Cold-Start Software Analytics

Jin Guo, Mona Rahimi, Jane Cleland-Huang, Alexander Rasin
School of Computing, DePaul University, Chicago, IL USA
jinzz@gmail.com, jhuang@cs.depaul.edu

Jane Huffman Hayes,
Computer Science Department,
University of Kentucky, USA
hayes@cs.uky.edu

Michael Vierhauser
CDL MEVSS,
Johannes Kepler University,
Linz, Austria
michael.vierhauser@jku.at

ABSTRACT

Software project artifacts such as source code, requirements, and change logs represent a gold-mine of actionable information. As a result, software analytic solutions have been developed to mine repositories and answer questions such as "who is the expert?," "which classes are fault prone?," or even "who are the domain experts for these fault-prone classes?" Analytics often require training and configuring in order to maximize performance within the context of each project. A cold-start problem exists when a function is applied within a project context without first configuring the analytic functions on project-specific data. This scenario exists because of the non-trivial effort necessary to instrument a project environment with candidate tools and algorithms and to empirically evaluate alternate configurations. We address the cold-start problem by comparatively evaluating 'best-of-breed' and 'profile-driven' solutions, both of which reuse known configurations in new project contexts. We describe and evaluate our approach against 20 project datasets for the three analytic areas of artifact connectivity, fault-prediction, and finding the expert, and show that the best-of-breed approach outperformed the profile-driven approach in all three areas; however, while it delivered acceptable results for artifact connectivity and find the expert, both techniques underperformed for cold-start fault prediction.

CCS Concepts

•**Software and its engineering** → **Software libraries and repositories;**

Keywords

Cold-start, Software Analytics, Configuration

1. INTRODUCTION

Software analytics allows us to answer various interesting questions about a software project such as "Which parts of the code are more prone to defects?," "Who is the best person to fix this bug?," or "Do any nocuous ambiguities exist in the requirements?". Recent significant advances in Software Engineering research and practice have delivered a slew of *software analytics solutions* that have the ability to answer such questions by retrieving and analyzing data from diverse software repositories. Among other things, techniques exist to compute the complexity of code, identify bug-prone methods, discover architecturally significant areas of the code, analyze test-case coverage, recommend experts across multiple phases of the development process, and identify ambiguous requirements.

Numerous researchers have reported the need to configure analytics functions within the context of a specific project. For example, in the area of software traceability, Falessi et al. [18] combined and evaluated several natural language processing techniques and concluded that techniques should be customized for each dataset. Similarly, Lohar et al. [33] demonstrated that trace accuracy could be increased when a trace engine is configured according to the project context. Herbold [26] built fault prediction models for 44 datasets using various configurations of EM clustering and nearest neighbor and found that models built within individual projects outperformed cross-project defect prediction models.

However, the practical cost and effort needed to customize a configuration in an industrial environment, including instrumenting the project with diverse tools and algorithms and conducting potentially long-running experiments, is often prohibitively expensive. Furthermore, especially in Greenfield projects, the data needed to configure analytic functions may be unavailable in early stages of the software development life-cycle. For example, configuring a trace engine for a specific project requires a set of source artifacts (such as requirements), target artifacts (such as source code), as well as a relatively complete set of trace links established between them. However, stakeholders may wish to use artifact connectivity solutions to generate trace links automatically, long before a sufficiently large set of validated links are available for training purposes. Similarly, developers may wish to use fault-prediction models in a Greenfield project, even before an initial set of reported faults is available.

A **Software Analytics cold-start** problem therefore exists when project stakeholders need to execute an analytic function in a new project environment, without the benefit of customizing the function's configuration. This differs from the better known cold-start phenomenon that occurs in systems that recommend products such as books, music, or news items to users [43]. Such recommender systems fall

MSR'16, May 14-15 2016, Austin, TX, USA
© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ISBN 978-1-4503-4186-8/16/05…$15.00
DOI: http://dx.doi.org/10.1145/2901739.2901740

loosely into two categories of *content-based* and *collaborative*, where content-based recommenders focus on textual descriptions of products and collaborative recommenders make recommendations based on the preference of nearest-neighbors (or similar users). Collaborative-filtering techniques often outperform content-based ones; however, they can only be employed after sufficient users have indicated their preferences for a product – leading to the cold-start problem [48, 32].

In this paper we address the problem of cold-start analytics by proposing and furthermore comparatively evaluating two different cold-start configuration solutions. While similar approaches have been used in the Recommender System community to recommend items such as books and music, to the best of our knowledge they have not previously been evaluated in the Software Engineering domain for recommending configurations for cold-start projects. Both of our solutions require the pre-construction of a **configuration profile** for each analytic function. Each row in the profile represents one project, depicts its unique project characteristics, and documents the best known configuration of the analytic function for that project. Our two proposed solutions leverage the configuration profile in different ways to recommend a configuration for a new, previously unseen, project without the need for project-specific customization.

• **Best-of-Breed** identifies the best overall performing configuration, across all projects in the profile, and adopts that default configuration for use in cold-start projects.

• **Profile-Driven** matches the project characteristics of a cold-start project against those in the configuration profile. It then selects the most similar project and adopts that project's configuration for the cold-start project.

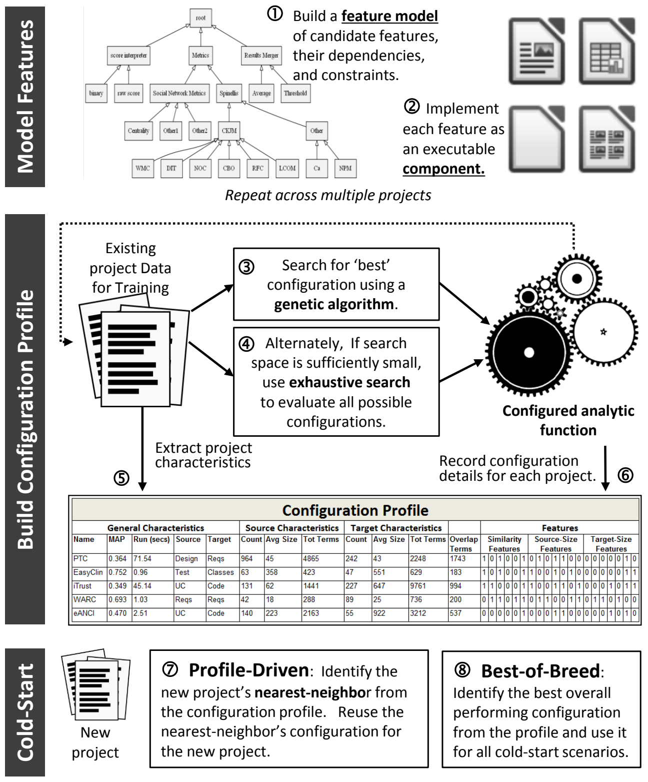
Our approach includes three major phases as depicted in Figure 1. The first phase consists of selecting, modeling, and implementing suitable features for the analytic functions, while the second involves building a configuration profile. Finally, the third phase uses the profile to recommend an actual configuration during a cold-start scenario.

To illustrate and evaluate our approach, we focus on three different analytic functions: (i) *artifact connectivity* (AC) between artifacts types – often referred to as 'automated traceability,' (ii) *prediction of fault prone* (FP) classes in Java source code, and finally (iii) *recommending experts* (XP). We selected these three functions because they cover both source code and text-based artifact types, address important software engineering tasks, and are well-described in the literature. Exploring three different functions allows us to answer some initial questions about generalizability.

The remainder of the paper is structured as follows. Section 2 describes the process of selecting, modeling, and implementing features. Section 3 describes our approach for building the configuration profile and then using it to select a configuration for a cold-start scenario. Section 4 describes experiments which were conducted to evaluate the best-of-breed versus profile-driven configurations. Finally, Sections 5, 6, and 7 present related work, threats to validity, and conclusions.

