Code Quality Prediction Under Super Extreme Class Imbalance

Noah Lee
Meta Platforms, Inc.
Menlo Park, USA
noahlee@fb.com

Rui Abreu
Meta Platforms, Inc.
Menlo Park, USA
rui@computer.org

Nachiappan Nagappan
Meta Platforms, Inc.
Bellevue, USA
nnachi@fb.com

***Abstract*—Predicting the quality of software in the early phases of the development life cycle has various benefits to an organization's bottom line with wide applicability across industry and government. Yet, developing robust software quality prediction models in practice is a challenging task due to "super" extreme class imbalance. In this paper, we present our work on a code quality prediction framework, we call Automated Incremental Effort Investments (AIEI), to fasten the process of going from data to a performant model under super extreme class imbalance. Experiments on a large scale real-world dataset, from Meta Platforms, show that the proposed approach competes with or outperforms state-of-the art shallow and deep learning approaches. We evaluate the practical significance of the model predictions on test case prioritization efficiency, where AIEI achieves the top rank reducing code review time by 2.5% and test case resource utilization by 9.3%.**

***Index Terms*—Software Reliability, Code Quality Prediction, Super Extreme Class Imbalance**

I. Introduction

Delivering software at scale and speed without jeopardizing quality is of importance to many industries and governments to maximize resource efficiency, competitive advantage, and avoid preventable costs [15]. As an example, poor software quality, in the U.S. alone, is estimated to be at 2 trillion dollars[1] attributable to failed software projects, technical debt from legacy systems, and operational failures.

Software quality prediction [7], [16], [19] aims to identify risky code at authoring time to prevent future operational failures in production. Arguably, it is amongst the best allies to improve reliability [2], quality assurance [13], and resource management activities [12]. Yet, developing robust software quality prediction models in practice is a challenging task. This is notoriously the case at Meta where thousands of engineers are committing code changes into production every day. The code is large, comprises multiple languages, covers the full stack, and undergoes rigorous code quality management processes [9].

We introduce the term "super" extreme class imbalance, where the minority class label only accounts for $< 0.1\%$. While the literature has dealt in depth with moderate or extreme class imbalance, empirical evidence on super extreme class imbalance in the code quality prediction context is scarce. As the minority class (i.e. an adverse code quality event) becomes extremely rare, standard best practices start to fail and need special considerations.

In this paper, we propose a learning framework, we call Automated Incremental Effort Investments (AIEI) (see **Figure 1**). The goal of AIEI is to i) assess the feasibility and significance of performing code quality prediction under super extreme class imbalance and ii) fasten the process to go from data to a performant model while gaining a better understanding of the modeling intricacies that arise. Experiments on large-scale real-world data show that our approach competes with or outperforms state-of-the art shallow and deep learning approaches. We further evaluate model performance on test case prioritization efficiency by looking at savings in code review time and test case resource utilization. AIEI achieves the top rank ($\#1$) improving review time by 2.5% and test case resource utilization by 9.3% when compared to the best performing model in the model pool. Comparing to baseline models BM-25 and BM-50, we observe a 99.3% and 99.6% improvement. The main contributions of this paper are summarized below:

- We present a code quality prediction framework that can deal with super extreme class imbalance and efficiently go from data to a high performant model. We compute empirical lower bounds on the required training samples to prevent cross validation fold failures while keeping the training size minimal. This type of work and context has not been addressed in prior research (see **Section III**).
- We provide large-scale empirical results on our codebase covering +20 languages, +1K projects, a quarter million data samples, and evaluate +40 models in the full model pool. The scale of our study provides stronger empirical evidence compared to prior studies (see **Section IV**).
- We discuss how our approach competes with or outperforms state-of-the art shallow and deep learning approaches based on the Matthew Correlation Coefficient (MCC) (see **Section V**).

