Assessing Software Fault Risk with Machine Learning

Naveen Ashish, Greg Barish and Steven Minton

Abstract: This paper presents a data-driven solution for assessing fault (defect) risk in software code. Our particular interest is in code bases of large software projects (including open-source projects) that involve multiple developers, and where the project code base evolves over an extended period of time (multiple years). Our goal is to predict fault risk at the level of code check-in episodes, where individual commits are assessed for risk. Our solution predicts code risk by analyzing a rich variety of data that has been aggregated about software engineering development behavior. It is powered primarily by machine learning. A central aspect of this work is on determining the most effective machine learning algorithm(s) for this problem, after a comprehensive investigation of a wide variety of algorithms and frameworks. The resulting system is effective in predicting risk code commits in repositories with a very sparse code fault density (1-3%). We are able to achieve an average of 10-fold improvement in the precision (Precision@100) of identifying faulty commits, and an as high as 20-fold improvement in some cases. The improved efficiency in identifying risky code translates directly to expected improvement in efficiency (reduction of time and effort) for code review.

Keywords: software quality, code review, fault detection, machine learning, code risk assessment, artificial intelligence

---

Naveen Ashish
InferLink Corporation
El Segundo, USA
nashish@inferlink.com

Greg Barish
InferLink Corporation
El Segundo, USA
gbarish@inferlink.com

Steven Minton
InferLink Corporation
El Segundo, USA
sminton@inferlink.com

I. INTRODUCTION

Maintaining large software repositories, especially those that involve multiple developers and evolve over a long period of time (years), is a challenge [8]. The ability to identify risky code, as it is checked in, would be of significant value from a software quality perspective. By risky code we mean portions of code (checked in) that are likely to result in a *fault* in the future in that repository. A fault is an error or "bug" in software that cause the software to behave in a manner different and unwanted from as designed (and intended). This paper presents our work on developing an approach for efficiently identifying code fault risk in such large software project code bases. One of the key novel aspects of the research has been to investigate how the behavior of developers - in the context of software development, bears upon the quality of the software code.

Our approach is data centric, where we leverage a variety of dynamically aggregated data for predicting code fault risk. This work focuses on the machine learning details of CodeFault [15], a system for automatically identifying software risk based on engineering behavior. CodeFault aggregates data from multiple sources associated with a particular software repository of interest, uses a novel approach to markup faults, and leverages machine learning and code-as-document techniques [11] to predict commits of risk based on patterns mined from this combined data set.

We have modeled the problem as a classification problem i.e., where *individual* code commits are classified for fault risk. The application of this (originally) commit level risk assessment for practical software quality management, is discussed ahead in the paper. Our solution achieves an (on-average) 10-fold increase in the precision of faulty commit identification - compared to an exhaustive search or examination of random subsets of commits for faults. Additionally, we are able to identify the primary factors that do (and do not) influence software quality in terms of code fault risk. The aggregated data, the basis of our data-driven approach, includes 1) data related to development behavior i.e., the behavior of individual developers', and 2) data about software vulnerabilities which is employed to learn behavioral patterns associated with faulty code. Code risk in a new (unseen) software project can then be assessed based on the patterns that emerge from analysis of the integrated database (which has prior data).

This paper is organized as follows. Section 2 is a discussion summary on code risk assessment, in general. Section 3 starts with a brief summary of the database

curation, and then describes the feature engineering for machine learning done over this data. We also introduce the machine learning paradigms investigated. In Section 4 we summarize the key conclusions from the investigation, namely (i) The most effective machine learning approach (algorithm) for this problem. (ii) Explaining the machine learning in terms of the specific most influencing factors we uncovered that are associated with code fault risk. (iii) The configuration and parameters for training as well as operating such a code risk prediction system in a real-world and dynamically evolving software repository environment. Section 5 discusses related work on code risk assessment. This has been an area of investigation for decades but human behavioral aspects have been relatively unexplored. Finally Section 6 provides a conclusion and directions for future work.

II. DISCUSSION

Software vulnerability has been investigated at various levels, such as identifying vulnerable regions (defined by specific range of lines of code), specific modules that are vulnerable, vulnerable files, a more granular identification at the level of individual code changes and commits [1]. The work in [2] is a first of its kind empirical study to determine the *characteristics* of vulnerable code changes, and in particular the characteristics of developers likely to introduce code vulnerabilities. The study uncovered interesting insights, for instance the more experienced developers authored the majority of the vulnerabilities but changes by less experienced developers were 2-24 times more likely to inject vulnerabilities. The likelihood of a vulnerability being introduced increases with number of lines of code changed. The code author experience and author's (place of) employment were also found to have an influence on the propensity of vulnerability injection. VCCFinder [3] is a system that identifies commits in a repository that are likely to introduce defects. The system does this on the basis of features related to the code commit activity and aspects of the developers. It also factors text based features, coming from developer comments. VCCFinder was developed based on datasets from Github where (known) CVEs were mapped to Github commits. An SVM is trained to then classify faulty commits based on (i) repository associated features, namely the total number of commits and the number of developers for that repository. (ii) commit associated features such as the number of additions, deletions and also the number of commit hunks. (iii) the commit message, treated as free text (iv) functions in the code (v) keywords (from the code programming language). The system uncovered interesting patterns such as new developers being more likely to

introduce defects. Or, commits over code that has been iterated over frequently, including by multiple developers, are more likely to inject faults.

