Xu YS, Jia XY, Wu F *et al.* Automatically identifying calling-prone higher-order functions of Scala programs to assist testers. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 35(6): 1278–1294 Nov. 2020. DOI 10.1007/s11390-020-0526-y

Automatically Identifying Calling-Prone Higher-Order Functions of Scala Programs to Assist Testers

Yi-Sen Xu[1], Xiang-Yang Jia[1], *Member, CCF*, Fan Wu[2], Lingbo Li[2], and
Ji-Feng Xuan[1,*], *Member, CCF, ACM, IEEE*

[1] *School of Computer Science, Wuhan University, Wuhan 430072, China*
[2] *Turing Intelligence Technology Limited, London, EC2Y 9ST, U.K.*

E-mail: {xuyisen, jxy}@whu.edu.cn; {fan, lingbo}@turintech.ai; jxuan@whu.edu.cn

Received April 11, 2020; revised October 31, 2020.

Abstract For the rapid development of internetware, functional programming languages, such as Haskell and Scala, can be used to implement complex domain-specific applications. In functional programming languages, a higher-order function is a function that takes functions as parameters or returns a function. Using higher-order functions in programs can increase the generality and reduce the redundancy of source code. To test a higher-order function, a tester needs to check the requirements and write another function as the test input. However, due to the complex structure of higher-order functions, testing higher-order functions is a time-consuming and labor-intensive task. Testers have to spend an amount of manual effort in testing all higher-order functions. Such testing is infeasible if the time budget is limited, such as a period before a project release. In practice, not every higher-order function is actually called. We refer to higher-order functions that are about to be called as calling-prone ones. Calling-prone higher-order functions should be tested first. In this paper, we propose an automatic approach, namely PHOF, which predicts whether a higher-order function of Scala programs will be called in the future, i.e., identifying calling-prone higher-order functions. Our approach can assist testers to reduce the number of higher-order functions of Scala programs under test. In PHOF, we extracted 24 features from source code and logs to train a predictive model based on known higher-order function calls. We empirically evaluated our approach on 4 832 higher-order functions from 27 real-world Scala projects. Experimental results show that PHOF based on the random forest algorithm and the Synthetic Minority Oversampling Technique Processing strategy (SMOTE) performs well in the prediction of calls of higher-order functions. Our work can be used to support the scheduling of limited test resources.

Keywords higher-order function, testing tool, test management, Scala program, internetware

1 Introduction

Internetware, a widely-used software paradigm, connects domain applications via Internet-based computing [1, 2]. For the rapid development of internetware, functional programming languages, such as Haskell, Scala, and Lisp, can be used to implement complex domain-specific applications. As an important feature of functional programming languages, higher-order functions are a family of functions that take functions as inputs or return functions. Due to the high scalability, algorithms written in higher-order functions can be generalized by changing their input or output functions [3, 4]. Moreover, higher-order functions are derived from the field of mathematics. Thus, using higher-order functions can make source code concise and precise [5, 6].

Unit testing, i.e., testing a function, is to execute

Regular Paper
Special Section on Software Systems 2020
A preliminary version of the paper was published in the Proceedings of Internetware 2019.
This work is supported by the National Key Research and Development Program of China under Grant No. 2018YFB1003901, the National Natural Science Foundation of China under Grant No. 61872273, and Advanced Research Projects of the 13th Five-Year Plan of Civil Aerospace Technology, Intelligent Distribution Technology of Domestic Satellite Information under Grant No. B0301.
*Corresponding Author

paths in a function to detect hidden faults[7,8]. Many current testing tools, such as JUnit[1] in Java and QuickCheck in Haskell[2], can support the test management and execution of first-order functions (i.e., non-higher-order functions). However, it is not easy to test higher-order functions with these testing tools. We list two difficulties of testing higher-order functions as follows. First, a higher-order function receives one or more functions as parameters. Testers have to write or call functions as parameters, which are expected to change program states of the higher-order function under test[7]. Second, a higher-order function can return a function as output. A returned function is expected to be consistent with the requirements. Testers have to fully check and understand requirements and then write tests for higher-order functions. Therefore, manually testing of higher-order functions becomes a time-consuming and labor-intensive task.

To save the time of testers in testing higher-order functions, we propose a view that testers can first test those higher-order functions that are likely to be called in the future within a limited budget. We refer to these functions likely to be called as calling-prone. That is, in a limited budget, instead of testing all higher-order functions, testers only need to test those higher-order functions that will be called in the future. To this end, we propose a predictive approach PHOF to identifying calling-prone higher-order functions and evaluate this approach on Scala programs. PHOF is a predictive model that is trained on known higher-order function calls. In PHOF, we extract 24 features from source code and logs to characterize whether a higher-order function is calling-prone.

We conduct an empirical evaluation on 4 832 higher-order functions from 27 real-world Scala projects and answer four research questions, including effectiveness, imbalanced data processing, impactful features, and efficiency. Experimental results show that PHOF based on the random forest algorithm and the Synthetic Minority Oversampling Technique Processing strategy (SMOTE) performs well in the prediction of calling-prone higher-order functions and reaches the accuracy of 0.803. The results indicate that SMOTE is effective among all techniques of imbalanced data processing under evaluation. We use Pearson correlation coefficient to rank the top-10 features most relevant to higher-order function calls in each project. The results suggest that the top-10 features can be used to partially represent the whole set of all features, but cannot fully replace the original set. The efficiency result reports that the average time cost is 473 seconds and can be accepted. Our proposed approach can assist testers to prioritize higher-order functions under test and save the cost of testing.

Application Scenario. Given a limited time budget, e.g., the time before a new release of the project, our approach PHOF can be used to identify whether a higher-order function can be called in the future. Then a tester can prioritize higher-order functions under test to avoid testing uncalled ones. Our work can be used to support the scheduling of limited test resources and reduce the cost of testers.

Extension. This paper is an extension of our previous work[9]. In this extension, we add new data processing of code clones, an extended experiment on 27 real-world Scala projects, and a detailed analysis of empirical results with new results of impactful features and efficiency. The new data processing of code clones can improve the reliability of function data via filtering out 50 sets of cloned functions. The experiment is extended from the original six projects to 27 projects and improves the generality of the proposed approach. The experimental result shows that the average accuracy without SMOTE is increased from 0.670 to 0.757, compared with our previous work. The analysis on impactful features shows that dominant features can partially represent the whole set of all features; the analysis on efficiency confirms that the average time cost is 473 seconds and suggests that our approach is efficient.

This paper makes the following major contributions.

• We propose an automatic approach, namely PHOF, which predicts whether a higher-order function will be called in the future. This approach is the first work that employs function features to prioritize higher-order functions for testers.

• We empirically evaluate 4 832 higher-order functions from 27 real-world Scala projects. We find that: the proposed approach is effective and the accuracy reaches 0.803; the imbalanced data processing is useful for the effectiveness while feature selection does not improve the result; the time cost of our approach is 473 seconds on average.

The rest of this paper is organized as follows. Section 2 shows the background and motivation of studying the prediction of higher-order function calls. Section 3 presents the proposed approach in our work. Section 4

[1] http://junit.org/junit5/, Oct. 2020.

[2] http://hackage.haskell.org/package/QuickCheck/, Oct. 2020.

presents the study setup, including four research questions and data preparation. Section 5 describes the results of our exploratory study. Section 6 discusses the threats to the validity. Section 7 lists related work and Section 8 concludes the paper.

2 Background and Motivation

In this section, we present the background and the motivation of our work.

2.1 Background

Scala is a programming language that supports both object-oriented programming and functional programming. Scala contains a powerful static type system and shares many features of functional programming languages with Standard ML and Haskell, including currying, type inference, and immutability [10]. The design of Scala is to make up for the deficiencies of the Java language. Source code written in Scala is compiled into bytecodes and runs on a virtual machine. For the compatibility, Scala programs can directly use libraries of Java programs.

As a feature of functional programming, Scala directly supports higher-order functions. In higher-order functions, functions can be used as input parameters or as outputs. Higher-order functions make the Scala programs useful in many scenarios, including constructing distributed systems and web projects. For instance, Nystrom [11] presented a Scala framework for experimenting with super-compilation techniques; Twitter has moved several basic frameworks from Ruby to Scala[3].

Fig.1 shows an excerpt of a real-world higher-order function subName() in project scala/scala[4]. The higher-order function subName(), defined inside another function ClassfileParser.sigToType(), is designed to get a subset of Class Name. Class Name has the same functions and fields as Class String, such as the method charAt() and the field length. The definition of this higher-order function contains one input parameter and a return type. The only input parameter isDelimiter() at line 5 is a first-order function, which receives a Char object as input and returns a Boolean object. The parameter isDelimiter() is called at line 7 to determine whether the Char object is a delimiter. The return type of the function subName() at line 5 is a Name object. The return statement of the function subName() locates at line 8.

```
1  private def sigToType(sym: Symbol, sig: Name):
       Type = {
2      var index = 0
3      val end = sig.length
4      ...
5      def subName(isDelimiter: Char => Boolean):
           Name = {
6          val start = index
7          while (! isDelimiter(sig.charAt(index))) {
               index += 1 }
8          sig.subName(start, index)
9      }
10     ...
11 }
```

Fig.1. Excerpt of a real-world higher-order function subName() from Class scala.tools.nsc.symtab.classfile.ClassfileParser in Project scala/scala.

