Expert Systems With Applications 215 (2023) 119398

Contents lists available at ScienceDirect

Expert Systems With Applications

journal homepage: www.elsevier.com/locate/eswa

Analyzing syntactic constructs of Java programs with machine learning

Francisco Ortin [a,b,*], Guillermo Facundo [a], Miguel Garcia [a]

[a] University of Oviedo, Computer Science Department, c/Federico Garcia Lorca 18, 33007, Oviedo, Spain
[b] Munster Technological University, Computer Science Department, Rossa Avenue, Bishopstown, Cork, Ireland

ARTICLE INFO

Keywords:
Abstract syntax tree
Programming language
Data mining
Feature engineering
Programming idiom
Heterogeneous dataset

ABSTRACT

The massive number of open-source projects in public repositories has notably increased in the last years. Such repositories represent valuable information to be mined for different purposes, such as documenting recurrent syntactic constructs, analyzing the particular constructs used by experts and beginners, using them to teach programming and to detect bad programming practices, and building programming tools such as decompilers, Integrated Development Environments or Intelligent Tutoring Systems. An inherent problem of source code is that its syntactic information is represented with tree structures, while traditional machine learning algorithms use *n*-dimensional datasets. Therefore, we present a feature engineering process to translate tree structures into homogeneous and heterogeneous *n*-dimensional datasets to be mined. Then, we run different interpretable (supervised and unsupervised) machine learning algorithms to mine the syntactic information of more than 17 million syntactic constructs in Java code. The results reveal interesting information such as the Java constructs that are barely (and widely) used (e.g., bitwise operators, union types and `static` blocks), different language features and patterns mostly (and barely) used by beginners (and experts), the discovery of particular types of source code (e.g., helper or utility classes, data transfer objects and too complex abstractions), and how complexity is an inherent characteristic in some clusters of syntactic constructs.

1. Introduction

In the last decade, there has been an important growth in the use of source code repositories, such as GitHub, SourceForge, BitBucket and CodePlex (Ortin, Escalada, & Rodriguez-Prieto, 2016). Taking GitHub as an example, it reached 1 million repositories in July 2010, 2.4 years after its foundation in February 2008 (GitHub, 2022b). Today, GitHub hosts more than 200 million repositories and 83 million developers (GitHub, 2022c). The vast amount of open-source code projects available in such repositories represent important information to learn from. In fact, different open-source corpora have been utilized to improve software development scenarios, such as language translation (Aggarwal, Salameh, & Hindle, 2015), error correction (Bhatia & Singh, 2016), and automatic code documentation (Barone & Sennrich, 2017), completion (Bhoopchand, Rocktäschel, Barr, & Riedel, 2016) and generation (GitHub, 2022a).

In textual programming languages, programs are collections of source code files—together with additional resources—coded as text. That textual information actually encloses syntactic and semantic information that compilers and interpreters, after different analysis phases, represent internally as trees and graphs (Rodriguez-Prieto, Mycroft, & Ortin, 2020). Abstract Syntax Trees (ASTs) are tree data structures that most language processors use to represent the syntactic information of the input program, once parsing has taken place (Andrew & Jens, 2002). Each AST node represents a syntactic construct in an input program, such as method definition, field definition, assignment statement or arithmetic expression.

ASTs represent syntactic information besides the textual data provided in the source code. State-of-the-art machine learning techniques, such as Graph Neural Networks (GNNs), are able to learn not only from node features, but also from the structure of ASTs (syntactic information) and graphs (semantic information) (Allamanis, 2022). Such capability has been exploited to implement advanced software development tools, including the detection of bugs not captured by common program analyzers (Pradel & Sen, 2018), probabilistic type inference (Allamanis, Barr, Ducousso, & Gao, 2020), and semantic code search (Arakelyan, et al., 2022).

Although GNNs represent a powerful mechanism to build predictive models from graph and tree data structures, the trained models act as black boxes to classify programs, and hence such models are not straightforwardly interpretable by humans (Allamanis, 2022). On the contrary, interpretable machine learning algorithms could be used to extract information from syntactic constructs. Such information

* Corresponding author at: University of Oviedo, Computer Science Department, c/Federico Garcia Lorca 18, 33007, Oviedo, Spain.
E-mail addresses: ortin@uniovi.es (F. Ortin), facundoguillermo@uniovi.es (G. Facundo), garciarmiguel@uniovi.es (M. Garcia).
URL: https://www.reflection.uniovi.es/ortin/ (F. Ortin).

https://doi.org/10.1016/j.eswa.2022.119398
Received 25 June 2022; Received in revised form 17 October 2022; Accepted 1 December 2022
Available online 5 December 2022

could be useful for different scenarios. For example, it could be documented the recurrent constructs written by beginners, average and expert programmers. Programming lecturers can identify the recurrent programming patterns used by students, including those that are error-prone, and explain how they could be improved with better alternatives (e.g., programming patterns used by experts) (Iyer & Zilles, 2021). Those patterns would also be useful in the construction of decompilers, Integrated Development Environments (IDEs) and Intelligent Tutoring Systems (ITSs) (Losada, Facundo, Garcia, & Ortin, 2022).

The source code could also be used to cluster programmers regarding the syntactic constructs that appear in their programs. Such clusters may be utilized later to improve the programmer's skills. For example, an IDE could suggest the programmer syntactic constructs that barely occur in her cluster, written by more experienced programmers. Together with the syntactic construct suggested a brief explanation of the benefits would be provided. The syntactic constructs of those clusters with the highest number of programmers could also be analyzed to know which potential IDE features may have a stronger impact.