2. FEATURES

Analytic functions are constructed from primitive features, combined and parameterized in ways that aim to optimize qualities such as performance, accuracy, and usability. For


Model Features
root
score interpreter
Metrics
Results Merger
binary
raw score
Social Network Metrics
Spinellis
Average
Threshold
Centrality
Other1
Other2
CKJM
Other
WMC
DIT
NOC
CBO
RFC
LCOM
Ca
NPM
① Build a feature model of candidate features, their dependencies, and constraints.
② Implement each feature as an executable component.
Repeat across multiple projects
Build Configuration Profile
Existing project Data for Training
③ Search for 'best' configuration using a genetic algorithm.
④ Alternately, If search space is sufficiently small, use exhaustive search to evaluate all possible configurations.
Configured analytic function
Extract project characteristics ⑤
Record configuration details for each project. ⑥
Cold-Start
New project
⑦ Profile-Driven: Identify the new project's nearest-neighbor from the configuration profile. Reuse the nearest-neighbor's configuration for the new project.
⑧ Best-of-Breed: Identify the best overall performing configuration from the profile and use it for all cold-start scenarios.

Configuration Profile

General Characteristics					Source Characteristics			Target Characteristics				Features																			
Name	MAP	Run (secs)	Source	Target	Count	Avg Size	Tot Terms	Count	Avg Size	Tot Terms	Overlap Terms	Similarity Features						Source-Size Features							Target-Size Features						
PTC	0.364	71.54	Design	Reqs	964	45	4865	242	43	2248	1743	1	0	1	0	0	1	0	1	0	1	1	0	0	0	0	0	0	0	1	0
EasyClin	0.752	0.96	Test	Classes	63	358	423	47	551	629	183	1	0	1	0	0	1	1	0	0	1	0	1	0	0	0	0	0	0	1	1
iTrust	0.349	45.14	UC	Code	131	62	1441	227	647	9761	994	1	1	0	0	0	1	1	0	0	1	1	0	1	0	0	0	1	0	1	1
WARC	0.693	1.03	Reqs	Reqs	42	18	288	89	25	736	200	0	1	1	0	1	1	0	1	1	0	0	1	1	0	1	1	0	1	0	0
eANCI	0.470	2.51	UC	Code	140	223	2163	55	922	3212	537	0	0	0	0	0	1	0	0	0	1	1	0	0	0	0	0	1	0	1	0

Figure 1: An analytic function is trained and evaluated against a feature model of valid configurations.

example, researchers have explored diverse solutions for achieving artifact connectivity through dynamically generating trace links. Approaches have included the well known Vector Space Model (VSM) [25], Latent Semantic Indexing (LSI) [42], Latent Direchlet Allocation (LDA), natural language processing techniques, as well as various combinations. They may also include diverse pre-processing or post-processing steps to cleanse or manipulate the data and/or to organize or collate results. Similar types of choices exist for most analytic functions.

2.1 Feature Selection

As it is infeasible to include a complete set of viable features in our configuration process, we followed a systematic approach for selecting a diverse and representative set of features for each of the three analytic functions. First, we retrieved several recent publications in each relevant area (i.e., artifact connectivity [18, 24, 12], fault proneness [50, 28, 45], and recommend the expert [41, 13, 4]), and then identified a selection of seminal and state-of-the-art techniques for each function. Second, we searched for open-source solutions and/or affordable commercial products that (1) implemented the selected techniques and (2) could easily be integrated into our experimental environment. While we also built several components from scratch, we excluded features which we estimated would take more than one week to build and/or integrate.

For the *Fault Proneness* configurator for example, we included a range of fairly traditional coupling and cohesion (C&C) metrics [27, 6] as well as a set of metrics from network analysis [50]. We utilized two existing tools, CKJM

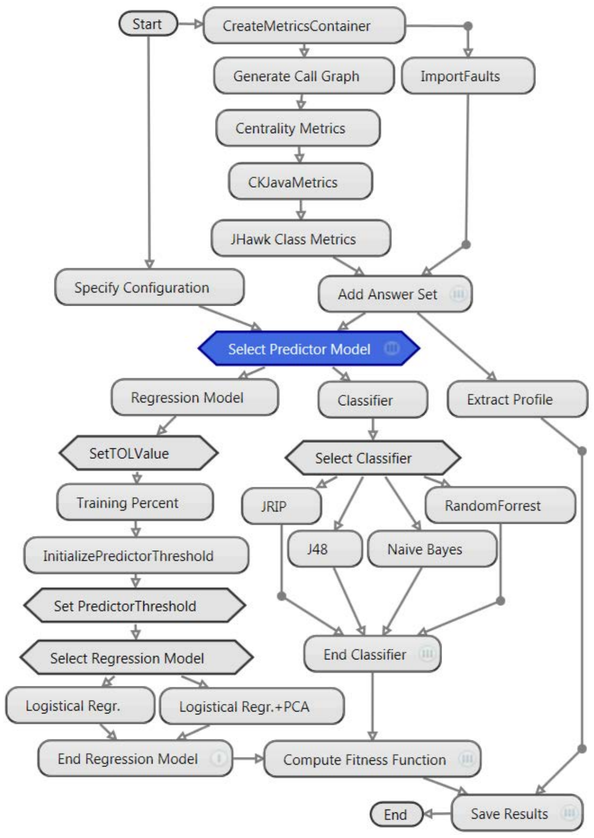


Figure 2: TraceLab workflow for predicting fault-prone code.

[11] and JHawk [1], to extract C&C metrics from Java code and then utilized JavaCallGraph and NetworkX to compute the network analysis metrics. For predicting purposes we selected several different techniques including various logistical regression models with and without Principal Component Analysis (PCA) and several well-known classifiers, namely: Naïve Bayes, Random Forest, J48, and JRip. A diverse set of features, as shown in Table 5 (Appendix A), were similarly selected for each of the other analytic functions. Our approach is scalable and allows additional features to easily be added.

2.2 Feature Modeling

In addition to listing available features, we need to model their dependencies and constraints so that we can explore the space of viable configurations. We used a textual notation, inspired by the TVL (Textual Version Language) [8], to construct a feature model for each analytic function. The model specifies the features, their parameters and associated value ranges, as well as constraints on the use of the feature including mandatory, optional, and prohibits relations. To illustrate, we present the feature model developed for the *Fault Proneness* function.

```
root SYS (allOf){
    Metrics,MainPredictor
}
Metrics (atLeastOneOf){
    CKJM, JHawk, NetworkAnalysis
}
MainPredictor (oneOf){
    WekaClassifier, LogisticsModel
}
LogisticsModel (allOf){
    LogisticalRegressionAlgorithm, FaultThreshold, TOLVal
}
LogisticalRegressionAlgorithm (oneOf){
    LogisticalPCA (Config:FaultThreshold, TOLVal),
    ProbitPCA (Config:FaultThreshold, TOLVal),
    LogisticalBasic (Config:Threshold, TOLVal)
}
Classifier (oneOf){
    J48, NaiveBayes, JRip, RandomForest
}
CKJM (atLeastOneOf){
    WMC,DIT,NOC,CBO,RFC,LCOM,Ca,NPF
}
NetworkAnalysis (atLeastOneOf){
    DegreeCentrality, InDegreeCentrality, LoadCentrality,
    OutDegreeCentrality, ClosenessCentrality, PageRank,
    BetweennessCentrality, EigenvectorCentrality
}
JHawk (atLeastOneOf){
    FOUT, LCOM2, MAXCC, TCC, WMC, NLOC, NOS, ODF
}
Configurations: {
    FaultThreshold {-0.5,0.3:0.1}
    TOLVal {10,90:10}
}
```

This feature model specifies that any solution configured to identify fault-prone code must contain a group of metrics and must contain a MainPredictor. The metrics may include any combination of the CKJM, JHawk, and/or network analysis metrics included in the listing. The predictor model may be based on a logistical regression model with or without Principal Component Analysis, or may be built using a classifier. For example, if the LogisticalRegressionAlgorithm is chosen to satisfy the 'one of LogisticsModel' constraint, then either LogisticalPCA, ProbitPCA, or LogisticalBasic could be selected. Furthermore, the feature model specifies ranges of values for the FaultThreshold and TOLVal used by the LogisticalRegressionAlgorithm.

2.3 Feature Implementation

To explore the performance of various configurations, all features need to be implemented in executable form. We chose to implement them as TraceLab components [30] because several useful components were already available in the TraceLab libraries [15] and also because TraceLab provides capabilities for interfacing with external DLLs which allowed us to seamlessly integrate our own home-built components with external tools. For each analytic function, we thus constructed a TraceLab workflow, capable of accommodating all valid combinations of features. For example, the workflow depicted in Figure 2 for the fault-prone function reads in datasets including source code and bug lists, executes a sequence of metric analyzers (java call graph, network analysis metrics, JHawk, and CKJM), and then uses these metrics to predict potentially *fault prone* code. It accommodates all combinations of features supported by the feature model.

