ISTANBUL TECHNICAL UNIVERSITY ★ GRADUATE SCHOOL

SOFTWARE DEFECT PREDICTION WITH A PERSONALIZATION FOCUS AND CHALLENGES DURING DEPLOYMENT

Ph.D. THESIS

Beyza EKEN

Department of Computer Engineering

Computer Engineering Programme

JANUARY 2022

ISTANBUL TECHNICAL UNIVERSITY ★ GRADUATE SCHOOL

SOFTWARE DEFECT PREDICTION WITH A PERSONALIZATION FOCUS AND CHALLENGES DURING DEPLOYMENT

Ph.D. THESIS

Beyza EKEN
(504142523)

Department of Computer Engineering

Computer Engineering Programme

Thesis Advisor: Asst. Prof. Dr. Ayşe TOSUN KÜHN

JANUARY 2022

İSTANBUL TEKNİK ÜNİVERSİTESİ ★ LİSANSÜSTÜ EĞİTİM ENSTİTÜSÜ

KİŞİSELLEŞTİRME ODAKLI YAZILIM HATA TAHMİNİ VE ENTEGRASYON ZORLUKLARI

DOKTORA TEZİ

Beyza EKEN
(504142523)

Bilgisayar Mühendisliği Anabilim Dalı

Bilgisayar Mühendisliği Programı

Tez Danışmanı: Dr. Öğr. Üyesi Ayşe TOSUN KÜHN

OCAK 2022

Beyza EKEN, a Ph.D. student of ITU Graduate School student ID 504142523 successfully defended the thesis entitled “SOFTWARE DEFECT PREDICTION WITH A PERSONALIZATION FOCUS AND CHALLENGES DURING DEPLOYMENT ”, which he/she prepared after fulfilling the requirements specified in the associated legislations, before the jury whose signatures are below.

Thesis Advisor :	**Asst. Prof. Dr. Ayşe TOSUN KÜHN** Istanbul Technical University	.....
Jury Members :	**Assoc. Prof. Dr. Yusuf YASLAN** Istanbul Technical University	.....
	Assoc. Prof. Dr. Gülfem ALPTEKİN Galatasaray University	.....
	Assoc. Prof. Dr. Feza BUZLUCA Istanbul Technical University	.....
	Assoc. Prof. Dr. Hasan SÖZER Özyeğin University	.....

Date of Submission : **23 December 2021**
Date of Defense : **21 January 2022**

To my family,

FOREWORD

First and foremost, I would like to thank my supervisor, Dr. Ayşe TOSUN, for being a helpful, supportive, and insightful mentor during my Ph.D. journey. It was a wonderful experience to work with a supervisor who always involves in her students' studies wholeheartedly and with her never ending motivation. I am also happy to be a part of the Software Modeling and Analysis research group that has very kind and brilliant people.

I also would like to thank my thesis committee members Dr. Gülfem ALPTEKİN and Dr. Yusuf YASLAN for their time and guidance during my thesis. I am also grateful to Dr. Ayşe BENER and Dr. Francis PALMA for their supervision and collaboration during my visit to Ryerson University, Canada.

This thesis was supported in part by the Scientific Projects Unit of Istanbul Technical University (Grant No: MOA-2019-42321), Scientific and Technological Research Council of Turkey (TUBITAK) and Ericsson Turkey (Grant no: 5170048), Mevlana Exchange Programme (Grant No: 258) by Council of Higher Education of Turkey (YÖK), and Collaborative Research and Development (Grant No: CRDPJ 499518-16) from NSERC, Canada. I also would like to thank Ass. Prof. Emad SHIBAB and Ass. Prof. Yasukata KAMEI for creating and sharing the dataset I used in part of my thesis.

Finally, I would like to thank my mother, father, and brothers for being there with their support during my academic studies and my entire life as well. It is so comforting and encouraging to know that my family walks with me through the challenging roads of life.

January 2022

Beyza EKEN
(Computer Engineer)

TABLE OF CONTENTS

ABBREVIATIONS

SQA : Software Quality Assurance
AI : Artificial Intelligence
ML : Machine Learning
SDLC : Software Development Life Cycle
SDP : Software Defect Prediction
IDE : Integrated Development Environment
RQ : Research Question
SLR : Systematic Literature Review
CDP : Class Data should be Private
CC : Complex Class
FD : Functional Decomposition
GC : God Class
SC : Spaghetti Code
LM : Long Method
OS : Organizational Silo
RS : Radio Silence
ML : Missing Links
PM : Personalized Model
SM : Selected General Model
GM : General Model
LR : Logistic Regression
NB : Naive Bayes
RF : Random Forest
ADTree : Alternating Decision Tree
ESD : Effect Size Difference
pd : Probability of detection
pf : Probability of false alarm
ROC : Receiver Operating Characteristic
AUC : Area Under ROC Curve
MCC : Matthews Correlation Coefficient
BScore : Brier Score
LOC : Lines of Code
ADD : LOC added in a commit
DEL : LOC deleted in a commit
CHURN : Churn of commit
NF : Number of modified files
ND : Number of modified directories
ENT : Entropy
NDEV : Number of developers had changed the modified files
NPC : Number of prior changes to modified file
AGE : The average time interval between the last and the current change

EXP : Experience of developer
REXP : Recent experience of the developer
SEXP : Experience of developer the on a subsystem
NNMF : Non-negative matrix factorization
LDA : Latent Dirichlet Allocation
SMOTE : Synthetic Minority Oversampling Technique
InfoGain : Information Gain
TT Gap : Train-Test Gap
UP : Update Period

LIST OF TABLES

LIST OF FIGURES

Page

SOFTWARE DEFECT PREDICTION WITH A PERSONALIZATION FOCUS AND CHALLENGES DURING DEPLOYMENT

SUMMARY

Organizations apply software quality assurance techniques (SQA) to deliver high-quality products to their customers. Developing defect-free software holds a critical role in SQA activities. The increasing usage of software systems and also their rapidly evolving nature in terms of size and complexity raise the importance of effectiveness in defect detection activities. Software defect prediction (SDP) is a subfield of empirical software engineering that focuses on building automated and effective ways of detecting defects in software systems. Many SDP models have been proposed in two decades, and current state-of-the-art models mostly utilize artificial intelligence (AI) and machine learning (ML) techniques, and product, process, and people-related metrics which are collected from software repositories.

So far now, the people aspect of the SDP has been studied less compared to the algorithm (i.e., ensembling or tuning machine learners) and data aspects (i.e., proposing new metrics). While the majority of people-focused studies incorporate developer or team related metrics into SDP models, recently personalized SDP models have been proposed. On the other hand, the majority of the SDP research so far now focuses on building SDP models that produce high rates of prediction performance values. Real case studies in industrial software projects and also the number of studies that research the applicability of SDP models in practice are relatively few. However, for an SPD solution to be successful and efficient, its applicability in real life is as important as its prediction accuracy. This thesis focus on two main goals: 1) assessing people factor in SDP to understand whether it helps to improve the prediction accuracy of SDP models, and 2) prototyping an SDP solution for an industrial setting and assessing its deployment performance.

First, we made an empirical analysis to understand the effect of community smell patterns on the prediction of bug-prone software classes. The “community smell” term is recently coined to describe the collaboration and communication flaws in organizations. Our motivation in this part is based on the studies that show the success of incorporating community factors, i.e., sociotechnical network metrics, into prediction models to predict bug-prone software modules. Also, prior studies show the statistical association of community smells with code smells (which are code antipatterns) and report the predictive success of using code smell-related metrics in the SDP problem.

We assess the contribution of community smells on the prediction of bug-prone classes against the contribution of other state-of-the-art metrics (e.g., static code metrics) and code smell metrics. Our analysis on ten open-source projects shows that community

smells improve the prediction rates of baseline models by 3% in terms of area under the curve (AUC), while the code smell intensity metric improves the prediction rates by 17%. One reason for that is the existing ways of detecting community smell patterns may not be rich in terms of capturing communication patterns of the team since it only mines patterns through mailing archives of organizations. Another reason is that the technical code flaws (code smell intensity metric) are more successful in representing defect related information compared to community smells. Considering the challenging situation in extracting community patterns and the higher success of the code small intensity metric in SDP, we direct our research to focus on the code development skills of developers and the personalized SDP approach.

Second, we investigate the personalized SDP models. The rationale behind the personalized SDP approach is that different developers tend to have different development patterns and consequently, their development may have different defect patterns. In the personalized approach, there is an SDP model for each developer in the team which is trained with the developer's own development history solely and its predictions target only the developer. Whereas in the traditional approach, there is a single SDP model that is trained with the whole team's development history, and its predictions target anyone in the team. Prior studies report promising results on the personalized SDP models. Still, their experimental setup is very limited in terms of data, context, model validation, and further explorations on the characteristics that affect the success of personalized models.

We conduct a comprehensive investigation of personalized change-level SDP on 222 developers from six open-source projects utilizing two state-of-the-art ML algorithms and 13 process metrics collected from software code repositories that measure the development activity from size, history, diffusion, and experience aspects. We evaluate the model performance using rigorous validation setups, seven assessment criteria, and statistical tests.

Our analysis shows that the personalized models (PM) predict defects better than general models (GM), i.e., increase recall by up to 24% for the 83% of developers. However, PM also increases the false alarms of GM by up to 12% for 77% of developers. Moreover, PM is superior to GM for those developers who contribute to the software modules that have been contributed by many prior developers. GM is superior to PM for the more experienced developers. Further, the information gained from various process metrics in prediction defects differs among individuals, but the size aspect is the most important one in the whole team.

In the third part of the thesis, we build prototype personalized and general SDP models for our partner from the telecommunication industry. By using the same empirical setup that we use for the investigation of personalized models in open-source projects, we observe that GM detects more defects than PM (i.e., 29% higher recall) in our industrial case. However, PM gives 40% lower false alarms than GM, leading to a lower code inspection cost than GM.

Moreover, we observe that utilizing multiple data sources such as semantic information extracted from commit descriptions and latent features of development activity and applying log filtering on metric values improve the recall of PM by up to 25% and lowers GM's false alarms by up to 32%. Considering the industrial team's perspective

on prediction success criteria, we pick a model to deploy that produces balanced recall and false alarm rates: the GM model that utilizes the process and latent metrics and log filtering. Also, we observe that the semantic metrics extracted from the commit descriptions do not seem to contribute to the prediction of defects as much as process and latent metrics.

In the fourth and last part of the thesis, we deploy the chosen SDP prototype into our industrial partner's real development environment and share our insights on the deployment. Integrating SDP models into real development environments has several challenges regarding performance validation, consistency, and data accuracy. The offline research setups may not be convenient to observe the performance of SDP models in real life since the online (real-life) data flow of software systems is different than offline setups. For example, in real life, discovering bug-inducing commits requires some time due to the bug life cycle, and this causes a data label noise in the training sets of an online setup. Whereas, an offline dataset does not have that problem since it utilizes a pre-collected batch dataset. Moreover, deployed SDP models need a re-training (update) with the recent commits to provide consistency in their prediction performance and to keep up with the non-stationary nature of the software.

We prose an online prediction setup to investigate the deployed prototype's real-life performance under two parameters: 1) a train-test (TT) gap, which is a time gap between the train and test commits used to avoid learning from noisy data, and 2) model update period (UP) to include the recent data into the model learning process.

Our empirical analysis shows that the offline performance of the SDP prototype reflects its online performance after the first year of the project. Also, the online prediction performance is significantly affected by the various TT gap and UP values, up to 37% and 18% in terms of recall, respectively. In deployment, we set the TT gap to 8-month and UP to 3-day, since those values are the most convenient ones according to the online evaluation results in terms of prediction capability and consistency over time.

The thesis concludes that using the personalized SDP approach leads to promising results in predicting defects. However, whether PM should be chosen over GM depends on factors such as the ML algorithm used, the prediction performance assessment criteria of the organization, and developers' development characteristics. Future research in personalized SDP may focus on profiling developers in a transferable way instead of building a model for each software project. For example, collecting developer activity from public repositories to create a profile or using cross-project personalized models would be some options.

Moreover, our industrial experience provides good insights regarding the challenges of applying SDP in an industrial context, from data collection to model deployment. Practitioners should consider using online prediction setups and conducting a domain analysis regarding the team's practices and prediction success criteria and project context (i.e., release cycle) before making deployment decisions to obtain good and consistent prediction performance. Interpretability and usability of models hold a crucial role in the future of SDP studies. More researchers are becoming interested in such aspects of SDP models, i.e., developer perceptions of SDP tools and actionability of prediction outputs.

KİŞİSELLEŞTİRME ODAKLI YAZILIM HATA TAHMİNİ VE ENTEGRASYON ZORLUKLARI

ÖZET

Yazılım sistemlerinin hayatımızın hemen her alanındaki yeri giderek artarken, hızlı gelişen doğaları gereği tipik bir sistemin büyüklüğü ve karmaşıklığı da zaman içerisinde hızla artabilmektedir. Dolayısı ile organizasyonlar tarafından uygulanan yazılım kalite güvencesi (YKG) aktivitelerinin karmaşıklığı da artmaktadır. Organizasyonların en önemli hedeflerinden biri kullanıcılara hatasız ve kaliteli bir yazılım sunabilmektir. Bu kapsamda en çok bütçe ve zaman ayırdığı aktivitelerden biri yazılımdaki hataların giderilmesidir. İçinde bulunduğumuz yapay zeka (YZ) çağında yazılım mühendisliği alanındaki araştırmacılar yazılım hatalarının tespitini otomatize eden, akıllı öneri sistemleri modellemeye odaklanmıştır.

Bir yazılım hata tahmini (YHT) modeli tipik olarak, yazılım depolarından projeye dair geçmiş kodlama ve hata aktivitelerinin örüntülerini makine öğrenmesi (MÖ) yöntemleri ile öğrenir ve ekibin yeni geliştirdiği yazılım modüllerinin (örn., yazılım sınıfı, kod çevrimi) yazılım sisteminde bir hataya sebebiyet verme durumuna dair bir tahminde bulunur. Bu modeller yazılım geliştirme ortamının geliştirme, test ve kod inceleme gibi aşamalarına entegre edilerek ekibin riskli yazılım modüllerini daha kolay tespit edebilmesi ve karar verme mekanizmasını desteklemek için kullanılır. Örneğin kod inceleme için kısıtlı süresi olan bir ekip öncelik stratejisini YHT modelinin tahminlerine göre belirleyebilir.

Araştırmacılar yaklaşık 20 senedir farklı açılardan yaklaşarak YHT modellerinin daha iyi tahmin performansına ulaşması için çeşitli yöntemler önermektedirler. Bu yöntemlerin çoğu probleme algoritmik (örn., MÖ algoritmalarının sonuçlarını birleştirmek, hiper-parametre kestirimi uygulamak) yada veri açısından yaklaşmaktadır (örn., yeni ölçütler önermek).

Yazılım süreçlerinin ayrılmaz bir parçası olan kişi faktörünü modelleyen çalışmalar ise nispeten daha az sayıdadır. Kişi faktörünü ele alan çalışmalar ise ikiye ayrılmaktadır. Çoğunlukta olan ilk tip çalışmalar ekipteki yazılım geliştiricilerine dair bazı ölçütler elde ederek model eğitimi sırasında kullanmıştır. Bu ölçütlerin arasında bir yazılım modülüne katkıda bulunan geliştirici sayısı, bir geliştiricinin odağı (aynı anda değişiklik yaptığı modül sayısı), geliştiricinin tümleşik geliştirme ortamı ile olan etkileşimi (ne kadar süre bir modülde gezindiği) ve yazılım ekibinin birbirleri ile olan sosyo-teknik etkileşimleri bulunmaktadır. İkinci grup çalışmalar ise sayıca çok az olmakla birlikte (sadece iki) kişiselleştirilmiş YHT modellerine odaklanmıştır. Kişiselleştirilmiş YHT modelleri kişi faktörünü model eğitimine dahil etmek yerine, ekipteki her bir geliştirici için ayrı ve kişisel bir model kurmayı kapsamaktadır.

Bunların yanı sıra, literatürdeki çalışmaların çoğu son yıllara kadar yüksek tahmin performanslı YTH modelleri kurmaya odaklanmıştır. Fakat bir tahmin modelinin başarılı ve efektif olabilmesi için tahmin modellerin hata tahmin doğruluklarının yanı sıra gerçek hayata uygulanabilirlikleri, tahmin performanslarının sürdürülebilirliği, tahminlerin kullanıcı perspektifinden yararlı ve güvenilir olması da önemlidir. YHT modellerinin pratik uygulanabilirliğine ve gerçek hayat senaryolarına odaklanan çalışmaların sayısı azdır ve üretilen YHT çözümlerinin gerçek hayat senaryolarındaki değerini değerlendirebilmek için bu çalışmalara ihtiyaç vardır.

Biz bu tezde iki ana amaca odaklandık: 1) kişi faktörünün YHT modellerinin tahmin başarısına olan etkisi, ve 2) endüstriyel bir yazılım projesi için bir YHT çözümü geliştirmek ve gerçek geliştirmek ortamına entegre etmek.

Tezin ilk analizinde, topluluk kokularının hata tahminine olan etkisini inceledik. "Topluluk kokusu" ekibin doğru iş birliği ve iletişim pratiklerini uygulamayı ertelediği durumlara ithafen ortaya atılmış bir kavramdır. Örneğin iki kişinin aynı yazılım modülünde çalışması fakat aralarında direk bir iletişim bağlantısı olmaması durumu bir topluluk kokusu olarak ele alınmaktadır. Çünkü bu durum işin çift yapılmasına ve kod tekrarı denilen istenmeyen bir duruma sebebiyet verebilmektedir. Ekipteki böyle kusurlu iletişim ve etkileşim pratiklerinin ekstra masrafa ve gecikmeye yol açabildiği gözlemlenmiştir. Ayrıca önceki çalışmalar topluluk kokularının, "kod kokusu" terimi ile tarif edilen kusurlu kodlama pratikleri ile aynı yazılım modüllerinde oluştuklarını göstermiştir. Dahası kod kokusu içeren yazılım sınıflarının hata içermeye daha meyilli olduğu gösterilmiştir. Bu duruma ve sosyo-teknik faktörleri YHT modellerine entegre ederek başarılı tahmin performanslarına ulaşan diğer çalışmalardan motivasyonumuzu alarak biz de topluluk kokularının YHT modellerine eğitimine dahil ederek hatalı yazılım sınıflarının tahminine olan etkisini deneysel olarak inceledik.

Deneysel çalışmamızda topluluk kokularının hata tahminine olan katkısı kod kokusu ölçütlerinin katkısı ile karşılaştırılmıştır. Çalışmada temel model olarak ise son teknoloji ölçütleri (statik kod ölçütleri, nesneye yönelimli ölçütler, bir dosyada çalışan geliştirici sayısı ve geliştiricinin odağı) kullanan YHT modelleri baz alınmıştır. Sonuçlar göstermiştir ki eğri altında kalan alan kriterine göre topluluk kokuları temel modellerin hata tahminini başarısını %3, kod kokusu yoğunluğu ölçütü ise %17 artırmaktadır.

Sonuç olarak bu analizde topluluk kokularını YHT model eğitimine dahil etmenin tahmin başarısını artırdığı fakat kod kokusu yoğunluğu ölçütü kadar tahmine katkısının olmadığı gözlemlenmiştir. Ayrıca veri toplama aşamasındaki deneyimlerimiz topluluk kokularını ölçmenin diğer metrikleri ölçmeye göre daha çok efor ve kaynak gerektirdiğini gösterdi. Mevcuttaki kullanılabilir araçlar iletişim örüntülerini sadece mail arşivlerinin analizi ile çıkarabilmektedir. Fakat her proje için takımın mail arşivlerine erişim olmayabilmektedir. Hatta organizasyonel iletişim mail haricinde bir araç ile yapılabilmekte ve organizasyonun iletişim örüntülerini yakalamak mümkün olmayabilmektedir. Bu durum bizi topluluk kokularını model eğitimine dahil etmek yerine sadece koddan ölçülen kod geliştirme becerileri bilgilerini kullanarak kişisel modeller oluşturmaya yönlendirmiştir.

Tezin ikinci analizinde kişiselleştirilmiş YHT modellerinin hata tahmini performansına olan katkısını geniş kapsamlı olarak inceledik. Bu analizimizin arkasındaki

motivasyonlardan biri kişi odaklı öneri yaklaşımlarının, arama motorları, sosyal medya siteleri, ve e-ticaret sistemleri gibi bir çok sistemde başarılı olması. Bir diğer motivasyonumuz ise ekipteki geliştiriciler kendilerine özgü geliştirme stillerine ve deneyimlere sahip olabilmesi ve bu durumun farklı geliştiricilerin geliştirdiği yazılım parçalarının farklı hata örüntülerine sahip olmasına sebep olabilmesidir.

Kişilerin farklı hata örüntülerine sahip olmasına dayanarak, daha başarılı tahmin skorları sağlayabilmek adına, kişiselleştirilmiş YHT modelleri ortaya atılmıştır. Fakat önceki çalışmalar bir çok açıdan sınırlı kalmıştır. Örneğin veri setlerinin küçük olması, alandaki son model ölçütlerin çalışmaya dahil edilmemiş olması gibi sebeplerden ötürü bulgularının geçerliliği kısıtlıdır. Biz bu tezde ise çok daha zengin bir deneysel ortam kurarak kişisel modelleri geleneksel modeller ile karşılaştırmalı olarak etraflıca inceledik. Geleneksel yaklaşım, tüm ekibin geçmiş geliştirme aktiviteleri üzerinden eğitilmiş ve ekipteki herkes tarafından kullanılabilen tek bir YHT modeli geliştirmeyi kapsamaktadır. Kişiselleştirilmiş yaklaşım ise, ekipteki her bir geliştiriciye bireysel bir YHT modeli kurulmasını ve modellerin sadece ilgili kişinin geçmiş aktivitesi ile eğitilmesini kapsamaktadır.

Tezin bu bölümünde geliştirilen YHT modelleri yazılım değişikliği seviyesinde çalışmaktadır. Değişiklik seviyesindeki YHT sistemlerinde, yazılımda bir geliştirme yapılarak kod deposuna çevrim (commit) olarak gönderildiğinde tahmin modeli devreye girer ve ilgili geliştiriciye yaptığı çevrimin yazılım sistemine hata enjekte ediyor olma ihtimaline dair geri bildirim verir. Bu tip YHT modellerinin avantajları ise yapılan değişiklik hafızada taze iken geliştiriciye anında geri bildirim vermesidir.

Modellerin eğitimi sırasında yazılım depolarından toplanmış geçmişe dönük yazılım çevrimleri (commit) ve bu çevrimlerin yazılıma hata enjekte edip etmediği bilgisi kullanılmıştır. Ayrıca, toplanan çevrimlerden yazılım geliştirme sürecini tarihsel (örn., daha önce bir kaynak kod dosyasının kaç kere değiştiği), boyutsal (örn., kaynak kod dosyasında değiştirilen kod satır sayısı), dağılımsal (örn., ilgili çevrimin kaç dosyayı değiştirdiği) ve geliştirici tecrübesi (örn., önceki çevrimlerin sayısı) açısından yansıtan 13 adet ölçüt eğitimde kullanılmıştır. YHT modelleri bu alanda son teknoloji olan Naive Bayes (NB) ve Random Forest (RF) algoritmaları ve 10-katlı çapraz-doğrulama tekniği kullanarak eğitilmiştir. Eğitilen modellerin performans değerleri literatürde iyi bilinen yedi farklı kriter ile ölçülmüştür: hata yakalama olasılığı, yanlış alarm olasılığı, kesinlik, F1-skoru, eğri altında kalan alan, Matthews korelasyon katsayısı ve Brier Skoru.

Altı adet açık kaynak kodlu projenin toplam 222 geliştiricisi üzerinde yapılan analiz gösterdi kişiselleştirilmiş modellerin hata yakalama olasılığı, geleneksel modellerinki ile karşılaştırılınca geliştiricilerin çoğunluğu (222 kişiden 184'ü) için %23'e kadar artırmaktadır. Aynı zamanda, geleneksel yerine kişisel yaklaşım tercih etmek 174 geliştirici için yanlış alarm olasılığını %12'ye kadar artırmıştır.

Önceki çalışmalardan farklı olarak bu çalışma, bir geliştirici için kişisel modelin geleneksel modelden daha başarılı olmasına yada tam tersine vesile olan geliştirme karakteristiklerini incelemiştir. Sonuçlara göre, daha önce çok fazla geliştiricinin üzerinde çalıştığı yazılım modüllerinde katkıda bulunan geliştiricilerin hatalarını kişisel model daha iyi tahmin etmekte iken, projedeki tecrübesi fazla olan geliştiricilere geleneksek model daha doğru tahminler üretmiştir.

Bununla birlikte, hata tahmininde önemli olan ölçütlerin kişiden kişiye ne ölçüde değiştiği incelenmiştir. Sonuçlara göre, farklı kişiler için eğitilmiş kişisel modeller tecrübe, dağılım, ve tarihsel ölçütlerden farklı oranlarda yararlanmaktadırlar. Fakat yine de büyük çoğunluğu için en önemli ölçüt bir çevrimdeki eklenen satır sayısıdır.

Tezin üçüncü analizinde ise kişiselleştirilmiş YHT yaklaşımını bir akademi-sanayi iş birliği projesinde birlikte çalıştığımız sanayi partnerimizin seçilmiş bir yazılım projesi kapsamında değerlendirdik. Bu seçilmiş projeden elde edebildiğimiz altı geliştirici verisi üzerinde kurulan kişisel ve geleneksel modellerin karşılaştırılması gösterdi ki, geleneksel modeller hata yakalama olasılığı açısından daha başarılılar (%29 daha yüksek). Fakat aynı zamanda kişisel modellere göre daha çok yanlış alarm üretmektedirler (%40 daha yüksek). Yüksek yanlış alarm değerleri geliştiricileri gereksiz yere uyararak daha fazla kod inceleme eforuna sebep olabilmektedirler.

Endüstriyel projeye ait veri kümesinin küçük ve tipik olarak gürültü içermesi sebebiyle çeşitli istatistiksel yaklaşımlar aracılığı ile mevcuttaki veri kaynaklarından daha çok bilgi ölçmenin YHT modellerine katkısı olup olmayacağı incelenmiştir. Matris ayrıştırma ile elde edilen gizli ölçütlerin eğitime dahil edilmesi ve logaritmik filtrelemenin ölçütler üzerinde kullanılması geleneksel modellerin yanlış alarm değerlerini %32 değerinde düşürmüştür. Bunlara ek olarak kod çevrim açıklamalarından elde edilen semantik bilgilerin de eğitime eklenmesi kişisel modellerin hata yakalama olasılığını %25 artırmıştır.

Aynı zamanda ölçütlerin kişisel ve geleneksel modellerin hata tahminine olan katkıları incelenmiş ve iki yaklaşım için de benzer katkılara sahip oldukları gözlemlenmiştir. En çok yazılım süreci ölçütleri katkıda bulunurken, gizli ölçütler ikincil öneme sahiptir. Semantik ölçütlerin ise hata tahminine diğerleri kadar katkıda bulunmadığı gözlemlenmiştir.

Analizler göstermiştir ki kişisel modeller geleneksellere üstün gelebilmekte iken, pratikte hangi yaklaşımın tercih edileceği, geliştiricinin yazılım geliştirme karakteristiği, performans ölçme kriterleri birçok faktöre bağlıdır. Hatta geliştiricinin ekibe yeni katılması ve/veya kendisine bir tahmin modeli kurulabilecek kadar geliştirme geçmişine sahip olmaması da literatürde soğuk başlangıç problemi olarak bilinen, tercihi zorunlu olarak geleneksel modele yönlendirecek bir durumdur. Bu durum bize modellerin araştırma ortamındaki ulaştığı başarı değerleri kadar uygulanabilirliğinin de önemli olduğunu göstermiştir.

Tezin dördüncü ve son analizinde ise endüstriyel partnerimizin projesi için prototiplenen YHT çözümünün gerçek geliştirme ortamına entegre edilmesine odaklanmış ve buna dair tecrübelerimiz paylaşılmıştır. Akıllı modelleri gerçek hayata entegre etmenin bir araştırma ortamında model prototiplemeye kıyasla kendine has bambaşka zorlukları vardır. Bu yüzden çevrimiçi bir değerlendirme ortamı kurularak modelin tahmin performansının gerçek hayata uygun bir analizinin yapılması ve modelin kullanışlılığının değerlendirilmesi önem kazanmaktadır.

Model prototipleme yapılan araştırma ortamında (çevrimdışı ortam) kullanılan önceden toplanılmış veri kümesinde hataya sebebiyet veren kod çevrim bilgileri halihazırda mevcuttur ve modelin eğitimi için direk erişilebilmektedir. Fakat gerçek geliştirme ortamında hayatta (çevrimiçi ortam) anda yapılan kod çevrimlerinin hataya

sebebiyet verip vermediği bilgisi gecikmeli (seçilen proje bazında ortalama yedi ay) olarak elde edilmektedir ve veride gürültüye sebebiyet vermektedir. Bu gecikme kod değişikliği bazında dizayn edilen YHT modellerinde yazılım hata yaşam döngüsünden kaynaklanan tipik bir durumdur.

Gerçek geliştirme ortamına entegre edilen YHT modelinin sürdürülebilir bir tahmin performansına sahip olması önemlidir. Bu sebeple yazılım projelerinin hızlı değişen ve gelişen yapısına ayak uydurarak son geliştirme aktiviteleri ile yeniden eğitilerek güncellenmek zorundadır. Ayrıca model eğitim/güncelleme sırasında eğitim kümesindeki gürültülü verinin (henüz tespit edilmemiş fakat hataya sebebiyet veren kod çevrimleri) tahmin performansına etkisinin minimize edilmesi gerekmektedir. Modelin eğitiminde kullanılan kod çevrimleri ile hataya meyilleri tahmin edilecek çevrimler arasında zamansal bir aralık bırakmak modelin gürültülü veriden öğrenmesine karşı alınan tipik bir yöntemdir.

Biz bu çalışmada, modelin gerçek hayattaki performansını simüle etmek için bir çevrimiçi tahmin ortamı tasarladık. Öncelikle çevrimdışı tahmin ortamındaki (prototipleme ortamı) performans değerlendirmelerimizin çevrimiçi tahmin ortamındaki (gerçek hayattaki) performanslarını yakalayıp yakalayamayacağını araştırdık. Sonuçlar gösterdi ki, modelin çevrimiçi değerlendirme performansı projenin ilk senesinde düşükken sonrasında çevrimdışı performansı ile benzer değerlere ulaşıyorlar. Bu da çevrimdışı değerlendirme tekniğinin modelin gerçek hayattaki potansiyel performansını yansıtabildiğini göstermektedir.

Daha sonra çevrimiçi tahmin dizaynımızı kullanarak entegre edilecek modelin güncellenirken kullandığı eğitim kümesi ile hataya meyilleri tahmin edilecek kod çevrimleri arasında bırakılan farklı uzunluktaki zaman aralıklarının ve farklı model güncelleme periyotlarının tahmin performansına etkisini araştırdık. Sonuçlarımıza göre, modelin eğitimi sırasında eğitim ve tahmin kod çevrimleri arasında bırakılan zaman aralığı modelin hata yakalama olasılığını %37'ye kadar etkilemektedir. Modelin güncelleme periyodunu ayarlamak ise hata yakalama olasılığını %18'e kadar etkilemektedir. Bu sonuçlara göre, sekiz aylık bir eğitim-tahmin kümesi zaman boşluğu kullanımı ile modelin üç günde bir güncellenmesi en başarılı ve istikrarlı performans değerlerini üreten durumdur ve modelin entegrasyon ayarları bu şekilde yapılmıştır.

Ayrıca tahmin çıktılarının daha anlaşılır ve aksiyon alınabilir olması için bir kod çevriminin yazılıma hata enjekte etme olasılığına ek olarak ilgili kod çevriminden hesaplanan ölçütler de (örn., çevrimden etkilenen kod dosyası sayısı) kullanıcıya sunulmuştur.

Bu tezdeki analizler göstermiştir ki kişiselleştirilmiş YHT modellerinin geleceği parlak olmakla birlikte, performanslarını etkileyen faktörler üzerinden daha çok analize ihtiyaç vardır. Ayrıca gelecek çalışmalar transfer edilebilir profiller oluşturmaya ve benzer özelliklerine göre segmentlenmiş geliştiricilere özel grup modelleri geliştirmeye odaklanabilir. Diğer taraftan YHT modellerinin pratiğe uygulanabilirliği ve tahmin sonuçlarının yorumlanabilirliği önemlidir. Bir yazılım projesine YHT modeli geliştirilirken çevrimiçi bir değerlendirme ortamının kurulmasını desteklemekteyiz. Ayrıca projenin yapısına, takımın geliştirme alışkanlıklarına ve tahmin modelinden beklentilerine dikkat edilmeli.

1. INTRODUCTION

Software systems have become essential parts of our modern society, and they continue diffusing into many parts of our lives [1]. Assuring and maintaining the quality of such systems are critical needs of today's organizations. However, the fast evolving nature of software and their increasing size and complexity complicate the software quality assurance (SQA) activities of organizations (i.e., planning, software testing, defeating bugs) [2,3].

Meanwhile, SQA practices of development teams and researchers also evolve to "keep up with the fast pace" of software development [2]. In the era of artificial intelligence (AI) and machine learning (ML), both academic and business communities have proposed and integrated many intelligent systems into the software development life cycle (SDLC) [4]. Such intelligent models provide insights and recommendations to people on software products and processes to make SQA activities more effective in terms of budget and time [5]. Automatic code generation [6], bug fixing [7], test case generation [8], and bug triage detection tools [9] are a few samples of such intelligent models.

Software defect prediction (SDP) has become a well-established field of study for decades and it has still been one of the focus areas of empirical software engineering community for SQA. Intelligent defect predictors enable the detection of defect-prone (i.e., buggy or fault-prone) software modules in a more robust and quicker way when compared to manual inspection [10,11]. Shull et al. [12] report that manual inspection of source codes can catch nearly 60% of the bugs, whereas the proposed SDP models could perform around 70% in terms of the probability of bug detection [13]. Moreover, these models help to reduce the effort spent on code inspection: 35% of all predicted defect prone code changes could be identified by spending only the 20% of the total inspection effort. Organizations are highly interested in SDP models since they prefer

to deliver higher quality products to their customers while reducing the development and maintenance effort.

1.1 An Overall View of SDP Models

A typical SDP model is trained with AI/ML algorithms by utilizing a set of software metrics collected from software archives to provide development team feedback on the defect-proneness of software modules [14]. Figure 1.1 shows the typical process of SDP model building.

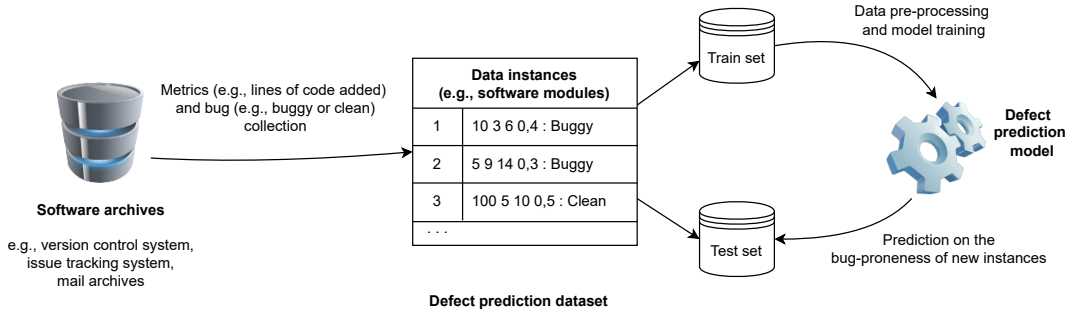

Figure 1.1 : An overview of a typical model building methodology.

The first step of building an SDP model is data collection. Depending on the prediction level, i.e., code change-level or software file or class or method level, a set of historical metrics are collected from software archives. Collected metrics represent valuable information regarding the software product (i.e., the complexity of source code [15]), the process (i.e., the many times a software module is changed priorly [16]), or the people involved in the development (i.e., the number of developers contributed to a software module [17]). The most commonly used software archives for metric collection are source code versioning and bug tracking tools. But information sources are not limited to these, another example is the use of mail archives to capture community communication patterns (see Chapter 2).

The majority of SDP studies utilize supervised learning strategies to train predictors [11]. Therefore, in addition to the metrics extracted from software modules, the bug-proneness information of those software modules is also required during SDP model training. A bug-prone software module means that the module includes a bug and in the case of software changes, a change is accepted as bug-inducing if it

injects a bug into the software system. Bug-proneness of a module is often represented with a binary label, i.e., bug-prone or clean, however, there are studies that utilize the severity of bugs a module includes [18]. Predictions that an SDP model make is also either at binary level or various severity levels depending on the used bug-proneness information during training.

Defect predictors could be designed to make prediction at different levels of software modules. A software module is often a method, class, file, package, or software change (commit). The choice of the prediction level (software module) of an SDP model depends on the project context, data availability or programming language [19]. Recently, an increasing rate of studies focuses on software change-level SDP, which is also called "just-in-time" SDP by researchers, e.g., in [20]. Using a change-level prediction model allows a developer to receive feedback on her/his changes in a much shorter time when compared to a file or class level prediction model. An advantage of this quick feedback is that developers could review their code when the change is still fresh in their minds [16]. The prediction models that we designed in this thesis also make predictions at change-level. Further knowledge on change-level SDP could be found in the following chapters, particularly Sections 3.2 and 5.2.1 report literature review on change-level studies.

The next step after data collection is often applying some AI/ML techniques to pre-process data, i.e., data sampling, and to build/train SDP models. A common challenge in SDP studies is the imbalanced nature of datasets due to fewer defect-prone software modules compared to clean (non-defective) modules. Therefore, a typical SDP dataset is pre-processed by applying data sampling techniques to balance the number of data instances belongs to clean and defect-prone categories. Many ML algorithms have been used to train an SDP model. Widely used and well performed algorithms in SDP field are Logistic Regression [21,22], Naive Bayes [13,23] and tree based algorithms such as Decision Tree and Random Forest [24]–[26].

Typical model validation strategies commonly used in SDP studies are cross-validation [27] and time-sensitive strategies [28]. In a cross-validation strategy, the dataset is split into k subsets, and each one of the k subset is used as test set once, and while

the other $k-1$ subsets are used as training set. On the other hand, time-sensitive validation strategies consider the temporality of software code changes, whereas the cross-validation strategy does not. Therefore, a time-sensitive strategy is more realistic since it is designed to predict defects in the future software modules (at time $t+1$) by learning from the historical defect information of prior changes on software modules (until time t). Furthermore, online validation strategies are proposed for change-level SDP that mimic the real-life data flow in a more realistic way compared to other model validation strategies [29]. One of those is called online prediction which is also a time-sensitive learning strategy, but it also considers the actual labels of the training instances determined based on the real-life discovery time of defects. More discussion and knowledge regarding validation approaches are reported in Chapters 3 and 5.

1.2 Modeling People in SDP

People are one of the most important factors of software development as it is a human-intensive engineering discipline, in which the final product quality and the implementation of processes are heavily dependent on prior experience, knowledge and skills of teams.

So far now, SDP studies have incorporated people factor into the SDP model training process in various ways. One way of modeling people in SDP is measuring developer related information to be used during model training as an indicator of defect proneness of a software module. The most used developer metrics in the field are the number of different developers that modified the code files [17] and the developer experience on software modules [30]. Also, there are more sophisticated metrics, e.g., Posnett et al. [31] suggest using developer focus on software modules. The focus of a developer is computed over the distribution of the developer's contribution to the modules that the developer modifies. Lee et al. [32] use micro-interactions of developers with the Integrated Development Environment (IDE), such as the time/duration a module is browsed by the developer, to predict defects. Çalıklı et al. [33] analyze the relationship of confirmation bias of developers with the injected defects. The authors support the fact that a tester may have a tendency to make the code successfully execute rather than to look for a strategic approach to make it fail.

Considering that the socio-technical structure of the organizations has an effect on the quality of the developed software product, some studies suggest to use socio-technical metrics to predict defects [22,34]. Recently, "Community smell" term is coined by Tamburri et al. [35] to describe the community flaws occurred in organizations due to missing communication and collaboration links. Community flaws may cause inconvenience in development activities such as duplication of work piece or extra cost in development. More explanation on this is in Chapter 2.

Another way of modeling people in SDP is building personalized models. Personalized SDP studies include people factor into the model building strategy and propose separate prediction models for each developer in the team.

1.2.1 Personalized models

Developers in software development teams can represent different development characteristics, i.e., different commit frequencies, different level of experience in the project or on specific software modules, and hence the defect injection patterns of these developers also have different patterns [17,36].

Recommendation systems in other fields, e.g., search engines [37], social media platforms [38], customize their outputs based on users' characteristics such as prior preferences. Thus, these businesses provide more personal and satisfying results for their customers [39].

SDP field also incorporates personalization into the prediction models to provide more accurate and satisfying feedback to developers on their code changes. The rationale is that developers have different defect patterns and personalized models would capture defects more accurately. Jiang et al. [36] propose personalized SDP models that learn from each developer's own historical development activity, while traditional models learn from the whole team's development. The personalized SDP approach builds separate SDP models for everyone in the team, if development change history is available, instead of using a single general model as in the traditional approach.

1.3 Applicability of SDP into Practice

Studies building recommendation systems emphasize the differences between offline and online evaluations [40,41]. The offline evaluation refers to the research environment where some prototypes are built with various algorithms and model building approaches, and assessed using a pre-collected (offline) dataset. In contrast, online evaluation refers to training and assessment of the models in a real-life environment that includes real users and data. The majority of the SDP studies work in offline setups to focus on the prediction performance of model prototypes. Offline studies often adopt new ways of model learning by utilizing various statistical and ML approaches [11] or knowledge extraction ways from data [18]. Using offline setups is an effective and easy-to-use way to compare and assess various prediction models. However, the practical applicability of the SDP models and the interpretability of recommendations are as important as the models' prediction performance [42]. SDP studies that focus on the practical usability of proposed models are relatively few, but there is an increasing interest in adopting SDP models into the real-life development environment [29,43]–[46]. These studies have aspects such as prediction performance of SDP models in a real development environment and the consistency of prediction accuracy, interpretability of prediction outputs, and industrial domain challenges.

The behavior of prediction models may not be similar in research and real-life environments since both have different data characteristics. The studies in an offline environment often use pre-collected and labeled data, whereas the real-life data requires more careful investigation from the perspective of SDP. For instance, the discovery of bug-inducing software changes requires more time in a real development environment [29]. This delay causes a label noise in online environments, which is not an issue in an offline setup as it uses a batch dataset. Tan et al. [29] propose using an online prediction setup to simulate the real life data flow of the software systems in order to assess the model's prediction accuracy more realistically.

Considering the perspective of real users is the second crucial factor for software recommendation systems [40]. Since an SDP model aims to support development teams during SQA activities, the prediction outputs should be trustable, interpretable,

and actionable [29,46]. To address that, studies in the SDP field focus on enriching the output with informative clues such as the metric values of commits and the importance of metrics presented to developers to support the code reviewing process [29,46].

Another challenge regarding the practical applicability of SDP models is the assessment of models in different domains. The number of reported SDP studies focus on industrial contexts is relatively low compared to the number of studies conducted in open-source projects [2]. The reason for that is that open-source projects are publicly and freely available to researchers, whereas industrial software projects are not accessible easily. Conducting research in an industrial context often requires collaboration with a company. While datasets of some industry case studies are shared in public software data repositories [47], many companies do not share their dataset due to confidentially reasons [2].

Nevertheless, conducting research in industrial domains is very valuable since it provides assessing the generality of research findings and the practical applicability of produced models into the cases that involve real customers [2,48]. SDP studies focusing on various industrial domains, e.g., communication [29], e-commerce [44], automotive [43], maritime [45], are significant resources for both researchers and practitioners to comprehend the real development environments.

1.4 Purpose of the Thesis

In this thesis, we have two main goals: 1) assessing the people factor in SDP to understand whether it helps to improve the prediction accuracy of SDP models and 2) prototyping an SDP solution for an industrial setting and assessing its deployment performance. In this section, we explain these two goals in detail, respectively, with the motivation behind them.

Studies in the SDP field have so far investigated this modeling problem in two ways, oriented around the algorithm or the data . While some focus on the algorithm aspect to improve the accuracy of prediction models, i.e., incorporating various AI/ML and statistical approaches [11], some other studies focus on the data aspect to enrich the knowledge extracted from software data [18].

Menzies et al. discuss the "ceiling effect" on the performance of SDP models [49]: performance of SDP models that utilize the static code attributes has reached a ceiling, and researchers need to enrich the information gathered from data to improve the performance of SDP models. Also, Lessmann et al. [50] show that 17 out of 22 ML algorithms widely used by the SDP researchers report statistically the same performance in predicting defects. Both of these findings imply that future studies need more focus on enhancing information collected from the elements of software development, i.e., product, process, and people, to build more accurate SDP models.

SDP studies have focused on product and process metrics for a long time [19]. Studies also state tuning the parameters of ML algorithms and focusing on specific defect types [27,51] to improve the prediction accuracy of prediction models [52]. However, despite being one of the key factors in development, the people aspect in SDP has been studied relatively little. As we mentioned in Section 1.2, while the majority of people-focused SDP studies work on people metrics, i.e., measuring people development patterns to incorporate them as features into the model learning process, only two ([36] and [53]) focus on personalized SDP models, i.e., building separate models for each team member. Therefore, we incorporate the people aspect in SDP and investigate the effect of people aspect on the prediction performance. We modeled people in two ways: 1) incorporate the socio-technical factors into the model learning process of an SDP model, 2) build separate models for developers using the personalized approach.

First, we assess the effect of socio-technical factors, more specifically the "community smells" on the prediction of defect-prone software classes. "Community smell" is a term that was recently coined in the software engineering community and indicates the communication and development flaws that occur in software organizations [54]. For example, if two people appear to be working on the same software module but do not communicate directly to each other, this situation is categorized as the "lone wolf" community smell. This smell type may cause duplication of a workpiece due to missing communication between them and extra development cost to the organization in the long term [55]. Community smells are the state-of-the-art measures that represent more accurately the socio-technical dynamics of organizations [54] compared to prior socio-technical network analysis methods [22]. Prior studies also

show that community smells highly occur along with the code smells in a project [55] since the former one indicates community flaws and the latter one indicates the coding flaws that occur in a project. While there is not any empirical evidence on the effect of community smells on the prediction of defect-prone modules, a prior study shows that code smells are one of the good indicators of the defective software classes [56]. Therefore, based on this finding, we aim to explore whether the community smells could be used to predict defective software classes.

Second, we explore the personalized approach that builds separate models for each developer in the team instead of building a single typical, general model that targets all team members. Since a personalized model of a developer solely learns from the developer's own development history, its output filters the irrelevant information from the other developers' development history. Prior studies [36,53] show the success of personalized SDP in terms of defect prediction capability, but they have limited experimental setups. For example, they assess only 100 commits from 10 developers, do not utilize the state-of-the-art process metrics, and only evaluates the proposed models in open-source projects. Therefore, we extensively investigate the performance of personalized change-level SDP models using six open-source, and one industrial project, state-of-the-art process metrics and AI/ML approaches, and seven model assessment metrics. Further, we investigate the development characteristics related to the superiority of the personalized models over general models or vice versa. Moreover, we analyze the contribution of metrics on the prediction of defects to understand whether the importance of metrics changes for different developers and different prediction model types (i.e., personalized and general).

So far now, very few studies have reported the deployment of a change-level SDP prototype into an industrial context [29,44] as we also mention earlier in Section 1.3. Therefore, in this thesis, we focus on the integration of an SDP solution into the software engineering pipeline of an industrial project in communication domain. We analyze the factors that should be considered in deployment of change-level SDP models in order to provide consistently successful and useful recommendations to developers. To accomplish that, we built SDP prototypes including personalized and traditional change-level SDP models for a chosen project of Ericsson, Turkey. After

choosing a successful prototype model using an offline evaluation setup, we deployed the prototype into the real development pipeline of Ericsson, Turkey. However, during the deployment phase, we had some challenges and concerns relating to applicability, model performance, label noise [29], and model update cycle. Hence, in this thesis, we investigate whether the personalized approach could be successfully applied to an industrial setting. Further, we propose an online prediction methodology to observe the real life performance of deployed model by simulating different levels of label noise and model update cycles.

1.5 Structure of the Thesis

Chapter 2 analyzes the impact of community smells on the prediction of defect-prone software classes. Specifically, it is aimed to observe the contribution of community smells in comparison with the other state-of-the-art metrics used to predict defect-prone software classes, e.g., code smell intensity, number of developers.

Chapter 3 reports a comprehensive investigation of personalized SDP models against traditional models on open-source projects. This chapter further analyzes the development characteristics that lead to the superiority and inferiority of personalized models compared to the traditional approach. Also, the importance of process metrics in defect prediction is analyzed for team members over their personalized SDP models.

Chapter 4 investigates the personalized SDP models against traditional models in an industrial setting. Also, the effect of combining information from various data sources such as commit messages and using AI/ML techniques on prediction accuracy is explored. Further, the importance of metrics for both personalized and traditional approaches are analyzed.

Chapter 5 reports the deployment experience and challenges in an industrial setting. Particularly, an online prediction setup is proposed to simulate the real-life performance of the SDP model under various levels of label noise and values of the model update period.

Chapter 6 concludes the thesis, shares the final remarks and future research directions.

2. AN EMPIRICAL STUDY ON THE EFFECT OF COMMUNITY SMELLS ON BUG PREDICTION[1]

2.1 Introduction

Detecting and fixing bug-prone modules in software systems is an essential part of the maintenance process. Predicting the residual bugs in the software before its release prevents companies from additional costs [22]. A variety of metrics and methodologies has been proposed to improve the performance of the bug prediction models. Some of these models utilize static code metrics [57], process metrics, [58], social network metrics [59], network metrics [60], developer metrics [61], code smell metrics and antipatterns [62].

Code smells are the symptoms of design flaws in software systems [63]. Poor design choices prior to code development, i.e., antipatterns, may propagate to failures which require extra refactoring and bug-fixing effort [62]. Researchers show that classes associated with antipatterns or code smells are more likely to be involved in bug fixing changes than other classes [62,64]. Taba et al. [65] propose a set of antipattern metrics, e.g., antipattern complexity and antipattern re-currency length, in addition to the structural metrics, e.g., lines of code, churn, and the number of methods, to predict software bugs. Later, [56] incorporate the *code smell intensity* metric (i.e., the severity of code smells) into antipattern metrics to compare their performances with the state-of-the-art bug prediction models. Both of the proposed code-smell related metrics (antipattern, code smell intensity) improve the performance of the state-of-the-art bug prediction models [56,66].

Software development is a collaborative activity including both individual effort and cooperative work among the developers through synchronization and communication. Researchers propose ways to define the organizational and social structures of software engineering, and their effects on problem solving and welfare of development

[1]This chapter is based on the paper "Eken, B., Palma, F., Ayşe, B., & Ayşe, T. (2021). An empirical study on the effect of community smells on bug prediction. Software Quality Journal, 29(1), 159-194.".

communities [67]. One approach to model collaborations among the developers is proposed in [59] to assign the issues to the related developers. In another study, [22] utilize social and technical networks to predict failures in software systems. Other metrics representing the developers of this cooperative work are based on, for example, the number of developers worked on a software module [17], the scattering of the code changes produced by developers [68], or contextual experience based on historical changes [61].