II. Related Work

Machine Learning for Software Engineering (ML4SE) — ML4SE is at the intersection between machine learning and software engineering and aims to automate software engineering tasks such as requirement analysis [24], effort

[1]https://www.it-cisq.org/cost-of-poor-software-quality-in-the-us/index.htm (accessed August 26, 2022)

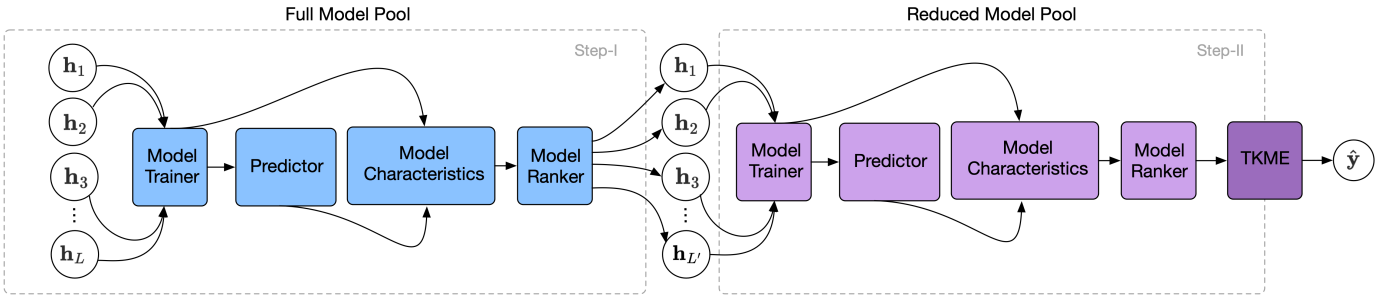

Fig. 1. **The Automated Incremental Effort Investment (AIEI) framework and Top-K Meta Ensemble (TKME) model architecture.** A diagram describing how in Step-1 we train a large model pool on small data (low effort). The intent is to reduce the model pool to identify top model candidates quickly that then undergo a more rigorous learning process with increased training effort and rigor. The TKME architecture combines the inferences from the reduced model pool to deliver the final predictions.

estimation [14], software defect prediction [6], [17], [18], type inference [5], [11], code completion [23], summarization [8], [22], embeddings [25], [27], synthesis [4], and translation [21].

Software quality prediction — Nagappan *et al.* [16] empirically studied the influence of organizational structure on software quality and proposed a metric scheme to quantify organizational complexity. They presented empirical evidence that organizational metrics are effective predictors of failure-proneness.

Reddivari and Raman [19] presented an evaluation of 8 ML techniques in the context of reliability and maintainability by training models on software derived metrics. They reported that Random Forest is the best performer with an AUC of 0.8. Compared to our work, we perform model validation on datasets that are 250X larger in size with a class imbalance that is 20-500X more extreme.

Goyal and Bhatia [7] compared 30 prediction models based on ANNs, SVMs, decision trees, KNNs, and Naïve-Bayes classifiers on 6 datasets. Key features were static code metrics such as McCabe complexity. They reported that ANN is the best performer in terms of ROC, AUC, and accuracy measures. In contrast, our work considers a model pool of +40 models as part of the AIEI framework, in addition to static code metrics, we leverage author and team-specific (organizational) information, covers +20 languages, and +1K different projects.

Software {reliability, defect, bug, fault} prediction — Closely related to software quality prediction are sub categories such as defect, bug, vulnerability, and system prediction.

Wang, *et al.* [26] proposed a representation learning algorithm to learn the semantic representation of code from token vectors that are derived from Abstract Syntax Trees (ASTs) in preference to manual feature engineering. They leveraged Deep Belief Networks (DBS) and validated their approach on 10 open source projects for within-project and cross-product defect prediction. The scale of their evaluated datasets has less than 1K files and the reported class imbalance ranges from 10% to 50%.

Paterson *et al.* [18] presented a test case prioritization strategy based on defect prediction for Java on 6 real-world programs instead of coverage-based approaches. They showed that using defect prediction to prioritize test cases reduces the number of test cases required to find a fault by on average 9.48% when compared with existing coverage-based strategies.

Fan *et al.* [6] proposed a defect prediction framework based on an attention-based recurrent neural network deep learning architecture. They leveraged abstract syntax trees (ASTs) to learn syntactic and semantic features and validated their approach on 7 open-source Java projects using the F1-measure (mean=0.56) and AUC (mean=0.71). Information on the scale of the datasets could not be verified.