3. THE CONFIGURATION PROFILE

The second phase of our process involves constructing a configuration profile which captures project characteristics and describes the way the analytic function was configured for the project.

Building the configuration profile requires a relatively large and diverse set of project data for each analytic function.

Table 1: Profile Characteristics for Each Configurator

Artifact Connectivity	**Fault Proneness**	**Find the Expert**
Source Artifact Type (Code/Text)	Cum. Halstead Length (HLTH)	Number of Issues
Target Artifact Type (Code/Text)	Max. Cycl. Complex. (MAXCC)	No. of Input Lines to Recommender
Number of Source Documents	Number of Stmts (NOS)	Number of Experts
Number of Target Documents	Lack of Cohes. of Methods (LCOM)	Number of Topics
Total Number of Terms in Source	Response Set for Class (RFC)	Sparsity of Profile
Total Number of Terms in Target	Fan In (FIN)	% of experts with only 1 Topic
Average Size of Source Documents	Depth of Inheritance Tree (DIT)	% of experts with at least 5 Topic
Average Size of Target Documents	Maint. Index (MI)	% of experts with at least 10 Topic
Number of Overlapping Terms	Total number of Classes (NOC)	Topic Type
	Total number of Comments (CCOM)	Type of Recommendation task

Table 2: Representative Entries Extracted from the "Artifact Connectivity" Configuration Profile

Name	MAP	Run	SRC	TGT	Source			Target			Over	Configuration																			
		(secs)	Type	Type	Cnt	Avg	Tot	Cnt	Avg	Tot	-lap	Similarity						Source-Side							Target-Side						
						Size	Terms		Size	Terms	Terms	Features						Features							Features						
PTC	0.364	71.54	Design	Reqs	964	45	4865	242	43	2248	1743	1	0	1	0	0	1	0	1	0	1	1	0	0	0	0	0	0	0	1	0
EasyClin	0.752	0.96	Test	Classes	63	358	423	47	551	629	183	1	0	1	0	0	1	1	0	0	1	0	1	0	0	0	0	0	0	1	1
iTrust	0.349	45.14	UC	Code	131	62	1441	227	647	9761	994	1	1	0	0	0	1	1	0	0	1	1	0	1	0	0	0	1	0	1	1
WARC	0.693	1.03	Reqs	Reqs	42	18	288	89	25	736	200	0	1	1	0	1	1	0	1	1	0	0	1	1	0	1	1	0	1	0	0
eANCI	0.470	2.51	UC	Code	140	223	2163	55	922	3212	537	0	0	0	0	0	1	0	0	0	1	1	0	0	0	0	0	1	0	1	0

Datasets related to *Fault Proneness* require source code as well as a list of related faults for each version of the software. *Find the Expert* datasets require records of tasks performed by different experts - for example, stakeholders who have contributed requirements knowledge or coders who have fixed bugs or approved pull requests in open source projects. Finally, *Artifact Connectivity* data requires related pairs of artifact types (e.g., requirements and regulatory codes, or use cases and source code) as well as a trace matrix depicting which pairs of artifacts are related to each other.

To acquire such data we used publicly available datasets wherever possible, scouring repositories such as PROMISE (openscience.us/repo), SIR (sir.unl.edu), and COEST (coest.org). We also mined additional datasets of faults following the technique described by Zimmerman et al. [49] and mined data concerning user tasks directly from open-source project repositories. Details of specific data sources are provided in Section 4.

We now describe the two main parts of the configurator, namely the project profile (shown on the left hand side of Table 2) and the configuration (shown on the right hand side of the same table).

3.1 Project Characteristics

Each analytic function has its own configuration profile and its own unique set of project characteristics. The pertinent question is which set of project characteristics should be included in a configuration profile in order to effectively differentiate one project from another. This issue is further complicated by the fact that different differentiators are needed for each analytic function. Therefore, project characteristics differ across each of our three configuration profiles.

To identify pertinent characteristics for each analytic function we followed the following three selection strategies: (1) our own knowledge of the analytic area, (2) clues found in the literature, and (3) discussions with experts in each of the fields e.g., we found a general belief that LSI performs better on large datasets therefore "size may matter" for the artifact connectivity function. The project characteristics identified for each analytic function are shown in Table 1. We implemented and/or integrated a set of tools capable of extracting project characteristics from each project so that we could populate the configuration profiles.

3.2 Customizing the configuration

Identifying the best performing configuration for an existing project dataset requires searching through the space of valid feature combinations and parameterizations and evaluating the quality of each configuration using a fitness function.

3.2.1 Fitness Functions

There are many plausible fitness functions that could be used for each analytic function. We decided to select a well-known and well-accepted metric for each one. For example, in the case of Artifact Connectivity, fitness could be evaluated using a well accepted metric such as Mean Average Precision (MAP), Receiver Operating Characteristic (ROC) which measures the area under the curve when true positives are plotted against false positives, or the F2-Measure which represents the weighted harmonic mean of recall and precision [44]. We opted to use MAP which evaluates the extent to which the complete set of targeted links is placed at the top of a ranked list of links. Fitness functions could incorporate multiple objectives – for example evaluating not only the quality of the links produced, but also the execution time needed to generate them. For purposes of this paper we focused on single objective functions: MAP for *Artifact Connectivity*, Mean Reciprocal Rank (MRR) for *Find the Expert*, and F2-Measure for *Fault Proneness.* We also used lowest runtime as a tie-breaker when two configurations returned the same score. We selected MAP because it provides a quality measure across all recall levels, is widely used in Information Retrieval settings, and has been frequently used to evaluate trace retrieval results. We selected MRR because it is widely used in settings where the user only wants to see one relevant document retrieved from a ranked list of results. We selected F2 because it weights recall higher than precision and is used in settings where it is more important to not make Type II errors. Formulas and associated references are provided for each metric in Appendix A.

3.2.2 *Search Techniques*

We adopted two different strategies for searching through the space of viable configurations. In the case of the Find the Expert analytic function, the search-space of features was sufficiently small to run an **exhaustive search** through all combinations of features.

In the case of Fault Proneness and Artifact Connectivity, the configuration space was too large to reasonably run an exhaustive search and we therefore used a Genetic Algorithm (GA) to search for the top performing configuration(s). Several researchers have shown this to be effective for configuring features – especially in the area of artifact connectivity [33, 40]. The GA is a good match because each candidate trace configuration can be encoded in a chromosome represented as a string of bits where each bit represents a different feature that is either present (1) or not present (0) in the configuration. To create an initial population, we randomly generated chromosomes by turning bits on and off, and checked them against the feature model for correctness until an initial population of 50 unique, valid configurations was produced.

The fitness of each chromosome was then computed by instantiating its configuration and using the fitness function to evaluate how well it performed its intended task. A stochastic process was used to select the best chromosomes to be carried forward as parents into the next generation. Our implementation carried ten chromosomes forward. One of these was the *elite* chromosome, i.e., the chromosome with the highest fitness score from the previous generation. The remaining nine chromosomes were selected using a standard practice based on the *roulette wheel*, which works on the premise that chromosomes scoring higher fitness values have a greater chance of survival than weaker ones. The next generation of chromosomes was then derived using standard techniques of *cross-over* (0.1) and *mutation* (0.5) to generate offspring chromosomes from the parents. This process is described in detail in our prior work [33]. For purposes of our experiment we ran the GA for fifty generations. Figure 3 plots runtime in milliseconds versus the appropriate fitness scores achieved for various configurations of the artifact connectivity function applied against a single dataset. Because the GA converges on faster scores over time, more data points are shown at higher levels of MAP than at lower ones. Ties were again resolved by selecting the configuration with the lowest runtime.

The configuration identified by either exhaustive search or the genetic algorithm is referred to throughout the remainder of this paper as the **customized configuration**. We use it in two places, first within the configuration profile and secondly to comparatively evaluate the efficacy of the profile-driven and best-of-breed approaches.

3.3 Cold Start

Both of our cold-start solutions leverage the configuration profile.