The work of [5] contributes a new dimension for feature engineering for defect prediction. Traditionally, developer metrics such as their experience, coding skills, commit activities etc. are measured in a single snapshot. In these system these metrics are captured *periodically*, which enables capturing the change in developer behavior over time. The model for defect has been evaluated on the Lucene and Jackrabbit open-source repositories. A number of supervised machine learning classifiers such as Random Forest, Naïve Bayes, Regression, J48 Decision Trees etc. were evaluated. For test datasets with fault (commit) densities of 33-50% the system achieved commit fault classification accuracies in the range of 0.5-0.7 F1 score. [1] presents an approach to *personalized* defect prediction where a separate defection prediction model is developed for each developer. Defects are classified at the file change level and the system does so based on features in three main categories namely (i) code characteristic vectors (ii) bag of words and (iii) metadata, such as hour , min and day of commit and also file path names etc. This system was able to achieve a four-fold increase in bug detection compared to traditional change classification on inspecting the top 20% lines of code that are predicted buggy. The classification accuracy of classifying defects F1 is in the range of 0.59-0.74 on datasets where the percent of buggy changes (fault density) is 25-30%.

III. APPROACH

We describe 1) feature engineering over the data and 2) the machine learning investigation in this section.

A. Feature Engineering

The aggregated database is assembled over data coming from open-source software repositories (such as GitHub), publicly available databases, and profiles of software developers on open Internet forums and social-media. It is collected using a combination of API access, web scraping, and git-style data collection scripts. This database also has information on whether each individual commit is faulty, or not. The fault/no-fault labeling was obtained by first identifying commits that indicated a fix and then selectively working backwards to identify which prior commits

introduced code that was removed as part of the fix, using detailed techniques described in [15].

The aggregated database has data which is structured but we need to further synthesize *features* from this data, to then feed into the machine learning algorithms investigated. We considered the various aspects that can potentially influence code faultiness, and determined three distinct categories of potentially influencing features. The categories relate to properties of particular commits, those of developers, and those of the code.

Commit oriented features are aspects of an individual commit itself. The features include 1) aspects of commit *activity,* such as modifications, additions, deletions counts associated the commit. Each of the counts becomes a commit related feature. 2) temporal aspects, starting with date (time) stamp of the commit and including derived features such as the hour of the day and the day of the week of that commit. 4) commit description, such as the commit message provided with the commit

Developer oriented features are features related to software developers associated with a code repository (as authors) or associated with particular commits in the code. These include 1) the developer identity and associated attributes such as their login ID, name, employer, location etc. 2) developer reputation, in measures (attributes) such as star ratings, number of “likes” (on their repository, profile etc.). 3) developer knowledge and skills such as their set of programming languages etc.

Code oriented features which are features of the code files associated with the commit. Such as the folder path details of a code file, file extension type etc.

Table I provides the specific *features* that are extracted or synthesized from the analysis database. These features are in three categories, (a) developer oriented features, (b) commit oriented features and (c) (code) file oriented features.

TABLE I. FEATURES FROM THE ANALYSIS DATABASE

(a) Commit oriented features

Feature	**Description**	**Example**
ID	*Commit identifier*	12345
Commit SHA	*Commit hash identifier*	2acee567eed8889f7ae

Commit message	*The text description included as commit documentation*	'Updated PCRE used for win32 builds.'
Modifications count	*Modifications in a commit*	45
Additions count	*Additions in a commit*	32
Deletions count	*Deletions in a commit*	39
Author name, login, ID	*Code author name, login, and identifier*	John Smith, Jsmith123, 4563
Author email	*Code author email*	jsmith@microsoft.com
Committer name, login, ID	*Code committer name, login, and identifier*	Mike Foster, mfoster, 3322
Commit date Hour of day* Day of week*	*Date stamp of the commit. Note that (the commit) 'hour of day' and 'day of week' are derived from the commit date stamp*	'2019-04-24 13:38:51' 13 Friday

(b) Developer oriented features

Feature	**Description**	**Example**
ID	*(Unique) developer identifier*	1222
login	*Developer login name*	JSmith123
avatar_url	*Profile URL*	Github.com/JSmith123
Company	*Organization affiliated with*	Microsoft
Blog	*Developer has a blog (Y/N)*	Y
Location	*Geolocation if given*	Seattle
Email	*Email address*	jsmith@microsoft.com
Hireable	*Hireable (Y/N)*	N
Bio	*Developer bio (if any)*	
Public repos count	*Count of developer's repositories*	73
Public gists count	*Count of developer's gists*	4
Followers count	*Count of followers*	35
Following count	*Count of people followed*	22

(c) Code oriented features

Feature	**Description**	**Example**
ID	*File identifier*	8
Commit meta ID	*Associated commit identifier*	2
File path	*Directory path to the file*	src/main/scripts/ proxy_module.java
Status	*File modified*	modified
Modification, additions, deletions	*Activity counts*	23, 17, 4
Path 1*	*Root folder name*	src
Path 2*	*File name*	proxy_module
Ext	*File extension*	"java"
Path as text	*Path folder names*	"src main scripts proxy_module"
Deleted code		`return X+ ","+ wordLength(str)+ ","+ endsNumber(str)+ "," + hasYesNo(str)+ ","+ charFractions(str);`

We have above 14 commit oriented features, 11 developer oriented features and 10 code oriented features – a total of 35 features in the feature vector associated with a commit.