Testing is an important phase to improve the quality of source code. ScalaTest, like JUnit in Java, is a testing framework for Scala and Java testers[5]. Many popular Scala projects, including the 27 projects in our study, have deployed ScalaTest to support test management and execution.

2.2 Motivation

Testing higher-order functions is difficult. For manually testing, to write a test case for a higher-order function, a tester needs to check the requirements of the function and then write another function as an input or output for the higher-order function being tested. However, for the input of a test case, creating a function parameter is different from creating a primitive parameter (e.g., an integer or a floating-point number) or an object parameter (an object of a class). For automatically testing, a study by Selakovic*et al.* [7] shows test cases generated by four test generation methods can reach higher code coverage when testing higher-order functions in JavaScript. In their study, several basic higher-order functions were tested, including higher-order functions of `filter()`, `map()`, and `then()`.

The resources available for testing are limited. Due to the complexity of the internal structure of higher-order functions, testing higher-order functions is a time-consuming and labor-intensive task. In a limited budget (such as the time period before release), it is in-

[3]http://news.ycombinator.com/item?id=542716, Oct. 2020.
[4]http://github.com/scala/scala/, Oct. 2020.
[5]http://scalatest.org/, Oct. 2020.

feasible to test all higher-order functions in a project. Instead of testing all higher-order functions, we consider testing higher-order functions that will be called in the future.

Motivated by the cost of testing higher-order functions, we propose a new approach, namely PHOF, which predicts whether a higher-order function will be called in the future. These higher-order functions that are most likely to be called can be tested first to save the cost. Our approach can assist testers to reduce the number of higher-order functions under test and reduce the cost of testing higher-order functions.

3 Predicting Callings for Higher-Order Functions

We show the overview, the feature extraction, and the learning algorithm of our proposed approach.

3.1 Overview

We refer to higher-order functions that are likely to be called in the future as calling-prone ones. In this paper, we design PHOF, which is an abbreviation for Prediction for Higher-Order Functions. PHOF is an automatic method of identifying calling-prone higher-order function. The problem of such identification can be viewed as a classification problem with binary labels: called or uncalled. If there is one or more calls to a higher-order function, the higher-order function is labeled as called, and if there is no call for the higher-order function, the higher-order function is labeled as uncalled. A tester can then use our approach to prioritize higher-order functions to save the cost of testing within a limited time budget.

Fig.2 shows the overview of our proposed approach PHOF. PHOF aims to predict whether a higher-order function is calling-prone. The input of PHOF is source code and logs of a Scala project. The output of PHOF is the binary prediction result of a new higher-order function. To build a predictive model in PHOF, we extract 24 features from source code and logs of Scala programs. Then we build a predictive model based on the higher-order functions with known calls and apply the model to predict results for new higher-order functions.

3.2 Feature Extraction

To build our model, we extract 24 features from source code and logs of Scala programs. These 24 features are divided into three groups: group CS — 16 features related to code statements (CS01 to CS16), group CP — 5 features related to function properties (CP01 to CP05), and group CG — 3 features extracted from git logs (CG01 to CG03). Table 1 lists the 24 features in PHOF. In group CS, we use the 16 features to represent the structure information of higher-order functions. In group CP, features related to function properties, such as the cyclomatic complexity and the executable lines of code, are used to reveal the overall state of a higher-order function. In group CG, we distill three features from commit logs since the information of commits may be critical for the calls of higher-order functions. Note that our study is conducted via intra-project evaluation; thus, the names of authors could be viewed as enumerated values.

Given the source code of a Scala project, we extract features of group CS and group CP by converting source files into abstract synthetic trees (ASTs), and then traverse ASTs to collect features related to

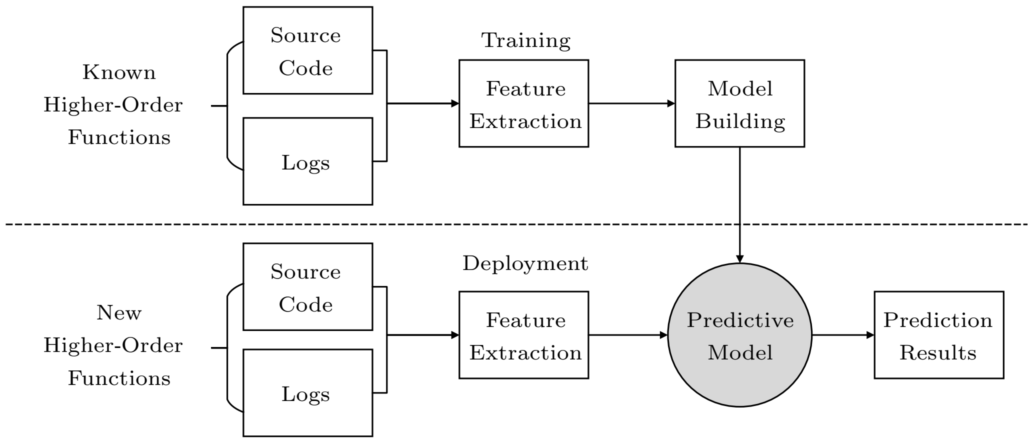


Fig.2. Overview of PHOF, an automated approach to predicting whether a higher-order function will be called in the future.

Table 1. Summary of 24 Extracted Features in Three Groups in PHOF

Group	Feature	Description
CS	CS01	Whether the higher-order function contains primitive types of parameters
	CS02	Whether the higher-order function contains a parameter that is a higher-order function
	CS03	Whether the higher-order function contains generic parameters, such as the parameter `A` in a class definition `class Stack[A]`
	CS04	Whether the higher-order function contains a return statement
	CS05	Number of parameters in the definition of the higher-order function
	CS06	Number of `for` statements in the higher-order function
	CS07	Number of `while` statements in the higher-order function
	CS08	Number of `else` statements in the higher-order function
	CS09	Number of `match` statements in the higher-order function
	CS10	Number of `if` statements in the higher-order function
	CS11	Number of `assign` statements in the higher-order function
	CS12	Number of lambda expressions in the higher-order function
	CS13	Number of `try` statements in the higher-order function
	CS14	Number of `apply` statements in the higher-order function
	CS15	Number of local variables that can be changed in the higher-order function
	CS16	Number of local variables that cannot be changed in the higher-order function
CP	CP01	Executable lines of code (eLoC) of the higher-order function
	CP02	Cyclomatic complexity of the higher-order function
	CP03	Number of style warnings in the higher-order function
	CP04	Containing input functions or output functions in the higher-order function (three values: functions only as input, functions only as output, and functions as both input and output)
	CP05	Modifier of the higher-order function (`public`, `protected`, `private`, or `default`)
CG	CG01	Name of the first author of the higher-order function
	CG02	Number of commits to source code of the higher-order function
	CG03	Number of authors of the higher-order function

code statements and function properties, such as the number of `if` statement, the executable lines of code (eLoC), and the cyclomatic complexity of higher-order functions. The cyclomatic complexity is a software metric of linearly independent path[12]. We also count the number of code style warnings in a higher-order function (see Subsection 4.1 for implementation).

For group CG (that is, log-related features), we extract logs from the version control system and collect all historical commits related to changes of higher-order functions. We then traverse these commits and collect features of authors and commits[13].

3.3 Learning Algorithms

PHOF uses a classification algorithm to build a predictive model. Any binary classification algorithm can be used as the learning model. We evaluate six algorithms in PHOF: C4.5, random forest, SVM, MLP, BayesNet, and SimpleLogistic. C4.5 is a decision tree algorithm and uses the information gain rate as a criterion of selecting branch attributes to achieve inductive classification of data[14]. Random forest is a classifier that contains multiple decision trees and its output category is determined by the mode of the output category of individual trees[15]. SVM is a classifier to find a hyperplane to segment samples[16]. The principle of segmentation is to maximize the intervals and to finally transform into a convex quadratic programming problem. MLP is a forward-structured artificial neural network that maps input vectors onto output vectors[17]. BayesNet is a classification algorithm based on the Bayes theorem, which can be used for predictive modeling[18]. SimpleLogistic is a widely used classification algorithm, which is a kind of linear classification[19].

The distribution of called and uncalled higher-order functions is imbalanced. For most of the machine learning algorithms, the issue of data imbalance may cause incorrect prediction results[20]. Thus, we adopt the Synthetic Minority Oversampling Technique Processing strategy (SMOTE) to address the data imbalance issue. The SMOTE strategy is a typical oversampling technique[21] of analyzing the minority samples and adding new samples to the dataset according to the minority samples to achieve the data balance.

4 Experimental Setup

In this section, we introduce the data preparation and the evaluation metrics.

4.1 Data Preparation

Our study aims to build a learning model to predict whether a higher-order function will be called in the future, i.e., called or uncalled. We mined 27 Scala projects and extracted features for the construction of classifiers. Datasets in this paper are publicly available[6].