• **Best-of-Breed** To identify the best-of-breed configuration for each analytic function, each configuration in the profile was used to perform the analytics function on each of the other projects in the profile. Metrics were computed for each function (i.e., mean MAP, F2-Measure, and MMR). The configuration returning the highest mean score was considered the best-of-breed for that analytic function. Using this approach, all cold-start projects adopt the same configuration.

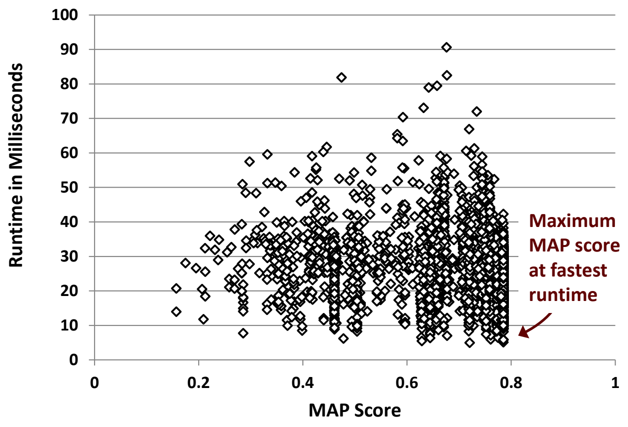


Figure 3: Runtime versus MAP output by the Genetic Algorithm during a search for the best software Artifact Connectivity configurations.

• **Profile-Driven** The profile-driven approach is dependent upon correctly matching a cold-start project to its most similar 'neighbor.' The data of the cold-start project is parsed in order to extract its project characteristics. The similarity of the cold-start project CS with an existing project P in the configuration profile is computed by representing each project as a vector of characteristics, and then computing the cosine of the angle between the two vectors as follows:

$$Similarity = \frac{\Sigma_{i=1}^{N}(CS_i \times W_i) \times (P_i \times W_i)}{\sqrt{\Sigma_{i=1}^{N}(CS_i \times W_i)^2 \times \Sigma_{i=1}^{N}(P_i \times W_i)^2}} \quad (1a)$$

where CS_i and P_i represent the scores ascribed to project CS and P in dimension i, W_i is a weighting assigned to dimension i, and N is the total number of dimensions. The project in the configuration profile exhibiting the highest similarity score to CS is selected as its nearest-neighbor, and its configuration used for the CS project. If two projects in the configuration profile are deemed to exhibit equal similarity to the cold-start project, then the configuration with the fastest runtime is selected.

4. EVALUATION

In this paper we have proposed two cold-start configuration solutions: best-of-breed and profile-driven. We also used a third search-based approach to identify the customized-configuration for each dataset. In this section we describe a series of experiments that were conducted to evaluate whether best-of-breed or profile-driven performed best as a cold-start solution. We also investigate whether the cold-start solutions returned results which were similar in quality to those achieved using customized-configurations.

4.1 Experimental Design

We first constructed configuration profiles for each analytic function. Each configuration profile was composed

Table 3: Datasets used to Construct Configurators

Artifact Connectivity				
Project	*Source*	*Target*	*Links*	*Src*
Pos Train (AC1)	Reqs (964)	Design (242)	6961	Ind
Pos Train (AC2)	Reqs (39)	SubDes (241)	40	Ind
iTrust (AC3)	Reqs (131)	Code (227)	398	C
HIPAA (AC4)	Safeguards(10)	Reqs(1064)	78	C
HIPAA(AC5)	Safeguards(10)	Reqs(100)	31	C
WARC1(AC6)	Func.Reqs(42)	Sys.Reqs(89)	78	C
WARC2(AC7)	NFRs(21)	Reqs(59)	58	C
Easy-Clinic(AC8)	TestCase (63)	Classes(47)	204	C
English (AC9)	UseCases (30)	Class(47)	93	C
Italian (AC10)	UseCase (30)	Classes(47)	93	C
SMOS(AC11)	Reqs (67)	Code(100)	1044	C
CM1 (AC12)	Reqs(1064)	Des(116)	587	C
Event Trace (AC13)	Reqs(41)	Classes(50)	98	C
EBT2(AC14)	Reqs(41)	TestCase(25)	39	C
Med Pump (AC15)	Component(21)	Req(126)	131	C
Intel. Kiosk (AC16)	Reqs(167)	Process (167)	1847	LM
GANNT(AC17)	Reqs(17)	Design(69)	68	M
Albergate(AC18)	Reqs(17)	Code(55)	54	M
eANCI(AC19)	UseCase(140)	Code(55)	567	M
IceBreaker(AC20)	UseCase(201)	Class(73)	457	M

Data source: C=COEST, Ind = Industry, R=Miscellaneous

Fault Proneness		
Project	*Versions (No. Classes, No. Classes with Bugs)*	*Src*
Ant (FP1-FP7)	1.6.0(669,19); 1.6.5(669,17); 1.7.0(799,75); 1.7.1(598,24); 1.8.1(674,16); 1.8.2(845,35); 1.9.1(1080,35)	BZ
Batick-Jira (FP8)	1.8(1676,137)	BZ
JMeter (FP9-FP11)	2.2(365,31); 2.3.2(352,15); 2.4(734,33)	BZ
POI (FP12-FP15)	3.0(1080,43); 3.8(1099,21); 3.9(1115,53); 3.10(1477,30)	BZ
Tomcat (FP16-FP19)	6.0.20(793,19); 6.0.29(1100,19); 7.0.26(2201,70); 7.0.27(1819,27)	BZ
Velocity (FP20)	1.4(217,145)	PR[R

Data Source: BZ=Bugzilla, PR=Promise repository.

Find the Expert							
Project	*Issues*	*Lines*	*Type*	*Exp*	*Top*	*Spars.*	*Src*
Sugar CRM (XP1)	523	330	B	58	60	5.5	SG
(XP2)	6880	4955	V	2422	50	99.1	
Railway Sys. (XP3)	137	1633	B	118	54	30.2	*
(XP4)	1031	826	V	118	54	16.2	
Second Life (XP5)	2120	1880	B	273	50	37.6	SL
Apache (XP6)	3309	1291	V	322	64	20.17	BZ
Tomcat 4 (XP7)	3304	2197	CB	97	50	43.94	BZ
(XP8)	3304	2197	CB	97	20	67.09	
Tomcat 5 (XP9)	2384	444	V	151	18	24.67	BZ
FOP-Jira (XP10)	1209	2238	V	81	10	23.80	BZ
Ant (XP11)	4384	2352	CB	90	50	47.04	BZ
(XP12)	4384	1372	CB	90	20	68.60	
Tomcat (XP13)	381	838	CB	28	50	16.76	BZ
Connectors (XP14)	381	451	CB	28	20	22.55	
Lenya (XP15)	870	287	V	34	24	11.96	BZ
Guava (XP16)	330	285	V	93	21	13.57	GH
Zurb (XP17)	446	514	V	413	11	46.72	GH
Junit (XP18)	388	1023	CB	53	50	20.46	GH
(XP19)	388	697	CB	53	20	34.85	
JHipster (XP20)	458	707	CB	53	20	35.35	GH

Source: BZ=Bugzilla, GH=Github. SG=SugarCRM, SL=SecondLife, *=multiple sources

from the data of 20 projects as described in Table 3. For Artifact Connectivity, twelve of the project datasets were acquired from the COEST.org library with additional proprietary datasets made available by our industrial partners and research colleagues. The datasets ranged in size and included trace links between diverse artifacts such as requirements, regulatory safeguards, use cases, test cases, and component descriptions. The Fault Proneness datasets included different versions of Ant, Batack-Jira, POI, TomCat, and Velocity acquired primarily from Bugzilla or the PROMISE repository. We were unable to use many of the PROMISE datasets because the provided pre-computed metrics were insufficient for our experiments and it was difficult to retrieve original versions of the source code. Finally, the meta-profile for Find the Expert was constructed from five sets of feature requests (containing requirements and users) and fifteen sets of bug assignments (consisting of bug reports with associated fixes as well as the names of developers assigned to fix the bugs).

A new metric, which we refer to as **PercentOfAchieved-Maximum** (PAM), was used to comparatively evaluate techniques. As its name suggests, it takes the highest score discovered using the exhaustive search (Find the Expert) or the genetic algorithm search (Fault Proneness and Artifact Connectivity) for a specific dataset and analytic function. It then computes the percentage of that score achieved by a given configuration C. For example, if the highest MAP score achieved for artifact connectivity in a specific project were 0.550, and configuration C returned 0.359, then the PAM score of C would be $\equiv \frac{.359}{.550} = .653$.

