of precision, recall, F1-score, and the Area Under the receiver operator characteristic Curve (AUC). Precision is the ratio of correctly predicted bug-type issues to the total number of issues that were predicted to be bug-type (Precision= $\frac{TP}{TP+FP}$), and Recall is the ratio of actual bug-type issues that a model can detect (Recall= $\frac{TP}{TP+FN}$). There is a natural tradeoff between recall and precision. To account for this, we use the F1-score, which is the harmonic mean of recall and precision: $F1 = \frac{2 \times Precision \times Recall}{Precision + Recall}$. AUC estimates the area under a curve that plots the false positive rate against the true positive rate as the classification threshold is varied. When the AUC exceeds 0.5, the classifier is said to perform better than random guessing.

III. EXPERIMENTAL RESULTS

In this section, we present the results of our study with respect to our two research questions.

(RQ1) Can our classifiers accurately predict when an issue report describes a defect?

Observation 1—Our classifiers outperform baseline approaches. Table II shows that the AUC of our models never drops below 0.5, indicating that all of our classifiers outperform random guessing approaches.

Moreover, our classifiers outperform Zero-R—a naïve technique, which in our case assumes that all issues are bugs. Zero-R achieves an overly optimistic recall of 1.0000 (since all bug-type examples are found) and an overly pessimistic precision (equal to the proportion of bug-type issues). Therefore, we compare against the F1-score, which is 0.5816 for SHOPIFY CORE and 0.5401 for PARTNERS and STARSCREAM.

Table II shows that our classifiers achieve F1-scores of 0.5620–0.7573, outperforming Zero-R classifiers in all cases by 0.31–21.72 percentage points. A closer look reveals that rebalancing has an large impact on the precision and recall of our KNN classifiers, leading to drops in the F1-score of 15.25 (SHOPIFY CORE), 16.78 (PARTNERS), and 6.41 (STARSCREAM) percentage points. If these rebalanced KNN rows are ignored, the minimum F1-score increases to 0.6483 and our classifiers outperform the Zero-R baseline by a minimum of 10.54 percentage points. These results suggest that our (non-KNN) classifiers are a solid starting point for the issue type-detection problem at Shopify.

Observation 2—Better performance does not imply more suitable for deployment. Recall that our classifiers will be deployed as a preliminary step to other software analytics modeling (e.g., defect prediction). For this reason, precision is a more important measure to optimize, since the cost of false positives is larger that of false negatives. Table II shows that rebalancing (SMOTE = Y) improves recall at the cost of precision. Even though the F1-scores tend to improve, indicating the benefit to recall is larger than the cost to precision, the importance of precision implies that the non-rebalanced classifiers may be more suitable for deployment.

(RQ2) How well does classifier performance transfer to among projects?

Observation 3—Classifiers trained with multi-project pools of data tend to achieve performance scores on par with intra-project classifiers. Table III shows the change in performance between intra-project and inter-project settings. Inter-project refers to distinct projects within SHOPIFY. The area of Table III above the double horizontal line suggests that there are often considerable performance drops when classifiers trained on one project are applied to another. This is not unexpected, as the projects vary in scope, domain, and programming language. On the other hand, the area of Table III below the double horizontal line shows the performance of models trained using pooled data from the other projects. The results suggest that larger pools of data often yields performance scores within five percentage points of the intra-project performance scores, echoing earlier observations about the importance of training sample size [7], [11]. Indeed, the F1-score and AUC of the other classifiers only exceeds five percentage points when KNN and MNB-based classifiers are tested against SHOPIFY CORE and STARSCREAM. In fact, we observe seven instances of F1-score and AUC improvements with respect to the intra-project performance.

TABLE II
AN OVERVIEW OF THE INTRA-PROJECT PERFORMANCE SCORES (RQ1). VALUES IN BOLDFACE ARE TOP PERFORMERS FOR EACH PROJECT.