Our work is to train a learning model, which requires sufficient data for higher-order functions. Thus, we selected 27 widely-used and open-sourced Scala repositories. The main steps of project selection are listed as follows. First, we sorted all Scala repositories in GitHub according to the stars[7]. We considered that a repository with many stars indicates that the quality of the repository is identified by many developers. We selected top-100 repositories with the most stars. Among these repositories, we removed three repositories, including `fpinscala/fpinscala` (a supplement material of practices in a book), `scala-exercises/scala-exercises` (exercises for many libraries of Scala), and `jacksu/utils4s` (learning materials for Scala and Spark) since these repositories are not software projects. Second, we leveraged SemanticDB, a tool of static program analysis, to extract definitions and callings of functions[8]. Third, we leveraged SourcererCC [22], a tool of token-based code clone detection, to detect the code clone in higher-order functions. We filtered out code clone groups in higher-order functions and selected the repositories that contain more than 50 higher-order functions as the repositories in the study. Table 2 shows the 4 832 higher-order functions from 27 Scala projects in the study. We present the steps of data processing as follows.

Higher-Order Function Identification. We employed the static analysis tool SemanticDB to extract the semantic structure, such as types and function signatures. Then we collected definitions and callings of functions. SemanticDB is a Scala library to analyze and compile Scala source code. The main steps to extract higher-order functions are as follows. First, we used SemanticDB to create a semantic database for each project. Second, we collected the definitions of higher-order functions from the semantic database if parameters or return values of the definition contain a function. Third, we filtered out override higher-order functions that are defined by default to implement an abstract function in a super class in the Scala language. The reason for such filtering is as follows. It is required to implement an abstract function with an override function in a sub class, but calling this override function is not required. Fourth, we extracted callings of higher-order functions by matching the function definitions. Therefore, we are able to extract definitions and calls of all higher-order functions in each project.

Code Clone Removal. Code clone is common in large software repositories. To avoid the impact of code clone on the experimental results, we leveraged SourcererCC, proposed by Sajnani *et al.* [22], to detect the code clone of higher-order functions in Scala projects. SourcererCC is a state-of-the-art tool of token-based code clone detection. SourcererCC employs tokens to represent code fragments and collects the frequency of each token. To detect code clones, SourcererCC obtains the similarity between two pieces of code via calculating the ratio between the frequencies of the same tokens and the frequencies of all involved tokens. This makes SourcererCC efficient in detecting similar code pairs in large-scale codebases [23]. In our study, we set the threshold of SourcererCC to 0.99, i.e., higher-order functions with the similarity greater than 0.99 will be considered as code clone groups. The reason for choosing 0.99 as the threshold is to fully remove the same code clones. From the 27 projects in the study, 50 clone groups are detected and filtered out. Each code clone group contains two or more higher-order functions. We further analyzed whether one group of code clones are called in the same calling-proneness, i.e., higher-order functions in a group are all called (or uncalled) by other functions. We checked all clone groups and find that only six out of 50 clone groups are not called in the same calling-proneness. This suggests that most of code clones of higher-order functions show the same calling-proneness.

Feature Extraction. We leveraged static analysis tools Scalamata[9] and ScalaStyle[10] to extract features in group CS and group CP (Subsection 3.2). Scalameta is a Scala library to parse Scala files and construct ASTs. ScalaStyle is an off-the-shelf checking tool of the code style; we leveraged it to extract the number of

[6] Phof. http://cstar.whu.edu.cn/p/phof/, Oct. 2020.

[7] http://github.com/search?l=Scala&o=desc&q=scala&s= stars&type=Repositories, Oct. 2020.

[8] http://scalameta.org/docs/semanticdb/guide.html, Oct. 2020.

[9] http://scalameta.org/, Oct. 2020.

[10] http://scalastyle.org/, Oct. 2020.

Table 2. Summary of 27 Scala Projects in the Study

Project	Abbreviation	#Stars ($\times10^3$)	eLoC ($\times10^3$)	#HOFs	#HOFs w/o Clone	Project Description
scalaz/scalaz	scalaz	4.4	35.4	991	987	A package for functional programming
scala/scala	scala	12.8	143.6	583	577	The programming language
zio/zio	zio	2.4	13.9	337	335	A toolkit for concurrent programming
lift/framework	framework	1.2	65.7	348	334	A web framework
akka/akka	akka	11.0	114.8	288	280	A tool for building applications
scalatest/scalatest	scalatest	0.9	420.7	168	162	A testing tool
tpolecat/doobie	doobie	1.7	14.7	162	160	A functional JDBC layer
sbt/sbt	sbt	4.2	34.9	157	157	A build tool
twitter/scalding	scalding	3.2	29.6	155	155	A toolkit for cascading
lampepfl/dotty	dotty	4.0	387.1	137	137	A compiler
typelevel/cats-effect	cat-effect	1.0	13.4	130	128	A functional runtime system
twitter/util	util	2.4	27.4	121	120	A package of utilities
ornicar/lila	lila	7.2	70.3	117	116	A chess game server
twitter/algebird	algebird	2.0	22.7	101	101	A tool for abstracting algebra
JetBrains/intellij-scala	intellij-scala	1.0	363.4	100	100	A plug-in for Intellij IDEA
polynote/polynote	polynote	3.9	16.1	99	96	A notebook
slick/slick	slick	2.4	20.8	93	93	A toolkit of database query and access
sangria-graphql/sangria	sangria	1.7	15.0	89	86	A GraphQL toolkit
http4s/http4s	http4s	1.9	29.1	82	82	A Scala interface for HTTP
twitter/finagle	finagle	7.7	63.0	82	81	A RPC system
playframework/playframework	playframework	11.6	41.5	77	77	A web framework
gatling/gatiling	gatling	4.8	25.6	73	73	A testing tool
scalikejdbc/scalikejdbc	scalikejdbc	1.1	23.5	71	71	A toolkit of database query and access
getquill/quill	quill	1.7	15.6	66	63	A toolkit of database query and access
json4s/json4s	json4s	1.3	11.8	57	57	A JSON toolkit
circe/circe	circe	2.0	6.9	95	55	A JSON toolkit
spotify/scio	scio	1.9	30.5	53	53	A toolkit for Apache Beam
Total		101.4	2 057.0	4 832	4 738	

Note: For the sake of space, each project will be denoted by their abbreviation in the following sections. #HOFs is short for the total number of higher-order functions. #HOFs w/o Clone denotes the number of higher-order functions whose cloned code has been removed.

code style warnings in higher-order functions. For features in group CG, we first used the Git API to extract Git logs and collected all historical commits related to the changes of higher-order functions [24]. We traversed these commits and collected the author information, including names, e-mails, and timestamps of all commits related to higher-order functions. Then we extracted the first author of the higher-order function according to the timestamp.

In our experiment, we used Weka for the implementation of machine learning algorithms[25]. Feature selection techniques and imbalanced processing methods are also integrated into Weka⑪.

All experiments were run on a laptop with an Intel Core i5-7360U 2.30 GHz CPU, 8 GByte memory, and an SSD disk of 256 GByte. The time cost of the evaluation was collected on this platform.

4.2 Evaluation Metrics

In the evaluation, we used four metrics, including precision, recall, F-measure, and accuracy. We define the four evaluation metrics based on true positive (TP), false positive (FP), true negative (TN), and false negative (FN). We list the definitions as follows:

- TP: number of higher-order functions in Category called that are predicted as called;
- FP: number of higher-order functions in Category uncalled that are predicted as called;
- TN: number of higher-order functions in Cate-

⑪http://www.cs.waikato.ac.nz/ml/weka/, Oct. 2020.

gory uncalled that are predicted as uncalled;

• FN: number of higher-order functions in Category called that are predicted as uncalled.

Then we define the precision, recall, $F1$ (short for F-measure), and accuracy as follows:

$$Precision(\text{called}) = \frac{TP}{TP+FP},$$

$$Precision(\text{uncalled}) = \frac{TN}{TN+FN},$$

$$Recall(\text{called}) = \frac{TP}{TP+FN},$$

$$Recall(\text{uncalled}) = \frac{TN}{TN+FP},$$

$$F1(\text{called}) = \frac{2 \times Precision(\text{called}) \times Recall(\text{called})}{Precision(\text{called}) + Recall(\text{called})},$$

$$F1(\text{uncalled}) = \frac{2 \times Precision(\text{uncalled}) \times Recall(\text{uncalled})}{Precision(\text{uncalled}) + Recall(\text{uncalled})},$$

$$Accuracy = \frac{TP+TN}{TP+TN+FP+FN}.$$

5 Empirical Results

We design four research questions (RQs) to evaluate our proposed approach, including effectiveness of our proposed approach, imbalance data processing strategies, feature correlation of callings of higher-order functions, and efficiency of our approach.