We used a leave-one-out approach to evaluate both techniques. Given a configuration profile composed of 20 projects, we set one project aside to serve as the cold-start project during each iteration. The remaining 19 projects were retained in the configuration profile. The best-of-breed and profile-driven configurations were identified for the cold-start project following the techniques described in Section 3. The recommended configuration was then used to perform the respective task in the cold-start project and MAP, MMR, and F2 metrics computed as appropriate for each task. We repeated this process for each of the three analytic tasks until all 20 projects from the initial configuration profile had been tested as a cold-start project.

4.2 Results

We evaluate results for the best-of-breed and profile-driven techniques and compare them to the results achieved using the customized-configurations.

4.2.1 *Profile-Driven Approach*

In the profile-driven approach, the cold-start project is matched against the profile by computing the cosine similarity of project characteristics. We evaluated two alternate options for identifying the nearest neighbor: *non-weighted* – in which each project characteristic was assigned equal importance, and *weighted* – in which we applied a two-layered leave-one-out experimental process to customize weights.

Weightings were learned according to the following process. In each run of the experiment, 19 projects were placed into the training set and one was set-aside as a cold-start project (CS^*) for evaluation purposes. Within the training set of 19 projects we performed a leave-one-out approach in which the configuration profile was constructed from 18

Table 4: Average PAM scores for Three Analytic Functions

Analytic Function	Best-of-Breed	Profile-Driven	
		With Weights	Without Weights
Artifact Connectivity	94.33%	91.21%	87.66%
Find the Expert	95.44%	80.87%	82.70%
Fault Proneness	69.28%	41.52%	50.60%

projects and the remaining project played the role of a cold-start project (CS^+). We then systematically applied multiple weighting configurations for matching CS^+ to the 18 projects in the configuration profile. Weights ranged from 0 to 1 at intervals of 0.1 for each characteristic and all possible combination of weights were used to match CS^+ to its closest neighbor in order to recommend a configuration. The weighting that produced the best result was recorded in each case. The process was repeated until each of the 19 datasets had served as the cold-start project, and the 19 weightings were then averaged. The averaged weights were then used to find the nearest neighbor of the 20th project (i.e., CS^*) in order to recommend a configuration. This configuration was used to execute the analytic function and metrics (MAP, MMR, or F2) results were recorded. This entire process was repeated 20 times until each project had played the role of CS^*.

For Artifact Connectivity, in 18 out of 20 cases, the best result was achieved when only a single dimension in the reduced space was used for matching purposes (i.e., weights were set to 1 for this dimension, and 0 for all others). The primary characteristics contributing to this dimension are *Average Size of Source Documents* 30.16%, *Number of Terms in Source Document* 29.09%, and *Number of Overlapping Terms* 27.73%. In the case of Find the Expert and Fault Proneness, we did not identify any form of reoccurring dominant dimensions.

In Table 4 we compare PAM scores for weighted versus unweighted approaches. In the case of Artifact Connectivity, the weighted approach improved the quality of the configuration returning PAM of 91.21% versus 87.99% without weightings. However, in the other two cases, *without weighting* outperformed *weighting.* Fault Proneness returned PAM of 69.28% without weighting and only 41.52% with weighting, while Find the Expert returned 82.87% without weighting and only 80.87% with weighting. Overall, the non-weighted approach performed best and we therefore adopt it throughout the remainder of the paper.

4.2.2 Best-of-Breed Approach

For the best-of-breed approach, we again used a leave-one-out approach in which 19 projects were used to discover the best overall approach and then applied to the left-out (cold-start) project. Results from this experiment are reported. However, to provide insights into our approach we discuss the best overall configuration as learned from all 20 projects – as this configuration would be used for future datasets. The winning configuration for Artifact Connectivity included a Semeru source-side stopper and splitter, a target-side stemmer, and a voting mechanism (Majority Rules) to integrate results from VSM smoothing and standard tf-idf techniques. For Fault Proneness, the winner included the Logistical regression model with PCA (FaultThreshold 0.2, TOL value 90), response set for class, weighted methods per class, lack of cohesion in methods, total cyclomatic complexity, average complexity of all methods, cohesion, closeness centrality, and eigenvector centrality. For Find the Expert, the overall winner incorporated a boolean recommender and used the tanimoto similarity and a threshold neighborhood.

4.3 Analysis of Results

To determine whether the best-of-breed or profile-driven approach performed best, we computed PAM scores from the experiment. We used the non-weighted version of the profile-driven approach. The distribution of these scores was plotted in Figure 4 as box plots. The PAM scores for profile-driven configurations and best-of-breed configurations are depicted as blue Xs and red squares, respectively.

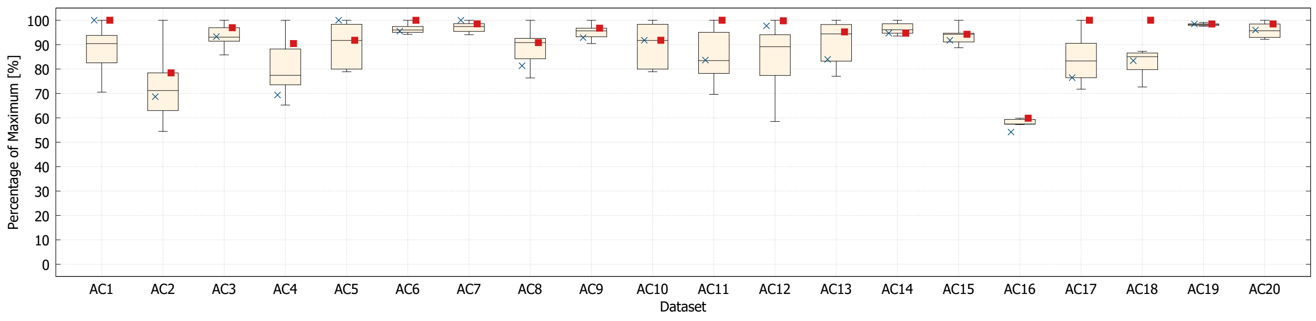
From observing these box plots, and the PAM scores reported in Table 4, it is immediately apparent that the best-of-breed configuration outperformed the profile-driven approach in most cases. We then performed a series of statistical tests to determine whether the differences were statistically significant. Because the data is not normally distributed (per the Shapiro-Wilk test), variances were potentially unequal, and sample observations overlap, we utilized the Wilcoxon Signed Rank test to compare results. For Artifact Connectivity and Fault Proneness, there is a statistically significant difference between the medians of the two groups at confidence of 0.05 (AC:$z-value = -3.06$, $p-value = 0.002$) (FP:$z-value = -2.45$, $p-value = 0.014$). In the case of Find the Expert, the difference is not significant (XP:$z-value = -1.89$, $p-value = 0.059$).

Based on these results we conclude that the best-of-breed approach outperformed the profile-driven approach in each of the three analytics areas, although differences are not statistically significant in the case of Find the Expert.

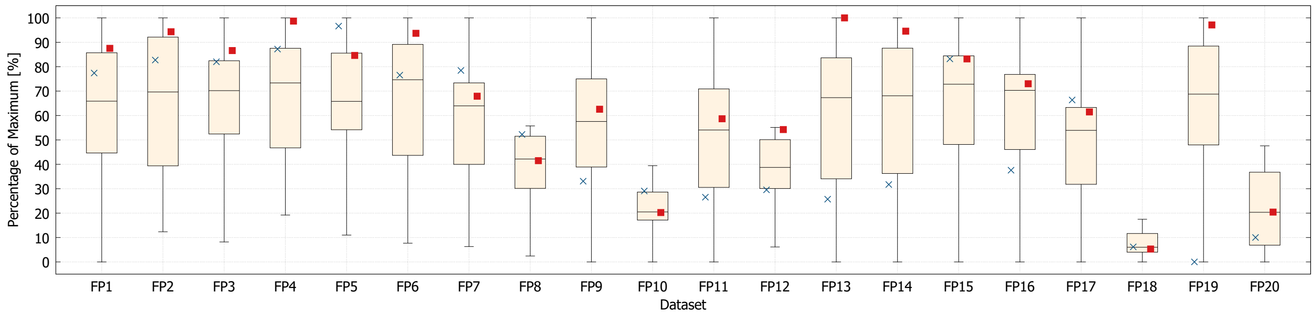
We also evaluated how well each technique performed in comparison to the **customized configuration**. From Table 4 we see that the best-of-breed solution returned an average of 94.33% of the customized-configuration scores for Artifact Connectivity and 95.44% for Find the Expert. However, in the case of Fault Proneness the best of breed returned an average of only 69.28% of the customized-configuration PAM score suggesting that neither of our solutions adequately solve the cold-start solution across all areas of software analytics. Insights into the difference in results may be partially achieved by analyzing the range of PAM scores depicted in the box plots of Figure 4 for each of the three analytic functions. The range is far greater for Fault Proneness than for the other analytic functions – suggesting that applying the wrong configuration will have more significant impact upon the quality of the prediction. However, we also observe that in Find the Expert projects XP4, XP5, XP16, and XP17 for which PAM scores covered a large range of the spectrum, the best of breed still performed close to the maximum.

