

Heterogeneous tree structure classification to label Java programmers according to their expertise level

Francisco Ortin^{a,*}, Oscar Rodriguez-Prieto^a, Nicolas Pascual^a, Miguel Garcia^a

^a*University of Oviedo, Computer Science Department,
c/Federico Garcia Lorca 18, 33007, Oviedo, Spain*

Abstract

Open-source code repositories are a valuable asset to creating different kinds of tools and services, utilizing machine learning and probabilistic reasoning. Syntactic models process Abstract Syntax Trees (AST) of source code to build systems capable of predicting different software properties. The main difficulty of building such models comes from the heterogeneous and compound structures of ASTs, and that traditional machine learning algorithms require instances to be represented as n -dimensional vectors rather than trees. In this article, we propose a new approach to classify ASTs using traditional supervised-learning algorithms, where a feature learning process selects the most representative syntax patterns for the child subtrees of different syntax constructs. Those syntax patterns are used to enrich the context information of each AST, allowing the classification of compound heterogeneous tree structures. The proposed approach is applied to the problem of labeling the expertise level of Java programmers. The system is able to label expert and novice programs with an average accuracy of 99.6%. Moreover, other code fragments such as types, fields, methods, statements and expressions could also be classified, with average accuracies of 99.5%, 91.4%, 95.2%, 88.3% and 78.1%, respectively.

Keywords: Big code, machine learning, syntax patterns, abstract syntax trees, programmer expertise, decision trees, big data

1. Introduction

Big data is aimed at extracting value from large datasets, creating predictive models and reports, visualizing and describing data, and finding relationships between variables. Big data is being used in many different fields such as medicine, finance,

*Corresponding author

Email addresses: ortin@uniovi.es (Francisco Ortin), rodriguezoscar@uniovi.es (Oscar Rodriguez-Prieto), uo245366@uniovi.es (Nicolas Pascual), garciarmiguel@uniovi.es (Miguel Garcia)

URL: <http://www.reflection.uniovi.es/ortin> (Francisco Ortin)

healthcare, education, social networks and genomics. Considering programs as data, the existing open-source code repositories (GitHub, SourceForge, BitBucket and CodePlex) provide massive codebases to be used in the creation of programming tools and services to improve software development, making use of machine learning and probabilistic reasoning [1, 2]. This research area has been termed “big code”, due to its similarity with big data and the use of source code [3].

In the big code area, existing source-code corpora have already been used to create different systems such as deobfuscators [1], statistical machine translation [4], security vulnerability detection [5] and decompilation [6, 7]. Probabilistic models are built with machine learning and natural language processing techniques to exploit the abundance of patterns in source code. Three categories of models have been identified, based on the way they represent the structure of programs [8]: token-level models, that represent code as a sequence of tokens (terminal symbols in the language); syntactic models, that represent code as a trees (abstract syntax trees or ASTs); and semantic models, that use additional graph structures (e.g., control-flow graphs and data-dependency graphs).

One of the challenges of big code is to classify and score the level of programming expertise of developers, by analyzing the source code they write [9]. Then, new tools and IDEs¹ to teach programming can be developed. Such tools would provide different hints to programmers depending on their level of expertise. A novice Java programmer could be instructed to use inheritance and polymorphism; for average developers, functional idioms using lambda expressions could be introduced [10]; and advanced patterns to avoid performance bottlenecks or security vulnerabilities could be advised to expert programmers [11].

A system capable of classifying programmers by their expertise level can also be used to analyze the recurrent idioms written by expert programmers. Such idioms could be published and used to improve the skills of average programmers. Likewise, programming lecturers can identify the recurrent programming patterns used by beginners, explaining how they could be improved with better alternatives.

A model that scores the expertise level of programmers can be used to check the improvement of student’s programming skills during a programming course. The model would identify those students that do not obtain the expected level of programming expertise, so lecturers could help them at the earliest. It would also identify those who have better programming skills, so they could be motivated with additional activities.

The scoring model could also be used by an Intelligent Tutoring System (ITS) that considers how the student evolves. If the student score increases, more advanced programming constructs will be taught. If the score stays the same, the ITS will offer new activities to strengthen the new language construct taught. Finally, if the score drops, the system will revisit some language constructs formerly explained. In this case, the language construct to be revisited would depend on the idioms coded by the student.

¹The classifier could also be included in existing IDEs such as Eclipse, IntelliJ and NetBeans.

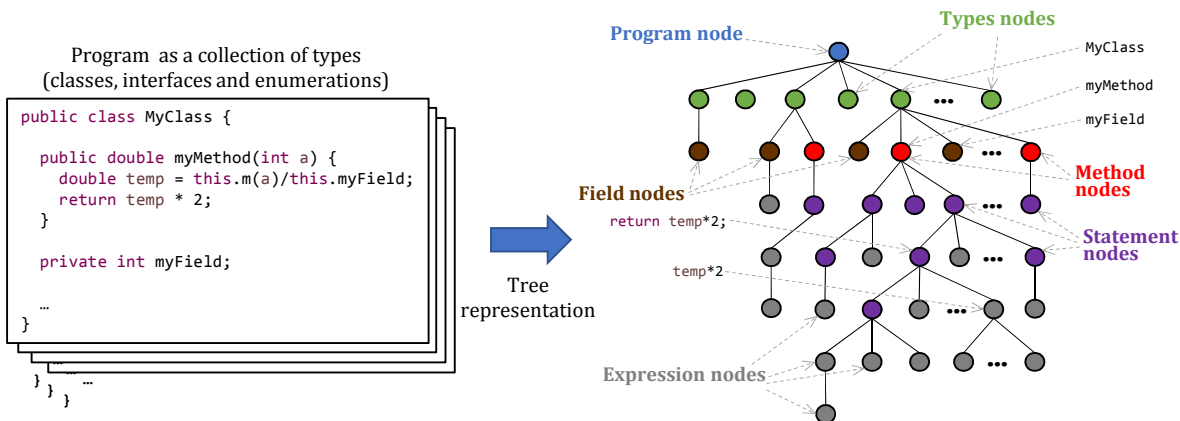


Figure 1: Program representation as heterogeneous compound syntax trees.

1.1. Requirements

In this work, we face the challenge of building syntactic models to classify and score the programming level of expertise of Java developers. What follows are the main requirements we must fulfill.

Different levels of syntax constructs

When classifying programmers, different fragments of their code could be analyzed. Therefore, a classifier must consider different levels of syntax constructs, such as expressions, statements, methods, fields, types (classes, interfaces and enumerations) and whole programs (Figure 1). A whole program will give the classifier more information to label the developer, but a useful tool should give hints to the programmer when one single statement, method or even expression is typed. Therefore, a programmer classifier should be constructed with different models that classify different levels of syntax constructs (expressions, statements, methods, fields, types and whole programs).

Heterogeneous compound structures

Figure 1 shows that the syntax of the different language constructs are heterogeneous. For example, the syntax of methods is different to the syntax of statements and expressions. Moreover, many program constructs are composed of other program constructs. For example, the assignment statement “`temp = this.m(a)/this.myField;`” comprises the two expressions on the left- and right-hand side of the assignment operator (the right-hand side is also subdivided in other subexpressions). Likewise, object-oriented programs are composed of a set of types, types may comprise methods and fields, methods contain statements, and statements and fields are commonly built using expressions. Therefore, a Java syntax classifier should be able to label those heterogeneous compound AST structures.

Interpretable white-box models

As mentioned, the syntax patterns used to classify expert and novice programmer are valuable information. We described how they could be used to assist lecturers in

a programming course, and to create Intelligent Tutoring Systems. Additionally, we propose to use the extracted patterns in a feature learning process (Section 3) to build classifiers with different kinds of syntax constructs.

Scalability

Model construction must be scalable, since we follow the big code philosophy of using massive datasets. It must allow the construction of classifiers from millions of instances. For example, just the dataset we used to build the expressions classifier holds 13,498,005 instances (see Section 4.1).

Models from trees

An important challenge of syntax pattern classification is to build predictive models from trees, since most supervised classification algorithms require instances (individuals or rows) to be represented as fixed size n -dimensional vectors [12]. While there are standard techniques to compute such vectors for documents, images and sound, there are no similarly standard representations for programs [5]. There exist alternative structured prediction methods such as Graph Neural Networks (GNNs) and Conditional Random Fields (CRFs) —discussed in Section 2—, but they unfortunately seem to suffer sufficiently high computation and space costs to be used with massive codebases [13].

1.2. Contribution

In this work, we use decision trees (DTs) as the supervised learning algorithm, because DTs create interpretable white-box models, and perform well with large datasets [14]. They are also able to handle both numerical and categorical data.

In order to build DTs, we tabularize the ASTs of the input programs. We represent as features the main syntactic characteristics of each kind of node (expression, statement, method, field, type and program), including its category (e.g., arithmetic operation, method invocation, field access, etc.), and multiple information about their context (data about its parent and child nodes, its role in the enclosing node, its depth and height, etc.).

We create different datasets for each kind of node. Then, we build different homogeneous DT models that classify each kind of syntax construct (e.g., expressions, statements, methods, etc.). Finally, we take the patterns used by the homogeneous models to build new classifiers of compound heterogeneous syntax constructs (e.g., a method classifier that also considers the syntax patterns of the statements and expressions written within the method).

The main contributions of this paper are:

1. A new feature learning approach to classify great amounts of trees made up of compound heterogeneous structures.
2. A system to classify the programming expertise level of Java developers by analyzing the syntax constructs of their code. The system can also be used to measure the probability of a code fragment to be written by an expert or beginner.

3. The identification of Java syntax patterns used by both expert and novice programmers.

The rest of this paper is structured as follows. Next section discusses the related work, and Section 3 details the proposed system. Section 4 evaluates our system with different experiments, and the conclusions and future work are presented in Section 5.

2. Related Work

We discuss the work related to source code classification with syntactic models, according to its objective and the method used.

2.1. Classification of programmers by their expertise level

In the field of programmer classification, Lee *et al.* used biometric sensors data to detect the programmer’s level of expertise [15]. In particular, they used a 16-channel-amplifier V-amp to collect electroencephalographic data, and a SMI RED-mx eye-tracker to classify eye movement according to the velocity of shifts in the programmer eye direction. They conducted a study with 38 expert and novice programmers, investigating how well electroencephalography and an eye-tracker data can be utilized to classify novice and expert programmers in two kinds of programming tasks (easy and difficult). By using Support Vector Machines, they built three models: one using the eye-tracker, another one with the electroencephalographic sensors, and the third one with both data sources. The F1 performances of the three models to classify expert programmers were, respectively, 90.3%, 93.1% and 97%.