Tamburri et al. [35] define a term called '*community smell*' to explain social and organizational antipatterns, and propose ways to mitigate these smells based on their observations in industry. Analogous to code smells leading to technical debt, community smells are defined as measures of socio-technical debt [35]. Later, Magnoni [69] and Giarola [55] extended an existing socio-technical network analysis tool CodeFace [70] to measure community smells in software teams, and Giarola [55] points out that code smells and community smells tend to co-exist in the software systems. In fact, both terms measure the technical debt in software development through different aspects. A recent approach shows that the community smells influence the intensity of code smells more than other community-related metrics [71].

Although significant associations between code smells and community smells are identified [55], and community smells are modelled to predict code smell intensity of the source code [71], there has been so far no study investigating the effect of community smells on the bug-proneness of software modules. Tamburri et al. [35] mention that "*both forces should be reckoned with together during the software process*". Since code smells and code smell intensity are proven to be successful indicators of bugs in software systems [62,64] compared to the state-of-the-art process metrics [56], we aim to investigate if community smells, as they are found closely associated to code smells [55,71], would also successfully identify bugs in the software systems. We base our research on the prior works [55,56,71], and extend those works by building bug prediction models with community smells.

Our key contributions in this study include:

- We conducted an empirical analysis to examine the effects of community smells on bug-proneness of software classes. Palomba et al. [56] report that code smell intensity is the best indicator of bugs compared to several baseline models. In this study, we take this best performing metric set and its associated model from Palomba et al.'s study [56] and compare it against the models that we build with community smell metrics to investigate their explanatory power to predict bug-prone classes.

  Our results show that the code smell intensity metric and community smells are the most contributing ones for bug prediction depending on the dataset characteristics and smelly class ratios. Also, a combination of code and community smell metrics does not necessarily contribute to the prediction of bug-prone classes.

- We have built new datasets consisting of two parts: (1) *ReplicatedDB* which is an extended version of the dataset used by Paloma et al. [56] by incorporating three community smells and six code smells for seven open-source projects from the original dataset, and linking those to the rest of the metrics and bugs at class level; and (2) *CollectedDB* that includes three additional open-source projects used in Palomba et al.'s study [71], and Giarola's study [55] to enrich our datasets. We extracted similar metrics that *ReplicatedDB* have, and linked those with bugs at the class level. Finally, they have become very comprehensive datasets including static code attributes [72], number of developer metrics [73], churn [74], scattering metrics [68], code smell intensity [75], code smells and community smells [55], and bugs. Both datasets are available online with a replication package of our study [76].

2.1.1 Structure of the chapter

The rest of the chapter is organized as follows: Section 2.2 discusses the bug prediction and community smells-related studies in the literature. Section 2.3 explains our methodology to answer all the research questions, including the validation strategies, algorithms, and performance measures. Section 2.4 shows the results of the experiments while Section 2.5 provides an in-depth discussion on the results. Section

2.6 discusses the threats to the validity of the experiments, and, finally, Section 2.7 concludes the paper.

2.2 Related Work

We organize related work as follows: Section 2.2.1 gives an overview of the bug prediction studies; more specifically, on the metrics that are found as the best indicators of bugs in these models. Later in Section 2.2.2, we discuss the social aspects of software development, their usage on bug prediction studies, community smells introduced by Tamburri et al. [35] and related studies on community smells.

2.2.1 Literature on bug prediction

In the field of software engineering, bug prediction has been widely studied by both researchers and practitioners through offline or field studies. Existing bug prediction models utilizing static code metrics perform well with a probability of detection rate between 70% and 80% [57,77]. Studies in the literature approach bug prediction in different ways. Many techniques for data collection, metric extraction and aggregation, and training methodologies are investigated. We briefly report the studies from those aspects in this section. We suggest readers refer to systematic literature reviews (SLRs) that synthesize the current evidence in the bug prediction literature and discuss the studies in terms of algorithms, metrics, model construction methodology in more detail [11,14,18].

The static code metrics, object-oriented metrics, call-graph based dependency metrics [78], process metrics [77,79], network metrics [60] are very popularly used input features in bug prediction studies. According to the SLRs on bug prediction [14,18], there is not a single set of metrics that fits for any project and/or any modeling technique to build an accurate bug prediction model. There are other data-oriented solutions, e.g., Turhan et al. [80] analyze the effect of using cross-project or mixed-project data to predict bugs, whereas Zhang et al. [81] observe different types of aggregations at class/module-level metrics while building bug prediction models at file-level. Tsakiltsidis et al. [51], and Tosun et al. [27], on the other hand, focus on a specific type of bugs to improve the usefulness of predictions.

Some studies focus on the algorithm, and methodology aspects, e.g., Lessmann et al. [50] analyze the performances of different machine learners on bug prediction models, whereas Fu et al. [52] analyze the effect of parameter tuning on the performance of bug prediction. Other studies also apply different model validation techniques when measuring the performance of bug prediction models [82].

Fowler [83] introduced the term *code smell* as the symptoms of design flaws (e.g., duplicate code) in software. Studies report that code smells have high survivability [63], and, in general, very few smells are removed from a project, so they tend to increase over time [84]. Developers tend to postpone removing code smells [85] to avoid modification on the systems [86], or they do not perceive code smells as a critical problem [87]. The studies on bug prediction, on the other hand, show how critical these code smells are in terms of their bug-introducing patterns. For example, Khomh et al. [62] analyze four open-source systems and report that classes having poor design choices, namely antipatterns, are more likely to be involved in bug fixing changes than other classes. Palomba et al. [64] also confirm that classes involved in code smells are more prone to contain bugs in 30 open-source systems. Tabe et al. [65] use antipattern information (e.g., the average number of antipatterns) in addition to the traditional metrics to predict bugs, and state that the models built with antipattern metrics can increase the prediction performance up to 12.5% in terms of F-measure.

Following the work of Taba et al. [65] and Palomba et al. [56] we include code smell intensity metric into four different state-of-the-art bug prediction models (baselines), each of which utilizes *Structural*, *Basic Code Change*, *Developer*, and *Developer Changes Based* metrics, respectively. They also added antipattern metrics into these baselines and compare the predictive power of code smell intensity against the antipattern and baseline metrics on bug prediction. Their analyses show that the code smell intensity significantly improves the F-measure of the four baseline models. More specifically, when the code smell intensity is added into the Developer baseline model, the performance increases up to 21% in terms of F-measure. The best performing baseline model, on the other hand, is Developer Changes Based with 69% F-measure. Antipattern metrics, on the other hand, slightly improve the performance of the baseline models. Nevertheless, a combination of the baseline metrics, antipattern

metrics, and code smell intensity gives the best prediction performance. In another study, Soltanifar et al. [66] propose to build a defect prediction model using code smells metrics, churn metrics, and a combination of them. The authors conclude that those models with code smell metrics mostly outperform the ones that are trained with churn metrics only.

2.2.2 Literature on people aspect and community smells

The people aspect incorporated into bug prediction models is represented through developer-related metrics, e.g., the number of developers contributing to software modules [17], the focus of a developer on a software module [68], or developer's confirmation bias levels [33]. Lately, personalized bug prediction models are introduced to train specialized models for each developer by using the selected developer's historical development activity only [88]. Recent approaches to model people aspect also propose metrics capturing developer experience [61], developers' review activities, and their involvement in the review process [89].

Studies in the literature also model the social aspect of software development. For example, Bird et al. [22] propose *socio-technical* networks to predict the bugs in Windows Vista, their prediction performance improve 5% in terms of recall. Çalıklı et al. [90] model software team interactions to improve software quality management. Later, Tamburri et al. [35] investigate social debt in software engineering, describing as "*not quite right development community, which we postpone making right*". The authors discuss that socio-technical structures and decisions of software organizations may cause additional costs or delays in software development activities. They define a set of *community smells*, and propose a tool called CodeFace4Smells to automatically reflect the socio-technical issues [55,91]. Empirical analysis on open-source communities points out the diffuseness of community smells [91] and how developers perceive those as relevant issues regarding software evolution. Lately, two novel approaches are proposed to capture eight common types of community smells (e.g., "sharing villainy" which caused by poor quality information exchange within the team) in software projects by utilizing social network, community, geographic dispersion, formality, and truck number properties [92,93]. A machine learning-based

approach by Almarimi et al. [93] achieved a detection performance of 94% in terms of AUC. There are other studies focus on community smells: Catolino et al. [94] found that women's presence in teams tends to reduce the amount of community smells, Catolino et al. [95] suggests a guideline on refactoring techniques for community smells, e.g., restructuring the team, creating a communication plan for team members.

Palomba et al. [71] further study the relationship between the community smells, socio-technical congruence, and code smell intensity, and report that community-related factors contribute to the intensity of code smells. Studies on the relationship between code and community smells inspire us to further investigate the community smells, and the extent to which community smells contribute to predicting bugs. As code smells and other social aspects of software development are shown to be good indicators of software bugs, we think that community smells would likely explain software bugs. Based on the findings of Palomba et al. [56,71] and Giarola [55], we aim to investigate the explanatory power of community smells on software bugs, compared to the best performing metrics, code smells and code smell intensity. We want to assess whether community smells are as good indicators of bugs as code smell related metrics and other process metrics.

2.3 Methodology

We have three research questions (RQs) to empirically investigate the effect of community smell-related information on the prediction of bug-prone classes. Our empirical analyses are conducted on two datasets consisting of different open-source software projects. We selected those projects from prior works [55,56,71] that our study is inspired by. Unfortunately, accessing all the projects' repositories and messaging logs was not possible, and thus, we ended up having a total of 10 projects in our datasets. More details on the datasets are reported in Section 2.3.1. Our RQs are formulated as follows:

RQ1: *To what extent does the community smell contribute to the prediction of bug-prone code components?*

Our main goal is to analyze the contribution of community smells in predicting bug-prone classes. To do this, we include the community smells into the four baseline bug prediction models using process metrics, similar to those built by Palomba et al. [56]. We assess the impact of community smell information on bug prediction performance compared to the process metrics.

RQ2: *To what extent does the proposed model contribute to the bug prediction when compared to a state-of-the-art model built using code smell related metrics?*

Based on the findings of Palomba et al. [56], the best performing metric for bug prediction is code smell intensity. Therefore, we aim to compare the performance of a baseline model with community smells and with code smell related metrics. We use both code smell metrics extracted via *CodeFace4Smells* tool and code smell intensity metric calculated according to Fontana et al. [75], and incorporated those into the baseline models. Then we assess their performance against the model using baseline metrics and community smells.

RQ3: *To what extent do the combined smell-related information contribute to the bug prediction when compared to the best contributing one?*

We aim to assess the combined ability of community smells and code smell-related information in predicting bug-prone classes. We build the models with the most contributing metric group found in the prior research question for each dataset and combine these models with the rest of the metric groups.

An overview of our research methodology to answer our research questions is depicted in Figure 2.1. We summarize the steps shown in the figure below before explaining those in the subsequent sections in detail:

- **Data collection:** We utilize two datasets in our research: (1) the one used by Palomba et al. [56] that we extend by including the community and code smells and refer in this study as *ReplicatedDB*, and (2) the one used Palomba et al. [71] that we extend by collecting community smells, code smells, code smell intensity, and the structural metrics. We name this dataset *CollectedDB*. Section 2.3.1 provides more details on data collection.

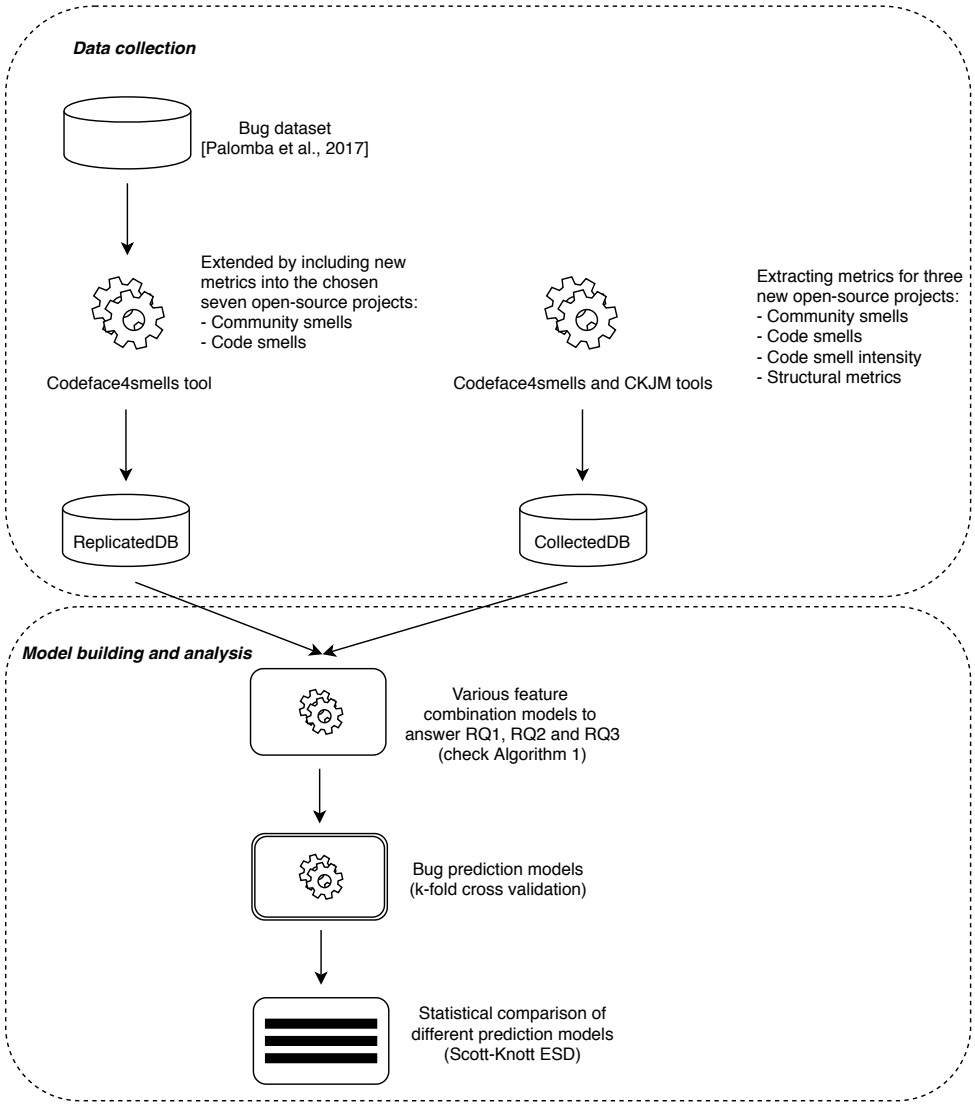

Figure 2.1 : Overview of our research methodology.

- **Model building and analysis:** We build seven bug-prediction models utilizing different combinations of the metrics to answer our three RQs. These seven models utilize baseline metrics, and a combination of baseline and community and code smell-related metrics depending on the RQs. For RQ1, two models are reported utilizing baseline metrics and baseline+community smell metrics. For RQ2, two more models are reported utilizing baseline metrics+code smell related information. For RQ3, models using combinations of community smells and code smell-related information are reported. This model building and analysis step is also depicted in Algorithm 1 (Section 2.3.2). Models are compared to each other by using Scott-Knott ESD test to confirm the statistical significance of the contribution of

community smells on the prediction of buggy software classes compared to those of the process metrics or code smell related metrics.

2.3.1 Datasets used in this study

We have two datasets to answer our RQs, namely *ReplicatedDB* and *CollectedDB* [76]. The *ReplicatedDB* is an extension of the dataset provided in [56], while the *CollectedDB* consists of newly collected software projects, particularly for this study. The projects are selected based on the work by Palomba et al. [71], but due to the fact that authors did not share their full dataset, we crawled the data for this work. In this section, we give the details of the data collection process and discuss the metric extraction process in Section 2.3.1.1.

The first dataset – *ReplicatedDB* – is an extended version of the seven projects used in the study by Palomba et al. [56]. The original dataset used by Palomba et al. [56] is composed of class-level bug information, the state-of-the-art structural metrics [72], number of developers [74], code churn [73], developer scattering metrics [68], and a code smell intensity metric collected from a total of 34 releases of 11 open-source projects.

We have extended this dataset by adding community smells and code smells. To extract the community smells and code smells, we executed the *CodeFace4Smells* tool [55] on the selected open-source projects. The *CodeFace4Smells* tool generates developer collaboration and communication networks of the software projects by using mailing lists and source code repositories. Generated developer networks are used to perform the social and technical analyses of the software projects, and later, the tool generates an output in the form of classes with community and code smell information. The tool gives the community and code smell information as a binary value (0 or 1 for the absence or presence of smells) for each class. After gathering the projects' class-level community and code smells, we matched this information with the classes listed in the original dataset [56] using the class names and the release information of the projects.

The original dataset from Palomba et al. [56] contains the earlier releases of the projects between the years 2000-2008. The *CodeFace4Smells* tool could not give

an output for the initial few releases of the projects because they do not have a collaboration and communication history. Also, the source code repository or mailing list of some projects were not reachable. As a result, our extended bug prediction dataset *ReplicatedDB* contains fewer projects and releases than those listed in the study of Palomba et al. [56]. The final *ReplicatedDB* contains an extended version of a total of 16 releases of the seven open-source projects with more metrics.

The second dataset – *CollectedDB* – is newly constructed for this study by also utilizing the *CodeFace4Smells* tool. *CollectedDB* contains three open-source projects, namely Cayenne, Eclipse-CXF, and Mahout. All of these projects were selected based on the 'Known Projects' list of Giarola's study [55], i.e., *CodeFace4Smells* tool was able to extract the code and community smells successfully for these projects in the earlier study. For these three projects, the *CodeFace4Smells* tool created three-month time windows for the years between 2013-2016 and computed these smells at the class level. We then aggregated these three-month subsets to form a single dataset for each project. We removed the classes that contain no smells from the *CollectedDB* to avoid bias during analysis. We also collected structural metrics for the *CollectedDB* by utilizing the Ckjm tool [96] to create a baseline model for this dataset.

The number of classes, number of community and code smells that each project has also reported in Table 2.1 and 2.2. We created this new *CollectedDB* dataset because the projects in *ReplicatedDB* contains a limited number of classes with respect to the community smells and other code smells. From Table 2.1 and 2.2 we can observe that the maximum number of classes associated with any smell type in *ReplicatedDB* is 92 with *Missing Link* (ML) community smell. In *CollectedDB*, there exist up to 238 classes associated with this community smell, and up to 800 classes associated with any of the code smells.

2.3.1.1 Metrics

Both the *ReplicatedDB* and *CollectedDB* contain code and community smell metrics, code smell intensity, and structural metrics. In addition to those metrics, *ReplicatedDB* also contains churn, developer metric, and scattering metrics. The descriptions of the full list of metrics are provided in Table 2.3. The last two columns show which dataset

Table 2.1 : Properties of the datasets used in this study.

Dataset	Project	Releases	Total classes	Bug-prone classes
ReplicatedDB	Ant	1.3-1.6	943	184
	Ivy	2	352	40
	Log4j	1.1-1.2	298	223
	Lucene	2.0-2.4	750	437
	Velocity	1.5-1.6.1	443	176
	Synapse	1.2	251	86
	Xerces	1.2-1.4.4	1,222	353
CollectedDB	Cayenne	2013-2016	518	312
	CXF	2013-2016	1,222	530
	Mahout	2013-2016	322	48

contains the corresponding metrics set. We include all metrics sets from the original data taken from Palomba et al.'s study [56]. However, we intend to mainly analyze the effect of code and community smell related metrics on bug prediction. Structural, churn, developer, and scattering metrics are the state-of-the-art features that used in the bug prediction modelling and we kept them in the dataset to create our baseline models.

The output of the *CodeFace4Smells* tool is the list of classes in the selected projects with a binary code and community smell information, i.e., 0 indicates a class does not have the specified smell while 1 refers to the presence of the specified smell. These outputs are used as the code and community smell metrics when building the models. Based on the definitions in Giarola's study [55], a brief description of code and community smells considered in our study are presented below.

- **Code smells:**
  - *Class Data should be Private* (CDP) occurs when a class has more than 10 public variables, it is considered to be involved in CDP smell.
  - *Complex Class* (CC), if the McCabe Cyclomatic Complexity of a class is above 200, the class is considered to be involved in CC smell.
  - *Functional Decomposition* (FD) may occur when a class is intended to be object-oriented and implemented by non-object-oriented developers.

Table 2.2 : Number of code and community smells associated with software classes.

Project	Code Smells						Community Smells		
	CDP	CC	FD	GC	SC	LM	OS	RS	ML
Ant	6	2	0	58	46	19	24	49	49
Ivy	1	3	1	9	10	6	8	15	15
Log4j	4	1	0	4	3	2	3	8	8
Lucene	4	8	0	22	19	8	2	15	17
Velocity	2	5	0	6	6	2	3	4	4
Synapse	3	0	0	2	2	11	6	14	14
Xerces	21	19	0	81	68	52	59	73	92
Cayenne	168	58	12	320	168	66	65	3	76
CXF	262	193	33	821	534	274	27	119	238
Mahout	97	24	12	158	71	67	19	16	29

Table 2.3 : Metrics and their descriptions.

Name	Descriptions	ReplicatedDB	CollectedDB
Structural metrics	Size, complexity and object-oriented metrics [72]	X	X
Churn	Entropy of changes of a given time period [73]	X	
Developer	Number of developers worked on a class [74]	X	
Scattering	Scattering of developers' development on the code modules [68]	X	
Code smell intensity index	A numeric value that measures how much a class included into code smells	X	X
Code smells	Extracted via CodeFace4Smells [55]	X	X
Community smells	Extracted via CodeFace4Smells [55]	X	X

- *God Class* (GC) occurs when a class is large and depends on the data stored in surrounding data classes.
- *Spaghetti Code* (SC) may occur when a class is implemented in the way of procedural thinking.
- *Long Method* (LM), if a class has one or more methods with more then 100 lines and more then two input parameters, it is considered to be involved in LM smell.

- **Community smells:**
  - *Organizational Silo* (OS) is a software development pattern in an organization where the developers collaborate with others but do not communicate within the analyzed communication channel.
  - *Radio Silence* (RS) is a pattern such as there is unique knowledge and information brokers toward different sub-communities.
  - *Missing Links* (ML) is a pattern where there is a collaboration between developers and there is a lack of communication between collaborators.

ReplicatedDB has already had the code smell intensity metric. In this study, we calculated the code smell intensity for the *CollectedDB* based on the code smell intensity calculations provided by Fontana et al. [75]. Firstly, we determine the 10%, 25%, 50%, 75%, and 90% quartiles for the values of each structural metric extracted with the Ckjm tool [72]. Secondly, we find the minimum value of the quartile that corresponds to each structural metric. Thirdly, we take the average of the determined values for each structural metric and set it as a code smell intensity metric. The code smell intensity metric is only calculated for the classes containing code smells. If a class does not contain any code smell, we take its code smell intensity as zero (0).

2.3.2 Model building and analysis

We conducted our empirical analysis by assessing the performance of the class-level bug predictors that incorporate the community smells into a set of state-of-the-art baseline models and comparing them with the other models built with the

state-of-the-art metrics as well as code smell related metrics. The performance of the models is statistically compared to each other based on the related RQ. Our methodology is shown as pseudocode in Algorithm 1.

Algorithm 1 Pseudocode for answering RQs.

```
Projects1 = (Ant, Ivy, Log4j, Lucene, Synapse, Velocity, Xerces)
Projects2 = (Cayenne, Eclipse-CXF, Mahout)
Learners = (Simple Logistic, Multilayer Perceptron, ADTree, Naive Bayes, Decision Table, Logistic Regression)

ModelsForReplicatedDB = (Baseline (RQ1), +CommSmells (RQ1), +CodeSmells (RQ2), +CodeInt (RQ2),
+CommSmells+CodeInt (RO3), CodeSmells+CodeInt (RO3), +CommSmells+CodeSmells+CodeInt (RQ3))

ModelsForCollectedDB = (Baseline (RQ1), +CommSmells (RQ1), +CodeSmells (RQ2), +CodeInt (RQ2),
+CommSmells+CodeSmells (RQ3), +CommSmells+CodeInt (RQ3), +CommSmells+CodeSmells+CodeInt (RQ3))

StatisticalTests = (Scott-Knott ESD)
N = 3
M = 10
for projectData in ReplicatedDB do
    for m in ModelsForReplicatedDB do
        projectData' = apply m to projectData
        for i = 1 to M do
            projectData'' = randomize the order of projectData'
            S = generate N folds from projectData''
            for i = 1 to N do
                trainSet = S[j]
                testSet = S − S[j]
                for l in Learners do
                    model = apply l on trainSet
                    predictions += apply model on testSet
                end for
            end for
            recall, pf, f − measure, auc = calculate from predictions
        end for
    end for
    for s to StatisticalTests do
        apply s to the recall, pf, f − measure, auc of all models of both datasets
    end for
end for
for projectData in CollectedDB do
    for m in ModelsForCollectedDB do
        projectData' = apply m to projectData
        for i = 1 to M do
            projectData'' = randomize the order of projectData'
            S = generate N folds from projectData''
            for i = 1 to N do
                trainSet = S[j]
                testSet = S − S[j]
                for l in Learners do
                    model = apply l on trainSet
                    predictions += apply model on testSet
                end for
            end for
            recall, pf, f − measure, auc = calculate from predictions
        end for
    end for
    for s to StatisticalTests do
        apply s to the recall, pf, f − measure, auc of all models of both datasets
    end for
end for
```

The models were implemented using six machine learning algorithms, i.e., Simple Logistic, Multilayer Perceptron, ADTree, Naive Bayes, Decision Table (based on an

ordered set of rules generated by decision trees [97]), and Logistic Regression. We chose these algorithms since they were also used in the previous research by Palomba et al. [56] and shown to be successful at predicting class-level bug proneness. We implement the same algorithms of the prior work to avoid potential effects of the experimental setup on the findings.

For each dataset, we assess the performance of seven bug prediction models listed in Table 2.4, then compared the performances with each other, with respect to their associated RQs. For *ReplicatedDB*, this comparison is repeated for each of the four baselines built for structural metrics, churn, number of developers, and scattering metric (see Table 2.3). For *CollectedDB*, the comparison is made against one baseline built with the structural metrics.

Table 2.4 : List of bug prediction models built in this study.

Model abbreviation	Metrics included into the model
Baseline*	Structural, churn, developer, scattering
+CommSmells	Baseline + Community Smells
+CodeSmells	Baseline + Code Smells
+CodeInt	Baseline + Code Smell Intensity
+CommSmells+CodeSmells	Baseline + Comm. Smells + Code Smells
+CommSmells+CodeInt	Baseline + Comm. Smells + Code Smell Intensity
+CodeInt+CodeSmells	Baseline + Code Smell Intensity + Code Smells
+CommSmells+CodeInt+CodeSmells	Baseline + Comm. Smells + Code Smell Int. + Code Smells

*There is only one Baseline model for the *CollectedDB* that contains the structural metrics while there are four different Baseline models for the *ReplicatedDB* that utilizes only one of the structural, churn, developer, and scattering metrics respectively.

In RQ1, we compare two bug prediction models listed in Table 2.4: *Baseline* vs. *+CommSmells*. The *+CommSmells* model includes community smells in addition to the *Baseline* model's metric sets, while the *Baseline* only includes one of the structural metrics, churn, number of developers, or scattering metric.

In RQ2, we build and assess the performance of three bug prediction models listed in Table 2.4: *+CommSmells*, *+CodeSmells*, and *+CodeInt*. The *+CodeSmells* model includes the *Baseline* model's metric set and the code smells, while the *+CodeInt* model includes the *Baseline* model's metric set and the code smell intensity metric.

In RQ3, we would like to assess the effect of the combined smell-related models against the best performing model in RQ2. According to the results of RQ2, the best performing model changes among datasets. In particular, it is *+CodeInt* for *ReplicatedDB*, while the *+CommSmells* is the best performing metric set for *CollectedDB*. Therefore, in RQ3, for *ReplicatedDB*, we analyze the effect of combining the code smell intensity metric with community smells (*+CommSmells+CodeInt*), code smells (*+CodeSmells+CodeInt*), and both community and code smells (*+CommSmells+CodeSmells+CodeInt*) on the prediction of bug-prone classes. For *CollectedDB*, we analyze the effect of combining community smells with code smell intensity (*+CommSmells+CodeInt*), code smells (*+CommSmells+CodeSmells*), and both code smells and their intensity (*+CommSmells+CodeSmells+CodeInt*) on the prediction of bug-prone classes.

All of the bug prediction models listed in Table 2.4 are trained and tested by following a *within-project* strategy: We train and evaluate the models for each project aggregated over all releases. With this, we have seven bug prediction models (see Table 2.4) for each project listed in *ReplicatedDB* and *CollectedDB*. While building and evaluating the bug prediction models, 10 repetitions of 3-fold cross-validation are applied. Firstly, we take a project from the datasets and randomize the order of the instances to avoid sampling bias [98] before building and evaluating a model. Secondly, we split the data into three folds and take one fold as the test set and the remaining two folds as the training set. Then we build a model using the selected training set and evaluate the model on the remaining test set. We collect the prediction results of the repeated cross-validation steps. The reason that we set the fold number as three when applying the cross-validation technique is the low ratios of buggy and smelly class instances (see Table 2.1 and 2.2). If we set a higher fold number, some training sets might have almost no smelly instances or very few buggy instances, and in turn, there may not be enough samples to learn from for the predictors.

We evaluate all the models in terms of the *probability of detection* (pd or recall), *probability of false alarm* (pf), *F-measure*, and the *AUC* (Area Under The ROC -Receiver Operating Characteristics- Curve). The recall, pf, and F-measure values are calculated from the confusion matrix of the prediction results. A simple confusion

matrix for the bug prediction is given in Table 2.5 and the calculation formulas of the recall, pf, and F-measure are provided in Table 2.6. The recall measures the ratio of correctly predicted bug-prone classes to the actual bug-prone classes. The pf measures the ratio of incorrectly predicted bug-prone classes to the actual clean (not buggy) classes. The F-measure is the harmonic mean of the recall and precision values. The precision measures the ratio of the actual bug-prone classes to predicted as bug-prone classes. Precision measurement is not used in our model assessment step, but its calculation formula is provided in Table 2.6 for information purposes.

Table 2.5 : Confusion matrix for bug prediction problem.

	Actually bug-prone	Actually clean
Predicted as bug-prone	TP (True Positive)	FP (False Positive)
Predicted as clean	FN (False Negative)	TN (True Negative)

Table 2.6 : The equations for recall (or pd), pf, precision, and F-measure.

Accuracy measures	Formula
Recall (or pd)	$\frac{TP}{TP+FN}$
Pf	$\frac{FP}{FP+TN}$
Precision	$\frac{TP}{TP+FP}$
F1-measure	$\frac{2\times precision\times recall}{precision+recall}$

The performance results of the proposed bug prediction models are compared with each other to analyze the contribution of each smell related metric type in terms of recall, pf, F-measure, and AUC values. To perform statistical tests among the bug prediction models, we aggregated a model's performance (in terms of recall, pf, F-measure, and AUC individually) over all the projects and machine learning algorithms. The aggregated performances of prediction models are compared with each other by applying Scott-Knott Effect Size Difference (ESD) test [99] implemented by Tantithamthavorn et al. [82]. The results of different projects and algorithms are further discussed in Section 2.5.

2.4 Results

This section presents the results obtained through the experiments to answer RQ1, RQ2, and RQ3. Performances of prediction models associated with each RQ are

reported as boxplots in terms of recall, pf, F-measure, and AUC. The names of the prediction models are shown along the y-axis of each group of box plots. The performances are aggregated over six machine learning algorithms used during training and for all projects within each dataset. Scott-Knott ESD analysis for the prediction models built for each RQ is provided in our online appendix [76], as well as the performance plots of the predictors. The detailed analysis of the algorithms and projects are reported in Section 2.5.

2.4.1 RQ1: To what extent does the community smell contribute to the prediction of bug-prone code components?

First, the performances of *Baseline* and *+CommSmells* are aggregated over all the baselines and the datasets and reported in Figure 2.2. Then, the performances of the predictors are reported separately for *ReplicatedDB* and *CollectedDB* in Figures 2.3, and 2.4, respectively. Note that in this paper, we only report the performance of one out of four baselines of *ReplicatedDB* (number of developers), and report the other baselines in our online appendix since it would take too much space in the paper.

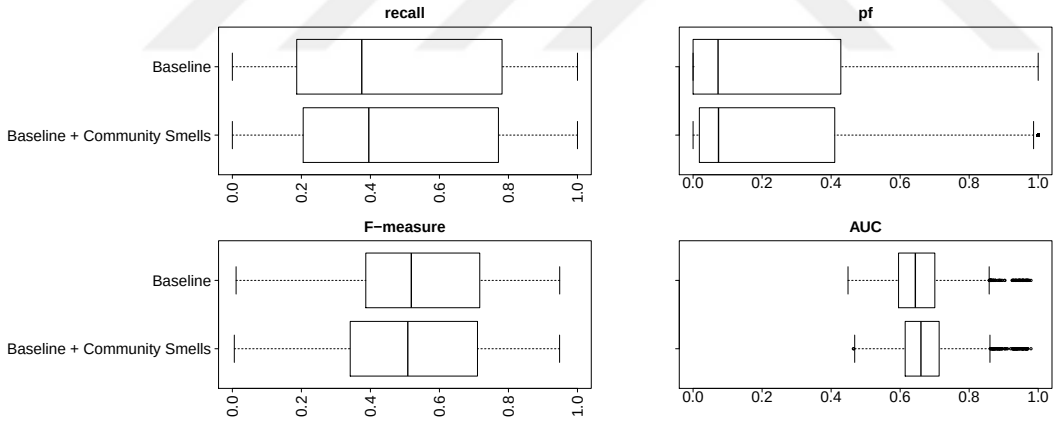

Figure 2.2 : Performances of the bug prediction models built to answer RQ1. The performance values are aggregated over both datasets.

Overall performance results in Figure 2.2 show that *Baseline* and *+CommSmells* models perform closely to each other. Both models achieve a recall of ∼38%, pf of ∼7%, F-measure of ∼51%, and AUC of ∼65%. According to Figures 2.3, and 2.4, the models achieve different performances among the two datasets. While the median recall values of the predictors built for *CollectedDB* are around 75%, the median recall

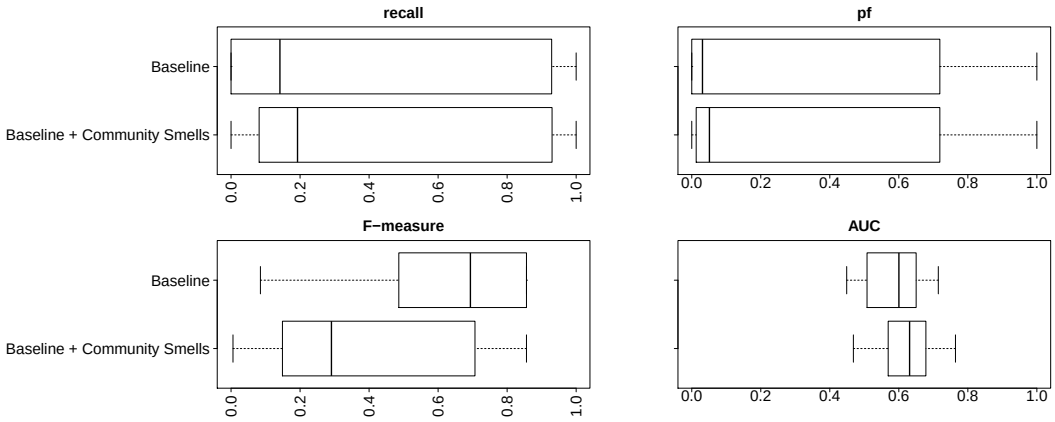

Figure 2.3 : Performances of the bug prediction models built to answer RQ1, for ReplicatedDB, with the baseline model utilizes number of developers metric.

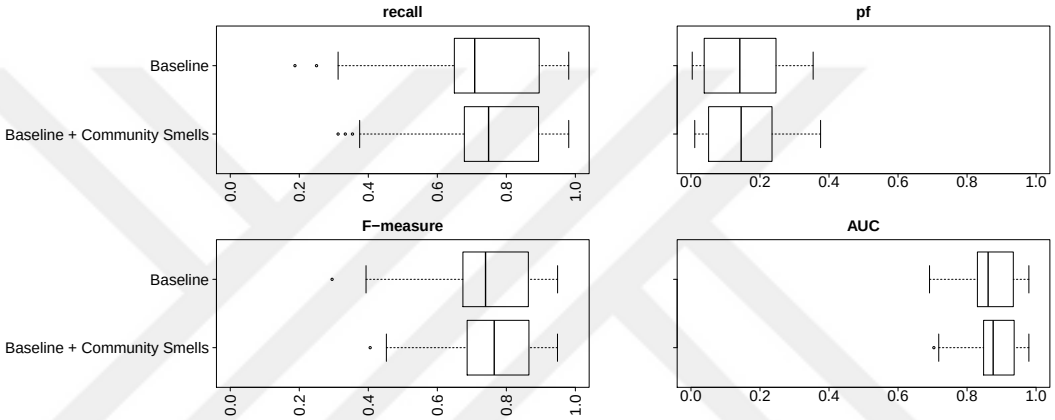

Figure 2.4 : Performances of the bug prediction models built to answer RQ1, for CollectedDB, with the baseline model utilizes structural metrics.

values of the predictors built for *ReplicatedDB* are between 20% (with the number of developers) and 39% (with the structural metrics).

The +*CommSmells* model outperforms three out of four baselines of *ReplicatedDB* (number of developers, churn, scattering), and the baseline of *CollectedDB* by improving the recall between 2% and 6%, and AUC between 1% and 3%. However, according to the Scott-Knott ESD comparison results, there are no significant differences among the *Baseline* and +*CommSmells* in terms of recall, pf, and F-measure. The contribution of community smells on bug prediction is statistically significant only in terms of AUC.

The reason that we cannot observe the contribution of community smells on *Baseline* models might be class distributions with respect to community smells and bugs. To

check that, we compute the ratio of classes associated with community smells over the total number of classes, and the bug-prone classes associated with community smells. In *ReplicatedDB* up to 7% of classes have one type of community smell, and among those, 38% are bug-prone. In other words, only 26% of classes in *ReplicatedDB* are both bug-prone and have community smells. In *CollectedDB* only 15% of classes are both bug-prone and have community smells. These relatively low ratios might be the reason for not observing the contribution of community smells. Having a model with a combined set of metrics may improve performance. We further assess this effect in RQ3.

Summary on RQ1: +*CommSmells* model statistically outperforms the *Baseline* in terms of AUC, while it performs statistically the same as *Baseline* in terms of recall, pf and F-measure.

2.4.2 RQ2: To what extent does the proposed model contribute to the bug prediction when compared to a state-of-the-art model built using code smell related metrics?

Similar to answering RQ1, at first, the performances of the predictors built for all the baselines of two datasets (*ReplicatedDB*, *CollectedDB*) are aggregated and reported as boxplots in Figure 2.5. Then, the performances of the predictors reported separately for one baseline of *ReplicatedDB* (number of developers), and *CollectedDB* in Figures 2.6, and 2.7, respectively.

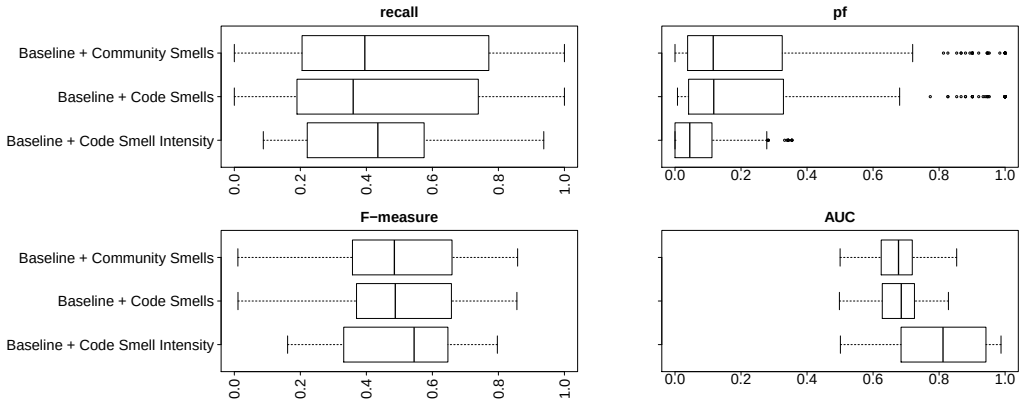

Figure 2.5 : Performances of the bug prediction models built to answer RQ2.

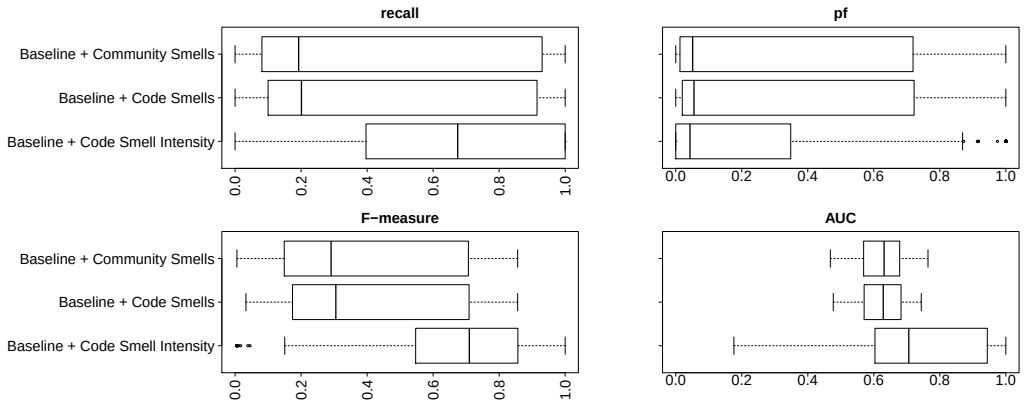

Figure 2.6 : Performances of the bug prediction models built to answer RQ2, for ReplicatedDB, with the baseline model utilizes number of developers metric.

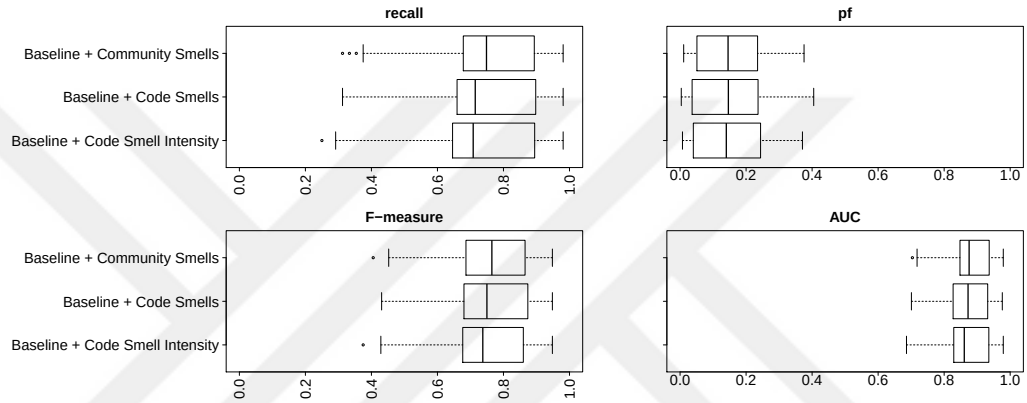

Figure 2.7 : Performances of the bug prediction models built to answer RQ2, for CollectedDB, with the baseline model utilizes structural metrics.

According to the overall performance results in Figure 2.5, the +*CodeInt* model achieves a higher performance in terms of recall, pf, F-measure, and AUC when compared to +*CodeSmells* and +*CommSmells*. A recall of 43%, 39%, and 37%, pf of 4%, 12%, and 11%, F-measure of 54%, 49%, and 48%, and AUC of 82%, 69%, and 68% are shown in the figures for +*CodeInt*, +*CodeSmells*, and +*CommSmells* models, respectively.

For *ReplicatedDB*, adding code smells into the baseline models seems to improve the performance more than community smells. According to Scott-Knott ESD tests, +*CodeInt* outperforms the +*CommSmells* and +*CodeSmells* in terms of recall, pf, F-measure, and AUC for all baseline models of *ReplicatedDB*. However, for *CollectedDB*, +*CommSmells* achieves 5% higher recall than +*CodeSmells* and

+*CodeInt*, and 2% higher AUC. Only the difference in terms of AUC is statistically significant, according to the Scott-Knott ESD test.

Similar to the performance values in RQ1, the bug prediction models in RQ2 achieve different performances among the datasets. While the median recall values of the predictors built for *CollectedDB* are between 70% and 75%, the median recall values of the predictors built for *ReplicatedDB* are between 20% and 69%.

As shown in Figure 2.7, we could not observe the contribution of code smell intensity metric in the models for *CollectedDB*. This might be related to the ratio of classes associated with code smell intensity. In *ReplicatedDB* 60% - 73% of classes are associated with code smell intensity. On the other hand, in *CollectedDB*, almost all classes (99%) are associated with code smell intensity. Therefore, this metric seems to have no distinguishing effect on predicting bug-prone classes in *CollectedDB*.

Summary on RQ2: Community smells contribute to the bug prediction performance for *CollectedDB* in terms of AUC only, while code smell intensity improves the bug prediction performance of the *ReplicatedDB* in terms of recall, pf, F-measure, and AUC, when compared to other smell-related metrics.

2.4.3 RQ3: To what extent do the combined smell-related information contribute to the bug prediction when compared to the best contributing one?

We report the performance of +*CodeInt*, +*CodeInt*+*CodeSmells*, +*CodeInt*+*CommSmells*, and +*CodeInt*+*CodeSmells*+*CommSmells* for all the baseline models of *ReplicatedDB* in Figures 2.8, 2.9, 2.10, and 2.11. The performance of the combined models on *ReplicatedDB* has different conclusions with regard to the baseline metrics, as in other RQs. For the baseline models with the number of developers and scattering metric, the combined models are higher than +*CodeInt* in terms of recall and F-measure. On the contrary, for the baseline model with the churn metric, combining smell-related metrics with the +*CodeInt* model decreases the recall and F-measure. According to the Scott-Knott ESD tests, these differences among the performances of the prediction models with churn, scattering and number of developers metric sets are not statistically significant in terms of recall, F-measure and AUC. In terms of pf, +*CodeInt*

outperforms the combined prediction models when the baselines are built with churn, scattering and the number of developers. When the baseline model is built with structural metrics, *+CodeInt*, *+CodeInt+CodeSmells*, *+CodeInt+CommSmells*, and *+CodeInt+CodeSmells+CommSmells* perform statistically the same.

For *CollectedDB*, the performance of *+CommSmells* and the combined models seem to be the same in terms of recall, pf, F-measure, and AUC, which are 75%, 14%, 77%, and 88% respectively. Thus, including code smell related information into the *+CommSmells* model does not improve the performance for *CollectedDB*.

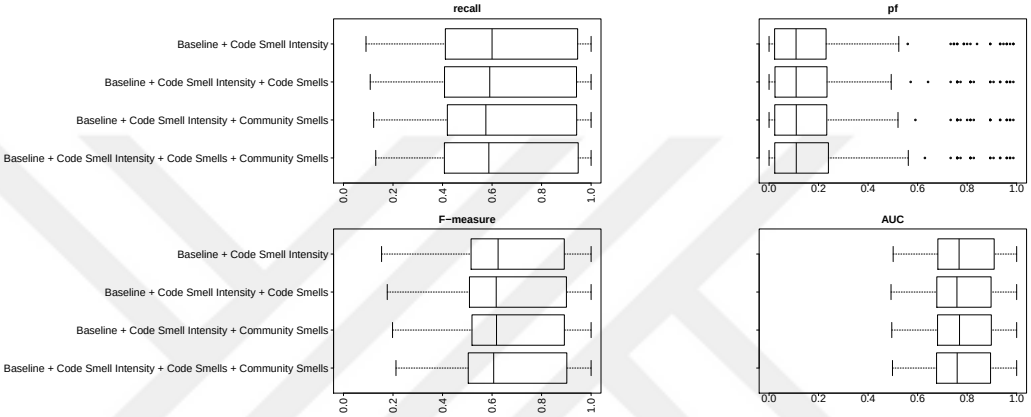

Figure 2.8 : Performances of the bug prediction models built to answer RQ3, for ReplicatedDB, with the baseline model utilizes structural metrics.

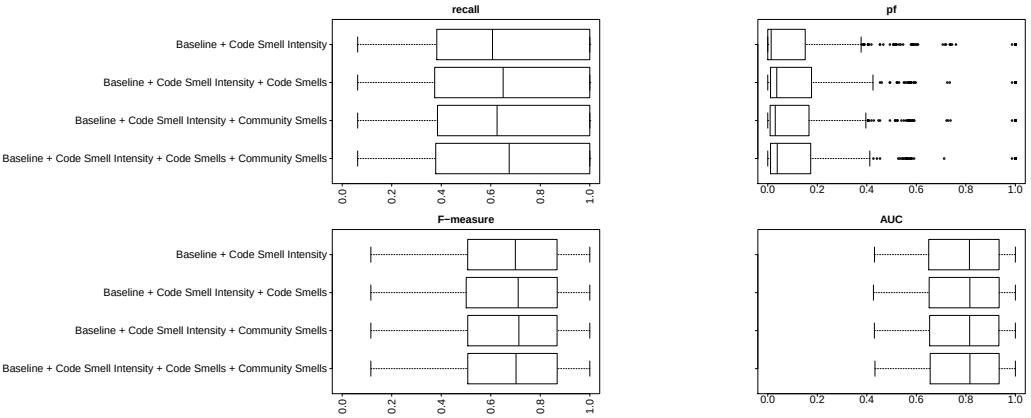

Figure 2.9 : Performances of the bug prediction models built to answer RQ3, for ReplicatedDB, with the baseline model utilizes churn metric.

Moreover, incorporating combined smell-related metrics into different baselines leads to different findings. Using more smell-related metrics increases the performance values for the number of developers, churn, and scattering baselines. However, it does not

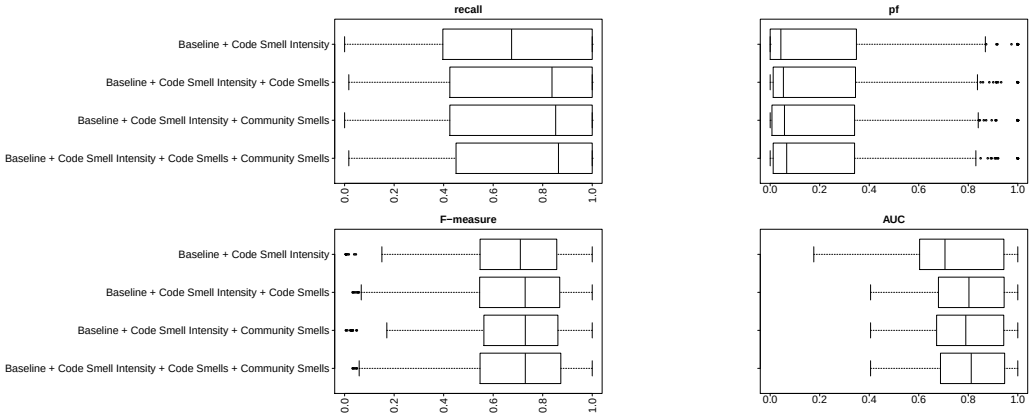

Figure 2.10 : Performances of the bug prediction models built to answer RQ3, for ReplicatedDB, with the baseline model utilizes number of developers metric.