B. Fault Risk Prediction: Machine Learning Classification

The fundamental problem is that of predicting whether a commit will result in a fault or not. However, and as with most predictive analysis goals, rather than a binary Y/N of whether the commit will be faulty we want the *probabilistic likelihood* of a commit being faulty. The likelihood translates directly to the commit risk. The aggregated database contains the fault status of each individual commit – essentially the target class in our classification problem. Thus *supervised learning* based classification [13] becomes feasible.

The classification problem is formulated at a commit level. The goal is, for new data (i.e., commits), to assign a *quantitative faultiness likelihood* to the commit. This likelihood is simply the probabilistic likelihood that the commit belongs to the fault=Y class, by any classifier.

We have investigated three different paradigms (and associated frameworks):

1) **Feature driven classifiers** These are "traditional" classifiers that work off synthesized features (the features provided in Table I). The options are classifiers such as Decision Tree [9], SVM [9], Random Forest [9] and many others that are available in commonly used machine learning frameworks. Here we also treat certain fields of data as unstructured text. Then, text classifiers, that are somewhat different from the discrete feature driven classifiers (such as Decision Tree, SVM etc.) are employed over these text fields to predict fault risk.
2) **Ensembles over multiple classifiers** In simple terms, a classifier ensemble is akin to a *committee of experts*, where the opinion of multiple experts is taken into account [9]. In this case we assembled and evaluated a variety of classifier ensembles.
3) **Deep learning classifiers** These are classifiers in the unsupervised feature learning category where the system itself learns the features to be employed. The deep learning classifiers i employ multiple layers of neural networks to perform the classification. Rather than developing custom deep learning models ourselves we use cloud based *automated machine learning* [12] frameworks where the (deep learning based) models are assembled automatically. This is done using techniques such as neural architecture search (NAS) [12]. The cloud automated machine learning services also provide the high end computational resources that deep learning approaches typically require.

IV. EVALUATION

As mentioned above, our goal was to arrive at an efficient code risk assessment solution after a comprehensive exploration of various machine learning algorithms and frameworks. By efficient risk assessment we mean finding the maximum amount of risky code commits with the least effort. An *inefficient* approach would be (i) Exhaustively examining all code commits in a repository – this is practically infeasible in large code bases. (ii) Examining randomly sampled subsets of code commits with the hope of finding risky commits. We refer to this approach as the brute force approach. The overall goal is to then apply an efficient approach in a production setting of a large software repository development.

The evaluation progressed as follows. We first evaluated a wide spectrum of feature driven classifiers with features of different kinds. We then extended that analysis to *ensembles* of multiple different classifiers, including embeddings-based text classifier. We further evaluated the same (commit fault) classification task with the same feature set on a deep learning automated machine learning framework. Orthogonally, we a) investigated the *explainability* of the approach in terms of determining which factors do (or do not) bear upon fault risk, and b) conducted a *learning curve analysis* to establish the limits of fault risk prediction accuracy that can be achieved with such data and characterize the amount of data (number of commits) required for good predictions. We present the results of these evaluations, in the above order.

A. Classification

There are three facets of the classification approach – individual classifiers, ensembles of multiple classifiers, and deep learning using automated machine learning services.

1) Feature Driven Classification

We started by evaluating individual feature driven classifiers, such as Decision Tree, Random Forest, Naïve Bayes , Support Vector Machine (SVM) and others. In this project we employed the SciKit Learn machine learning framework in Python [13], that provides a wide spectrum of machine learning algorithm implementation as well as accompanying data manipulation utilities. In Table II we provide experimental results on the accuracy of (ten) individual feature-driven classifiers for classifying commit faultiness. Given space constraints, results are provided for only 3 of the (over) 60 datasets we have conducted the evaluation on. The results presented are representative of the entire dataset. Note: P=Precision, R=Recall and F=F-Measure in the tables hereon.

TABLE II. INDIVIDUAL, FEATURE DRIVEN CLASSIFIERS

Classifier	Nginx (Baseline: 0.02)			Apache (Baseline: 0.01)			Wget (Baseline: 0.02)		
	P	**R**	**F**	**P**	**R**	**F**	**P**	**R**	**F**
Random Forest	0.11	0.79	0.19	0.03	0.74	0.06	0.09	0.81	0.16
Decision Tree	**0.59**	0.74	0.66	**0.34**	0.46	0.39	**0.31**	0.27	0.29
AdaBoost	0.09	0.71	0.16	0.01	0.56	0.02	0.00	0.00	NA
Naïve Bayes	0.20	0.36	0.26	0.03	0.07	0.04	0.00	0.00	NA
SVM	0.00	0.00	NA	0.00	0.00	NA	0.00	0.00	NA
QDA	0.19	0.37	0.25	0.03	0.09	0.04	0.00	0.00	NA
KNN	0.10	0.20	0.13	0.00	0.00	NA	0.03	0.08	0.04
Message text	0.12	0.40	0.18	0.07	0.19	0.10	0.12	0.11	0.11
File tokens	0.05	0.84	0.09	0.02	0.74	0.04	0.03	0.43	0.06
Code snippets	0.08	0.14	0.10	0.02	0.05	0.03	0.29	0.05	0.09