Samy S. Abu-Naser built an artificial neural network to classify the academic performance of students in linear programming [9]. Naser used the Linear Programming Intelligent Tutoring System (LP-ITS), created in the Al-Azhar University to teach linear programming. LP-ITS automatically generates programming problems for the students to solve [16]. The system stores information about the student interaction with LP-ITS, that is used to train the model. They used the log files of 67 learners, with 2144 program submissions; half for training and half for testing. A multilayer perceptron neural network (9 nodes in the input layer, 5 for the hidden one, and one node in the output layer) with sigmoid activation function was used to classify student’s level of expertise. The average accuracy obtained was 92%.

2.2. Clone detection

Machine learning has been used to detect syntax patterns in source code for different purposes. One of the most active fields is clone detection, which is aimed at finding similarities between source-code fragments. Clone detection is used to identify repeated (not reused) code for software maintenance, program understanding, code refactoring, plagiarism detection and program compaction [17]. There exist different approaches to detect clones: text-based techniques, lexical, syntactic and semantic approaches, and hybrid methods [18].

The syntactic approaches to detect clones use parsers to convert source code into parse trees or ASTs, which are then processed using tree-matching or structural metrics [18]. Tree-matching techniques compare tree structures; whereas metrics-based systems gather metrics for code fragments and compare the metrics vectors, rather than the AST, to perform the classification [19].

CloneDr is a tree-matching clone detector that compares trees with the same structure [20]. To that aim, they propose three algorithms: one to detect sub-tree clones; another one for variable-size sequences of sub-trees; and the third algorithm to generalize combinations of other clones [21]. A combination of those three algorithms is used to compare tree structures. The tool was applied to a production software of more than 400K lines of code written in C, detecting average levels of source code duplication of 28%.

Wahler *et al.* find exact and parameterized code fragments, defining a method based on the concept of frequent itemsets, which works with an XML representation of ASTs [22]. In data mining, frequent itemsets are used to illustrate relationships within large amounts of data. For ASTs, frequent itemsets represent sequences of consecutive statements constituting source code clones. The implementation uses an XML database plus a link structure and a hash table to speed up the data access. It took 60 minutes to detect clones in the JDK source, and 90 minutes for the DisLog Developer’s Kit (DDK) of SWI-Prolog [22].

Evans *et al.* built Asta, a clone detection system that works with so-called *structural abstractions* (i.e., AST subtrees) [17]. Programs are parsed, and their ASTs are stored as XML documents. Subtrees are represented as patterns that are matched with other subtrees to detect potential clones. Those patterns are sometimes defined with holes, which are wildcards that allow any subtree match. This feature allows Asta to detect clones with variations in arbitrary subtrees. In a variety of 425K lines of code of Java and 16K lines of C#, 20-50% of the clones found by Asta were structural, and thus beyond lexical methods [17]. Its C++ implementation took 10 minutes to analyze all the code.

Koschke *et al.* defines a clone detection system based on abstract trees, but with linear time and space, similar to lexical alternatives [23]. The proposed algorithm parses the input program, creates the AST, serializes the trees, applies suffix tree detection, and decomposes the resulting token sequences into syntactic structures. Suffix tree comparison is linear in space and time with respect to their length. The proposed system, **cscope**, was tested with the Bellon benchmark, providing accuracies similar to syntactic approaches, with runtime performance of lexical ones [23].

The tool DECKARD computes certain characteristic vectors to approximate the structure of ASTs in a Euclidean space [24]. DECKARD identifies relevant and irrelevant nodes in the AST for clone detection. Characteristic vectors are created by counting the number occurrences of relevant nodes in a subtree. Then, DECKARD detects similar clones by computing locality sensitive hashing, which clusters similar vectors using the Euclidean distance metric. DECKARD was used to find clones in the open JDK and Linux kernel, performing better than CloneDr in accuracy and scalability [24].

2.3. Other scenarios of syntactic classifiers

Ahmad Taherkhani used decision tree classifiers to recognize sorting algorithms from Java code[25]. Algorithms are first converted into vectors of characteristics, including syntactic features such as the number of expressions, tokens, loops, algorithm length, and roles of variables. Then, a C4.5 decision tree classifier is built. The model is trained with five different types of sorting algorithms: Quicksort, Mergesort, Insertion sort, Selection sort and Bubble sort. Taherkhani collected 209 programs for the five sorting algorithms. The programs were gathered from various textbooks on data structures and algorithms, together with course materials available on the Web. A leave-one-out cross-validation technique was used to test the model. The average classification accuracy of the C4.5 model was 98.1%.

NATE is a tool to localize novice type errors in OCaml programs [12]. They convert the ASTs of source programs into Bags-of-Abstracted-Terms (BOATs), where each subtree is abstracted as a feature vector, comprising the numeric values returned by feature functions applied to the tree. NATE is trained with a large corpus of ill-typed programs and their “fixed” version, creating different models with distinct supervised-learning techniques. The resulting models take an ill-typed program and produces a list of potential blame assignments ranked by likelihood. One important feature of NATE is that it just works with expression nodes in the AST. Thus, it does not support error prediction of any syntax construct different to expressions (e.g., statements, functions or data structures). NATE is able to predict correctly the exact sub-expression that must be changed 72% of the time.

There are some other scenarios where machine learning has been used to predict properties of programs by using syntax (and sometimes semantic) information of their source code. Some examples are vulnerabilities detection in source code [5], source-code decompilation [6, 26], code completion [27], and code idiom mining [28].

2.4. Structured methods

Alternative structured methods such as Graph Neural Networks operate on graph structures to provide node and graph classification [29]. In the context of program classification, Lu *et al.* conducted an experiment comparing Tree-Based Convolutional Neural Networks (TBCNN), Gated Graph Neural Networks (GGNN) and Gated Graph Attention Neural Networks (GGANN) [30]. They took C++ code from students, aimed at solving different programming tasks. They evaluated the performance of the three abovementioned neural networks to classify source code into two predefined categories (five similar programming tasks in each group). They created a graph representation that integrated ASTs with the semantic information of Function Call Graphs (FCG) and Data Flow Graphs (DFG). When only syntactic information (i.e., ASTs) was used in the classification, the average accuracies of TBCNN, GGNN and GGANN were, respectively, 94.1%, 96.4% and 97.1%. When semantic information was added, the performance improvement of the classifiers was not significant [30].

Syntax trees were also used with structured prediction methods such as Conditional Random Fields [1]. CRF is a probabilistic framework for labeling and segmenting

structured data, such as sequences, trees and graphs [31]. CRFs define conditional probability distributions over label sequences, given particular observation sequences. CRFs was used to implement JSNice, a tool that predicts names of identifiers and type annotations of variables in obfuscated JavaScript code [1]. By analyzing the usage of variables in a function body, JSNice is able to label the type of the variable (if it is built-in) and a suitable name. In the evaluation of JSNice, it predicted correct names for 63% of the identifiers, and its type annotation predictions were correct in 81% of the cases [1]. A limitation of CRF is that it suffers high computation and space costs to be used with massive codebases [32].

Relational learning is another technique already identified as mechanism to undertake graph mining [33]. Inductive Logic Programming (ILP) is a relational machine learning technique that can be used to classify graphs. Applied to programming language analysis, Sivaraman *et al.* implemented ALICE, an ILP tool to search source code by specifying syntax constructs [34]. The syntax of the language to be queried is specified as logic facts, representing tree structures. Users define their queries by annotating pieces of code with similar structure to the one to be searched. After executing the query, irrelevant examples could be labeled by the user. Then, ALICE learns the query that includes the program structure, by using ILP. When labeling two positive examples and three negative ones, ALICE successfully identifies similar code locations with 93% precision and 96% recall in 2.7 search iterations [34]. Some other performance experiments have indicated that ILP does not seem to scale well with big data [35].

Tree kernels measure similarity between two trees in terms of their sub-structures [36]. Besides natural language processing, tree kernels have been used for source-code plagiarism detection [37]. WASTK is a tool that builds the ASTs of two programs and gets the similarity between them by computing the tree kernels of both trees. In WASTK, Term Frequency-Inverse Document Frequency (TF-IDF) weights are assigned to each node to avoid the misjudgment caused by common code snippets. In the experiment conducted by Deqiang Fu *et al.*, the average performance of WASTK was 48.32%, significantly higher than the JPlag and Sim software plagiarism tools [37]. Training time of tree kernels is quadratic in the size of the trees [38].

3. System Architecture

Figure 2 shows the architecture of our system, and Algorithm 1 details how it works. We first provide a brief high-level description of the modules in the architecture. Forthcoming subsections detail the behavior of each module.

The input of the system is a database of labeled Java programs (expert or beginner); the output is a collection of heterogeneous decision tree models to classify programmers,

²The pseudocode in Algorithm 1 uses the following functions: *classificationRules*, described in Section 3.2; *select* and *simplify*, depicted in Section 3.3; *potentialChildNodesOf*, which returns the syntax constructs that may occur as subtrees of another given syntax construct; and *columns*, which returns the syntax pattern columns in a given dataset.