The analysis of syntactic constructs of source code has been previously tackled (detailed in Section 2). Some of the works retrieve information by analyzing the grammar rules used to parse a program (Qiu, Li, Barr, & Su, 2017), reducing the information that could be mined from the AST. Nonparametric Bayesian probabilistic substitution grammars have shown good results to extract programming idioms (Allamanis & Sutton, 2014). Even though such idioms are parameterized with metavariables, they represent very specific code fragments rather than general syntactic constructs (Section 2). Association rules have also been used to mine syntactic information of source code, producing too many rules with very low support, being hard to understand, and representing too specific information (Losada et al., 2022). Other works use AST similarity measures to numerically compare two ASTs, but they have only shown benefits when applied to small pieces of code rather than to whole programs (Choudhury, Yin, & Fox, 2016; Yin, Moghadam, & Fox, 2015). Various approaches build classifiers of syntactic constructs, but the resulting models lack interpretability (Allamanis, 2022; Baxter, Yahin, Moura, Sant'Anna, & Bier, 1998; Ortin, Rodriguez-Prieto, Pascual, & Garcia, 2020).

Given the limitations of the existing works to analyze the syntactic constructs in source code, we present a system with the following contributions:

- A feature engineering process to translate heterogeneous tree structures into n-dimensional datasets (Section 3.1). We apply that feature engineering process to the particular case scenario of ASTs. ASTs are translated into a collection of homogeneous and heterogeneous datasets, so that interpretable machine learning algorithms could be run.
- An open-source implementation of a Java compiler plugin that implements the proposed feature engineering process (Section 3). It takes any compilable Java project and stores its ASTs into seven homogeneous and five heterogeneous datasets.
- An analysis of the syntactic constructs used by Java programmers (Section 5). We document those syntactic constructs mostly (and barely) used, the constructs that categorize programmers' expertise, and the most influential variables in that categorization.
- An analysis and visualization of the similar recurrent syntactic patterns found in Java code (Sections 5.3 and 5.6). For example, the k-means clustering algorithm was able to identify a cluster that represents the helper and utility classes used in Java projects.

The rest of this article is structured as follows. The next section describes the related work, and Section 3 presents the architecture of our system. Section 4 details the methodology used, and the results and discussions are depicted in Section 5. Section 6 presents a discussion about language dependency. Conclusions and future work are presented in Sections 7 and 8.

2. Related work

Allamanis and Sutton define a method to automatically extract programming idioms from the source code, finding similarities recurring across Java projects (Allamanis & Sutton, 2014). Their system, called Haggis, retrieves frequent idioms from source code by using nonparametric Bayesian probabilistic tree substitution grammars. They trained their model with multiple open-source projects and found common idioms for object creation, exception handling, and resource management. The main difference between Haggis and our system is that they retrieve idioms rather than abstract syntactic constructs. Idioms represent code fragments that may hold metavariables. For example, the most common idiom they found was "`channel=connection.createChannel();`", whereas its corresponding syntactic construct would be *an assignment statement that stores in a variable the result of a method invocation with no parameters*. Allamanis and Sutton do not perform other analyses such as anomalous detection, idiom association to programming expertise and clustering.

Iyer and Zilles studied 12 first-year programming courses in Computer Science degrees, from 9 distinct universities. They manually analyzed the syntactic patterns that students must handle to pass all the exams and assignments (Iyer & Zilles, 2021). According to their work, 15 different patterns are needed to solve all the proposed activities. Nine of those patterns were taught in 9 of the 12 courses, and 5 of them were only addressed in 3 courses. That shows that students must be able to use certain syntactic patterns not taught by lecturers. Unlike Iyer and Zilles, our system automatically retrieves the syntactic constructs from Java code, making it easier to analyze massive code bases and perform more analyses.

Baxter et al. used ASTs to create a tool capable of detecting duplicate code fragments (Baxter, et al., 1998). They analyzed the ASTs of program fragments with more than 400,000 lines of code, finding that around 12.7% of that code was duplicated. Additionally, their system uses the duplicated code patterns to suggest modifications to the programmer, assisting them in the refactoring actions needed to avoid code duplication. Baxter et al. defined a three-step tree similarity algorithm to compare ASTs. That algorithm is used as a kernel function for Support Vector Machine (SVM) classifiers. Although the trained models could be used to classify programmers by their expertise, SVM models are hard to interpret.

Dong Qiu et al. presented an empirical analysis of the use of language constructs in Java (Qiu et al., 2017). They focus their study on three different approaches relative to grammar rules: popularity, usage over time and dependency among constructs. After analyzing more than 140 million lines of code from open-source repositories, they found that 20% of the most-used rules account for 85% of all rule usage. They also discovered that most of the syntactic constructs remain stable over time and are not influenced by new language features. They also concluded that 6% of the syntactic constructs strongly depend on other constructs (e.g., 1/5 of `if` statements contain another `if` statement in their body). Qiu et al. count the number of times each grammar rule is applied upon parsing. This limits the kind of information to be mined, compared to our AST analyses. For example, one conclusion is that method declarations appear in almost every project. However, their approach is not able to identify which method features are used the most—such features include method modifiers (e.g., `static`, `final`, `abstract`, `synchronized`, `strictfp` and `native`), visibility, annotations, generics, return types and parameters.

Several works have used machine learning to classify syntactic patterns. In the work carried out by Ortin et al. models are created to classify programmers by their experience level (Ortin et al., 2020). They start from a set of ASTs with the same structure, and manually translate them into tables that represent the characteristics of the tree nodes. The tables are used to build decision trees. Classification rules are then extracted from the decision trees and used as new features to enrich the existing classifiers. The resulting decision trees are formed

with a combination of the antecedents of classification rules extracted from other decision trees. The resulting models are very hard to interpret, because they use classification rules with a huge number of conditions in the antecedent (features of the trees are, in turn, rule antecedents). Although they lack interpretability, those models provide very high accuracy, telling novices from experts with up to 99.6% accuracy when the source code of the whole project is passed to the classifier.