The (representative) results in Table II provide several important insights, which include:

1. We observe a rather **high variance** in the classification accuracy across different types of classifiers. For instance the precision ranges from **0** (for SVM) to as high as **0.59** (for the Decision Tree classifier and for the nginx dataset).
2. For effective classifiers, such as the Decision Tree, the fault classification precision is high, relative to the precision achieved with brute force random sampling examination (the repository fault density essentially). This classifier is able to achieve a precision of **0.59** for nginx where the fault density is **0.02** i.e. a **30X improvement** is achieved in fault identification. A similar **25-30X precision improvement** is observed for the apache and wget datasets as well.
3. The three text data driven classifiers (over the commit message, file path as text, and code snippets as text) are able to achieve a moderately high precision improvement as well – a **5-10X** improvement over the brute force examination precision.

2) Ensembles

We next evaluated *ensembles* of multiple classifiers. A classifier ensemble is a "meta-classifier" [13] that is a classifier on top of other individual classifiers with the output of the latter treated as features of the meta classifier. Table III provides the results of ensemble evaluation with two different meta classification schemes. As above, results are reported for only a small subset of datasets evaluated but are representative of the findings in all datasets.

In the ensembles results, we do not see any merits of a voting based ensemble. The ensemble accuracies are inferior to the best individual classifier (Decision Tree) accuracies in terms of precision as well as F-score. The XGBoost driven ensemble is however significantly better than the voting based ensemble. It does well when taking an ensemble of all the (ten) individual classifiers as well as when taking an ensemble of only the top two classifiers – Decision Tree (DT) and Random Forest (RF). Nevertheless its notable that the accuracy of the individual Decision Tree classifier stands better than any ensemble, even with XGBoost – we attribute this to possible over-fitting [13] by more complex models and ensembles.

TABLE III. ENSEMBLES

(a) Ensemble, with voting for meta-classification

Votes	**Nginx (Baseline: 0.02)**			**Apache (Baseline: 0.01)**			**Wget (Baseline: 0.02)**		
	P	**R**	**F**	**P**	**R**	**F**	**P**	**R**	**F**
2	0.09	0.81	0.16	0.02	0.80	0.04	0.09	0.81	0.16
4	0.20	0.71	0.31	0.15	0.38	0.22	0.31	0.27	0.29
6	0.31	0.37	0.34	0.29	0.03	0.05	0.00	0.00	NA
8	0.75	0.09	0.16	0.00	0.00	NA	0.00	0.00	NA
9	1.00	0.01	0.02	0.00	0.00	NA	0.00	0.00	NA
10	0.00	0.00	NA	0.00	0.00	NA	0.00	0.00	NA

(b) Ensemble, with XGBoost meta-classifier

Votes	**Nginx (Baseline:0.02)**			**Apache (Baseline: 0.01)**			**Wget (Baseline: 0.02)**		
	P	**R**	**F**	**P**	**R**	**F**	**P**	**R**	**F**
ALL Classifiers	0.57	0.73	0.64	0.30	0.39	0.34	0.36	0.11	0.17
Only Decision Tree & Random Forest	0.53	0.73	0.61	0.35	0.42	0.38	0.49	0.49	0.49

3) Deep learning; automated machine learning

Finally we evaluated deep learning based classification for fault classification. We employed an automated machine learning (AML) framework where an AML framework automatically generates the “best” deep learning model through a neural architecture search over a large space of possible models. Specifically, we used Google Cloud AutoML Tables [13] where the very same

feature set and target class (commit fault = Y/N) is provided to the service for classification. Table IV provides these results. The results, shown for six datasets but representative of the overall trend, demonstrate that AutoML does *not* provide any improvement in fault classification accuracy over the better performing feature driven classifiers such as the Decision Tree. The results, while not obvious are not entirely surprising as well. Not all machine learning driven classification problems benefit (significantly) from deep learning algorithms. Plus the classifier framework effectiveness also depends on the features that are ultimately useful for classification – the topic of the next sub-section.

TABLE IV. AUTOMATED MACHINE LEARNING RESULTS

Dataset	Best In-house P	F	R	AutoML P	F	R
nginx	0.59	0.74	0.66	0.38	0.25	0.31
apache	0.34	0.46	0.39	0.14	0.04	0.06
curl	0.21	0.22	0.22	0.52	0.08	0.14
wget	0.00	0.00	NA	0.00	0.00	NA
videolan	0.43	0.43	0.43	0.53	0.29	0.38
podofo	0.22	0.62	0.32	0.33	0.33	0.33

For code risk assessment and at the commit level, this evaluation informs us that the deep learning approach provides no additional benefits. It is also a solution that is computationally and monetarily expensive.