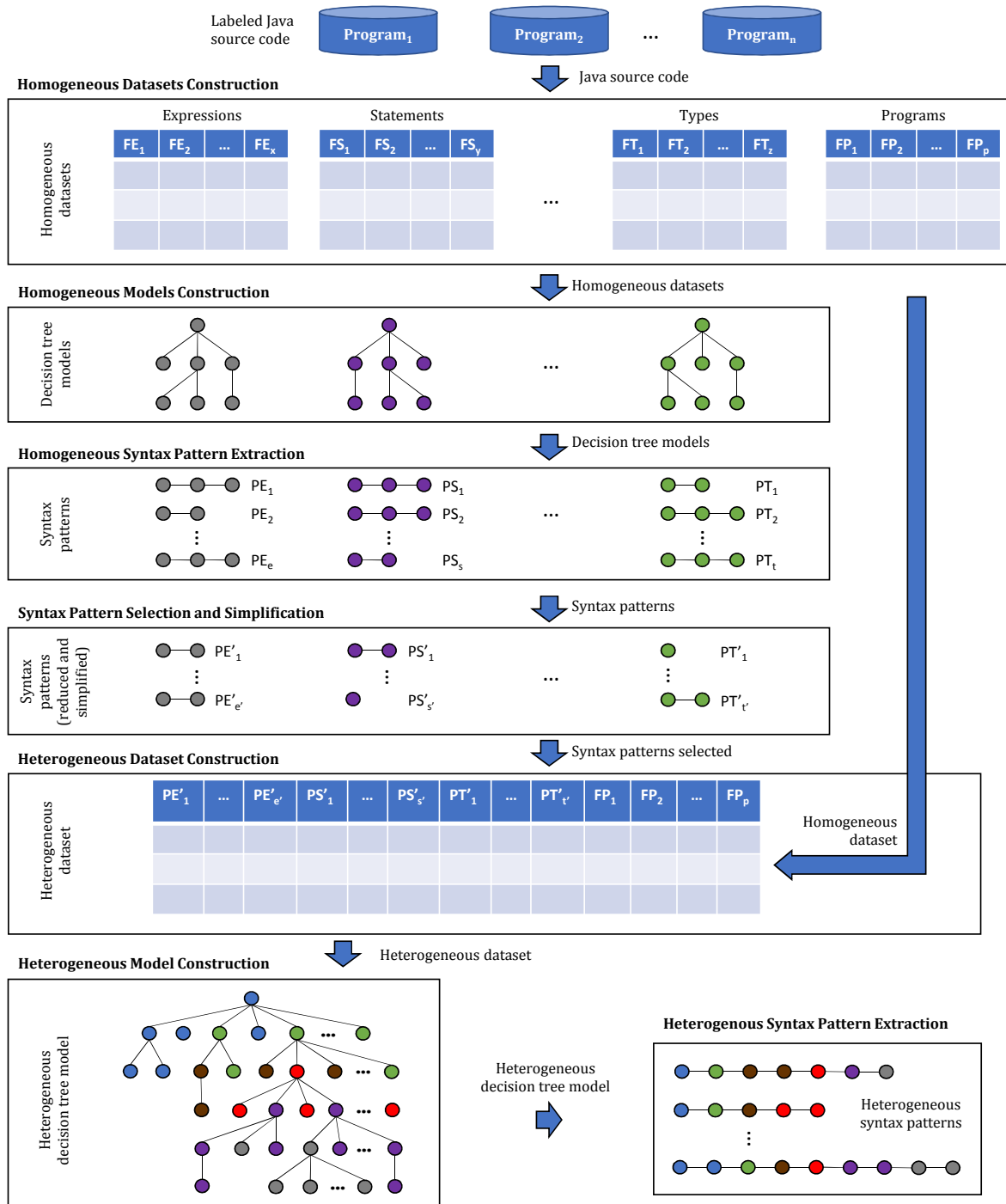


Figure 2: Architecture of the feature learning process.

Input : *sourceCodeDB*: collection of labeled Java programs.
Output: *heteroDT*: decision tree models, *heteroPatterns*: syntax patterns.

```

// Homogeneous model construction
foreach syntaxConstr in {expression, statement, method, field, type, program} do
  // Homogeneous dataset construction
  Define the structure of homoDSsyntaxConstr dataset, including its syntactic category and
  information about its context
  foreach AST in sourceCodeDB do
    foreach node of type syntaxConstr in AST do
      Include the node features and context information as a record (instance) in
      homoDSsyntaxConstr dataset
    end
  end
  // Homogeneous model construction
  Build a homoDTsyntaxConstr DT model using the homoDSsyntaxConstr dataset
  // Homogeneous syntax pattern extraction
  homoPatternssyntaxConstr  $\leftarrow$  classificationRules(homoDTsyntaxConstr)
  // Syntax pattern selection and simplification
  homoPatternssyntaxConstr  $\leftarrow$  select(homoPatternssyntaxConstr)
  homoPatternssyntaxConstr  $\leftarrow$  simplify(homoPatternssyntaxConstr)
end
// Heterogeneous model construction
foreach syntaxConstr in {expression, statement, method, field, type, program} do
  // Heterogeneous dataset construction
  Define the structure of heteroDSsyntaxConstr dataset, including all the features
  (columns) of homoDSsyntaxConstr
  foreach childSyntaxConstr in potentialChildNodeOf(syntaxConstr) do
    Add to the structure of heteroDSsyntaxConstr dataset one feature (column) per
    syntax pattern in homoPatternschildSyntaxConstr
  end
  Copy all the values from homoDSsyntaxConstr dataset to heteroDSsyntaxConstr
  foreach instance in heteroDSsyntaxConstr do
    foreach syntaxPattern in columns(heteroDSsyntaxConstr) do
      Update the (instance, syntaxPattern) cell in heteroDSsyntaxConstr with the
      percentage of occurrences of syntaxPattern in the subASTs of the AST
      represented by instance
    end
  end
  // Heterogeneous model construction
  Build a heteroDTsyntaxConstr DT model using the heteroDSsyntaxConstr dataset
  // Heterogeneous syntax pattern extraction
  heteroPatternssyntaxConstr  $\leftarrow$  classificationRules(heteroDTsyntaxConstr)
end

```

Algorithm 1: Pseudocode describing the proposed method².

plus the syntax patterns used by the classifier. Such patterns describe common idioms used by experts and beginners.

First, we create six homogeneous datasets with different features for each syntax construct: expressions, statements, methods, fields, types and programs. For each node in the ASTs, we store its features in a homogeneous dataset defined for that syntax construct. Such features include its syntactic category and context information (see Section 3.1). Then, for each syntax construct, we create a DT model capable of classifying all the different types of AST nodes defined for that syntax construct. For example, the expression DT model classifies any arithmetic, comparison, logical, variable, literals and cast expressions.

The next step is to obtain the syntax patterns from the homogeneous DT models (Section 3.2). DTs are traversed to obtain the decision rules used by the classifiers. The homogeneous syntax patterns are the antecedent parts of such decision rules. Those patterns are then reduced in number, and simplified to make them more readable (Section 3.3).

Once the homogeneous syntax patterns are reduced in number and simplified, we create the heterogeneous models (Section 3.4). As mentioned, AST structures are heterogeneous (e.g., programs, types, methods, fields, etc.) and some of them comprise other ones (e.g., programs contain types, and types contain methods and fields). Therefore, each heterogeneous dataset for one syntax construct (e.g., program) is built with its homogeneous dataset, plus all the syntax patterns of the subASTs it may contain (e.g., methods, fields, etc.). In this way, heterogeneous classifiers utilize not only their homogeneous features, but also the most relevant syntax patterns of their subASTs.

The last step of the process involves extracting the heterogeneous syntax patterns from the heterogeneous DT models (Section 3.5), the same way as we did with the homogeneous ones. The extracted patterns represent common Java idioms used by experts and beginners.

3.1. Homogeneous datasets and models construction

As mentioned, classical supervised learning algorithms work on *feature vectors*: n -dimensional vectors of features that represent each instance (i.e., each individual in a given problem). Our approach is to translate homogeneous syntax constructs (expressions, statements, etc.) into vectors of features. For each syntax construct, we define a *feature abstraction* set of functions f that map each AST node to a numeric value encoding one property [12]. Given the set of feature abstractions f_1, \dots, f_n , we can represent a given AST t as the feature vector $[f_1(t), \dots, f_n(t)]$. In this way, feature abstraction functions represent a parameterizable mechanism to translate ASTs as feature vectors.

The first feature abstraction function we define is the syntactic category of each node in the AST. Figure 3 shows the syntactic category feature abstraction for Java expressions (we use the ANTLR meta-language notation [39]). This feature simply denotes the kind of expression an AST node represents (its syntactic category).

Production	Syntactic category
expression \rightarrow expression ('+' '-' '*' '/' '%') expression	Arithmetic (binary)
expression ('>' '>=' '<' '<=' '==' '!=') expression	Comparison
expression ('&&' ' ') expression	Logic (binary)
expression ('&' ' ' '^') expression	Bitwise (binary)
expression 'instanceof' type	Instance of
expression ('=' '+=' '-=' ...) expression	Assignment
expression ('>>' '<<' '>>>' '<<<') expression	Shift
expression '?' expression ':' expression	Conditional
('++' '--') expression	Inc-Dec prefix
expression ('++' '--')	Inc-Dec postfix
('+' '-') expression	Arithmetic (unary)
'!' expression	Logic (unary)
'~' expression	Bitwise (unary)
'(' type ')' expression	Cast
expression '.' ID	Field access
expression '::' ID	Method reference
expression '[' expression ']'	Indexing
expression '(' (expression (',' expression)*)? '('	Invocation
lambda_parameters '->' lambda_body	Lambda
'new' type '(' (expression (',' expression)*)? '('	New object
'new' type '(' '[' expression ']' '+' ('[' ']*	New array
ID	Identifier
INT_LITERAL CHAR_LITERAL ...	Int literal, char literal ...

Figure 3: Feature abstraction function for the syntactic category of expressions.

Besides its syntactic category, we also store context information of AST nodes. Each AST node occurs in some surrounding context (e.g., parent and child nodes), and we want the classifier to make decisions based on such contexts. For example, object construction using the `new` operator does not discriminate the level of expertise. However, our system detects that beginners rarely use such expression to initialize a `final` field in a class. Thus, the context of a non-discriminating expression may be discriminating.

Table 1 includes the context information stored for expressions (Appendix A shows the features used for the rest of syntax constructs). We use feature abstraction functions to represent the syntactic categories of its three potential child nodes (if no child exists, e.g. unary expressions, the feature is assigned zero), together with its parent. We also store the role that the expression is playing in the parent node. For example, if the parent is the conditional ternary operator, the current node could be playing the role of the condition (first child node), the if-true expression (second child node) or the if-false expression (third child node). As shown in Table 1, we also store the node depth and height in the AST.