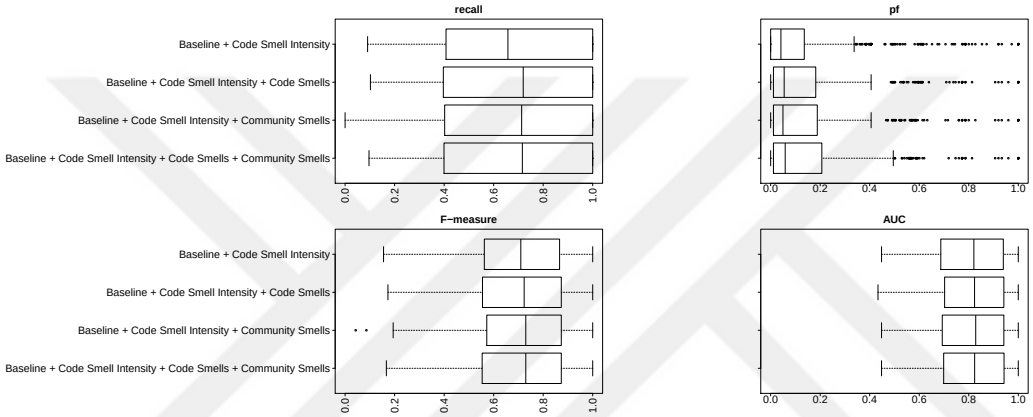

Figure 2.11 : Performances of the bug prediction models built to answer RQ3, for ReplicatedDB, with the baseline model utilizes scattering metric.

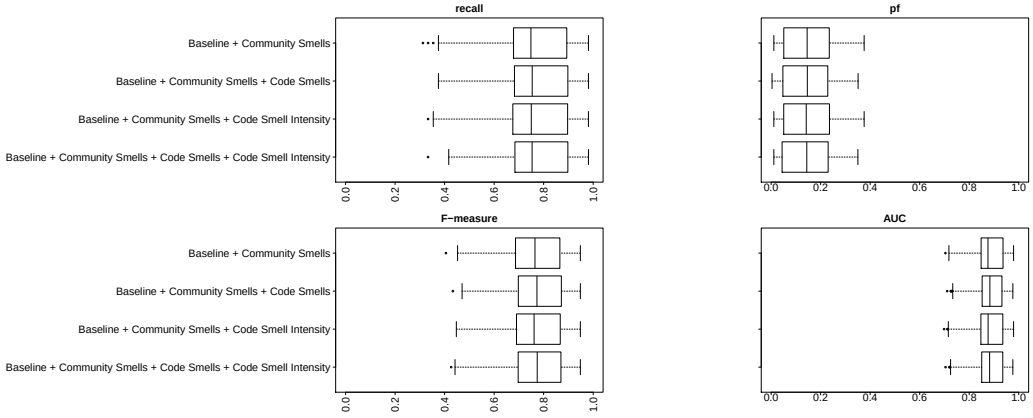

Figure 2.12 : Performances of the bug prediction models built to answer RQ3, for CollectedDB, with the baseline model utilizes structural metrics.

change the performance values for the baselines that utilize structural metrics. For example, for *ReplicatedDB*, the median recall of *+CodeInt+CodeSmells+CommSmells* is higher than the *+CodeInt* (86% and 67% respectively) when the baselines are built

with the number of developers metric. However for the baselines built with structural metrics, the median recall of *+CodeInt+CodeSmells+CommSmells* and *+CodeInt* is the same (60%) for *ReplicatedDB*, whereas the median recall of *+CommSmells* and *+CodeInt+CodeSmells+CommSmells* is the same for *CollectedDB* (75%).

Summary on RQ3: Combining smell-related information to build bug prediction models does not improve the performance of the models using the best contributing metric set.

2.5 Discussion

In this section we discuss our findings with regard to the projects in the two datasets, individual learners' performance, and the validation strategy used. Furthermore, we build and assess the models using community-aware metrics only, such as number of developers, scattering, and community smells. We also discuss the contribution of the other community-aware metrics on bug prediction models proposed in the literature.

2.5.1 Project-wise assessment of bug prediction models

Bug prediction models built for the *ReplicatedDB* and *CollectedDB* show different results in terms of the best performing metrics: The model utilizing code smell intensity information (i.e., *+CodeInt*) has a higher accuracy for *ReplicatedDB*, whereas the model utilizing community smells information (i.e., *+CommSmells* has a higher accuracy for *CollectedDB*. Furthermore, the overall performance values of the predictors built on the two datasets differ significantly: The median recall rates achieved for *ReplicatedDB* are between 20% and 69%, whereas they are between 68% and 80% for *CollectedDB*.

Section 2.4 reports the aggregated performance results of the bug prediction models for the open-source projects in *ReplicatedDB* and *CollectedDB*. Here, we also report the performances of the bug prediction models built individually on every project, and whether these findings are consistent with those reported for the associated datasets. Project-wise analysis, available in our online appendix [76], shows that bug prediction performance of the models also vary among different projects. For example, while the seven prediction models built for Xerces in *ReplicatedDB* perform between 30% and

57%, they perform 86% for Log4j in terms of F-measure. It seems the low prediction performance is related to the low ratio of the number of bug-prone classes over the total number of classes, i.e., the bug-prone class ratio of Xerces is 0.29 while the bug-prone class ratio of Log4j is 0.58. The seven models built for Velocity perform around 60% in terms of AUC, while the seven models built for the other projects of *ReplicatedDB* perform between ∼60% and ∼90%. The prediction performance of the models built for the projects in *CollectedDB* are similar to each other, i.e., the seven models built for the Cayenne, Eclipse-Cxf, and Mahout projects perform within the range of 67% and 89%, in terms of F-measure, and ∼90% in terms of AUC.

Moreover, we observe that the best performing metric sets are different in the project-specific models. We group the models according to Scott-Knott ESD test results, and compute the number of times each group outperforms the other groups in terms of F-measure and AUC. Figures 2.13 and 2.14 report the statistics for all projects in *ReplicatedDB* and *CollectedDB*, respectively. Values inside the cells represent (in percent) the number of times the corresponding model outperforms the others. Darker gray in the figures indicates that the corresponding model outperforms the majority of the models. Lighter gray means the less superiority of the corresponding model over other models. White, on the other hand, shows the models with no statistical superiority over the others.

	ReplicatedDB							CollectedDB		
	Ant	Ivy	Lucene	Log4j	Velocity	Synapse	Xerces	Cayenne	Cxf	Mahout
Baseline	50	33,3	0	0	50	33,3	0	0	0	0
+CommSmells	0	33,3	0	0	50	0	0	0	50	0
+CodeSmells	25	0	0	0	0	0	0	0	0	50
+CodeInt	75	66,7	50	50	50	66,7	50	0	0	0
+CommSmells+CodeSmells	-	-	-	-	-	-	-	0	50	50
+CommSmells+CodeInt	75	66,7	50	50	50	66,7	50	0	50	0
+CodeInt+CodeSmells	75	66,7	50	50	50	66,7	50	-	-	-
+CommSmells+CodeInt+CodeSmells	75	66,7	50	50	50	66,7	50	0	50	50

Figure 2.13 : The ranking of models with respect to the number of times they are superior than the others (in percent) among all the projects, in terms of F-measure.

From the figures we observe that *+CommSmells* beats 33% to 50% of the models in three out of ten projects (Ivy, Velocity, and Eclipse-Cxf) in terms of F-measure. *+CommSmells* significantly better than 25% to 50% of the models for the seven out of ten projects (Ant, Ivy, Lucene, and Xerces, Eclipse-Cxf, and Mahout) in terms of AUC. These ratios confirm our findings RQ1: The contribution of community smells

	ReplicatedDB							CollectedDB		
	Ant	Ivy	Lucene	Log4j	Velocity	Synapse	Xerces	Cayenne	Cxf	Mahout
Baseline	0	0	0	0	0	0	0	0	0	0
+CommSmells	25	50	33,3	0	0	25	25	0	50	50
+CodeSmells	50	25	33,3	0	0	0	25	0	0	0
+CodeInt	75	75	66,7	50	50	50	50	0	0	0
+CommSmells+CodeSmells	-	-	-	-	-	-	-	0	50	50
+CommSmells+CodeInt	75	75	66,7	50	50	75	75	0	50	50
+CodeInt+CodeSmells	75	75	66,7	50	50	75	75	-	-	-
+CommSmells+CodeInt+CodeSmells	75	75	66,7	50	50	75	75	0	50	50

Figure 2.14 : The ranking of models with respect to the number of times they are superior than the others (in percent) among all the projects, in terms of AUC.

on bug prediction is more visible in terms of AUC. +*CodeInt* shows similar ratios both in F-measure and AUC. This model beats at least 50% of the other models in seven out of ten projects. +*CodeInt* shows a greater success than +*CommSmells* when we assess project-wise statistics. The combined models also have high ratios both in F-measure and AUC among the majority of the projects. However, they cannot outperform the best performing models in project-specific models. This also confirms our findings in RQ3. Overall, project-wise analysis shows the contribution of code smell intensity on bug predictors compared to the process metrics and the other smell-related metrics. Community smells also outperforms at least one out of four groups of models, but it seems the contribution of community smells is dependent on the selected project.

2.5.2 Analysis of the models with different learners

We applied six different machine learning algorithms (i.e., *Simple Logistic*, *Multilayer Perceptron*, *ADTree*, *Naive Bayes*, *Decision Table*, and *Logistic Regression*) to build our bug prediction models. The results reported in Section 2.4 present the aggregated performances of six different machine learning algorithms. However, aggregated performances may be sensitive to low/high performing algorithms, and it restricts us to see the results of the best performing algorithms with respect to the selected metrics sets. Thus, we also report the performances of the algorithms on bug prediction in this section. The performances of the models are reported in terms of median F-measure and AUC in Table 2.7, according to the six machine learners. The results are aggregated performance of the models built on both datasets.

Table 2.7 : Bug prediction performance of the six machine learning algorithms.

Model type	Metric	SL	MP	ADT	NB	LR	DT
Baseline	f-measure	0.48	**0.55**	**0.55**	0.49	0.48	**0.57**
	auc	0.62	0.65	**0.65**	0.65	0.65	0.62
+CommSmells	f-measure	0.43	**0.56**	**0.54**	0.48	0.43	**0.53**
	auc	0.64	0.67	**0.67**	0.68	0.67	0.64
+CodeSmells	f-measure	0.41	**0.52**	**0.54**	0.49	0.42	**0.53**
	auc	0.63	0.66	**0.67**	0.69	0.66	0.64
+CodeInt	f-measure	0.57	0.71	**0.97**	0.54	0.58	**0.98**
	auc	0.67	0.8	**0.95**	0.81	0.68	**0.94**
+CommSmells+CodeSmells	f-measure	0.70	0.89	**0.79**	0.67	0.77	**0.84**
	auc	0.86	0.93	**0.94**	0.8	0.88	**0.93**
+CommSmells+CodeInt	f-measure	0.57	0.75	**0.97**	0.54	0.58	**0.98**
	auc	0.67	0.84	**0.95**	0.8	0.69	**0.94**
+CodeInt+CommSmells	f-measure	0.56	0.74	**0.98**	0.54	0.55	**0.98**
	auc	0.64	0.84	**0.95**	0.8	0.67	**0.94**
+CodeInt+CodeSmells	f-measure	0.58	0.75	**0.97**	0.55	0.57	**0.98**
+CommSmells	auc	0.67	0.85	**0.95**	0.8	0.68	**0.94**

SL: Simple Logistic, MP: Multilayer Perceptron, ADT: ADTree, NB: Naive Bayes
LR: Logistic Regression, DT: Decision Table

We highlight the performance of the statistically best-performing model (according the Scott-Knot ESD tests) in bold in the table.

Results in Table 2.7 show that *Decision Table* and *ADTree* perform considerably well regardless of the extent of the knowledge enriched in the prediction model. They even performed well just with the *Baseline* model with an F-measure rate around 57% and an AUC rate around 67%, or for the model enriched with all available knowledge (+*CodeInt*+*CodeSmells*+*CommSmells*) F-measure rate around 98% and AUC rate around 95%. This could be because the *ADTree* combines the simplicity of a single decision tree with the effectiveness of boosting, robustness, and handling of irrelevant attributes. *Multilayer Perceptron* also performs statistically the same with *ADTree* and *Decision Table* for *Baseline*, +*CommSmells*, and +*CodeSmells* models in terms of F-measure with a rate around 56%.

The *Logistic Regression*, *Naive Bayes*, and *Simple Logistic* are not useful learners for bug prediction as *ADTree*, *Decision Table*, and *Multilayer Perceptron* in our setting. The reason for that could be, for example, while using the *Logistic Regression* on high dimensional datasets, the model could over-fit the training set. Thus the model may not be able to predict well enough on the test set. Also, using *Logistic Regression*, it is challenging to capture complex relationships. Besides, for *Naive Bayes*, each feature should make an independent and equal contribution to the outcome, which might not meet most of the time in our study. Another reason could be we had significantly smaller sets of classes labeled as buggy. However, in *Naive Bayes*, the training data should represent the population well. More specifically, with no class label occurrences and a particular attribute value, the posterior probability will be zero. Thus, it is possible if the training data is not representative of the population *Naive Bayes* may not work well. Although, in some cases, *Naive Bayes* has higher AUC, i.e., when a *Baseline* model is enriched with community smells, and code smells.

Our conclusions reported in Section 2.4 do not change if the performance results of best performing learners, i.e., *ADTree*, are considered only, except for RQ1, *i.e.*, +*CommSmells* statistically outperforms *Baseline* in terms of recall in addition to AUC. The Scott-Knott comparisons of bug predictors built with different machine learning

algorithms and their performance measures, in terms of recall, pf, F-measure, and AUC, are also provided in detail in our online Appendix [76].

2.5.3 Validation strategies of the models

In Section 2.4, the performance of the bug predictors are trained by following a *within project* strategy: We combined all releases' data of a project and applied cross-validation to build and assess bug prediction models. There are other validation strategies used in software bug prediction, such as *within-release* and *cross-release*. In *within-release*, a model is built and assessed on a single release. In *cross-release*, a model is built for a release n using the prior releases' data (1^{st} to $(n-1)^{th}$). The selection of validation strategy is also considered to have an effect on the findings [100]. Hence, we conduct additional experiments to observe the performance of the models trained according to *within-release* and *cross-release* strategies.

Figures 2.15 and 2.16 report the performance of the predictors built for the *ReplicatedDB* according to *within-release* and *cross-release* strategies, respectively. Since the data collection process of *CollectedDB* is not conducted according to a release strategy, this analysis could be performed for *ReplicatedDB* only.

We analyze the performance of *within-release* and *cross-release* reported here, in comparison with the performance of *within-project* reported in Section 2.4.

The box plots show that *within-project* and *within-release* strategies perform very similar to each other. The performance of the seven bug prediction models slightly increase in terms of recall (between 3% and 8%) when they are trained with a *within-release* strategy. In *cross-release* strategy, we also observe an increase on the recall values (between 2% and 6%) of the *Baseline*, *+CommSmells*, and *+CodeSmells* models compared to the other validation strategies. However, applying a *cross-release* strategy has an adverse effect on the *+CodeInt*, *+CodeInt+CodeSmells*, *+CodeInt+CommSmells*, and *+CodeInt+CommSmells+CodeSmells* models, i.e., recall values of those models decrease between 12% and 25%.

The metric that affects the results seem to be the code smell intensity metric. Statistical tests on the seven bug prediction models with the three validation strategies also

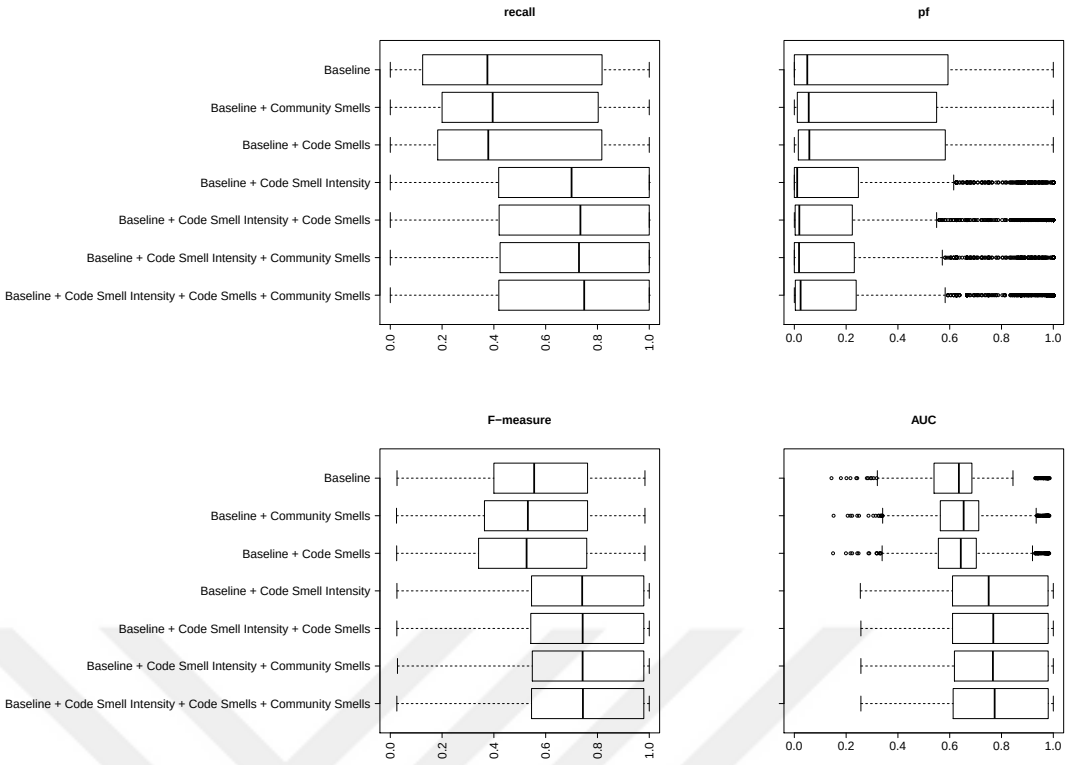

Figure 2.15 : Performances of bug prediction models trained with within-release technique on the ReplicatedDB dataset.

supports that claim. Incorporating the code smell intensity information into the other metrics does not statistically improve the prediction performance when the *cross-release* strategy is used, whereas it statistically improves the performance when a *within-release* or *within-project* strategy is used. This shows us the importance of validation strategy applied in interpretation of the findings.

2.5.4 Comparing models with community-aware metrics

In this section, we compare the performance of a model that only utilizes community smells with the performances of community-aware baseline models built in this study. Later, we report a comparison of our models with the state-of-the-art models in bug prediction literature using community-aware metrics.

2.5.4.1 Community-aware baseline models

In Figure 2.17, we compare the community smell-aware model with the two baselines, e.g., the number of developers and scattering metric. Both of these baselines

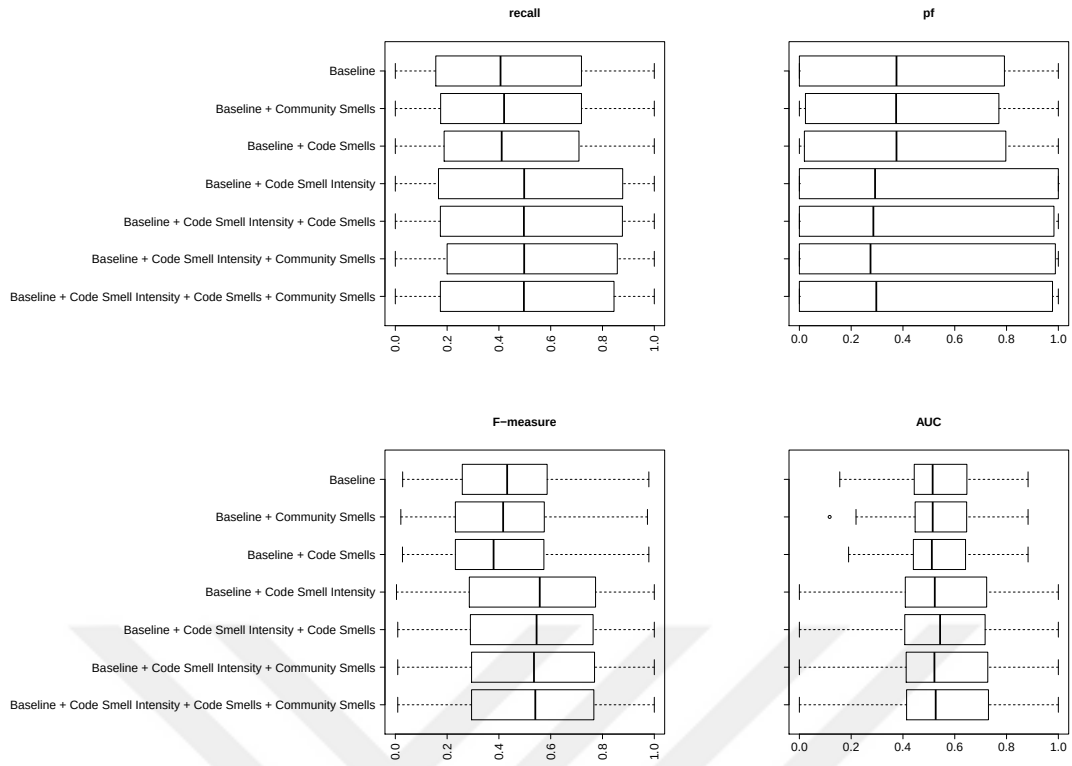

Figure 2.16 : Performances of bug prediction models trained with cross-release technique on the ReplicatedDB dataset.

utilize metrics illustrating developer activities. Although these metrics do not cover the collaborations between developers or social interactions, they are used in bug prediction studies to illustrate the effect of people aspect [68,74]. Hence, we would like to check if community smells as a baseline can produce comparable performance with those *community-aware* metrics.

Overall, the model utilizing the community smells only perform poorly than the models utilizing the other community-aware metrics, in terms of recall, F-measure, and AUC. The scattering baseline gives statistically the best performance in terms of AUC, while the number of developers baseline gives statistically the best performance in terms of F-measure. Both perform statistically the same in terms of recall. In terms of pf, scattering and community smells models performs statistically the same, and outperform the number of developers model.

The model utilizing only community smells does not outperform the other models for all the machine learning algorithms we applied. This could be due to the low ratio of community smell-prone classes in *ReplicatedDB* dataset. *CollectedDB* dataset

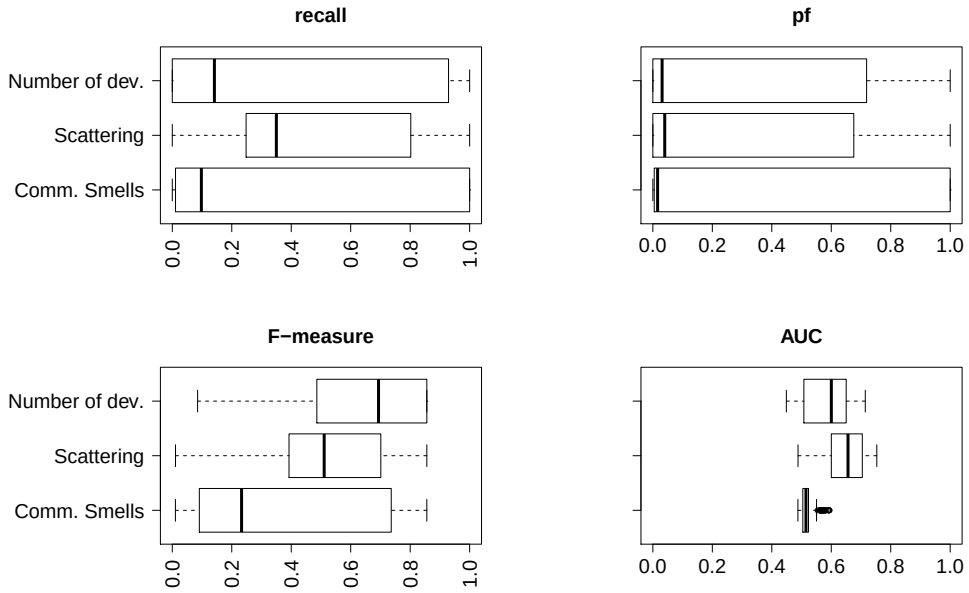

Figure 2.17 : Performance comparison among the community-aware baselines on the ReplicatedDB dataset.

does not have scattering and developer metrics, i.e., only structural metrics form the baseline. Thus, we are not able to perform a similar comparison for *CollectedDB*. At this point, it is observable that the community smell-aware model performs higher when accompanied by other process metrics, such as structural, churn, developer, and scattering metrics.

2.5.4.2 The state-of-the-art models using community-aware metrics

In this section, we show a comparison of the results of our study with the state-of-the-art community-aware metrics used for bug prediction. Table 2.8 reports the results of the studies that report bug predictors using community-aware metrics. The metric sets used in those models, classifiers, performance of the best performing model, and the improvement gained by incorporating the community-aware information in prediction model are provided in the table.

Nagappan et al. [79], Bird et al. [22], Bell et al. [74], and Di Nucci et al. [68] incorporate the community-aware information into their baseline models. Their studies report up to 56% improvement depending on the projects, leading to recall rates between 64 - 81%, when the community-aware metrics are included. Soltanifar et al. [89], and Kını and Tosun [61] build a combined model using community-aware

metrics and other state-of-the-art metric sets, e.g., process metrics. Both studies report a recall rate around 75%.

The studies listed in Table 2.8 utilize various community-aware information, i.e., socio-technical networks, organizational structure, number of developers who previously changed a code module, developer's focus on a software component, and developer's experience on a software module. All those studies report improvement in bug prediction. In this study, we contribute to the bug prediction literature by investigating the effect of community smells, which is another type of community-aware metrics, on the prediction of bug-prone software classes. Our findings show that adding community smells into *Baseline* and +*CodeInt* improves the bug prediction performance from 71% to 75%, and from 62% to 68%, respectively.

2.5.5 Practical implications

Our study brings a novelty into socio-technical relationships observed for bug prediction. Community smells capture both communication and collaboration patterns between the developer communities. Knowing these patterns would not only help the communities improve their work, but also gives insights on bug-proneness of their codes. Our results conclude that incorporating community smells into the models built with process metrics may give better bug prediction performance. However, we could not observe this in many projects. Although community smells are associated with code smells and code smell intensity in earlier works [55,71] , the latter metric seems to be the best indicator of bugs for many projects. Code smell intensity incorporated into process metrics could be considered as the first choice for practitioners who aim to build bug prediction models.

We further inspect the predictions made by the model that incorporates the community smells and the model that includes code smell intensity metric into the baseline metrics, and the development and communication history of software projects used in our experimentation. Our goal is to gain insight on the cases in which the community smells would be useful or not on the prediction of bug-prone classes.

Table 2.8 : Comparison of performance and methodology with the community-aware state-of-the-art studies.

Studies	Subject Systems	Community-aware Metrics Used	Other metrics Used	Classifier(s) Used	Performance (Recall)	Improvement (Recall)
[79]	Windows Vista	Organizational structure	Code churn, complexity, dependency metrics	Step-wise regression, Principle Component Analysis (PCA)	84%	5-30%
[22]	Windows Vista, Eclipse (6 releases)	Developer contribution network, socio-technical network	component dependency network	Logistic regression	87%	4-17%
[74]	Seven industrial system	Number of developers	-	Negative Binomial Regression	-	*13%
[31]	7 Apache projects	Developer's focus	-	Negative Binomial Regression	**64%	-
[89]	Mozilla Core module	Number of developers & interaction of developers	Product & process metrics	Logistic Regression, Naive Bayes, Bayesian Network	76%	-
[61]	2 open-source systems	Periodic developer experience metrics	Code churn	Decision Tree & Random Forest	75%	-
[68]	26 Apache projects	Developer's scattering	Number of developers & code churn & static code metrics & developer's focus	Decision Table Majority	81%	4-56%
Our study	10 open-source systems	Community smells	Structural, churn, number of developers, developer's scattering	ADTree, Decision Tree, Naive Bayes, etc.	75%	6%

*13% improvement of [74] models is reported in terms of fault-percentile average
** [31] does not provide prediction model, the performance value for the bug prediction model that utilizes the developer's focus is reported by [68].

Our observation on the predictions made by +*CommSmells* and +*CodeInt*, show that 74% of the total software classes are correctly classified as buggy or clean by both +*CommSmells* and +*CodeInt*, whereas 22% of the total software classes are misclassified by both models. Also, the cases that only one model (i.e., +*CommSmells*) make correct prediction when the other (i.e., +*CodeInt*) cannot are 2% of all data instances, for each model. Most software classes that cannot be correctly labeled with +*CodeInt* have zero code smell intensity values. Those incorrectly classified classes by +*CommSmells* are not associated with a community smell, except a few instances. Therefore, we think the limitations of the existing tools used to capture community smells of a software project play a crucial role in the success of +*CommSmells*.

The findings on community smells could be influenced by how they are measured in prior studies [55,71] as well as in this work. It is a big challenge to capture communication and collaboration patterns in developer communities. The existing tool *CodeFace4Smell* assumes that the communication is handled via mailing lists, and the collaboration is found in the co-changed modules in commit logs. In practice, communication channels in the projects might not be accessible or they may be distributed along different systems. For example, Log4j releases used in our experimentation only have development history until 2002 and ten developers have been contributed to the project until that year. The lower number of contributors may be the cause of lower number of community smells [54]. On the other hand, the Eclipse-CXF development team seems to use IRC communication logs, whereas the team of Ivy uses another forum in addition to their mailing lists. Using multiple resources to communicate with teammates makes it difficult to capture the communication patterns. Considering that, *CodeFace4Smells* is limited to extract the communication patterns for the projects, and improvements on the community smell detection strategies, specifically for capturing communication patterns, are necessary.

2.6 Threats to Validity

In this section, we discuss the threats to the validity of our study by following the guidelines reported by Yin [101]. *Construct validity* refers to the relation between theory and observations that might be jeopardized through measurement errors. We

tried to eliminate measurement bias by following previously published papers on community smells and their calculation process via the *CodeFace4Smell* tool.

We face various challenges during the collection of our datasets and metrics. One challenge is reaching the necessary information for some projects, i.e., mailing lists, and source code repository, to extract community and code smells. Historical collaboration and communication information of some projects we took from Palomba et al.'s study [56] is not available since those projects are old (between the years 2000 and 2008). Also, the first releases of projects in Palomba et al.'s study [56] are not included in our dataset since they do not have collaboration and communication history. Therefore, we eliminate those releases and projects whose mailing list and/or source code repository is not available in order to avoid potential construct validity threats to our empirical findings. A configuration file is constructed for each project in *ReplicatedDB* by listing the commit id associated with all releases of a project included in our dataset. We double-checked the release dates of the projects from the projects' websites and source code repositories before associating with a commit id. We also ensure that the time interval between releases is at least three months since it is considered to be sufficient time to capture the development of community patterns [102]. We match the output of *CodeFace4Smell* with the classes in the original dataset of Palomba et al.'s study [56] linking releases and the software class names. We eliminate unmatched class names between the original dataset and the *CodeFace4Smells* tool results to form the final version of the *ReplicatedDB*. The final version of the *ReplicatedDB* contains 16 releases of the seven open-source systems. Configuration files used for *ReplicatedDB* are included in our online appendix.

Configuration files of the *CollectedDB* are taken from a prior study [69] whose authors propose a database that contains a list of open-source projects with their corresponding configuration files, downloadable links to the related source code repository and mailing lists. The configuration file of a project lists 13 snapshots of the project with three-month time intervals. We chose three open-source projects from the list to build *CollectedDB*. Later, we extracted the structural metrics by using the Ckjm tool, which uses the compiled Java codes to extract the static code attributes. We eliminate those software classes that we cannot extract the metrics for, due to compilation errors.

Conclusion validity refer to the relation between the treatments and the outcomes. To minimize the threats to the conclusion validity, we perform several adjustments on experimental design: The prediction models are built using six different machine learning algorithms – *Simple Logistic*, *Multilayer Perceptron*, *ADTree*, *Naive Bayes*, *Decision Table*, and *Logistic Regression*. Validation of the models is carried by applying a 3-fold cross-validation technique. Before each repetition, the dataset is randomized to avoid data order bias. All prediction models built in our experimentation are evaluated in terms of recall, pf, F-measure, and AUC. We used these evaluation metrics because they are widely used in literature, e.g., [56]. We applied the Scott-Knott Effect Size Difference (ESD) test to conduct a statistical comparison among the performances of the prediction models [82]. The findings are also confirmed among the two datasets containing ten projects in different sizes, bug ratios and smelly class ratios. We observe that the conclusions are subject to change depending on the project, performance measure or the validation strategy used. We report our findings accordingly.

Internal validity is related to the factors that are not considered during the experimentation but may affect the conclusion of the study. Some of these might be related to the metrics used from the prior study's dataset, or measurement procedures. Tools and techniques used to extract the metrics may impact the performance of bug prediction models built. We relied on the metric sets of the original dataset taken from our base study [56]. The baseline metrics are popularly accepted and used in prior bug prediction studies. We also relied on the *CodeFace4Smells* tool for detecting code and community smells from the given projects [55]. It is still possible that the community smells would be different among classes if a different tool was utilized. However, the current state-of-the-art tool for detecting those is *CodeFace4Smells*, and based on our analysis on extracted metrics, it sufficiently captures the collaboration and communication patterns among the developers. The methodology is carefully defined according to our RQs, and only the comparisons between the models are conducted to answer these RQs.

External validity deals with the generalization of the outcomes of the experimentation. Our analysis made on ten open-source projects that are widely used in the related

literature, e.g., *ReplicatedDB* contains datasets from Promise repository [47], and *CollectedDB* projects are used in [55]. To minimize the threat to the external validity of our findings, we analyze the contribution of community smells on bug prediction on these two datasets including multiple open-source projects. The findings confirm that dataset characteristics significantly affect the conclusions. While our experimentation covers the widely used metrics and open-source projects, further investigation can be done for industrial projects to generalize the results.

2.7 Conclusion

In this chapter, we empirically analyze the effect of community smells on bug-proneness of software classes by comparing the findings against the other metrics, i.e. process metrics, code smells and code smell intensity, used in bug prediction literature. During this research, two datasets are constructed and shared online with the community: (1) *ReplicatedDB* is a new bug prediction dataset constructed by collecting community and code smells related information of classes of seven open-source systems and (2) *CollectedDB* provides class-level code and community smells information of six open-source systems [76]. Bug prediction models are built on both datasets to investigate the contribution of including community smells into state-of-the-art metric sets, e.g., structural metrics [72], number of developers [74], code churn [73], developer scattering metrics [68], in comparison with the other smell-related metrics, i.e., code smells [103] and code smell intensity [56].

The performance of the bug prediction models indicates that community smells contribute to the prediction of bug-prone classes on both datasets (statistically improved AUC by up to 3%), while it does not contribute as much as code smell intensity metric: For *ReplicatedDB*, *+CodeInt* outperforms *+CommSmells* by up to 17% in terms of AUC. Our experimental findings show that community smells and code smell intensity are good indicators of bug-prone modules depending on the project characteristics and experiment validation strategies. This support the findings of Palomba et al.'s study [56] on the success of code smell intensity on the prediction of bug-prone software classes. However it does not always hold for all kinds of projects. Both smells represent different aspects of technical and communication

flaws, and should be observed together. In other words, technical flaws encountered during the design and development of the software, and communication flaws of the software development team significantly affect the bug-proneness of software classes. Measuring those kinds of flaws by utilizing proposed metrics and tools in the literature [55,56] and combining them into bug predictors, lead to build more successful bug predictors.

Measuring community smells requires a serious effort, even if there is a readily available tool, i.e., CodeFace4Smells, to detect smells. Detection of community smells involves the analysis of communication and collaboration patterns of the development team. Capturing collaboration patterns is easier than capturing communication patterns since it requires access to the source code repository, while capturing communication patterns requires access to the communication channel. Most open-source projects have a structured communication channel like mailing lists, but not every organization has a documented communication channel in real life. Furthermore, a communication channel, i.e., mailing archives, may not represent all communication lines between the organization members. Therefore, we conclude that modeling people factors for software recommendation systems is a very challenging task. Also, our experimental findings indicate that community factors are not good indicators of bug-prone software classes as other metrics measured from source code, i.e., code smell intensity. These lead us to assess people factors in defect prediction models from another aspect. Instead of utilizing community factors as an information source during the model training, it would be better to employ individual development skills of the team by building personalized models for each of them. Therefore, in the next two chapters, we investigate the personalized defect prediction approach.

3. INVESTIGATING THE PERFORMANCE OF PERSONALIZED MODELS FOR SOFTWARE DEFECT PREDICTION[2]

3.1 Introduction

Studies on software defect prediction (SDP) propose automated tools that mine historical development data from version control systems, source code bases and issue management systems and identify development patterns to recommend defect-prone modules in software systems. For more than two decades in empirical software engineering, researchers are building SDP models using different datasets, different input features, and different algorithms to catch more bugs compared to manual code review and other bug detection activities performed by the development teams [10,11]. The recommendations given by SDP models to assist developers are discovered to improve the quality of the produced software [104,105].

Software systems produced by different developers tend to have distinguishing defect patterns [17] since each developer has unique characteristics regarding their development styles, experience, and defect proneness of the code they produce. Developers' profiles and their way of working can be quantified through a set of features, including but not limited to years of experience, age, gender, collaboration with other developers, commit activity, quality of the code produced, and bug resolution activity. Some of these features have previously been used as developer metrics in SDP studies. For example, the number of developers that changed a code module [106] is included in the prediction models to incorporate the developer aspect into defect prediction. In another study, developer focus is quantified and included into the defect prediction models [107]. Gender is also studied in bug prediction to assess the effect of gender on introducing smells [94].

Recommendation systems targeting other businesses, e.g., e-commerce websites, search engines, and social media platforms utilize users' features such as search history

[2]This chapter is based on the papen, B., Tosun, A. (2021). Investigating the performance of personalized models for software defect prediction. Journal of Systems and Software, 181, 111038.".

or locations to give customized and more useful search results or advertisements [37,38]. The software engineering field also adopts the techniques used in recommendation systems to improve the process of software systems [40], e.g., a recommendation system to decide which source code modules need to be modified [5]. Recently, instead of quantifying developers' characteristics as a set of metrics, personalized defect predictors have been built separately for each developer by utilizing the corresponding developer's change history only [36,53,88]. While general defect prediction models utilize the whole change history produced by the team, a personalized model utilizes less data during training compared to a traditional model. Moreover, predictions made by these personalized SDP models target the relevant developer only. By making developer-specific predictions, researchers aim to improve the usability and accuracy of defect prediction tools.

The personalized SDP models proposed so far (e.g., [36,53]) give promising results in terms of defect detection rates. These models lead to investigate fewer files modified by a selected developer only, and still catch more defects than the general models. The conclusions drawn from these early personalized models are hard to generalize due to certain assumptions made in their methodology. In particular, the personalized model findings are limited in terms of reported performance measures, the used metrics, and model construction strategies. First, the proposed models in the literature could reach up to 7% increase in F1-measure and 8% increase in recall, however, the effect sizes of these performance values and individual personalized models' performance were not discussed in detail. Second, well-known metrics that characterize bugs in SDP, e.g., lines of code, code changes, or the number of developers, were not used while building personalized models, but the source codes were transformed into characteristic word vectors. Third, developer selection and data collection strategies are ad-hoc, i.e., 100 commits from 10 developers were used for model building. Considering that SDP models still lack generalizability in different contexts, the results suffer from external validity of their prediction performance [108]. We believe the need for further investigation on the benefits and the general applicability of personalization in SDP.

In this study, we aim to investigate the performance of personalized SDP models compared to those of traditional approaches in an empirical setup on six open-source

software projects: Gimp, Maven-2, Perl, PostgreSQL, Rails, Rhino [27]. We utilize change-level defect prediction datasets of those projects, each has 9 to 26 years of development history and embody the state-of-the-art process metrics [30,109]–[113]. Our study extends and contributes to the previous work on personalized defect predictors in the following aspects:

1. We investigate a total of 222 developers over six large-scale open source projects, and build personalized models for each of these selected developers. In RQ1 (Section 3.3), we analyze the performance of personalized models against the traditional approaches, namely a general model (GM) and a general model for a selected set of developers (SM). The developer selection strategy that we follow is based on previous discussions on the size of training data in SDP [100], and data availability in recommendation systems [114].

2. To increase the conclusion validity of our empirical analyses, we evaluate our proposed models with respect to seven performance measures, namely probability of detection, probability of false alarm, precision, F1-measure, Area Under ROC Curve, Matthews Correlation Coefficient, and Brier score. We also utilize additional assessment techniques, such as effect size calculation and Nemenyi's test, for model comparisons. Our findings are based on statistically significant differences with medium to large effect size only.

3. We conduct an empirical analysis on the development characteristics of individual developers with respect to 1) the number of commits made by the developers, 2) the ratio of bug-inducing commits to the total commits of the developers, and 3) the process metric values. We would like to understand whether these characteristics reflect when personalized models are superior over traditional models, or vice versa (RQ2 in Section 3.3).

4. We conduct a feature analysis using Information Gain on the process metrics used to predict defects in a personalized model, and assess the distinguishing features among personalized models (RQ3 in Section 3.3).

Empirical analyses on 222 developers of six open-source projects show that personalized defect prediction models improve the traditional models' probability of detection rates up to 24% for 83% of the analyzed developers. Even though overall results show the superiority of the personalized approach, personalized models do not improve the prediction performance for every developer per se. We observe that the developers whose traditional model outperforms the personalized model have a higher experience (i.e., more contribution as commits) on their projects. On the other hand, the personalized models are more successful than the traditional models for those developers who contribute to the files that were modified by many developers. Also, the importance rank of the process metrics differs among 222 developers, except the fact that the size metrics (i.e., added lines of code) are the most important ones for the majority of developers. Finally, the performance of the personalized model highly depends on the selected machine learning algorithms and performance assessment criteria.

3.1.1 Structure of the chapter

Section 3.2 reports the related work on personalized defect prediction literature. In Section 3.3, we report the experimentation conducted in this study in detail. Section 3.4 discusses the results of the experimentation conducted to answer our research questions, while Section 3.5 discusses the outcome of our experimentation from various aspects (e.g., machine learning algorithms applied, model validation techniques). In Section 3.6 threats to the validity of our findings are discussed. Section 3.7 summarizes the key take-away messages and reports future research directions.

3.2 Related Work

In this section, we report several change-level defect prediction approaches over the history of SDP models. In the subsections below, we explain the evolution of SDP models with respect to using people related metrics, and discuss prior studies that propose personalized SDP models.

SDP models are designed to make predictions at different granularity levels, e.g., file, class, method, and code change level [105]. A change-level SDP model makes it easier

to find the responsible developer for a bug-prone software entity [16]. Majority of the change-level SDP studies in the literature utilize developer experience, history of the changed files, diffusion of code changes, and size of changes [16,27,30,115]. Mockus et al. [30] propose models that predict the high risky (being prone to failure) initial maintenance requests (IMR) that contain multiple software changes. Their analysis on a large scale telecommunication system shows that the number of subsystems modified, and developer experience are found to be indicators of these high risky changes. Similarly, Shihab et al. [115] also use change-level software metrics, such as the time when the change is made, the purpose of change (bug fix or not) in addition to size, history, and experience metrics to predict risky changes. They show that the developer experience, number of added lines, bug-proneness of the modified files, number of bugs, and the number of bug reports linked to a change are found to be good indicators of a change's risk-proneness.

Sliwerski et al. [116] propose a method called SZZ to locate fix-inducing changes of software development by using the software archives and bug reporting systems. The SZZ algorithm contributes to the SDP field by providing a new data source to researchers and practitioners of the field: bug-inducing changes [2]. Later, Kim et al. [117] classify the code changes as bug-inducing or clean. Their prediction model utilizes textual features of the modified file and directory names, and change metadata information (i.e., commit hour, commit day), complexity metrics of the modified files in addition to the size metrics. Kamei et al. [16] utilize diffusion, size, history, purpose, and developer experience metrics to predict bug-inducing changes of six open-source and five commercial software projects. Their empirical assessment demonstrates a reduction in the code review effort (i.e., 35% of all predicted bug-inducing changes could be identified by spending 20% of the total effort) using the prediction model and a recall of 64%.

Researchers have so far approached the SDP problem from different perspectives, i.e., from modeling the algorithm and validation techniques to enriching the data, and to modeling the people. Various statistical and machine learning techniques are utilized [11,105], i.e., Logistic Regression (LR), Naive Bayes (NB), and Random Forest (RF) algorithms are widely used and performed quite well on predicting defects. Some

studies focus on assessing the experimental setup to depict the data more accurately, i.e., investigating other model validation techniques [82], algorithm tuning [52], data pre-processing approaches [118], enriching the data by adding new metrics, and new class types [27,51]. Recently, SDP models that make predictions at a fine-granularity level are proposed, such as predicting bug-prone files [28] or code lines [119,120] in software changes.

More recent studies consider the effect of developers on defect proneness of software modules [17,31,32,106,121]. As people are the third essential unit in software development, in addition to product and process, researchers investigate software developers to model their development behavior, interactions among each other through the modified source code modules, and other communication channels. Besides, new studies are putting the whole focus on the developers, and propose personalized SDP models that aim to give customized prediction results to each developer in a team [36,53]. Customized feedback is provided by separately-built models for each developer instead of including developer metrics into general SDP models. Below, we report studies that use developer metrics to build defect prediction models in Section 3.2.1, and build personalized defect prediction models in Section 3.2.2.

3.2.1 Defect prediction using developer metrics

Schröter et al. [121] report differences in the bug density of source code files developed by different developers. According to their study, specific developers more likely generate bugs than others, and this reflects the complexity of the code rather than a competency between developers. Later works study the relationship between the defect density of code modules and the *experience* of the developer. The more experienced developers develop more complex code because more risky, larger and thus more complex tasks are assigned to more experienced developers [122,123]. Further, the results of Eyolfson et al.'s study [124] show that more experienced developers tend to introduce fewer bugs. Rahman and Devanbu [125] report that there is no clear correlation between bugginess of the code and the developers' overall experience on

the project. On the other hand, their study states that a developer's experience on one file is more important than the developer's overall experience on the project.

The *number of developers* who worked on a software module is popularly used as one of the metrics in SDP models [17,74,106,110]. A study by Ostrand et al. [17] reports that including the number of developers who modified a code module into the predictor makes a modest improvement in the performance of defect prediction. Matsumoto et al. [110] indicate that developers' defect injection rates vary, and modules edited by many developers contain more defects.

Posnett et al. [31] proposed *developer focus* to refer to the activity of a developer. The authors measured the focus of a specific developer to a specific module, and concluded that more focused developers would cause fewer defects. Di Nucci et al. [107] extended the *focus* metrics by adding a distance measure between the modules. The smaller the distance between the modules means that the modules are more related to each other. The proposed model outperforms their baselines including the study of Posnett et al. [31].

Lee et al. [32] proposed a metric set that models the developers' *interaction behaviour at a micro-level*. Micro interaction metrics are collected from Mylyn, such as browsing and editing times of code files, frequencies and edited file information. They concluded that micro interaction metrics improved the prediction performance when they were used with source code and change history metrics. Also reported in [32], the number of developers that edited a file in the history, the number of edit events observed for a file, and the number of selections of a file are relatively good predictors.

Calikli et al. [33] also modeled developers in terms of their *confirmation bias* levels to identify their relationships with the post-release defects. For instance, testers may exhibit confirmatory behavior in the form of a tendency to make the code run rather than employing a strategic approach to make it fail.

3.2.2 Personalized defect prediction

Bettenburg et al. [126] claim that software engineering data contains a large amount of variability and models built with a global context are often irrelevant and less

successful than models built with a local context. They applied Multivariate Adaptive Regression Splines (MARS) [127] which is a global model that considers the local regions in the data. Their results show that the MARS approach outperforms the traditional global approach in defect prediction.

We identified two studies proposing personalized SDP models [36,53]. These personalized models are built at change-level rather than file/method level since the ownership can be defined easier on a code change than a file. A file may be modified by several developers, and hence, a defect in a file may be the consequence of these code changes, whereas a code change is only associated with a single developer [16,27].

Jiang et al. [36] propose change-level personalized SDP models for six different open-source projects, namely Linux kernel, PostgreSQL, Xorg, Eclipse, Lucene and Jackrabbit. They identified 10 developers who contributed the most to these projects. Then, 100 consecutive commits of each developer were collected and used for building the personalized prediction models separately, while a total of 1000 commits of all 10 developers are used for building the traditional, general prediction model. In addition to the personalized and general SDP models, a weighted model which combines different developers' data, and a meta-classifier which ensembles the predictions of general and personalized approaches are built. They employed Alternating Decision Tree (ADTree), Naive Bayes (NB) and Logistic Regression (LR) algorithms to build their models.

Jiang et al. [36] built their models utilizing characteristic vectors that represent the syntactic structure of the changed source codes in a commit, and bag-of-words for both commit messages and source code. Moreover, commit hour and day, cumulative change and bug-inducing change counts, source code file/path names, and file age were the other input metrics. Their results indicate that the F1-measure of the personalized model was 1 to 6% more than the traditional (general) model among six projects. Besides, when the top 20% of defective lines of code are inspected, a personalized approach could detect up to 155 more defects than a traditional approach.

Xia et al. [53] propose an alternative personalized approach that utilizes other developers' commit history, as other developers' commit history could be useful to

predict the defects of a specific developer. That idea is similar to the weighted personalized approach in [36], but instead of randomly selecting the other developers' data to build a training set, other developers' change data was included in the training set utilizing a genetic algorithm. The authors in [53] experimented on the same datasets used by Jiang et al. [36] and filtered the same number of developers (10) and the same number of commits (100 each). They also used the same metric set used in Jiang et al.'s study [36]. The results reveal that their proposed personalized solution could reach up to 13% higher F1-measure than the traditional models and detect up to 245 more defects than the previously built models in [36].

Early personalized approaches for SDP [36,53] report that personalized defect predictors could reach up to 13% higher F1-measure rates than the general models. On the other hand, the recall values reported in [36] are not very high, between 39% and 74%, compared to the previously reported defect predictors [105], while the precision values reflect that false alarm rates may also be higher. Their data selection and developer selection techniques are strict, i.e., only 10 developers and 100 commits from those developers were chosen to build the personalized models. Furthermore, the general models built in [36,53] only utilized the selected developers' commits whereas the non-selected developers' development history are not included in the general SDP models. To further understand the intrinsic characteristic of personalized approaches we need to observe the results of personalized approaches under different circumstances. Thus, in this study, we aim to investigate the performance of personalized defect predictors with a different experimental setup regarding data sets, metrics, model construction and additional performance measures.