5.1 RQ1. How Effective Is Our Approach in Predicting Whether a Higher-Order Function Will Be Called in the Future?

We build a predictive model to predict whether a higher-order function can be called in the future. Machine learning algorithms play an important role in the prediction. We analyze the effectiveness of our proposed approach PHOF. Six classification algorithms are used in PHOF, including random forest, C4.5, SVM, MLP, BayesNet, and SimpleLogistic. In random forest, the number of decision trees is set by default to 100; in C4.5, the minimum number of instances per leaf is 2 and the confidence factor is set to 0.25; in SVM, the complexity is set to 1.0 and the calibrator is the logistic regression; in MLP, the learning rate for weight updates is set to 0.3; in BayesNet, the estimator is the SimpleEstimator; and in SimpleLogistic, the maximum number of iterations for LogitBoost is set to 500. The SMOTE strategy is combined with each classification algorithm to eliminate the risk of data imbalance.

We used 5-fold cross validation to evaluate the effectiveness of the experiment. For each project, we randomly divided the higher-order functions into five equal-sized folds. Then we built five rounds of experiments. In each round, one fold is used as a test set and the other four folds are used as a training set. Then the average of five rounds is reported as the result.

We present the evaluation of precision, recall, F-measure, and accuracy on individual projects. For the sake of space, Table 3 shows the prediction results of the top 10 projects, which contain the most higher-order functions. And we bold the maximum value of different algorithms in all metrics for each project. Among six algorithms under evaluation, the random forest beats the others in most projects. In 27 projects, random forest achieves the highest accuracy in six projects except projects `scalatest`, `sbt`, `scalding`, and `dotty`; SVM achieves the highest accuracy in projects `scalatest` and `scalding`; SimpleLogistic achieves the highest accuracy in project `scalding`, and C4.5 achieves the highest accuracy in project `dotty`. In the project `doobie`, the random forest performs best in all metrics, i.e., precision, recall, F-measure, and accuracy of class called and class uncalled. Moreover, project `doobie` is the best classified project with the accuracy of 0.956.

As shown in Table 3, random forest achieves the highest values of accuracy in most of the projects. Then we count the average of 27 projects in Table 4. As shown in Table 4, random forest achieves the highest value in six metrics except recall for uncalled ones. Meanwhile, MLP obtains the best recall for uncalled higher-order functions. All six algorithms exceed 0.700 in the accuracy. From Table 3 and Table 4, we find that the evaluation values of six algorithms are similar. Random forest performs well in most of the evaluation metrics while MLP performs well in the recall of uncalled higher-order functions.

Finging 1. Our proposed approach, PHOF is effective in predicting whether a higher-order function will be called in the future. Random forest and the other algorithms, i.e., C4.5, SVM, MLP, BayesNet, and SimpleLogistic, exceed the accuracy of 0.700 in the evaluation. Moreover, random forest performs better than the other algorithms.

5.2 RQ2. Can Imbalanced Data Processing Strategies Improve Prediction Results?

The data imbalance of two categories of higher-order functions may lead to inaccurate classification[20]. We

Table 3. Precision, Recall, F-Measure, and Accuracy of Prediction Results for the Top 10 Projects

Project	Algorithm	Called			Uncalled			Accuracy
		Precision	Recall	F-Measure	Precision	Recall	F-Measure	
scalaz	Random forest	**0.760**	0.787	**0.773**	0.677	**0.642**	**0.659**	**0.727**
	C4.5	0.704	0.802	0.750	0.645	0.516	0.573	0.685
	SVM	0.705	**0.830**	0.762	0.672	0.501	0.574	0.695
	MLP	0.688	0.821	0.749	0.644	0.464	0.539	0.675
	BayesNet	0.741	0.751	0.746	0.635	0.622	0.628	0.698
	SimpleLogistic	0.717	0.826	0.768	**0.680**	0.531	0.596	0.705
scala	Random forest	**0.833**	0.923	0.876	0.600	0.383	**0.468**	**0.799**
	C4.5	0.769	**1.000**	0.870	N/A	0.000	N/A	0.769
	SVM	0.805	0.959	0.876	0.625	0.226	0.331	0.790
	MLP	0.830	0.838	0.834	0.442	**0.429**	0.435	0.744
	BayesNet	0.809	0.944	0.871	0.576	0.256	0.354	0.785
	SimpleLogistic	0.805	0.964	**0.877**	**0.644**	0.218	0.326	0.792
zio	Random forest	**0.674**	**0.604**	**0.637**	**0.847**	0.883	**0.865**	**0.803**
	C4.5	0.577	0.469	0.517	0.802	0.862	0.831	0.749
	SVM	0.638	0.385	0.481	0.787	**0.912**	0.845	0.761
	MLP	0.652	**0.604**	0.627	0.846	0.870	0.858	0.794
	BayesNet	0.598	**0.604**	0.601	0.840	0.837	0.839	0.770
	SimpleLogistic	0.656	0.438	0.525	0.801	0.908	0.851	0.773
framework	Random forest	**0.656**	0.628	**0.642**	**0.749**	0.772	**0.760**	**0.713**
	C4.5	0.596	0.657	0.625	0.743	0.690	0.716	0.677
	SVM	0.633	0.504	0.561	0.698	0.797	0.744	0.677
	MLP	0.632	0.577	0.603	0.722	0.766	0.744	0.689
	BayesNet	0.567	**0.679**	0.618	0.741	0.640	0.687	0.656
	SimpleLogistic	0.610	0.445	0.515	0.675	**0.802**	0.733	0.656
akka	Random forest	0.815	**0.846**	**0.830**	**0.777**	0.737	**0.757**	**0.800**
	C4.5	0.784	0.741	0.762	0.669	0.720	0.694	0.732
	SVM	0.778	0.802	0.790	0.717	0.686	0.701	0.754
	MLP	0.782	0.821	0.801	0.736	0.686	0.711	0.764
	BayesNet	**0.831**	0.636	0.720	0.622	**0.822**	0.708	0.714
	SimpleLogistic	0.783	0.802	0.793	0.719	0.695	0.707	0.757
scalatest	Random forest	0.844	0.806	0.824	0.867	0.895	0.881	0.858
	C4.5	0.915	0.806	0.857	0.874	**0.947**	0.909	0.889
	SVM	**0.921**	**0.866**	**0.892**	**0.909**	**0.947**	**0.928**	**0.914**
	MLP	0.885	0.806	0.844	0.871	0.926	0.898	0.877
	BayesNet	0.817	0.731	0.772	0.824	0.884	0.853	0.821
	SimpleLogistic	0.918	0.836	0.875	0.891	**0.947**	0.918	0.901
doobie	Random forest	**1.000**	**0.930**	**0.964**	**0.896**	**1.000**	**0.945**	**0.956**
	C4.5	0.957	0.890	0.922	0.836	0.933	0.882	0.906
	SVM	0.690	0.890	0.777	0.645	0.333	0.440	0.681
	MLP	0.968	0.900	0.933	0.851	0.950	0.898	0.919
	BayesNet	0.741	0.830	0.783	0.646	0.517	0.574	0.713
	SimpleLogistic	0.848	0.890	0.868	0.800	0.733	0.765	0.831
sbt	Random forest	0.755	0.762	0.758	0.510	0.500	0.505	0.675
	C4.5	0.730	0.695	0.712	0.439	0.481	0.459	0.624
	SVM	0.695	**0.933**	**0.797**	**0.563**	0.173	0.265	0.682
	MLP	0.748	0.762	0.755	0.500	0.481	0.490	0.669
	BayesNet	**0.802**	0.771	0.786	0.571	**0.615**	**0.593**	**0.720**
	SimpleLogistic	0.736	0.743	0.739	0.471	0.462	0.466	0.650
scalding	Random forest	0.683	0.755	0.717	0.549	0.459	0.500	0.639
	C4.5	0.655	0.766	0.706	0.511	0.377	0.434	0.613
	SVM	0.667	**0.851**	**0.748**	**0.600**	0.344	0.437	**0.652**
	MLP	**0.692**	0.766	0.727	0.569	**0.475**	**0.518**	**0.652**
	BayesNet	0.658	0.798	0.721	0.537	0.361	0.431	0.626
	SimpleLogistic	0.682	0.798	0.735	0.578	0.426	0.491	**0.652**
dotty	Random forest	0.762	0.877	0.816	0.133	0.065	0.087	0.693
	C4.5	0.778	**0.991**	**0.871**	**0.500**	0.032	0.061	**0.774**
	SVM	0.771	0.953	0.852	0.167	0.032	0.054	0.745
	MLP	**0.808**	0.755	0.780	0.316	**0.387**	**0.348**	0.672
	BayesNet	0.798	0.934	0.861	0.462	0.194	0.273	0.766
	SimpleLogistic	0.798	0.934	0.861	0.462	0.194	0.273	0.766

Note: N/A: the value is not available since no higher-order function is predicted as an uncalled one.

Table 4. Average of Precision, Recall, F-Measure, and Accuracy of Prediction Results for All the 27 Projects

Algorithm	Called			Uncalled			Accuracy
	Precision	Recall	F-Measure	Precision	Recall	F-Measure	
Random forest	**0.766**	**0.779**	**0.771**	**0.671**	0.630	**0.645**	**0.757**
C4.5	0.734	0.743	0.729	0.657	0.552	0.635	0.721
SVM	0.734	0.750	0.734	0.592	0.543	0.552	0.717
MLP	0.755	0.736	0.743	0.632	**0.640**	0.634	0.731
BayesNet	0.737	0.741	0.733	0.622	0.583	0.586	0.719
SimpleLogistic	0.748	0.748	0.741	0.610	0.580	0.584	0.729

examine whether imbalanced data processing strategies can improve the prediction. Fig.3 presents the distribution of higher-order functions in both categories of each project. We find that the distributions of categories are imbalanced in 20 out of 27 projects.