To fill the datasets with the syntactic information taken from the labeled source code, we modify the Java compiler in OpenJDK. We analyze the Java programs, and convert the AST structures into the tabular information defined for each syntax construct. Once the homogeneous datasets for the different syntax constructs are built, we

Name	Description
Category	Syntactic category of the current node, detailed in Figure 3.
First, second and third child	Syntactic category of the corresponding child node.
Parent node	Syntactic category of the parent node.
Role	Role played by the current node in the structure of its parent node.
Height	Distance (number of edges) from the current node to the root node in the enclosing type (class, interface or enumeration).
Depth	Maximum distance (number of edges) of the longest path from the current node to a leaf node.

Table 1: Feature abstractions used for expressions.

create the homogeneous DT models.

3.2. Homogeneous syntax pattern extraction

We get the classification rules from the homogeneous decision trees. DTs are traversed with an instance of the *Visitor* design pattern [40], storing the paths from root to leaf nodes as classification rules. The antecedents of the classification rules represent syntax patterns, and the consequent is the outcome of the classification. For example, one decision rule for expressions is:

```

if category(node) == assignment and category(first_child(node)) == field_access
    and category(second_child(node)) == new_object
    and role_in_parent(node) == expression
then expertise_level = expert

```

This rule classifies the example expression “obj1.m(obj2.f = new MyClass())” as code written by an experimented programmer. For this rule, the syntax pattern gathered is an assignment that plays the role of an expression (instead of statement), its left-hand side expression is a field access, and the right-hand side is an object construction.

3.3. Syntax pattern selection and simplification

The heterogeneous datasets include one feature per homogeneous syntax pattern of their potential subASTs. This process would produce a huge number of features, since we expect the number of homogenous patterns to be high. Moreover, the compound nature of ASTs make the number of features to be even higher. For example, the heterogeneous features for programs include the syntax patterns of types, methods, fields, statements and expressions.

Our intention is thus to reduce the number of patterns with the minimal reduction of the accuracy of the classifier, finding a trade-off between these two conflicting variables. To this end, we consider two measures of syntax patterns: coverage and confidence [41]. Coverage is defined as the relative number of instances that satisfy the pattern (how frequently the pattern appears in the dataset):

$$\text{Coverage}(\text{pattern}) = \frac{\text{occurrences of pattern in dataset}}{\text{number of instances}} \quad (1)$$

Confidence is a measure for the whole classification rule, not just the antecedent. The confidence of a rule is an indication of how often a rule has been found to be true:

$$\text{Confidence}(\text{rule}) = \frac{\text{instances fulfilling the rule}}{\text{occurrences of rule antecedent in dataset}} \quad (2)$$

We analyze the influence of these two measures on the accuracy of the classifier. Then, we discard all those rules that do not involve a significant reduction in the classifier performance (experimental results are presented in Section 4.3). We also perform some rule simplifications to make syntax patterns more readable.

3.4. Heterogeneous dataset and model construction

The previous pattern extraction and selection processes undertake automatic feature learning to build the final heterogeneous compound classifiers. The dataset of each syntax construct is made up of the selected syntax patterns of their subASTs (Section 3.3), together with the features of the corresponding homogeneous dataset (Section 3.1) —see Figure 2 and Algorithm 1. The outcome is a collection of different datasets that are used to build the final heterogeneous compound models.

In the heterogeneous datasets, we set the value of each syntax pattern to the percentage of occurrence of such patterns. For each compound instance (e.g., one statement), we count the syntax patterns that its child nodes (its subexpressions) fulfill. Then, the cells for such syntax patterns are filled in with the percentage of occurrence of the pattern in the AST represented by that instance (statement). In this way, the homogeneous features of one syntax construct are enriched with the syntax patterns of its child subASTs, providing better classification performance. In Figure 2, the homogeneous program features $(FP_1, FP_2, \dots, FP_p)$ are enriched with all the heterogeneous syntax patterns that could be found in the program subASTs: types $(PT'_1, \dots, PT'_{t'})$, methods, fields, statements $(PS'_1, \dots, PS'_{s'})$ and expressions $(PE'_1, \dots, PE'_{e'})$.

From the implementation point of view, much computation time is required to check whether subtrees in a program fulfill a specific syntax pattern. To optimize this operation, we convert all the syntax patterns to SQL queries against the homogeneous datasets in the database. The premises in the pattern are translated to SQL “where” clauses. Those queries are executed programmatically, and their results are used to fill in the heterogeneous datasets.

3.5. Heterogeneous syntax pattern extraction

As with the homogeneous datasets, we traverse the resulting heterogeneous DTs to obtain the final heterogeneous syntax patterns. Each classification rule obtained represents a compound syntax pattern for a given programming expertise level. Such rules are expressed not only with features of the syntax construct to be classified, but also with syntax patterns for its child nodes.

4. Evaluation

In this section, we evaluate the performance of the proposed system to label Java programmers according to their expertise level. We first describe the experimental data (Section 4.1) and environment (Section 4.2). Then, we describe and show the results of the following experiments within the framework of the proposed system:

1. Syntax pattern selection (Section 4.3). This experiment applies the method for pattern selection described in Section 3.3 to reduce the number of syntax patterns taken from different DTs.
2. Heterogeneous AST classification (Section 4.4). Evaluates the accuracy of DTs to classify heterogeneous ASTs. DTs are compared with the existing related work, and other common machine-learning approaches.
3. Heterogeneous syntax pattern extraction (Section 4.5). We analyze the final syntax patterns obtained by our system.
4. Scoring the expertise level of programmers (Section 4.6). The heterogeneous dataset is used to score the programming expertise level of Java developers.
5. Execution time of the proposed method (Section 4.7). We measure the execution time required for each module of the architecture described in Section 3.

4.1. Experimental data

To build the datasets, we took Java code from different sources and labeled them as either beginner or expert programmer. For beginners, we gathered code from first year undergraduate students in a Software Engineering degree at the University of Oviedo. We took the code they wrote for the assignments in two year-1 programming courses in academic years 2017/18 and 2018/19. All the code was 100% written by students from scratch. Overall, we collected 35,309 Java files from 3,884 programs.

For expert programmers, we took the source code of different public Java projects in GitHub. We selected active projects with the highest number of contributors: Chromium, LibreOffice, MySQL, OpenJDK and Amazon Web Services. These software products are implemented with 43,775 Java files in 137 programs (AWS comprises 133 different projects).

To get the syntactic information (features in the homogeneous dataset) from the Java code, we modified the implementation of the Java compiler in OpenJDK 12.0.1. After syntax analysis, we traverse the AST several times using the *Visitor* design pattern [40]. In those traversals, we compute all the values for the features of the different

	Beginner	Expert	Total	Final
Expressions	4,616,807	8,881,198	13,498,005	9,233,614
Statements	1,304,585	2,292,791	3,597,376	2,609,170
Methods	237,285	370,618	607,903	474,570
Fields	96,175	135,826	232,001	192,350
Types	35,910	58,719	94,629	71,820
Programs	3,884	137	4,021	274
Total	6,294,646	11,739,289	18,033,935	12,581,798

Table 2: Number of AST nodes.

syntax constructs, filling in the values in the homogeneous datasets. Then, we label each instance with its expertise level.

Table 2 shows the number of AST nodes. To balance data, we removed random instances from the over-representing class to get the same exact number of instances for both classes (final column in Table 2).

4.2. Experimental environment

To implement the experiments, we used Python 3.7.2 and scikit-learn 0.21.1. All the datasets were stored in a PostgreSQL database 11.3. Since PostgreSQL limits the maximum number of columns to 1600, we modified its open source implementation to allow 12,800 columns (features). We run all the code in a Dell PowerEdge R530 server with two Intel Xeon E5-2620 v4 2.1GHz microprocessors (32 cores) with 128GB DDR4 2400MHz RAM memory, running CentOS operating system 7.4-1708 for 64 bits.

DTs were created with the CART algorithm implemented by scikit-learn (**DecisionTreeClassifier**) [42]. CART constructs binary trees using the feature and threshold that yield the largest information gain at each node. This implementation of decision trees permits the use both categorical and numerical data. We selected the best hyper-parameters with exhaustive parallel search across common parameter values (**GridSearchCV**), using stratified randomized 10-fold cross validation (**StratifiedShuffleSplit**). For the hyper-parameter to measure the quality of a split, we tried **gini** and **entropy**; for selecting the strategy to choose the split at each node, we tried **best** and **random**.

4.3. Syntax pattern selection

Our source code database of 35,309 Java files generated the six homogeneous datasets detailed in Table 2. On aggregate, the datasets contain 12.5 million AST nodes (instances). The six homogeneous DT models were created, and syntax patterns were extracted as explained in Section 3.2. That pattern extraction process produced 45,590 patterns for all the homogeneous models (10,562 for expressions; 28,163 for statements; 4,806 for methods; 336 for fields; 1,702 for types; and 21 for programs). Since this number poses high dimensionality to build a predictive model [43], we define the mechanism to reduce the number of syntax patterns described in Section 3.3.