Given these results, we conclude that in two cases of Artifact Connectivity and Find the Expert, the cold-start solution proposed in this paper is relatively effective; however, in the case of Fault Proneness neither of our proposed solutions effectively solved the cold-start problem.

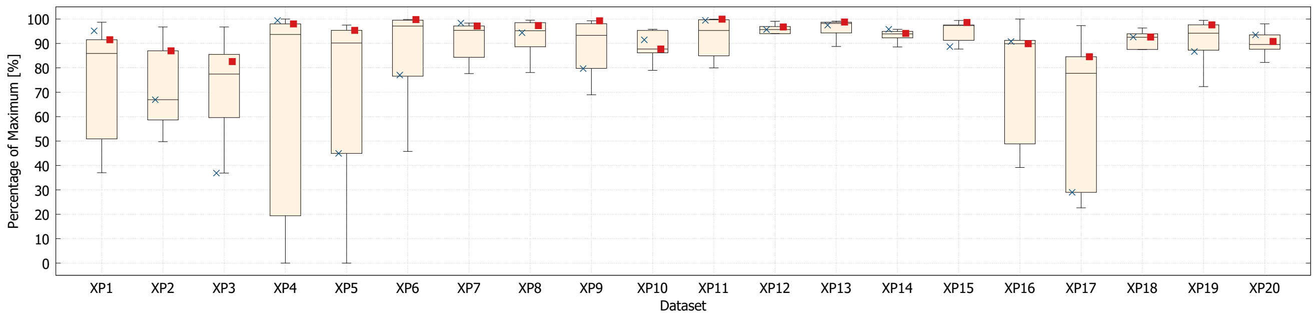
Intuitively it makes sense that a profile-driven approach should outperform a best-of-breed approach. We therefore discuss reasons that the profile-driven approach may have underperformed in our experiments. First, it is possible that the identified project characteristics (cf. Table 1) did



(a) Artifact Connectivity



(b) Fault Proneness



(c) Find the Expert

Figure 4: Percentage of Achieved Maximum (PAM) for 🟥 best-of-breed and **X** profile-driven configurations.

not adequately capture the differentiators which drive feature selection. However, the characteristics *seemed reasonable*, and additional characteristics we initially proposed exhibited high degrees of correlation with the set we finally evaluated. A second plausible explanation is that the configuration profiles were too small to perform well and that significantly increasing the number and diversity of projects in each profile would allow closer matches to be made. However, we note the very significant time and effort that went into building profiles containing even 20 projects, and the practicality barrier that exists in extending the profiles to hundreds or even thousands of datasets.

5. RELATED WORK

Much of the related work in this area has focused on demonstrating that analytic functions should be customized for each project environment.

There is a large body of work in each of the three analytic areas. We therefore provide only a brief summary here. In the area of traceability (i.e., artifact connectivity), researchers have proposed numerous diverse techniques [15]. The most similar work comes in the area of integrating tracing solutions. Biggers et al. [7] explored different configurations of a Latent Dirichlet Allocation (LDA)-based Feature Location Technique (FLT). Panichella et al. [40] used a Genetic Algorithm (GA) to adapt and configure LDA solutions and showed that it led to higher accuracy than previously non-configured results. Falessi, Cantone, and Canfora [18] combined and evaluated several natural language processing techniques for identifying associations between artifacts. They concluded that techniques should be customized for the artifacts. Lohar et al. [33] demonstrated that trace accuracy increased if a trace engine were configured according to the project context. Finally, Dietrich et al. [14] and Moreno et al. [37], have all explored customization at the query level. Together, this body of work highlights the importance of customizing a tracing solution; however, at the dataset level, prior work has not addressed the cold-start configuration problem that we address in this paper.

There is a similarly large body of work in the area of fault prediction. For the most part, the work highlights the diversity of candidate techniques. For example, Canfora and Oliveto [9] proposed cross-project defect prediction using a multi-objective logistic regression model built using a genetic algorithm. Jiang et al. [29] showed that integrating requirements-based textual metrics with code-based ones improved predictive ability. Singh et al. [45] compared logistic regression with multiple machine learning methods for predicting fault-prone code. Ostrand and Weyuker [39] applied a statistical model based on historical fault information and file characteristics to predict faults at the file level. Zimmermann et al. [51] built a model using mapped defects from the Eclipse bug database combined with source code metrics to predict bug-prone modules. One of our regression modeling features is closely built upon their approach. Elish and Elish [17] showed that support vector machines can outperform statistical and other machine learning techniques. Others reported success with random forests [20], statistical regression models [38], and change management data [19]. Each technique requires different tools and different data, and all of them would be candidates for modeling as available features and for integration into our software analytics configurator.

Finally, several researchers have explored the use of recommender systems in the software engineering domain. One primary focus has been on identifying people to fix bugs. Researchers such as Cubranic and Murphy [13], Ahsan et al. [3], and Anvik et al. [4] have trained classifiers to identify associations between bug descriptions and assigned developers. Mockus and Herbsleb [36] directly analyzed and used developers' source code changes to identify experts for various parts of the system. Other researchers have focused on recommending domain experts during the requirements engineering process. Castro-Herrera et al. [10] explored a variety of content-based and collaborative-filtering recommender systems for identifying domain experts based on their contribution to online discussion forums. Maalej et al. [21] used sentiment analysis to identify users interested in specific features. These techniques provided the inspiration for the techniques we incorporated into our 'Find the Expert' configurator.

In a more general sense, Thornton et al. [47] address the combined algorithm selection and hyperparameter optimization (CASH) problem to assist non-expert users using WEKA to select the best machine learning algorithm and attendant hyperparameters. Our approach addresses the cold-start problem using three different analytic functions in the software engineering domain, while Thornton et al. examine the ability to learn parameters and the best machine learning approach for a large collection of unrelated datasets. Our work uses objective/fitness functions tailored to the task versus more generic measures.

6. THREATS TO VALIDITY

External validity evaluates the generalizability of the approach. To address this threat we applied our approach to three different analytic functions. We expect that the lessons learned will be applicable to other types of functions, but broader analysis is necessary to understand the scope and constraints of our claims. Furthermore, we incorporated 20 relatively distinct datasets for each function; however, the challenge of collecting so much data meant that our choices were limited. Nevertheless, the collected data was evaluated and shown to be quite diverse.

Construct validity evaluates the degree to which the claims were correctly measured. For each of the analytic functions we identified a commonly used metric which has been broadly adopted in the domain. This metric was used to measure the quality of an individual configuration. Furthermore, we opted to compare configurations according to the percentage of the maximum metric score they were able to achieve. This allowed us to compare top-ranking configurations against each other but with a comparative, rather than an absolute, measure of success. We justify this because our goal was to compare the performance of different configurations.

Internal validity reflects the extent to which a study minimizes systematic error or bias, so that a causal conclusion can be drawn. We attempted to reduce bias by using a diverse and relatively extensive set of data samples for each of the three analytic areas. However, constructing or acquiring such datasets can be extremely time-consuming and challenging and therefore we were limited to 20 datasets per analytic function. Nevertheless, we believe this to be the largest number of datasets used so far for investigating any of the three areas of Artifact Configuration, Fault Proneness, or Find the Expert.