Choudhury et al. created AutoStyle, a system to automatically provide students with instructor-authored guidance for their programming assignments (Choudhury et al., 2016). First, AutoStyle uses the normalized tree edit distance (n-TED) of the ASTs as the similarity metric to translate ASTs into numeric values. n-TED is a common measure of similarity between two code fragments (Yin et al., 2015). Then, they run the k-means, DBSCAN and OPTICS clustering algorithms to detect groups of similar constructs. Once the clusters are analyzed, the instructor writes feedback and guidance reports about each cluster, and hence avoids the analysis of all the programs submitted by the students. When a student submits a new program, its cluster is found and automatic feedback is given. This process has to be repeated for each programming assignment. In their evaluation, they observed that 70% of the students using AutoStyle were able to finally reach the optimal solution, compared to the 13% of the students in the control group. The system was only used to evaluate different implementations of the same function. n-TED measures the number of modifications necessary to transform an AST into another one, weighting nodes with values inverse to their distance to the root. This makes n-TED not to be able to group significantly different programs with many similar subASTs—a common way to identify expert and novice programmers.

This paper extends the work in Losada et al. (2022), which used association rules to extract information from Java source code. Such rules express the relationships among syntactic constructs, but show the common drawbacks of association rules (Kaur, 2014): a huge number of rules are generated, most of them hold obvious information, and all the features in the dataset must be binary. The rules found showed low support and hence represent too specific information. The discretization of numeric features also caused rules harder to understand and with too many conditions in the antecedents. In the present article, we show how the classification rules obtained with decision tree learning and rule induction provide rules with fewer conditions and higher support, being able to retrieve more valuable information from data. Logistic regression models provide us with information about the syntactic constructs used by experts and beginners. The most common syntactic constructs are analyzed. We also discuss the Java patterns found by a clustering algorithm and analyze if programming expertise is inherent in some clusters. Syntactic constructs are also visualized as two-dimensional data to analyze patterns in different language constructs.

3. Architecture

As shown in Fig. 1, we take Java files from GitHub and Java students enrolled in two year-1 programming courses of a Software Engineering degree (Section 4.1). The output of our system is a collection of reports describing the syntactic information extracted from the programs.

The first module of our system is a modification of the Open JDK compiler. We developed a new plugin that modifies the ASTs created by the Java compiler (Oracle, 2022b). This is done by implementing the Visitor design pattern (Gamma, Helm, Johnson, & Vlissides, 1994) that, traversing the original AST, creates a new AST with more specific information. We defined new and more specific AST classes. For example, the general `BinaryTree` node for expressions is replaced with `Arithmetic`, `Logical`, `Comparison` and `Bitwise`, among others (Losada et al., 2022). The original 56 AST classes were extended to 111.

A feature engineering process (detailed in Section 3.1) translates tree structures (ASTs) into *n*-dimensional datasets, so that interpretable machine learning algorithms can be executed. For Java, we identified seven types of homogeneous syntactic constructs: programs (a collection of Java files), type definitions (classes, interfaces, enumerations and records), field definitions, method definitions, statements, expressions and types. The Visitor design pattern is used to traverse the ASTs and store their information in the seven homogeneous datasets (each type of syntactic construct is stored in a different dataset). Then, five heterogeneous datasets are generated from the homogeneous ones (detailed in Section 3.1).

Table 1
Features defined for the expressions dataset.

Name	Type	Description
Category	Nominal	Syntactic category of the current node (e.g., Arithmetic, Comparison and Logical).
1st, 2nd and 3rd child	Nominal	Syntactic category of the corresponding child node.
Parent node	Nominal	Syntactic category of the parent node.
Role	Nominal	Role played by the current node in the structure of its parent node.
Height	Integer	Distance (number of edges) from the current node to the root node in the enclosing type (class, interface or enumeration).
Depth	Integer	Maximum distance (number of edges) of the longest path from the current node to a leaf node.
Expertise	Nominal	Beginner or Expert.

Fig. 1 also shows how the *n*-dimensional datasets are processed through different steps to generate the final reports of distinct analyses. First, the datasets are processed to convert data (e.g., normalization and discretization) before running each machine learning algorithm. Then, univariate and multivariate anomalies are detected. Some outliers are removed from the dataset because they do not represent valid programs (e.g., incomplete assignments of students). Afterwards, we train the interpretable models with the datasets. The parameters of each model are finally analyzed to emit the final reports about the different syntactic constructs used by Java programmers.

3.1. Homogeneous and heterogeneous datasets

As mentioned, our system generates seven homogeneous and five heterogeneous datasets from ASTs. Each type of syntactic construct is stored in a homogeneous dataset: programs, type definitions, field definitions, method definitions, statements, expressions and types. Models created with homogeneous datasets provide us with information about each particular type of syntactic construct.

Table 1 shows the structure of the dataset defined for expressions—for the sake of clarity, the rest of the datasets are shown in Appendix. The node category feature is a nominal variable that identifies the AST node type, which is also used to provide information about the parent and child nodes. Besides the (integer) height and depth of the node in the tree, the role (nominal) feature indicates the role played in the parent node (e.g., an assignment node could be part of either the condition or the body of a while parent statement). The last feature indicates whether the programmer is either a beginner (year-1 student) or an expert programmer.

The homogeneous datasets store information about the same type of syntactic constructs. However, syntactic patterns comprise different kinds of syntactic constructs. For example, an assignment statement consists of its left- and right-hand side expressions; a method definition is made up of the statements inside its body; and a Java program is a collection of type definitions. Thus, we also create heterogeneous datasets that combine the information of different homogeneous syntactic constructs. The parameters of the models created with the heterogeneous datasets allow us to analyze the combination of different kinds of syntactic constructs.