Pandey *et al.* [17] performed an in-depth literature review and analysis of over 154 articles on Software Fault Prediction (SFP) spanning a period from 1990 to 2019. They found that ML approaches had higher AUC performance than classical statistical methods (mean AUC of 0.78 to 0.84) and highlighted data quality, overfitting, and class imbalance to be key challenges.

Hershkovich *et al.* [10] proposed a test prioritization approach by learning a bug prediction model on 5 open-source projects. They demonstrated that the model can detect more than double the number of bugs compared to a coverage-oriented approach with a mean AUC of 0.94 and mean PRC of 0.32.

Zheng *et al.* [29] proposed a semi-supervised graph-based approach to perform balanced defect prediction. They compared their approach with other state-of-the-art deep learning-based methods (mean MCC of 0.3, 0.35, 0.48) and reported impressive results (MCC 0.844 - 0.958) for the case of extreme class imbalance.

Our work differs in the following aspects: i) we attempt code quality prediction under super extreme class imbalance, ii) our datasets, covered languages, and projects are larger in scope, iii) our framework focuses on automated ways to go from data to performant models under resource constraints and iv) our primary performance evaluation measure similar to [29] is using MCC to account for class imbalance bias when using metrics such as AUC, F1, precision (P), and recall (R).

III. METHODOLOGY

A. Problem formulation

We frame code quality prediction as a supervised binary classification task. Given $\mathcal{D} = \{\mathbf{x}_i, y_i\}_{i=1}^{n}$ with n data points, $\mathbf{X} \in \mathcal{R}^{n \times d}$ features, and $y_i \in \{0, 1\}$ labels, we are interested in predicting code changes that lead to adverse quality events (QEs). In our case n denotes the number of pull requests (we internally call diffs), $\mathbf{X}$ a feature space describing quantitative characteristics of diffs, and $\mathbf{y}$ is a label vector to denote if a diff $\mathbf{x}_i$ resulted in an adverse QE (i.e., $y_i = 1$). Due to the high bar on code quality the distribution of $\mathbf{y}$ is extremely imbalanced.

Given an initial model portfolio $\mathcal{M}_F = \{\mathbf{h}_j\}_{j=1}^{L}$ full of individual and diverse learners $\mathbf{h}_j$ the goal is to find automated means to go from $\mathbf{X}$ to a high performing model or set of models $\hat{\mathcal{M}} = \{\mathbf{h}_j\}_{j=1}^{l}$ efficiently under resource constraints, where L denotes the number of models in the full model portfolio, L' the number of models in the reduced model pool, and l the number of models in the final model generating predictions, with $l << L' << L$ and j an index variable (see **Figure 1**).

B. Super extreme class imbalance

Super extreme class imbalance requires special considerations when using modeling best practices such as in feature engineering, model selection, tuning, and cross validation. While class imbalance has been addressed at length in the community and acknowledged to be a challenge [17], not much research or empirical evidence can be found on the case where the imbalance is "super" extreme, i.e., $\sum y_i = 1/n < 0.1\%$ (see **Table I**).

For instance, during cross-validation, due to label scarcity, the sampling of instances for each class label could lead to folds without any label instances of the positive class even for stratified k-fold cross validation. Such failure folds bias our model inference estimates when quantifying aggregate model performance to assess generalizability. We performed a sensitivity analysis to empirically determine the lower bound on the minimum required training samples to efficiently train the full model pool with minimal effort and resource overhead. At least 10K samples are required for Stratified K-Fold cross validation and 100K samples for K-Fold cross validation under super extreme class imbalance.

TABLE I
CLASS IMBALANCE SEVERITY LEVELS. A DESCRIPTION OF DIFFERENT CLASS IMBALANCE DEGREE CLASSIFICATIONS AND THEIR PROPORTIONS.