7. CONCLUSION

In this paper we addressed the cold-start software analytics problem by proposing two different configuration techniques: a 'best-of-breed' and a 'profile-driven' approach. We focused on the three different analytic areas of artifact connectivity, fault-prediction, and finding the expert to evaluate the approaches. We have determined that in computing profile similarity, considering all of the features is preferrable to focusing on the dominant ones. Our results have furthermore shown that the best-of-breed approach outperformed the profile-driven approach; however, it returned acceptable results for only two of the three analytic functions.

As previously discussed, it is plausible that the profile-driven approach might perform better if we could collect additional data sources and construct larger and more diverse configuration profiles. As techniques for mining datasets within the Software Engineering community continue to improve through the advent of tools such as BOA [16], such an approach may be viable. However, we believe that including additional characteristics is likely to result in small marginal performance improvement. Similarly, we expect that noticeable improvement from new sources (if possible) would require a very large number of new datasets.

We conclude by pointing out that cold-start solutions are only intended for use in early phases of a project and have natural performance limitations. As sufficient project-specific data is accumulated, the analytic function should be retrained on its own data and the cold-start configuration replaced with a customized configuration.

Acknowledgements

This work was partially funded by the US National Science Foundation Grants CCF-1319680 and CCF-1511117, and by the Christian Doppler Forschungsgesellschaft, Austria. We thank Clinton Woodson and Wei Wen for assistance in researching Fault Prediction classification. We thank the numerous researchers who have contributed datasets to the PROMISE, COEST and SIRS repositories. Without their generosity, this type of meta-level analysis would be impossible.

Table 5: Appendix A: A Summary of Features used in each Configurator

Artifact Connectivity (aka Just-in-time Traceability)		
Goal:	To leverage information retrieval techniques to generate trace links between artifacts (e.g., between requirements and design artifacts), Functions are composed from preprocessors, similarity computations, post-processors, etc.	
Objective:	Maximize Mean Average Precision (MAP): $MAP = \sum_{q=1}^{Q} AP(q)/Q$ where Q = total number of queries and AP = Average Precision per query (more)	
Preprocessors	Stemmers (3): Reduces each word to its morphological root.	TraceLab Lib.
	Stoppers (3): Removes common words.	TraceLab Lib.
	Splitters (2): Splits compound terms, e.g., camelCase style variable names.	TraceLab Lib.
	Character cleanser (2): Removes certain characters from the text.	TraceLab Lib.
Similarity Computations	Latent Semantic Indexing (LSI): Identifies latent topics and leverages these topics to identify similarities. Configured by number of latent topics.	BlueBit Matrix lib.(http://bluebit.gr)
	Vector Space Model (VSM): Represents each document as a vector of terms. Utilizes various formulas to compute similarity between vectors, e.g., Cosine, Jacard, SimpleCount.	Existing TraceLab Component
Post-Proc.	Voter: Merges results from multiple similarity computations.	New component

Fault Proneness		
Goal	To identify classes which are likely to exhibit faults in the future. Numerous publications have proposed a wide variety of solutions including predictions based on coupling and cohesion metrics [28, 29], network analysis metrics [50], bug and change history [31], etc.	
Objective:	F-measure:the harmonic mean of precision and recall. $F = \frac{2 \cdot precision \cdot recall}{precision + recall}$.	
Metrics	CKJM Metrics: Traditional Chidamber and Kemerer metrics [11, 46]	www.spinellis.gr /sw/ckjm/
	Jhawk Metrics: A commercial tool providing 105 system, package, class, and method level metrics, e.g., Cyclomatic Complexity[34], Halstead measures [23]	virtualmachinery.com/ jhawkprod.htm [1]
	Network Analysis Metrics: DegreeCentrality, InDegreeCentrality, LoadCentrality, OutDegreeCentrality, ClosenessCentrality, PageRank, BetweennessCentrality, EigenvectorCentrality	https://networkx. github.io/
Regression Models	Logistical regression: (i) Normalized data, (ii) Probit and Logit models, (iii) with/without PCA, (iv) Prediction thresholds (range: -0.5-0.3), (v)TOLValues (range: 10%-90%)	R Statistics Package [2]
Classifier	Naïve Bayes: Probabilistic classifier based on independence assumption between predictors.	Weka [22]
	JRIP: Implements RIPPER rule learner that builds a ruleset by repeatedly adding rules to an empty ruleset until all positive examples are covered.	Weka [22]
	J48: Implements the C4.5 decision tree algorithm, breaks the data into smaller subsets and uses information gain to determine attribute for splitting data.	Weka [22]
	Random Forest: An ensemble learning method that constructs many different decision trees during training, and during classification, outputs the class that represents the mode of the runtime decisions.	Weka [22]

Find The Expert		
Goal	To identify an expert in a specific area, for example, somebody who has the skills to fix a specific bug or is a domain expert in a specific requirements topic.	
Objective:	Maximize Mean Reciprocal Rank, $MRR = 1/n \sum_{i=1}^{n} 1/rank(i)$, where n is the number of queries, and rank(i) the rank of the item within the list of elements proposed by the recommender	
Preprocessors	Topic Modeling: Mallet is a Java-based tool for natural language processing, topic modeling, and other machine learning applications to text.	Mallet [35]
Recommender System	Recommender: Apache Mahout 0.10.2 [5], NReco C# Recommendation Engine, Boolean Recommender, User-based	http://mahout.apache. org, www.nrecosite.com
	Topic Similarity: Spearman-Correlation, PearsonCorrelation, UncenteredCosine, CityBlock Similarity, TanimotoCoefficient	
	Neighborhood: Nearest-Neighbor, Nearest-Neighbor Threshold	

8. REFERENCES

[1] JHawk Metrics Tool, http://www.virtualmachinery.com/jhawkprod.htm, 2015.

[2] The R Project for Staistical Computing, https://www.r-project.org/, 2015.

[3] S. N. Ahsan, J. Ferzund, and F. Wotawa. Automatic software bug triage system (bts) based on latent semantic indexing and support vector machine. In *Proc. of the 4th Int'l Conf. on Software Engineering Advances, ICSEA'09*, pages 216–221. IEEE, 2009.

[4] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *Proc. of the 28th Int'l Conf. on Software Engineering*, ICSE '06, pages 361–370, New York, USA, 2006. ACM.

[5] Apache Mahout, http://mahout.apache.org. 2015.

[6] V. R. Basili, L. C. Briand, and W. L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Trans. Software Eng.*, 22(10):751–761, 1996.

[7] L. R. Biggers, C. Bocovich, R. Capshaw, B. P. Eddy, L. H. Etzkorn, and N. A. Kraft. Configuring latent dirichlet allocation based feature location. *Empirical Software Engineering*, 19(3):465–500, 2014.

[8] Q. Boucher, A. Classen, P. Faber, and P. Heymans. Introducing TVL, a text-based feature modelling language. In *Proc. of the 4th Int'l WS. on Variability Modelling of Software-Intensive Systems (VaMoS'10)*, pages 159–162, 2010.

[9] G. Canfora, A. De Lucia, M. Di Penta, R. Oliveto, A. Panichella, and S. Panichella. Multi-objective cross-project defect prediction. In *Proc. of the 6th Int'l Conf. on Software Testing, Verification and Validation (ICST)*, pages 252–261. IEEE, 2013.

[10] C. Castro-Herrera, J. Cleland-Huang, and B. Mobasher. Enhancing stakeholder profiles to improve recommendations in online requirements elicitation. In *Proc. of the 17th IEEE Int'l Requirements Engineering Conf., RE 2009*, pages 37–46, 2009.

[11] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. on Software Eng.*, 20(6):476–493, 1994.

[12] J. Cleland-Huang, O. Gotel, J. H. Hayes, P. Mäder, and A. Zisman. Software traceability: trends and future directions. In *Proc. of the on Future of Software Engineering, FOSE 2014*, pages 55–69, 2014.

[13] D. Čubranić. Automatic bug triage using text categorization. In *Proc. of the 16th Int'l Conf. on Software Engineering & Knowledge Engineering (SEKE 2004)*. KSI Press, 2004.

[14] T. Dietrich and J. Cleland-Huang. Seeking better queries. In *North American Search Based Software Engineering.*

[15] B. Dit, E. Moritz, M. L. Vásquez, and D. Poshyvanyk. Supporting and accelerating reproducible research in software maintenance using tracelab component library. In *Proc. of the 2013 IEEE Int'l Conf. on Software Maintenance (ICSM)*, pages 330–339, 2013.