The heterogeneous datasets are built by applying drill-down operations to the homogeneous tables, producing the following datasets:

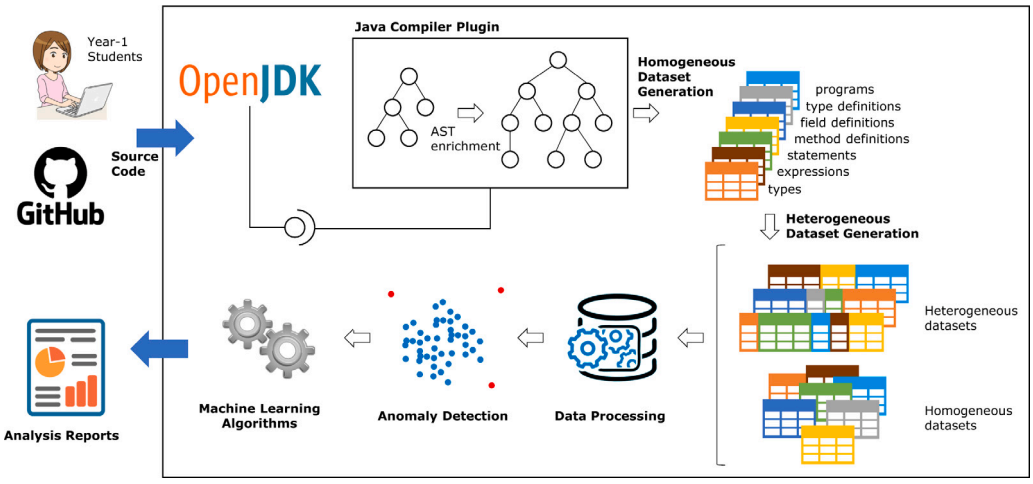

Fig. 1. Architecture of the syntactic constructs reports system.

Table 2
Number of entries (AST nodes) of the homogeneous datasets.

	Beginner	Expert	Total
Expressions	3,939,626	7,296,366	11,235,992
Statements	1,173,327	1,963,408	3,136,735
Types	628,100	1,131,536	1,759,636
Method definitions	256,562	369,074	625,636
Field definitions	104,579	135,419	239,998
Type definitions	38,452	58,504	96,956
Programs	4515	136	4651
Total	6,145,161	10,954,443	17,099,604

1. One entry per type definition, including the features of the corresponding program.
2. One entry per field definition, including the features of the program and the type used to define the field.
3. One entry per method definition, adding the features of the program and the return and parameter types.
4. One entry per statement, including the program and method features it was defined in, together with the first three child expressions (when applicable).
5. One entry per expression, including the features of its program, method and statement.

4. Methodology

This section describes how the different datasets were created. Afterwards, we detail the machine learning and data processing algorithms used.

4.1. Datasets

To build the datasets, we took Java code from different sources and labeled it as either beginner or expert. In the case of beginners, the code was gathered from first-year undergraduate students in the Software Engineering degree at the University of Oviedo. We took the code they wrote for the assignments in two programming courses. Overall, we collected 35,309 Java files from 4515 programs (Table 2).

For expert programmers, we took the source code of different public open-source Java repositories from GitHub. We selected the active projects with the highest number of contributors: Chromium, LibreOffice, MySQL, OpenJDK and Amazon Web Services. These projects are implemented with 43,765 Java files in 136 different programs (AWS comprises 133 distinct projects).

We passed all these files to the Java compiler plugin we developed (Fig. 1). Their ASTs are enriched with more specific information, and traversed to generate the seven homogeneous datasets in Fig. 1. Table 2 summarizes the number of entries in those datasets, holding more than 17 million AST nodes.

4.2. Anomaly detection

Anomaly or outlier detection aims to identify unusual data records which deviate significantly from the majority of the data. In our case study, outliers represent anomalous syntactic constructs coded by the programmers. Anomalous samples provide important information to analyze, and sometimes reflect, invalid data (e.g., programs that should not be included in the dataset).

For univariate outlier detection, we used Tukey's fences that identify as an outlier those instances that do not belong to the interval described in Eq. (1), where Q_n represents the n quartile.

$$[Q_1 - 3 \times (Q_3 - Q_1), \quad Q_3 + 3 \times (Q_3 - Q_1)] \tag{1}$$

For multivariate anomaly detection, the isolation forest algorithm was used (Liu, Ting, & Zhou, 2008) (`IsolationForest` class in scikit-learn (SciKit-Learn, 2022d)). Isolation forest identifies outliers by considering how far a data point (instance) is from the rest of the data. The contamination hyperparameter specifies the proportion of outliers in the dataset. We found 1% (0.01) as the contamination value that identified outliers in our datasets the best.

The previous two approaches for identifying univariate and multivariate anomalies are applicable to numeric features. For categorical values, we distinguish anomalous values when the number of occurrences is lower than 0.2% divided by the number of possible values. For example, the value of a binary feature is considered anomalous when its value occurs in less than 0.1% of all the instances in the dataset.

Section 5.1 analyzes and discusses the anomalous syntactic constructs detected in our dataset. When an outlier represents an invalid program, it is deleted from the dataset (i.e., it is not included as an input for the machine learning algorithms).

4.3. Most frequent syntactic constructs

After analyzing the anomalous syntactic constructs in Java code, we studied the most frequent constructs. For that purpose, we performed a frequency analysis of the different features in the dataset. To compute the frequency of the numeric features, we undertook an equal-with discretization (each bin has the same width) with three different bins. Categorical features were not modified.

After converting all the features to categorical values, the following analysis was performed:

Table 3
Number of features and clusters of the 12 datasets.