Imbalance degree	Minority class (y=1) proportion
Mild	$20 - 40\%$
Moderate	$1 - 20\%$
Extreme	$< 1\%$
Super extreme	$\mathbf{< 0.1\%}$

C. Automated incremental effort investment (AIEI) framework

At a high level, our framework architecture consists of a full model pool $\mathcal{M}_F$, a reduced model pool $\mathcal{M}_R$ and our final model(s) $\hat{\mathcal{M}}$, which leverages a meta ensemble component $\sum_j \hat{\mathbf{h}}(\cdot)$ to fine-tune our inferences and generate code quality predictions. Our focus is on being intentional on how we discover model insights early in the modeling process and correspondingly adjust training effort investments through an agile incremental process to reach competitive model performance faster. Rather than brute forcing through all models with all available data and analyses steps, providing early indicators of model performance and resource utilization can help the modelers to be more efficient with their own time and their use of constrained system resources. The idea is simple. Initially, assume we have a training investment budget of E that we evenly distribute equally across the full model pool $\mathcal{M}_F$ to gain a broad understanding of model behavior and various learning settings (e.g., feature space, architecture configurations, model parameters) (see **Eq. 1**). Here E refers to effort investments such as the chosen training size and effort spent on modeling due diligence.

$$E(\mathcal{M}_F) = \sum_j 1/L \tag{1}$$

In Step-1 we train $\mathcal{M}_F$ with a reduced set of training data samples that we empirically determined to avoid the bias introduced by the super extreme class imbalance. The goal is to quickly discover and learn the top candidate models that are optimized for prediction performance, runtime, and resource utilization efficiency. In Step-2, we allocate future effort investments with larger training data, rigorous model validation, and modeling due diligence to assess prediction generalizability. By doing so, we alleviate the need to iterate through the full space of model and parameter combinations freeing up time and resources for the most promising path forward.

$\mathcal{M}_F$ consists of a set of available individual learners with the assumption that $\mathcal{M}_F$ contains a variety of different model types, runtime, and resource utilization characteristics [2]. In practice this model pool is dynamically changing as new models or improvements to existing models are being developed and become available. To obtain the reduced model pool of top candidate models, we perform the following steps (see **Algorithm 1**), where i is an index variable, k the number of folds, t_k the run time and r_k the resource utilization of the models, which is used together with the model performances

[2]For this study we used the following models: AdaBoost, Bagging, BernoulliNB, CalibratedCV, CategoricalNB, ClassifierChain, ComplementNB, DecisionTree, ExtraTree, GaussianNB, GradientBoosting, HistGradientBoosting, KNeighbors, LabelPropagation, LabelSpreading, LinearDiscriminantAnalysis, LinearSVC, LogisticRegression, LogisticRegressionCV, MLP, MultiOutput, MultinomialNB, Nearest-Centroid, NuSVC, OneVsOne, OneVsRest, OutputCode, PassiveAggressive, Perceptron, QuadraticDiscriminantAnalysis, RadiusNeighbors, RandomForest, Ridge, RidgeCV, SGD, SVC, Stacking, Voting, XGB, LGBM, and GaussianProcess.

to rank the full model pool. Based on the ranking the modeler chooses the optimal cut off point.

Algorithm 1 Step-1: Reduced model pool identification

Input: $\mathcal{M}_F, \mathbf{X}, \mathbf{y}, \Theta$
Output: $\mathcal{M}_R, \lambda, n_\gamma$
1: Compute super extreme class imbalance rate $\lambda = \frac{1}{n}\sum_i y_i = 1$
2: Compute data batches of different sizes $\mathbf{s} = [n_1, n_2, n_3, ..., n_i]$, with $n_i = 10^i$
3: **for** <all s_i> **do**
4: Compute cv-folds $\{F\}_i^k$ with label distribution $\hat{y} = [1-\lambda, \lambda]$
5: **for** <all cv-folds $\{F\}_i^k$> **do**
Check if fold contains minority class misses, i.e. $\sum y_i = 0$
6: **end for**
7: Compute cv-folds failure rate distribution $r_i = (\{F\}_i^k, \mathbf{s})$
8: **end for**
9: Compute the lower bound $n_\gamma = \arg\min(\mathbf{r}, \mathbf{s})$, s.t. $\bar{r}_i = 0, min\{s_i\}$
10: **for** <all $\mathcal{M}_F$, n_γ> **do**
11: Compute $\mathbf{y}$, $\mathbf{h}_j, t_k, r_k$
12: **end for**
13: Compute model ranking rank($\mathcal{M}_F$)
14: Resolve ties by incorporating model performance of $\mathbf{h}_j$ and t_k, r_k

In Step-2, we perform full model training with all data and rigorous model validation on $\mathcal{M}_R$. We compute the full model performance distribution to assess generalization performance and filter out any models that exhibit high variability. We then re-rank the remaining model pool by performance and resource utilization characteristics similar to Step-1. Lastly, we aggregate model predictions to fine tune the predictions as described next.