Proj.	Class. tech.	SMOTE	Prec.	Rec.	F1	AUC
Shopify Core	KNN	N	0.8296	0.6744	**0.7372**	**0.7898**
		Y	0.4198	**0.9802**	0.5847	0.5271
	SVC	N	**0.8758**	0.5774	0.6870	0.7602
		Y	0.8376	0.6268	0.7084	0.7700
	MNB	N	0.8412	0.6020	0.6931	0.7602
		Y	0.7427	0.7137	0.7213	0.7704
	MLP	N	0.7864	0.6446	0.6999	0.7605
		Y	0.7728	0.6630	0.7045	0.7624
Partners	KNN	N	0.8723	0.6405	0.7298	0.7932
		Y	0.3982	**0.9962**	0.5620	0.5627
	SVC	N	0.9014	0.6109	0.7166	0.7876
		Y	0.8618	0.6690	**0.7423**	**0.8052**
	MNB	N	**0.9127**	0.5347	0.6632	0.7523
		Y	0.8753	0.6354	0.7248	0.7923
	MLP	N	0.8515	0.6568	0.7297	0.7959
		Y	0.8378	0.6757	0.7343	0.8007
Starscream	KNN	N	0.6909	0.6322	0.6483	0.7281
		Y	0.4193	**0.9908**	0.5842	0.5693
	SVC	N	**0.8784**	0.6083	0.7067	0.7767
		Y	0.8300	0.6517	0.7196	0.7827
	MNB	N	0.8599	0.5785	0.6780	0.7575
		Y	0.7663	0.7635	**0.7573**	**0.8070**
	MLP	N	0.7771	0.6282	0.6841	0.7566
		Y	0.7642	0.6421	0.6882	0.7582

TABLE III
THE CHANGE IN PERFORMANCE SCORES IN INTER- AND INTRA-PROJECT SETTINGS (RQ2). VALUES GREATER THAN ONE INDICATE THAT INTER-PROJECT CLASSIFIERS PERFORM BETTER THAN INTRA-PROJECT ONES, WHILE NEGATIVE VALUES INDICATE THE OPPOSITE. VALUES IN BOLDFACE ARE TOP PERFORMERS FOR EACH PROJECT.

Train	Test	Class. tech.	ΔPrec.	ΔRec.	ΔF1	ΔAUC
Shopify Core	Partners	KNN	-0.1485	-0.0018	-0.0512	-0.0443
		SVC	**-0.0300**	-0.0983	-0.0711	-0.0531
		MNB	-0.1311	**+0.0367**	**-0.0030**	**-0.0127**
		MLP	-0.1400	-0.0350	-0.0660	-0.0578
	Starscr.	KNN	+0.0459	-0.2605	-0.1542	-0.0835
		SVC	+0.0466	-0.2809	-0.2230	-0.1212
		MNB	-0.0474	**-0.1183**	**-0.0904**	**-0.0604**
		MLP	**+0.0562**	-0.1857	-0.1061	-0.0628
Partners	S. Core	KNN	-0.0318	-0.1445	-0.1004	-0.0708
		SVC	+0.0058	**-0.0774**	**-0.0489**	**-0.0332**
		MNB	**+0.0232**	-0.2214	-0.1646	-0.0903
		MLP	-0.0417	-0.1222	-0.0859	-0.0605
	Starscr.	KNN	**-0.0197**	-0.1986	-0.1214	-0.0772
		SVC	-0.0587	-0.1658	-0.1320	-0.0857
		MNB	-0.0932	-0.1714	-0.1462	-0.0924
		MLP	-0.0747	**-0.1061**	**-0.0851**	**-0.0642**
Starscream	S. Core	KNN	-0.2092	**-0.0401**	-0.1099	-0.1053
		SVC	-0.0040	-0.0699	**-0.0455**	**-0.0320**
		MNB	-0.0037	-0.1020	-0.0669	-0.0434
		MLP	**+0.0136**	-0.1073	-0.0570	-0.0378
	Partners	KNN	-0.2562	-0.0607	-0.1324	-0.1077
		SVC	**-0.0067**	-0.0395	-0.0192	-0.0213
		MNB	-0.0685	**+0.0115**	**+0.0001**	**-0.0083**
		MLP	-0.0420	-0.0854	-0.0597	-0.0490
All other projects	S. Core	KNN	+0.0095	-0.1296	-0.0766	-0.0531
		SVC	-0.0062	**+0.0196**	**+0.0210**	**+0.0077**
		MNB	+0.0113	-0.2139	-0.1598	-0.0891
		MLP	**+0.0354**	-0.0252	+0.0065	+0.0033
	Partners	KNN	**-0.0447**	-0.0355	-0.0308	-0.0271
		SVC	-0.0753	+0.0278	+0.0038	-0.0071
		MNB	-0.0934	**+0.0367**	**+0.0101**	**-0.0030**
		MLP	-0.1151	+0.0239	-0.0223	-0.0260
	Starscr.	KNN	**+0.0556**	-0.1632	-0.0722	-0.0430
		SVC	-0.0088	-0.0773	-0.0474	-0.0359
		MNB	-0.0266	-0.0918	-0.0635	-0.0444
		MLP	-0.0192	**+0.0090**	**+0.0082**	**-0.0012**

IV. RELATED WORK

Prior work has tackled the problem of misclassified issue reports. For example, Herzig et al. [3] found that 34% of bug reports are recorded with the incorrect label. Kochhar et al. [4] formulated the fine-grained issue reclassification problem, which aims to identify when issues are misclassified and which type should have been applied. Similar to the prior work, we set out to discover the correct type for issue reports; however, in our setting, issue type are rarely present (see DP2).