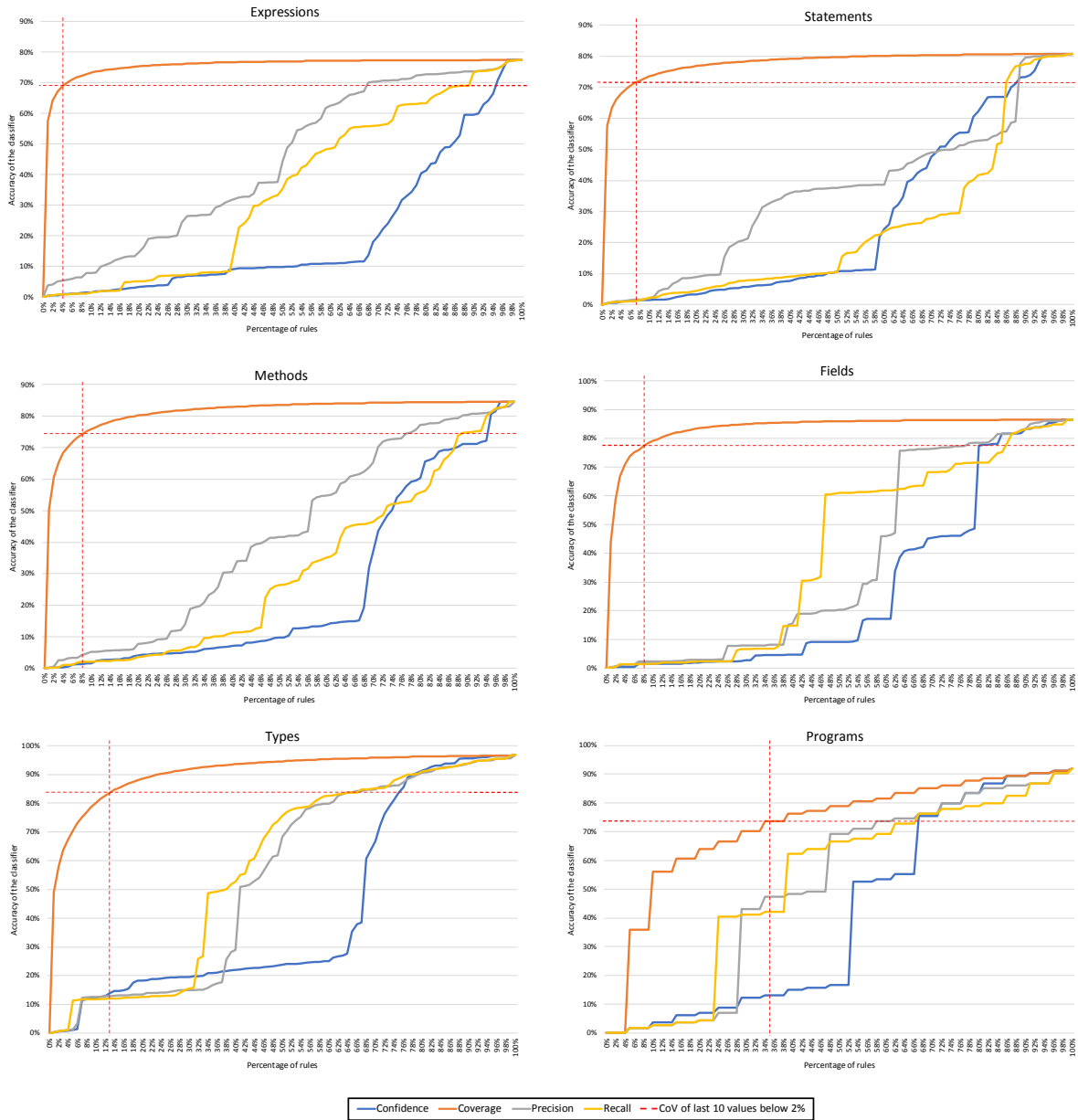


Figure 4: Classifier accuracy (y-axis) obtained with a percentage of rules (x-axis) with the highest confidence, coverage, precision and recall. For confidence, it is shown the CoV of the last 10 values below 2%.

We want to reduce the number of syntax patterns with the minimal reduction of classifier accuracy. To this aim, we analyze the influence of rule (syntax pattern) coverage, confidence, precision and recall on the accuracy of the whole DT classifier. Figure 4 presents the results. For each measure, it shows the accuracy of the classifier (y-axis) built with $n\%$ of the rules (x-axis) with the highest coverage, confidence, precision and recall. Figure 4 shows how, for all the datasets, coverage was the measure that

	Original rules	Rules selected	Rule reduction	Accuracy loss
Expressions	10,562	422	96,0%	11,1%
Statements	28,163	1,971	93,0%	11,6%
Methods	4,806	384	92,0%	12,3%
Fields	336	27	92,0%	10,6%
Types	1,702	221	87,0%	13,5%
Programs	21	7	65,0%	16,2%
Total	45,590	3,034	93.3%	12.4%

Table 3: Results of pattern selection.

selected the lowest number of patterns with the highest accuracy of the classifier.

Sorting the classification rules by coverage, we have to choose a percentage of rules (preferably low) with little penalty on the classifier accuracy. To this end, we used the Coefficient of Variation (CoV), defined as the ratio of the standard deviation to the mean. We measured the CoV of the classifier accuracy for the last ten percentages of rules in Figure 4, and selected the first percentage of rules where such CoV is lower than 2%. As shown in Figure 4, that value approximates the elbow value in all the coverage curves, representing a good trade-off between syntax pattern pruning and classifier accuracy.

Table 3 details the results of pattern selection. Out of 45.569 rules (syntax patterns), we select 3.027 (6.6%). Moreover, the average accuracy of the model is reduced in just 12.4% (from 10.6% in fields to 16.2% in programs).

4.4. Heterogeneous AST classification

Section 3.4 describes how heterogeneous datasets are created by combining the homogeneous datasets with the syntax patterns selected in the previous experiment. Now, we evaluate the performance of the heterogeneous DT models. For that purpose, we divide all the datasets into 80% of the instances for training and 20% for testing, using a stratified random sampling method [44]. We repeat the training plus testing process 30 times, measuring the mean, standard deviation and 95% confidence intervals of accuracy, F1 and AUC values [44]. Data split was random and stratified to ensure that the proportions between classes are the same in each fold, as they are in the whole dataset (50% / 50%).

In Figure 5, we can see the accuracy of the heterogeneous DT models. When a whole program is classified, the average accuracy of DTs is 99.6%. This performance is reduced when we classify smaller AST structures: 99.5% for types, 95.2% for methods, 91.4% for fields, 88.3% for statements, and 78.1% for expressions. Variability (confidence intervals) of the results is low (Table 4), because models were created with an important amount of data (more than 12 million instances in total; see Table 2). 95% confidence intervals for all the models are lower than 0.04% (Table 4).

Figure 5 compares our system with the two related works to classify the expertise

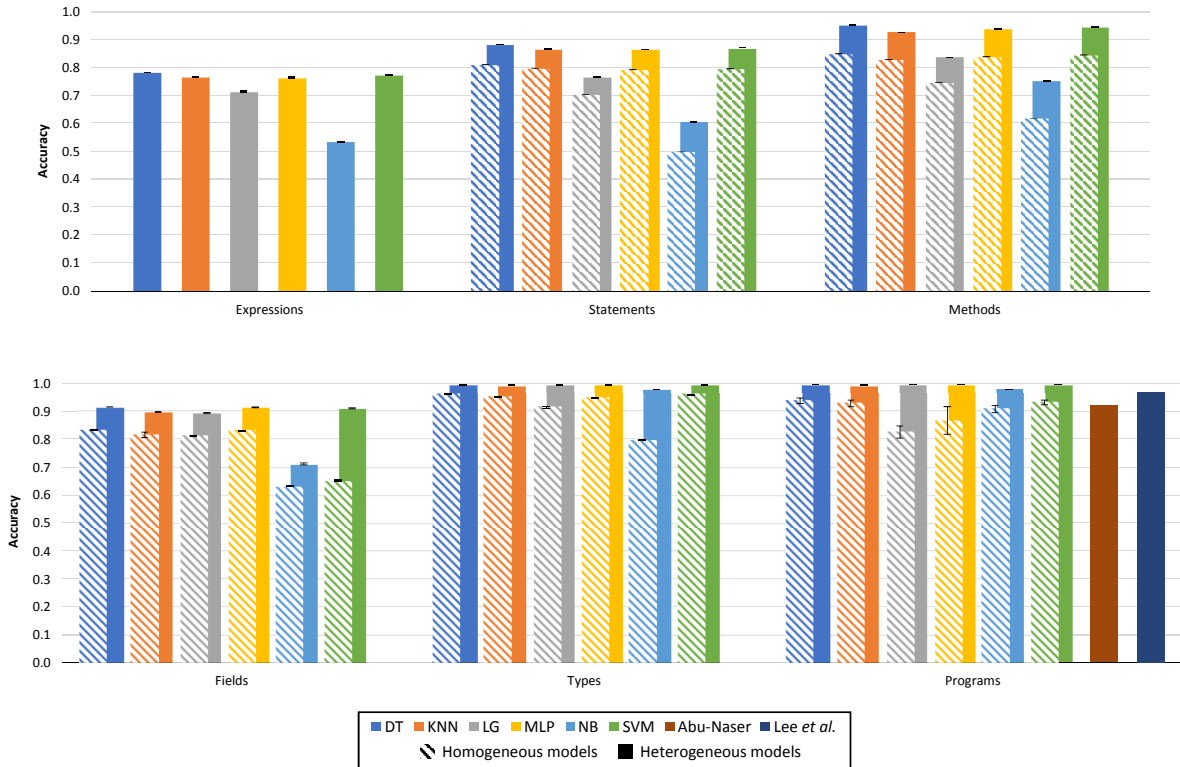


Figure 5: Accuracy of all the classifiers (whiskers represent 95% confidence intervals).

level of programmers, discussed in Section 2. The research work undertaken by Abu-Naser and Lee *et al.* classifies programmers with 92% and 97% accuracy, respectively. Our system provides 99.6% average accuracy with 0.02% error, so there seem to be a significant benefit³. Moreover, our system predicts the expertise level of programmers by just analyzing their code; it does not need to interact or observe them while they are coding.

Figure 5 also presents the performances of the homogeneous DT models, comparing them with the heterogeneous ones. The purpose of this comparison is to see whether the addition of child subAST patterns actually increases the accuracy of the classifiers. All the heterogeneous DT models improve the performance of the corresponding homogeneous ones. Accuracies of statements, methods, fields, types and programs are increased in 9%, 12%, 9.6%, 3.4% and 6%, respectively.

Besides the DT models to classify programmers, we built other classifiers with distinct techniques, using the same datasets. As mentioned, we selected DTs because they are interpretable white-box models, which allow us to extract the syntax patterns of the classifiers in order to implement the feature learning approach proposed in this article.

³The two related works do not provide 95% confidence intervals or data to compute a statistical hypothesis test. In addition, we could not repeat their experiments because they use electroencephalographic sensors, an eye-tracker, and a LP-ITS system that is not available for download.