B. Explainability

The above evaluation of various classifiers, ensemble configurations and automated services enabled us to hone in on the best classifier for the problem. In this case the XGBoost ensemble and also the Decision Tree (by itself) perform the bst. We are also interested in *explaining* the classification results. Explanation benefits include: (i) getting a better understanding of which specific factors (features) most influence code risk, and also those that do not. Knowledge of the most influential factors can significantly help in optimizing the code review process. If we determine, say, that code faults are unusually more (or less) frequent during certain days of the week and/or certain hours of the day then one can prioritize code review towards (or against) commits for those days and/or hour windows in the day. (ii) understanding why structurally relatively simpler classifiers like the Decision Tree are best suited for this problem

1) **Feature importance**

Machine learning based classification frameworks provide capabilities for determining the relative importance of various features for certain kinds of classifiers on a specific classification task. Classifiers such as Decision Tree are constructed by a process of supervised segmentation [13] which is to segment a dataset considering one factor (feature) at each step. The feature selected at each step is based on maximizing the information gain that is achieved at that step [13]. In SciKit Learn, feature importance is a measure that is calculated as the decrease in node impurity weighted by the probability of reaching that node. The node probability can be calculated by the number of samples that reach the node, divided by the total number of samples. The higher the value the more important the feature. Functions for providing the feature importance are built into the SciKit framework.

Table V provides the feature importance scores extracted the Decision Tree classifier for our problem. The scores are illustrated for four (out of several) datasets and we only provide scores where the score is greater than zero (a feature importance score of zero for a feature implies that the feature has no influence on the classification/prediction whatsoever).

TABLE V. FEATURE IMPORTANCE (REPRESENTATIVE DATASETS)

Feature importance		**Feature importance**	
author_name	0.93816	author_name	0.87569
additions_count_commit	0.04141	additions_count_commit	0.06819
modifications_count_commit	0.01663	modifications_count_commit.	0.02683
deletions_count_commit	0.00041	deletions_count_commit	0.01080
committer_date_weekday	0.75268	committer_date_hour	0.61274
author_name	0.07270	modifications_count_commit	0.16359
additions_count_commit	0.05066	committer_name	0.07454
committer_date_hour	0.05057	author_name	0.05906
committer_name	0.04726	additions_count_commit	0.05896
deletions_count_commit	0.00996	committer_email_type	0.01382
modifications_count_commit	0.00590	last_month_faulty_commits	0.00893

The feature analysis reported in Table V reveals the following:

1) The (code) developer (reflected as ‘author_name’) is, by far, the most significant influencing factor for faultiness !
2) The activity during a commit, reflected in the additions, modifications and deletions counts is the next most significant factor.

3) The hour of day ('committer_date_hour') can at times be a very significant predictor of commit fault.

2) Statistically assessing feature importance

Another means of obtaining feature importance is to employ statistical analysis based on *risk ratios* [14] and *odds ratios* [14] of individual factors (attributes) associated with faults. The (faultiness) risk ratio of an attribute instance is simply the fraction of cases when that instance is associated with a fault. For instance if the faulty commits that developer 'John Doe' is associated with comprise 0.015% of the total number of commits (in the repository) then the risk ratio for 'John Doe' is 0.015. If on *average* a developer's faults comprise say 0.005% of the total commits then the odds-ratio for John Doe is 0.015/0.005=3.

The feature importance, provided by classifiers as well statistical analysis shows that 1) ultimately, there are only a handful (6 or so) features that have some finite association with commit risk (classification). 2) Even within this small subset, the feature importance is skewed towards just 2 or 3 features, such as the developer or committer (name). The fact that for this classification problem we have only a small number of influential features, and all discrete valued, explains why a supervised segmentation algorithm like the Decision Tree is well suited for the problem. Classifiers that are structurally more complex, such as say SVMs or deep learning multi-layer neural network classifiers are unable to extract further signal over the non-influential features (the majority), even with complex analysis over sets of such features.

C. Risk Analysis for Real-world Software Development

Our eventual goal is the realization of a working product that helps in efficient software code quality management in real settings. There are two dimensions that we added onto our analysis framework in this regard. The first is to factor *risk ratios* that come from statistical summaries of the data. The second is to evaluate the predictive power of the approach for future events where "future" is defined explicitly in the database. Our work is in the context of large software project repositories that evolve over a (typically) long period of time (multiple years). Any underlying hypothesis of this work is that the fault risk patterns determined at any point of time in the repository are predictive of the *future* fault risk behavior in that repository (from that point of time on). The fault prediction analysis presented in

Tables 2,3,4 and 5 is based on general patterns where while the training set (commits) are distinct from the test (holdout) set commits, the training set commits do *not* necessarily all predate the test set commits !

We now divide a dataset into train and holdout sets in a *temporal* fashion i.e., all train portion commits predate all test portion commits. Additionally, we are now in a position to define and incorporate features based on recent activity *history* from a certain time point. For instance features that are based on code commit activity from the day(s), week(s), month(s) *prior* to the time of that commit. Table VI provides the additional, history based features that we have added to the space. These are divided into two categories. One category is that of "regular" features such as the number of commits in the last day, week, month etc. The other category has features like the number of *faulty* commits in the last day, week, month etc.