We evaluated the effectiveness of imbalanced data processing techniques. We used three imbalanced data processing strategies, SMOTE, Adaptive Synthetic Sampling (ADASYN), and no strategy (called NoStrategy for short), to solve the imbalanced problem. As mentioned in Subsection 3.3, the SMOTE strategy is a typical oversampling technique [21]. ADASYN is another typical oversampling technique [26]. The key idea of ADASYN is to weight different minority samples according to the learning difficulty of data. ADASYN can synthesize the data for the minority class that is difficult to be modeled.

Fig.4 shows the evaluation values of precision, recall, F-measure, and accuracy of random forest with three sampling strategies in 27 projects. As shown in Fig.4, SMOTE achieves the maximum value in all seven metrics, i.e., the precision, recall, F-measure for both categories and the accuracy. NoStrategy achieves a higher value than ADASYN only in the recall for called higher-order functions.

Finding 2. Experiments show that the SMOTE strategy is an effective strategy of the imbalanced data processing. Compared with using no strategy, the precision, recall, F-measure and the accuracy can be improved with SMOTE.

5.3 RQ3. Which Features Are More Impactful on the Prediction Results?

Higher-order functions are expected to abstract the calling patterns of functions [27]. The features in PHOF are divided into three groups: code statements, function properties, and logs.

We tend to find out features that affect the prediction of higher-order functions calls. In each project, we used Pearson correlation coefficient to evaluate the correlation between a feature and the predicted result. Then we selected the top-10 features, which correlate the most with the prediction results. We referred to these top-10 features as dominant features.

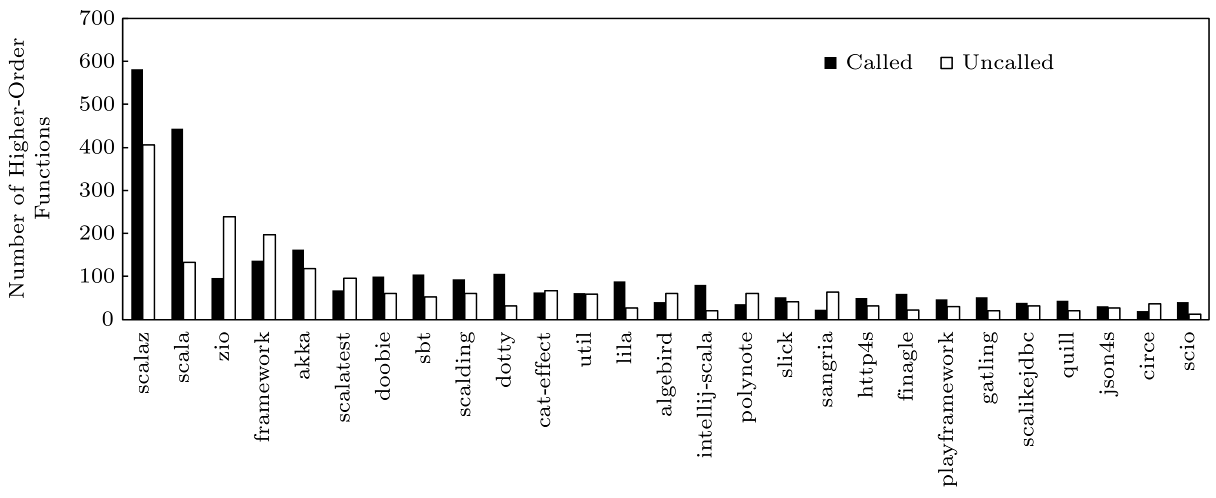


Fig.3. Distribution of called and uncalled higher-order functions in each project.

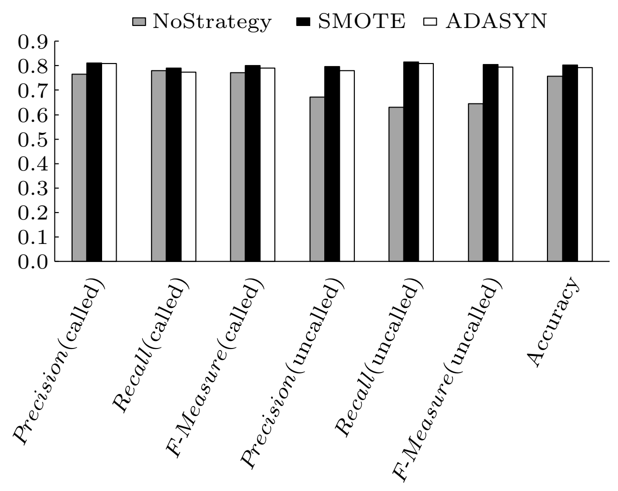


Fig.4. Average results of the random forest algorithm using three strategies of imbalanced data processing in 27 projects.

In statistics, Pearson correlation coefficient is used to measure the degree of the linear correlation between two variables X and Y and the coefficient value is between -1 and 1[28]. The absolute value of a coefficient is 1 if X and Y can completely linearly correlate and a coefficient is 0 if there exists no linear correlation. A positive coefficient indicates that Y increases when X increases; a negative coefficient indicates that Y decreases when X increases.

For each project in the experiment, the Pearson correlation coefficient between each feature and the predicted result is calculated; then we sort all features according to the absolute value of Pearson correlation coefficient. Table 5 presents the list of top-10 dominant features in the top 10 projects, which contain the most higher-order functions among all projects. CP01 (eLoC of higher-order functions) appears in 21 out of 27 projects. In projects `scalaz`, `zio`, `sbt`, `util`, and `gatling`, CP01 correlates the most with the predicted result; meanwhile, CP04 (containing input functions or output functions in the higher-order function) appears in 18 out of 27 projects. CP04 shows the greatest correlation with the predicated result in project `finagle`.

Table 5. List of Top-10 Dominant Features for the Top 10 Projects

Project	1	2	3	4	5	6	7	8	9	10
scalaz	**CP01**	CS03	CP02	CS09	**CP04**	CS16	CS05	CS12	CS14	CS10
scala	CP05	CG03	CS03	CS05	CG01	**CP01**	CS01	CS11	CS07	CS15
zio	**CP01**	CP02	CS16	CS07	CS10	CS11	CS15	CS14	CS03	CS09
framework	CS01	CP05	**CP04**	CS16	**CP01**	CS06	CS04	CG01	CP02	CS14
akka	CP05	**CP01**	CS04	CG01	CP02	CS10	CS16	CP03	CG03	CS08
scalatest	CP05	CS09	CS16	**CP01**	CP03	CS08	CS11	CS13	CS10	**CP04**
doobie	CS14	CS12	CP03	CG02	CS05	CG01	CS11	**CP04**	CP05	CS13
sbt	**CP01**	CP02	CS16	CS09	CP03	CS03	CS14	CG02	CS10	CS04
scalding	CS05	CG02	CP05	CS14	CS10	CG03	CP02	CS09	CG01	CS08
dotty	CS05	CS04	**CP01**	CP05	CS01	CS09	CP03	CS16	CS11	CP02
cat-effect	CP03	**CP01**	CS16	CS11	CS15	CG02	CS14	CS05	CP02	**CP04**
util	**CP01**	CP02	CS14	CS09	CS10	CG02	**CP04**	CS01	CG03	CS15
lila	CP05	CS12	CG02	CS01	CP03	CG03	CS05	CS04	CS09	**CP04**
algebird	CG01	CS03	CS10	**CP01**	**CP04**	CS16	CP02	CS01	CS12	CS05
intellij-scala	CP05	CS05	CG01	CS01	**CP01**	CS08	CS09	CS15	**CP04**	CS14
polynote	CP05	CP03	**CP04**	CS12	CS04	CG02	CG01	CS16	CS03	CS02
slick	CP02	**CP01**	CS10	CS07	CS11	CS16	CS09	CS15	CG03	CS08
sangria	CP05	**CP04**	CP03	CS05	CS10	CS07	CS11	CS03	CS13	CS12
http4s	CS03	CP05	**CP04**	CS16	CS11	CS08	CS10	**CP01**	CG03	CS12
finagle	**CP04**	**CP01**	CS09	CS16	CP02	CS08	CS12	CS05	CS07	CG01
playframework	CP05	CG01	CS01	CG02	CS11	**CP01**	CS12	CG03	CS09	CP02
gatling	**CP01**	CP02	CS05	CS16	CS14	**CP04**	CP05	CS09	CG02	CS10
scalikejdbc	CS12	**CP04**	CS14	CS13	CS01	CG01	CS10	CS08	CS03	CS11
quill	CP05	**CP01**	**CP04**	CG01	CS04	CS14	CS12	CS03	CG02	CS10
json4s	CS12	CS05	CG01	CP03	CS03	**CP01**	**CP04**	CP02	CS16	CS02
circe	CS03	**CP01**	CG03	CG02	CS14	CS16	CS09	CS01	CP03	CS12
scio	CS03	CS14	**CP01**	CS01	CS08	**CP04**	CP02	CG02	CS15	CS07

Note: We labeled features that appear in over 2/3 projects, i.e., 18 times, in bold.