	Dataset	Features	Clusters
Homogeneous	Expressions	8	6
	Statements	12	6
	Types	7	7
	Method definitions	25	5
	Field definitions	10	5
	Type definitions	24	6
	Programs	8	5
Heterogeneous	Type definitions + programs	29	6
	Field definitions + programs + types	39	4
	Method definitions + programs + types	81	4
	Statements + programs + method definition + types + expressions	112	6
	Expressions + programs + method definition + types + statements	96	6

1. Types of AST nodes. For those individuals (syntactic constructs) that hold the type of its AST node (i.e., the *category* feature in Tables 1, A.1, A.2 and A.5), we computed the frequency of each AST node in the dataset. In this way, we can document the most (and least) frequent expressions, statements, types and type definitions.
2. Features of syntactic constructs. We undertook the same analysis for every single feature of all the homogeneous datasets. With such an analysis, it is possible to know, for example, the most frequent field and method modifiers, visibility levels, return types and field initializations.
3. Combinations of different features. Iterating through all the combinations of 2, 3 and 4 features, we analyzed the frequency of all the syntactic constructs made up of those combinations. In this way, it is possible to know the frequency of `public static final` (constant) field definitions, what is the most common type of array, and whether it is more common to overload methods or constructors.

4.4. Data visualization

Dimensionality reduction is the transformation of data from a high-dimensional space into a low-dimensional one, retaining meaningful properties of the original data. Table 3 shows the number of features of the 12 datasets used in our study. By applying dimensionality reduction techniques, the high-dimensional datasets can be embedded in a low-dimensional space for visualization. In those visualizations, points represent instances (syntactic constructs in our study) in such a way that similar instances are modeled by nearby points, and dissimilar instances are represented by distant ones. By plotting the syntactic constructs coded by experts and beginners with different colors, it is possible to visually identify different patterns for both groups.

We used four different algorithms: Principal Component Analysis (PCA) (Jolliffe & Cadima, 2016), NonNegative Matrix Factorization (NMF) (Lee & Seung, 2000), t-distributed Stochastic Neighbor Embedding (t-SNE) (van der Maaten & Hinton, 2008) and Kernel PCA (Schölkopf, Smola, & Müller, 1997) (with radial basis as the kernel function)—the `PCA`, `NMF`, `TSNE` and `KernelPCA` classes in scikit-learn were utilized (SciKit-Learn, 2022a). The first two algorithms are linear and the two last ones are nonlinear.

The datasets were reduced to two-dimensional data, and the resulting data were visualized to graphically analyze the existence of different patterns related to the programmer's expertise. Before running the algorithms, the categorical features were translated to one-hot encoding. The numeric features were normalized to values between 0 and 1 with Eq. (2), where vector $\boldsymbol{x}$ represents all the values of the numeric feature x, and x_i the value of the i^{th} sample or instance.

$$x_i^{normalized} = \frac{x_i - min(\boldsymbol{x})}{max(\boldsymbol{x}) - min(\boldsymbol{x})} \tag{2}$$

4.5. Logistic regression

After applying the data transformation and normalization described in the previous paragraph, we built 12 logistic regression models (one per dataset) to classify programmers regarding their expertise. The aim is not only to obtain classifiers, but also to interpret the information provided by the classifiers to analyze the syntactic constructs used by programmers (the objective of this article).

Logistic regression is a statistical model that outputs the probability of one sample being classified as one of two possible groups (Cabrera, 1994). By choosing a cutoff value, the probability can be used to obtain a binary classifier. The model applies the logistic function to a linear combination of all the independent variables (features), as depicted in Eq. (3).

$$p(y = 1) = logistic(\beta_0 + \beta_1 x_1 + \cdots + \beta_n x_n) \tag{3}$$

In Eq. (3), β_i are the model parameters to be learned from data, x_i are the independent variables and y is the target or dependent variable. The β_i coefficients can be interpreted as the expected change of having the outcome per unit change in x_i (Peng, Lee, & Ingersoll, 2002). That is, when β_i is greater than zero, larger (or smaller) values of x_i are associated with larger (or smaller) probabilities of $y = 1$ (positive classification). Conversely, if β_i is lower than zero, larger (or smaller) values of x_i are associated with larger (or smaller) probabilities of $y = 0$ (Peng et al., 2002). This interpretation of the β_i coefficients provides us with valuable information to interpret which syntactic constructs positively and negatively influence the expertise of the programmer.

We used L_1 (Lasso) and L_2 (Ridge) regularization penalties (Elastic Net) (Zou & Hastie, 2005) provided by the `LogisticRegression` model implemented by scikit-learn (SciKit-Learn, 2022f). A stratified split of the datasets was performed, using 80% for training and 20% for testing. The best hyperparameters are found with exhaustive parallel search across common parameter values (`GridSearchCV`), using stratified randomized 10-fold cross-validation (`StratifiedShuffleSplit`) against the training set.

4.6. Classification rules

We used two different mechanisms to mine classification rules that provide information about the syntactic constructs. Decision tree classifiers and rule induction were the two machine learning models built for that purpose. Categoric features were encoded as one-hot vectors.

Decision tree learning is a supervised learning technique to predict values from previous observations. Decision trees are used to foresee one target value from the known features of a given sample. When the values to predict are discrete, decision trees act as classifiers; for continuous values, they become regressors.

A good characteristic of decision trees is that they are easy to understand and interpret. That is why they are useful for both machine learning and data mining. For example, the decision tree in Fig. 2 classifies type definitions (classes, interfaces, enums or records). When a type is not defined in the default package, and it has one or more annotations, then the programmer is classified as expert.

Each path from the root node to a leaf (i.e., a tree branch) represents a classification rule. For example, the following rule can be obtained from the decision tree in Fig. 2: *default package* = False AND *number of annotations* $\geq 1 \Rightarrow$ Expert.