3.3 Experimental Setup

In this study, our objectives are to build *personalized* defect predictors using individual code changes of developers, to assess the models' performance against the traditional models, and to understand the factors that may have an effect on the performance of personalized predictors. We would like to evaluate the usefulness of the personalized approach by setting up an improved empirical setup that conforms to the state-of-the-art change-level SDP. We believe that the findings of this study would

contribute to software practitioners while deciding whether the personalized approach in the context of SDP can be worth building. We define three research questions (RQs) as follows:

RQ1: How does a personalized SDP approach perform compared to traditional SDP approaches?

RQ2: To what extent do development characteristics have an effect on the superiority of PM?

RQ3: How does the importance of metrics used for defect prediction differ among personalized models?

Please note that for RQ2, we define development characteristics over commit activities of developers: 1) number of commits of a developer, 2) ratio of bug-inducing commits to the total number of commits of a developer, 3) metric (Table 3.1) values of the commits of a developer, and 4) the importance rank of the metrics for a developer. More details on the methodology of each of our research questions are reported in Section 3.3.2.

3.3.1 Dataset details

We conducted our research on six datasets containing historical commit information of six open-source projects, namely Gimp, Maven-2, Perl, PostgreSQL, Rails and Rhino. This dataset is collected in a prior study whose steps are described in [27]. The dataset contains five types of metrics at commit (change) level; the size, history, experience, diffusion and purpose related metrics, and bug-inducing commits were extracted from the commit histories of the six projects. The full list of metrics is given in Table 3.1. The *Size* metrics represent the amount of change during a commit. The *History* metrics represent the number of developers that changed the modified files in a commit, the change history of the modified files in a commit and the average time interval between a commit and the last change time of modified files in the commit. The *Experience* metrics are calculated based on prior commits of the developer who made the commit. The higher the number of previous commits of the developer, the higher the value of

experience metrics. The *Diffusion* metrics represent how a change is spread across source files. These metrics are calculated by counting the number of modified files, subsystems, directories in a commit and calculating the entropy of each commit. The *Purpose* dimension involves a single metric named as FIX that represents whether the purpose of a commit is bug-fixing or not (binary). The detailed explanations of the metrics can also be found in the previous work [27]. We think the selected projects represent rich information in terms of their development periods (e.g., from 1987 till 2013 for Perl) and the extracted metrics. The dataset contains 13 well-known process metrics which measure the commits through various aspects. Previous studies also report that the process metrics extracted from the software projects contain rich and useful information for defect prediction models, and these perform better than code metrics [16,27,108]. Therefore, we chose this dataset to make a better comparison of our approach with the prior studies in terms of prediction performance as well as to increase the replicability of our methodology on publicly available projects. Building a new dataset was not the focus of our study. Instead, we selected an existing dataset that has already been validated in the context of change-level SDP studies.

Table 3.1 : Software metrics used in this study.

Metric	Description
Size	Lines of code added in a commit (ADD)
	Lines of code deleted in a commit (DEL)
History	Number of developers had changed the modified files (NDEV)
	Number of prior changes to modified files (NPC)
	The average time interval between the last and the current change (AGE)
Experience	Experience of the developer (EXP)
	Recent experience of the developer (REXP)
	Experience of the developer on a subsystem (SEXP)
Diffusion	Number of modified files (NF)
	Number of modified subsystems (NS)
	Number of modified directories (ND)
	Entropy: distribution of modified code across each file (ENT)
Purpose	Is the purpose of a commit to fix a bug? (FIX)

The date range of the commits, the number of total and bug-inducing commits, and the number of total developers for all the projects are given in Table 3.2. Besides, it also reports our developer selection statistics which we mention in the next section.

3.3.2 Our methodology

We share the same initial goal with Jiang et al. [36] of building personalized defect predictors, but our methodology differs in many aspects: Selecting the developers to be modeled for personalized prediction, the data sampling approach, the algorithms applied, and the performance assessment methods. We focus on an empirical evaluation of personalized models with traditional models on different open-source projects as the prior study, but also we analyzed whether the importance of metrics used for prediction differs across developers and whether the development characteristics (i.e., developer experience) affect the performance of the personalized approach.

All of these aspects are explained below. Figure 3.1 illustrates the steps of model building.

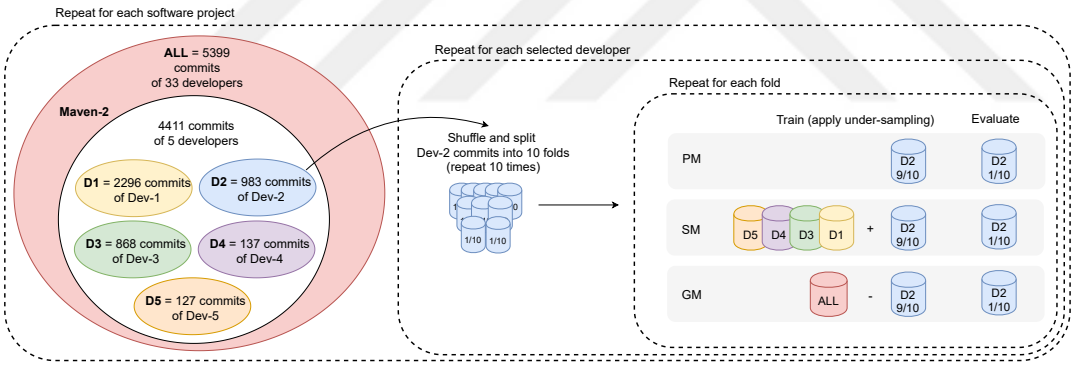

Figure 3.1 : An overview of the model building methodology.

3.3.2.1 Developer selection

We pick a specific number of developers from each project to build their corresponding personalized defect prediction models. Our aim is to select as many developers as possible while keeping a sufficient number of total and bug-inducing commits that belong to each developer for building specialized defect predictors. Indeed, a good amount of data is needed to build a successful predictor [128]. Although we cannot make a generalized comment on the number of data instances used during the training

Table 3.2 : Details of the dataset.

	Gimp	Maven-2	Perl	PostgreSQL	Rails	Rhino
Time period of commits in years	1997 - 2013	2003 - 2012	1987 - 2013	1996 - 2013	2004 - 2013	1999 - 2013
Total number of commits	32.875	5.399	50.485	35.005	32.866	2.955
Total number of bug-inducing commits	11.940	551	12.172	13.511	6.224	1.291
Total number of developers	499	33	1116	38	2287	35
Statistics regarding developer selection						
Number of selected developers	51	5	87	27	45	7
Total number of commits of all selected developers	28.286	4.411	45.876	34.848	22.592	2.801
Total number of bug-inducing commits of all selected developers	11.057	514	10.954	13.491	5.223	1.254
Range of commits of selected developers'	45 - 9.222	127 - 2.296	47 - 7.850	47 - 12.755	53 - 3.314	77 - 1.095
Range of bug-inducing commit ratio of selected developers' (between 0 and 1)	0.08 - 0.8	0.05 - 0.2	0.05 - 0.65	0.13 - 0.8	0.02 - 0.53	0.17 - 0.62

of a machine learning model, a prior study [100] in our field report that using 100 data instances would be enough to learn for an adequate defect predictor.

Cold-start problem is also another common problem in personal recommendation systems, and it occurs when there is not available knowledge or data to make a recommendation [39,40], i.e., making a recommendation for a not yet rated movie or a new user in a movie recommendation system [114]. In our context, we face the cold-start problem in situations such as when 1) a developer might have not any prior commit, i.e., she has joined to the software team and/or just started to contribute to the software project, 2) a developer might have not any enough prior commits and/or bug-inducing commits to build a personalized model for her even though she is an active contributor to the project (i.e., 49% of all developers over six projects have only one commit, and a developer from Rails has 87 commits but only two of them are bug-inducing).

Considering all these, we select the developers among all the contributing developers whose total number of commits and bug-inducing commits are at least 45 and 10, respectively. When we apply an under-sampling (Section 3.3.2.3) on training data with 10-fold cross-validation, the number of total commits for a developer in his/her training set would be at least 18 (9 of 10 bug-inducing commits and 9 of 35 clean commits). This number can be quite low to train a machine learner, so we also check the prediction performance of personalized models for those developers with very few data instances. We further discuss this in Section 3.6.

We ended up having a total number of 222 developers over six projects. Since the developer selection criterion is applied to each project separately, the selected number of developers differs among the projects, e.g., 87 in Perl and 5 in Maven-2. Still, we cover the majority of the commits, i.e., 87% of the total commits over six projects.

Table 3.2 reports the dataset details regarding developer selection process: (i) the number of selected developers for each project, (ii) total number of commits made by all the selected developers, (iii) total number of bug-inducing commits made by all the selected developers, (iv) range of the number of commits made by each selected

developer, (v) range of the ratio of bug-inducing commits to all commits of each selected developer.

The reported numbers confirm that the Pareto principle is valid in software projects [129]: the majority of the commits are made by the minority of the developers. The percentage of the selected developers over all developers ranges between 2% to 71%, whereas the percentage of the number of commits of those selected developers over all commits ranges between 70% and 99% for six projects. 49% of all developers over six projects contributed to the project with only a single commit. Moreover, the numbers show that the bug-inducing commit ratios over the total commits of a selected developer range between 0.02 and 0.8.

3.3.2.2 Model construction

Three different SDP models are built for the selected 222 developers by using two different machine learning algorithms, namely Naive Bayes (NB) and Random Forest (RF). These algorithms have been known as popular and well-performing in defect prediction [50,105]. So, we utilized these two algorithms using a 10x10-fold cross-validation technique to build our prediction models. Our models are designed to provide predictions at change level.

Personalized model (PM): We propose a personalized defect prediction model for each selected developer in six projects. A personalized model includes only the corresponding developer's commit history to training, and it is intended to make predictions only for the relevant developer at the commit level.

General model (GM): We build a general defect prediction model for each of the six open-source projects. A general model incorporates all commit history of the projects into the training set to build a single model which corresponds to the traditional defect predictors in the literature so far. This general model provides recommendations at the commit level to every contributor in the selected projects' software teams.

General model for a set of developers (SM): We also build additional general models for these projects by narrowing down the whole commit history to the commits of the most active developers instead of utilizing all developers' commit history. This

approach is inspired by the general model proposed in the personalized SDP study [36]. We include this model into our empirical analyses to compare our findings with the prior study.

We apply 10-fold cross-validation to evaluate PM, SM, and GM models by following prior studies on change-level defect prediction [16,20,27,130]–[132], and the prior personalized SDP study [36]. Recently, studies discuss the effect of dividing the commit data into 10 folds, without considering the time dependencies between the commits, on the performance of SDP models [28,29]. Empirical studies applying different strategies for change-level SDP (e.g., [20,130]), on the other hand, conclude that the findings of the selected strategies are consistent. The performance differences between the strategies are also relatively small. In particular, studies suggest that the conclusions about model performance can change when a different time period is utilized for training [133], or when a time-sensitive validation strategy is chosen [29]. We are aware of this issue, and hence, we further discuss the effects of applying cross-validation on the findings in Section 3.5. However, for the sake of comparability with the prior personalized models (e.g., [36]), we report, in the Results Section, the statistics based on 10-fold cross-validation. Furthermore, our objective in this study is not to generalize our conclusions regarding the performance of PMs or to report the best performance of PMs, but to compare those with traditional models. Hence, we use the same experimental setup for GM, SM and PM models throughout this study. All the experiments are repeated 10 times by shuffling the data before training-test split in order to avoid data order bias [19].

Please note that the test folds are kept the same across all models' performance evaluation in order to conduct a fair comparison among GM, SM and PM models. During each fold, PM, SM, and GM models share the same test set which corresponds to the commits of a developer associated with the selected fold and the project as illustrated in Figure 3.1. On the other hand, the training set varies among the models, i.e., it is filtered based on the criteria of the model. Figure 3.1 illustrates the methodology for the three selected developers of Rhino, but the same procedure is applied for the selected developers in every project in this study.

3.3.2.3 Under-sampling on training data

In order to handle the popular problem of imbalanced class distribution in SDP datasets (e.g., [16,80,100]), we apply under-sampling technique on the majority class by randomly selecting the majority class instances until the size of the minority (bug-inducing) and majority class instances is the same. Note that we do not apply sampling on test data. We further discuss its effects by comparing against a no-sampling strategy and another sampling technique in Section 3.5.

3.3.2.4 Evaluation of RQ1

Performance of the prediction models are evaluated in terms of probability of detection (pd, also called recall), probability of false alarm (pf), precision, F1-measure, area under the ROC curve (AUC), Matthews Correlation Coefficient (MCC) and Brier score. Pd, pf, precision, F1-measure and MCC are calculated from a typical confusion matrix (see Table 3.3). Explanations of these evaluation metrics and their formulas are given in Table 3.4. Pd, pf, F1-measure, AUC are well-known measures and reported in defect prediction studies, whereas MCC and Brier score are recently used in empirical software engineering studies [56,134]. We report and assess the models against all these measures as we should look for a trade-off between these to conclude whether a classifier is accurate and useful in predicting defects.

Table 3.3 : The confusion matrix for the defect prediction problem.

	Actually defected	Actually clean
Predicted as defected	TP (True positive)	FP (False positive)
Predicted as clean	FN (False negative)	TN (True negative)

To answer our RQ1, we compare the performance of PM with those of the traditional models (SM and GM). Pairwise comparisons between the three models are made using the Nemenyi significance test [135], and the model whose performance evaluation metrics are significantly different from the others is identified (according to $p < 0.05$). Effect sizes of the comparisons are measured via Cohen's d using corrected Hedges' g [136]. Measuring effect size is a simple way to quantify the difference between the performance values of the pairs of SDP models under comparison. The $|d| < 0.2$ is

Table 3.4 : The equations and explanations of performance metrics used in this study.

Name(s)	Description	Formula
Probability of detection (pd) Recall	Ratio of correctly predicted defected modules to actually defected modules	$\frac{TP}{TP+FN}$
Probability of false alarm (pf)	Ratio of incorrectly non-defected modules to actually non-defected modules	$\frac{FP}{FP+TN}$
Precision	Ratio of the actually defected modules to predicted as defected	$\frac{TP}{TP+FP}$
F1-measure	Harmonic mean of the precision and recall	$\frac{2*precision*recall}{precision+recall}$
Matthews correlation coefficient (MCC)	How well a defect predictor makes a binary classification as bug-inducing commit or not [137] (the value of 1 represents perfect classification while -1 represents a completely wrong classification)	$\frac{TP*TN-FP*FN}{\sqrt{(TP+FP)*(TP+FN)*(TN+FP)*(TN+FN)}}$
Brier score	The accuracy of predictions based on the prediction probabilities [138,139] (N: number of commits in the validation set, p_i: the probability of the prediction made by the SDP model, a_i: actual outcome of the commit (1 if it is a bug-inducing commit, 0 otherwise))	$\frac{1}{N}\sum_{i=1}^{N}(p_i-a_i)^2$
Area under curve (AUC)	How much the predictor is capable of distinguishing between classifying a commit as bug-inducing and non-bug-inducing	-

interpreted as negligible, $|d| < 0.5$ interpreted as small, $|d| < 0.8$ interpreted as medium, otherwise corresponds to large effect size. Comparisons with negligible and small effect sizes are not considered when we derive our conclusions, since larger data might be needed to claim a strong difference between the models' performance.

3.3.2.5 Evaluation of RQ2

We think that the development characteristics such as the ratio of bug-inducing commits to the total number of commits of a developer, the total number of commits of a developer, the size, history, and diffusion of the developers' changes, and the developers' experience might have an impact on the performance of PM. To understand the effect of these data characteristics on PM performance, we split 222 developers into three groups based on their success on PM:

1. *PM > SM/GM:* The developers whose PM model's performance is significantly better than that of SM or GM or both.

2. *PM < SM/GM:* The developers whose PM model's performance is significantly worse than that of both SM and GM.

3. *PM = SM/GM:* The developers whose PM model's performance is statistically not different from that of SM and GM.

We form these groups according to the Nemenyi pairwise tests conducted on all the performance evaluation metrics (pd, pf, precision, F1, AUC, MCC and BScore) in RQ1, and according to the test results with medium to large effect sizes. Then we compare the development characteristics across the three groups. For each group, the characteristic to be analyzed, e.g., number of commits of a developer, is aggregated over all developers in that group. Then, the aggregated values of the three groups are compared with each other by applying the Mann-Whitney U Test [140]. Later, the effect sizes of comparisons are measured via Cohen's d using corrected Hedges' g [136].

3.3.2.6 Evaluation of RQ3

Our motivation for RQ3 is to assess if a) PM models utilize a common metric set to predict bug-inducing changes of the corresponding developers, or b) each PM model has its own, unique metric set that provides the highest amount of information to predict bug-inducing changes for the corresponding developer. We applied Information Gain (InfoGain) feature ranking technique [141] on the personal commit data of each of the 222 developers in order to analyze the effect of each process metric (Table 3.1) on the prediction of bug-inducing changes for that developer. The metrics are ranked according to the information provided for bug prediction, and later, the rank values of each metric are compared to each other by using Scott-Knott ESD test to analyze if rank values of each metric are statistically different or the same among the developers.

3.4 Results

Empirical results conducted on six projects to answer our RQs are reported and discussed in this section.

3.4.1 RQ1: How does a personalized SDP approach perform compared to traditional SDP approaches? (PM vs. SM and PM vs. GM)

In this section, we discuss the findings of PM versus traditional models, namely SM and GM, by considering the algorithms' performance (NB and RF) individually. Later, we discuss the performance values of PM on an individual basis. As explained in our methodology, PM is trained using historical code changes of the selected developer only, whereas GM and SM are trained using all or a subset of the change history of the project respectively. Overall, we observe that the personalized SDP models are better at predicting bug-inducing changes compared to the traditional models. A more detailed discussion is provided below

There are two comparisons, namely PM vs. SM and PM vs. GM, based on the performance achieved with two machine learning algorithms applied (NB, RF), and based on seven performance evaluation metrics (pd, pf, precision, F1-measure, AUC, MCC and Brier Score). The winner model of each comparison according to the

Nemenyi test is given in the corresponding cells in Table 3.5. The "-" sign in the table means that the two models performed statistically the same.

Table 3.5 : Win/Loss results of the comparisons between PM and SM, and PM and GM. Bold cells indicate medium to large effect sizes.

	NB		RF	
	PM vs. SM	PM vs. GM	PM vs. SM	PM vs. GM
Pd	**PM**	**PM**	SM	GM
Pf	**SM**	**GM**	SM	GM
Prec.	SM	GM	SM	GM
F1	**PM**	**PM**	SM	GM
AUC	PM	PM	SM	GM
MCC	PM	PM	SM	-
Brier	PM	PM	SM	GM

The performance of PM, SM and GM aggregated over six projects are also provided as boxplots in Figure 3.2. The results of the models built with the NB algorithm are shown on top of the figure, while the results of the models built with the RF algorithm are given at the bottom of the figure. From left to right, the boxplots correspond to pd, pf, precision, F1-measure, AUC, MCC and Brier Score values.

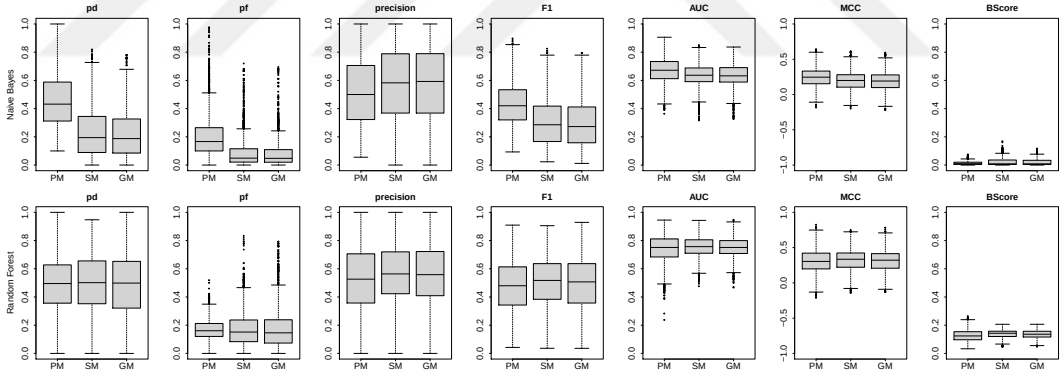

Figure 3.2 : Performance of PM, SM and GM.

According to the reported performance of the SDP models built with the NB algorithm in Figure 3.2 and the performance comparisons in Table 3.5, PM outperforms both of the traditional models in terms of pd, F1, AUC, MCC and Brier Score. Although PM increases the pd value, it also increases the pf value for 77% of the developers. The increase in pf also triggers a decrease in precision and hence PM cannot win over the traditional models in terms of pf and precision. The effect sizes of the comparisons between PM and the traditional models are large, around 1.3, 0.9, 1.0 in terms of pd,

pf and F1 respectively. For the rest of the evaluation metrics, the effect sizes of the comparisons are small or negligible.

According to the performance with the RF algorithm in Figure 3.2 and statistical test comparisons in Table 3.5, PM under-performs than SM and GM in terms of all model assessment metrics. The measured effect sizes of the comparisons between PM and traditional models, on the other hand, are negligible to small (0.1 to 0.3).

Personalized models in detail: The performance of PM models for 222 selected developers vary between 10 and 100% in terms of pd according to the results reported in Figure 3.2. This figure also shows us that while prediction performance of PMs for some developers are very low, others are high in terms of pd. Here, we investigate the PM performance of developers in detail. PM significantly differs from the traditional models in terms of pd, pf, and F1 when NB is utilized. These differences have large effect sizes, and hence here, we report the performance of PM versus SM/GM using NB algorithm and in terms of pd, pf and F1.

Listing the performance results of all of the 222 developers in this paper would take too much space. Thus, we chose nine developers with the best, worst and medium performing PM models, and report their PM, SM and GM model performance in Figure 3.3. The full list is available in [142].

Each developer's ID, the project name that the developer contributed are given on the left of the figure[3]. The total number of commits done by the developers and their bug-inducing commit ratio are also given respectively under their aliases. Each PM boxplot in Figure 3.3 has a specific colour that represents if the PM wins over the traditional models (SM and GM) according to Nemenyi pairwise tests. If PM outperforms at least one of the traditional models, the corresponding boxplot is coloured in blue, while it is red if SM and/or GM outperforms PM. If the personalized and traditional models perform the same, the boxplot is coloured in black.

Figure 3.3 shows that the individual PM performance could reach very high median values in terms of pd, e.g., 85% for Dev-4 and 92% for Dev-51 from Rails. On the other hand, PM models could not detect more than 11% of the defects for some developers,

[3]Developer names are replaced with aliases within each project due to privacy

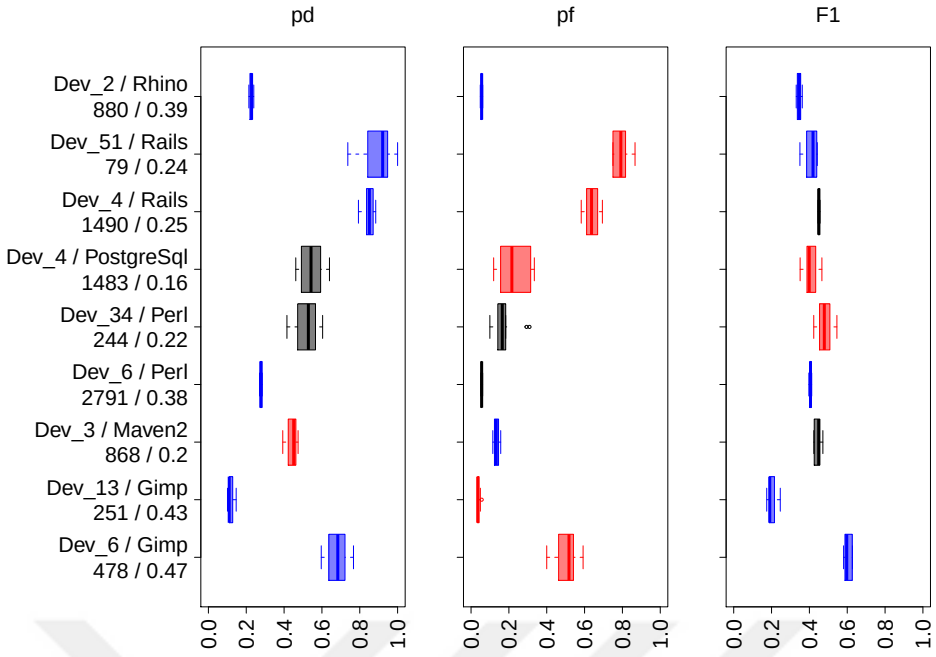

Figure 3.3 : PM performance of a sample of developers (blue color indicates superiority of PM over traditionals, red color indicates inferiority of PM, whereas black color indicates the equality of models).

e.g., Dev-13 from Gimp. PM, SM and GM perform the same for the other developers, e.g., Dev-34 from Perl and Dev-4 from PostgreSQL in terms of pd.

Besides, among all the 222 developers, there is not any developer whose PM results are better than the SM and/or GM in terms of all performance measures. While a developer's PM performance is better than at least one of the traditional models in terms of pd, it may not be better in terms of pf. For example, in Figure 3.3, PM for Dev-6 from Gimp significantly outperforms the traditional models in pd, but the traditional models outperform the personalized approach in terms of pf. This trade-off can be observed in all the developers' PM versus SM/GM comparisons. Thus, we believe the decision of the type of SDP model to be used for a developer seems to be highly dependent on the prioritized performance metric.

Building a PM is more convenient for majority of the developers contributed to six open-source projects: PM outperforms the traditional models in terms of predicting defects (pd) and F1-measure, while PM produces higher false alarm rates (pf) than traditional models.

3.4.2 RQ2: To what extent do development characteristics have an effect on the superiority of PM?

To understand what factors lead to the success of PMs in predicting bug-inducing changes, we would like to analyze development characteristics in detail. As explained in our methodology, we first identify development characteristics in terms of the size and bug-proneness of the development activity, and process metrics used as input features of our model. Later, we formed the groups of developers considering the performance of PM versus SM and GM. The number of developers that belongs to each group is reported in Table 3.6. We know from our findings in RQ1 that PM outperforms traditional models in terms of pd and F1 when NB is applied, whereas PM is worse in terms of pf. Therefore, the groups are formed according to pd, pf and F1 separately. The first group illustrates that there are 184 developers and 169 developers whose PMs are the best when pd and F1 values are compared, respectively. For 172 developers, PMs perform the worst in terms of pf. Once the groups are formed, we analyze the significant differences in terms of development characteristics among these groups.

Table 3.6 : Number of developers belong to each group.

	PM >SM/GM	PM <SM/GM	PM = SM/GM
pd	184	17	21
pf	28	172	22
F1	169	17	36

First of all, we compare the total number of commits of the developers across the three groups. Figure 3.4 shows the boxplots of the total number of commits made by the developers associated to the three groups. Pairwise comparisons among the three groups are conducted with Nemenyi tests in terms of pd, pf and F1. The tests show that the total number of commits made by the developers in the second group, where SMs or GMs win over PMs, is significantly different and larger than those of developers in the other groups. The effect size of this finding is medium to large. We observe many developers in the second group, i.e., four developers from PostgreSQL, three developers from Rails, and one developer from Gimp, who are among the top contributors to their project. The number of changes made by those developers are

in the range of 53 - 12755 (average is 2060). In such a case, it seems the traditional models perform better than PMs.

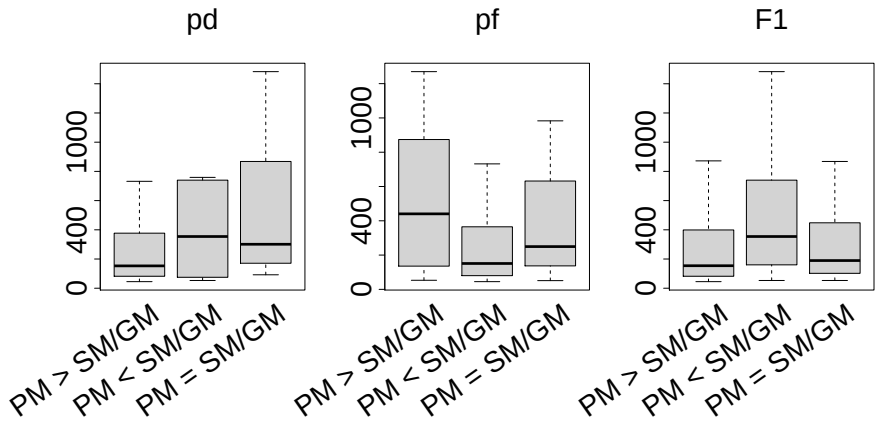

Figure 3.4 : Number of commits of developers belonging to each group.

Second, we compare the ratio of bug-inducing commits to the total number of commits of developers that belong to each group. Figure 3.5 shows the boxplots of the ratios of bug-inducing commits over total commits of developers associated to the three groups. According to the boxplots, the third group of developers, where PM, SM and GM performs statistically the same, has a lower ratio of bug-inducing commits when compared to the other groups of developers with a medium to large effect sizes. The difference among the first and the second group of developers has small and negligible effect sizes. Therefore, we conclude that bug-inducing commit ratio of developers does not seem to have an effect on which model (PM or traditional) performs better.

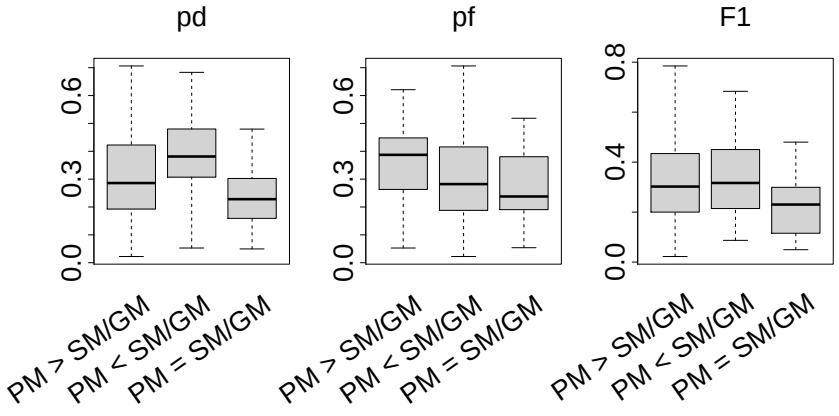

Figure 3.5 : Bug-inducing commit ratios of developers belong to each group.

Third, we investigate if there are differences or similarities among the contributions made by developer groups in terms of 13 process metrics listed in Table 3.1. We

aggregated the changes of developers within each group, and compared the values of each metric calculated from these changes among the groups. Results show that there are differences in five out of 13 metric values calculated from the changes among the three groups: NDEV, NPC, EXP, REXP, and SEXP. In Figure 3.6, we only report the values of these five metrics calculated for the three groups.

According to Figure 3.6, traditional SDP models perform better than PMs for the developers who have higher EXP, REXP, and SEXP values. This finding is consistent with our analysis on the first development characteristics, i.e., number of commits made by a developer. Combining both, we can argue that the traditional models perform better than PMs for those developers who have more experience on the project (contributed as the bulk of commits). On the other hand, PMs is better than the traditional models when the associated developers contribute to the modules which are modified by many developers (higher NDEV). Furthermore, when developers commit to modules which are modified many times (higher NPC) the difference between the PM and traditional models has a larger effect size. However, the winning model might change with respect to the performance measure.

We would like to highlight the fact that the findings regarding the developer group-based analysis also have similar patterns when the analysis setup is extended by including the performance results measured by other model assessment metrics and when the RF algorithm is used.

> The development characteristics significantly reflect under which settings PM performs better than SM and/or GM. PM is a more successful approach for predicting defects of the developers that contribute to the modules that have been changed by more developers. When a developer is among the most experienced developers in a project, PM underperforms compared to SM and/or GM.

3.4.3 RQ3: How does the importance of metrics used for defect prediction differ among personalized models?

The heat map in Figure 3.7 reports InfoGain results for all 222 selected developers' changes across all six projects. The columns represent the process metrics, whereas the rows represent the ranks one to thirteen. Values in cells represent *"how many*

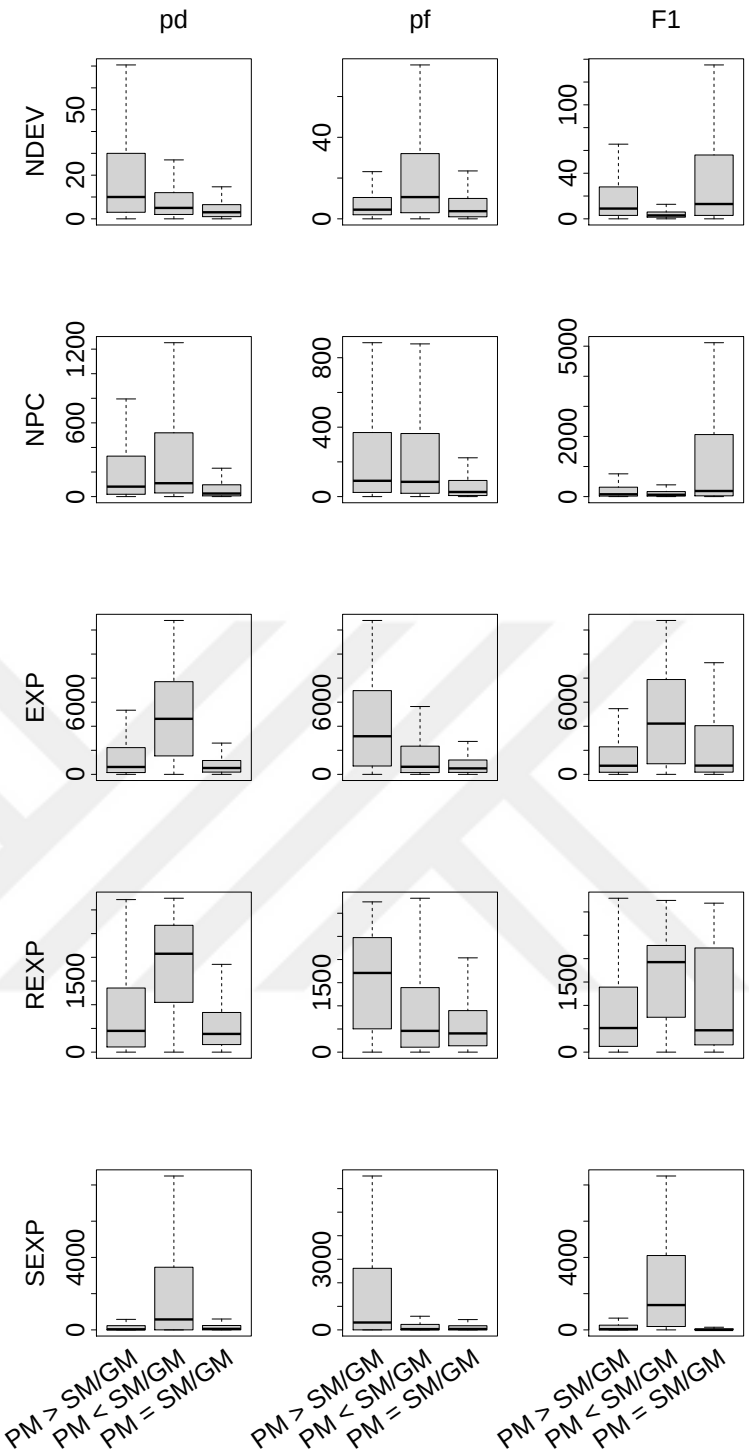

Figure 3.6 : Metric values of developers belong to each group.

times a metric is ranked as the first (or second to thirteenth) during an InfoGain evaluation?". The bigger values are represented with darker gray, whereas the smaller values are represented with a lighter gray. Figure 3.8 also reports the comparison of metric ranks based on Scott-Knott tests: The metrics that belong to the same group

are represented with the same color. The lower the rank of a metric in this figure, the higher its contribution to the prediction model.

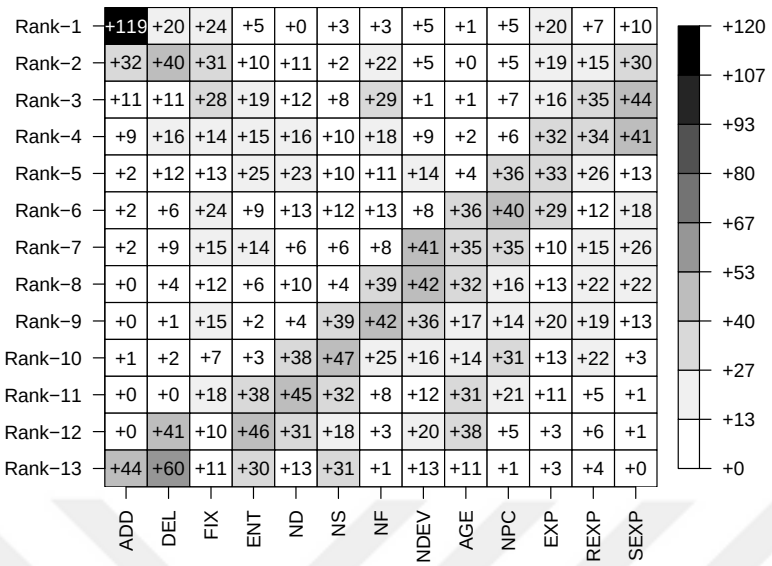

Figure 3.7 : Ranks of metrics based on InfoGain over each PM.

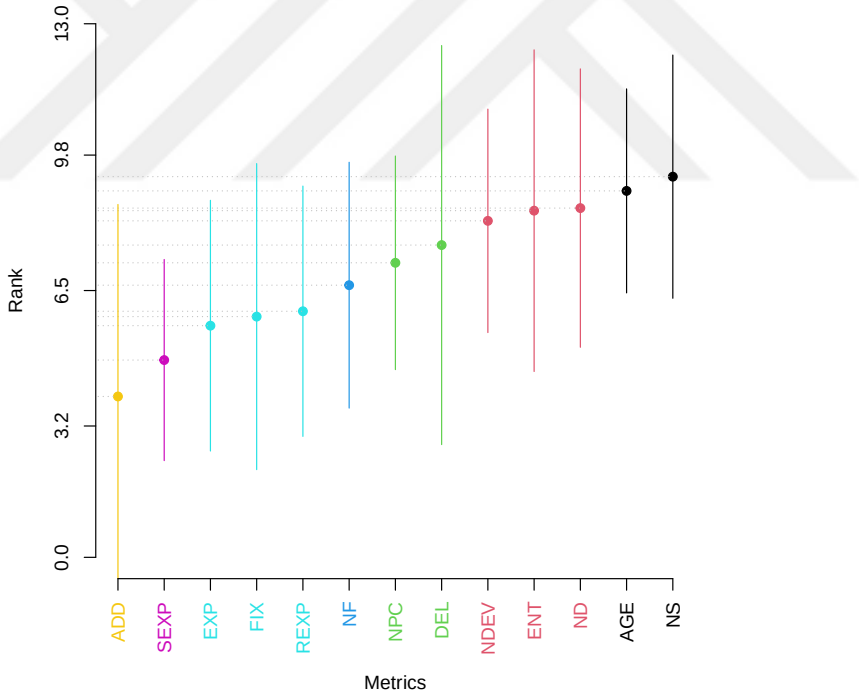

Figure 3.8 : Statistical comparison of the metric ranks.

Apart from the size metrics, Figure 3.7 shows that each PM gains different amount of information from different metrics. When we look at the Scott-Knott ESD clustering in Figure 3.8, we observe that the highest ranked metric is the added lines of code

in a commit (ADD). ADD is ranked as the first for 119 out of 222 developers. The subsystem experience (SEXP) is the second best metric and followed by the experience (EXP), the purpose (FIX) and the recent experience (REXP) metrics. The diffusion metrics, NF in particular, also appear in higher ranks than the history metrics (NPC, NDEV, and AGE).

Please note that we perform similar analyses on GM and SM, and observe that the *experience* dimension is more important when predicting defects with PM compared to SM and GM. In contrast, the diffusion dimension is more important in predicting defects with SM and GM. Particularly, the mean ranks of SEXP, EXP and REXP metrics are between five and six (Figure 3.8), while those metrics are ranked around six to 11 for SM and GM. In terms of the diffusion dimension, ENT, NS, NF, and ND metrics are one to two ranks higher in SM and GM compared to PM. Another important observation is that FIX becomes the last dimension in SM and GM models, whereas it is ranked as the third important dimension for PM models.

> PMs differ from each other in terms of the amount of information gained from the experience, diffusion and history metric dimensions respectively. According to the majority of PMs, the process metric representing the number of added lines is the most contributing one to the performance.

3.5 Discussion

Empirical results demonstrate that the selection of which SDP model (i.e. PM or GM) to use in real life may depend on several factors, such as machine learning algorithms, performance metrics and validation strategies. In this section, we discuss how all of these factors might affect the performance of the proposed personalized SDP models, and compare our findings with the prior personalized models in the literature. We also discuss the applicability of the PMs in industrial settings.

3.5.1 NB versus RF

Figure 3.2 given in Section 3.4.1 indicate that the performance values of an SDP model vary depending on the machine learning algorithms utilized during training. For example, the range of median of pd values across all three models is 0.19 - 0.43 when

models are built with NB, whereas the median pd values obtained with RF algorithm are 0.5 for all three models.

Pairwise statistical comparison between RF and NB for each of the three models points out that RF is better at predicting defects in terms of F1-measure, AUC, MCC, and Brier Score than NB. In terms of pd, PM performs statistically the same when it is built with NB or RF, whereas SM and GM perform better when they are built with RF. Although statistical tests on the models' performance values suggest that RF must be chosen over NB for answering our research questions, the effects of the models' performance differences are not large enough in the case of RF.

Both machine learning algorithms are widely used by the SDP researchers. A recent benchmark on the performance of defect predictors reports that RF may perform better according to one measure, namely H-measure [143], but the best performing classifier significantly depends on the project. The authors suggest that instead of utilizing a complex learning algorithm like RF, using a simpler one like NB would be more convenient. Based on the literature and our analyses, we also suggest using a simpler algorithm like NB since it reports significant differences with larger effect sizes in the context of PM versus traditional models.

3.5.2 Recall (pd) versus false alarm rate (pf)

The personalized approach often increases pd rates while it also increases pf rates, according to the performance values given in Figure 3.2. It is different than the common pattern observed in data-oriented SDP studies, e.g. using cross-company or cross-project data to predict defects [100]. Prior studies show that using other projects' data (global context) increases both recall and pf compared to using a project's data only (local context). But here in our context, using a developer-specific data increases both recall and pf compared to using all developers' commit data. On the other hand, when we look at the false alarm rates of PM model, it has a median around 20%. Having a false alarm rate of 20% is acceptable among many state-of-the-art defect predictors, considering that it achieves a better recall rates. Thus personalized approaches in SDP might be preferable over the traditional approaches as the former gives a better prediction performance.

False alarm rate (pf) of an SDP model is an important measure that should be evaluated according to the context. We believe the false alarm rates should be considered in terms of two perspectives, namely project and people perspective. Earlier studies discuss the project perspective [13,105] and state that in a safety-critical or a mission-critical software project, developers would prefer to have a model with high pd rates with the cost of high false alarms. However, a more cost-effective SDP solution would be to have fewer false alarms with the cost of low pd rates. Similarly, the developer's personal choice and/or development methodologies used by the team may affect the choice of having high pf or low pd rates. A developer may dislike the situation of frequent false alarm triggers, or even true positives if she chooses to review her code regularly, and detects the defects by herself instead of following the output of a commit-level defect predictor. In that case, only the bug-inducing commits that have a high probability should be given as recommendations to the developers using traditional models. This would in turn reduce the pd rates but eliminates potential false alarms significantly. On the other hand, a person who wants to explore all potential issues may prefer to use PM with the cost of false alarms.

Please note that, we chose seven performance measures to report the prediction performance of the three types of models in this study. We picked the most commonly used ones, namely pd, pf, precision, F1 and AUC, to make a fair comparison of our models' performance with the related studies. We also report other two measures, namely MCC and Brier Score, that are proposed to avoid biased assessment of F1-measure [144]. Unfortunately, the statistical pairwise comparisons among the models with respect to the measures, precision, AUC, MCC and Brier Score report small to negligible effect sizes. Thus, the conclusions derived from those performance measures in particular could be biased due to sample size or other data characteristics. It is our future goal to investigate ways to increase the effect sizes of the significance test outcomes for MCC and Brier Score.

3.5.3 Data sampling

During our empirical analysis, we applied random under-sampling on the training data to balance the number of data instances that belong to different classes. We observe

that under-sampling significantly improves the prediction performance of PM, SM and GM by 5% in terms of pd compared to a no-sampling strategy when models are built with NB. In addition to under-sampling, we also applied another well-known and successful data balancing technique from the literature, i.e., Synthetic Minority Oversampling Technique (SMOTE) [145]. Both SMOTE and under-sampling produce very similarly performing PMs, i.e., a median of 44% in terms of pd. Although SM and GM using SMOTE achieves 13% higher pd values compared to the no-sampling strategy, our test results on the superiority of PM over SM and GM do not change. In our online appendix [142], we share the prediction performance obtained by applying under-sampling, SMOTE, and no-sampling.

SMOTE does not lead to a better prediction performance for PM when compared to the under-sampling technique. In fact, applying SMOTE takes longer time than applying under-sampling, as the former technique creates synthetic data instances of the minority class using the k-nearest neighbor technique on data instances [145]. Under-sampling, on the other hand, simply selects random data instances from the majority class until the sample sizes of both classes are equal. Due to its simplicity and performance, random under-sampling gives us more advantage over SMOTE. We choose the under-sampling as our data sampling technique and report our performance results obtained with under-sampling in the paper.

3.5.4 Model validation strategy

We conduct our empirical analysis on personalized SDP models using 10-fold cross-validation strategy. We discuss our rationale behind this in Section 3.3.2. Cross-validation technique has been widely used in SDP literature to evaluate the performance of the change-level defect predictors [16,20,27,130]–[132], as well as the personalized defect predictors [36,53]. A study by Falessi et al. [146] also reports that the majority of the SDP studies (61%) uses k-fold-cross-validation, while a very few of them (9%) are validated by considering release-based data splitting. Recent studies argue that a time-sensitive approach that preserves the temporal order of commit activities, i.e., learning from past commits to make defect predictions on future commits, would resemble real-life scenarios. Thus, several

studies adopt the time-sensitive approach to their training and validation steps of the SDP models [20,28,29,130]. Two studies compare change-level SDP models by using both time-aware validation and cross-validation strategies [20,130]. Their results demonstrate that both strategies yield similar performance, and their conclusions are consistent among different settings. The ongoing discussion on the validation methodologies of defect predictors remarks that the conclusions derived from the experimentation should be limited within the experimentation context [133,146].

We believe both cross-validation and time-sensitive validation strategies are required to draw more generalized conclusions on the personalized SDP. The former is important to be consistent with the earlier studies in the field, whereas the latter is important in order to understand the practical applicability of personalized defect predictors. Due to the ongoing discussion on the validation strategy in the SDP field, we would also like to investigate the performance of the personalized models against traditional models (RQ1) in a time-sensitive validation strategy. In this subsection, we report the experimental setup for this analysis and its results.

A typical time-sensitive model construction makes a prediction on a commit at time t, by learning from the commits before time t. Considering that, new train-test data pairs are generated with a sliding window technique. For each developer (d), we stratify all his/her commits into time splits ($d_0, d_1, ...d_n$). Instead of having a fixed length time splits, i.e., six-month, we follow a strategy that produces consistent number of commits in each time split. Stratifying the data with a fixed length time window produces data subsets having no bug-inducing instances. Therefore, we ensure that our splits include at least 10 bug-inducing commits and 10 clean commits. This way, we maximize the number of selected developers and their commits. While one time split (d_n) of a developer becomes a test set, all the commits of that developer prior to that split (d_0 to d_{n-1}) constitute the train set of PM. All commits of *all* developers between the splits d_0 and d_{n-1} constitute of the training set of GM, whereas all commits of the *selected* developers between the splits d_0 and d_{n-1} constitute of the training set of SM. Similar to our setup in Figure 3.1, we keep the test set the same among three SDP models (PM, SM, and GM). The developer selection criteria and under-sampling on training are also applied similarly. However, due to the time-sensitive stratification strategy, we

can build PMs for 179 out of 222 developers. We report the performance of PM, SM, and GM trained with NB using a time-sensitive validation strategy in Figure 3.9.

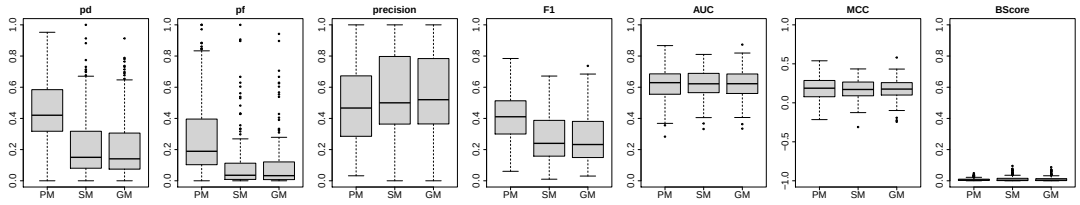

Figure 3.9 : Performance of PM, SM and GM using a time-sensitive validation strategy.

Findings on the personalized SDP approach are similar among both model validation approaches. We observe that median performance values of models are consistent with those models trained with NB in Figure 3.2. The superiority of PM over traditional models is also observed when predictors are evaluated in a time-sensitive validation setup. According to pd and F1, PM significantly outperforms the traditional models, but PM also produces higher false alarm rates, and hence both traditional models significantly outperform PM in terms of pf and precision. The statistical comparisons between PM and traditional models has large effect sizes in terms of pd (1.2), pf (0.8) and F1 (0.9), and small effect sizes for precision (0.2).

It might be possible to reach different conclusions on PM by changing the time-sensitive model validation strategy, i.e., stratifying data with fixed length time intervals and/or changing the training set size. Therefore, our results are open to discussion. Besides the model validation strategy, development characteristics of projects would also affect the conclusions on PM. For instance, in our experimentation, the contributors of a project and their contribution amount to the project changes over time. Fluctuation of the developers involved in the training of general models slightly reduces the prediction performance of general models. Although our results support the applicability of personalized defect prediction over the traditional approach in real life, further analysis and discussion are necessary. We consider applying other time-sensitive data stratification strategies as future research directions for assessing the success of PM in real-life settings.

3.5.5 Further insights on PM combined with the prior studies

As we report in Section 3.2.2 there are two studies on personalized SDP models [36,53]. Xia et al. [53] complements the work by Jiang et al. [36] by following the same methodology except the algorithm. The improved PM model in [53] achieves an average of 64% recall and 63% F1-measure over 60 developers. The prior studies and ours follow a completely different experimental design: We utilize different open-source projects except PostgreSQL, different machine learning techniques and process metrics to build personalized our PM and traditional models. Therefore, we cannot report here a one-to-one comparison between the earlier PM models and our proposed PM models. Even though both studies and ours share one open-source project, namely PostgreSQL, the time period of the collected data, the selected developers as well as the sampled data to train the PM models might not be the same. We are, in fact, covering a larger period of commit activity in our work. Our PM models report a median of 40% recall and 40% F1-measure, when NB is utilized, and 50% F1-measure, when RF is utilized, over 222 developers. We have lower PM performance in our context, and we think this might be due to incorporating more commits and developers into our PMs. However, our conclusions have a large effect size (RQ.1), and they support the success of PMs for defect prediction over traditional models. Furthermore, we did not restrict the commit size during training set of PM models, which, we have seen, has a major effect on the performance of PM against the traditional models (RQ2). We further provide insights on development characteristics, and discuss under which circumstances PM could reach a better prediction than a traditional model.