TABLE VI. ADDITIONAL, HISTORY BASED FEATURES

History but no fault knowledge	**History including fault knowledge**
Commits in the past day/week/month	Faulty commits in the past day/week/month
Days since last commit	Days since last faulty commit
Daily variance	Daily variance

1) Results

Table VII provides the results of the risk analysis with incorporating statistical risk ratios, history based features (additional), and ensuring that the train data portion completely predates the test data portion. Some key aspects: (i) The results are reported in terms of the factor improvement over brute force examination of **Precision@100**. (ii) Both schemes for risk identification, classifier based as well as statistical analysis are reported. For statistical analysis Table VII provides only the odds-ratio based results (being superior to risk-ratio based). (iii) The best classifier, the XGBoost ensemble, is employed with three different feature sets – no history based features, including history but *without* knowledge of which commit (in the history) was faulty, and then history with the knowledge of which commits in the history are faulty.

TABLE VII. FAULTY DENSITY: FACTOR IMPROVEMENT

Dataset	**Statistical Analysis: Odds ratio**	**Classifier scheme no history**	**Classifier with previous month history**	**Classifier with activity history but no fault knowledge**
apache 1	3	2	15	10
apache 3	4	8	25	20
apache 5	1	5	10	10
i-magick1	1	2	6	10
curl 1	5	3	5	5
curl 3	4	0	10	10
curl 5	1	3	7	7
wget 1	1	1	1	1
wget 3	0	3	5	5
wget 5	1	1	3	3
openssl 1	3	5	10	11
openssl 3	13	0	8	16
openssl 5	8	3	15	13
nginx 1	5	5	10	10
libraw 1	4	4	5	5
libraw 5	3	5	6	6

The Precision@100 factor improvement is summarized, for the various schemes, in terms of the mean, minimum and maximum factor improvement obtained with that scheme. We also provide the 68-95-99 distribution.

Statistical analysis (odds-ratio based)

Mean improvement factor = 3.8 ; Minimum = 0 ; Maximum = 13

68-95-99.1: [0.6, 7] [0,10.2] [0,13.4]

Classifier (ensemble) no history

Mean improvement factor = 3.2 ; Minimum = 0, Maximum = 8

68-95-99.1: [1.25, 5.25] [0, 7.25] [0, 9.25]

Classifier with history, NO fault knowledge

Mean improvement factor = 8.9 ; Minimum = 1, Maximum = 20

68-95-99.1: [4.2, 13.6] [0, 18.3] [0, 23.0]

Classifier with history features, including fault knowledge

Mean improvement factor = 8.9 ; Minimum = 1, Maximum = 25

68-95-99.1: [3.2, 14.4] [0, 20.0] [0, 25.4]

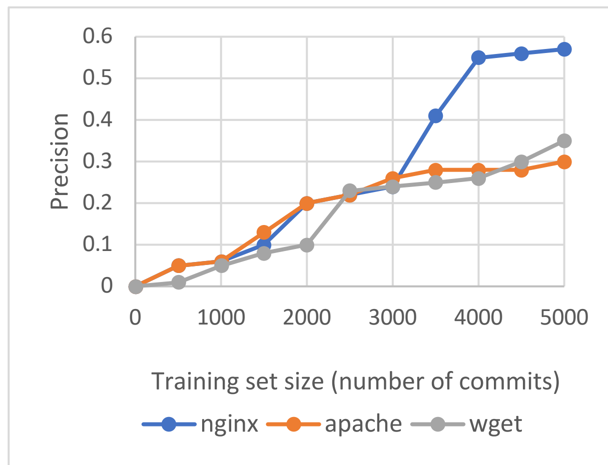
We conclude that 1) The inclusion of history based features significantly improves the risk prediction accuracy, even without fault knowledge in the recent history in some cases.2) Overall, a significant **average improvement of 9X** is achieved for Precision@100, this is **as high as 20-25X** in some cases. From a code inspection and review perspective these factor improvements translate directly to the potential savings in code examination for faults. In Table VIII we provide the feature importance analysis, with now also including the history based features that have been added.

TABLE VIII. FEATURE IMPORTANCE (HISTORY BASED FEATURES INCLUDED)

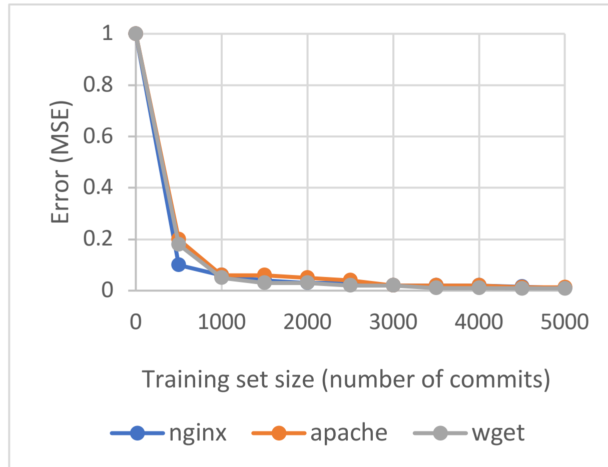
Feature importance		**Feature importance**	
author_name	0.93816	author_name	0.87569
additions_count_commit	0.04141	additions_count_commit	0.06819
modifications_count_commit	0.01663	modifications_count_commit	0.02683
last_month_commits	0.00324	last_month_commits	0.01550
deletions_count_commit	0.00041	deletions_count_commit	0.01080
last_month_faulty_commits	0.00014	last_month_faulty_commits	0.00295
committer_date_weekday	0.75268	committer_date_hour	0.61274
author_name	0.07270	modifications_count_commit	0.16359
additions_count_commit	0.05066	committer_name	0.07454
committer_date_hour	0.05057	author_name	0.05906
committer_name	0.04726	additions_count_commit	0.05896
deletions_count_commit	0.00996	committer_email_type	0.01382
last_month_faulty_commits	0.00675	last_month_faulty_commits	0.00893
modifications_count_commit	0.00590	deletions_count_commit	0.00447
last_month_commits	0.00325	last_month_commits	0.00390