		DT	KNN	LG	MLP	NB	SVM
Accuracy	Statements	0.883 ± 0.04%	0.866 ± 0.04%	0.765 ± 0.03%	0.864 ± 0.03%	0.605 ± 0.06%	0.869 ± 0.04%
	Methods	0.952 ± 0.03%	0.927 ± 0.03%	0.837 ± 0.02%	0.938 ± 0.03%	0.752 ± 0.05%	0.946 ± 0.04%
	Fields	0.914 ± 0.03%	0.896 ± 0.03%	0.891 ± 0.04%	0.912 ± 0.04%	0.711 ± 0.06%	0.910 ± 0.04%
	Types	0.995 ± 0.02%	0.992 ± 0.02%	0.995 ± 0.02%	0.995 ± 0.01%	0.978 ± 0.04%	0.993 ± 0.02%
	Programs	0.996 ± 0.02%	0.992 ± 0.03%	0.996 ± 0.02%	0.996 ± 0.02%	0.980 ± 0.04%	0.996 ± 0.01%
F1	Statements	0.883 ± 0.03%	0.866 ± 0.04%	0.765 ± 0.03%	0.864 ± 0.03%	0.606 ± 0.05%	0.870 ± 0.04%
	Methods	0.952 ± 0.03%	0.927 ± 0.03%	0.837 ± 0.02%	0.938 ± 0.03%	0.753 ± 0.05%	0.947 ± 0.04%
	Fields	0.914 ± 0.02%	0.896 ± 0.03%	0.891 ± 0.04%	0.912 ± 0.04%	0.711 ± 0.06%	0.910 ± 0.04%
	Types	0.995 ± 0.02%	0.992 ± 0.02%	0.995 ± 0.02%	0.995 ± 0.01%	0.979 ± 0.04%	0.993 ± 0.02%
	Programs	0.996 ± 0.02%	0.992 ± 0.03%	0.996 ± 0.02%	0.996 ± 0.02%	0.980 ± 0.04%	0.996 ± 0.01%
AUC	Statements	0.883 ± 0.04%	0.866 ± 0.04%	0.765 ± 0.03%	0.864 ± 0.03%	0.605 ± 0.06%	0.869 ± 0.04%
	Methods	0.952 ± 0.03%	0.927 ± 0.03%	0.837 ± 0.02%	0.938 ± 0.03%	0.752 ± 0.05%	0.946 ± 0.04%
	Fields	0.914 ± 0.03%	0.896 ± 0.03%	0.891 ± 0.04%	0.912 ± 0.04%	0.711 ± 0.06%	0.910 ± 0.04%
	Types	0.995 ± 0.02%	0.992 ± 0.02%	0.995 ± 0.02%	0.995 ± 0.01%	0.978 ± 0.04%	0.993 ± 0.02%
	Programs	0.996 ± 0.02%	0.992 ± 0.03%	0.996 ± 0.02%	0.996 ± 0.02%	0.980 ± 0.04%	0.996 ± 0.01%

Table 4: Performance of all the heterogeneous models (95% confidence intervals are expressed as percentages). Bold font represents the highest value. If one row has multiple cells in bold type, it means that there is not significant difference among them (p-value ≥ 0.05 , $\alpha = 0.05$).

		DT	KNN	LG	MLP	NB	SVM
Accuracy	Expressions	0.781 ± 0.01%	0.766 ± 0.17%	0.714 ± 0.01%	0.762 ± 0.15%	0.534 ± 0.01%	0.773 ± 0.16%
	Statements	0.810 ± 0.02%	0.795 ± 0.16%	0.701 ± 0.03%	0.793 ± 0.02%	0.496 ± 0.03%	0.797 ± 0.14%
	Methods	0.850 ± 0.04%	0.828 ± 0.06%	0.748 ± 0.06%	0.837 ± 0.05%	0.616 ± 0.06%	0.844 ± 0.05%
	Fields	0.834 ± 0.08%	0.818 ± 1.31%	0.814 ± 0.08%	0.833 ± 0.07%	0.629 ± 0.11%	0.651 ± 0.10%
	Types	0.962 ± 0.05%	0.954 ± 0.09%	0.916 ± 0.10%	0.951 ± 0.10%	0.798 ± 0.12%	0.962 ± 0.07%
	Programs	0.940 ± 1.18%	0.932 ± 1.36%	0.827 ± 2.53%	0.868 ± 5.68%	0.910 ± 1.42%	0.932 ± 1.02%
F1	Expressions	0.780 ± 0.01%	0.771 ± 0.16%	0.709 ± 0.01%	0.770 ± 0.13%	0.139 ± 0.13%	0.773 ± 0.16%
	Statements	0.801 ± 0.02%	0.804 ± 0.15%	0.682 ± 0.03%	0.786 ± 0.03%	0.631 ± 0.02%	0.807 ± 0.13%
	Methods	0.843 ± 0.05%	0.842 ± 0.05%	0.732 ± 0.08%	0.829 ± 0.06%	0.431 ± 0.18%	0.831 ± 0.06%
	Fields	0.837 ± 0.07%	0.818 ± 1.61%	0.816 ± 0.08%	0.836 ± 0.06%	0.431 ± 0.60%	0.738 ± 0.05%
	Types	0.961 ± 0.06%	0.953 ± 0.09%	0.914 ± 0.10%	0.949 ± 0.11%	0.766 ± 0.16%	0.961 ± 0.07%
	Programs	0.940 ± 1.17%	0.930 ± 1.42%	0.839 ± 2.16%	0.869 ± 6.45%	0.912 ± 1.39%	0.931 ± 1.02%
AUC	Expressions	0.781 ± 0.01%	0.766 ± 0.17%	0.714 ± 0.01%	0.762 ± 0.15%	0.534 ± 0.01%	0.773 ± 0.16%
	Statements	0.810 ± 0.02%	0.795 ± 0.16%	0.701 ± 0.03%	0.793 ± 0.02%	0.496 ± 0.03%	0.797 ± 0.14%
	Methods	0.850 ± 0.04%	0.828 ± 0.06%	0.748 ± 0.06%	0.837 ± 0.05%	0.616 ± 0.06%	0.844 ± 0.05%
	Fields	0.834 ± 0.08%	0.818 ± 1.31%	0.814 ± 0.08%	0.833 ± 0.07%	0.629 ± 0.11%	0.651 ± 0.10%
	Types	0.962 ± 0.05%	0.954 ± 0.09%	0.916 ± 0.10%	0.951 ± 0.10%	0.798 ± 0.12%	0.962 ± 0.07%
	Programs	0.940 ± 1.18%	0.932 ± 1.35%	0.827 ± 2.52%	0.868 ± 5.72%	0.910 ± 1.40%	0.932 ± 1.03%

Table 5: Performance of all the homogeneous models (95% confidence intervals are expressed as percentages). Bold font represents the highest value. If one row has multiple cells in bold type, it means that there is not significant difference among them (p-value ≥ 0.05 , $\alpha = 0.05$).

However, we want to see to what extent the classification accuracy of DTs is similar to other common machine-learning approaches.

In particular, we built other classifiers using logistic regression, (Gaussian) naïve Bayes, multilayer perceptron, support vector machines, and k -nearest neighbors. For all these models, we first perform a feature selection process and then hyper-parameter tuning. Features were selected with the `SelectFromModel` meta-transformer that chooses features depending on importance weights. The estimator used to select the features was a random forest classifier with 100 trees. Hyper-parameter tuning was done the

same way as for DTs —the different hyper-parameter options used can be consulted in [45]. We repeat the training plus testing process 30 times, computing the 95% confidence intervals. All the algorithms, including feature selection and hyper-parameter tuning, were executed in parallel using all the cores in our server.

Figure 5 and Table 5 show the performance of the different classifiers for the homogeneous datasets. DT is the method with the best average performance. We performed a statistical hypothesis test ($\alpha = 0.05$) to compute whether the performance of each method is significantly different to DT ($p\text{-value} < 0.05$). For accuracy and AUC measures, DT provides the highest performance (in three cases, statistical differences with SVM are not significant; see Table 5). For F1, there is one case (statements) where SVM performs better than DT; in the rest of scenarios, DT provides the highest F1 measures. For the heterogeneous models (Table 4), DT is the technique with the highest performance for all the measures (accuracy, F1 and AUC). When classifying types and programs, LG, MLP and SVM are not significantly different to DT. All these results validate that DT not only builds interpretable white-box models, but also provide excellent performance results for the given datasets.

4.5. Heterogeneous syntax pattern extraction

The expert and novice syntax patterns found in the heterogeneous models are a valuable outcome of our research. As mentioned, they could be used in programming courses, to improve the hints given by development environments, and to create Intelligent Tutoring Systems. As described in Section 3.5, we traversed the heterogeneous DT models to extract the final syntax patterns used by expert and novice Java programmers. For example, the following rule classifies a programmer as expert, using syntax patterns of program, type and method constructs:

```

if enumeration_percentage(program)>0 and interface_percentage(program)>1
  and  $\exists$  type1 . category(type1)==class and generic_types(type1)>0
  and  $\exists$  type2 . category(type2)==class and extend_classes(type2)>0
    and implement_interfaces(type2)>1
  and  $\exists$  method . number_statements(method)<=3
then expertise_level = expert

```

The previous pattern describes programs that contains enumeration and interface (more than 1%) types, implements no less than one generic type, at least one class extend another class and implements more than one interface, and one method has three or fewer statements.

Likewise, the following method pattern was extracted to classify beginners:

```

if not(isOverride(method)) and numberOfAnnotations(method)==0
  and numberOfParameters(method)==0 and not(isFinal(method))

```

```

and numberOfThrows(method)==0 and numberOfStatements(method)<=2
and visibility(method)==public and numberOfGenericTypes(method)==0
and namingConvention(method)==snake_case
and  $\exists$  statement . depth(statement)>=67
then expertise_level = beginner

```

Our system extracted 742, 721, 782, 636 and 575 heterogeneous syntax patterns for, respectively, statements, methods, fields, types and programs. All the patterns found are available for download at [45].

4.6. Scoring the expertise level of programmers

An important discussion regarding the classification method proposed in this article is about the binary character of classifying programmers as either experts or beginners. As we know, the classification of programmers by their expertise level is not binary, since many programmers may be classified as intermediate level. Fortunately, since the classifier infers the syntax patterns for novice and expert programmers, it is possible to measure the probability of being in one of these groups, and hence to score how close the programmer is to be an expert or beginner.

Logistic regression is a calibrated probabilistic classifier that provides a score that can be directly interpreted as a confidence level [46]. We built logistic regression models from the heterogeneous datasets to compute the probability of a programmer to be classified as novice or expert. Figure 6 shows the percentage of instances per score, for the two labels (beginner and expert). We can see how the most common score is a number between 0.9 and 1, because all the instances in the dataset are code written by either beginners or experts. As expected, expressions are the syntax patterns with the worst performance (71.4% of the instances are classified correctly), and programs outperform the rest of syntax constructs (99.6%). We can also see in Figure 6 that the model classifies experts better than beginners. It seems to be easier to identify the syntax patterns that expert programmers write, rather than those coded by beginners.