Furthermore, prior studies propose collective personalized models to improve the prediction performance of PM and state the superiority the collective models over PM. There are multiple collective models used by the prior studies. One of the methods is leveraging a collective training set in which half of the commits belongs to an individual, while the other half of the commits are taken from other developers' [36]. Another method is making an ensemble of the predictions of personalized and traditional models [36]. Xia et al. [53] also use genetic algorithms to create training sets by collecting various amount of commits from various developers.

Although our focus in this study is to assess PM against traditional models (SM and GM), we also set up collective models to validate their success in our context. Similar to Jiang et al. [36], we built a weighted personalized model (WPM) that utilizes a collective training set that 50% of the training commits belong to a developer, whereas the other 50% belong to other developers. We also built an ensemble model called PM+ that combines predictions of PM, SM, GM and WPM using the majority voting technique. WPM and PM produce very similar performance values and their statistical difference has small or negligible effect sizes. Depending on the algorithm, the superiority of PM+ or PM changes. When models are built with NB, PM outperforms the PM+ in terms of pd, whereas PM+ outperforms PM in terms of pf and Brier Score. On the contrary, when models are built with RF, we observe that PM+ outperform PM in terms of pd and F1 with the cost of higher pf values. These conclusions have medium to large effect sizes only. We conclude that an ensemble of PM, SM, GM and WPM models may perform better than PM when they are built with NB, and better than all when they are built with RF. Although the prior studies also support the usage of collective models (especially the ensemble approaches) over the personalized or traditional models, our empirical analysis highlights the importance of the selected performance measures on the final conclusion. The collective models' performance plots are available in our online appendix [142].

3.5.6 Effort-aware performance assessment:

Both prior personalized defect prediction studies [36,53] and the state-of-the-art change-level defect prediction studies [16,130,132] utilize an effort-aware performance assessment for their prediction models. Therefore, we also assess our prediction models' performances using an effort-aware measurement.

Effort-aware performance criterion basically assesses how much bug-inducing changes could be detected by inspecting only 20% of the total lines of code (LOC) during the review of changes predicted as bug-inducing. We calculated a benefit-cost ratio for each commit in the test set using the formula $P(c)/Effort(c)$ [16]. $P(c)$ is a binary value that represents the prediction made for commit c by a defect predictor (i.e., PM): 1 means (c) is bug-inducing, 0 means c is clean. $Effort(c)$ is the total number of

changed lines (added and deleted) in the commit c. Then, we rank the commits in the test set based on their benefit-cost ratio in descending order. Later, we count the number of bug-inducing commits that could be detected when only 20% of the total effort is spent on code inspection (when the commits are inspected in descending order of their benefit-cost ratio).

Our effort-aware performance assessment shows that PM catches more bug-inducing commits than SM and GM when only 20% of the total code inspection effort is spent. While 30% of the total bug-inducing commits could be detected with PM, SM detects and GM detects 16% of the total bug-inducing commits over six projects. Our effort-aware performance results also support the prior studies' findings on PM against the traditional approaches. While prior studies report that the bug-inducing commit detection rates are increased by 12% ([36]) and by 21% ([53]) when PM is chosen over the traditional models, we observe an increase by 14%.

3.5.7 Applicability to industrial settings

Building SDP models for the industry has several challenges, some of which are data availability for training, imbalanced bug-inducing versus not buggy commits, and input features [77,147]. Deploying these models to industrial settings, on the other hand, has other challenges, such as training data size, training/update period, and changes in the development team. Personalized SDP models would be affected more than traditional models from the data availability during training and changes in the development team. For example, lack of commit history for a developer or lack of bug-inducing commits in a developer's commit history would prevent the developer from utilizing a personalized approach which was trained and customized for her. In such a scenario, only traditional models can provide recommendations to the developer until a sufficient amount of training data with two classes is collected. Another challenge of deploying personalized SDP models is related to the changes in the development team. During offline studies on historical data, we do not encounter cases when a new developer starts making commits, or when a senior developer stops contributing to the project. We aggregate all the commits, group those according to the developer who made these commits, and train personalized models for each developer. On the other hand, in a

real-life scenario, some developers leave the project, whereas some join the project in the middle of the development process. Therefore, there is a high possibility that some of the developers in the selected project would not have commit history for building a personalized model, while there would be former developers whose previous commit data would no longer be useful to anybody. Hence, a mechanism that utilizes the other developer's data to build a mixed model, or an alternative mechanism that switches between the general model and the personalized model for a specific developer depending on the data availability could be useful. Also, grouping developers based on their development similarities and building group-customized models would be another solution to a cold-start problem. In this study, we report our findings on open source projects by collecting their historical commit data. We are also working with our industrial partner, for which we had already built SDP models [148], to design personalized SDP models. We are currently working with them to address some of the issues mentioned above as well as time-sensitive model validation strategy mentioned in Section 3.5.4 in order to deploy such models into real-life industrial settings.

3.6 Threats to Validity

Construct validity: The descriptive characteristics of the commits should be quantified through a metric set to measure the defect proneness of the commits. The metric set utilized in this study is widely used in change-level SDP studies in the literature (e.g., [16,27,30,109]–[113,115]). The metrics quantify four different aspects of the changes, namely size, history and diffusion of the commits, and experience of the developers who made the commit. These aspects of the commits are measured by multiple metrics to avoid mono-metric bias in modelling, e.g., history dimension is measured in terms of three metrics, as it can be seen in Table 3.1: The number of developers who changed the modified files, the number of prior changes to modified files, and the average time interval between the last and the current change. We did not use complexity metrics or other object-oriented metrics used in the SDP literature because recent benchmarking studies [108,143] show that process metrics extracted from the code changes perform better than the other metrics on multiple projects.

Bug-fixing and bug-inducing commits were previously identified in [27] by using SZZ, and hence, we also used that dataset. The SZZ algorithm was used in the selected six open-source projects to define which parts of the code introduced the defects. The SZZ algorithm is frequently used in the literature to identify the bug-inducing commits by linking those with bug-fixing commits [116]. Bug-fixing commits are the locations where a bug is fixed, and fix locations are determined using the issue tracking system used by the development team. Later, the SZZ algorithm uses bug-fix-locations and traces back from bug-fix-locations to the previous commits to identify the commit where the bug was first injected into the software product. We are aware that there are still limitations of the SZZ algorithm, such as high false alarms, lack of identifying causes of new code additions, and variations in implementations by different researchers [149]. As prior research on SZZ [150,151] indicated that SZZ implementations still need improvement to reduce the noise that causes mislabelled changes. Fan et al. [151] empirically assessed the various SZZ implementations, and they reported that the original SZZ approach [116] does not yield a significant performance reduction in just-in-time defect prediction models due to the noise in the change labels (bug-inducing or not). Therefore, we rely on the results of the SZZ algorithm used to collect the projects' data.

We try to cover as many developers as possible in our empirical analysis. Accordingly, PMs built in this study are trained with various numbers of commits, i.e., between 45 and 12.755. Due to the under-sampling, training set sizes also vary from 18 to 8.000. Some PMs have very few data instances in their training sets, and it might be possible to observe poorly performing PMs for those developers. We checked the performance of those PMs built with very few data instances, i.e., between 18 and 100 commits, against the other PMs, which are trained with more than 100 commits. We obtained a median of 0.5 pd rate for PMs in the first group (trained with less than 100 commits), whereas PMs in the second group (trained with more than 100 commits) produce a median of 0.35 pd rate when NB is used. Moreover, 89% of PMs in the first group statistically outperform SM and GM, while 75% of the PMs in the second group outperform SM and GM. These findings also support our answer for RQ2 on the commit counts of developers (Figure 3.4): PM underperforms compared to SM and/or

GM for those developers who contributed to a project with higher number of commits. A more strict developer selection procedure might be a primary choice considering that a good amount of data is needed to build a successful predictor [128]. However, our empirical analysis shows that we can build successful PMs with less than 100 commits.

Internal validity: In this study, we assume that each developer who contributed to each project made his/her commits according to some development principles. One of these assumptions is that developers linked their commits with the right issues in the issue repository. Another assumption is that each developer only commits his/her development codes. In some cases, the development of different developers may be committed by the authorized personnel only. We did not collect the data set, however, we double-checked the mentioned issues and ensured all were correctly addressed in the prior studies using the same dataset.

We build a PM based on a developer's commits including bug-inducing ones and others, such as bug-fixing commits and other code changes. The author of a bug-fixing commit may not be the developer who has introduced the bug into the software system. Thus, a PM built for that developer does not include bug-fixing commit of another developer. Further, a bug-fixing commit may contain changes related to legacy code, i.e., the code is fixed due to a list of changes over the years, and the developer who fixed is not responsible in this case. SZZ is used to trace back to all prior commits which are made by the selected developers, and the data is added to their PMs respectively. Those situations may affect the performance of the personalized SDP models. However, there is not an accurate approach that can detect the exact reason and/or the source of the bugs in the software system. We believe that utilizing information from the historical aspect of the commits (i.e., number of prior changes to the modified files, number of developers had changed the modified files) helps to partially capture the indicators of above-mentioned situations [152].

Assessment of PM against traditional models is conducted on the commits of the selected developers as we described in Figure 3.1. To assess personalized models, we have to focus on a set of developers whose available commits are enough to build PMs for them. Therefore, those developers' commits with limited contributions are

not included into the analysis we conducted to answer our RQs. However, we also checked the prediction performance of both traditional models (SM and GM) on those developers' commits with the limited contribution, and obtained an average of 20% pd with NB. This ratio is very close to the value reported for traditional models in Figure 3.2, and confirms that the even a state-of-the-art general model with all commits would reach a similar performance for the selected open source projects.

External validity: We performed our analyses on a set of open-source projects. Applying the proposed approach to different projects in industrial settings may not yield the same results obtained in this study. We think the project characteristics is an important factor that could affect the applicability and accuracy of the personalized SDP approaches. For instance, when the contributions in a project are too scattered among too many developers, PMs would perform better than traditional models. This is because, developers having bulk commits do not dominate the project (e.g., Rails), and each developer's commits contain sufficient amount of data indicating defect proneness. On the other hand, when the project has few developers responsible for majority of the all commits (e.g., PostgreSQL), traditional models perform better. Furthermore, the time period selected to collect the commits, the distribution of commits over this time period and the number of active developers should be considered during project selection. A similar conclusion is also reported in the latest benchmark [143]. To further understand the personalization in the SDP field, we plan to assess the performance of the proposed personalized SDP approach on a commercial project in the future.

Conclusion validity: During our experimentation, we use well-known open-source projects, model building and assessment techniques, and statistical tests (Section 3.3). The findings of our work are valid under the projects selected for this study, the metrics used, the algorithms used, and the strategies followed during model construction. To increase the conclusion validity of our research, we applied statistical tests on the differences between models' performance, and only base our conclusions on the differences with medium to large effect sizes. This way, we believe that we avoid potential bias related to sample size. The selected validation strategy, 10-fold cross-validation, might have influenced the results as it does not consider

the temporality between changes. In Section 3.5, we elaborate on the validation strategy in detail, discuss prior works that compare different strategies, and we perform all empirical analyses regarding RQ1 with a time-sensitive data split strategy. Time-sensitive results also confirm that PM models are significantly better than traditional models for defect prediction, but we need more sample for large effect sizes. As the dataset is already publicly available, our results can also be reproduced and refuted in future studies.

3.7 Conclusion and Future Work

In this paper, we investigate the performance of personalized change-level software defect predictors by defining three research questions (RQs). Our personalized models (PM) achieve 24% higher probability of detection and 14% higher F1-measure rates than the two traditional models used in our study (SM and GM) (RQ1). Furthermore, we provide valuable insights for both researchers and practitioners regarding the set of development characteristics which highlight the superiority of PM over traditional models (RQ2). Furthermore, we investigate the common and the best indicators of bugs across different PMs (RQ3). We derive our conclusions based on a cross-validation setup built with NB algorithm, seven performance assessment metrics and statistically significant differences with medium/high effect sizes, but we also consider the experimental setup details that may affect our conclusions in our discussions. Over 222 developers from six open-source projects, we summarize our key take-away messages below:

- Even though the overall comparison between the performance of PM and traditional models leads us to prefer PMs over SMs or GMs, PM may not be suitable for every developer in a team. We observe that PMs built for most of the developers using NB algorithm detect more bugs, but general approaches may still be more suitable for the other developers.

- According to our empirical analysis, both traditional models (SM and GM) are more successful than PM for those developers who dominate the project's development activity with many commits. Those developers whose PM is similar to or more

successful than at least one traditional model have less experience than others in software projects.

- PMs would give better predictions than traditional models for the developers who work on the software modules priorly modified by many developers. The group of developers whose PM is significantly better than SM/GM has higher NDEV (the number of developers had changed the modified files) metric values than the group of developers whose PM is equal to or significantly worse than SM/GM.

- Building a PM for a developer would need careful consideration on the process metrics since our analysis shows that the best indicators of bug-inducing changes differ among 222 PMs. Nevertheless, the number of added lines of code seems to be the most successful indicators of bugs. The developer experience metrics and the purpose metric are also the next successful indicators of bug-prone changes for the majority PMs after the number of added lines of code.

- The selection of PM over other models is also dependent on which performance measure is selected/prioritized: a personalized model could be better in detecting defects (pd) whereas it is worse in terms of false alarms (pf). Hence, the model assessment should be made according to the selected/prioritized performance measures.

- PM looks promising on its adoption to real-life since PM also outperforms the traditional models in a time-sensitive model validation setting, which resembles the real-life by preserving the temporal order of data during training and prediction. As we discussed in Section 3.5.4, further model validation scenarios are needed on the time-sensitive evaluation of PM to make a generalized conclusion for real-life applicability of PMs.

In this research, we provide more insights than prior studies on the performance of personalized models. Still, we need to continue investigating the factors affecting the performance of PM as future work. Although we did not cover the project characteristics on the performance of PM, we think they are also important. For example, the time period of collected commits and the distribution of the commits

of developers through the project time period may be some of the factors affecting the performance of personalized models. We have recently observed that a combination of statistical and machine learning techniques to extract multi-dimensional information from commit history would perform better than extracting a single aspect (processing through code changes) [148]. We plan to understand the effect of such a combined technique on the performance of personalized models. Plus, industrial case studies will provide more insight on the applicability of personalized SDP models and the usability of the personalized recommendations. Therefore, in the next chapter, we investigate the personalized SDP approach in an industrial software project.

4. INVESTIGATING THE PERFORMANCE OF PERSONALIZED MODELS FOR AN INDUSTRIAL SETTING

4.1 Introduction

Personalized SDP models have been proposed and comprehensively investigated in open-source domain by prior studies (see Chapter 3). Our prior research reported in Chapter 3 show that personalized SDP approach improve prediction performance up to 24% in terms of recall in open-source projects. However, performances of personalized models have never been analyzed in an industrial context. Conducting research in industrial domains is very valuable since it provides assessing the generalizability of research findings and practical applicability of the prototype models into the cases that involves real customers [2,48]. Therefore, industrial software projects are significant data resources for SDP researchers to understand the real-world environment.

However, research in industrial cases have its own challenges as well as those encountered in empirical software engineering studies regardless of the context. On the one hand, accessing an industrial project repository to collect data requires more effort compared to accessing a publicly available repositories due to the confidentiality policies of companies [3]. Even if a collaboration has been established between the academic team and the company, the agreement is often limited to a single or a few projects. On the other hand, empirical research on software engineering data could be quite challenging due to noise [3]. Therefore, the scarce and noisy structure of the data force researchers to extract as much information as possible from available data sources through statistical and AI/ML techniques [148].

During this thesis, we had an industry and academia collaboration project with Ericsson, Turkey. One of the objectives of this research project is investigating personalized and general change-level defect prediction models for a chosen pilot project of Ericsson, Turkey. We collected data from the chosen project, train SDP prototypes for the company, and deploy a chosen prototype into the real development

environment of the project. Even though we share the deployment experience in the next chapter (Chapter 5), in this chapter, we focus on personalized model prototyping phase.

We collect 1773 commits of 36 developers of the pilot project. Personalized defect prediction models are built for six developers and their performances are assessed against general models. During model training, we utilize multiple data sources, such as semantic of commit messages, the state-of-the-art process metrics and the latent features of them.

We ask three research questions in this chapter as follows:

RQ1: How does a personalized SDP approach perform compared to traditional SDP approaches? (PM vs. SM and PM vs. GM)

RQ2: How can we improve the performance of defect predictors by utilizing the available commit data through a combination of statistical approaches?

RQ3: How does the importance of metrics used for defect prediction differ among PM, SM, and GM?

We borrow RQ1 from the research reported in Chapter 3. The aim in RQ1 is to investigate the performance of PM against traditional models in the chosen industrial project's context. We borrow RQ2 from our prior research [148]. In RQ2, we aim to assess the performance of PM, SM, and GM when they trained by utilizing a combination of statistical approaches, i.e., topic modeling and matrix factorization. RQ3 aims to observe how much the important metrics differ among different SDP models.

Our empirical analysis show that traditional approaches are superior to the personalized approach in a setting that utilizes only the state-of-the-art process metrics. However, combining latent features of the process metrics and semantic features extracted from commit messages, and applying log filtering leads PMs that achieves similar performance with GM. Further, we observe that the most important metric group for all SDP models is the process metrics, followed by latent features and semantic features, respectively. Even though PMs' performance could be improved

with extra data sources and statistical approaches, its applicability into practice depends on team related factors. In our case, for instance, cold start problem occurs since the current contributors of the project have not had rich development history yet and majority of data belongs to developers that left the organization.

4.2 Related Work

Some recent research use topic modeling technique to extract semantic information from software repositories, and use the extracted semantic features to train SDP models. For example, Nguyen et al. [153] and Chen et al. [154] analyze the correlation between the hidden topic models of source codes and the defect proneness of the source code. Nguyen et al.'s prediction model's predictive power improved by 12% when topic models of code are used instead of churn of code metrics. Barnett et al. [155] extract topic models from description of software changes, and utilizing them to train SDP models improve the explanatory power of the model up to 72%.

On the other hand, matrix factorization techniques also used to capture the underlying hidden information from data. Utilizing matrix factorization in ML/AI tasks such as pattern recognition [156], and recommendation systems [157] is more common than SDP field. Studies use matrix factorization is relatively less, specifically, it used to impute missing data [158] and to predict defect prone modules [159].

4.3 Collection of Dataset

We collect the historical development activity of a chosen pilot project of Ericsson, Turkey. The historical development activity includes commits from various developers belong to a five year time period. Our prior work reported in Chapter 3 guide us during metric collection and data labeling. We extract state-of-the process metrics (see Table 3.1) of the collected commits, and label commits as bug-inducing or clean using the SZZ algorithm [116].

The collected dataset has gone through some cleaning and manual matching before taking its final shape with the help of the industrial team. First, the commits that belong to test modules of the project are filtered. Also, a set of commits that does not involve real development are determined with the manual inspection of industrial

team and those are eliminated from the dataset. Second, we realize that there are some developers that has been contributed to the project with different usernames. This causes the contribution of the same person in the software repository to appear as if they belong to different people. Therefore, with the help of industrial team, we identified the usernames that belongs to the same person and match them under a single username. This step is crucial since we aim to build personalized models. Third, industrial team manually identified some bug-fixing commits that are not obvious to our eye. Correctly determining the bug-fixing commits is very crucial, since the SZZ algorithm requires the bug-fixing commits as input to detect the bug-inducing commits. Although the bug reporting and fixing activities often reported properly by the team, exceptions can occur. Bugs may not be reported in the issue tracking system or bug id may not be specified in the commit that fixes that bug.

The final dataset have a total number of 1773 commits, a total number of 392 bug-inducing commits, and a total number of 36 developers contributed to the project between the years of 2013 and 2018. The details of the dataset are reported in Table 4.1. The table also contains information on developer selection, which is one of the steps of the model building phase that is reported in detail in Section 4.4.4.

Table 4.1 : Details of the dataset.

Information	Value
Time period of commits in years	2013 - 2018
Total number of commits	1.773
Total number of bug-inducing commits	392
Total number of developers	36
Number of selected developers	6
Total number of commits of all selected developers	1.028
Total number of bug-inducing commits of all selected developers	335
Range of commits of selected developers'	68 - 343
Range of bug-inducing commit ratio of selected developers' (between 0 and 1)	0.28 - 0.4

The other details regarding data collection and project context, such as how SZZ is executed over software repositories, development practices of the team, and the

distribution of development activity over time, are reported in the next chapter that reports our industrial deployment experience (Chapter 5).

4.4 Methodology

Our objective in this study is to analyze personalized models against traditional in an industrial setting. Since our collected data is scarce, we utilized various approaches to extract more information from the available data. We assess whether including more information, i.e., latent and semantic features, into the model building process improves the prediction performance of predictors. Further, we aim to understand the importance of all feature groups used in the experiment to the prediction of bug-inducing changes.

This section first reports our metric collection approaches in Sections 4.4.1, 4.4.2, and 4.4.3, second depicts the model building process in Section 4.4.4, and third the performance assessment of the built models in Section 4.4.5.

4.4.1 Base metric collection

The process metrics used in Chapter 3 and this chapter are the state-of-the-art metrics that also used in other change-level SDP studies in the field [16,27]. In this chapter, we call them as our "base metrics' and take the models built using the base metrics s our baseline models. Then, we add the other metrics, i.e., latent and semantic features, to the base metrics to assess the contribution of new metrics to the baseline models.

Base metrics measures the development activity from size, history, diffusion, and experience aspects. The full list of metrics is given in Table 3.1 in Chapter 3. The size and diffusion aspects of a commit is directly calculated from the logged information of commit by the software repository. Whereas, the history and experience metrics for a commit are calculated tracing back through the all prior commits' logs in the repository.

4.4.2 Latent feature extraction through matrix factorization

Non-negative matrix factorization (NNMF) [160] is a popular technique used to capture hidden information of data and to represent information with fewer dimensions, i.e., low rank representations [161].

In this study, we use NNMF to extract latent information of development activity that may not be captured by the base process metric sets listed in Table 3.1. First, we define a matrix $V \varepsilon \mathbb{R}^{cx13}$ which includes the extracted base metric set of each commits in our training set. While c is the number of commits in the training set, 13 is the number of base metrics used in the experiment. Second, we aim to get a 5-rank approximation of the V by factorizing it into two matrices $W \varepsilon \mathbb{R}^{cx5}$ and $H \varepsilon \mathbb{R}^{5x13}$. W matrix is used as latent features to train models. Third, we obtain a 5-rank approximation of the test set in the form of the training set to use it in model validation phase. We multiply the test set $T \varepsilon \mathbb{R}^{nx13}$ (n is the number of commits in the test set) with the transpose of H to get the latent features of the test set $T \varepsilon \mathbb{R}^{nx5}$.

4.4.3 Semantic feature collection through topic modeling

Topic modeling aim to capturing underlying concepts in data. Latent Dirichlet Allocation (LDA) is a commonly used technique in topic modeling [162].

In our experiment, we use LDA to reveal the hidden concepts of software changes, i.e., we extract topic models of commit descriptions written by the developer of the commit. First, we apply standard natural language pre-processing steps to use topic modeling in a more effective way. Stop words (e.g., the, a, on) in commit messages are cleaned, and all letters in messages converted to lowercase. Second, LDA is used to capture topic models. LDA takes commit messages as inputs and creates a vocabulary from all unique words used in the all commit messages. LDA assumes that each commit message includes a mixture of hidden topics, while each topic includes a mixture of the words in the vocabulary. Then, LDA generates a probability distribution for each commit over a set of topics, and also a probability distribution for each topic over a set of words. We extract 10 topic models, i.e., 10 dimensional probability distribution

vectors, from each commit messages. Third, we use these 10 dimensional vectors during the training of SDP models as the semantic features.

4.4.4 Model building

In the industrial setting, we use the same model building methodology as we used in the open-source setting, which is reported in Chapter 3.

We start with picking developers that have sufficient number of commits to build a personalized model for them. The developers in the team have few commits, i.e., the half of the developers contributed to the project have a number of commits between one and 23. Therefore, we cannot build a PM for all developers in the team, wince we need good amount of data to train a prediction model [128]. In addition to have sufficient number of commits, we need to have sufficient bug-inducing commits to train a two-class SDP model. While some developers in the team contributed with large amounts of commits, i.e., around 80, their commits induced a few bugs into the software, i.e., one or two. For such developers, we cannot build a personalized model. Considering all these, we pick developers that have at least 45 commits and 10 bug-inducing commits for the experiment. At the end of developer selection process, we have pick a total number of six developers to build personalized models for them. The total number of commits of these six selected developers is 1028 while the total number of bug-inducing commits of the selected developers is 335. Moreover, the number of individual commits of these six developers range from 68 to 343, while their ratio of bug-inducing commits to their total individual commits range from 0.28 to 0.4. All these details regarding the selected developers are reported in Table 4.1.

After identifying a set of selected developers, we build three SDP models by applying 10-fold cross-validation: 1) personalized model (PM), 2) general model for the selected set of developers (SM), and 3) general model (GM). As specified in Figure 3.1, we split each selected developer's commits into 10 folds. We keep one fold of a developer as the test set, and use the other nine folds as the training set when building PM. We keep test folds the same when building PM, SM, and GM, to make a fair performance assessment across the three SDP models. To construct SM, we combine all selected developers' commits as the training set and exclude the test fold of the

developer to assess the performance of SM for that developer in particular. To construct GM, we apply a similar methodology. We combine all developers' commits, regardless of whether they selected to build PM or not, into the training set of GM. We exclude the developer's test fold from the training set of GM to use it to as the test set to assess the performance of GM for the developer. We repeat these process 10 times by shuffling the data to avoid learning bias caused by data ordering [19].

Unlike the experiment conducted with open-source projects in Chapter 3, we do not utilize the under-sampling technique in the industrial setting. The reason for that is the size of industrial dataset, i.e., the selected developers has 19 to 96 bug-inducing commits. Nevertheless, we try other data sampling techniques to balance the ratio of bug-inducing and clean (non bug-inducing) commits. We apply over-sampling and Synthetic Minority Oversampling Technique (SMOTE) [145] on the training sets before building SDP models. Our analysis show that applying over-sampling and SMOTE does not significantly improve the models' performance in the Ericsson's pilot project's context. Therefore, we build the models without applying data sampling.

One of our research objectives in this chapter is to analyze the effect of combining different statistical approaches on the prediction performance of PM, SM, and GM. Hence, we use three AI/ML and statistical approaches with different combinations during model training to observe the effect of each on the models' performance. 1) We utilize latent features of development activity measured over base metrics using NNMF, 2) we utilize semantic features measured over commits' messages using topic modeling technique, and 3) we apply log-filtering onto all features. Log-filtering basically takes the logarithm of numeric feature values to make them evenly spread across the distributions [163]. Log-filtering helps the prediction models to distinguish differences among features during training.

We construct eight different models for each SDP model (PM, SM, and GM) that utilize the statistical approaches with various combinations. Combinations of approaches are listed in Table 4.2. The rows represent eight models, while columns represent the approaches. If a model utilizes an approach, the corresponding cell includes the "+" symbol. Abbreviations used in the model names represent the

statistical approaches, i.e., "B" is used for base metrics, "LAT" is used for latent features, "SMN" is used for semantic features, and "LogF" is used for log-filtering.

Table 4.2 : SDP models built various metric and data processing combinations.

	Base metrics	Latent features	Semantic features	Log filtering
B	+			
B+LogF	+			+
B+LAT	+	+		
B+LAT+LogF	+	+		+
B+SMN	+		+	
B+SMN+LogF	+			+
B+LAT+SMN	+	+	+	
B+LAT+SMN+LogF	+	+	+	+

We build a baseline model "B" (see the first row in Table 4.2) that only utilizes the base metric set during model training. Then, we combine each statistical approach with the baseline model separately and together to observe the effect of approaches on the prediction performance. For example, "B+LogF" applies log-filtering on baseline model "B" (see the second row in Table 4.2). Another example, 'B+LAT+SMN+LogF" utilizes base metrics, latent features, and semantic features with log-filtering (see the last row in Table 4.2).

Similarly to the experiment conducted on open-source projects (Chapter 3), we also use Naive Bayes (NB) and Random Forest (RF) algorithms in this experiment.

4.4.5 Performance evaluation

Assessment of the models' performance is done using the same metrics as used in our prior research reported in Chapter 3: probability of detection (pd), the probability of false alarm (pf), precision, F-measure, area under receiver operating characteristics (ROC) curve (AUC), and Matthew Correlation Coefficient (MCC). Explanations and calculations of these metrics are reported in Table 3.4.

4.4.6 Methods to answer research questions

The experiment conducted to answer RQ1 follows the same structure we use in Section 3.3.2.4 of Chapter 3. In summary, we compare the PM against SM and GM and assess

the comparison results using Nemenyi significance test [135] and Cohen's d using corrected Hedges' g [136].

To answer RQ2, we build all eight models listed in Table 4.2 for PM, SM, and GM. Then, for each SDP model (PM, SM, and GM), we compare eight models' performance values with each other. We use Scott-Knott ESD test [82] to assess the statistical differences between the models.

To answer RQ3, we apply Information Gain (InfoGain) feature selection on the "B+LAT+SMN" model. InfoGain ranks the features in the given model according to their contributions to the prediction [141]. Based on InfoGain results, we rank the three feature groups (base, latent, and semantic) the highest to lowest according to the information gained by each feature group. We use the ranking results to analyze the importance of feature groups for PM, SM, and GM.

4.5 Results

4.5.1 RQ1: How does a personalized SDP approach perform compared to traditional SDP approaches? (PM vs. SM and PM vs. GM)

In this section, we report the comparison of PM with the two traditional models, SM and GM. The performance values of three SDP models are aggregated over six selected developers and reported as boxplots in Figure 4.1, while the statistical comparison of PM with the traditional models is reported in Table 4.3.

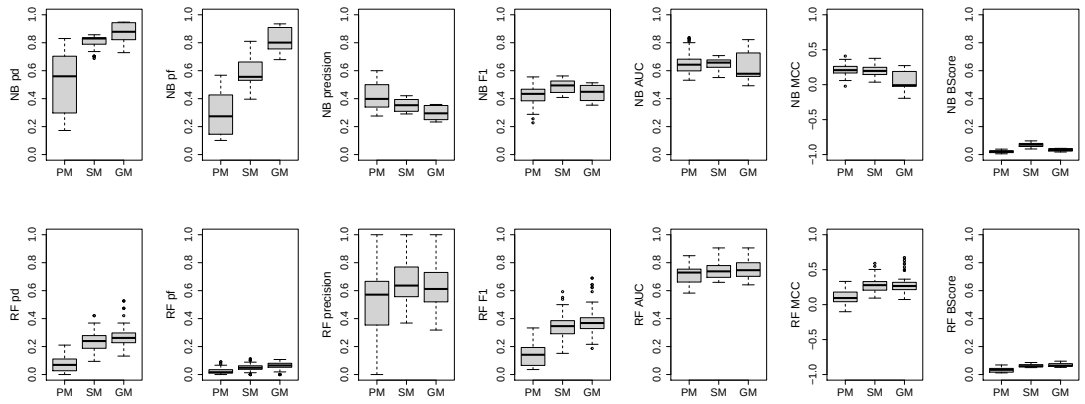

Figure 4.1 : Performance of PM, SM and GM.

Each box in Figure 4.1 includes three boxplots that show the performance values of PM, SM, and GM, respectively. The plots on the upper row show the performance values when models are built with NB, while the plots at the bottom row show the performance values when RF is used to train models. From left to right, plots report performance values in terms of pd, pf, precision, F1-measure, AUC, MCC and Brier Score.

Table 4.3 : Win/Loss results of the comparisons between PM and SM, and PM and GM. Bold cells indicate medium to large effect sizes.

	NB		RF	
	PM vs. SM	PM vs. GM	PM vs. SM	PM vs. GM
Pd	**SM**	**GM**	**SM**	**GM**
Pf	**PM**	**PM**	**PM**	**PM**
Prec.	**PM**	**PM**	**SM**	**GM**
F1	**SM**	-	**SM**	**GM**
AUC	-	-	**SM**	**GM**
MCC	PM	**PM**	**SM**	**GM**
Brier	**SM**	**GM**	**SM**	**GM**

Table 4.3 reports two comparison namely PM vs. SM and PM vs. GM. These two comparisons are repeated for the two machine learning algorithms (NB and RF), and for seven model evaluation metrics (pd, pf, precision, F1-measure, AUC, MCC and Brier Score). While rows represent each evaluation metric, columns represent model pair and used algorithms. The winner model is reported in the corresponding cell of the table. If the statistical comparison has medium or large effect size ($|d| > 0.5$), it is highlighted with bold. The "-" sign means that the two models perform statistically the same.

Figure 4.1 and Table 4.3 show that PM is not superior to SM and GM in the majority of cases. Particularly, when predictors are built with NB, SM and GM outperform PM in terms of pd, F1, and Brier Score. For example, PM produces 0.54 pd rate (median) while GM produces 0.84. However, traditional models also produce much higher false alarms (pf) than PM, i.e., PM's pf rate is 0.3 median while SM's and GM's is around 0.77. The lower pf rate of PM makes it superior to the traditional models in terms of pf, precision, and MCC.

On the other hand, when RF algorithm is used, SM and GM outperform the PM in terms of all seven model evaluation metrics. For example, PM produces 0.15 pd (median) while GM produces 0.3 pd (median). Please note that, pd rates of all three models are much lower, and practically not useful, when RF is used instead of NB during model training.

We do not report an analysis regarding individual developer results as we do in Chapter 3 to answer RQ1. There are very few selected developers in our industrial setting compared to those during the experiments performed on open-source projects. Therefore, the boxplots in Figure 4.1 already reflect the developers' individual performance.

> Considering the best performing algorithm (NB), general models, SM and GM, could detect more defects than PM ($\sim$0.83 versus 0.54 pd, respectively). On the other hand, traditional models gives very high false alarms compared to PM ($\sim$0.77 versus 0.3 pf, respectively). The lower pf rate of PM leads to a lower code inspection cost compared to high pf rates of SM and GM which alert developers on their changes redundantly.

4.5.2 RQ2: How can we improve the performance of defect predictors by utilizing the available commit data through a combination of statistical approaches?

This section reports the comparison of SDP models trained with various feature combinations and log filtering technique. Figure 4.2 and 4.3 show pd and pf rates of PM, SM, and GM, when the models are built with eight feature combinations. Each boxplot represents an SDP model listed in Table 4.2. In this section, we report results in terms of pd and pf only. In the appendix, you may find the results in terms of precision, F1, AUC, MCC, and Brier Score.

Results in Figure 4.2 and 4.3 show that using various data sources and data preprocessing techniques could improve the prediction performance. Particularly, including new features and applying log filtering always improve the prediction performance of PM in our experiment in terms of pd, while it also increases the pf rates. For example, combination of latent features, semantic features, and log filtering (B+LAT+SMN+LogF) improves the pd rate of baseline (B) from 0.53 to 0.78

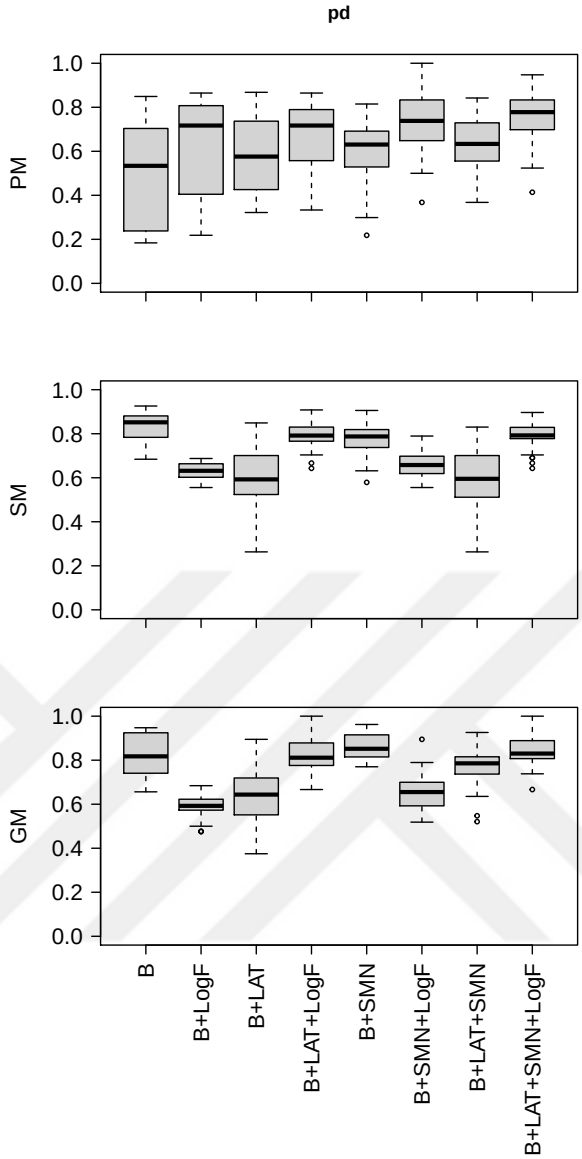

Figure 4.2 : Performance (in terms of pd) of PM, SM and GM built with combination of statistical approaches.

(median). However, there are still traditional models, i.e., GM with B+LAT+LogF, that outperform the PM with B+LAT+SMN+LogF model in terms of pd. On the other hand, PM with B+LAT+SMN+LogF produces higher pf rates (0.53) than the baseline model (B) PM (0.3).

Using various combinations of features and applying log filtering often deteriorates the traditional models' performance in terms of pd (except a few cases for GM), but also provides improvements on pf rates on the majority of models. For example, pd

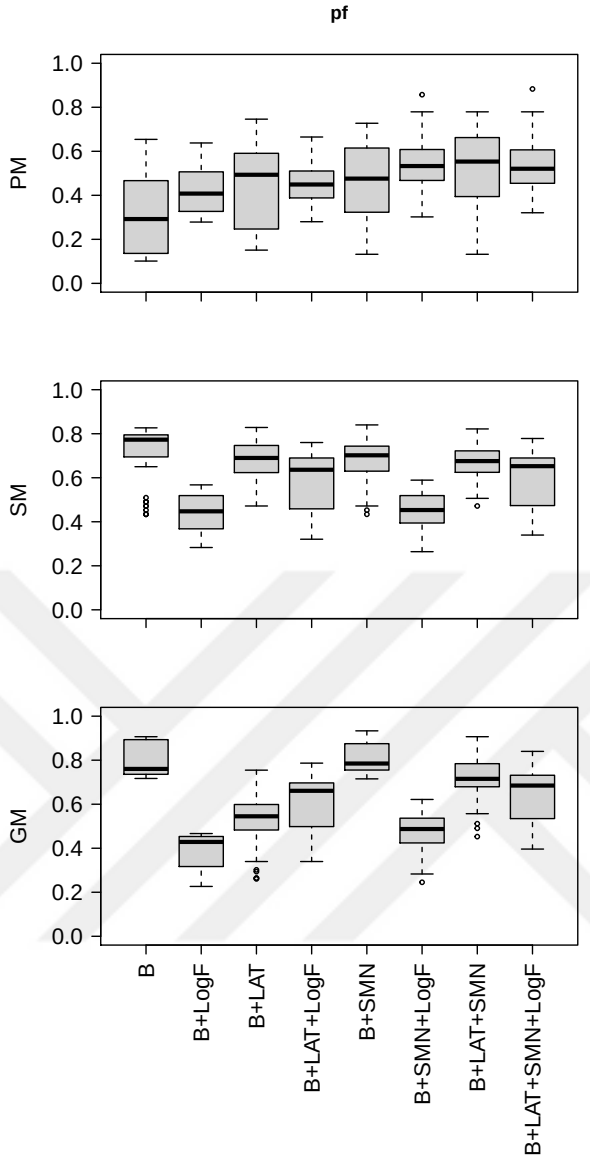

Figure 4.3 : Performance (in terms of pf) of PM, SM and GM built with combination of statistical approaches.

rate of SM with baseline (B) reduces from 0.85 to 0.63 when log filtering is applied (B+LogF), but pf rate of SM with baseline (B) decreases from 0.77 to 0.45 when log filtering is applied (B+LogF).

Considering the balance between the pd and pf rates, the best performing model among traditional models is GM with the base and latent features as well as log filtering (B+LAT+LogF). GM with B+LAT+LogF model produces one of the best pd values among all models (0.81). It performs statistically the same as GM with the baseline

(B) in terms of pd. Also, GM with B+LAT+LogF model reduces the baseline (B) GM's pf rate by 0.1.

Considering the Scott Knott ESD test results on all performance assessment criteria over the models, PM with B+LAT+SMN+LogF model and GM with B+LAT+LogF model are chosen as the best performing models. Both models utilize combination of different features and log filtering to train successful models. While PM with B+LAT+SMN+LogF performs better in terms of pf, precision, GM with B+LAT+LogF performs better in terms of pd and Brier Score. In terms of F1, AUC, and MCC, these two models are statistically the same.

As a conclusion, we pick GM with B+LAT+LogF model as the prototype to be deployed into the real development setting of the pilot project due to the applicability limitation of PM due to the developer turnover (see Section 4.7. Our industry partner also prioritizes having higher pd rates since both of the best performing models produce similarly high false alarms that need to be further challenged.

> Combining various features and statistical approaches lead us build more successful SDP models. More specifically, including latent features and applying log filtering improve the baseline GM's in terms of pf. While including latent features and topic models of commit messages, and applying log filtering technique improves the baseline PM's performance in terms of pd, it also increases false alarms.

4.5.3 RQ3: How does the importance of metrics used for defect prediction differ among PM, SM, and GM?

This section reports the InfoGain results for PM, SM, GM and assess the importance of metric groups for three SDP models. Table 4.4 shows the percentage of how many times each metric group (base, latent, and semantic) ranked in top 10 for each SDP model. The base metric group abbreviated as B, latents as LAT, and semantic metrics abbreviated as SMN. Each row indicates a rank, while columns represents the SDP models and metric groups.

Table 4.4 : Information gained from different metric groups in the first 10 rank.

	PM			SM			GM		
	B	LAT	SMN	B	LAT	SMN	B	LAT	SMN
1st	93,3	6,7	0,0	24,3	75,7	0,0	87,0	13,0	0,0
2nd	77,3	22,7	0,0	62,0	38,0	0,0	23,3	69,7	7,0
3rd	52,7	46,0	1,0	48,0	52,0	0,0	39,0	50,7	10,3
4th	48,0	51,3	0,7	66,7	33,3	0,0	56,3	38,7	5,0
5th	18,3	54,7	0,0	65,3	34,7	0,0	65,3	24,7	10,0
6th	49,0	49,3	1,3	61,0	39,0	0,0	65,0	26,7	8,3
7th	48,0	41,3	10,7	70,3	29,7	0,0	45,3	29,0	25,7
8th	45,3	20,0	34,7	72,0	23,3	0,0	33,0	44,3	22,7
9th	45,3	13,0	41,0	65,3	19,0	1,0	44,0	49,3	6,7
10th	38,7	17,3	42,3	52,3	10,7	0,0	45,7	51,7	2,7
Total	0,53	0,33	0,14	0,62	0,38	0,00	0,50	0,40	0,10

We apply the InfoGain on the model type that includes all metrics used in the experiment, which is B+LAT+SMN. Output of InfoGain shows the contribution of each metric on the prediction of bug-inducing software changes. We rank the output of InfoGain for each SDP model, PM, SM, and GM. The most important metric for the prediction is ranked first, and the second important metric ranked second and so on. For each SDP model, and for each rank, we calculate what percentage of each group of metrics appears in that rank. We conduct the analysis on the results for first 10 rank.

Results in Table 4.4 show that the base metrics' contribution to PM performance constitutes 53% of total contributions. The contribution of latent and semantic group constitutes 33% and 14% of total contributions, respectively. The importance of base metric set for PM especially stands out among the first ranked metrics: 93.3% of the metrics that are ranked first are from the base metric group.

GM InfoGain results follow a similar pattern to those of PM. When the top 10 ranked metrics are considered, 50% of total contributions come from base metrics, whereas 40% comes from latent, and 10% comes from semantic features. More specifically, the importance of base metrics in the first rank is very high, i.e., 87% of the first ranked metrics are base metrics.

SM results show that, when the top 10 ranked metrics are considered, the most contributing metric group is the base metrics (62%) followed by latent (38%). However, unlike the results of PM and GM, the latent group is the most important group when the first rank is considered only, i.e., 75.7% of the first ranked metrics are latent metrics. The semantic group does not appear in the top 10 list for SM.

> The proportion of contribution of each metric group to the prediction of bug-inducing changes slightly changes among PM, SM, and GM. The most important metric group consist of base metrics for all three SDP models. The latent group follows the base metrics and appears as the second important metric group. Furthermore, the semantic group does not seem to contribute to the prediction as much as the other metrics.

4.6 Threats to Validity

The data is collected utilizing the state-of-the-art techniques in the field. The base metrics of collected commits are calculated following the descriptions given in a prior study [27]. The SZZ algorithm is used to label bug-inducing changes [116].

Furthermore, a manual review is conducted to ensure the data quality. We were aware that SZZ is the state-of-the-art bug-inducing change technique that may produce false alarms [151]. Also, the bug reporting practices of development team may cause a noise in the marking process fixing changes. For example, the bug reports may not be properly reported by the development team or the bug report id may not be reported in the commit description when fixing a bug. All these cause a noise in the data labeling process. Therefore, manual reviews of the data regarding these issues with the help of industrial team help to reduce the noise in data. Also, the commits that does not include changes regarding software development (i.e., test commits) are identified and eliminated from the dataset before model building process.

Moreover, the SDP models are trained with well-known machine learning algorithms, Naive Bayes and Random Forest. Also, we apply 10x10-fold cross-validation. We repeat the model learning step 10 times after shuffling the data order each time to avoid learning bias [19] and utilize a state-of-the-art model validation technique used popularly in the field [16].Also, we validate our results using a time-sensitive model validation setup which is recently preferred by the researchers in the community [133]. The SDP models' performance are assessed with seven different well-known criteria (see Table 3.4), and statistical tests, i.e., Nemenyi test [135] and Cohen's d with corrected Hedges' g [136].

Suitable data pre-processing and feature extraction techniques are chosen from the literature to incorporate into our study to improve the performance of SDP models. Further, the SDP models are iterated with different number of semantic features in topic modeling and latent features in NNMF, but we report the best cases.

The findings in this study cannot be generalized to other projects, even though the experiment covers the state-of-the-art process metrics and techniques. Nevertheless, the results and challenges reported would guide other researchers in the community.

4.7 Concluding Remarks

In this chapter, 1) we investigate the performance of personalized model (PM) against two traditional models (SM and GM) in an industrial setting, 2) we utilize a combination of state-of-the-art process metrics, latent features of process metrics, topic models of commit messages, and log-filtering to improve the prediction models' performance, and 3) we analyze the importance of metric groups for PM, SM, and GM.

Our empirical analysis in the context of our industrial partner's pilot project show that SM and GM are able to detect more defects than PM, i.e., 0.83 vs. 0.54 probability of detection (pd) rates. On the other hand, PM produces lower false alarms compared to SM and GM, i.e., 0.3 vs. 0.77 pf rates.

Furthermore, applying log filtering, and utilizing extra information sources such as latent features of development activity and topic models of commit messages could improve the performance values of SDP models. Particularly, utilizing extra features and techniques improves pd rates of PM by 0.23, while decreasing pf rates of traditional models by 0.33. Considering the pd and pf rates and the balance between them, PM utilizes all feature groups and log filtering (B+LAT+SMN+LogF), and GM utilizes base metrics, semantic features and log filtering (B+LAT+LogF) models are the best performing ones among all assessed models.

Moreover, we observe that the base metric set used in the experiment, i.e., the state-of-the-art process metrics used in many change-level SDP studies [16,27], is the most important metric group for prediction of bug-inducing changes regardless of the model approach (i.e., personalized or traditional). The base metric set is followed by the latent features of process metrics. Further, the using topic models of commit messages does not contribute to the prediction performance as much as other metrics. When considering the models' performance and the extra calculation cost of topic modeling, we focus on a model that utilizes the base and latent metrics for deployment.

Even though our experiment shows that PM is capable of producing quite well prediction performance values in the research setting, a successful application of PM into practice could not be possible in the pilot project's setting mainly due to the cold-start problem [39,40]. The collected dataset includes a total number of 36 developers but only six contributors have sufficient development history to build a PM for each. The team turnover for the chosen pilot project is also high. The majority of those developers included in the dataset are not currently in the team. Furthermore, our collected dataset does not include the activity of developers who currently and actively work in the project. Therefore, PM cannot be built for developers who currently contribute to the pilot project.

We would like to note that, we also repeat our analysis using a time-sensitive experimental design, since the software change (commit) data has temporal properties. The main criterion in a typical time-sensitive approach is training a model with past data (i.e., commits before time t) while testing the trained model using the future commits (i.e., after time t). We use the same time-sensitive model validation approach depicted in Section 3.5.4: sliding window technique with 6-month time windows is used to build and test models. Our findings and answers to RQs have not changed when the time-sensitive model validation is used instead of cross-validation. Traditional models (SM and GM) better than PM in terms of pd, while PM is better than traditionals in terms of pf. Also, using latent features in addition to the base metrics and applying log filtering leads better performing prediction models, i.e., improves the pd rates of PM by 0.3.

This study is the first one that investigates personalized defect prediction in an industrial setting. Our findings show that the personalized approach is promising in a research setting, but the applicability of personalized models is limited due to the dynamically changing development team. We hope our research would be a guide to the researchers and practitioners in the SDP field on the challenges of personalized SDP models.

5. DEPLOYMENT OF A CHANGE-LEVEL SOFTWARE DEFECT PREDICTION SOLUTION INTO AN INDUSTRIAL SETTING[4]

5.1 Introduction

Software defect prediction (SDP) models support quality assurance activities of software teams to reduce the maintenance cost and effort by identifying the defect-prone modules of the software systems [19]. Recent SDP studies focus on predicting the bug-proneness of software systems at the change-level different than the conventional SDP models that predict bugs at a software entity level (i.e., software class, file, or method) [16]. Change-level defect predictors highlight defect-proneness of a software change immediately when a code change is committed to the software repository. Predicting defect-prone changes is a more useful strategy than predicting defects at a software entity (e.g., file) level since the former provides the possibility of taking immediate actions on the bug-prone modules while the change details are still fresh in developers' minds [16]. Another popular usage of change-level SDP models is prioritizing the software changes during the code review process in order to reduce the code inspection effort [16,130,132,164]. It decreases the review time as well as the amount of reviewed code by prioritizing bug-prone changes according to the predictions and by directly linking the owner of the reviewed software changes [16].