D. Learning Curve Analysis

We determined above the kind of improvement we can expect to achieve in the precision of finding faulty commits. We now apply *learning curve analysis* [14] to determine what amount of training data is required for the classifier to achieve stability. Figure 1 provides the learning curves.



a) Learning curve: Precision



b) Learning curve: Mean square error (MSE)

Figure 1. Learning curves

Here a) presents classifier accuracy as it varies with the training data size (number of commits in training set) and b) provides the mean-square-error (MSE) as a function of training set size. We employed the Decision Tree classifier for learning curve analysis and the results shown for the three datasets in Figure 3 are representative of the overall trend.

We note from a) that the classifier precision plateaus at around 4000 commits in the training data, which we can take as an estimate of the amount of training data required for a stable risk predictor for any code base. From b) we observe that the "elbow" of the MSE appears to at around a training data size of 500 commits. However the error continues to decrease and stabilizes at around 4000 commits.

V. RELATED WORK

The area of software code risk analysis has many studied from many different perspectives (as described in the Discussion section) but it is only a handful of relatively recent work with a focus on developers and development behavior characteristics. The work in [5] for instance contributes a new dimension for feature engineering for defect prediction. Traditionally, developer metrics such as their experience, coding skills, commit activities etc. are measured in a single snapshot. In these system these metrics are captured *periodically*, which enables capturing the change in developer behavior over time. The model for defect has been evaluated on the Lucene and Jackrabbit open-source repositories. A number of supervised machine learning classifiers such as Random Forest, Naïve Bayes, Regression, J48 Decision Trees etc. were evaluated. For test datasets with fault (commit) densities of 33-50% the system achieved commit fault classification accuracies in the range of 0.5-0.7 F1 score. [1] presents an approach to personalized defect prediction where a separate defection prediction model is developed for each developer. Defects are classified at the file change level and the system does so based on features in three main categories namely (i) code characteristic vectors (ii) bag of words and (iii) metadata, such as hour , min and day of commit and also file path names etc. This system was able to achieve a four-fold increase in bug detection compared to traditional change classification on inspecting the top 20% lines of code that are predicted buggy. The classification accuracy of classifying defects F1 is in the range of 0.59-0.74 on datasets where the percent of buggy changes (fault density) is 25-30%. Finally, [10] is a patent on a technology for fault identification that factored in recent activity *history* in the analysis.

Our work is distinguished from the one above on the following key aspects: 1) We gave achieved a significant (10-fold) fault identification precision increase on repositories with much sparser faulty density – 1-2% in our case versus 25-30% in the works above. This was made possible with a) richer feature engineering, and b) investigating a more comprehensive space of potentially applicable machine learning paradigms for the problem. 2) Richer exploration and identification of recent activity history based features. The work in [10] is among the only to have employed some such history based features, we have explored this space more extensively, and identified some features highly correlated with fault risk. 3) Finally our curated database has data from repositories over several years of activity, this has enabled us to do a true "predict forward in time" analysis that earlier works have not provided.

VI. CONCLUSIONS and FUTURE WORK

We envision commit level risk identification being applied in practical software quality management in multiple ways. We could identify particular commits that are highly and that deserve review, and also identify regions of code (a set of commits from say a particular developer or in a particular time window) that should be reviewed. The feature importance analysis also provides cues on what particular developers, and particular file types should (or should not) be prioritized for examination.

The key new contributions from our investigation and system lie in two broad categories: 1) **feature engineering,** where we have investigated some new kinds of features not previously explored, and 2) **machine learning classification,** where our work is to the best of our knowledge the first to investigate the gamut of machine learning algorithms and paradigms comprehensively. Especially for data (repositories) where the fault density is sparse and thus harder to detect. In feature engineering we investigated some previously unexplored features such as specific code authors or committers (including distinguishing between author and committer), aspects of developers such as their (geographic) location, types of code files, treating the code itself as text and also semantically categorizing the temporal attributes (for instance placing hour of commit in buckets such as morning , evening, late night etc.). We also analyzed many recent history based features derived from the developer activity (commits) in time windows (days, weeks, months) before a commit. In machine learning classification we investigated the effectiveness of

about a dozen different feature driven classifiers. This also included text based classifiers for certain features that were text to begin with (the commit message for instance) or features synthesized as text (set of tokens of folder names in the path to a file for instance). We then evaluated *ensembles* of these multiple classifiers, with exploring multiple algorithms for the ensemble assembly. Finally, we have investigated the use of deep learning classifiers, including automated machine learning frameworks with evaluating our fault identification task on Google Cloud AutoML. The evaluation on a wide variety of machine learning frameworks and with over 60 datasets allowed us to conclude that ultimately only a rather small number of factors or features, and with discrete values, have some association with commit fault risk. A supervised segmentation algorithm such as a Decision Tree is this well suited for the risk classification problem. Over datasets with a very sparse (1-3%) distribution of code faults, we have achieved a fault identification precision improvement of about 10-fold on average and as much as 20-fold in some cases.

