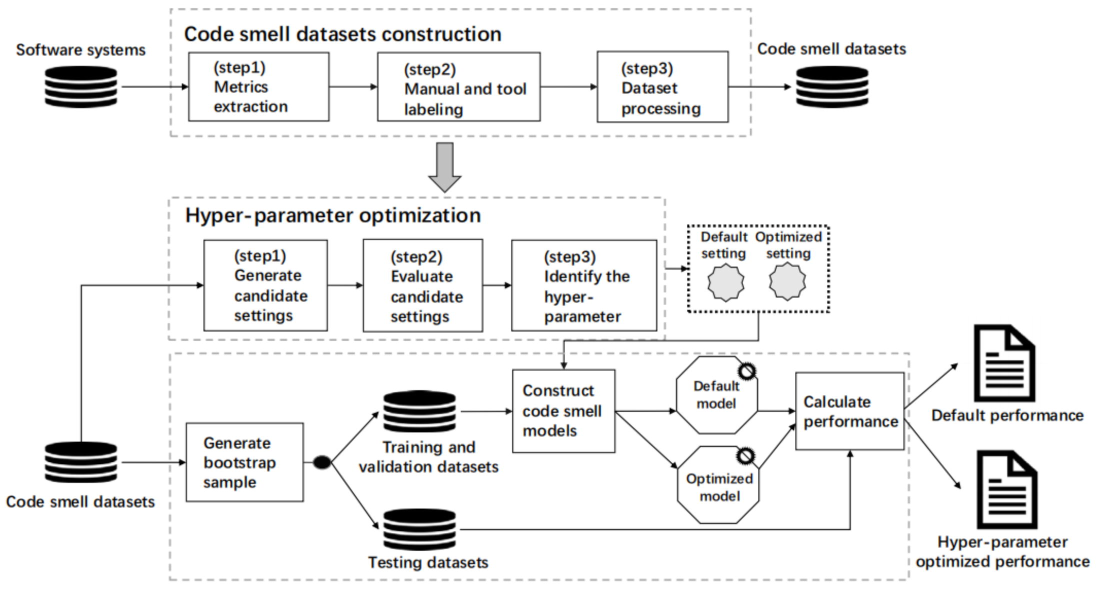
Improving Machine Learning-based Code Smell Detection via Hyper-parameter Optimization

Lei Shen[†], Wangshu Liu[†§*] , Xiang Chen[‡§], Qing Gu[§], Xuejun Liu[†]
[†]*School of Computer Science and Technology, Nanjing Tech University*, Nanjing, China
[‡]*School of Information Science and Technology, Nantong University*, Nantong, China
[§]*State Key Laboratory for Novel Software Technology, Nanjing University*, Nanjing, China


Software systems
Code smell datasets construction
(step1) Metrics extraction
(step2) Manual and tool labeling
(step3) Dataset processing
Code smell datasets
Hyper-parameter optimization
(step1) Generate candidate settings
(step2) Evaluate candidate settings
(step3) Identify the hyper-parameter
Default setting
Optimized setting
Code smell datasets
Generate bootstrap sample
Training and validation datasets
Testing datasets
Construct code smell models
Default model
Optimized model
Calculate performance
Default performance
Hyper-parameter optimized performance

A. Data Collection

Qualitas Corpus (QC) datasets , which cover more than 100 projects, have been widely studied in machine learning-based methods for code smell detection. Fontana et al. manually sought the missing third-party libraries to resolve class dependencies and chose 74 compilable projects among them. We carefully verify Fontana's datasets and find issues related to data quality, such as sample sparsity and class imbalance, which may affect the experimental validation. To obtain a reliable result, in this study, we define two collection criteria for project selection:

1) The number of samples must be more than a suitable value (e.g., 350),

2) The class ratio between non-smell and smelled samples must be restricted within a suitable range (e.g., $1 \leq$ class ratio < 14).

Finally, 8 of the 74 projects are selected as experimental subjects in our experiments. Table I provides statistical information about the selected projects. In the last column, *Data Class* represents the class-level code smell, while *Feature Envy* belongs to the method-level code smell.

TABLE I
SUMMARY OF STATISTICAL INFORMATION ABOUT THE SELECTED PROJECTS

Name	LOC	Package	Samples (Smelled) Data Class	Samples (Smelled) Feature Envy
castor	213,479	149	1,351 (101)	11,050 (1,129)
exoportal	102,803	382	1,801 (169)	10,264 (895)
fitlibrary	25,691	108	716 (89)	3,755 (479)
itext	117,757	24	479 (46)	5,657 (680)
jena	117,117	48	815 (64)	8,077 (1,208)
jspwiki	69,144	40	367 (28)	2,531 (319)
wct	69,698	85	472 (70)	4,971 (345)
xalan	312,068	42	992 (77)	9,883 (1,167)

B. Classifiers

In our experiments, six well-established classifier families have been involved. To guarantee the generalization of our empirical results, we choose one representative for each family and implement them by *python*..

In the decision tree family, we use a typical decision tree algorithm **CART** (CT), which is similar to C4.5. In brief, the construction process can constantly split nodes to build a binary tree model based on the information gain of features. Unlike C4.5, CART does not calculate “if-then-else” rule sets and can be compatible with the regression task.

In the neighbors-based family, **K-Nearest Neighbors** (KNN) is the classical “lazy” supervised learning algorithm. The principle behind KNN is that labeling neighbor close to each other as the same class. But KNN is sensitive to class imbalance, especially for minority class, and requires more computation when the dataset contains more features.

In the probabilistic model family, we use **Naive Bayes** (NB), which is a supervised learning algorithm based on Bayes’ theorem. Naive Bayes requires a strong (or naive) independence assumption between the features. But in software data mining, even the “naive” assumption is violated, the prediction performance is fairly good as other classifiers.