From Table 5, we observe that CP01 and CP04 are highly correlative with the prediction result. As reported in Table 1, CP01 and CP04 belong to group CP, i.e., features related to function properties. This observation shows that properties of higher-order functions, such as the eLoC, have highly affected the prediction result, i.e., the callings of higher-order functions.

To analyze features that affect the prediction result, we calculate the distribution of the top-10 dominant features in the top 10 projects. Fig.5 presents the percentage of each group of features that belong to the top-10 dominant features in each project. Features of group CS are the majority of all 27 projects. One possible reason is that group CS contains 16 features, which are the majority of all extracted features.

We then count the ratio of dominant features in each feature group as follows:

$$ratio(\text{group}) = \frac{\#\text{ dominant features from the group}}{\#\text{ features in the group} \times \#\text{ projects}},$$

where dominant features of each project are directly extracted from Table 5. Fig.6 presents the ratio of dominant features in each feature group. We can observe that the ratio of group CP is the highest, i.e., 0.600, and the ratio of group CS is the lowest, i.e., 0.359. This observation is consistent with the conclusion of Table 5: features in group CP are more influential on the prediction result than features in the other two groups.

We evaluate whether the dominant features can represent all features in the prediction. Table 6 presents the comparison between the top-10 dominant features and all the 24 features in the top 10 projects. As shown in Table 6, three out of 10 projects, i.e., `akka`, `scalatest` and `dotty`, can obtain better prediction with the dominant features than with all features. In project `dotty`, all the evaluation metrics of prediction with the top-10 dominant features are better than the prediction with 24 features except for recall for called higher-order functions. In projects `akka` and `scalatest`, prediction with top-10 dominant features achieves a higher value in precision for called higher-order functions, recall for uncalled higher-order functions, and F-measure for uncalled higher-order functions. In the other seven projects under evaluation, the accuracy with top-10 dominant features is lower than the accuracy with all 24 features.

From Table 6, we find that dominant features cannot fully represent all the 24 features. To this end, we study whether the feature selection algorithms are suitable for PHOF. Feature selection, also called feature subset selection, aims to improve prediction results by removing redundant and irrelevant features[29]. In this study, we use two feature selection algorithms, i.e., information gain and correlation-based feature subset selection (called CfsSubset for short). In information gain, the feature selection is calculated based on how much information the feature can bring to the classification system[30]. The more information it brings, the more important the feature is. The CfsSubset algorithm is a feature selection that evaluates each feature

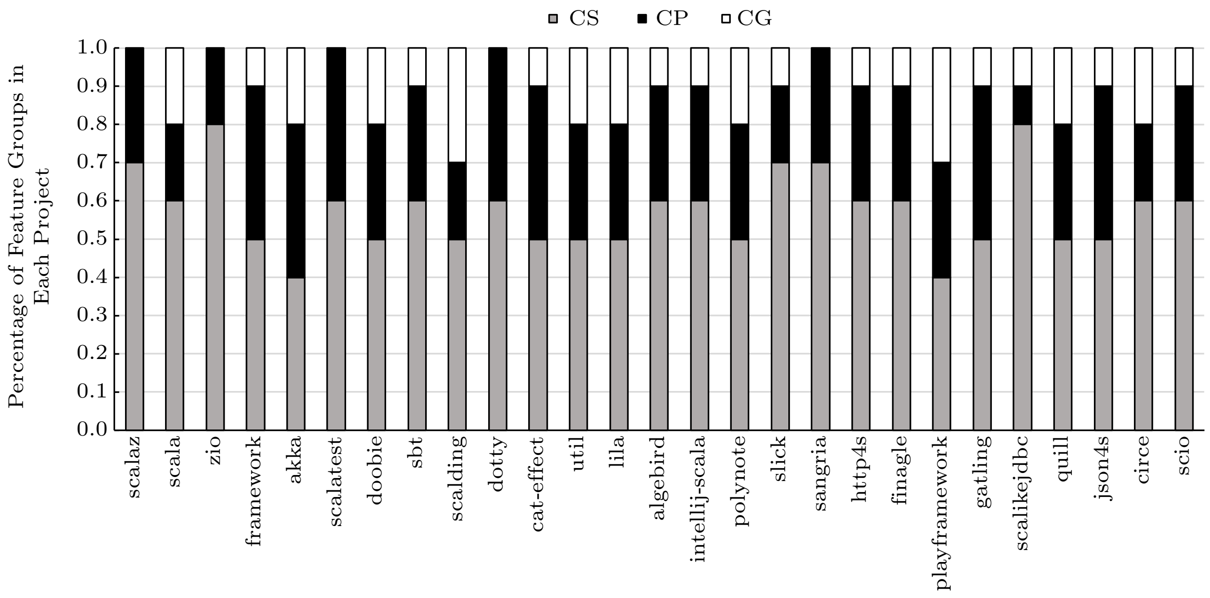


Fig.5. Percentage of three feature groups in each project.

based on the predictive ability in the attribute subset and the correlation [31].

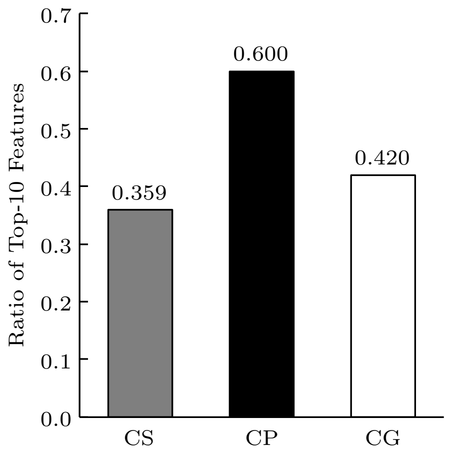


Fig.6. Ratio of dominant features in each feature group in each project.

Fig.7 presents the evaluation values of precision, recall, F-measure, and accuracy of random forest with feature selection in 27 projects. In Fig.7, results with the information gain algorithm and results with the CfsSubset algorithm are worse than those with no feature selection algorithm (Default). From Fig.7, we can find that using feature selection cannot lead to significant differences, compared with the prediction with no feature selection.

Finding 3. Two features, CP01 (eLoC of higher-order functions) and CP04 (containing input functions or output functions in the higher-order function), highly correlate with the prediction result. The top-10 dominant features can partially represent all 24 features in the prediction. Meanwhile, the results obtained using feature selection algorithms are not better than those without feature selection.

5.4 RQ4. How Efficient Is Our Approach to Prediction?

We calculate the time cost of PHOF in each project with the random forest algorithm and the SMOTE strategy. The reason is that the random forest algorithm and the SMOTE strategy achieve the highest accuracy in the prediction in Subsection 5.1 and Subsection 5.2. We show the time cost of feature extraction, model training, and model deployment for each project: the cost of feature extraction is the time of extracting features from source code and logs; the cost of model training is the time of training the random forest algorithm in the prediction; and the cost of model deployment is the time of deploying the trained model to the test set. Let $T_{\text{extraction}}$, T_{training}, and $T_{\text{deployment}}$ be the time cost of feature extraction, model training, and model deployment, respectively. The total time cost of PHOF is called T_{total}.

Table 6. Comparison Between the Whole Set of 24 Features and the Top-10 Features on Higher-Order Functions from the Top 10 Projects

Project	Feature	Called			Uncalled			Accuracy
		Precision	Recall	F-Measure	Precision	Recall	F-Measure	
scalaz	All 24	**0.760**	**0.787**	**0.773**	**0.677**	**0.642**	**0.659**	**0.727**
	Top-10	0.735	0.784	0.759	0.657	0.595	0.624	0.706
scala	All 24	0.833	**0.923**	**0.876**	**0.600**	0.383	0.468	**0.799**
	Top-10	**0.836**	0.910	0.872	0.574	**0.406**	**0.476**	0.794
zio	All 24	**0.674**	**0.604**	**0.637**	**0.847**	**0.883**	**0.865**	**0.803**
	Top-10	0.613	0.479	0.538	0.808	0.879	0.842	0.764
framework	All 24	**0.656**	**0.628**	**0.642**	**0.749**	**0.772**	**0.760**	**0.713**
	Top-10	0.620	0.584	0.602	0.722	0.751	0.736	0.683
akka	All 24	0.815	**0.846**	**0.830**	**0.777**	0.737	0.757	0.800
	Top-10	**0.832**	0.827	**0.830**	0.765	**0.771**	**0.768**	**0.804**
scalatest	All 24	0.844	**0.806**	0.824	**0.867**	0.895	0.881	0.858
	Top-10	**0.883**	0.791	**0.835**	0.863	**0.926**	**0.893**	**0.870**
doobie	All 24	**1.000**	**0.930**	**0.964**	**0.896**	**1.000**	**0.945**	**0.956**
	Top-10	0.989	0.900	0.942	0.855	0.983	0.915	0.931
sbt	All 24	**0.755**	**0.762**	**0.758**	**0.510**	**0.500**	**0.505**	**0.675**
	Top-10	0.736	0.743	0.739	0.471	0.462	0.466	0.650
scalding	All 24	**0.683**	**0.755**	**0.717**	**0.549**	**0.459**	**0.500**	**0.639**
	Top-10	0.627	0.734	0.676	0.444	0.328	0.377	0.574
dotty	All 24	0.762	**0.877**	0.816	0.133	0.065	0.087	0.693
	Top-10	**0.786**	0.868	**0.825**	**0.300**	**0.194**	**0.235**	**0.715**