D. Top-K Meta Ensemble (TKME)

We use a Top-K Meta Ensemble (TKME) to fine tune the model performance by combining the hypothesis space from $\mathcal{M}_R = \{\mathbf{h}_1, \mathbf{h}_2, \mathbf{h}_3, ..., \mathbf{h}_{L'}\}$ in order to make the final predictions. Various combination schemes are possible such as majority, averaging, stacking, and their weighted or non-weighted derivatives [20]. The main assumption is that combining individual learners into an aggregate prediction will lead to increased model performance. In this study, we start with a simple majority voting scheme of the top performing models in the reduced model pool (see **Eq. 2**).

$$\arg\max_{k\in\{0,1\}} \sum_{i,c} I_k(y) \tag{2}$$

Given $\mathcal{M}_R$ we use $\phi = \sum_{i=1}^{\hat{M}} \mathbf{h}_i$ to combine the top-k models, with $k \in \{2n+1\}_{n=1}^{m}$ to prevent ties in the aggregated model decisions. We cap k to be $k < 10$, $k = \{3, 5, 7, 9\}$, given empirical evidence that beyond 10 combining individual models leads to diminishing returns when trading off model run times vs. performance. The benefits of the AIEI framework are i) shorter modeling cycles, ii) faster time-to-failure, iii) increased resource efficiency, and iv) higher agility to get from data to a good performing model or set of models.

E. Performance metrics

The performance metric we primarily look at is the Matthew's Correlation Coefficient (MCC) [1], a more complete and reliable alternative than the F1 measure, AUC, or Accuracy especially for problems that involve super extreme class imbalance [28]. MCC produces a high score only if all elements of the confusion matrix perform well.

IV. Experiments and results

For all our experiments, we used a single machine with 114GB RAM, 24 cores, 500GB disk space, running CentOS (version 8), and no GPU. We found that AIEI allows us to incrementally build intuition on model performance, runtime, and resource characteristics to identify top model candidates and reach competitive model performance efficiently.

We can use the model predictions to prioritize the diffs and associated test cases to gain savings in diff review time (DRT) and resource utilization (RU). If a diff is predicted to not cause a QE incident, we can relax or remove the review and test requirements for this diff. By review time, we refer to the time it takes to review the code change, the time it takes to run the associated tests, and in the case of QEs, the incident management and root causing time. With resource utilization we refer to the number of tests that need to be run per diff, which directly maps to the actual resource utilization profile of our testing infrastructure.

A. Baseline and model comparison for test case prioritization efficiency

We computed two baseline models (BM) we call BM-50 and BM-25. BM-50 simulated the predictive behavior of a random chance model (e.g. 50% of diffs were randomly chosen to undergo review and testing efforts). BM-25 simulated a policy where we only test 25% of all code changes to stay within certain capacity constraints. For both baselines we used a super extreme class imbalance ratio and a sample size of 10K diffs/day. For each model we computed the confusion matrix and assigned cost estimates to compute % savings between our approach, models from the reduced model pool, and the two baseline models (see **Table II**) and (see **Table III**). Cost measures have been normalized to account for anonymity and internal publication policies.

We found AIEI to improve DRT by 2.5% compared to the best performing model. In comparison to the baseline models BM-25 and BM-50 we observed a 99.3% and 99.6% improvement, respectively. In the case of RU we observed a 9.3% improvement.

B. Real-world dataset

We have collected a dataset of internal pull requests (aka diffs) from the period between 2021-01-01 to 2021-06-30 ($n = +1.5M$). To provide a sense of rough scale and diversity, the data comprise $+1K$ top level directories and $+20$ languages.