Other researchers have also trained classifiers to identify issue report types. For instance, Limsettho et al. [5] categorized bug-type issue reports using an unsupervised topic modeling and clustering approaches, observing that their unsupervised approach yields performance scores comparable to supervised ML-based approaches. Qin et al. [10] propose a bug classification method based on Long Short-Term Memory (LSTM), a recurrent neural network widely used for text classification tasks. In this paper, we make two novel contributions with respect to their prior work. First, to mitigate misclassification noise [3], we manually curate a sample of issue reports from these studied projects. Second, we apply four classification techniques in intra- and inter-project settings.

Previous work has also explored configuration-only bug-type classifiers, i.e., classifiers that can detect bug reports that describe configuration issues, based on their textual information [13]. Pingclasai et al. [9] applied topic modeling to three projects where they achieved F1-scores between 0.65–0.82. Our bug-type classifiers achieve similar F1-scores (0.6483–0.7573, ignoring the unstable KNN results).

V. CONCLUSIONS

Software analytics approaches rely on high-quality software process data. A shift towards a more flexible software process hinders the adoption of software analytics. To combat this, we propose the use of machine learning classifiers that can automatically identify bug-type issue reports for use in a software analytics context. Our experimental results indicate that such an approach shows promise (Observations 1 and 3), but standard model construction operations like rebalancing should be omitted to produce classifiers that are more suitable for deployment (Observation 2).

This paper is the result of an industrial-academic collaboration between a team of academics and members of the Shopify technical staff. It is the first of several ongoing steps towards the adoption of software analytics approaches at Shopify.

REFERENCES

[1] K. Beck, M. Beedle, A. Van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, et al. Manifesto for agile software development. 2001.

[2] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer. Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16:321–357, 2002.

[3] K. Herzig, S. Just, and A. Zeller. It's not a bug, it's a feature: how misclassification impacts bug prediction. In *Proceedings of the International Conference on Software Engineering*, pages 392–401, 2013.

[4] P. S. Kochhar, F. Thung, and D. Lo. Automatic fine-grained issue report reclassification. In *Proceedings of the International Conference on Engineering of Complex Computer Systems*, pages 126–135, 2014.

[5] N. Limsettho, H. Hata, A. Monden, and K. Matsumoto. Unsupervised bug report categorization using clustering and labeling algorithm. *International Journal of Software Engineering and Knowledge Engineering*, 26(07):1027–1053, 2016.

[6] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn. Bug localization using latent dirichlet allocation. *Information and Software Technology*, 52(9):972–990, 2010.

[7] S. McIntosh and Y. Kamei. Are Fix-Inducing Changes a Moving Target? A Longitudinal Case Study of Just-In-Time Defect Prediction. *Transactions on Software Engineering*, 44(5):412–428, 2018.

[8] M. B. Miles, A. M. Huberman, M. A. Huberman, and M. Huberman. *Qualitative data analysis: An expanded sourcebook*. Sage, 1994.

[9] N. Pingclasai, H. Hata, and K. Matsumoto. Classifying bug reports to bugs and other requests using topic modeling. In *2013 20th Asia-Pacific Software Engineering Conference (APSEC)*, pages 13–18, 2013.

[10] H. Qin and X. Sun. Classifying bug reports into bugs and non-bugs using lstm. In *Proceedings of the Asia-Pacific Symposium on Internetware*, pages 20:1–20:4, 2018.

[11] F. Rahman, D. Posnett, I. Herraiz, and P. Devanbu. Sample Size vs. Bias in Defect Prediction. In *Proceedings of the International Symposium on the Foundations of Software Engineering*, pages 147–157, 2013.

[12] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry. Improving bug localization using structured information retrieval. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 345–355, 2013.

[13] X. Xia, D. Lo, W. Qiu, X. Wang, and B. Zhou. Automated configuration bug report prediction using text mining. In *Proceedings of the Annual Computer Software and Applications Conference*, pages 107–116, 2014.

[14] J. Zhou, H. Zhang, and D. Lo. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *Proceedings of the International Conference on Software Engineering*, pages 14–24, 2012.