Many change-level SDP models have so far been built in industrial contexts [29,30,43,44,165]. Conducting studies in the industrial field is valuable in terms of validating the findings obtained in theory in practice [48]. Still, the number of industrial studies on change-level SDP models is quite low compared to the studies using publicly available datasets and open-source software projects [2]. One of the main reasons for this is the difficulty of accessing the commercial projects' data sources. This is not easy due to confidentiality reasons, while open-source projects'

[4]This chapter is based on the paper "Eken B., Tufan S., Tunaboylu A., Güler T., Atar R., Tosun A., 2021. Deployment of a change-level software defect prediction solution into an industrial setting. Journal of Software: Evolution and Process, 33(11), e2381.".

archives are publicly available and consist of longer development histories. Kamei et al. [2] point this out as a future challenge and suggest conducting more partnerships to overcome it.

Although change-level SDP aims to support a practical software quality assurance activity, academic studies often work in offline settings. To put it another way, researchers are mostly interested in the theoretical challenges of the SDP by utilizing historically collected batch data while they do not consider the real-life, e.g., online/dynamic, data flow. Recently, there is an increasing rate of studies that approach the SDP problem from the practical applicability perspective [29,43]–[46]. Based on these reported studies and our experience, the adoption of a change-level predictor into a real development environment brings brand-new challenges. First and foremost, in a real-life online setup, there is a temporal and active flow of commits, unlike the batch structure of an offline setup. Thus, the existing model training and validation strategies (i.e., cross-validation or a time-sensitive validation) do not reflect the reality of an online environment [29,166]. On the other hand, the continuous structure of real-life environments requires "keeping up with the fast pace of development" as Kamei et al. state [2]. A deployed model requires a continuous update, since the development environments may have shifting development characteristics in the long run [133,167]. Besides, during a real-life software development cycle, developers need some time to identify bug-inducing changes [29]. Therefore, successful deployment of an SDP model is not possible solely with the equipment, such as algorithms and input features, and the experience gained through the offline research studies. It requires a carefully designed integration and update mechanism to produce consistently 'good' prediction performance over time.

This study shares our experience in deploying a change-level defect prediction model into an industrial context. The prototypes we built in the offline setting for our industrial partner have been published in our previous work [148]. Here, we focus on the challenges faced in deploying the selected SDP prototype into the real-life development life cycle, such as determining a suitable model re-training period, and preventing the model from learning from noisy data, and producing consistently good prediction performance over time. We completely revise our experimental design

through an online defect prediction customized for our industrial partner. We assess the performance of an online SDP model against its offline version and further investigate the impact of training with potentially mislabeled (noisy) data and model update period on the prediction performance.

Our contributions and related findings in this study are:

1. We design an online prediction approach that simulates the real-life commit activity flow of the software project. The proposed online prediction approach trains the SDP model by considering the actual labels of bug-inducing commits, which are determined according to current observations at the time of prediction. The online prediction uses two parameters, namely "train-test (TT) gap" and "update period (UP)". TT gap determines the length of the gap between the train and test commits to avoid training with mislabeled commits. Besides, UP determines the model update (re-training) period using new data. We are inspired by the online prediction approach of Tan et al. [29], but our approach distinguishes from theirs while determining the labels of bug-inducing commits in the training set and setting the TT gaps and UP. We elaborate more on this in Section 5.4.1.

2. The SDP model's performance in offline and online prediction settings shows that both strategies produce similar performance depending on the available history of the development activity and the amount of mislabeled commits used during training.

3. We empirically analyze the effect of the TT gap on the prediction performance by setting it to 11 different values, i.e., namely 0-month to 10-months, during online prediction. Whereas the existing studies [29,168] suggest a fixed time gap to avoid learning from noisy labels. Our results indicate that keeping a suitable time gap between the train and test commits would improve prediction performance, by up to 37% in terms of probability of detection, by reducing the undiscovered bug-inducing commit rates.

4. We empirically analyze the impact of model update frequency on the prediction performance using nine different UP values, namely, between updating every day

to every 120 days. The analysis over all TT gaps shows that the UP value would affect the performance by up to 18% in terms of probability of detection.

5. We share our learned lessons regarding the deployment of our change-level SDP solution in Section 5.8. We discuss the challenging aspects of the communication, data collection, development habits, the software project structure, and the interpretability of the produced predictions that potentially affect the applicability and the usability of any SDP model in practice.

5.1.1 Structure of the chapter

Section 5.2 reports the related literature on change-level SDP, change-level SDP studies conducted in industrial context, and on the model validation techniques applied in change-level SDP studies. Section 5.3 describes the research, data collection, and model prototyping phases. Section 5.4 reports challenges our SDP model deployment experience and describes our online prediction methodology used during our experimentation. Section 5.5 reports the research questions and our empirical setup methodology. Section 5.6 shares the performance for the defined research questions and discusses the achieved results. Section 5.7 discusses the threats to the validity of the conducted empirical study. Section 5.8 reports additional lessons learned during this industrial case study. Section 5.9 concludes the paper.

5.2 Related Work

Studies in the SDP field exploit data collected from various software development units (people, product, and process [18,105]), and utilize various machine learning and statistical approaches [11] to predict defective software modules in both open-source and industrial projects. Besides, SDP models are designed to make predictions for various software artifacts such as software class or method. According to the prior studies' classification on the SDP strategies [16,28], the long-term SDP models make long-term predictions about the bug-proneness of a software entity, e.g., package, file, class, method, whereas the short-term SDP studies focus on predicting the bug-proneness of a software change to locate buggy parts of the software as soon as possible by giving immediate feedback to developers. Just-in-time [16], change-level

[29], commit-level and short-term refers to such SDP models that make prediction at software change-level. Moreover, the prediction performance of the change-level SDP models are often evaluated using two popular model validation strategies, namely cross-validation and time-sensitive validation.

In the next subsections, we report the related work focusing on change-level SDP prediction, more specifically those studies conducted in an industrial context and different model validation approaches used in change-level SDP.

5.2.1 Change-level SDP studies

The earliest software change-level defect prediction is reported by Mockus et al. [169]. They define a set of software change measurements that measure a code modification from size, diffusion, and developer expertise aspects in order to understand and quantify the software changes. They predict the quality of changes in a large-scale telecommunication system by using the proposed measurements. Later, Mockus and Weiss [30] proposed a model that predicts the risky software changes at a coarse-grained granularity in an industrial telecommunication system. Their model identifies the high-risk initial maintenance requests (IMR) that contain multiple software changes. Failure probabilities of the IMRs are predicted by utilizing the size, diffusion, developer expertise, and the purpose (whether the change was a fix) properties of a change. The number of subsystems modified and developer experience are found to be indicators of these high-risk changes. Czerwonka et al. [165] share their experiences with a tool named CRANE that analyzes the risk of software changes during the development of Windows Vista. Shihab et al. [115] utilize change properties such as developer experience, history of the changed files, the time when the change is made, the size of the change, and the purpose of change (bug fix or not) to predict the risky changes in an industrial context, similar to the work in [30], but at a finer granularity. The number of added lines and chunks, bug-proneness of the modified files, number of bugs, the number of bug reports linked to a change, and the developer experience are found to be good indicators of a change's risk-proneness.

The introduction of the SZZ algorithm [2] and the emergence of machine learning techniques [11] led to many change-level SDP studies that utilize measures from

historical bug-inducing changes to detect future bug-inducing changes. Kim et al. [117] propose classifying the software changes as buggy or clean using the number of added and deleted code lines, change metadata information (i.e., commit hour, commit day), complexity metrics of the modified files (i.e., cyclomatic complexity), and textual features of the modified file and directory names. They conducted their work on 12 open-source projects. Kamei et al. [16] empirically assess their just-in-time SDP for six open-source projects and five commercial projects. Their prediction models utilize the diffusion, size, history, purpose, and developer experience aspects of the change and perform with an average recall of 64%. Their prediction model is able to reduce code review effort, i.e., 35% of all predicted bug-inducing changes could be identified by spending 20% of the total effort. The authors also report effort-aware performance measures instead of traditional performance metrics only, such as recall, false-alarm rate, F1-measure [170]. Effort-aware performance is basically calculated as the development team's effort required to inspect code changes predicted as bug-inducing [16,44,130,132,164,171].

There are also other studies that focus on applying alternative techniques for change-level SDP in open-source projects. Tosun et al. [27] show that prioritizing the prediction of high-impact bug-inducing changes improves the inspection effort by 4%. Other researchers propose predicting bug-inducing changes at a finer granularity level, such as risky files [28] or code lines [120] in software changes. Kamei et al. [172] utilize cross-project data to predict risky changes. Catalino et al. [173] apply the change-level SDP on mobile software development context. Barnett et al. [155] incorporate textual features of commit messages into the change-level SDP. Last but not least, Yang et al. [174] and Hoang et al. [20] use deep learning for just-in-time SDP.

5.2.2 Change-level SDP studies in industrial context

The majority of the reported change-level SDP studies are conducted on open-source projects [20,27,28,36,53,155,172,174,175]. Fewer studies focus on the change-level defect prediction in an industrial context [16,29,30,115,165].

Kamei et al. [16] discuss the differences between industrial and open-source systems over the characteristics of the bug-inducing changes. They found out that different factors affect the prediction performance among different project domains, i.e., the number of files changed, the information of whether or not the change fixes a bug, and the average time interval since the previous change are important factors for open-source projects, while the diffusion factors are important for industrial projects.

Tan et al. [29] apply the change-level SDP in practice with a Cisco project. Their study focuses on data re-sampling techniques to cope with the low buggy rates in a propriety software project. They also evaluate their change-level SDP model by using an online prediction approach to overcome the incorrect performance results gathered with cross-validation. We elaborate more on their online evaluation technique below in this section. Their experience regarding the integration of the proposed SDP model shows that prediction results need to be supported with explanations to be actionable.

Altinger et al. [43] report a performance tuning experience of a change-level SPD model built for an automotive software project. They investigate the effect of data balancing techniques and classification algorithms. Their study indicates re-sampling techniques should be considered carefully, and the Naive Bayes algorithm yields a good and stable performance.

Yan et al. [44] share their experience of effort-aware just-in-time defect identification on 14 Alibaba projects. They select three supervised and two unsupervised state-of-the-art SDP model building approaches to investigate them in the Alibaba context. Their results show that the best performing approach is the 'Classify Before Sorting' approach proposed by Huang et al. [171]. Moreover, they indicate that the importance of features in change-level defect prediction differs for open-source and industrial projects. However, the diffusion features (i.e., the number of changed files) are the most important ones for both types of software settings. Finally, their investigation with the software development team shows that the deployed effort-aware SDP tool helps developers to reduce their effort on code inspection while detecting the buggy changes correctly 33% of the time.

Kang et al. [45] build a change-level SDP model for the maritime industry and report that their prediction model finds 56% of the total bug-inducing changes when 20% of the total effort is spent. Their research points out that the maritime industry has high post-release quality cost due to domain characteristics, i.e., bug-inducing changes most likely occur again in the entire series of ships once they occur. Using a cost estimation model that reflects the industry domain characteristics, Kang et al. [45] achieves a reduction in the post-release quality cost by 37%.

A recent study by Khanan et al. [46] does not specifically focus on an industrial context but focuses on the integration of change-level SDP into the real development environment. They propose a framework that integrates into the GitHub development workflow of a project. Their proposed application collects the commits and pull requests of the project and trains an SDP model. When the SDP model is trained, it provides prediction to the developer on the bug-proneness of a commit. Besides the probability of being risky, the framework also reports the factors (i.e., metrics used during model training, such as entropy of the change) related to a commit. Although this study shows a real application of SDP models, the authors do not consider or report a way for the future updates (re-training) of the integrated model, and the potential noise in the labels of the dataset. In this study, we share the same focus on the integration of a change-level SDP model but report potential challenges and provide solutions and feedback to the developers.

5.2.3 Model validation approaches used in change-level SDP studies

Change-level SDP studies follow different model validation strategies that fall under two main approaches: cross-validation and time-sensitive. Cross-validation is widely used among researchers to evaluate their change-level SDP models [16,20,27,131,132]. A change-level SDP model predicts the bug-proneness of future changes by utilizing prior changes in the software. Therefore, recent studies [28,29,133] use a time-sensitive approach to train and assess the change-level defect predictors. They criticize the appropriateness of using cross-validation while building a change-level defect predictor. They argue that cross-validation ignores the temporal properties of the data instances by shuffling and dividing them without considering this property.

On the other hand, a couple of studies empirically assess their change-level SDP models by using both cross-validation and time-sensitive validation strategies [20,130]. Their results demonstrate that both strategies yield similar performance, and the conclusions are consistent among different strategies. Studies on machine learning also discuss that cross-validation based training may fail to make successful predictions for future data instances. However, it leads to more robust predictions than a time-sensitive training [176].

Recently, Tan et al. [29] emphasize the limitations of using a time-sensitive approach when training change-level SDP models. Prior change-level SDP experiments using both cross-validation and time-sensitive approaches ignore the temporality in the labeling process of bug-inducing changes. During an SDP model training, all bug-inducing changes in the training data are assumed to be known. Also, all bug-inducing changes in the test data are assumed to be known while evaluating the proposed SDP model. However, in real life, some training instances may be bug-free at time t, whereas they might be associated with a bug-fix later at time t+1. In other words, prior studies might have used the mislabeled change information. Tan et al. [29] apply "online machine learning" approach [177] to update the training data labels as well as to extend the training data with new instances. The training label updates are done based on the available information at the end of the specified test set.

Moreover, during their online prediction, they keep a gap between the training and the test sets in order to avoid the potential mislabeled changes because it may take years to discover a bug-inducing change [178]. We are inspired by Tan et al.'s online prediction approach [29] during the deployment process of our defect prediction model. We customize the train-test gap, test size and the model update strategy accordingly. However, we observe that their approach requires an improvement on the labeling strategy of training instances. In our online setup, we use the available information at the beginning of the test set only, instead of utilizing future information for the training data. More details are explained in Section 5.4.1.

5.3 Research Background

We had a nationally funded research collaboration between Ericsson Turkey and Software Modeling and Analysis Lab, Computer Engineering Department at Istanbul Technical University during the years 2018-2020. Ericsson Turkey develops software solutions for leading telecommunication companies in Turkey and exports their software solutions to various continents in the world. Software quality and end-user satisfaction are very important for the company, and to accomplish that, the teams have been using several tools for testing, test automation and other verification and validation activities. The company aims to reduce the cost of testing and code review activities by integrating a defect prediction model into their software development process. Therefore, we have been working on building and deploying customized software defect prediction models according to the needs of developers in our research collaboration.

The responsibilities of researchers in this project are to investigate the set of approaches and techniques to deploy a successful defect predictor at software change-level for a chosen software project of our industrial partner. During the early research phase of the project, we collected historical development activity, extracted software process metrics, and detected the bug-inducing commits to form the SDP dataset. Later, we continued investigating techniques to build an SDP model by applying various machine learning approaches. We report those experiences in a prior study of ours [148]. In this paper, we would like to report our experiments on the deployment phase of our research collaboration, and share our insights gained during this process. Although the focus of this paper is the deployment phase of the chosen defect prediction model, we also briefly summarize the earlier project phases to provide a solid background and to report a step-by-step guide for the readers.

5.3.1 Dataset construction

We selected a pilot software project together with our industrial partner to build and evaluate the SDP model prototypes. The practitioners describe the chosen pilot project as an active project (since 2013) that has been sold to multiple customers, continuously modified, refactored and enhanced according to the customer requests. The software

team states that each new feature or feature improvement request of customers fully occupies the team's annual resources. The software system has a core module with many features, and additional modules designed and customized according to customer needs. The software development methodology is similar to a waterfall process, such that feature requests are delivered in yearly releases, and there is a core team with predefined roles and responsibilities. The team utilizes version control systems and issue repositories, and puts emphasis on traceability of changes. The SDP solution we built in this research collaboration learns from the development activity of all modules in this pilot project, and provides feedback that addresses any code change in the whole project.

We prepared a change-level SDP dataset by collecting the historical development activity of the chosen software project. The prepared dataset contains a set of historical software changes (commits), a binary label for each of those changes that represent their defect-proneness (bug-inducing or clean), and a set of process metrics calculated for each collected software change.

5.3.1.1 Collecting historical commits and identifying bug-inducing changes

Historical software development activities of the pilot project are gathered through Git [179] source code repository. At the beginning, we collected the logs of a total number of 3927 commits between November 2013 and May 2018. A single commit log contains a commit hash, committer name, commit date, commit description, the name of the changed files, the number of files changed, and the number of added and deleted lines in each file. After discussing with the development team leaders, we decided to filter some of the commits which do not contain an actual development activity. 1694 automatic system commits and 164 merge commits are detected by searching a set of keywords in the commit logs (e.g., *merge* for merge commits). 940 commits belonging to the test modules are also identified and filtered. Also, eight bulk commits are detected manually by reviewing the commits' logs based on their number of files and number of modified lines. After cleaning the dataset from these automatic system commits, merge commits, test related commits, and bulk commits, we ended up having a total number of 1773 commits. Please note that a commit could

be filtered due to being in multiple categories, i.e., it could be a test commit and an automatic system commit. So, the number of filtered commits is fewer than the total of automatic, merge, test, and bulk commits.

We implemented the SZZ algorithm [116] to find out bug-inducing changes in our dataset. It is assumed that a bug is injected into the software by changing some code line(s), and the introduced bug is fixed by changing those line(s) again. First, the bug-fixing commits are determined using the bug reports stored in the issue management and tracking system JIRA [180] used by the software development team. A commit is marked as a fixing commit if its commit message mentions any issue ID that corresponds to a bug report in JIRA. Second, for each detected bug-fixing commit, we applied *git diff* command to query the modifications made by the bug-fixing commit. Later, each code line deleted or modified by that bug-fixing commit is searched down through the prior changes. The search process is conducted with *git log –until bug_report_date -S modified_line* command. Whenever a prior commit is matched with the searched code line, the matched commit is blamed for introducing the bug. Since a bug must be introduced into the software before the related bug report is created in JIRA, the SZZ algorithm only considers the prior software changes to the date the bug was reported. Moreover, we assume comment lines or configuration file modifications cannot be the cause of a bug. Therefore, we ignore the comment lines and Maven (a project management tool [181]), configuration file changes while searching back for the location of the commit where the bug is injected.

Bug-fixing commits were found by searching the issue key(s) of the bug reports in the commit messages of the collected commits. 365 of 1998 issues reported in JIRA between January 2014 and May 2018 were linked with the fixing commits. However, in practice, a bug that occurred in a software system and/or fixed by the software team may not always be reported properly. Sometimes, several development tasks are completed with a single issue ID, even though some of those are done due to a bug fix. To ensure the data correctness and capture all bug-fixing activities, the development leaders who have expertise on the pilot project manually went through the commits' messages and content if necessary and marked additional bug-fixing commits. 135 out of 394 bug-fixing commits were detected using manual investigation.

With the profound knowledge of the development leaders on the project, we made sure the bug-fixing commits are correctly specified for the pilot project through several iterations on the dataset.

A commit can consist of modifications on many code files and lines, and hence, more than one commit can be blamed for one bug-fix operation. Also, a bug-fixing commit can fix more than one bug. There are many-to-many relationships between bug-introducing commits and bug-fixing commits. In total, 394 bug-fixing commits are linked to a total number of 296 bug-inducing commits identified via SZZ.

The final version of our dataset contains the development activity of 36 developers made between November 2013 and May 2018, a total number of 1773 commits, and 296 bug-inducing commits. Figure 5.1 reports the distribution of bug-inducing and clean (not bug-inducing) commits over time.

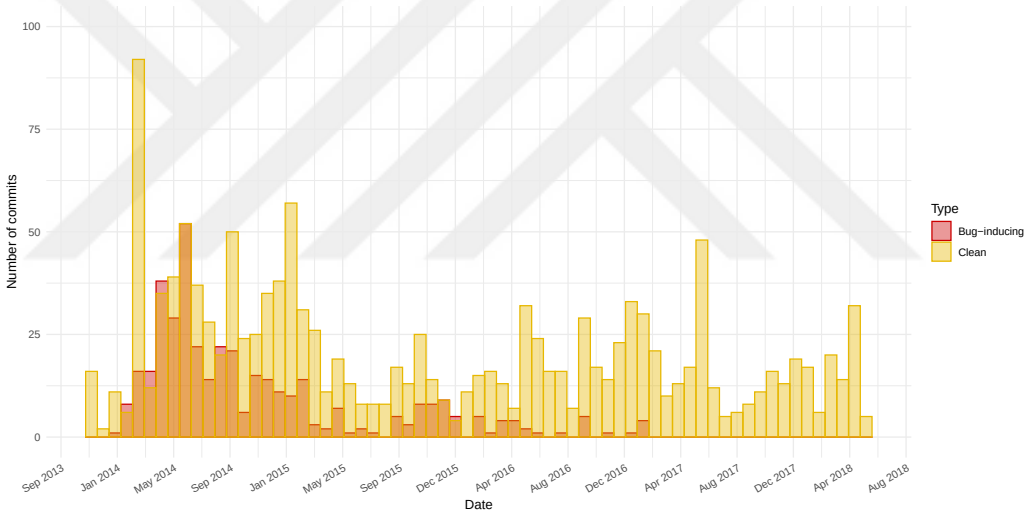

Figure 5.1 : A timeline of our SDP dataset that represents bug-inducing and clean commits.

There are more commits and bug-inducing commits made in 2014 and at the beginning of 2015 when compared to the lifetime of the project. The development team also states that there was a productization for two customers from 2014 to 2015, and the team later focused on improving and modifying the product. Therefore, the project's development activity was higher at the beginning. Later, the core requirements decrease, but improvement requests increase over time, so do the bugs and fixes as well. We don't have any bug-inducing commits made after February 2017 in our dataset,

even though the latest commits we collected go until May 2018. This is a natural consequence of our bug-inducing commit detection mechanism. SZZ algorithm traces back from the bug-fixing commits to find out the related bug-introducing commits. Since we did not identify any bug-fixing commits at the later periods of the collected data, bug-inducing commits could not also be traced back. This would eventually affect the time period used to train and test our SDP model.

Please note that the statistics on the collected commits and the details on the process metrics reported in this paper do not exactly match those reported in our prior publication [148], since we filtered out some commits (i.e., test commits) after our discussions with the development team. Also, the base metric set is updated when transitioning the model from the prototyping to the deployment stage.

5.3.1.2 Base metric extraction

We extracted 12 process metrics from the collected commits in our dataset. These process metrics measure software changes in four aspects; size, history, diffusion, and the developer experience. The complete list of metrics and their descriptions are given in Table 5.1.

Table 5.1 : Base metrics utilized in this study.

Metric	Description (abbreviation)
Size	Lines of Code added in a commit (ADD)
	Lines of Code deleted in a commit (DEL)
	Churn of commit: ADD + DEL (CHURN)
Diffusion	Number of modified files (NF)
	Number of modified directories (ND)
	Entropy: distribution of modified code across each file (ENT)
History	Number of developers had changed the modified files (NDEV)
	Number of prior changes to modified files (NPC)
	The average time interval between the last and the current change (AGE)
Experience	Experience of developer (EXP)
	Recent experience of the developer (REXP)
	Experience of developer the on a subsystem (SEXP)

We chose the metrics listed in Table 5.1 as our base metric set, since they are popularly used, state-of-the-art metrics utilized in change-level SDP studies [30,109]–[113].

The *Size* metrics measure the number of lines of code added (ADD), deleted (DEL), and changed (CHURN) in a single commit, and directly extracted from a commit's log kept by the version control system (Git).

The *Diffusion* metrics basically measure the distribution of a software change (commit) across the modified file(s). Similar to the *Size*, the *Diffusion* aspect of a single commit is also calculated directly from the commit's log. NF is the number of files modified in the commit, whereas ND is the unique number of directories where the modified file(s) are located. The entropy of a commit (ENT) is calculated by utilizing the churn (the number of total changed lines of code) of files modified in the commit by using the formula in Equation 5.1.

$$ENT = \sum_{k=1}^{NF} p_k log(p_k) \quad where: p_k = \frac{file\ churn}{commit\ churn} \tag{5.1}$$

The *History* metrics measure the history of a commit through the modified file(s). We traced back the modified file(s) across prior commits' logs to calculate NDEV, NPC, and AGE metrics. The formula used to calculate NDEV is *NF/ (the number of unique authors who previously changed the modified files)*. The formula used for NPC and AGE metrics are given in Equation 5.2 and 5.3, respectively.

$$NPC = \frac{total\ number\ of\ prior\ commits\ on\ the\ modified\ files}{NF} \tag{5.2}$$

$$AGE = \frac{\sum_{f=1}^{NF} number\ of\ days\ since\ the\ last\ commit\ f}{NF} \tag{5.3}$$

The *Experience* metrics measure the experience of the developer who made the change (commit) using the contribution history of the developer. EXP metric is the count of prior commits made by the developer, while the SEXP metric is the count of prior commits that modifies the same subsystem(s) modified in the current commit. The subsystem in which the modified file is located is identified by parsing the full name of the file. For instance, the subsystem of a source code file "X/Y/Z/fileName.java" is

determined as “X”. REXP metric is calculated similarly to EXP metric, but the recent commits of the developer have more importance than the older commits. A commit’s importance is calculated using the formula given in Equation 5.4. The time difference between the current commit and a prior commit is measured over three months time intervals. Therefore, the importance of a prior commit decreases every three months while going back through historically.

$$REXP = \sum_{p}^{prior\ commits_d} \frac{1}{1 + |p - current\ commit_d|\ \%3} \tag{5.4}$$

The calculations of the two process metrics (REXP and SEXP) listed in Table 5.1 are adjusted according to the pilot project’s process and file structure. REXP metric is the same as EXP, but it gives a higher weight to the recent commits of the developer. To define ‘recent’ experience, we agreed with the development leads, based on the commit distribution over time in our dataset (Figure 5.1), that it should be set to the last three months. Besides, the number of subsystems (NS) metric is removed, and SEXP metric is re-calculated to better represent the file/module structure of the software project.

5.3.2 Initial prototype SDP models

During the research phase of our project, we investigated various model building approaches and techniques to propose a successful change-level defect prediction model for our industrial partner’s pilot project.

We extracted additional metrics besides the base process metrics (Table 5.1) in order to improve the prediction performance of the SDP models by utilizing various and multiple data sources [148]. First, we extracted five new numeric latent features from the 12 base metrics (Table 5.1) using non-negative matrix factorization (NNMF) [160] in order to capture potentially hidden information. Second, we extracted semantic features from the commits’ textual messages through topic modeling. However, Information Gain (InfoGain) [141] analysis reported in our prior study [148] shows us that the semantic features are not important as the base and latent metrics for the prediction of bug-inducing changes in our dataset. Considering the additional calculation effort of semantic metrics and their lower contribution to the performance

of the defect prediction model, we proceeded to the deployment phase by utilizing only the base metrics and latent metric sets. The technical details on the extraction of these additional metrics can be found in our prior study [148].

We trained our SDP models using machine learning algorithms that are successful in the SDP studies, namely Naive Bayes (NB), Random Forest (RF), Logistic Regression (LR), XGBoost. The NB algorithm is more lightweight in terms of training and prediction when compared to RF, LR, and XGBoost, and models built with NB produce higher performance values in our context [148]. So, we chose to deploy the SDP model built with NB.

We applied two different data filtering approaches before training the SDP models: Log filtering and feature ranking. Selecting the top features using InfoGain gave inconclusive results, whereas applying log filter to the metric values improved the performance up to 20% when prediction models were trained with NB [148]. Therefore, we decided to apply only the log filtering approach on the final set of metrics (base and latent factors).

We proposed various types of SDP models to our industrial partner; a personalized model (PM), a general model (GM), and a selected general model (SM). Different types of models use different train data, and their output has different targets. More specifically, a PM is trained for each developer in the team to give recommendations to the relevant developer only, by using the developers' own commit history solely. The GM model is a typical, traditional defect prediction model; it learns from the whole teams' commit history to give recommendations to any developer in the team on their future commits. SM is a variant of the general model trained with a selected set of developers' (top contributors) development activity.

We chose the most promising approach in terms of prediction performance for deployment; GM and SM. During the deployment, we observe that GM performs better and is more robust than SM. GM utilizes more diverse data during training, whereas SM utilizes a subset of commit history according to the selected developers for the specified commit period. Accordingly, we deployed the GM as our primary prediction

solution, whereas the SM is currently under beta testing in the company. In this study, we only focus on the deployment and assessment of GM.

We assessed the performance of the model prototypes using two different validation strategies: 1) K-fold cross-validation, and 2) Offline time-sensitive validation. According to the results, we preferred to choose the offline time-sensitive approach during prototyping.

Time-aware model training and validation approaches are suitable for such cases in which data instances have temporal properties. We named our time-sensitive validation approach as *offline* time-sensitive prediction during prototyping to emphasize its difference from the *online* time-sensitive prediction approach used for deployment in the rest of the paper. The offline prediction refers to training the model with the commits taken from a prior time period using their ground truth class labels, and predicting the commits in the next time period. In other words, ground truth labels (bug-inducing or clean) of commits are known from the beginning of the model evaluation, and the training and test sets are formed according to the times of the commits only.

5.4 Deployment with an Online Prediction

Together with the development team of our industrial partner, we have decided to integrate the SDP models chosen among the proposed prototypes into a test development branch of the pilot software project. Using a test branch at the beginning of the deployment gives us the opportunity of testing the integrated model and the operation of our solutions by simulating the real development environment of the company as accurately as possible. In later stages, the company aims to integrate the SDP solution into their main development branch in order to use the solution prior to their code review activities. The plan is to assist the code reviewers with the predictions of the SDP model.

Initially, we integrated a set of pre-trained models into the development environment of the pilot software project. The integrated SDP models are trained with the commit activity we already collected at the beginning of our research. However,

the development team needs a sustainable solution that also learns from the recent development activity. Therefore, we design a solution that collects the recent commits of the project periodically and feeds the SDP model with the newly collected information. The integrated solution collects the recent commits, and extracts the process metrics (see Table 5.1) and latent features of those metrics. It also labels the bug-inducing commits among the collected commits using the SZZ algorithm, and re-trains the SDP model with all the historical information as well as the newly collected commit activity.

At this point, we need to make two design decisions for the deployment: 1) how often the SDP model should be updated (re-training), 2) what part of the commit data should be used during the model update. The first decision matters because we aim to update the deployed model frequently enough to learn from the recent software changes so that we do not compromise on the prediction performance, while we should avoid too frequent model update cycles. The second decision also matters because, in real life, we do not know whether the incoming commits are actually bug-inducing or not until a JIRA issue is associated with a bug-fix. So, the commits included into the training must be accurately labeled. This way, we would mitigate the impact of misclassified commits on the prediction performance.

We elaborate on these design decisions during an online prediction strategy we proposed for our industrial partner. We use the online prediction strategy to understand the real-life performance of the SDP model along with an empirical investigation on these deployment decisions.

5.4.1 Online prediction

We evaluate the deployed SDP model with an online prediction because the offline prediction used during prototyping does not reflect the structure of the real-life development environment. The key point of an online prediction is that it simulates the real-life data flow: it trains the prediction model with the available information at a specified time [29,182]. However, the offline prediction, i.e., cross-validation or offline time-sensitive approach, utilizes batch data containing the final, ground truth information. Even though the offline time-sensitive approach takes into account the

temporality of commits while splitting the data, it still introduces a measurement bias regarding the labels of training data instances. In the offline time-sensitive approach, we split data into many pieces over time: t_1, t_2, ..., t_n. To make a prediction for the commits at t_k, the training set contains metrics reflecting the commits done until t_k, whereas their class labels are not taken as the values are known at the time t_k. On the contrary, the actual class labels of the training instances, which may be found out after t_k, are fed into the model. Thus, even though the time splits help to feed the model with consequence commits, their labels do not reflect the actual scenario. In real life, the class labels of the commits must represent what we know at the time t_k.

Furthermore, bug-inducing commits require some time to be detected based on the life cycle of the bugs in a software system. The basic life cycle of a bug starts when a bug is introduced into the system by a developer through a commit. Then, that bug is discovered by a team member (often testers) and reported as an issue. Lastly, the reported bug is fixed by a developer. Using SZZ, we are able to utilize the bug-fixing commit to find the corresponding commits inducing that bug. Therefore, the bugs could not be linked to their fixing and inducing commits until they are fixed. So, in an online prediction strategy, some commits may look clean at any point in time, although in later fixing commits, it may be found out that those commits were actually bug-inducing.

Figure 5.2 shows the time required to fix a bug after it was injected into the selected pilot project. This figure indicates that the time required to discover a bug is 103 days on average (approximately three and a half months) and a median of 195 days (approximately six and a half months). Also, it goes up to 1325 days (approximately three and a half years) to find out a bug-inducing commit associated to a fixing commit.

Since it takes time to detect changes that cause bug(s), i.e., 195 days (median), keeping a temporal gap between the train and test data is reasonable regarding the deployed model's prediction performance. On the discovery of bug-inducing commits, Cabral et al. [168] suggest that accepting the bug-fixing time duration as 90 days is an appropriate choice in open-source systems. But, in this study, we want to

experimentally find a convenient time gap that filters out the noisy training labels during the model learning process for our selected project's context.

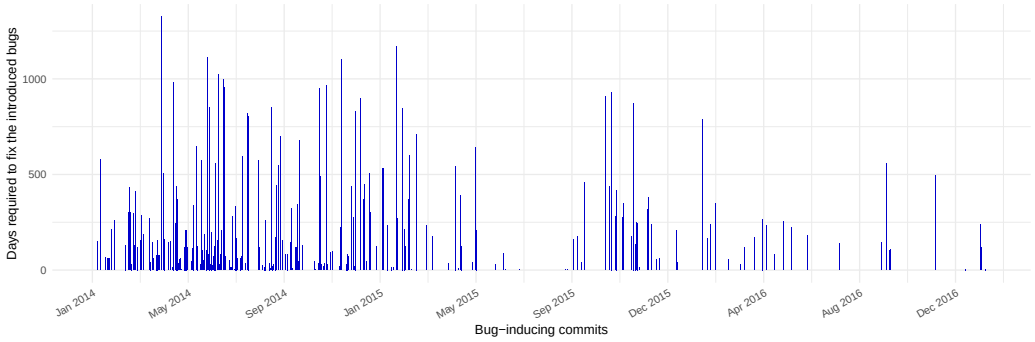

Figure 5.2 : A timeline of the time difference (in days) between the bug-inducing and fixing commit pairs.

Figure 5.3 shows the overall design of our online prediction strategy against the offline strategy. The train set in both prediction strategies involves the same commits in their training set, but the labels of bug-inducing ones might be different. While the commit labels are determined using the information known at the prediction time during the online model training, the ground truth labels of the commits are used to train the SDP model during the offline model training. We set the initial training set's length to four months, and it contains a total number of 166 commits for both prediction strategies. Due to the difference in labeling mechanism, there are four bug-inducing commits in the initial training set of the online prediction, whereas there are 33 bug-inducing commits during the initial model training of the offline prediction. Moreover, we set a gap called Train-Test (TT) Gap in the online strategy, whereas this gap does not exist in the offline strategy. As we move to the next time frame (t+UP) in Figure 5.3, a portion of the unused commits from the TT gap is added to the training set (assuming that their labels are identified during the TT gap duration). The model is then re-trained using these additional training data, and tested on a new test set. More details on TT gap and UP parameters are given below.

Train-test (TT) gap: Due to the time required to discover bug-inducing changes in the software, SDP models may learn from noisy data since there is a high chance that some data instances utilized during training are very likely to be mislabeled as clean. The training set inevitably contains noise to some extent, but we aim to reduce this rate

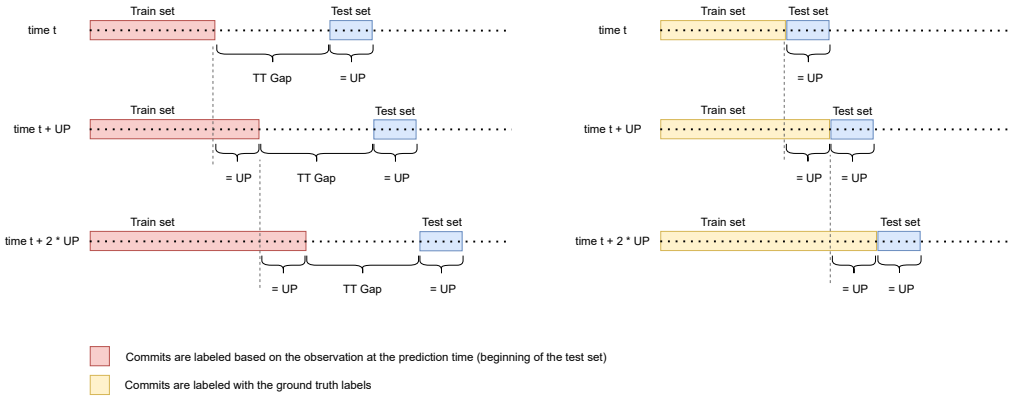

Figure 5.3 : Demonstration of the online and offline prediction strategies.

of mislabeled instances in the training set. Therefore, in the deployment environment, we exclude the potentially noisy data during the model re-training.

We use the online prediction strategy to decide how much data we should exclude during model re-training. We define an unused time gap between the training and the test set and call it TT gap that demonstrated in Figure 5.3 as TT Gap. Here, we aim to keep a distance between the last training instance and the first test instance based on the statistics observed in Figure 5.2. Setting the TT gap to an average of three and a half months would be the first choice. However, we aim to observe the effect of various lengths of the TT gap on the prediction performance to set the most convenient TT gap for the deployed model. We executed the online prediction experiment multiple times by setting the TT gap from zero (0) month to 10 months. Please note that, the metrics of the commits in the test set are calculated considering the total number of all prior commits.

Update period (UP): Another parameter we use in our online prediction design is the update period, UP in short. The value of UP determines how frequently the model is fed with new commits. UP parameter was also used by Tan et al. [29]. Figure 5.3 demonstrates three iterations of the online prediction. The first iteration of the online prediction experiment runs at time *t*, the second iteration runs at time *t + UP*, and the third iteration runs at time *t + 2*UP*. Suppose we set the UP to one day, that means our SDP model is updated every day: the initial SDP model is built on time *t*, the first model update is conducted one day later at time *t + 1 day*, and so on. Consequently, UP also determines the approximate amount of the new commits included into the

training set during the model update, as well as the length of the test set. If the UP is one day, that means our test set in every iteration contains the commits done on the specified day only. Similar to TT gap parameter, we assess the effect of UP on the prediction performance by setting UP to 1-day, 3-day, 5-day, 7-day, 10-day, 14-day, 30-day, 60-day, and 120-day.

We would like to mention that the online prediction strategy used by Tan et al. [29] is also very similar to ours. Both designs keep a time gap between the training and test sets to avoid the potential impact of mislabeled data instances (TT Gap in our study) on the prediction performance, and uses a parameter to define the period of the model update (UP in our study). The main difference between the two online prediction designs is the labeling of the software changes (bug-inducing or clean). Tan et al. [29] indicate that their online prediction determines the labels of the training instances based on the information at the *last* commit of a test set. However, this implies that the prior commits in the test set are predicted by a model trained with future observations. Our approach, on the other hand, updates the labels of training data instances based on the observations at the model update day, which corresponds to the *first* commit in the test set. Therefore, we believe our design reflects the real-life development environment more precisely.

5.5 Methodology

We focus on three research questions in this study to assess our design decisions during SDP model deployment:

- **RQ1**: Does the performance of the proposed SDP model assessed with the offline time-sensitive prediction strategy represent that with the online time-sensitive prediction strategy?
- **RQ2**: To what extent does the train-test gap affect the performance of the deployed model?
- **RQ3**: What impact has the update period of the proposed model on the prediction performance?

To answer our RQs, we setup an experimentation that contains multiple runs of the online prediction described in Section 5.4.1. The online prediction takes two parameters: TT gap and UP. Both parameters take various values in order to see the effects of these parameters on the prediction performance. In total, the online prediction strategy is assessed using 11 different TT gap values, namely 0-month, 1-month, 2-month, 3-month, 4-month, 5-month, 6-month, 7-month, 8-month, 9-month, and 10-month, and nine different UP values, namely 1-day, 3-day, 5-day, 7-day, 10-day, 14-day, 30-day, 60-day, and 120-day. In total, we investigate a total number of 99 (11×9) variations of the online prediction model. The results are used to address all three RQs. Additionally, to answer RQ1, we used an offline prediction strategy demonstrated in Figure 5.3. This offline prediction design is a variation of the offline time-sensitive prediction approach we used during model prototyping. We include a new parameter UP to the offline time-sensitive approach during prototyping, to be able to make a fair comparison between the offline and online prediction approaches. The offline prediction strategy is also assessed using nine different UP values: 1-day, 3-day, 5-day, 7-day, 10-day, 14-day, 30-day, 60-day, and 120-day. We assessed the performance of the SDP model achieved with both prediction strategies using the model assessment methods listed in Section 5.5.1.

Particularly in RQ1, our aim is to observe whether the performance of our SDP model prototype in an offline training strategy reflects the potential performance of a deployed model. To answer this RQ, we compare the prediction performance of the proposed SDP model achieved between the online and offline prediction strategies. Both strategies start building the initial SDP model with the first four months of the development activity, then update the model through iterations based on the UP parameter, by including new commits to the training set. Finally, we report the performance achieved by both strategies, for various UP and TT gap values as heatmaps in Section 5.6.1 (Figure 5.4). We want to note that the test set of the offline and online experiments except the Online (0) may not contain the same commits. Here, our aim is not to make a one-to-one statistical comparison of the offline experiment with each online experiment. Instead, we want to understand whether the offline validation strategy could represent the real-life prediction performance.

In RQ2, we aim to observe the effect of various TT gap values on the prediction performance of the SDP model to choose a convenient length for TT gap for the deployment. We analyze the impact of various TT gap values using 1) performance heatmaps reported in Figure 5.4 and 2) Scott Knott ESD analysis reported in Figure 5.6. During the Scott Knott ESD analysis, we aggregate the performance values of online experiments over UP parameters. We only consider the identical commits across the experiments with various TT gap values to fairly observe the effect of different TT gaps. The results of these analysis methods are reported and discussed in Section 5.6.2.

In RQ3, we aim to identify the most suitable UP for deployment by analyzing the effect of the various UP values on the prediction performance. For assessment, we conduct Scott Knott ESD analysis whose findings are also reported in Figure 5.7 in Section 5.6.3. We aggregate the performance values of online experiments over all TT gaps during the Scott Knott ESD analysis.

5.5.1 Model performance assessment

We assess our SDP models using six widely-used performance measures in the SDP literature: Probability of detection (pd), probability of false alarm (pf), precision, F-measure, Area Under receiver operating characteristics Curve (AUC), and Matthew Correlation Coefficient (MCC). Pd, pf, precision, F-measure and MCC values can be calculated by using the equations given in Table 5.3 over a confusion matrix [183]. A sample confusion matrix is given in Table 5.2.

Table 5.2 : Confusion matrix for the defect prediction problem.

	Actually bug-inducing	Actually clean
Predicted as bug-inducing	tp (true positive)	fp (false positive)
Predicted as clean	fn (false negative)	tn (true negative)

Pd measures the ratio of the detected bug-inducing changes over all existing bug-inducing changes in the dataset. *Pf* measures the number of bug-inducing changes that are mistakenly classified as bug-inducing over all existing bug-inducing changes in the dataset. The pd values of SDP models are expected to be high, while the pf values are expected to be low [50]. Precision represents the ratio of actual bug-introducing changes over all changes defected as bug-introducing, whereas *F1-measure* is the

Table 5.3 : Equations for pd, pf, precision, F1-measure, and MCC.

Measures	Formula
pd	$\frac{tp}{tp+fn}$
pf	$\frac{fp}{fp+tn}$
precision	$\frac{tp}{tp+fp}$
F1-measure	$\frac{2*precision*pd}{precision+pd}$
MCC	$\frac{tp*tn-fp*fn}{\sqrt{(tp+fp)*(tp+fn)*(tn+fp)*(tn+fn)}}$

harmonic mean of precision and pd. *MCC* measures how well a predictor makes binary classification [137]. Its value ranges between 1 and -1. The value of 1 represents a perfect classification, while -1 represents a completely wrong classification. *AUC* is a measure of the area under the ROC curve, and used to judge the discrimination ability of the predictor between classifying a commit as bug-inducing and not bug-inducing [184].

Researchers pursue the ideal case (high pd and low pf) for an SDP model's performance. But in practice, reaching the ideal state is not easy. Because when correctly classified bug-inducing changes increase, the false alarms, i.e., incorrectly classified ones as bug-inducing, also increase. This trade-off has been discussed both in machine learning [185,186] and software engineering domain [13,187]. Studies state that teams would prefer a model with high pd rates with the cost of high false alarms in a safety-critical or a mission-critical software project. On the other hand, a more cost-effective SDP solution would be to have fewer false alarms with the cost of low pd rates. Considering our industrial partner's viewpoint, we focus on both pd and pf when choosing the best model prototype among all the evaluated models. Our industrial partner prefers considering the pd and pf equally during the assessment of prototypes. Since our partner has a medium level of risk tolerance, we seek prototypes with a high pd and an acceptable cost of pf rate for the final deployment.

5.6 Results and Discussion

5.6.1 RQ1: Does the performance of the proposed SDP model assessed with the offline time-sensitive prediction strategy represent that with the online time-sensitive prediction strategy?

In Figure 5.4, we report the performance of the SDP model for both prediction strategies (offline and online) as heatmaps. Each heatmap in the figure reports the performances of the SDP model measured by one of the model assessment metrics we used in our study (pd, pf, precision, F1, AUC, and MCC). The *x*-axis of a heatmap corresponds to the experiments we conducted: an offline prediction experiment and online prediction experiments with various TT gap values. The *y*-axis of a heatmap corresponds to the UP values used in both online and offline predictions. Each cell in a heatmap shows the prediction performance of the SDP model achieved with the TT gap value (only if it is an online prediction) and a UP value combination. Cells with darker grays represent higher performance values, whereas the lighter grays represent lower values of performance.

For this research question, we do not aim to discuss the best TT gap or the best UP value. We aim to observe how close we can reach to the model's offline prediction performance in real life as we change TT gap and UP parameters. Further, we do not make a one-to-one statistical comparison of performance values obtained with offline and online experiments. The structure of offline and online settings are different from each other as reported in Section 5.4.1. Accordingly, the test sets utilized by online experiments are different. Solely, the Offline and the Online (0) share the same test set. We also elaborate on how we evaluate the results for RQ1 in Section 5.5.

Heatmaps show that our SDP model's online prediction performance is initially lower than the performance achieved during the offline evaluation. However, depending on the TT gap and UP, the online prediction could achieve as good performance as the offline prediction strategy. Particularly in the offline prediction setting, the SDP model's performance ranges between 65 - 76% pd, 52 - 56% pf, 33 - 38% precision, 40 - 48% F1, 60 - 67% AUC, and 0.10 - 0.18 MCC. In the online prediction, on the

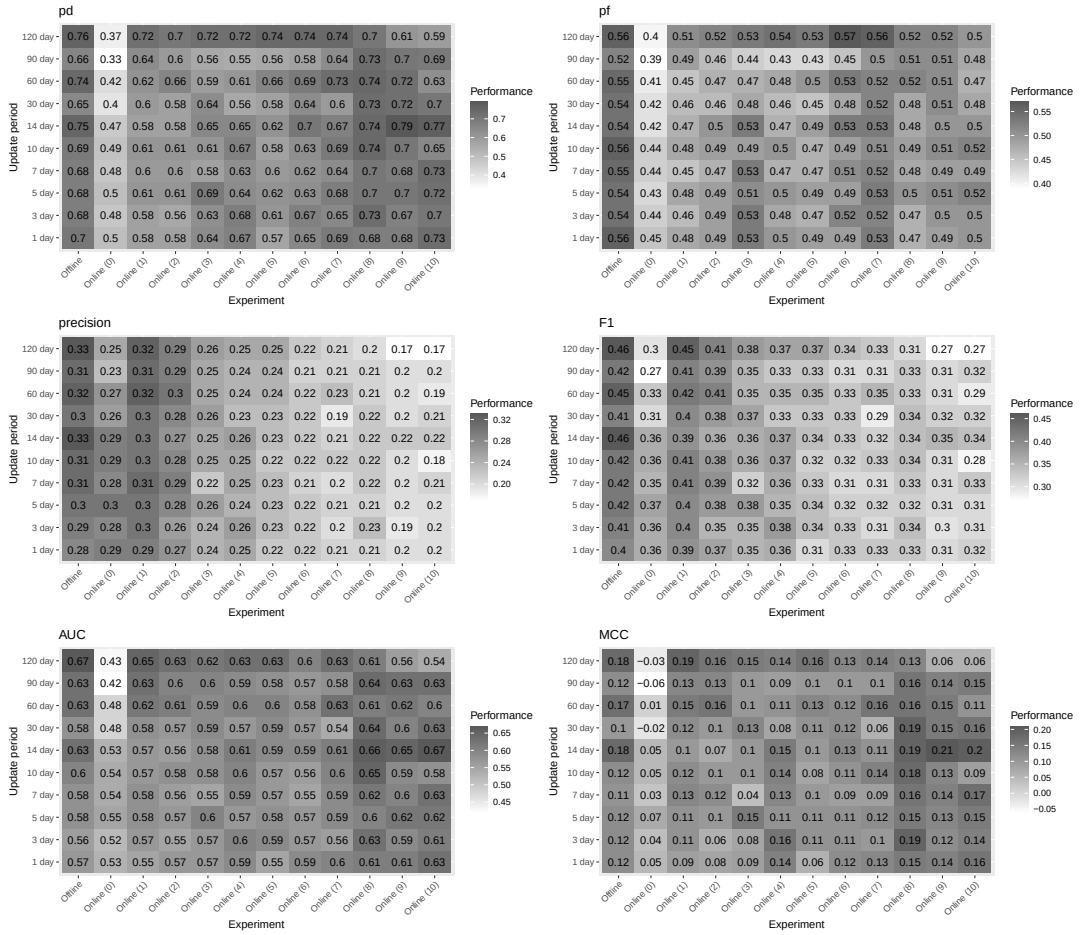

Figure 5.4 : Heatmaps of the performance achieved with the offline and online prediction strategies using different TT gap and UP.

other hand, the performance ranges between 33 - 79% pd, 39 - 57% pf, 17 - 32% precision, 27 - 45% F1, 42 - 67% AUC, and -0.06 - 0.21 MCC.

The online prediction reaches and sometimes exceeds the offline prediction performance with larger TT gaps with respect to pd, AUC, and MCC. But an opposite pattern exists with respect to precision and F1: The online prediction reaches offline prediction performance mostly with smaller TT gaps. This is due to the natural trade-off between hit rate (pd) and false alarm rate (pf) [185,187]. In the later RQs, we would choose a model with the best TT and UP gap based on this trade-off. In terms of pf, even the highest pf rates of the online setting (Online (3), Online (6), and Online (7)) are lower than the pf rates obtained with the offline setting. This is a positive finding regarding the potential review cost of the deployed model.

Regarding the UP parameter, the models, namely Offline and from Online (1) to Online (7) perform similarly. Better prediction values are achieved with larger UP values, i.e., 60-day or 120-day in terms of pd, precision, F1, AUC, and MCC, but also higher pf rates are obtained. The models, namely Online (0), from Online (8) to Online (10), achieves better performance when the SDP model is updated more frequently, i.e., every day or every 14 days.