4.7. Execution time of the proposed method

Table 6 shows the execution times of all the phases of the proposed system. The system takes as input the Java programs described in Section 4.1, and produces six heterogeneous classifiers plus the final syntax patterns for each syntax construct.

The whole process took 6 hours and 12 minutes (22,287 seconds) to run in the computer described in Section 4.2. The two most expensive phases are the construction of datasets, which take 2.44 and 3.08 hours for the homogeneous and heterogeneous datasets, respectively.

The generated DT classifiers are able to predict the expertise level of Java programmers almost instantaneously (average execution time is 4.6 microseconds). Logistic regression and multilayer perceptron perform similarly (differences are not statistically significant). Naïve Bayes, SVM and KNN require, respectively, 3.87, 987 and 6079 times more execution time than DT to classify a Java programmer.

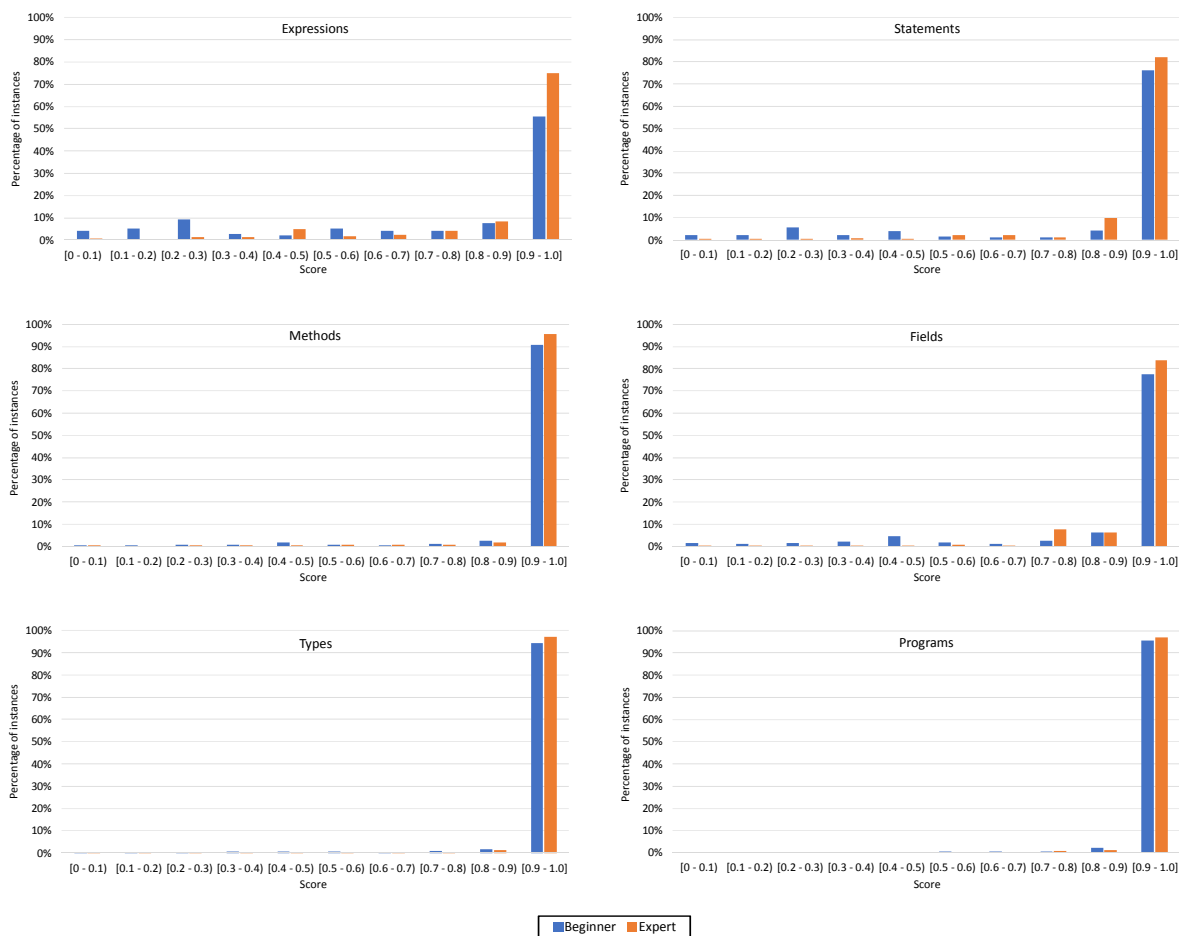


Figure 6: Percentage of instances per score, using a probabilistic LG model.

		Programs	Types	Methods	Fields	Statements	Expressions	Total
Homogen.	Dataset construction	1.961	46.14	296.4	113.1	1754	6582	8,793.2
	Model construction	0.030	0.193	0.064	0.028	0.487	2.375	3.2
	Syntax pattern extraction	0.009	1.043	13.24	0.439	312.5	1595	1,921.7
	Pattern selection and simplification	0.316	0.410	1.366	0.183	9.219	27.38	38.9
Heterogen.	Dataset construction	3320	3107	2399	1093	1168	—	11,087.6
	Model construction	0.210	0.241	0.209	0.117	0.654	—	1.4
	Syntax pattern extraction	0.914	1.164	21.21	0.994	401.1	—	425.4
	Pattern selection and simplification	0.449	0.534	2.456	0.436	11.954	—	15.8
Total		3,324	3,156	2,734	1,208	3,658	8,206	22,287.3

Table 6: Execution times (seconds) of all the modules in the architecture.

5. Conclusions

The proposed feature learning approach to classify heterogeneous compound tree structures provides higher accuracy than the existing methods to classify the programming expertise level of Java developers. The compound heterogeneous classifiers are enriched with the most determinant syntax patterns extracted from the homogeneous

models, significantly improving the performance of the classifiers. Our system allows classifying different syntax constructs found in Java programs. Decision trees provide good performance, scale well for large datasets, and create interpretable white-box models. Classification performance ranges from 78.1% when labeling expressions up to 99.6% when labeling programs. The interpretable white-box models obtained give us information about the syntax patterns used by expert and novice programmers. By using a probabilistic classifier, it is also possible to score the expertise level of programmers regarding the syntax patterns used in their code.

Future work will be using the same approach to see to what extent we can identify the programmer that developed a particular code fragment. We would also like to use it for identifying potential plagiarisms among students. We plan to use the heterogeneous models to see if student’s score is increased throughout a programming course.

Other lines of future work are focused on using alternative approaches for the same case scenario. Graph Neural Networks can be used to classify graph structures [29]. Although they do not build interpretable models, they might provide better performance. Inductive Logic Programming (ILP) is another approach to perform graph mining [33]. Some optimizations could be studied to optimize the particular problem of classifying programmers by their expertise level.

All the datasets, source code, selected features, model hyper-parameters, the syntax patterns found, and the evaluation data used in our work is available for download at <http://www.reflection.uniovi.es/bigcode/download/2019/fgcs>.

Acknowledgments

This work has been partially funded by the Spanish Department of Science, Innovation and Universities: project RTI2018-099235-B-I00. The authors have also received funds from the University of Oviedo through its support to official research groups (GR-2011-0040).

Appendix A. Features of the homogeneous datasets

Tables A1-A5 show the different features used to build the homogeneous datasets. We do not include any feature that may depend on the size of the program. For example, Table A5 does not include features such as the number of classes or interfaces. Their occurrence is considered, yet relative to the number of types used in the program.

Name	Description
Category	Syntactic category of the current node, given the abstract grammar for the Java language.
First, second and third child	Syntactic category of the corresponding child node.
Parent node	Syntactic category of the parent node.
Role	Role played by the current node in the structure of its parent node.
Height	Distance (number of edges) from the current node to the root node in the enclosing type (class, interface or enumeration).
Depth	Maximum distance (number of edges) of the longest path from the current node to a leaf node.

Table A1: Feature abstractions used for statements.

Name	Description
Visibility	Public, protected, package or private.
IsAbstract, IsStatic, IsFinal	True or false.
ReturnsVoid, Overrides	True or false.
Number of parameters	Number of declared parameters.
Number of generics	Number of generic types declared for that method.
Number of throws	Number of exceptions declared in the throws clause.
Number of annotations	Number of annotations declared for that method.
Number of statements	Number of statements used in the method body.
Number of local variables	Number of local variables declared.
Naming convention	Naming convention used for the method name (snake_case, upper, lower, camel_up or camel_down).
Main naming locals	Main naming convention used for local variables.

Table A2: Feature abstractions used for methods.

Name	Description
Visibility	Public, protected, package or private.
IsDefined	Whether a value is defined in its declaration.
IsStatic, IsFinal	True or false.
Number of annotations	Number of annotations declared for that field.
Value	If any, category of the expression assigned in its definition.
Naming convention	Snake_case, upper, lower, camel_up or camel_down.

Table A3: Feature abstractions used for fields.

Name	Description
Visibility	Public or package (non public).
Category	Class, interface or enumeration.
IsAbstract, IsStatic, IsFinal	True or false.
Extends	Whether the type extends another type.
Number of annotations	Number of annotations declared for that type.
Number of extends	Number of types that the current type extends (Java interfaces may extend any number of interfaces).
Number of implements	Number of interfaces implemented.
Number of generics	Number of generic types declared for that type.
Number of methods	Number of methods declared for that type.
Number of overloaded	Number of overloaded methods declared for that type.
Number of constructors	Number of constructors implemented in that type.
Number of fields	Number of fields defined in that type.
Number of nested classes	Number of nested classes defined in that type.
Number of inner classes	Number of inner classes defined in that type.
Number of fields	Number of fields defined in that type.
Naming convention	Naming convention used for the type name (snake_case, upper, lower, camel_up or camel_down).

Table A4: Feature abstractions used for types.

Name	Description
Class percentage	Percentage of classes (out of all the types) defined in that program.
Interface percentage	Percentage of interfaces (out of all the types) defined in that program.
Enum percentage	Percentage of enumerations (out of all the types) defined in that program.
Code in default package	Whether the program implements types in the default package.
Code in packages	Whether the program implements types inside packages.