Support and confidence are two widespread metrics to measure the performance of classification and association rules. Being $X \Rightarrow Y$ a classification rule with X the antecedent (a list of conjunctions) and Y one value of the binary target variable to be classified (e.g., beginner or expert), its support is defined with Eq. (4) and its confidence with Eq. (5). Support refers to how often a rule appears in the dataset, while confidence refers to the number of times the rule is fulfilled. Our

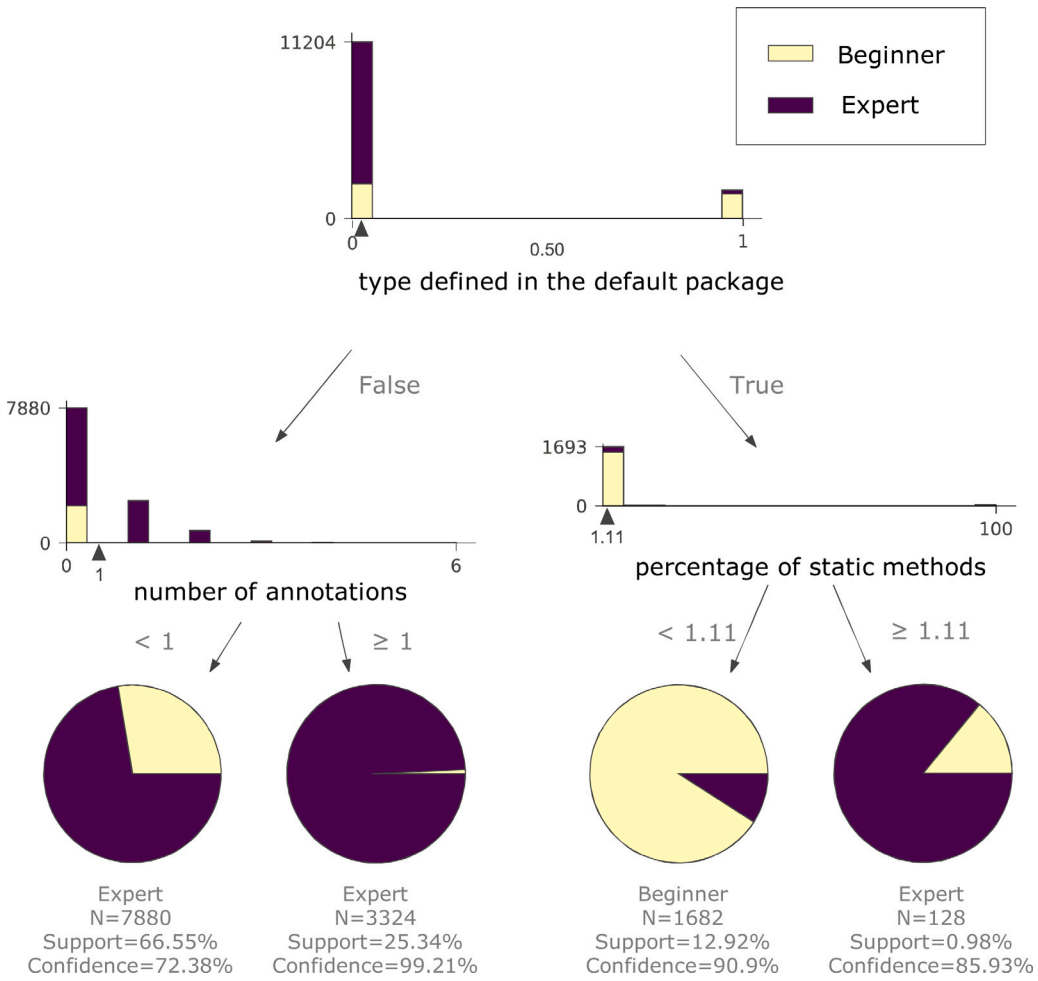

Fig. 2. Example decision tree (max depth 2) for the type definitions dataset.

previous classification rule has 25.34% support and 99.21% confidence.

$$Support(X \Rightarrow Y) = \frac{number\ of\ instances\ containing\ X\ and\ Y}{total\ number\ of\ instances} \quad (4)$$

$$Confidence(X \Rightarrow Y) = \frac{number\ of\ instances\ containing\ X\ and\ Y}{number\ of\ instances\ containing\ X} \quad (5)$$

An important hyperparameter in a decision tree is its maximum depth (the tree in Fig. 2 was created with a maximum depth of 2). Too deep trees tend to overfit, since one branch per sample could be created. Such over-complex models do not generalize well for the training data, and hence they do not produce useful information. For this reason, we created decision trees with maximum depths from 1 to 3 and analyzed the classification trees extracted from them (see Section 5.5). CART was the algorithm used to build the decision trees (Breiman, 1984), implemented by the `DecisionTreeClassifier` class in scikit-learn (SciKit-Learn, 2022c).

The second classification rule mining technique we used was rule induction. Rule induction is a technique in which formal rules are obtained from a set of observations. A classification rule is a collection of propositional predicates associated with a given value of the target feature, similar to the ones obtained from decision trees.

There exist different algorithms for classification rule induction. We used the RIPPERk (Cohen, 1995) and IREP (Fürnkranz & Widmer, 1994) algorithms. The former usually obtains error rates lower than the C4.5 algorithm, scales nearly linear to the number of training instances, and is able to efficiently process noisy datasets (Cohen, 1995). The latter includes an incremental reduced error pruning process that avoids overfitting with noisy data and provides good generalizations (Fürnkranz & Widmer, 1994).

We ran the decision tree, RIPPERk and IREP machine learning algorithms, obtained the classification rules, and analyzed only those with at least 90% confidence, 5% support and three or fewer conditions in the antecedent (too specific rules are ignored). We used the `RIPPER` and `IREP` classes of the `wittgenstein` Python module (Moscovitz, 2022).