5.6.1.1 Discussion on RQ1

We want to investigate further whether the predictions of offline and online approaches are consistent or fluctuates over time. Therefore, we additionally analyze the results by splitting the predicted commits into time frames and calculate the prediction performance for those splits. Figure 5.5 reports the models' performance over different time splits as boxplots. A boxplot represents the aggregated performance values of the corresponding time split over all UP values. The x-axis of the plots represents the beginning date of the prediction split. Each time split consists of 120 days for ease of calculation. Here, we only report the performance of the offline and some selected online experiments, namely Online (0), Online (1), Online (4), Online (8), and Online (9), to save publication space.

The performance analysis over different time periods of the project in Figure 5.5 shows that the prediction performance fluctuates over time regardless of the prediction strategy (offline or online). Often the maximum and minimum performance values are common over all experiments. For example, considering the pd values, all experiments' performance fluctuates over time and peaks around November 2015 (i.e., 94% for offline and 90% for online prediction) and March 2017 (i.e., 100% for all experiments). Considering the pf values, most of the experiments produce an increasing rate of pf over time. Considering the precision and F1 values, both offline and online models are able to reach more than 60% rates at first, while they are inconsistent with regard to the time period, e.g., from 65% to 36% to 54% in terms of F1.

Online (0) and Online (1) experiments produce low performance at the beginning of the project compared to the Offline. The reason for such lower performance with the online

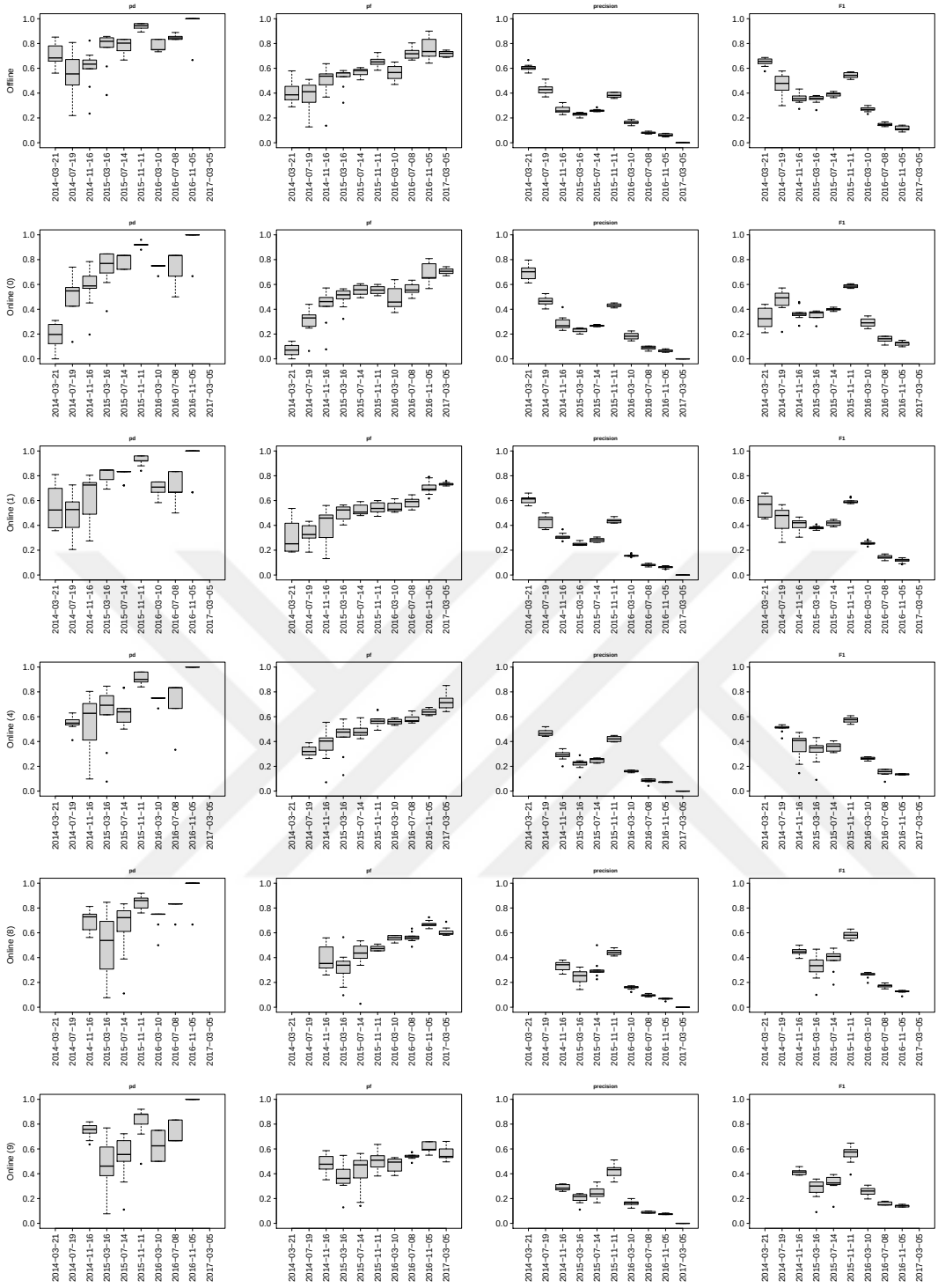

Figure 5.5 : Performance of the offline and online experiments over time.

model at the beginning of the project could be the lack of discovered bug-inducing commits in the training set. When available bug-inducing instances increase over time, the performance of the online models get better over time. Later, around November 2014, the Online (1), Online (8), and Online (9) models produce better performance

than the Offline model. The online prediction produces up to 76% pd rate with Online (9), whereas Offline produces 63% pd rate.

> The performance of the SDP model with the online prediction strategy is lower at the first year of the project due to low bug-inducing commit ratios in training. After the first year, the performance achieved with the online prediction strategy reaches and exceeds the performance achieved with the offline prediction strategy. Furthermore, adjusting the length of the TT gap and the UP parameter helps to improve the performance with the online prediction strategy.

5.6.2 RQ2: To what extent does the train-test gap affect the performance of the deployed model?

Heatmaps in Figure 5.4 show us an overall view of how the prediction performance changes with various TT gaps. From the overall results, we observe that the SDP model performs better when the models are run with larger TT gaps, i.e., 10-month, in terms of pd, AUC, and MCC, whereas the performance in terms of pf, precision and F1 is better with the smaller TT gaps, i.e., 1-month.

This picture shows the natural trade-off that occurs between the pd, precision, and pf in imbalanced datasets (the ratio of bug-inducing instances to total commits in our training sets is between 14% - 33%) where the majority of instances belongs to the clean commits. We have already mentioned this trade-off in Section 5.5.1.

The higher pd rates may cause lower precision and higher pf rates, which is not an ideally expected performance from a predictor. In practice, finding a balance between the benefits of different assessment metrics is required due to the low possibility of reaching the theoretically ideal case (high pd, low pf). Eventually, this trade-off depends on the company's choice. We assess our SDP models considering the balance between the pd, precision, and pf. We pick an SDP model that produces as good pd rates as we obtained with the offline strategy, while it does not lead to a high pf rate.

Furthermore, we observe that TT gap impacts the prediction performance of the SDP model. Adjusting the length of TT gap would improve the prediction performance by 37% in terms of pd (Online (0) vs. Online (5-6-7) with 120-day UP), 15% in terms of

F1 (Online (0) vs. Online (1) with 120-day UP), and 22% in terms of AUC (Online (0) vs. Online (1) with 120-day UP).

We statistically group the prediction performance achieved with various TT gaps by applying the Scott Knott ESD analysis to observe whether the impact of different TT gaps is statistically meaningful and which TT gap values produce the best performance. Due to different TT gaps, different commits are involved in the test sets of different online models. For example, when the TT gap parameter is set to 10-month, the commits used during the prediction are the commits done between January 2015 and May 2017. When the TT gap parameter is set to 1-month, the commits used during the prediction are the commits done between April 2014 and May 2017. Therefore, to make a fair comparison of the effect of various TT gap values on the prediction performance, we analyze their predictions only on the commits at the intersection of all test sets.

The Scott Knott ESD analysis is conducted in terms of pd, pf, precision, F1, AUC, and MCC, and reported in six plots in Figure 5.6. While the *x*-axis of each plot represents the TT gap values (zero (0) to 10 months), the *y*-axis of each plot represents the mean performance values of the SDP model. The length of the vertical lines represents the distribution of the performance values over various UP values, while the dot on each line represents the mean value of the performance. Colors of the lines represent the statistical grouping among the performance gathered with various TT gaps: if two or more TT gap lines are the same color, that means there is not any statistical difference among the performance gathered with those TT gap values.

Colors in Figure 5.6 show that there are at least six statistically different groups, e.g., in terms of F1, among the online models regarding the TT gap. The statistically best performing SDP model is achieved when we keep a 1-month time gap between the training and test sets, in terms of pd, F1, AUC, and MCC. However, our SDP model also reaches its highest pf rate with the 1-month TT gap. Besides, the 8-month TT gap produces a lower pf rate and the best precision. In terms of F1, AUC, and MCC the 8-month TT gap is among the top three best performing models. Although the 8-month does not give the best pd (78%), considering the trade-off between pd and

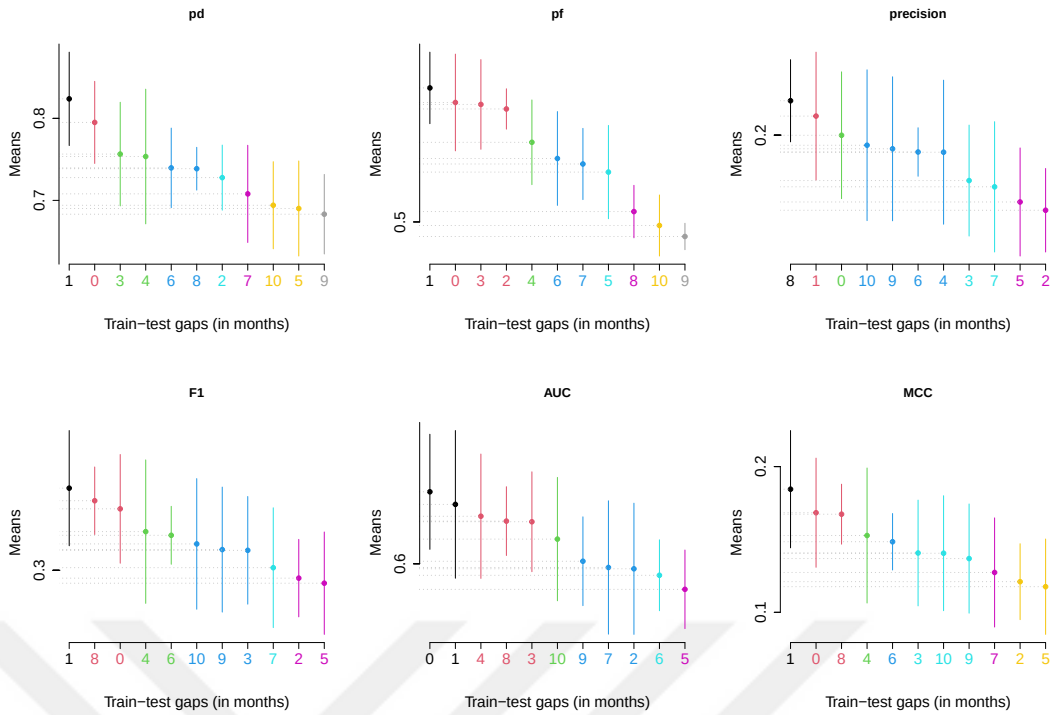

Figure 5.6 : Scott Knott ESD analysis of performance with different time gaps between train and test sets.

other measures, 8-month is decided as the most convenient option for this industrial context.

> Keeping a TT gap affects the prediction performance of the SDP model differently. Some TT gap values degrade the performance, e.g., 5-months in terms of pd, precision, F1, AUC, and MCC, whereas others improve the performance, e.g., 8-month in terms of precision. Considering the performance values and the trade-offs between model assessment measurements, we set the TT gap to 8-month for the deployed SDP solution.

5.6.3 RQ3: What impact has the update period of the proposed model on the prediction performance?

Performance values reported in heatmaps (Figure 5.4) show that the effect of the UP value on the prediction performance would be 18% in terms of pd, i.e., Online (9) produce 79% pd rate when UP is set to 14-day and 61% when UP is set to 120-day.

Figure 5.7 shows the Scott Knott ESD analysis for all the UP values. Similar to the Scott Knott ESD analysis in RQ2, each plot in the figure shows the prediction performance measured by a different performance assessment metric, i.e., pd. The

x-axis of each plot represents the UP values, whereas the *y*-axis of each plot represents the prediction performances of the proposed SDP model gathered by setting the UP to a particular value. Please note that the prediction performance with each UP is aggregated over all the TT gap values.

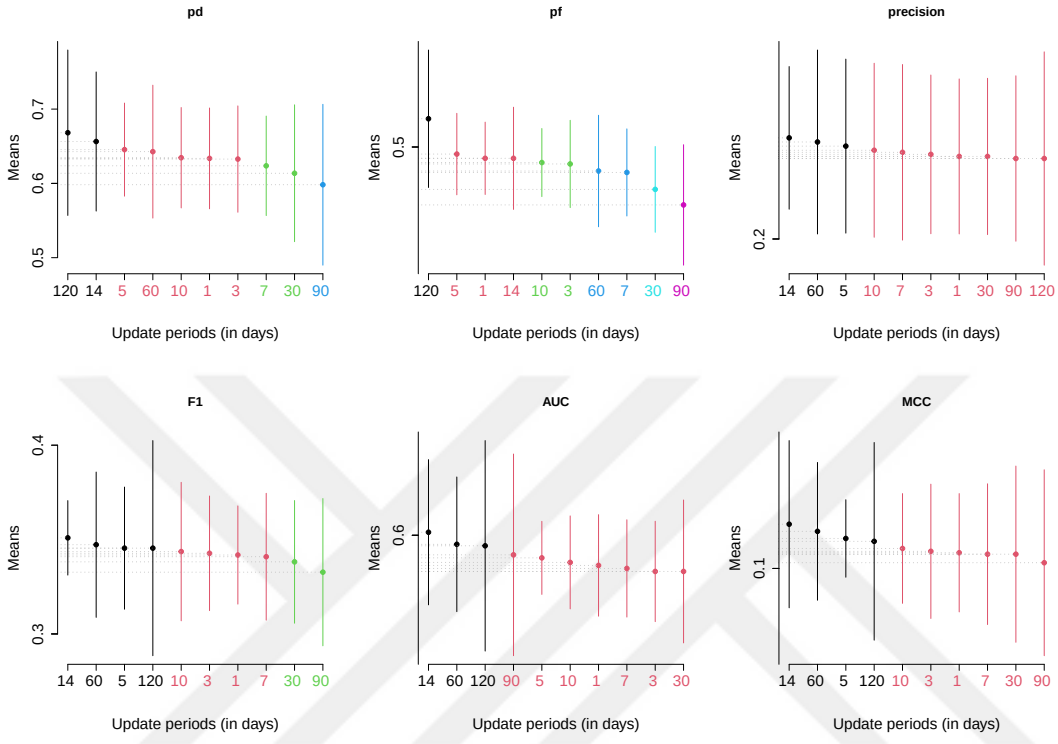

Figure 5.7 : Scott Knott ESD analysis of performance using different update frequencies.

The statistical groupings in Figure 5.7 show that the impact of the most UP values are statistically different from each other in terms of pd, pf, and F1. On the contrary, most of the UP values have the same impact on the prediction performance according to the precision, AUC, and MCC results, i.e., updating the model every ten, seven, three, one, 30, 90, or 120 days yields statistically the same performance in terms of precision. Furthermore, setting the UP parameter to 14-day, 60-day, and 120-day provides quite promising prediction results compared to the other UP values. Among those best performing UP setups (14-day, 60-day, and 120-day), 14-day seems a better choice than others since it is statistically one of the best performing model in five out of six model assessment measures (66% pd, 23% precision, 35% F1, 60% AUC, 0.15 MCC). At the same time, it provides a relatively lower pf rate (48%).

5.6.3.1 Discussion on RQ3

Before setting a decision on UP for the deployment, we assess the effect of the UP parameter on the prediction performance considering only the Online (8) model, since we chose to set the TT gap as 8-month for the deployed model according to the results of RQ2 (Section 5.6.2). When we consider the performance of the Online (8) model in Figure 5.4, we observe that there are many UP values that give quite promising results. In particular, setting UP to 3-day or 14-day reports similar performance with pf of 47-48%, precision of 23-22%, F1 of 34%, AUC 63-66%, and MCC of 0.19. However, 3-day UP gives significantly the best pd (81%) compared to 14-day (78%) according to additional Scott Knott tests. When we also assess the performance of Online(8) with UP values of 3-day and 14-day over different time splits, as we did for the TT gap shown in Figure 5.5, we observe that 3-day has a better performance in the majority of the splits. Considering the pd values corresponding to March 2015 split, while 14-day UP produces 54% pd rate, 3-day UP produces 85% pd rate. Therefore, updating the deployed model every three days is a more convenient choice when considering UP and TT gaps.

> Update period significantly affects the prediction performance of the online SDP model. Adjusting the UP parameter provides a gain improvement of 18% in terms of pd. Setting the UP parameter to 14-day leads to one of the best performing models in terms of pd, precision, F1, AUC, and MCC. However, when combined with our online model with an 8-month TT gap (in RQ2), 3-day is the best UP value in terms of pd rates. Therefore, we decide to update the deployed model every three days.

5.6.4 Effort-aware performance assessment

Effort-aware performance evaluation is another popular method to assess the performance of change-level SDP models in the field [16,130,132]. The effort-aware assessment basically reflects the total effort spent on code inspection during the code review activities.

We conducted an effort-aware assessment of our prediction model using a similar calculation to prior change-level SDP studies [16,130,132]. Effort-aware assessment produces a value representing the percentage of total bug-inducing changes that could

be detected by examining only 20% of the total lines of code (LOC) during the code review. Accordingly, we first calculate the benefit-cost ratio for each commit in the test set using the formula of $P(c)/Effort(c)$. $P(c)$ is the probability, between 0 and 1, which represents the bug proneness of a commit c. $Effort(c)$ is the total number of changed lines (added and deleted) in the commit c. Second, we rank the commits in the test set based on their calculated benefit-cost ratio values in descending order. Third, we count the number of bug-inducing commits that could be detected when only 20% of the total effort is spent on code inspection (when the commits are inspected in descending order of their benefit-cost ratio).

The effort-aware assessment is made for each offline and online experiment with various TT gap and UP values. Figure 5.8 reports the effort-aware performance values as a heatmap. The *x*-axis corresponds to the different experiments with various TT gap values. The *y*-axis shows the UP values used in the experiments. Values in the heatmap cells indicate the percentage of the total bug-inducing changes in the software could be detected by examining only 20% of the total LOC for the corresponding experiment.

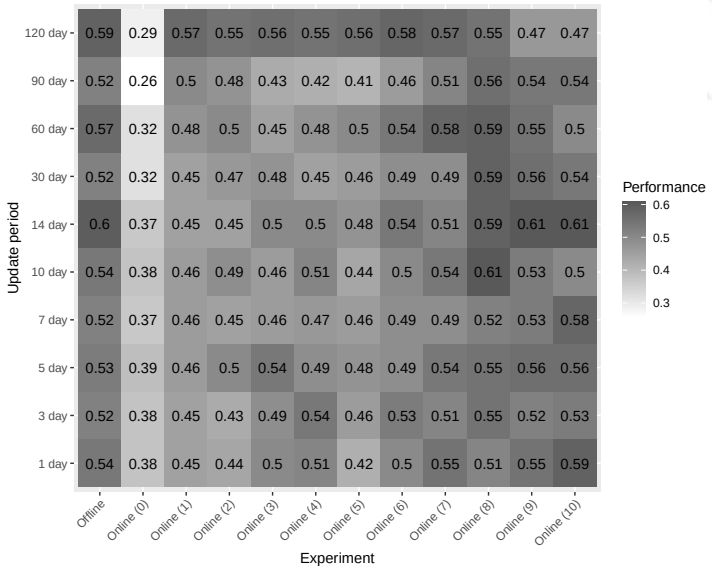

Figure 5.8 : Effort-aware performance.

Values reported for the offline prediction in Figure 5.8 are between 58% and 64%. While the values reported for online prediction are between 30% and 67%. Similar to the results reported in Figure 5.4, Figure 5.8 also shows that the online prediction with

lower values of TT gap (i.e., Online (0)) produces lower performance, while keeping higher TT gaps (i.e., Online (8)) leads to improvement in prediction performance.

Furthermore, the values we pick for TT gap (i.e., 8-month) and UP (i.e., 3-day) for the deployment is one of the best options according to the effort-aware assessment. We are able to detect 55% of the bug-inducing changes with the Online (8) experiment and 3-day UP. Also, a Scott Knott ESD analysis of the effort-aware assessment results supports the findings obtained with the other assessment metrics reported in Sections 5.6.2 and 5.6.3. While 8-month becomes the second best TT gap statistically, 3-day is the third best one among UP values. Please note that the overall statistical analysis is not our only consideration for deployment decisions. We also consider the consistency of prediction performance over time for picking a UP value, as we discussed in Section 5.6.3.1.

5.6.5 Sampling for class imbalance problem

SDP studies often suffer from the imbalanced nature of the collected datasets (e.g., [16,80,100]). Usually, the number of buggy data instances, i.e., commits, is much lower than the number of clean instances. This situation may prevent building an SDP model that successfully learns to classify future buggy instances with a good performance.

Our collected dataset for this study also has an imbalanced nature. Accordingly, the defected commits constitute 14% to 33% of the total commits in training sets. Due to the imbalanced nature of our data, we want to observe if we could obtain better prediction performance values if we balance the ratio of commits in our training sets. We would like to note that, inspired by a recent study [168], we analyze the evolution of the imbalanced ratio of training sets over time. Our longitudinal analysis shows that after April 2014, the imbalanced ratio of the training sets is fixed to the range of 28% - 33%. Accordingly, we apply two data sampling techniques that are commonly used for improving the SDP performance: under-sampling and SMOTE (Synthetic Minority Oversampling Technique (SMOTE) [145]). Under-sampling randomly removes clean commits until the sample sizes of clean and bug-inducing commits are equal. On the other hand, SMOTE creates synthetic bug-inducing commits while also randomly

removing the clean ones. We report our prediction performance values obtained with data sampling in our online Appendix [188].

The prediction results with the balanced training sets show that applying data sampling affects our SDP model's prediction performance inconclusively for some TT gap and UP values. Data sampling does not provide an obvious advantage over no sampling in our study. Also, none of the two sampling techniques increases the performance for our deployed model. For these reasons, we did not change our sampling strategy for our deployed setup. Particularly, under-sampling increases the pd rates by up to 48% of SDP models trained with larger UPs, i.e., 90-day and 120-day. On the other hand, under-sampling also causes a decrease, up to 26%, in the pd rates of some models with small UPs. The cases with larger UP values have more imbalanced training data compared to other cases with smaller UPs due to the lack of updating training data for a longer time. Therefore, applying under-sampling on highly imbalanced training sets improves the pd rates in our study. Nevertheless, under-sampling significantly increases pf rates up to 65%, and these rates are not applicable in practice. Since under-sampling increases both correct and wrong classification rates, prediction performance according to other measurements, i.e., F1, does not change much.

On the other hand, SDP models trained by applying SMOTE produce lower pf rates, i.e., 22%, which is the desired situation. But, using SMOTE during training is also not our choice due to decreasing performance values in pd and F1. SDP models with SMOTE produce up to 30% lower pd rates compared to models with no sampling.

5.7 Threats to Validity

Internal validity: During our data collection, we ensure that we correctly identified the bug-inducing commits of the pilot project and cover all of them. We apply the state-of-the-art methodology (SZZ) for determining the bug-inducing commits using the issues only reported in 'bug' category in JIRA. Besides, the team lead at the industrial side manually examined the commit messages to capture bug-fixing commits that were not linked to any bug issue. Moreover, we filtered out the non-development software changes from the collected dataset to ensure the data quality.

Construct validity: The proposed SDP model is built using the state-of-the-art process metrics, which are listed in Table 5.1 and their latent features of them. Those process metrics are widely used in SDP studies, accepted as good indicators of bugs at change-level prediction models (i.e., [16,27]) and quantifies the software changes from various aspects; size, history, diffusion, and experience of the committer. This metric set was also assessed by the research team of our industrial partner in order to adjust the metric set according to the structure of the pilot project. Moreover, we included the latent features of the process metrics into the model building process in order to capture potentially hidden, underlying information that the base metrics set may not capture. Latent features are extracted using Non-negative Matrix Factorization (NNMF) [160]. NNMF is used to learn low-rank approximations of the given data, and it is a well-known technique used for pattern recognition tasks [156] and building recommendation systems [157]. SDP models reported in this study are built with the NB algorithm, which is widely applied in many SDP studies and proved to be successful [50]. Prototype SDP models built with NB also perform better than the prototypes built with the RF and XGBoost algorithms.

Bug-inducing software changes are detected with the SZZ algorithm, which is the state-of-the-art way to detect the bug-inducing changes of software. We implemented the original version of SZZ [116]. SZZ detects buggy changes using a heuristic approach, and hence it may cause mislabeled changes. Various implementations of SZZ are used by different researchers [149]. However, according to Fan et al. [151], the original SZZ [116] does not yield a significant performance reduction in change-level SDP due to the noise in the change labels. Nugroho et al. [189] also investigates two different *git diff* commands, i.e., *Myers* and *Histogram* in SZZ and discuss that the type of *diff* command might have an impact on the results of SZZ. We use the default *git diff* command *Myers*. Moreover, we ensured the data quality by asking the team at the industrial side to go through all the commits manually. It is our future research direction to analyze different *diff* commands.

External validity: We use the online prediction strategy to assess the design decisions required for the deployment of the SDP model. The deployment decisions we made in our study are specific to the characteristics of the pilot project selected in this

work. Thus, the findings cannot be generalized to other software projects, but the methodology can be transferred to other settings when deploying an SDP prototype.

Conclusion validity: For comparing different models' performance in Sections 5.6.2 and 5.6.3, we applied the Scott-Knott Effect Size Difference (ESD) test implemented by Tantithamthavorn et al. [82]. This test provides a statistical comparison among the performance achieved by different variations of the online prediction models. We made sure the test sets are the same across different models' assessments, and report according to the statistically significant differences only. Moreover, we chose well-known performance assessment measures widely used in SDP literature: pd, pf, precision, F1, AUC, and MCC [105].

5.8 Lessons Learned

During our research, we learn that deploying a change-level SDP model into an industrial context requires rethinking and redesigning the traditional techniques that have so far been used in the SDP research. Besides, communication and understanding the industrial perspective are keys of proposing and deploying an SDP solution to an industrial team. In this section, we share our learned lessons from practicing a change-level SDP model in an industrial context. Key points to take away from our experience and key suggestions to build future change-level SDP models in our industrial context are summarized in sub-sections.

5.8.1 Mind the difference between industrial and academic views

The perspectives of industrial and academic teams might be different from each other in research partnerships [48]. Even though the software engineering research field mostly focuses on practical problems, it is not always easy to look at the problem from an industrial/practical perspective by academic researchers and vice versa. Here, we share our experience that would help other researchers while practicing a change-level SDP model in an industrial setting.

Deploying a consistently successful, user-friendly, and trustable SDP model is not a smooth task. It requires the involvement of the industrial side during different stages of the partnership, i.e., data collection to deployment, as situations might occur

where context-specific information is needed, such as project and team development structure. For instance, during the prototype assessment, the expectations from a prediction model regarding its performance could be different between academia and industry. While academia tends to value the prototypes that produce higher values of performance values (i.e., pd), the industry might consider many other factors before having a decision on a prototype. Instead of solely valuing the prototypes that produce higher performance values, they also consider the time/cost of the model learning. For instance, using a lightweight machine learning technique during the model training and updating is very crucial from the industrial perspective. Also, as we report in Section 5.5.1, the trade-off between pd and pf rates is also considered carefully by the industry. Practitioners would prefer a model that does not produce the best pd rate but have acceptable false alarm levels.

The usability and interpretability of the prediction output are other factors that academia and industry may have different perspectives. A researcher may find it more appealing for the integrated SDP solution to list the prediction results from multiple SDP prototypes, as she focuses on the continuous improvement of the SDP model. However, in such a case, a practitioner in the development team would be confused by seeing multiple prediction results for his/her changes. Therefore, for real-life practice, synchronization of both perspectives is crucial.

The deployed prediction model should also "keep up with the fast pace of development" [2]. Training a predictor and integrating it into the development cycle may not lead to consistent prediction performance. The need for future updates in the deployed solution may occur. Therefore, deployment is not a one-step task. It requires multiple rounds of negotiations between academic and industrial teams to see the needs of practitioners, and decide on the technical details. Active communication is crucial for teams to be aware of the expectations of both sides. Regular meetings and demonstrations of the current work as it matures are essential to be on the same page. Those meetings allow us to receive the practitioners' feedback on the functionalities of the deployed solution, as well as helping practitioners to picture the capabilities and limits of the SDP solution.

Moreover, the cooperated company's interest in the research and development activities, and the interest and enthusiasm of team members on recent research trends are other key factors that improve the communication and cooperation between the two sides. More specifically, allocating resources such as equipment, working space, and team members to support academics while working in the deployment are substantial.

5.8.2 Building an SDP solution requires full commitment of the industrial partner

During our research project, we faced several challenges from data collection to deployment. Integrating a stable and sustainable SDP solution into practice occasionally requires manual interventions and insights from experienced developers of the software project.

The team at the industrial side helped our research during data collection by reviewing the collected commits to assure the data quality. More particularly, they manually marked the bug-fixing commits that cannot be linked automatically to the bug reports collected from JIRA, checked SZZ results, and matched the multiple user accounts used by a single developer during the development. We decided to filter bulk, merge, and test module-related commits with the discussions at the meetings with the industrial side. Another benefit of holding regular project meetings is that the expertise of the developer team on the project provides us the early notice of the situations that may cause problems. For example, at the very beginning of the research, we realized that we collected the commits from some branches of the project instead of all branches. Moreover, some project-specific challenges cannot be overcome without the industrial team's expertise, i.e., interpreting empty file names or file references to external projects in the commit logs, accessing missing bug reports, matching multiple usernames belongs to the same user, and differentiating the non-development commits. Therefore, we think the development team of the software project, which is going to use the SDP model, must be committed to be actively involved in every step of the research project in order to monitor and give feedback to the research team. Otherwise, the model built and assessed may resemble a prototype rather than a real recommender system for the benefit of the developers.

5.8.3 Context specific factors impact the performance of the deployed SDP model

The deployment related findings that we report in this study are specific to the chosen pilot project's context. Although our experience would be insightful for other practitioners, each context requires its own analysis and consideration of the team's development practices.

The deployed SDP solution with a TT gap of 8 months and a UP of 3 days achieves around 81% pd rate and 47% pf rate. However, the deployment design decisions has a varying effect on the prediction performance, as our online models show. Avoiding the model learning from the noisy (undiscovered bug-inducing commits) data (with a TT gap) would improve the pd rate by 37%, while updating the prediction model with recent development activity (with UP) affects the pd rate by 18% (see Figure 5.4).

We observe that not only the TT gap and UP parameter, but also other project characteristics significantly change the performance of an SDP model. Considering the project's development intensity over time, i.e., number of daily commits, number of active contributors at any time, or the average time required for fixing bugs would be helpful to set up a successful SDP model in practice for a software project. In fact, the characteristic of a software project follow different patterns at different times [133]. Therefore, the factors such as bug life cycle characteristics and the development activity patterns also cause performance fluctuations over time splits. Our empirical analysis shows that those fluctuations exist regardless of whether an online or offline prediction is used (in Figure 5.5). For example, during 2014 - 2015 there are bulks of bug-inducing and clean commits in the dataset (in Figure 5.1) because of active development and testing process due to the initial productization. It is possible that these bulk data cause the performance fluctuation of the SDP model. Such active development and testing periods are inevitable since the team still receives large requests from customers and increases the development activity by also increasing the number of developers involved in the team. So, future fluctuations are also highly possible in deployed model's prediction performance. Therefore, the time gaps, update frequencies, development activity and bug-fixing and bug-introducing patterns, and

more contextual factors should also be considered in building a deployed model in any industrial context.

5.8.4 Focus on user-centric evaluations

Until deployment, the prediction performance of the SDP models is evaluated with respect to the correctness aspect [40], namely pd, pf, precision, F-measure, and MCC. Besides evaluating the SDP models with those correctness measures, it is also critical to evaluate a model from a user-centric point of view, i.e., the representation of the prediction results should provide trustworthiness to the developers [40]. Previous research also indicates the importance of producing the actionable prediction outputs in order to gain the user's trust in defect prediction solutions [29].

The proposed SDP solution is integrated into the development pipeline before the code review process. The development team has a reviewer group that often involves two or three developers. The essential aim of the deployed SDP solution is to assist the reviewers during the code review process. The system, therefore, provides a push-based notification after every commit. After a developer commits her changes, the prediction model is triggered and produces a prediction output for her commit. The code reviewers often use that feedback on the commits to assist their review process.

The leaders of the development team remark that they would prefer to see a bug risk probability on the commits instead of giving a binary output that indicates the committed changes are buggy or clean. Hence, we also provide the bug-proneness of a commit with a probability value of the prediction output. In this way, the defect predictor would turn into a system that supports developers during the code review activity by giving soft feedback on the bug-proneness of their code changes.

Furthermore, the development team leaders also state the reviewer quality might affect the usability of the prediction outputs. For example, the SDP model produces a bug-prone commit alarm. But the reviewer might not be able to interpret the SDP model's prediction, i.e., she may be a newcomer and has not much expertise on the project, which makes it harder to interpret the prediction outputs for the code review process. Therefore, we focus on improving the interpretability of the prediction

output to provide more actionable predictions. We enrich the prediction output with by-products, e.g., process metrics calculated for the software changes. For the development team, these by-products are considered as useful as the defect prediction result itself. Since our SDP solution measures a software change from size, diffusion, historical, and experience of the committer aspects, enhancing the prediction report by including those metrics' values leads developers to a better understanding of the characteristics of the bug-prone change.

We integrated all types of SDP approaches (i.e., general model, personalized model, and selected model) proposed during the prototyping phase (Section 5.3.2) into the development environment. During the early stages of the deployment, while testing the deployed SDP models, the team states that it is hard to examine the multiple prediction results gathered from different SDP models, especially when any two models' predictions contradict with each other. Accordingly, the team prefers to see only one SDP model's prediction results, which is the general model's prediction, and others' outputs are removed from the feedback.

As future work, the software team lead also wants to see an information dashboard that shows the core developers who contribute the most to the project, historical changes' and their bug-inducing probabilities, and the distributions of the process metrics over these historical changes. The usefulness of the SDP solution would be higher if this additional information is also provided to the developers.

5.8.5 Trust the algorithm choice in the offline setting

In our research project, we initially built prototypes utilizing various machine learning algorithms: Naive Bayes (NB), Extreme Gradient Boosting (XGBoost), Random Forest (RF) and Logistic Regression (LR). After assessing the built prototypes using the offline model validation strategies (cross-validation and offline time-sensitive), we picked the best performing classifier to use in the deployment phase. We use NB during deployment since it provides 70% pd rate, while other classifiers provide pd rate around 50%. Still, we could ask whether another classifier may perform better in the online prediction setting than NB.

Our online prediction experiments using XGBoost classifier show that XGBoost leads our SDP model to perform similarly both in offline and online settings, which is still lower than the performance values achieved with NB. Therefore, we suggest SDP practitioners who want to decide between classifiers assess the classifiers' performance with an offline evaluation and later stick to the best performing one in the online scenario. It would be more practical to assess various classifiers using a cross-validation approach rather than evaluating them in a more complex online prediction design.

5.8.6 Deployment is a continuous development

Machine learning models also require maintenance and continuous development, as well as a typical software product. It is not much possible to receive successful outputs from the initially deployed prediction models forever because the development activity pattern of the project may change. To sustain the stability and accuracy of the SDP solution, an examination may be required periodically. For example, we decide on a set of deployment design decisions for now, i.e., TT gap and UP. But those design decisions might require reconsideration in the future due to the potential change of development patterns over time. Moreover, other factors of the prediction model may also need to be reconsidered in the future, such as process metrics (Table 5.1). For example, the committer's recent experience (REXP) is calculated considering the recent three months of development history. The calculation rule of the REXP metric was decided by the team lead intuitively, considering the average contribution history of developers. However, in the future, depending on the developer contribution patterns, its calculation rule may require a re-consideration. Besides, we utilize all the historical development activity from the beginning of the project until the TT gap. However, the number of commits in the project's history may be too many in the future and may prevent the SDP model update itself in a considerable amount of time. In fact, using all historical commits may cause a performance drop in the future, since the characteristics of the recent commits may differentiate from the oldest commits [167].

Furthermore, we integrate additional SDP models to obtain a more successful SDP solution in the long run. During the prototyping phase, we propose three different

SDP approaches (Section 5.3.2) and pick the best performing one (GM) to deploy in the real development environment while the other two models (SM and PM) are still in the test phase. GM is more robust and provides more accurate results. The development team wanted a consistent prediction model but also wanted to integrate the other models into a test branch of the development environment since the other models also have promising results, i.e., the personalized model performs quite well for some developers. Hence, we integrated the SM and PM as beta models, as well as integrated the mature one, GM, as the default model. As future work, we plan prediction outputs of the SM and PM will be optional that can be on/off depending on the choice of the code reviewer.

5.9 Conclusion

Applying software defect prediction (SDP) in practice is a challenging process. Producing accurate, consistent, and useful predictions depends on many aspects such as data availability, characteristics of the project, the development team habits, and representation of the prediction output. This paper reports our experience in deploying a change-level SDP model into our industrial partner's software project. During deployment, we investigate two factors that impact the prediction performance of the deployed model: 1) The noise in training data due to the undiscovered, i.e., mislabeled, bug-inducing commits at the prediction model training time and 2) The model update (re-train) period with new commits. We empirically analyze the impacts of those factors on the prediction performance using an online prediction strategy that simulates the real-life development life cycle, considering the actual labels of the training instances at the model training time, filtering the time period from the training set that potentially has noise, and updates the model periodically with new commits.

Our empirical analysis shows that keeping an unused time period between the training and test commits impacts the prediction performance by 37% in terms of probability of detection (pd). On the other hand, the update cycle of the SDP model impacts the pd rate by 18%. We pick the best convenient combination of the investigated factors considering the accuracy and the consistency of prediction performance for deployment. Eventually, the deployed solution updates the predictor every three days

by utilizing the commits done until eight months prior to model re-training time and produces 73% pd and 47% pf rates. Furthermore, we observe that the SDP prototype built with an offline prediction strategy represents the real-life prediction performance of the predictor depending on deployment settings and the stage of the project.

We further elaborate on the lessons learned through this deployment process, considering the academic and industry perspectives, research and real-life environments, and the interpretability, accuracy, trust aspects of a recommender system. We would like to suggest to researchers who aim to practice change-level SDP in an industrial context that it is crucial to understand the team and project environment for designing a better prediction solution. Analyzing the application domain, i.e., the commit release cycle of the project, development habits of the team, practitioners' expectations from the SDP model are also crucial for a consistently good deployment performance. Maintaining transparent communication with the practitioners on the technical details, i.e., such as classification algorithm and metrics used during the model training, and representation of prediction outputs, leads to a satisfactory deployment. Also, from a technical point of view, building a change-level SDP in an industrial context needs to be analyzed in detail regarding the bug-fixing durations, labeling the data, and re-training the model with new commit data.

As future work, we continue working with our industrial partner to address their needs regarding the visualizations of the model outputs and byproducts, model usefulness, and increase the trustworthiness of our model in developers' eyes. In particular, we would like to focus on the fluctuations of the model performance. A better understanding of the reasons for fluctuations would help us to improve the accuracy of the predictions. Furthermore, an adaptive SDP model that has an awareness of the changing characteristics of the development life-cycle (i.e., number of daily changes) and bug life-cycle (i.e., the time required to fix a bug) are other future directions of ours. Besides, our assessment of recent de-noising techniques [190] to defeat the effect of noisy labels on the prediction performance yields promising results. Hence, adopting de-noising methods as an alternative to adjusting the TT gap is another plan.

6. FINAL REMARKS

In this thesis, we empirically investigate the people factors in SDP and the applicability of SDP models into an industrial setting. Thesis studies consist of four parts.

Specifically, in the first part (Chapter 2), our aim is to explore the effect of community smells on the prediction of defects in software systems. Our findings on ten open-source projects show that incorporating the community smell patterns of the team into the model training phase contributes to the prediction of defect-prone software classes. Performance values of base SDP models that utilize the state-of-the-art metrics, i.e., static code metrics, the number of developers touched a class, code churn, and the developer scattering. However, the other smell related metric, i.e., the code smell intensity, which is found to occur together with the community smells, improves the base models' performance values more than the community smells. The higher contribution of code smell intensity metric on the prediction of defective classes and the higher calculation cost of community smells (i.e., requires communication archives of the team) leads us to other ways of people modeling. Therefore, we direct our research towards the personalized SDP approach, which does not require mining communication channels such as mailing archives. Instead, personalized models are built by utilizing metrics measured from the source code repositories that represent the development skills of people, like code smell intensity metrics.

In the second part of the thesis (Chapter 3), we conduct an extensive investigation of personalized approach SDP at change-level in six open-source projects. Our analysis shows that personalized models are quite promising since they outperform the general models for the majority of developers. However, there are factors that affect the choice of personalized over general models, such as performance evaluation criteria, model building algorithms, and development characteristics.

The third part of the thesis (Chapter 4) also reports the investigation of change-level personalized models but in an industrial context. Analysis in the industrial context

shows that a base personalized approach that utilizes only the state-of-the-art process metrics is not superior to the general approach. However, personalized models could reach the performance of general models when they utilize latents of process metrics, topic models of commit messages, and log filtering. Although the prediction performance values of personalized models reach quite well performances in the offline research environment, deployment of personalized models into the real development environment could not be realized due to the high turnover rates of the development team.

The fourth part of the thesis (Chapter 5) focuses on the deployment of the general SDP model proposed in Chapter 4 that utilizes the process and latent metrics, and the log filtering approach. The deployment challenges regarding the differences between offline and online environments lead us to completely redesign our model evaluation approach to simulate the real development data flow. Our online model evaluation shows that the model performance in an offline environment could represent the real-life performance after the first year of the project. On the other hand, deployment decisions such as keeping a time gap between the model train and prediction sets to prevent learning the model from noisy data and the model update cycle have an effect on the prediction performance. Moreover, we observe the importance of factors that affect the trustability, usability, actionability of the prediction outputs.

6.1 Contributions of the Thesis

The people effect in SDP has been studied little compared to other aspects of prediction models such as algorithms and data. Therefore, this thesis makes a rich contribution to the field with a comprehensive investigation of people factors in SDP with a major focus on personalized SDP models.

As far as we know, our initial analysis (Chapter 2) regarding socio-technical factors and the predictability of bugs is the first research that incorporates the community smell patterns into the SDP. The findings of this study provide insights to researchers and practitioners on the predictive contribution of community smells in SDP compared to the contribution of other state-of-the-art metrics. Also, another significant output of this research is a publicly available SDP dataset [76] that includes community smell

patterns of software classes as well as including other state-of-the-art metrics such as code smell intensity and static code metrics. Moreover, we discuss the challenges of the community smell detection process through the CodeFace4Smells tool [55] which mines the software code repository and the mail archives of the organization. That also would lead the community in their decisions regarding practicing the community smells in their recommendation systems.

Furthermore, our research on the personalized SDP, reported in Chapter 3 and 4, is very comprehensive compared to existing personalized SDP studies. Our empirical setup includes a larger context compared to prior studies, i.e., 222 developers from open-source and six developers from an industrial project, while prior studies analyze only ten developers from open-source projects.Also, that means our study is the first one that investigates the personalized SDP in an industrial context. In addition to validating prior findings by demonstrating that the personalized SDP could be successful over the traditional approaches, we also explore the development characteristics that lead to the superiority of personalized models. Our analysis shows that the improvement of SDP models with various data sources and statistical methods can also support researchers and practitioners in situations where the available data is scarce and noisy.

SDP studies that focus on practical applicability and/or industrial cases are relatively few than those that focus on offline studies and/or open-source context. This thesis does not just evaluate the personalized approach in an industrial context but also reports the deployment experiences on an SDP solution designed for our industrial partner (Chapter 5). Since the online, i.e., real life, software data includes noise due to existence of bugs [29] and the deployed models entail re-training due to non-stationary nature of software [191], we point out the impact of this noise and the model update cycle on the real-life prediction performance using a rigorous online prediction design. Further, we share our insights on the actionability and the explainability of the prediction outputs and our industrial collaboration experience. Our research would guide SDP practitioners in managing their deployment decisions and researchers in performing research projects with the industry.

6.2 Recommendations and Feature Work

Incorporating people, which is one of the important aspects of software development, into SDP models is rewarding in defect detection performance depending on the selected ML algorithm or developer's development characteristics. While we are suggesting practitioners to consider the personalized approach in their settings to obtain higher detection rates, this field also needs more research regarding the underlying factors that may affect the performance and applicability of personalized SDP models.

One other future research direction for personalized models is profiling developers in a transferable way. The current state-of-the-art personalized SDP models utilize only the developer's development activity in a single project that the SDP models are built for. However, such an approach require training of a model for each software project that the developer contributes. Instead of mining project-specific developer attributes through their development history, creating a more general and transferable profiles for developers would be in the future of the personalized SDP approaches. Collecting developer data using their public software development profiles (like Github) or mining their activities from open-source projects that they contribute would be some options to create developer profiles. Building transferable developer profiles would overcome the cold-start problem which occurs when a new developer is joined to the team [5]. Besides utilizing public activities of developers, cross-project personalized SDP models would be an in-company solution in such a case when a developer joins a new project within the company.

Another future direction is to target groups of developers instead of targeting individuals as in the personal approach, or the whole team as in the traditional approach. Prior studies [36,53] and also this thesis (Section 3.5.5) already show the success of collective personalized models that are built for a developer by learning from other developers' development activities in addition to the developer's own history. Besides, segmentation of customers according to their profiles by measuring their interests, behaviors, geographic, demographic, and psychological data, and having specific strategies to those segments are widely used applications in data analytics

systems to gain business advantages [192,193]. The SDP field also would benefit from the segmentation of developers according to their profiles to gain more accurate and satisfying prediction results. There are different ways of grouping developers that are open to research. For example, groups can be shaped according to the development activity characteristics, e.g., developers' experience in the project, or by patterns of how developers' different characteristics contribute to defect estimation.

As the software engineering community also state [194], domain analysis is very crucial in the software data analytics. Therefore, for future studies, especially the ones targeting practical application of software recommendation models, we recommend validating prototypes using online prediction setups and examining the project's context, team's expectations, perspective, and practices to comprehend the factors that affect the model's performance and usability. For example, during conducting data mining and ML activities to build an SDP model, discussing the success criteria of the organization is important, i.e., false alarms are more tolerable in mission-critical systems [57]. Also, understanding the developers' practices, perceptions, and the continuous development structure of the organization is important to model useful and proactive recommendation systems [2]. For example, the developers with different roles or experiences have different preferences on using an SDP tool (i.e., more experienced developers do not want to use a tool while testers are more willing to use), on the success criteria of the tool (i.e., some can tolerate false alarms better than others), and the development context that the tool should be integrated (i.e., integration with IDE or code review tools) [195].

The rapid evolution of software systems would quickly increase the size and complexity of the software, as well as make its maintenance activities complicated [2]. Therefore, obtaining a consistent prediction performance from an SDP tool over time may be challenging. Further research on the non-stationary nature of the software and the defect patterns is needed.

Predictive success of SDP models has been reported many times [50,98,108,143] and it is still a major concern of the researchers. However, the different perceptions of individuals make the interpretability of models more critical [42,196]. For example,

developers want to see actionable outputs from SDP tools instead of receiving just a binary prediction value (defected or clean) [197]. Therefore, recently, the explainability of recommendation systems draw the required attention from the software research community [198] as well as other communities that focus on building AI/ML based solutions [199]. The state-of-the-art status in the explainability of SDP models is enriching the output of models with the contribution rates of metrics to the prediction to help developers to understand why the defect occurred, and also highlighting the code lines that are blamed for the defective development to give developers a start point for defect haunting [197]. Future of SDP studies may involve integrating the domain knowledge into explainability of models [42]. For example, tailoring the explanations in the prediction outputs according to the individual or group characteristics would be helpful, i.e., new beginners may need more direction in outputs.

REFERENCES

[1] **Tian, J.** (2005). *Software quality engineering: testing, quality assurance, and quantifiable improvement*, John Wiley & Sons.

[2] **Kamei, Y. and Shihab, E.** (2016). Defect prediction: Accomplishments and future challenges, *2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER)*, volume 5, IEEE, pp.33–45.

[3] **Malhotra, R.** (2019). *Empirical research in software engineering: concepts, analysis, and applications*, Chapman and Hall/CRC.

[4] **Dsouza, M.** (2018). *5 ways artificial intelligence is upgrading software engineering*, https://hub.packtpub.com/5-ways-artificial-intelligence-is-upgrading-software-engineering, date retrieved: 09.07.2019.

[5] **Gasparic, M. and Janes, A.** (2016). What recommendation systems for software engineering recommend: A systematic literature review, *Journal of Systems and Software*, *113*, 101–113.

[6] **Felleisen, M.**, **Findler, R.B.**, **Flatt, M.**, **Krishnamurthi, S.**, **Barzilay, E.**, **McCarthy, J. and Tobin-Hochstadt, S.** (2015). The racket manifesto, *1st Summit on Advances in Programming Languages (SNAPL 2015)*, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[7] **Bader, J.**, **Scott, A.**, **Pradel, M. and Chandra, S.** (2019). Getafix: Learning to fix bugs automatically, *Proceedings of the ACM on Programming Languages*, *3*(OOPSLA), 1–27.

[8] **Ali, S.**, **Briand, L.C.**, **Hemmati, H. and Panesar-Walawege, R.K.** (2009). A systematic review of the application and empirical investigation of search-based test case generation, *IEEE Transactions on Software Engineering*, *36*(6), 742–762.

[9] **Catolino, G.**, **Palomba, F.**, **Zaidman, A. and Ferrucci, F.** (2019). Not all bugs are the same: Understanding, characterizing, and classifying bug types, *Journal of Systems and Software*, *152*, 165–181.

[10] **Moser, R.**, **Pedrycz, W. and Succi, G.** (2008). A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction, *Proceedings of the 30th international conference on Software engineering*, ACM, pp.181–190.

[11] **Malhotra, R.** (2015). A systematic review of machine learning techniques for software fault prediction, *Applied Soft Computing*, *27*, 504–518.

[12] **Shull, F.**, **Basili, V.**, **Boehm, B.**, **Brown, A.W.**, **Costa, P.**, **Lindvall, M.**, **Port, D.**, **Rus, I.**, **Tesoriero, R. and Zelkowitz, M.** (2002). What we have learned about fighting defects, *Software Metrics, 2002. Proceedings. Eighth IEEE Symposium on*, IEEE, pp.249–258.

[13] **Menzies, T.**, **Greenwald, J. and Frank, A.** (2007). Data mining static code attributes to learn defect predictors, *IEEE transactions on software engineering*, *33*(1).