From the raw data, we computed a set of simple human-engineered features at the diff level such as meta-information

TABLE II
SIMULATION STUDY OF MODEL PERFORMANCE IMPACT ON TEST CASE PRIORITIZATION WITH RESPECT TO REVIEW TIME. TP=TRUE POSITIVE, FP=FALSE POSITIVE, TN=TRUE NEGATIVE, FN=FALSE NEGATIVE ARE REPORTED IN PERCENTAGES INDICATED BY A *. WE BOLDFACED AIEI'S PERFORMANCE.

Rank	Model	TP*	FP*	TN*	FN*	Cost
1	**AIEI (k=3)**	0.001	0	0.999	0	5,659
2	DecisionTree	0.001	0	0.998	0	5,808
3	AdaBoost	0	0	0.999	0.001	6,124
4	**AIEI (k=5)**	0.001	0.001	0.998	0	6,758
5	GradientBoosting	0	0.001	0.998	0.001	8,491
6	**AIEI (k=7)**	0.001	0.001	0.997	0	9,297
7	Perceptron	0	0.001	0.998	0.001	10,051
8	GaussianNB	0.001	0.003	0.996	0	15,048
9	LDA	0.001	0.003	0.996	0	15,307
10	NearestCentroid	0.001	0.003	0.996	0	15,624
11	Baseline (25%)	0	0.255	0.744	0.001	923,160
12	Baseline (50%)	0.001	0.504	0.495	0	181,7760

TABLE III
SIMULATION STUDY OF MODEL PERFORMANCE IMPACT ON TEST CASE PRIORITIZATION WITH RESPECT TO RESOURCE UTILIZATION.

Rank	Model	TP*	FP*	TN*	FN*	Cost
1	**AIEI (k=3)**	0.001	0	0.999	0	27
2	AdaBoost	0	0	0.999	0.001	30
3	DecisionTree	0.001	0	0.998	0	50
4	**AIEI (k=5)**	0.001	0.001	0.998	0	62
5	GradientBoosting	0	0.001	0.998	0.001	90
6	Perceptron	0	0.001	0.998	0.001	122
7	**AIEI (k=7)**	0.001	0.001	0.997	0	149
8	GaussianNB	0.001	0.003	0.996	0	315
9	LDA	0.001	0.003	0.996	0	322
10	NearestCentroid	0.001	0.003	0.996	0	331
11	Baseline (25%)	0	0.252	0.747	0.001	25,170
12	Baseline (50%)	0.001	0.497	0.502	0.001	49,700

(version count, repo name, test plan, summary plan), code metrics (code complexity, test coverage, churn, file/line count, comments), reviewer metrics (reviewer/subscriber count), growth metrics (percent change), author and team information, and linter flags.

A diff comprises one or multiple source code files and associated meta-data to characterize and track a code change throughout its life time. We used internal adverse quality events (QEs) data related to system reliability, which attributes QEs to diffs. All QEs usually have an association with the relevant diffs, which can include both, the diff that triggered the QE incident and the ones that remediated the root cause. For our purposes, we are mainly interested in the former to proactively identify the code changes that led to QEs. From the set of diffs that are attributed to a QE we identified the reversion diff (i.e., the diff that triggered the QE incident and got therefore reverted) and assigned a positive label to it ($y = 1$). All other diffs were labeled as negative ($y = 0$). We only focused on diffs that can fix the QE incident as in practice the first line of response is to quickly identify and mitigate the diff that triggered the QE.

C. Can we predict code quality under super extreme class imbalance?

We applied our AIEI framework to a real-world dataset described in **SectionIV-B**. We started with a full model pool of $+40$ shallow learners. In Step-1, we used a 0.7/0.15/0.15 train/valid/test split. We then used the computed lower bounds on the required training samples to train the full model pool with the minimum required samples. Out of the full model pool 7 learners achieved an MCC > 0.5, which we used to form the reduced model pool. In Step-2, we kept the train/valid/test split constant, but increased the dataset to a quarter million data points with a class imbalance of 0.1% for the training and validation set. To assess predictive generalization performance, we performed a stratified K-Fold cross-validation with 5 folds and computed the model performance distribution for the reduced model pool. Despite the class imbalance we did not perform any over- or under sampling nor class re-weighting. We found that we can predict code quality under super extreme class imbalance with an MCC of $+0.6$ (moderate to strong correlation) and that our results compete or outperform state-of-the-art shallow or deep learning approaches (see **Table IV**).