Table A5: Feature abstractions used for programs.

References

- [1] V. Raychev, M. Vechev, A. Krause, Predicting Program Properties from “Big Code”, in: Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL ’15, New York, NY, USA, 2015, pp. 111–124.
- [2] F. Ortin, J. Escalada, O. Rodriguez-Prieto, Big Code: new opportunities for improving software construction, *Journal of Software* 11 (11) (2016) 1083–1088.
- [3] Defense Advanced Research Projects Agency, MUSE Envisions Mining “Big Code” to Improve Software Reliability and Construction, <http://www.darpa.mil/news-events/2014-03-06a> (2014).
- [4] S. Karaiyanov, V. Raychev, M. Vechev, Phrase-Based Statistical Translation of Programming Languages, in: Proceedings of the 2014 ACM International Symposo-

sium on New Ideas, New Paradigms, and Reflections on Programming & Software, Onward! 2014, ACM, New York, NY, USA, 2014, pp. 173–184.

- [5] F. Yamaguchi, M. Lottmann, K. Rieck, Generalized Vulnerability Extrapolation Using Abstract Syntax Trees, in: Proceedings of the 28th Annual Computer Security Applications Conference, ACSAC '12, ACM, New York, NY, USA, 2012, pp. 359–368.
- [6] D. Levy, L. Wolf, Learning to align the source code to the compiled object code, in: D. Precup, Y. W. Teh (Eds.), Proceedings of the 34th International Conference on Machine Learning, Vol. 70 of Proceedings of Machine Learning Research, PMLR, International Convention Centre, Sydney, Australia, 2017, pp. 2043–2051.
- [7] J. Escalada, F. Ortin, An adaptable infrastructure to generate training datasets for decompilation issues, in: Á. Rocha, A. M. Correia, F. B. Tan, K. A. Stroetmann (Eds.), New Perspectives in Information Systems and Technologies, Volume 2, Springer International Publishing, Cham, 2014, pp. 85–94.
- [8] M. Allamanis, E. T. Barr, P. Devanbu, C. Sutton, A survey of machine learning for big code and naturalness, ACM Computing Surveys 51 (4) (2018) 81:1–81:37.
- [9] S. Abu-Naser, Predicting learners performance using artificial neural networks in linear programming intelligent tutoring system, International Journal of Artificial Intelligence & Applications 3 (2012) 65–73.
- [10] D. Mazinanian, A. Ketkar, N. Tsantalis, D. Dig, Understanding the use of lambda expressions in java, Proceedings of the ACM on Programming Languages 1 (OOPSLA) (2017) 85:1–85:31.
- [11] Carnegie Mellon University, Software Engineering Institute, Java coding guidelines, <https://wiki.sei.cmu.edu/confluence/display/java/Java+Coding+Guidelines> (2019).
- [12] E. L. Seidel, H. Sibghat, K. Chaudhuri, W. Weimer, R. Jhala, Learning to blame: localizing novice type errors with data-driven diagnosis, in: Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications, OOPSLA '17, ACM, 2017, pp. 60:1–60:27.
- [13] H. Cai, V. W. Zheng, K. C.-C. Chang, A comprehensive survey of graph embedding: Problems, techniques, and applications, IEEE Transactions on Knowledge and Data Engineering 30 (2018) 1616–1637.
- [14] L. Rokach, O. Maimon, Top-down induction of decision trees classifiers - a survey, IEEE Transactions on Systems, Man, and Cybernetics: Systems, Part C 35 (4) (2005) 476–487.

- [15] S. Lee, D. Hooshyar, H. Ji, K. Nam, H. Lim, Mining biometric data to predict programmer expertise and task difficulty, *Cluster Computing* (2017) 1–11.
- [16] S. A. Naser, A. Ahmed, N. Al-Masri, Y. A. Sultan, Human computer interaction design of the LP-ITS: Linear programming intelligent tutoring systems, *International Journal of Artificial Intelligence & Applications* 2 (3) (2011) 60–70.
- [17] W. S. Evans, C. W. Fraser, F. Ma, Clone detection via structural abstraction, in: *14th Working Conference on Reverse Engineering (WCRE 2007)*, 2007, pp. 150–159.
- [18] C. K. Roy, J. R. Cordy, R. Koschke, Comparison and evaluation of code clone detection techniques and tools: A qualitative approach, *Science of Computing Programming* 74 (7) (2009) 470–495.
- [19] J. Mayrand, C. Leblanc, E. Merlo, Experiment on the automatic detection of function clones in a software system using metrics, in: *Proceedings of the International Conference on Software Maintenance (ICSM)*, 1996, pp. 244–253.
- [20] I. Baxter, A. Yahin, L. Moura, M. Sant’Anna, L. Bier, Clone detection using abstract syntax trees, in: *Proceedings of the International Conference on Software Maintenance*, 1998, pp. 368–377.
- [21] Axivion, Project Bauhaus (2019).
URL <http://www.bauhaus-stuttgart.de>
- [22] V. Wahler, D. Seipel, J. W. v. Gudenberg, G. Fischer, Clone detection in source code by frequent itemset techniques, in: *Proceedings of the Fourth IEEE International Workshop in Source Code Analysis and Manipulation, SCAM ’04*, Washington, DC, USA, 2004, pp. 128–135.
- [23] R. Koschke, R. Falke, P. Frenzel, Clone detection using abstract syntax suffix trees, in: *Proceedings of the 13th Working Conference on Reverse Engineering, WCRE ’06*, Washington, DC, USA, 2006, pp. 253–262.
- [24] L. Jiang, G. Misherghi, Z. Su, S. Glondu, Deckard: Scalable and accurate tree-based detection of code clones, in: *Proceedings of the 29th International Conference on Software Engineering, ICSE ’07*, Washington, DC, USA, 2007, pp. 96–105.
- [25] A. Taherkhani, Using decision tree classifiers in source code analysis to recognize algorithms, *The Computer Journal* 54 (11) (2011) 1845–1860.
- [26] J. Escalada, F. Ortin, T. Scully, An Efficient Platform for the Automatic Extraction of Patterns in Native Code, *Scientific Programming* 2017 (2017) 1–16.

- [27] V. Raychev, M. Vechev, E. Yahav, Code Completion with Statistical Language Models, in: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, New York, NY, USA, 2014, pp. 419–428.
- [28] M. Allamanis, C. Sutton, Mining idioms from source code, in: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2014, New York, NY, USA, 2014, pp. 472–483.
- [29] J. Zhou, G. Cui, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, M. Sun, Graph neural networks: A review of methods and applications (2018). [arXiv:1812.08434](https://arxiv.org/abs/1812.08434).
- [30] M. Lu, D. Tan, N. Xiong, Z. Chen, H. Li, Program classification using gated graph attention neural network for online programming service (2019). [arXiv:1903.03804](https://arxiv.org/abs/1903.03804).
- [31] J. D. Lafferty, A. McCallum, F. C. N. Pereira, Conditional random fields: Probabilistic models for segmenting and labeling sequence data, in: Proceedings of the Eighteenth International Conference on Machine Learning, ICML, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001, pp. 282–289.
- [32] T. Cohn, Efficient inference in large conditional random fields, in: J. Fürnkranz, T. Scheffer, M. Spiliopoulou (Eds.), European Conference on Machine Learning, ECML, Springer Berlin Heidelberg, Berlin, Heidelberg, 2006, pp. 606–613.
- [33] H. Blockeel, T. Witsenburg, J. N. Kok, Graphs, hypergraphs, and inductive logic programming, in: P. Frasconi, K. Kersting, K. Tsuda (Eds.), Proceedings of the 5th International Workshop on Mining and Learning with Graphs, MLG' 07, 2007, pp. 93–96.
- [34] A. Sivaraman, T. Zhang, G. Van den Broeck, M. Kim, Active inductive logic programming for code search, in: Proceedings of the 41st International Conference on Software Engineering, ICSE, IEEE Press, Piscataway, NJ, USA, 2019, pp. 292–303.
- [35] Q. Zeng, J. M. Patel, D. Page, Quickfoil: Scalable inductive logic programming, Proceedings of the VLDB Endowment 8 (3) (2014) 197–208.
- [36] M. Collins, N. Duffy, New ranking algorithms for parsing and tagging: Kernels over discrete structures, and the voted perceptron, in: Proceedings of the 40th Annual Meeting on Association for Computational Linguistics, ACL '02, Stroudsburg, PA, USA, 2002, pp. 263–270.
- [37] D. Fu, Y. Xu, H. Yu, B. Yang, WASTK: A Weighted Abstract Syntax Tree Kernel method for source code plagiarism detection, Scientific Programming 2017 (2017) 7809047:1–7809047:8.

- [38] K. Rieck, T. Krueger, U. Brefeld, K. Müller, Approximate tree kernels, *Journal of Machine Learning Research* 11 (2010) 555–580.
- [39] T. Parr, *The Definitive ANTLR 4 Reference*, The Pragmatic Bookshelf, 2013.
- [40] G. Erich, H. Richard, J. Ralph, V. John, *Design patterns: elements of reusable object-oriented software*, Addison-Wesley Professional Computing Series, 1995.
- [41] A. A. Freitas, On rule interestingness measures, in: R. Miles, M. Moulton, M. Bramer (Eds.), *Research and Development in Expert Systems XV*, Springer London, London, 1999, pp. 147–158.
- [42] L. Breiman, J. H. Friedman, R. A. Olshen, C. J. Stone, *Classification and Regression Trees*, Wadsworth and Brooks, Monterey, CA, 1984.
- [43] S. Theodoridis, K. Koutroumbas, *Pattern Recognition*, 4th Edition, Academic Press, Inc., Orlando, FL, USA, 2008.
- [44] Z. Reitermanová, Data splitting, in: *Proceedings of the 19th Annual Conference of Doctoral Student, WDS*, 2010, pp. 31–26.
- [45] F. Ortin, Heterogeneous tree structure classification to label Java programmers according to their expertise level (support material website), <http://www.reflection.uniovi.es/bigcode/download/2019/fgcs> (2019).
- [46] A. Niculescu-Mizil, R. Caruana, Predicting good probabilities with supervised learning, in: *Proceedings of the 22Nd International Conference on Machine Learning, ICML '05*, New York, NY, USA, 2005, pp. 625–632.