4.7. Clustering

We also applied clustering algorithms to detect groups of similar syntactic constructs (clusters). Clustering algorithms are unsupervised machine learning techniques that find similar groups of instances from unlabeled datasets. We automatically gathered the clusters of similar constructs from the different datasets of our study, suppressing the programming expertise variable. The centroids of each cluster were analyzed to document the similar syntactic constructs features of each group (intra-cluster analysis), and what makes each cluster to be different from the rest of the clusters (inter-cluster analysis). We also analyzed whether each cluster is made up of expert or beginners and their support.

Although we executed the k-means (Bock, 2007)—`Kmeans` class in scikit-learn (SciKit-Learn, 2022e)—, DBSCAN (Density-Based Spatial Clustering of Applications with Noise) (Ester, et al., 1996)—`DBSCAN` (SciKit-Learn, 2022b)—and OPTICS (Ordering Points To Identify the Clustering Structure) (Ankerst, Breunig, Kriegel, & Sander, 1999)—`OPTICS` (SciKit-Learn, 2022g)—algorithms, our server (Section 4.8)

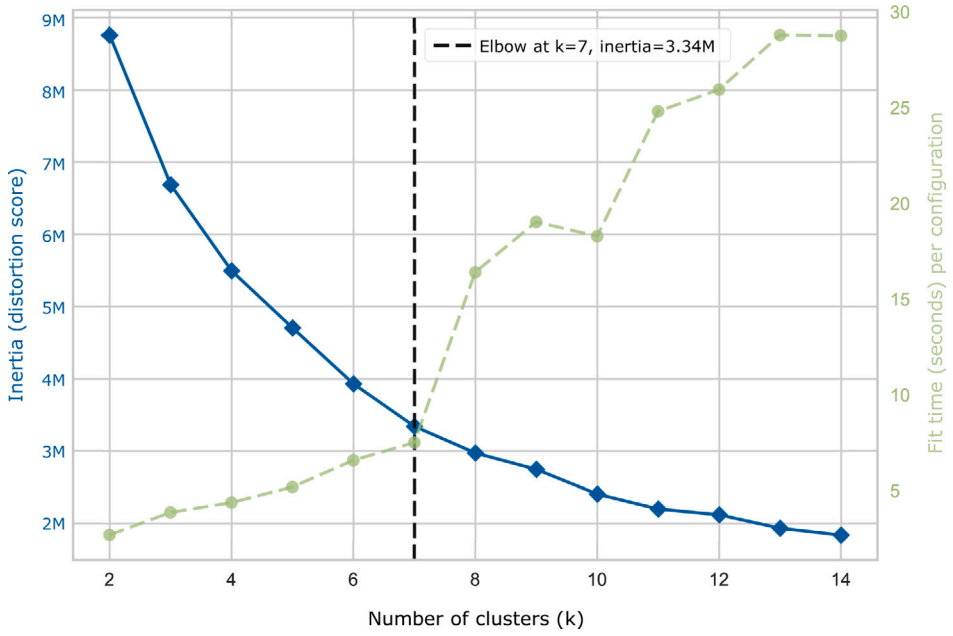

Fig. 3. Visual elbow method to find the optimal number of clusters in k-means for the types dataset.

ran out of memory with all the algorithms but k-means. Centroids were initialized with random values and we used the Euclidean distance.

K-means requires us to pass the number of clusters. Therefore, we ran k-means from 2 to 20 clusters and apply the elbow method (kneedle algorithm) to find the optimal number of clusters (Satopaa, Albrecht, Irwin, & Raghavan, 2011). Fig. 3 shows one example for the types dataset. When the number of clusters grows, the sum of squared distances of samples to their closest cluster center (called inertia) increases. Since we look for a tradeoff between the number of clusters and inertia (when the number of clusters is the same as the number of samples, we have very low inertia but clusters are useless), the elbow of the graph visually represents such a tradeoff. The number of clusters found for each dataset is depicted in Table 3.

Once the optimal number of clusters is found, their centroids were computed as the average values of their instances. For numeric variables, we also computed the 95% confidence intervals. ANOVA analyses were run for the numeric features of each dataset to see if there were significant differences among the clusters (inter-cluster analysis). When those differences exist (p-value<0.05), we used Tukey's Honest Significant Difference (HSD) post-hoc tests to see exactly where those differences lie (i.e., to find out which specific variables are different from one another) (Abdi & Williams, 2010).

4.8. Experimental environment

All the experiments were executed in an AMD Ryzen Threadripper 2990WX (64 cores) with 128 GB DDR4 3200 MHz RAM, Nvidia Quadro RTX 4000 (8 GB GDDR6), running CentOS operating system 7.4–1708 for 64 bits. All the code was implemented using Python 3.8.2, scikit-learn 1.0.2 and wittgenstein 0.3.2. Datasets were stored in a PostgreSQL 10.15 database.

5. Results and discussions

We ran the algorithms described in Section 4 to retrieve information about the syntactic constructs used by Java programmers. In this section, we analyze and discuss the results. All the data is available for download at Ortin, Facundo, and Garcia (2022).

5.1. Anomaly detection

Univariate and multivariate outliers were detected with the statistical methodology described in Section 4.2. After the analysis of the anomalous syntactic constructs, we found out that seven program instances (together with their types, methods, fields, statements and expressions) do not represent valid Java projects:

(a) Five of the beginner projects were made up of just a collection of interfaces, with no classes. That is because one of the student assignments consisted in implementing some given interfaces. Some students just delivered the original collection of interfaces, without any class, enum or record implementation.
(b) The multivariate outlier detection algorithm identified as anomalous two programs with the combination of two values: number of classes lower than 34% of the types defined in the program, and number of interfaces greater than 65%. This is a similar case scenario as in (a); students who delivered incomplete Java programs.

The two previous groups of outliers (10,901 AST nodes of 7 programs) were deleted from the datasets, since they do not represent proper Java projects. Thus, we did not consider any of the AST nodes belonging to those programs.