[14] **Hall, T.**, **Beecham, S.**, **Bowes, D.**, **Gray, D. and Counsell, S.** (2012). A systematic literature review on fault prediction performance in software engineering, *IEEE Transactions on Software Engineering*, *38*(6), 1276–1304.

[15] **McCabe, T.J.** (1976). A complexity measure, *IEEE Transactions on software Engineering*, (4), 308–320.

[16] **Kamei, Y.**, **Shihab, E.**, **Adams, B.**, **Hassan, A.E.**, **Mockus, A.**, **Sinha, A. and Ubayashi, N.** (2012). A large-scale empirical study of just-in-time quality assurance, *IEEE Transactions on Software Engineering*, *39*(6), 757–773.

[17] **Ostrand, T.J.**, **Weyuker, E.J. and Bell, R.M.** (2010). Programmer-based fault prediction, *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, pp.1–10.

[18] **Radjenović, D.**, **Heričko, M.**, **Torkar, R. and Živkovič, A.** (2013). Software fault prediction metrics: A systematic literature review, *Information and software technology*, *55*(8), 1397–1418.

[19] **Hall, G.A. and Munson, J.C.** (2000). Software evolution: code delta and code churn, *Journal of Systems and Software*, *54*(2), 111–118.

[20] **Hoang, T.**, **Dam, H.K.**, **Kamei, Y.**, **Lo, D. and Ubayashi, N.** (2019). DeepJIT: an end-to-end deep learning framework for just-in-time defect prediction, *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, IEEE, pp.34–45.

[21] **Nagappan, N.**, **Zeller, A.**, **Zimmermann, T.**, **Herzig, K. and Murphy, B.** (2010). Change bursts as defect predictors, *2010 IEEE 21st international symposium on software reliability engineering*, IEEE, pp.309–318.

[22] **Bird, C.**, **Nagappan, N.**, **Gall, H.**, **Murphy, B. and Devanbu, P.** (2009). Putting it all together: Using socio-technical networks to predict failures, *2009 20th International Symposium on Software Reliability Engineering*, IEEE, pp.109–119.

[23] **Caglayan, B.**, **Bener, A. and Koch, S.** (2009). Merits of using repository metrics in defect prediction for open source projects, *ICSE Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development*, pp.31–36.

[24] **Guo, L.**, **Ma, Y.**, **Cukic, B. and Singh, H.** (2004). Robust prediction of fault-proneness by random forests, *15th international symposium on software reliability engineering*, IEEE, pp.417–428.

[25] **Khoshgoftaar, T.M.**, **Allen, E.B.**, **Jones, W.D. and Hudepohl, J.** (1999). Classification tree models of software quality over multiple releases, *Software Reliability Engineering, 1999. Proceedings. 10th International Symposium on*, IEEE, pp.116–125.

[26] **Selby, R.W. and Porter, A.A.** (1988). Learning from examples: generation and evaluation of decision trees for software resource analysis, *IEEE Transactions on Software Engineering*, *14*(12), 1743–1757.

[27] **Misirli, A.T.**, **Shihab, E. and Kamei, Y.** (2016). Studying high impact fix-inducing changes, *Empirical Software Engineering*, *21*(2), 605–641.

[28] **Pascarella, L.**, **Palomba, F. and Bacchelli, A.** (2019). Fine-grained just-in-time defect prediction, *Journal of Systems and Software*, *150*, 22–36.

[29] **Tan, M.**, **Tan, L.**, **Dara, S. and Mayeux, C.** (2015). Online defect prediction for imbalanced data, *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, IEEE, pp.99–108.

[30] **Mockus, A. and Weiss, D.M.** (2000). Predicting risk of software changes, *Bell Labs Technical Journal*, *5*(2), 169–180.

[31] **Posnett, D.**, **D'Souza, R.**, **Devanbu, P. and Filkov, V.** (2013). Dual ecological measures of focus in software development, *2013 35th International Conference on Software Engineering (ICSE)*, IEEE, pp.452–461.

[32] **Lee, T.**, **Nam, J.**, **Han, D.**, **Kim, S. and In, H.P.** (2016). Developer Micro Interaction Metrics for Software Defect Prediction, *IEEE Transactions on Software Engineering*, *42*(11), 1015–1035.

[33] **Calikli, G. and Bener, A.** (2015). Empirical analysis of factors affecting confirmation bias levels of software engineers, *Software Quality Journal*, *23*(4), 695–722.

[34] **Zimmermann, T. and Nagappan, N.** (2008). Predicting defects using network analysis on dependency graphs, *Proceedings of the 30th international conference on Software engineering*, pp.531–540.

[35] **Tamburri, D.A.**, **Kruchten, P.**, **Lago, P. and Van Vliet, H.** (2015). Social debt in software engineering: insights from industry, *Journal of Internet Services and Applications*, *6*(1), 10.

[36] **Jiang, T.**, **Tan, L. and Kim, S.** (2013). Personalized defect prediction, *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, IEEE, pp.279–289.

[37] **Horling, B. and Kulick, M.** (2009). Personalized Search for everyone, *Official Google Blog*, https://googleblog.blogspot.com.tr/2009/12/personalized-search-for-everyone.html, date retrieved: 29.08.2021.

[38] **Tucker, C.E.** (2014). Social networks, personalized advertising, and privacy controls, *Journal of marketing research*, *51*(5), 546–562.

[39] **Aggarwal, C.C.** *et al.* (2016). *Recommender systems*, volume 1, Springer.

[40] **Avazpour, I.**, **Pitakrat, T.**, **Grunske, L. and Grundy, J.** (2014). Dimensions and metrics for evaluating recommendation systems, *Recommendation systems in software engineering*, 245–273.

[41] **Gomez-Uribe, C.** (2012). Challenges and limitations in the offline and online evaluation of recommender systems: A Netflix case study, *Proceedings of the Workshop on Recommendation Utility Evaluation: Beyond RMSE, CEUR Workshop Proceedings*, volume910, Citeseer, p. 1.

[42] **Mori, T. and Uchihira, N.** (2019). Balancing the trade-off between accuracy and interpretability in software defect prediction, *Empirical Software Engineering*, *24*(2), 779–825.

[43] **Altinger, H.**, **Herbold, S.**, **Schneemann, F.**, **Grabowski, J. and Wotawa, F.** (2017). Performance tuning for automotive software fault prediction, *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, IEEE, pp.526–530.

[44] **Yan, M.**, **Xia, X.**, **Fan, Y.**, **Lo, D.**, **Hassan, A.E. and Zhang, X.** (2020). Effort-aware just-in-time defect identification in practice: a case study at Alibaba, *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp.1308–1319.

[45] **Kang, J.**, **Ryu, D. and Baik, J.** (2021). Predicting just-in-time software defects to reduce post-release quality costs in the maritime industry, *Software: Practice and Experience*, *51*(4), 748–771.

[46] **Khanan, C.**, **Luewichana, W.**, **Pruktharathikoon, K.**, **Jiarpakdee, J.**, **Tantithamthavorn, C.**, **Choetkiertikul, M.**, **Ragkhitwetsagul, C. and Sunetnanta, T.** (2020). JITBot: an explainable just-in-time defect prediction bot, *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pp.1336–1339.

[47] **Sayyad Shirabad, J. and Menzies, T.** (2005). *The PROMISE Repository of Software Engineering Databases.*, School of Information Technology and Engineering, University of Ottawa, Canada, Retrieved from, http://promise.site.uottawa.ca/SERepository.

[48] **Marijan, D. and Gotlieb, A.** (2021). Industry-Academia research collaboration in software engineering: The Certus model, *Information and Software Technology*, *132*, 106473.

[49] **Menzies, T.**, **Turhan, B.**, **Bener, A.**, **Gay, G.**, **Cukic, B. and Jiang, Y.** (2008). Implications of ceiling effects in defect predictors, *Proceedings of the 4th international workshop on Predictor models in software engineering*, ACM, pp.47–54.

[50] **Lessmann, S.**, **Baesens, B.**, **Mues, C. and Pietsch, S.** (2008). Benchmarking classification models for software defect prediction: A proposed framework and novel findings, *IEEE Transactions on Software Engineering*, *34*(4), 485–496.

[51] **Tsakiltsidis, S.**, **Miranskyy, A. and Mazzawi, E.** (2016). On automatic detection of performance bugs, *2016 IEEE international symposium on software reliability engineering workshops (ISSREW)*, IEEE, pp.132–139.

[52] **Fu, W.**, **Menzies, T. and Shen, X.** (2016). Tuning for software analytics: Is it really necessary?, *Information and Software Technology*, *76*, 135–146.

[53] **Xia, X.**, **Lo, D.**, **Wang, X. and Yang, X.** (2016). Collective personalized change classification with multiobjective search, *IEEE Transactions on Reliability*, *65*(4), 1810–1829.

[54] **Tamburri, D.A.A.**, **Palomba, F. and Kazman, R.** (2019). Exploring community smells in open-source: An automated approach, *IEEE Transactions on software Engineering*.

[55] **Giarola, F.** (2018). *Detecting code and community smells in open-source: an automated approach*, (Master's Thesis). School of Industrial and Information Engineering, Politecnico di Milano, Italy.

[56] **Palomba, F.**, **Zanoni, M.**, **Fontana, F.A.**, **De Lucia, A. and Oliveto, R.** (2017). Toward a smell-aware bug prediction model, *IEEE Transactions on Software Engineering*, *45*(2), 194–218.

[57] **Menzies, T.**, **Milton, Z.**, **Turhan, B.**, **Cukic, B.**, **Jiang, Y. and Bener, A.** (2010). Defect prediction from static code features: current results, limitations, new approaches, *Automated Software Engineering*, *17*(4), 375–407.

[58] **Kirbas, S.**, **Caglayan, B.**, **Hall, T.**, **Counsell, S.**, **Bowes, D.**, **Sen, A. and Bener, A.** (2017). The relationship between evolutionary coupling and defects in large industrial software, *Journal of Software: Evolution and Process*, *29*(4), e1842.

[59] **Caglayan, B.** (2014). *An Issue Recommender Model Using the Developer Collaboration Network*, (Ph.D. Thesis). Boğaziçi Üniversitesi, Fen Bilimleri Enstitüsü, İstanbul.

[60] **Tosun, A.**, **Turhan, B. and Bener, A.** (2009). Validation of network measures as indicators of defective modules in software systems, *Proceedings of the 5th international conference on predictor models in software engineering*, ACM, p. 5.

[61] **Kini, S.O. and Tosun, A.** (2018). Periodic Developer Metrics in Software Defect Prediction, *18th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2018, Madrid, Spain, September 23-24, 2018*, pp.72–81.

[62] **Khomh, F.**, **Di Penta, M.**, **Guéhéneuc, Y.G. and Antoniol, G.** (2012). An exploratory study of the impact of antipatterns on class change-and fault-proneness, *Empirical Software Engineering*, *17*(3), 243–275.

[63] **Tufano, M.**, **Palomba, F.**, **Bavota, G.**, **Oliveto, R.**, **Di Penta, M.**, **De Lucia, A. and Poshyvanyk, D.** (2017). When and why your code starts to smell bad (and whether the smells go away), *IEEE Transactions on Software Engineering*, *43*(11), 1063–1088.

[64] **Palomba, F.**, **Bavota, G.**, **Di Penta, M.**, **Fasano, F.**, **Oliveto, R. and De Lucia, A.** (2018). On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation, *Empirical Software Engineering*, *23*(3), 1188–1221.

[65] **Taba, S.E.S.**, **Khomh, F.**, **Zou, Y.**, **Hassan, A.E. and Nagappan, M.** (2013). Predicting bugs using antipatterns, *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, IEEE, pp.270–279.

[66] **Soltanifar, B.**, **Akbarinasaji, S.**, **Caglayan, B.**, **Bener, A.B.**, **Filiz, A. and Kramer, B.M.** (2016). Software analytics in practice: A defect prediction model using code smells, *Proceedings of the 20th International Database Engineering & Applications Symposium*, pp.148–155.

[67] **Tamburri, D.A.**, **Lago, P. and Vliet, H.v.** (2013). Organizational social structures for software engineering, *ACM Computing Surveys (CSUR)*, *46*(1), 3.

[68] **Di Nucci, D.**, **Palomba, F.**, **De Rosa, G.**, **Bavota, G.**, **Oliveto, R. and De Lucia, A.** (2018). A developer centered bug prediction model, *IEEE Transactions on Software Engineering*, *44*(1), 5–24.

[69] **Magnoni, S.** (2016). *An approach to measure community smells in software development communities*, (Master's Thesis). School of Industrial and Information Engineering, Politecnico di Milano, Italy.

[70] **Mauerer, W.** (2010). *Codeface*, http://siemens.github.io/codeface, date retrieved 29.08.2021.

[71] **Palomba, F.**, **Tamburri, D.A.A.**, **Fontana, F.A.**, **Oliveto, R.**, **Zaidman, A. and Serebrenik, A.** (2018). Beyond technical aspects: How do community smells influence the intensity of code smells?, *IEEE transactions on software engineering*.

[72] **Jureczko, M. and Madeyski, L.** (2010). Towards identifying software project clusters with regard to defect prediction, *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, ACM, p. 9.

[73] **Hassan, A.E.** (2009). Predicting faults using the complexity of code changes, *Proceedings of the 31st International Conference on Software Engineering*, IEEE Computer Society, pp.78–88.

[74] **Bell, R.M.**, **Ostrand, T.J. and Weyuker, E.J.** (2013). The limited impact of individual developer data on software defect prediction, *Empirical Software Engineering*, *18*(3), 478–505.

[75] **Fontana, F.A.**, **Ferme, V.**, **Zanoni, M. and Roveda, R.** (2015). Towards a prioritization of code debt: A code smell intensity index, *2015 IEEE 7th International Workshop on Managing Technical Debt (MTD)*, IEEE, pp.16–24.

[76] **Eken, B.**, **Tosun, A.**, **Palma, F. and Bener, A.** (2019). *An Empirical Study on the Effect of Community Smells on Bug Prediction*, https://figshare.com/articles/BugDBzip/7749281, date retrieved 29.08.2021.

[77] **Tosun, A.**, **Bener, A.**, **Turhan, B. and Menzies, T.** (2010). Practical considerations in deploying statistical methods for defect prediction: A case study within the Turkish telecommunications industry, *Information and Software Technology*, *52*(11), 1242–1257.

[78] **Turhan, B.**, **Koçak, G. and Bener, A.B.** (2009). Data mining source code for locating software bugs: A case study in telecommunication industry, *Expert Syst. Appl.*, *36*(6), 9986–9990, https://doi.org/10.1016/j.eswa.2008.12.028.

[79] **Nagappan, N.**, **Murphy, B. and Basili, V.** (2008). The influence of organizational structure on software quality, *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*, IEEE, pp.521–530.

[80] **Turhan, B.**, **Mısırlı, A.T. and Bener, A.** (2013). Empirical evaluation of the effects of mixed project data on learning defect predictors, *Information and Software Technology*, *55*(6), 1101–1118.

[81] **Zhang, F.**, **Hassan, A.E.**, **McIntosh, S. and Zou, Y.** (2016). The use of summation to aggregate software metrics hinders the performance of defect prediction models, *IEEE Transactions on Software Engineering*, *43*(5), 476–491.

[82] **Tantithamthavorn, C.**, **McIntosh, S.**, **Hassan, A.E. and Matsumoto, K.** (2016). An empirical comparison of model validation techniques for defect prediction models, *IEEE Transactions on Software Engineering*, *43*(1), 1–18.

[83] **Fowler, M.** (2018). *Refactoring: improving the design of existing code*, Addison-Wesley Professional.

[84] **Chatzigeorgiou, A. and Manakos, A.** (2010). Investigating the evolution of bad smells in object-oriented code, *International Conference on the Quality of Information and Communications Technology*, IEEE, pp.106–115.

[85] **Peters, R. and Zaidman, A.** (2012). Evaluating the lifespan of code smells using software repository mining, *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*, IEEE, pp.411–416.

[86] **Arcoverde, R.**, **Garcia, A. and Figueiredo, E.** (2011). Understanding the longevity of code smells: preliminary results of an explanatory survey, *Proceedings of the 4th Workshop on Refactoring Tools*, ACM, pp.33–36.

[87] **Palomba, F.**, **Bavota, G.**, **Di Penta, M.**, **Oliveto, R. and De Lucia, A.** (2014). Do they really smell bad? a study on developers' perception of bad code smells, *Software maintenance and evolution (ICSME), 2014 IEEE international conference on*, IEEE, pp.101–110.

[88] **Eken, B.** (2018). Assessing personalized software defect predictors, *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, pp.488–491.

[89] **Soltanifar, B.**, **Erdem, A. and Bener, A.** (2016). Predicting defectiveness of software patches, *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pp.1–10.

[90] **Calikli, G.**, **Bener, A.B.**, **Caglayan, B. and Misirli, A.T.** (2012). Modeling Human Aspects to Enhance Software Quality Management, *ICIS*.

[91] **Tamburri, D.A.**, **Palomba, F. and Kazman, R.** (2017). Exploring Community Smells in Open-Source: An Automated Approach, *IEEE Transactions on Software Engineering*, *14*(8), 1–24.

[92] **Almarimi, N.**, **Ouni, A.**, **Chouchen, M.**, **Saidani, I. and Mkaouer, M.W.** (2020). On the Detection of Community Smells using Genetic Programming-based Ensemble Classifier Chain, *15th IEEE/ACM International Conference on Global Software Engineering (ICGSE)*, pp.1–12.

[93] **Almarimi, N.**, **Ouni, A. and Mkaouer, M.W.** (2020). Learning to detect community smells in open source software projects, *Knowledge-Based Systems*, *204*, 106201.

[94] **Catolino, G.**, **Palomba, F.**, **Tamburri, D.A.**, **Serebrenik, A. and Ferrucci, F.** (2019). Gender diversity and women in software teams: How do they affect community smells?, *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Society (ICSE-SEIS)*, IEEE, pp.11–20.

[95] **Catolino, G.**, **Palomba, F.**, **Tamburri, D.A.**, **Serebrenik, A. and Ferrucci, F.** (2020). Refactoring community smells in the wild: the practitioner's field manual, *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Society*, pp.25–34.

[96] **URL-1**. https://www.spinellis.gr/sw/ckjm/doc/index.html, date retrieved: 03.02.2022.

[97] **Kohavi, R.** (1995). The Power of Decision Tables, *8th European Conference on Machine Learning*, Springer, pp.174–189.

[98] **Hall, M.A. and Holmes, G.** (2003). Benchmarking attribute selection techniques for discrete class data mining, *IEEE Transactions on Knowledge and Data engineering*, *15*(6), 1437–1447.

[99] **URL-2**. https://cran.r-project.org/web/packages/ScottKnottESD/index.html, date retrieved: 03.02.2022.

[100] **Turhan, B.**, **Menzies, T.**, **Bener, A.B. and Di Stefano, J.** (2009). On the relative value of cross-company and within-company data for defect prediction, *Empirical Software Engineering*, *14*(5), 540–578.

[101] **Yin, R.K.** (2017). *Case study research: Design and methods.*, SAGE Publications, Incorporated.

[102] **Meneely, A. and Williams, L.** (2011). Socio-technical developer networks: Should we trust our measurements?, *Proceedings of the 33rd International Conference on Software Engineering*, ACM, pp.281–290.

[103] **Palomba, F.**, **Zanoni, M.**, **Fontana, F.A.**, **De Lucia, A. and Oliveto, R.** (2016). Smells like teen spirit: Improving bug prediction performance using the intensity of code smells, *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, IEEE, pp.244–255.

[104] **Robillard, M.**, **Walker, R. and Zimmermann, T.** (2009). Recommendation systems for software engineering, *IEEE software*, *27*(4), 80–86.

[105] **Hall, T.**, **Beecham, S.**, **Bowes, D.**, **Gray, D. and Counsell, S.** (2011). A systematic literature review on fault prediction performance in software engineering, *IEEE Transactions on Software Engineering*, *38*(6), 1276–1304.

[106] **Weyuker, E.J.**, **Ostrand, T.J. and Bell, R.M.** (2008). Do too many cooks spoil the broth? Using the number of developers to enhance defect prediction models, *Empirical Software Engineering*, *13*(5), 539–559.

[107] **Di Nucci, D.**, **Palomba, F.**, **De Rosa, G.**, **Bavota, G.**, **Oliveto, R. and De Lucia, A.** (2017). A developer centered bug prediction model, *IEEE Transactions on Software Engineering*.

[108] **D'Ambros, M.**, **Lanza, M. and Robbes, R.** (2012). Evaluating defect prediction approaches: a benchmark and an extensive comparison, *Empirical Software Engineering*, *17*(4-5), 531–577.

[109] **Graves, T.L.**, **Karr, A.F.**, **Marron, J.S. and Siy, H.** (2000). Predicting fault incidence using software change history, *IEEE Transactions on software engineering*, *26*(7), 653–661.

[110] **Matsumoto, S.**, **Kamei, Y.**, **Monden, A.**, **Matsumoto, K.i. and Nakamura, M.** (2010). An analysis of developer metrics for fault prediction, *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, ACM, p. 18.

[111] **Nagappan, N. and Ball, T.** (2005). Use of relative code churn measures to predict system defect density, *Proceedings of the 27th international conference on Software engineering*, pp.284–292.

[112] **D'Ambros, M.**, **Lanza, M. and Robbes, R.** (2010). An extensive comparison of bug prediction approaches, *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, IEEE, pp.31–41.

[113] **Nagappan, N.**, **Ball, T. and Zeller, A.** (2006). Mining metrics to predict component failures, *Proceedings of the 28th international conference on Software engineering*, pp.452–461.

[114] **Schein, A.I.**, **Popescul, A.**, **Ungar, L.H. and Pennock, D.M.** (2002). Methods and metrics for cold-start recommendations, *Proceedings of the 25th annual international ACM SIGIR conference on Research and development in information retrieval*, pp.253–260.

[115] **Shihab, E.**, **Hassan, A.E.**, **Adams, B. and Jiang, Z.M.** (2012). An industrial study on the risk of software changes, *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pp.1–11.

[116] **Śliwerski, J.**, **Zimmermann, T. and Zeller, A.** (2005). When do changes induce fixes?, *ACM sigsoft software engineering notes*, *30*(4), 1–5.

[117] **Kim, S.**, **Whitehead, E.J. and Zhang, Y.** (2008). Classifying software changes: Clean or buggy?, *IEEE Transactions on software engineering*, *34*(2), 181–196.

[118] **Jiang, Y.**, **Cukic, B. and Menzies, T.** (2008). Can data transformation help in the detection of fault-prone modules?, *Proceedings of the 2008 workshop on Defects in large software systems*, pp.16–20.

[119] **Yan, M.**, **Xia, X.**, **Fan, Y.**, **Hassan, A.E.**, **Lo, D. and Li, S.** (2020). Just-in-time defect identification and localization: A two-phase framework, *IEEE Transactions on Software Engineering*.

[120] **Pornprasit, C. and Tantithamthavorn, C.** (2021). JITLine: A Simpler, Better, Faster, Finer-grained Just-In-Time Defect Prediction, *arXiv preprint arXiv:2103.07068*.

[121] **Schröter, A.**, **Zimmermann, T.**, **Premraj, R. and Zeller, A.** (2006). If your bug database could talk, *Proceedings of the 5th international symposium on empirical software engineering*, volume 2, Citeseer, pp.18–20.

[122] **Zeller, A.** (2009). *Why programs fail: a guide to systematic debugging*, Elsevier.

[123] **Tufano, M.**, **Bavota, G.**, **Poshyvanyk, D.**, **Di Penta, M.**, **Oliveto, R. and De Lucia, A.** (2017). An empirical study on developer-related factors characterizing fix-inducing commits, *Journal of Software: Evolution and Process*, *29*(1), e1797.

[124] **Eyolfson, J.**, **Tan, L. and Lam, P.** (2011). Do time of day and developer experience affect commit bugginess?, *Proceedings of the 8th Working Conference on Mining Software Repositories*, pp.153–162.

[125] **Rahman, F. and Devanbu, P.** (2011). Ownership, experience and defects: a fine-grained study of authorship, *Proceedings of the 33rd International Conference on Software Engineering*, pp.491–500.

[126] **Bettenburg, N.**, **Nagappan, M. and Hassan, A.E.** (2012). Think locally, act globally: Improving defect and effort prediction models, *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*, IEEE, pp.60–69.

[127] **Friedman, J.H.** (1991). Multivariate adaptive regression splines, *The annals of statistics*, 1–67.

[128] **Raudys, S.J.**, **Jain, A.K.** *et al.* (1991). Small sample size effects in statistical pattern recognition: Recommendations for practitioners, *IEEE Transactions on pattern analysis and machine intelligence*, *13*(3), 252–264.

[129] **Yamashita, K.**, **McIntosh, S.**, **Kamei, Y.**, **Hassan, A.E. and Ubayashi, N.** (2015). Revisiting the applicability of the pareto principle to core development teams in open source software projects, *Proceedings of the 14th International Workshop on Principles of Software Evolution*, pp.46–55.

[130] **Yang, Y.**, **Zhou, Y.**, **Liu, J.**, **Zhao, Y.**, **Lu, H.**, **Xu, L.**, **Xu, B. and Leung, H.** (2016). Effort-aware just-in-time defect prediction: simple unsupervised models could be better than supervised models, *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp.157–168.

[131] **Young, S.**, **Abdou, T. and Bener, A.** (2018). A replication study: just-in-time defect prediction with ensemble learning, *Proceedings of the 6th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering*, pp.42–47.

[132] **Qiao, L. and Wang, Y.** (2019). Effort-aware and just-in-time defect prediction with neural network, *PloS one*, *14*(2).

[133] **Bangash, A.A.**, **Sahar, H.**, **Hindle, A. and Ali, K.** (2019). On the Time-Based Conclusion Stability of Software Defect Prediction Models, *arXiv preprint arXiv:1911.06348*.

[134] **Tantithamthavorn, C.**, **Hassan, A.E. and Matsumoto, K.** (2018). The impact of class rebalancing techniques on the performance and interpretation of defect prediction models, *IEEE Transactions on Software Engineering*.

[135] **Nemenyi, P.** (1963). *Distribution-free multiple comparisons*, (Ph.D. Thesis). Princeton University, Princeton, New Jersey.

[136] **Kampenes, V.B.**, **Dybå, T.**, **Hannay, J.E. and Sjøberg, D.I.** (2007). A systematic review of effect size in software engineering experiments, *Information and Software Technology*, *49*(11-12), 1073–1086.

[137] **Matthews, B.W.** (1975). Comparison of the predicted and observed secondary structure of T4 phage lysozyme, *Biochimica et Biophysica Acta (BBA)-Protein Structure*, *405*(2), 442–451.

[138] **Brier, G.W.** (1950). Verification of forecasts expressed in terms of probability, *Monthly weather review*, *78*(1), 1–3.

[139] **Rufibach, K.** (2010). Use of Brier score to assess binary predictions, *Journal of Clinical Epidemiology*, *63*(8), 938–939.

[140] **Conover, W.J.** (1999). *Practical nonparametric statistics*, volume:350, john wiley & sons.

[141] **Quinlan, J.R.** (1986). Induction of decision trees, *Machine learning*, *1*(1), 81–106.

[142] **URL-3**. https://kovan.itu.edu.tr/index.php/s/cR4dJpn6nL8wvdb, date retrieved: 03.02.2022.

[143] **Li, L.**, **Lessmann, S. and Baesens, B.** (2019). Evaluating software defect prediction performance: an updated benchmarking study, *arXiv preprint arXiv:1901.01726*.

[144] **Yao, J. and Shepperd, M.**, (2020). Assessing software defection prediction performance: why using the Matthews correlation coefficient matters, Proceedings of the Evaluation and Assessment in Software Engineering, pp.120–129.

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

[146] **Falessi, D.**, **Huang, J.**, **Narayana, L.**, **Thai, J.F. and Turhan, B.** (2018). On the Need of Preserving Order of Data When Validating Within-Project Defect Classifiers, *arXiv preprint arXiv:1809.01510*.

[147] **Hryszko, J. and Madeyski, L.** (2015). Bottlenecks in software defect prediction implementation in industrial projects, *Foundations of Computing and Decision Sciences*, *40*(1), 17–33.

[148] **Eken, B.**, **Atar, R.**, **Sertalp, S. and Tosun, A.** (2019). Predicting Defects with Latent and Semantic Features from Commit Logs in an Industrial Setting, *2019 34th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*, IEEE, pp.98–105.

[149] **Borg, M.**, **Svensson, O.**, **Berg, K. and Hansson, D.** (2019). SZZ Unleashed: An Open Implementation of the SZZ Algorithm - Featuring Example Usage in a Study of Just-in-time Bug Prediction for the Jenkins Project, *Proceedings of the 3rd ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation*, MaLTeSQuE 2019, ACM, pp.7–12.

[150] **Da Costa, D.A.**, **McIntosh, S.**, **Shang, W.**, **Kulesza, U.**, **Coelho, R. and Hassan, A.E.** (2016). A framework for evaluating the results of the szz approach for identifying bug-introducing changes, *IEEE Transactions on Software Engineering*, *43*(7), 641–657.

[151] **Fan, Y.**, **Xia, X.**, **da Costa, D.A.**, **Lo, D.**, **Hassan, A.E. and Li, S.** (2019). The Impact of Changes Mislabeled by SZZ on Just-in-Time Defect Prediction, *IEEE Transactions on Software Engineering*.

[152] **Arisholm, E. and Briand, L.C.** (2006). Predicting fault-prone components in a java legacy system, *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, ACM, pp.8–17.

[153] **Nguyen, T.T.**, **Nguyen, T.N. and Phuong, T.M.** (2011). Topic-based defect prediction (nier track), *Proc. of 33rd Int. Conf. on Software Engineering*, pp.932–935.

[154] **Chen, T.H.**, **Thomas, S.W.**, **Nagappan, M. and Hassan, A.E.** (2012). Explaining software defects using topic models, *9th IEEE Working Conference on Mining Software Repositories (MSR)*, pp.189–198.

[155] **Barnett, J.G.**, **Gathuru, C.K.**, **Soldano, L.S. and McIntosh, S.** (2016). The relationship between commit message detail and defect proneness in Java projects on GitHub, *Proc. of 13th Int. Conf. on Mining Software Repositories*, pp.496–499.

[156] **Guillamet, D. and Vitrià, J.** (2002). Non-negative matrix factorization for face recognition, *Catalonian Conference on Artificial Intelligence*, pp.336–344.

[157] **Koren, Y.**, **Bell, R. and Volinsky, C.** (2009). Matrix factorization techniques for recommender systems, *IEEE Computer*, (8), 30–37.

[158] **Bozcan, Ö. and Bener, A.B.** (2013). Handling missing attributes using matrix factorization, *2nd International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE)*, pp.49–55.

[159] **Chang, R.**, **Mu, X. and Zhang, L.** (2011). Software defect prediction using non-negative matrix factorization, *Journal of Software*, *6*(11), 2114–2120.

[160] **Lee, D.D. and Seung, H.S.** (1999). Learning the parts of objects by non-negative matrix factorization, *Nature*, *401*(6755), 788.

[161] **Zhang, J.S.**, **Wang, C.P. and Yang, Y.Q.** (2015). Learning latent features by nonnegative matrix factorization combining similarity judgments, *Neurocomputing*, *155*, 43–52.

[162] **Campbell, J.C.**, **Hindle, A. and Stroulia, E.** (2015). Latent Dirichlet allocation: extracting topics from software engineering data, *The art and science of analyzing software data*, Elsevier, pp.139–159.

[163] **Menzies, T.**, **Greenwald, J. and Frank, A.** (2006). Data mining static code attributes to learn defect predictors, *IEEE Transactions on Software Engineering*, *33*(1), 2–13.

[164] **Yu, X.**, **Bennin, K.E.**, **Liu, J.**, **Keung, J.W.**, **Yin, X. and Xu, Z.** (2019). An empirical study of learning to rank techniques for effort-aware defect prediction, *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, IEEE, pp.298–309.

[165] **Czerwonka, J.**, **Das, R.**, **Nagappan, N.**, **Tarvo, A. and Teterev, A.** (2011). Crane: Failure prediction, change analysis and test prioritization in practice–experiences from windows, *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, IEEE, pp.357–366.

[166] **Tantithamthavorn, C. and Hassan, A.E.** (2018). An experience report on defect modelling in practice: Pitfalls and challenges, *Proceedings of the 40th International conference on software engineering: Software engineering in practice*, pp.286–295.

[167] **McIntosh, S. and Kamei, Y.** (2017). Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction, *IEEE Transactions on Software Engineering*, *44*(5), 412–428.

[168] **Cabral, G.G.**, **Minku, L.L.**, **Shihab, E. and Mujahid, S.** (2019). Class imbalance evolution and verification latency in just-in-time software defect prediction, *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, IEEE, pp.666–676.

[169] **Mockus, A.**, **Eick, S.G.**, **Graves, T.L. and Karr, A.F.** (1999). On measurement and analysis of software changes, *IEEE Transactions on software engineering*, *20*, 29.

[170] **Ghotra, B.**, **McIntosh, S. and Hassan, A.E.** (2015). Revisiting the impact of classification techniques on the performance of defect prediction models, *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, IEEE Press, pp.789–800.

[171] **Huang, Q.**, **Xia, X. and Lo, D.** (2019). Revisiting supervised and unsupervised models for effort-aware just-in-time defect prediction, *Empirical Software Engineering*, *24*(5), 2823–2862.

[172] **Kamei, Y.**, **Fukushima, T.**, **McIntosh, S.**, **Yamashita, K.**, **Ubayashi, N. and Hassan, A.E.** (2016). Studying just-in-time defect prediction using cross-project models, *Empirical Software Engineering*, *21*(5), 2072–2106.

[173] **Catolino, G.**, **Di Nucci, D. and Ferrucci, F.** (2019). Cross-project just-in-time bug prediction for mobile apps: An empirical assessment, *2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, IEEE, pp.99–110.

[174] **Yang, X.**, **Lo, D.**, **Xia, X.**, **Zhang, Y. and Sun, J.** (2015). Deep learning for just-in-time defect prediction, *2015 IEEE International Conference on Software Quality, Reliability and Security*, IEEE, pp.17–26.

[175] **Yang, X.**, **Yu, H.**, **Fan, G.**, **Shi, K. and Chen, L.** (2019). Local versus global models for just-in-time software defect prediction, *Scientific Programming*, *2019*.

[176] **Bergmeir, C. and Benítez, J.M.** (2012). On the use of cross-validation for time series predictor evaluation, *Information Sciences*, *191*, 192–213.

[177] **Shalev-Shwartz, S. and Ben-David, S.** (2014). *Understanding machine learning: From theory to algorithms*, Cambridge university press.

[178] **Kim, S. and Whitehead Jr, E.J.** (2006). How long did it take to fix bugs?, *Proceedings of the 2006 international workshop on Mining software repositories*, pp.173–174.

[179] **URL-4**. https://git-scm.com/, date retrieved: 03.02.2022.

[180] **URL-5**. https://www.atlassian.com/software/jira, date retrieved: 03.02.2022.

[181] **URL-6**. https://maven.apache.org/, date retrieved: 03.02.2022.

[182] **Murphy, K.P.** (2012). *Machine learning: a probabilistic perspective*, MIT press.

[183] **Alpaydin, E.** (2009). *Introduction to machine learning*, MIT press.

[184] **Hanley, J.A. and McNeil, B.J.** (1982). The meaning and use of the area under a receiver operating characteristic (ROC) curve., *Radiology*, *143*(1), 29–36.

[185] **Tharwat, A.** (2020). Classification assessment methods, *Applied Computing and Informatics*.

[186] **Davis, J. and Goadrich, M.** (2006). The relationship between Precision-Recall and ROC curves, *Proceedings of the 23rd international conference on Machine learning*, pp.233–240.

[187] **Menzies, T.**, **Dekhtyar, A.**, **Distefano, J. and Greenwald, J.** (2007). Problems with Precision: A Response to" comments on'data mining static code attributes to learn defect predictors'", *IEEE Transactions on Software Engineering*, *33*(9), 637–640.

[188] **URL-7**. https://github.com/beken/DeploymentIndustrial, date retrieved: 03.02.2022.

[189] **Nugroho, Y.S.**, **Hata, H. and Matsumoto, K.** (2020). How different are different diff algorithms in Git?, *Empirical Software Engineering*, *25*(1), 790–823.

[190] **Northcutt, C.**, **Jiang, L. and Chuang, I.** (2021). Confident learning: Estimating uncertainty in dataset labels, *Journal of Artificial Intelligence Research*, *70*, 1373–1411.

[191] **Fenton, N.E. and Neil, M.** (1999). A critique of software defect prediction models, *IEEE Transactions on software engineering*, *25*(5), 675–689.

[192] **Nayebi, M. and Ruhe, G.**, (2015). Analytical product release planning, The Art and Science of Analyzing Software Data, Elsevier, pp.555–589.

[193] **Zhou, J.**, **Wei, J. and Xu, B.** (2021). Customer segmentation by web content mining, *Journal of Retailing and Consumer Services*, *61*, 102588.

[194] **Menzies, T. and Zimmermann, T.** (2018). Software Analytics: What's Next?, *IEEE Software*, *35*(5), 64–70.

[195] **Wan, Z.**, **Xia, X.**, **Hassan, A.E.**, **Lo, D.**, **Yin, J. and Yang, X.** (2018). Perceptions, expectations, and challenges in defect prediction, *IEEE Transactions on Software Engineering*, *46*(11), 1241–1266.

[196] **Freitas, A.A.** (2014). Comprehensible classification models: a position paper, *ACM SIGKDD explorations newsletter*, *15*(1), 1–10.

[197] **Jiarpakdee, J.**, **Tantithamthavorn, C. and Grundy, J.** (2021). Practitioners' Perceptions of the Goals and Visual Explanations of Defect Prediction Models, *arXiv preprint arXiv:2102.12007*.

[198] **Tantithamthavorn, C.**, **Jiarpakdee, J. and Grundy, J.** (2020). Explainable ai for software engineering, *arXiv preprint arXiv:2012.01614*.

[199] **Holzinger, A.** (2018). From machine learning to explainable AI, *2018 world symposium on digital intelligence for systems and machines (DISA)*, IEEE, pp.55–66.

APPENDICES

APPENDIX A : Performances of SDP models reported in Chapter 4 for RQ2

APPENDIX A : Performances of SDP models reported in Chapter 4 for RQ2

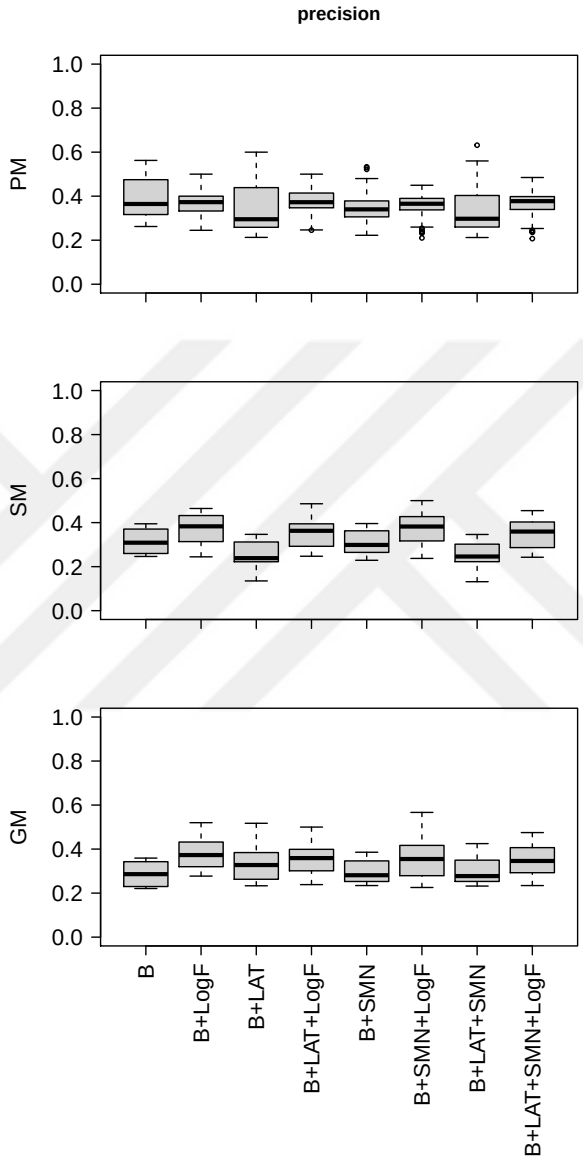

Figure A.1 : Performance (in terms of precision) of PM, SM and GM.

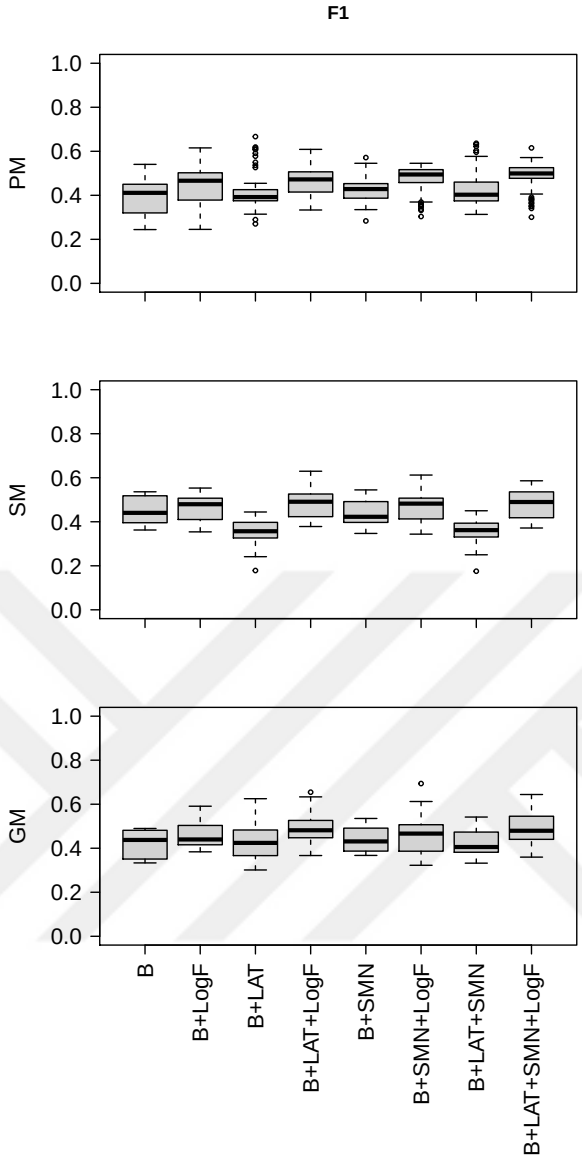

Figure A.2 : Performance (in terms of F1) of PM, SM and GM.

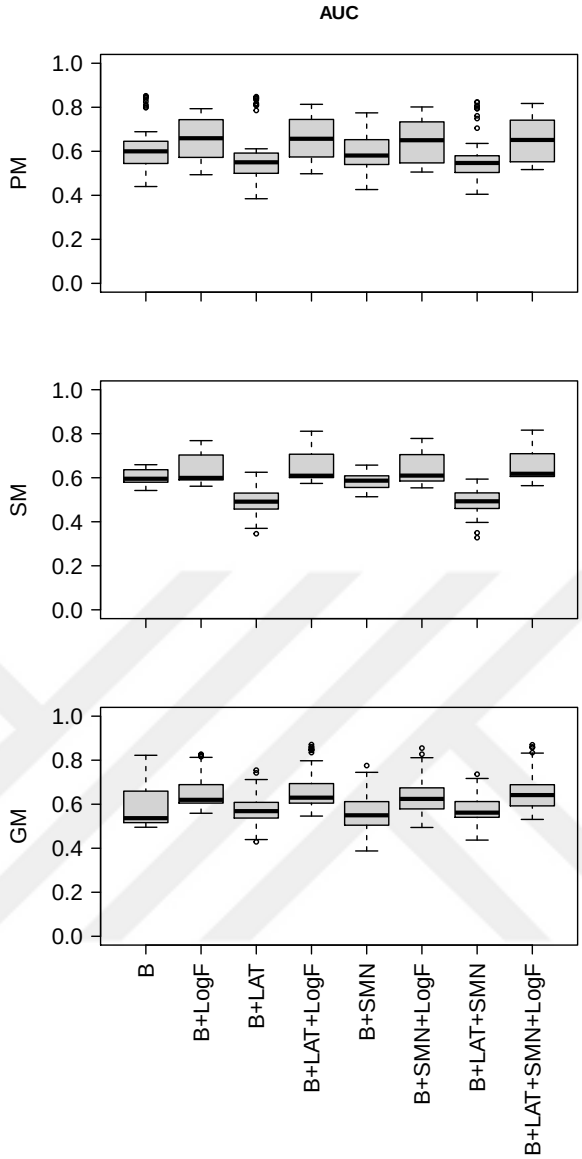

Figure A.3 : Performance (in terms of AUC) of PM, SM and GM.

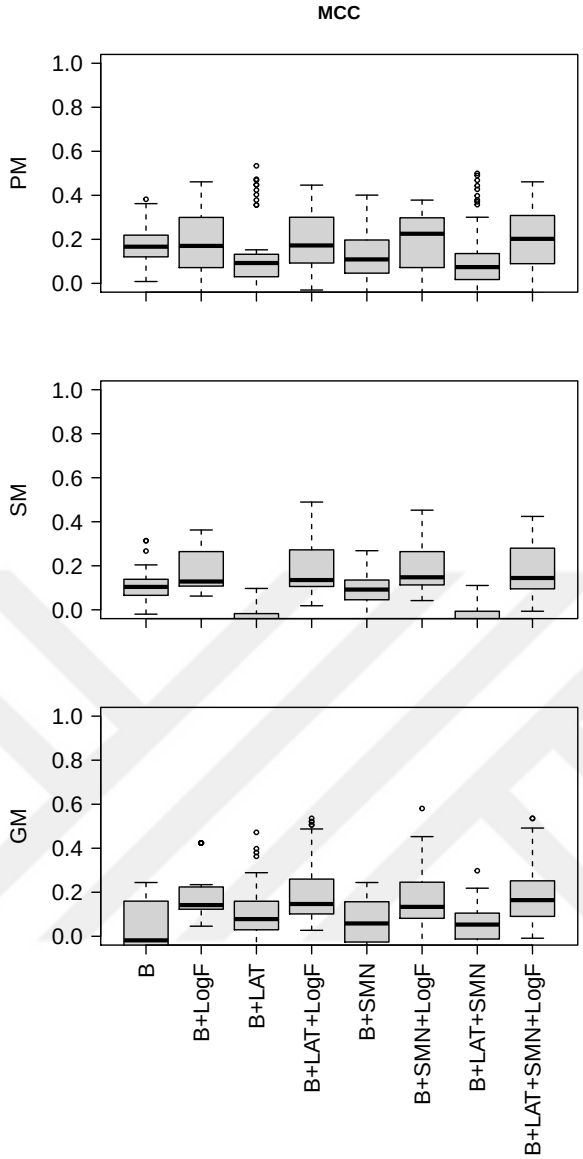

Figure A.4 : Performance (in terms of MCC) of PM, SM and GM.

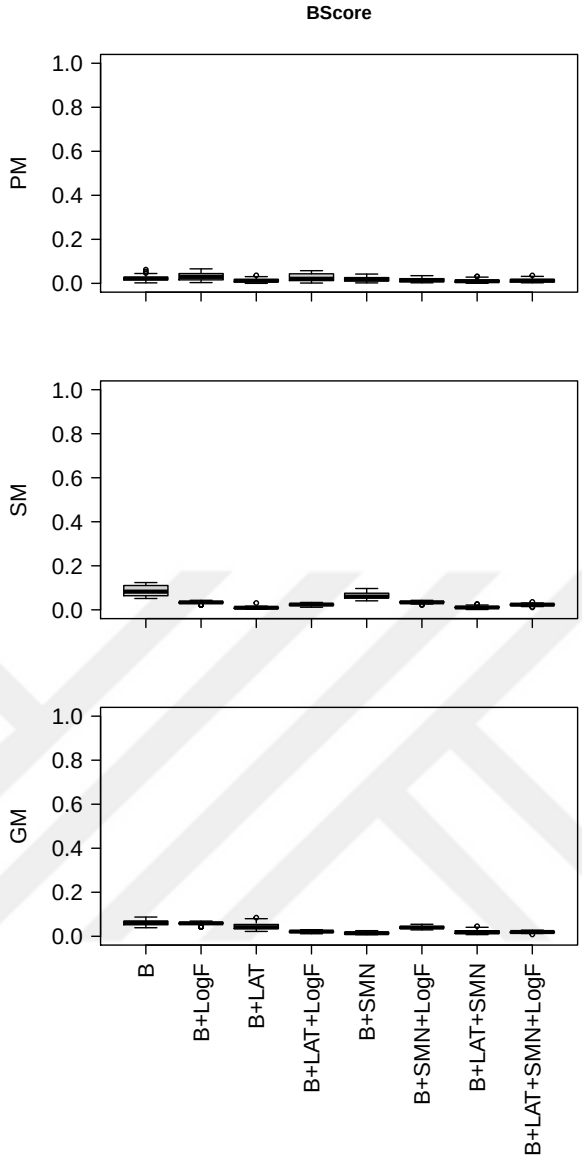

Figure A.5 : Performance (in terms of BScore) of PM, SM and GM.

CURRICULUM VITAE

Name SURNAME: Beyza EKEN

EDUCATION:

- **B.Sc.:** 2011, Sakarya University, Faculty of Engineering, Department of Computer Engineering
- **M.Sc.:** 2015, Istanbul Technical University, Graduate School of Science, Engineering and Technology, Computer Engineering Program

PROFESSIONAL EXPERIENCE AND REWARDS:

- 2014 – Present, Research and teaching assistant at Computer Engineering Department, Istanbul Technical University.
- September – December 2018, Visiting research student at Data Science Laboratory, Ryerson University, Canada (Mevlana Exchange Program).

PUBLICATIONS, PRESENTATIONS AND PATENTS ON THE THESIS:

- **Eken, B.,** Palma, F., Başar, A. Tosun, A., 2021. An empirical study on the effect of community smells on bug prediction. *Software Quality Journal*, 29(1), pp.159-194.
- **Eken, B.,** Tosun, A., 2021. Investigating the performance of personalized models for software defect prediction. *Journal of Systems and Software*, 181, p.111038.
- **Eken B.,** Tufan S., Tunaboylu A., Güler T., Atar R., Tosun A., 2021. Deployment of a change-level software defect prediction solution into an industrial setting. *Journal of Software: Evolution and Process*, 33(11), e2381.
- **Eken B.,** Atar R., Sertalp S. and Tosun A., 2019, November. Predicting Defects with Latent and Semantic Features from Commit Logs in an Industrial Setting. *In 2019 34th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)* (pp. 98-105). IEEE.
- **Eken B.,** 2018, May. Assessing personalized software defect predictors. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings* (pp. 488-491).

OTHER PUBLICATIONS, PRESENTATIONS AND PATENTS:

- Eren, K.K., Ozbey, C., **Eken, B.** and Tosun, A., 2020, March. Customer requests matter: early stage software effort estimation using k-grams. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing* (pp. 1540-1547).