[16] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *Proc. of the 35th Int'l Conf. on Software Engineering*, ICSE'13, pages 422–431, May 2013.

[17] K. O. Elish and M. O. Elish. Predicting defect-prone software modules using support vector machines. *Journal of Systems and Software*, 81(5):649–660, 2008.

[18] D. Falessi, G. Cantone, and G. Canfora. Empirical principles and an industrial casestudy in retrieving equivalent requirements via natural language processing techniques. *IEEE Trans. on Software Engineering*, 39(1):18–44, 2013.

[19] T. Graves, A. Karr, J. Marron, and H. Siy. Predicting fault incidence using software change history. *IEEE Trans. on Software Engineering*, 26(7):653–661, Jul 2000.

[20] L. Guo, Y. Ma, B. Cukic, and H. Singh. Robust prediction of fault-proneness by random forests. In *Proc. of the 15th Int'l Symp. on Software Reliability Engineering, ISSRE 2004*, pages 417–428. IEEE, 2004.

[21] E. Guzman and W. Maalej. Do users like this feature? A fine grained sentiment analysis of app reviews. In *Proc. of the 22nd IEEE Int'l Requirements Engineering Conf. (RE)*, pages 153–162, 2014.

[22] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The WEKA data mining software: An update. *SIGKDD Explor. Newsl.*, 11(1):10–18, Nov. 2009.

[23] M. H. Halstead. *Elements of Software Science (Operating and Programming Systems Series)*. Elsevier Science Inc., New York, NY, USA, 1977.

[24] J. Hayes, A. Dekhtyar, and S. Sundaram. Advancing candidate link generation for requirements tracing: the study of methods. *IEEE Trans. on Software Engineering*, 32(1):4–19, Jan 2006.

[25] J. H. Hayes, A. Dekhtyar, S. K. Sundaram, E. A. Holbrook, S. Vadlamudi, and A. April. Requirements tracing on target (RETRO): improving software maintenance through traceability recovery. *Innovations in Systems and Software Engineering, ISSE*, 3(3):193–202, 2007.

[26] S. Herbold. Training data selection for cross-project defect prediction. In *Proc. of the 9th Int'l Conf. on Predictive Models in Software Engineering*, PROMISE '13, pages 6:1–6:10, New York, USA, 2013. ACM.

[27] M. Hitz and B. Montazeri. Chidamber and kemerer's metrics suite: A measurement theory perspective. *IEEE Trans. on Software Engineering*, 22(4):267–271, 1996.

[28] Y. Jiang, B. Cuki, T. Menzies, and N. Bartlow. Comparing design and code metrics for software quality prediction. In *Proc. of the 4th Int'l WS. on Predictor models in software engineering*, PROMISE '08, pages 11–18, New York, USA, 2008. ACM.

[29] Y. Jiang, B. Cukic, and T. Menzies. Fault prediction using early lifecycle data. In *Proc. of the 18th IEEE Int'l Symp. on Software Reliability, ISSRE '07.*, pages 237–246, Nov 2007.

[30] E. Keenan, A. Czauderna, G. Leach, J. Cleland-Huang, Y. Shin, E. Moritz, M. Gethers, D. Poshyvanyk, J. Maletic, J. Huffman Hayes, A. Dekhtyar, D. Manukian, S. Hossein, and D. Hearn. Tracelab: An experimental workbench for equipping researchers to innovate, synthesize, and comparatively

evaluate traceability solutions. In *Proc. of the 34th Int'l Conf. on Software Engineering*, ICSE '12, pages 1375–1378. IEEE, 2012.

[31] S. Kim, T. Zimmermann, J. Whitehead, and A. Zeller. Predicting faults from cached history. In *Proc. of the 29th Int'l Conf. on Software Engineering*, ICSE '07, pages 489–498, Washington, DC, USA, 2007. IEEE Computer Society.

[32] B. Lika, K. Kolomvatsos, and S. Hadjiefthymiades. Facing the cold start problem in recommender systems. *Expert Systems with Applications*, 41(4, Part 2):2065 – 2073, 2014.

[33] S. Lohar, S. Amornborvornwong, A. Zisman, and J. Cleland-Huang. Improving trace accuracy through data-driven configuration and composition of tracing features. In *Proc. of the 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE'13*, pages 378–388, 2013.

[34] T. J. McCabe. A complexity measure. *IEEE Trans. on Software Engineering*, 2(4):308–320, July 1976.

[35] A. K. McCallum. Mallet: A machine learning for language toolkit. http://mallet.cs.umass.edu, 2015.

[36] A. Mockus and J. D. Herbsleb. Expertise browser: a quantitative approach to identifying expertise. In *Proc. of the 24th Int'l Conf. on Software Engineering, ICSE 2002*, pages 503–512. ACM, 2002.

[37] L. Moreno, G. Bavota, S. Haiduc, M. Di Penta, R. Oliveto, B. Russo, and A. Marcus. Query-based configuration of text retrieval solutions for software engineering tasks. In *Proc. of the 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 567–578, New York, USA, 2015. ACM.

[38] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *Proc. of the 27th Int'l Conf. on Software Engineering, ICSE 2005.*, pages 284–292. IEEE, 2005.

[39] T. J. Ostrand and E. J. Weyuker. A tool for mining defect-tracking systems to predict fault-prone files. *Proc. of the Int'l WS. on Mining Software Repositories*, pages 85–89(4), 2004.

[40] A. Panichella, B. Dit, R. Oliveto, M. D. Penta, D. Poshyvanyk, and A. D. Lucia. How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms. In *Proc. of the 35th Int'l Conf. on Software Engineering, ICSE '13*, pages 522–531, 2013.

[41] J.-w. Park, M.-W. Lee, J. Kim, S.-w. Hwang, and S. Kim. CosTriage: A cost-aware triage algorithm for bug reporting systems. In *AAAI*, 2011.

[42] D. Poshyvanyk, A. Marcus, V. Rajlich, Y. Guéhéneuc, and G. Antoniol. Combining probabilistic ranking and latent semantic indexing for feature identification. In *Proc. of the 14th Int'l Conf. on Program Comprehension (ICPC 2006)*, pages 137–148, 2006.

[43] A. I. Schein, A. Popescul, L. H. Ungar, and D. M. Pennock. Methods and metrics for cold-start recommendations. In *Proc. of the 25th Annual Int'l ACM SIGIR Conf. on Research and Development in Information Retrieval*, SIGIR '02, pages 253–260, New York, USA, 2002. ACM.

[44] Y. Shin, J. H. Hayes, and J. Cleland-Huang. Guidelines for benchmarking automated software traceability techniques. In *Proc. of the 8th IEEE/ACM Int'l Symp. on Software and Systems Traceability, SST 2015*, pages 61–67, 2015.

[45] Y. Singh, A. Kaur, and R. Malhotra. Prediction of fault-prone software modules using statistical and machine learning methods. *Int'l Journal of Computer Applications*, 1(22):8–15, 2010.

[46] D. Spinellis. Tool writing: a forgotten art? (software tools). *Software, IEEE*, 22(4):9–11, July 2005.

[47] C. Thornton, F. Hutter, H. H. Hoos, and K. Leyton-Brown. Auto-weka: Combined selection and hyperparameter optimization of classification algorithms. In *Proc. of the 19th ACM SIGKDD Int'l Conf. on Knowledge discovery and data mining*, pages 847–855. ACM, 2013.

[48] Z. Zhang, C. Liu, Y. Zhang, and T. Zhou. Solving the cold-start problem in recommender systems with social tags. *CoRR*, abs/1004.3732, 2010.

[49] P. R. Zimmermann, Thomas and A. Zeller. Predicting defects for eclipse. In *Proc. of the Int'l WS on Predictor Models in Software Engineering, PROMISE'07: ICSE Workshops 2007*, pages 9–9. IEEE, 2007.

[50] T. Zimmermann and N. Nagappan. Predicting defects using network analysis on dependency graphs. In *Proc. of the 30th Int'l Conf. on Software Engineering, ICSE 2008*, pages 531–540, 2008.

[51] T. Zimmermann, N. Nagappan, H. C. Gall, E. Giger, and B. Murphy. Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In *Proc. of the the 7th Joint Meeting of the European Software Engineering Conf. and the ACM SIGSOFT Symp. on The Foundations of Software Engineering, ESEC/FSE'09*, pages 91–100, 2009.