TABLE IV
COMPARISON OF OUR METHOD WITH OTHER SOTA MODELS. MODEL PERFORMANCE FOR ONE OF THE REPORTED DATASETS IN [29]

	MCC	Class imbalance
DPCAG	0.958	1.80%
AIEI	**0.652**	**0.10%**
DBN-CP (RF)	0.476	1.80%
GMNN	0.431	1.80%
DP-ARNN (RF)	0.389	1.80%
DP-CNN (RF)	0.388	1.80%
Node2defect	0.363	1.80%
DP-CNN (LR)	0.196	1.80%
DP-ARNN (LR)	0.163	1.80%
DBN-CP (LR)	0.075	1.80%

V. DISCUSSION

We found that AIEI enabled us to perform code quality prediction under super extreme class imbalance with practical significance. By leveraging the concept of incremental effort investment and computing empirical lower bounds on the minimum training data size allowed us to go from data to a performant model efficiently, discover early top model candidates, and save on unnecessary resource utilization. In [29], the authors compared several deep learning models to perform code defect prediction. In our experiments AIEI performed with a mean MCC of 0.652 and tight variance bounds (σ^2=0.003), despite the super extreme class imbalance, providing further evidence on the predictive generalizability of our approach (see **Table IV**). We hope these results can inform future meta-analysis studies.

A limitation of our approach is that we have not leveraged the textual code features to aid interpretability, i.e. once we know that a diff is likely to lead to a QE we would want to know what can be done about it at the code level. Nevertheless,

this initial study provides us with the confidence that code quality prediction under super extreme class imbalance can be addressed.

Drawing general conclusions from empirical studies in software engineering is difficult because any process depends on a potentially large number of relevant context variables [3]. Therefore, we cannot assume that the results of a study generalize beyond the specific environment in which it was conducted. Researchers become more confident in a theory when similar findings emerge in different contexts.

VI. Conclusion

We proposed a code quality prediction framework to efficiently learn meaningful predictive patterns under super extreme class imbalance. In the future, we plan on integrating new data sources into the learning problem and investigate code features. On the longer-term roadmap we plan to extend our work to other code quality dimensions.

VII. Acknowledgments

The authors thank Lawerence Chen, Tobi Akomolede, and Wes Dyer for their helpful discussions that prompted this work.

References

[1] Phi coefficient. https://en.wikipedia.org/wiki/Phi_coefficient. Accessed: 2022-02-01.

[2] Ayman Amin, Lars Grunske, and Alan Colman. An approach to software reliability prediction based on time series modeling. *Journal of Systems and Software*, 86(7):1923–1932, 2013.

[3] V.R. Basili, F. Shull, and F. Lanubile. Building knowledge through families of experiments. *IEEE Transactions on Software Engineering*, 25(4):456–473, 1999.

[4] Xinyun Chen, Dawn Song, and Yuandong Tian. Latent execution for neural program synthesis beyond domain-specific languages. In A. Beygelzimer, Y. Dauphin, P. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, 2021.

[5] Siwei Cui, Gang Zhao, Zeyu Dai, Luochao Wang, Ruihong Huang, and Jeff Huang. Pyinfer: Deep learning semantic type inference for python variables. *ArXiv*, abs/2106.14316, 2021.

[6] Guisheng Fan, Xuyang Diao, Huiqun Yu, Kang Yang, Liqiong Chen, and Autilia Vitiello. Software defect prediction via attention-based recurrent neural network. *Sci. Program.*, 2019, jan 2019.

[7] Somya Goyal and Pradeep Kumar Bhatia. Software quality prediction using machine learning techniques. In Manoj Kumar Sharma, Vijaypal Singh Dhaka, Thinagaran Perumal, Nilanjan Dey, and João Manuel R. S. Tavares, editors, *Innovations in Computational Intelligence and Computer Vision*, pages 551–560, Singapore, 2021. Springer Singapore.

[8] Sakib Haque, Aakash Bansal, Lingfei Wu, and Collin McMillan. Action word prediction for neural source code summarization. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 330–341, 2021.