In the ensemble model family, we choose the widely used algorithm **Random Forest** (RF), which is a standard perturb-and-combine technique. Random Forest preserves the inter pretable benefits of the base estimator (i.e., decision tree), and improves the diversity of trees by splitting nodes within a randomly selected feature subset.

In the decision rule family, we implement **RuleFit** (RULE),which is a learning algorithm that aims at creating logical rules for classification. As compensation for CART, the decision rule can express rules in the form of “if-then”.

In the support vector machine family, we carry out **LibSVM**(SVM), which supports a variety of kernel functions. It is useful for SVM to apply to the dataset with a small sample size.However, the configuration of the kernel function and penalty factor can directly influence SVM’s predictive power.

In our study, we extend the work of Tantithamthavorn et al. by exploring more hyper-parameters and their candidate values. Table II provides details of the analyzed hyper-parameters.

TABLE II
HYPER-PARAMETER TUNED FOR DIFFERENT CLASSIFIERS

Classifier	Hyper-parameters	Default	Tuning Range	Description
CART	$min\_samples\_split$	2	[0.1, 0.9]	The minimum number of samples required to split an internal node
	$min\_samples\_leaf$	1	[0.1, 0.5]	The minimum number of samples required to be at a leaf node
	$max\_features$	None	[sqrt, log2, None]	The number of features to consider when looking for the best split
	$criterion$	gini	[gini, entropy]	The function to measure the quality of a split
K-Nearest Neighbors	$n\_neighbors$	5	[1, 19]	The number of neighbors to be used for queries
	p	2	[1, 3]	The power parameter for the Minkowski metric
	$weights$	uniform	[uniform, distance]	The weight function used in prediction
Naive Bayes	α	1	[0.1, 1]	The Laplace smoothing parameter
	fit_prior	True	[True, False]	The indicator decised whether to learn class prior probability or not
Random Forest	$n\_estimators$	10	[10, 50]	The number of trees in the forest
	$max\_features$	auto	[sqrt, log2, auto]	The number of features to consider when looking for the best split
	$criterion$	gini	[gini, entropy]	The function to measure the quality of a split
RuleFit	$min\_samples\_split$	2	[0.1, 0.95]	The minimum number of samples required to split an internal node for each decision tree
	$max\_features$	auto	[sqrt, log2, auto]	The number of features to consider when looking for the best split
LibSVM	C	1	[0.25, 3.75]	The penalty parameter of the error term
	γ	1	[0.1, 1]	The kernel coefficient for 'rbf', 'poly' and 'sigmoid'
	$kernel$	rbf	[rbf, poly, sigmoid, linear]	The kernel type to be used in the algorithm

C. Optimizers for Hyper-parameter optimization

For some specific classifiers, many hyper-parameter value combinations are needed to explore to achieve the best performance, while in *Default* setting, it is insufficient and may lead to the sub-optimal result.Choosing the optimal one from hundreds of combinations is a challenge in hyper-parameter optimization. Here, we mainly use four popular optimizers:grid search, random search, Bayesian optimization, and differential evolution.

Grid search (Grid) is an exhaustive searching algorithm. Grid examines every combination among all candidate hyper parameter spaces. Thus, the optimizer Grid is straightforward but inefficient, mainly when the dataset contains many features. For continuous value, it is necessary to convert them manually to a bounded discrete value set before optimizing.

Random search (Random) is similar to grid search. Instead of searching all candidate combinations, Random limits the explored number by the specified number of iterations. In each iteration, the search method randomly chooses one candidate as the verified hyper parameter setting.

Bayesian optimization (Bayes) intuitively optimizes hyper parameters based on historical results. First, we often construct a surrogate model to approximate the detection performance over all hyper-parameter distribution. Second, the surrogate's global optimum is calculated as a posterior sample to update the foregoing model. Last, we replicate the above steps until the iteration threshold, similar to the random search, is reached.

Differential evolution (DE) is one of the evolutionary optimization techniques. Most heuristic search algorithms are inspired by natural selection. After mutation and crossover,individuals in the population with high-quality genes may be preserved. DE completely simulates the process of individual mutation, crossover, and population selection. Especially, the new individuals in DE inherit directly from their parents, thus compared to other techniques (e.g.,genetic algorithm), DE can provide a more stable and better global solution.

In the experiment, for a fair comparison, all optimizers except non-parameter Grid fix their iteration threshold at 30.Furthermore, DE contains more impact factors, and we set the mutation probability as 0.8 and the population size as 30.

D. Evaluation

In our experiments, we choose AUC (Area Under ROC Curve) to measure each constructed model’s performance.. For each combination test, we conduct 10-fold cross-validation to evaluate overall performance throughout the whole dataset.

Furthermore, to determine whether there is a significant difference between the two methods, we employ Cohen’s *d* effect size, which is widely studied by previous mining software repositories researchers. Cohen’s *d* is a standard effect indicator, and can be calculated as follows:

$$Cohen's\ d = \frac{\mu_1 - \mu_2}{s_p}$$

where μ_i is the mean of the respective method, and s_p is the pooled standard deviation defined by Cohen. We also follow Cohen’s suggestion to measure effect size based on four Cohen’s d thresholds. Finally, in this paper, the difference between two methods is defined as *large*, *medium*, *small*, and *negligible*. For example, *large* indicates there is a large difference between the performance of two methods.

$$effect\ size = \begin{cases} negligible & ,\ Cohen's\ d \leq 0.2 \\ small & ,\ 0.2 < Cohen's\ d \leq 0.5 \\ medium & ,\ 0.5 < Cohen's\ d \leq 0.8 \\ large & ,\ 0.8 < Cohen's\ d \end{cases}$$