The conclusions from this investigation, development of the fault risk assessment pipeline, and experimental evaluation all center on the key takeaway that a machine learning classification based approach and system for predicting faulty commits is effective. At a more specific level we learnt several aspects, namely:

1) From amongst a spectrum of machine learning classification approaches and frameworks evaluated, a "simple" feature driven Decision Tree classifier had the best performance. This is compared to other feature driven classifiers (such as Random Forests, SVMs etc.), ensembles of such classifiers, and even deep learning frameworks (AutoML).
2) Only a very small set of features seem influential for faulty commit prediction, based on our feature importance analysis of a very wide spectrum of features evaluated.
3) Factoring information of very recent development activity on the repository (from the last few days, weeks and months) significantly improves fault prediction capabilities. By factoring features based on recent history we were able to achieve an average of 10-fold improvement in fault prediction (over brute force examination), and it was as high as 20-fold for certain repositories evaluated.
4) The approach has the potential to be effective in a real-world setting for predicting fault in new code commits, based on the analysis of the prior (commit) activity in that repository. Stable predictions appear to require a training data size (of prior commits data) of about 4000 commits.

Overall, this effort has resulted in an approach and system that is effective in fault risk prediction and has also provided us with a confident understanding of the limits (prediction accuracies possible) in such a task. This forms a foundation for a tool that we are building for software developers and software project managers that can employ data for improving the quality of their code as well as the efficiency of the development process.

VII. ACKNOWLEDGMENTS

We acknowledge the contributions of Matt Michelson, Neha Kansal, Khan-Luan Phuc Nguyen, Jennilyn Dee, Maria Muslea and Arren Bustamante for their work on the data aggregation pipeline, machine-learning pipeline, and text classification respectively. This work was supported by SBIR contract #140D6319C0016.

REFERENCES

[1] T. Jiang, L. Tan and S. Kim, "Personalized defect prediction," *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2013, pp. 279-289, doi: 10.1109/ASE.2013.6693087.

[2] Bosu, A., Carver, J. C., Hafiz, M., Hilley, P., & Janni, D. (2014, November). Identifying the characteristics of vulnerable code changes: An empirical study. In *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering* (pp. 257-268).

[3] Perl, Henning, Sergej Dechand, Matthew Smith, Daniel Arp, Fabian Yamaguchi, Konrad Rieck, Sascha Fahl, and Yasemin Acar. "Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits." In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pp. 426-437. 2015.

[4] Ghaffarian, S. M., & Shahriari, H. R. (2017). Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey. *ACM Computing Surveys (CSUR)*, *50*(4), 1-36.

[5] Wang, S., Liu, T., & Tan, L. (2016, May). Automatically learning semantic features for defect prediction. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)* (pp. 297-308). IEEE.

[6] Baljinder Ghotra, Shane McIntosh, and Ahmed E. Hassan. 2015. Revisiting the impact of classification techniques on the performance of defect prediction models. In Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15). IEEE Press, 789–800.

[7] Niedermayr, R., Röhm, T., & Wagner, S. (2019). Too trivial to test? An inverse view on defect prediction to identify methods with low fault risk. *PeerJ Computer Science*, *5*, e187.

[8] Moeyersoms, J., de Fortuny, E. J., Dejaeger, K., Baesens, B., & Martens, D. (2015). Comprehensible software fault and effort prediction: A data mining approach. *Journal of Systems and Software*, *100*, 80-90.

[9] Grieco, G., Grinblat, G. L., Uzal, L., Rawat, S., Feist, J., & Mounier, L. (2016, March). Toward large-scale vulnerability discovery using machine learning. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy* (pp. 85-96).

[10] United States Patent 9,921,948 B2. Zieder, G., Kozorovitzky, B., Eliassaf, O., Levi, EF., Ohad, Y. (2018) Software Commit Risk Level

[11] Barish, G., Michelson, M., & Minton, S. (2017). Mining commit log messages to identify risky code. In *Proceedings on the International Conference on Artificial Intelligence (ICAI)* (pp. 345-349). The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp).

[12] Zöller, M. A., & Huber, M. F. (2021). Benchmark and survey of automated machine learning frameworks. *Journal of artificial intelligence research*, *70*, 409-472.

[13] Raschka, S., & Mirjalili, V. (2019). *Python machine learning: Machine learning and deep learning with Python, scikit-learn, and TensorFlow 2*. Packt Publishing Ltd.

[14] Kroese, D. P., Botev, Z. I., Taimre, T., & Vaisman, R. (2019). *Data science and machine learning: mathematical and statistical methods*. Chapman and Hall/CRC.

[15] Barish, G.; Ashish, N.; Minton, S. CodeFault: Identifying software risk based on engineering behavior. 2022. InferLink Technical Report. Available at https://www.inferlink.com/s/codefault-behavior-risk.pdf