[9] Mark Harman and Peter O'Hearn. From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis. In *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 1–23. IEEE, 2018.

[10] Eran Hershkovich, Roni Stern, Rui Abreu, and Amir Elmishali. Prioritized test generation guided by software fault prediction. In *2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 218–225, 2021.

[11] Kevin Jesse, Premkumar T. Devanbu, and Toufique Ahmed. Learning type annotation: Is big data enough? In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, page 1483–1486, New York, NY, USA, 2021. Association for Computing Machinery.

[12] Capers Jones and Olivier Bonsignour. *The Economics of Software Quality*. Addison-Wesley Professional, 1st edition, 2011.

[13] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E. Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering*, 39(6):757–773, 2013.

[14] Yasir Mahmood, Nazri Kama, Azri Azmi, Ahmad Salman Khan, and Mazlan Ali. Software effort estimation accuracy prediction of machine learning techniques: A systematic performance evaluation. *Software: Practice and Experience*, 52:39 – 65, 2022.

[15] Erik Meijer and Vikram Kapoor. The responsive enterprise: embracing the hacker way. *Communications of the ACM*, 57(12):38–43, 2014.

[16] Nachiappan Nagappan, Brendan Murphy, and Victor R. Basili. The influence of organizational structure on software quality. *2008 ACM/IEEE 30th International Conference on Software Engineering*, pages 521–530, 2008.

[17] Sushant Kumar Pandey, Ravi Bhushan Mishra, and Anil Kumar Tripathi. Machine learning based methods for software fault prediction: A survey. *Expert Systems with Applications*, 172:114595, 2021.

[18] David Paterson, Jose Campos, Rui Abreu, Gregory M. Kapfhammer, Gordon Fraser, and Phil McMinn. An empirical study on the use of defect prediction for test case prioritization. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 346–357, 2019.

[19] Sandeep Reddivari and Jayalakshmi Raman. Software quality prediction: An investigation based on machine learning. In *2019 IEEE 20th International Conference on Information Reuse and Integration for Data Science (IRI)*, pages 115–122, 2019.

[20] Lior Rokach. Ensemble methods for classifiers. In *Data mining and knowledge discovery handbook*, pages 957–980. Springer, 2005.

[21] Baptiste Rozière, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

[22] Ensheng Shi, Yanlin Wang, Lun Du, Junjie Chen, Shi Han, Hongyu Zhang, Dongmei Zhang, and Hongbin Sun. On the evaluation of neural code summarization. In *International Conference on Software Engineering (ICSE'22)*, February 2022.

[23] Alexey Svyatkovskiy, Ying Zhao, Shengyu Fu, and Neel Sundaresan. Pythia: Ai-assisted code completion system. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery amp; Data Mining*, KDD '19, page 2727–2735, New York, NY, USA, 2019. Association for Computing Machinery.

[24] Pratvina Talele and Rashmi Phalnikar. Software requirements classification and prioritisation using machine learning. In Amit Joshi, Mahdi Khosravy, and Neeraj Gupta, editors, *Machine Learning for Predictive Analysis*, pages 257–267, Singapore, 2021. Springer Singapore.

[25] Ke Wang and Zhendong Su. Blended, precise semantic program embeddings. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 121–134, New York, NY, USA, 2020. Association for Computing Machinery.

[26] Song Wang, Taiyue Liu, and Lin Tan. Automatically learning semantic features for defect prediction. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 297–308. IEEE, 2016.

[27] Yu Wang, Ke Wang, Fengjuan Gao, and Linzhang Wang. Learning semantic program embeddings with graph interval neural network. *Proc. ACM Program. Lang.*, 4(OOPSLA), nov 2020.

[28] Jingxiu Yao and Martin Shepperd. Assessing software defection prediction performance: Why using the matthews correlation coefficient matters. In *Proceedings of the Evaluation and Assessment in Software Engineering*, pages 120–129. 2020.

[29] Xianda Zheng, Yuan-Fang Li, Huan Gao, Yuncheng Hua, and Guilin Qi. Towards balanced defect prediction with better information propagation. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(1):759–767, May 2021.