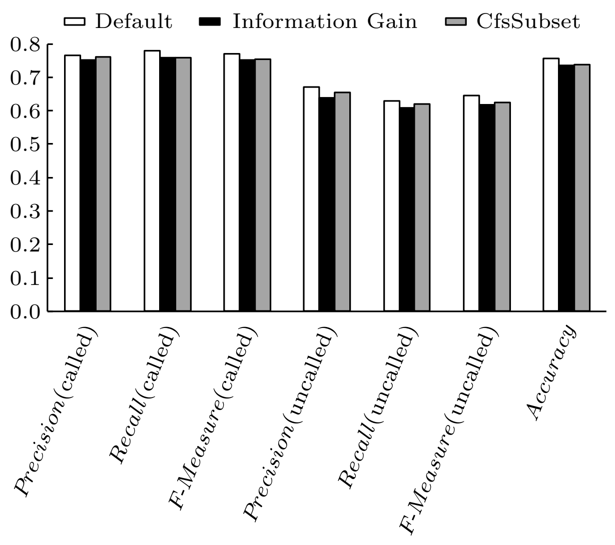


Fig.7. Average results of random forest algorithm with feature selection in 27 projects.

Table 7 presents the time cost of PHOF in seconds with the random forest algorithm and the SMOTE strategy. As shown in Table 7, the average time of PHOF in the top 10 projects is 472.795 seconds, which consists of 472.708 seconds for feature extraction, 0.084 seconds for model training, and 0.003 seconds for model deployment. This result suggests that our proposed method is efficient in identifying calling-prone higher-order functions.

Table 7. Time Cost of Feature Extraction, Model Training, and Model Deployment of PHOF in the Top 10 Projects

Project	$T_{\text{extraction}}$	T_{training}	$T_{\text{deployment}}$	T_{total}
scalaz	1 380.120	0.260	0.010	1 380.390
scala	916.590	0.170	0.010	916.770
zio	495.361	0.090	0.002	495.453
framework	223.944	0.070	0.001	224.015
akka	401.201	0.070	0.001	401.272
scalatest	175.465	0.030	0.001	175.496
doobie	158.625	0.040	0.001	158.666
sbt	445.430	0.040	0.001	445.471
scalding	139.642	0.030	0.001	139.673
dotty	390.702	0.040	0.001	390.743
Average	472.708	0.084	0.003	472.795

Finding 4. Our proposed approach is efficient: the average time cost of PHOF in the top 10 projects is 473 seconds.

6 Threats to Validity

We discuss the threats to the validity to our work in three dimensions.

Threats to Construct Validity. In our study, we extracted 24 features from source code and logs from three groups, including code statements, function properties, and logs. However, it is possible to design better features to characterize the prediction problem via detailed manual selection. Meanwhile, we chose six typical machine learning algorithms, three typical strategies of imbalanced data processing, and two typical feature selection algorithms according to our experience. There may exist several algorithms or processing strategies that can achieve better results. The design of this work is to assist testers in scheduling higher-order functions in manual testing. The evaluation in the paper does not count the manual effort by testers. Instead, we consider that higher-order functions that will be called in future should be tested first. This provides the scenario of using our approach. In addition, the motivation of this work is to prioritize higher-order functions in a limited resource. It is possible that uncalled functions contain bugs. We did not explore hidden bugs in uncalled functions since potentially called higher-order functions should expose bugs first.

Threats to Internal Validity. In machine learning, the setting of parameters is important: the prediction may be hurt by the setting of parameters. In our study, we set parameters according to the API document of the Weka document. These parameter values are not well-tuned for our dataset. A better way of setting parameter values is to conduct a large-scale experiment and tune parameter values accordingly. Moreover, this study did not analyze the calling-proneness for standalone libraries. Detecting the calling chains of these libraries requires huge efforts. We leave this work as a piece of future work.

Threats to External Validity. Our study selected 27 Scala projects from GitHub. Such selection may hurt the generality of our study. We do not claim that our prediction result can be generalized to other Scala projects or even other functional languages, such as Haskell. This results in a threat to the generality.

7 Related Work

We present the related work in two categories, the studies on higher-order functions and the studying on Scala programs.

7.1 Studies on Scala Programs

The Scala programming language has received much attention. Existing work has studied the Scala language and Scala programs. Reynders *et al.*[32] defined a multilevel language, Scalagna, which combines the existing Scala JVM and the JavaScript ecosystem into a single programming model. Nystrom[11] designed a Scala framework to implement efficient super-compilers for arbitrary programming languages. In the field of symbolic execution, Cassez and Sloane[33] proposed ScalaSMT, which supports the Satisfiability Modulo Theory (SMT) solving in Scala. In the field of education, van der Lippe *et al.*[3] used the Scala programming language and the WebLab online learning system to examine quizzes by students. Kroll *et al.*[4] proposed a framework that supports straightforward and simplified the translation between formal specifications and executable code.

7.2 Studies on Higher-Order Functions

The higher-order function is a feature of the Scala language. Higher-order functions are employed as a resolution for complicated problems. Xu *et al.*[34] conducted an empirical study on the use of higher-order functions and analyzed the Scala code in 35 open-source projects. Bassoy and Schatz[5] used optimized higher-order functions to quickly calculate tensors, and their optimized higher-order functions achieved 68% of the maximum throughput of the Intel Core i9-7900X. Nakaguchi *et al.*[6] treated services as functions and used higher-order functions to combine these services without creating new services.

Testing and validating higher-order functions is difficult due to the complexity of higher-order functions. To test a higher-order function, a tester has to understand the requirements and then writes an input function for the higher-order function under test. Koopman and Plasmeijer[35] proposed a method to test higher-order functions by mimicking and controlling the structure of functions. This method can find errors that have not occurred in higher-order functions for several years. Selakovic *et al.*[7] proposed LambdaTester, which uses feedback techniques to automatically generate test cases for higher-order functions in JavaScript. For the validation of higher-order functions, Madhavan *et al.*[36] proposed a novel method to specify and verify the resource utilization of higher-order functional programs using lazy evaluation and memory. Voirol *et al.*[37] proposed a validator for pure higher-order functional Scala programs; this validator supports the validation of arbitrary function types and arbitrary nested anonymous functions. Rusu and Arusoaie[38] embedded a higher order functional language with imperative features into the Maude framework to verify higher-order functional programs. Lincke and Schupp[39] proposed a transformation that converts higher-order functions to lower-order functions by mapping higher-order types to lower-order types.

Different from existing work, we design an automated approach to the prediction of future callings of higher-order functions. Instead of directly testing higher-order functions, we identify higher-order functions that need to be tested to assist the manual testing by testers.

8 Conclusions

Manually testing a higher-order function is difficult. In this paper, we proposed an approach to the identification of calling-prone higher-order functions in Scala programs, namely PHOF. This approach can predict whether a higher-order function will be called in the future and help testers to identify higher-order functions that need to be tested first. Our work can save the time cost of testing higher-order functions in a limited time budget. Our study showed that the random forest algorithm with the SMOTE strategy in PHOF is effective in the prediction.

In future work, we plan to extract new features such as semantic features to enhance the prediction performance. A study on understanding dominant features could help design the feature extraction. We also plan to investigate the calling-proneness of code libraries since the libraries are originally designed for callings. We are about to conduct a study to understand reasons for uncalled higher-order functions, including the reason from source code and the reason from developers or testers. Cross-project evaluation is another future work. We aim to examine the identification of calling-proneness higher-order functions beyond a single project.

References

[1] Mei H, Huang G, Xie T. Internetware: A software paradigm for Internet computing. *IEEE Computer*, 2012, 45(6): 26-31.

[2] Wang B, Zhao H, Zhang W, Jin Z, Mei H. A problem-driven collaborative approach to eliciting requirements of Internetwares. In *Proc. the 2nd Asia-Pacific Symposium on Internetware*, November 2010, Article No. 22.

[3] van der Lippe T, Smith T, Pelsmaeker D, Visser E. A scalable infrastructure for teaching concepts of programming languages in Scala with WebLab: An experience report. In *Proc. the 7th ACM SIGPLAN Symposium on Scala*, October 2016, pp.65-74.

[4] Kroll L, Carbone P, Haridi S. Kompics Scala: Narrowing the gap between algorithmic specification and executable code (short paper). In *Proc. the 8th ACM SIGPLAN International Symposium on Scala*, October 2017, pp.73-77.

[5] Bassoy C, Schatz V. Fast higher-order functions for tensor calculus with tensors and subtensors. In *Proc. the 18th International Conference on Computer Science*, June 2018, pp.639-652.

[6] Nakaguchi T, Murakami Y, Lin D, Ishida T. Higher-order functions for modeling hierarchical service bindings. In *Proc. the 2016 IEEE International Conference on Services Computing*, July 2016, pp.798-803.

[7] Selakovic M, Pradel M, Karim R, Tip F. Test generation for higher-order functions in dynamic languages. *Proceedings of the ACM on Programming Languages*, 2018, 2(OOPSLA): Article No. 161.

[8] Ma P, Cheng H, Zhang J, Xuan J. Can this fault be detected: A study on fault detection via automated test generation. *Journal of Systems and Software*, 2020, 170: Article No. 110769.

[9] Xu Y, Jia X, Xuan J. Writing tests for this higher-order function first: Automatically identifying future callings to assist testers. In *Proc. the 11th Asia-Pacific Symposium on Internetware*, October 2019, Article No. 6.

[10] Odersky M, Altherr P, Cremet V, Emir B, Maneth S, Micheloud S, Mihaylov N, Schinz M, Stenman E, Zenger M. An overview of the Scala programming language. Technical Report, École Polytechnique Fédérale de Lausanne, 2006. https://www.scala-lang.org/docu/files/ScalaOverview.pdf, September 2020.

[11] Nystrom N. A Scala framework for supercompilation. In *Proc. the 8th ACM SIGPLAN International Symposium on Scala*, October 2017, pp.18-28.

[12] McCabe T J. A complexity measure. *IEEE Trans. Software Eng.*, 1976, 2(4): 308-320.

[13] Wang T, Zhang Y, Yin G, Yu Y, Wang H. Who will become a long-term contributor? A prediction model based on the early phase behaviors. In *Proc. the 10th Asia-Pacific Symposium on Internetware*, September 2018, Article No. 9.

[14] Quinlan J R. C4.5: Programs for Machine Learning (1st edition). Morgan Kaufmann, 1993.

[15] Breiman L. Random forests. *Machine Learning*, 2001, 45(1): 5-32.

[16] Cortes C, Vapnik V. Support-vector networks. *Machine Learning*, 1995, 20(3): 273-297.

[17] Hastie T, Tibshirani R, Friedman J H. The Elements of Statistical Learning: Data Mining, Inference, and Prediction (2nd edition). Springer, 2009.

[18] Pearl J. Bayesian networks: A model of self-activated memory for evidential reasoning. In *Proc. the 7th Conference of Cognitive Science Society*, August 1985, pp.15-17.

[19] Tolles J, Meurer W J. Logistic regression: Relating patient characteristics to outcomes. *The Journal of the American Medical Association*, 2016, 316(5): 533-534.

[20] He H, Garcia E A. Learning from imbalanced data. *IEEE Trans. Knowl. Data Eng.*, 2009, 21(9): 1263-1284.

[21] Chawla N V, Bowyer K W, Hall L O, Kegelmeyer W P. SMOTE: Synthetic minority over-sampling technique. *J. Artif. Intell. Res.*, 2002, 16: 321-357.

[22] Sajnani H, Saini V, Svajlenko J, Roy C K, Lopes C V. SourcererCC: Scaling code clone detection to big-code. In *Proc. the 38th International Conference on Software Engineering*, May 2016, pp.1157-1168.

[23] Rahman W, Xu Y, Pu F, Xuan J, Jia X, Basios M, Kanthan L, Li L, Wu F, Xu B. Clone detection on large Scala codebases. In *Proc. the 14th IEEE International Workshop on Software Clones*, February 2020, pp.38-44.

[24] Zhang X, Chen Y, Gu Y, Zou W, Xie X, Jia X, Xuan J. How do multiple pull requests change the same code: A study of competing pull requests in GitHub. In *Proc. the 2018 IEEE International Conference on Software Maintenance and Evolution*, September 2018, pp.228-239.

[25] Hall M A, Frank E, Holmes G, Pfahringer B, Reutemann P, Witten I H. The WEKA data mining software: An update. *SIGKDD Explorations*, 2009, 11(1): 10-18.

[26] He H, Bai Y, Garcia E A, Li S. ADASYN: Adaptive synthetic sampling approach for imbalanced learning. In *Proc. the International Joint Conference on Neural Networks*, June 2008, pp.1322-1328.

[27] Karlsson O, Haller P. Extending Scala with records: Design, implementation, and evaluation. In *Proc. the 9th ACM SIGPLAN International Symposium on Scala*, September 2018, pp.72-82.

[28] Egghe L, Leydesdorff L. The relation between Pearson's correlation coefficient r and Salton's cosine measure. *J. Assoc. Inf. Sci. Technol.*, 2009, 60(5): 1027-1036.

[29] Han J, Kamber M, Pei J. Data Mining: Concepts and Techniques (3rd edition). Morgan Kaufmann, 2011.

[30] Wang G, Lochovsky F H. Feature selection with conditional mutual information maximin in text categorization. In *Proc. the 2004 ACM CIKM International Conference on Information and Knowledge Management*, November 2004, pp.342-349.

[31] Hall M A. Correlation-based feature subset selection for machine learning [Ph.D. Thesis]. University of Waikato, 1998.

[32] Reynders B, Greefs M, Devriese D, Piessens F. Scalagna 0.1: Towards multi-tier programming with Scala and Scala.js. In *Proc. the Conference Companion of the 2nd International Conference on Art*, April 2018, pp.69-74.

[33] Cassez F, Sloane A M. ScalaSMT: Satisfiability modulo theory in Scala (tool paper). In *Proc. the 8th ACM SIGPLAN International Symposium on Scala*, October 2017, pp.51-55.

[34] Xu Y, Wu F, Jia X, Li L, Xuan J. Mining the use of higher-order functions: An exploratory study on Scala programs. *Empirical Software Engineering*, 2020, 25(6): 4547-4584.

[35] Koopman P W M, Plasmeijer R. Automatic testing of higher order functions. In *Proc. the 4th Asian Symposium on Programming Languages and Systems*, November 2006, pp.148-164.

[36] Madhavan R, Kulal S, Kuncak V. Contract-based resource verification for higher-order functions with memoization. In *Proc. the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, January 2017, pp.330-343.

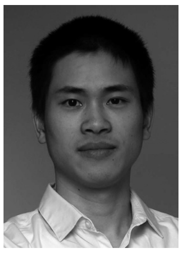
[37] Voirol N, Kneuss E, Kuncak V. Counter-example complete verification for higher-order functions. In *Proc. the 6th ACM SIGPLAN Symposium on Scala*, June 2015, pp.18-29.

[38] Rusu V, Arusoaie A. Executing and verifying higher-order functional-imperative programs in Maude. *Journal of Logic and Algebraic Methods in Programming*, 2017, 93: 68-91.

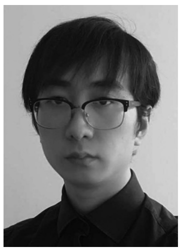
[39] Lincke D, Schupp S. From HOT to COOL: Transforming higher-order typed languages to concept-constrained object-oriented languages. In *Proc. the International Workshop on Language Descriptions, Tools, and Applications*, April 2012, Article No. 3.

Yi-Sen Xu is a Master student at the School of Computer Science, Wuhan University, Wuhan. He received his B.S. degree in software engineering at School of Computer Science, Wuhan University, in 2019. His research interests lie within software testing and mining software repositories.

Xiang-Yang Jia received his Ph.D. degree in computer software and theory from Wuhan University, Wuhan, in 2008. From 2014 to 2015, he was a visiting researcher with the Dependable Evolvable Pervasive Software Engineering Group, Politecnico di Milano, Milan. He is currently a lecturer with the School of Computer Science, Wuhan University, Wuhan. His current research interests include symbolic execution, software analysis, search-based software engineering, and mining software repositories.



Fan Wu is a co-founder of Turing Intelligence Technology Limited, London. He holds his Ph.D. degree in software engineering from University College London, London. He is renowned for his work on deep optimization on software systems, a research field he co-founded. He serves as a reviewer and program committee member for prestigious research conferences and journals, such as JSS, IST, and GECCO. His research interests include search-based software engineering, genetic improvement, evolutionary computation, and machine learning.



Lingbo Li is a co-founder of Turing Intelligence Technology Limited, London. He received his Ph.D. degree in software engineering from University College London, London, under the supervision of Prof. Mark Harman (Facebook). He was subsequently invited to take associate professorship at the School of Computer Science, Wuhan University, Wuhan. Academically, he serves on the program committee and as a reviewer for various prestigious research conferences and journals, such as, JSS IST, IEEE Intelligent Systems, and GECCO. His research interests include search-based software engineering, requirement engineering, evolutionary computation, and deep learning.

Ji-Feng Xuan is a professor at the School of Computer Science, Wuhan University, Wuhan. He received his B.S. degree in software engineering and his Ph.D. degree in computational mathematics from Dalian University of Technology, Dalian. He was previously a postdoctoral researcher at the INRIA Lille-Nord Europe, Lille. He is a reviewer of journals and conferences, including TSE, TOSEM, TKDE, and TEVC. His research interests include software testing and debugging, software data analysis, and search-based software engineering.