The rest of the anomalies denote unusual syntactic constructs used by Java programmers. What follows is a summary of the information inferred from the results:

- Expressions. The anomalous syntactic constructs are multiple assignments (*expression* = *expression* = *expression*) as expressions (not as statements), the prefix and postfix `--` operator, bitwise operators (|, &, ^ and ~) and *Type*::*member* references.
- Statements. Prefix decrement (`--`*expression*`;`) is the only statement construct detected as anomalous. Regarding its height and depth inside their method definitions, outliers are those with values greater than, respectively, 6 and 9.
- Types. The only anomalous type is the union type included in Java 7+ to catch multiple exception types (`catch(`*Exception*$_1$`|` *Exception*$_2$ *id*`)`).

- Method definitions: `native` and `synchronized` methods are outliers, and so are those written in capitals. Default implementations of interfaces (Java 8+) are barely used, and only by experts. A method is anomalous if it has more than 17 statements, 5 parameters or local variables, 4 annotations, or it is overloaded more than once.
- Field definitions: `volatile`, `transient` and field annotations are outliers, and they are never used by beginners.
- Type definitions: no class was defined as `strictfp`. Nested classes and `static` blocks are barely used, only by experts. Classes are anomalous when they have more than 25 methods, 12 fields, 4 annotations or implement more than 4 interfaces.
- Programs: those Java projects that define more than 17.4% of non-class types (enums, interfaces and records) were detected as anomalous.

Our translation method of tree structures to n-dimensional datasets has shown an important benefit in detecting anomaly constructs compared to the rule-based in Qiu et al. (2017). The only construct we detect that is also included in their study as anomalous is union types. They identified labeled statements as highly infrequent, whereas we did not include labels as a statement feature. They also consider the empty statement as anomalous, showing a limitation of their dataset because, as we discuss in Section 5.4 (Table 6), beginners commonly use that syntactic construct.

5.2. Most frequent syntactic constructs

Table 4 shows the most frequent syntactic constructs found with the method described in Section 4.3. For each type of AST node (i.e., *category* feature), we show the first ten most frequent constructs. For the rest of the constructs, we only show those with a percentage greater than 10% and that provide remarkable information.

Table 4 is self-explicative, so we only highlight those results that we think are more remarkable:

- Comparison expressions (`<`, `>`, `<=`, `>=`, `==` and `!=`) are used more than arithmetic ones (`+`, `-`, `*`, `/` and `%`); so is the `null` expression. Variables (identifiers) represent the most common expression.
- Method invocation is the most widespread statement, due to the object-oriented paradigm of the Java programming language. `return` is the third most used statement, and the postfix `++` statement (i.e., `expression++;`) occurs more than the `for` and `while` loops.
- References (objects excluding `String`) are used much more than the rest of the types (`int` and `String` are the next ones). `long` is used more than `char` and `float`. Types appear more as method parameters than as local variable definitions. 9.3% of the types used are generic, but only 1.6% are arrays—programmers use the generic Java collections more than arrays.
- For method definitions, the `native` modifier is used more than `synchronized`. Only 0.21% of the methods are generic and the most common return type is `void`, reducing the benefits of functional programming. 84.1% of the methods are `public`, which seems to go against the "minimize the accessibility of classes and members" recommendation (Bloch, 2008).
- Fields are mostly defined as `private` to obtain the benefits of information hiding. `final` and `static` modifiers are widely used—46.7% and 24.6%, respectively. The majority of fields are not initialized, and generic types are used 13.4% of the time (more than `boolean`). The common `public static final` constant definition pattern represents 21% of all the field definitions.
- Only 3.4% of the types defined are interfaces and 2.6% are enumerations. The "minimize the accessibility of classes and members" good practice to reduce coupling does not appear to be followed in type definitions (84.8% are `public`). Other remarkable results are that 38.1% of the types have any annotation, 16.2% of them are defined within another class, and 15.5% are implemented in the default package—those types cannot be imported from another project.
- Most of the projects do not define any `interface` or `enum`, and almost half of them define one or more types in the default package.

The method described in Section 4.3 generates more data than the results summarized in Table 4. It also analyzes multiple combinations of different features. Some of those combinations produced interesting results, although their frequencies tend to be low. What follows are some of those results:

- In statements, the combination of the type of statement and its first child shows that `return` *boolean_literal* (e.g., `return true;`) is quite common (4.4%). In fact, the frequency of `return` *boolean_literal* is very close to the most common return statement: `return` *variable* (5%). `return true` and `return false` constructs represent 29.6% of all the `return` statements—used by both experts and beginners.
- Arrays are barely used (1.6%). Arrays of references are the most common ones, and then comes the arrays of strings, 85.9% more common than the array of integers.
- For field definitions, almost every `public static final` field is defined with identifiers written in capital letters. Only 5,696 instances (out of 240 K) do not follow that naming convention. Surprisingly, just 11.7% of those instances were code written by students, who showed better fulfillment of the all-uppercase Java naming convention of constants (Oracle, 2022a).
- 10.5% of the constructors are `private`. Most of the occurrences appear when implementing the Singleton design pattern (Gamma et al., 1994).
- Methods are overloaded 13.7% of the time, while 44.2% is the percentage of overloaded constructors. Thus, method overloading seems to be more appropriate to provide different ways to initialize objects' states.

Our AST-based system shows distinct benefits compared to the analysis of Qiu et al. based on grammar rules (Qiu et al., 2017). First, they cannot analyze the syntactic constructs that involve more than one feature of AST nodes, such as the examples in the previous enumeration. Second, they do not manage to analyze those constructs that are not represented in a single rule, such as literal and `null` expressions, and method invocation, assignment and `catch` statements. Moreover, they are not able to analyze aggregated values such as the percentage of interfaces defined in a project or the number of types in the default package.

The most frequent idioms retrieved by Allamanis and Sutton also show differences from our analysis. First, they do not indicate any measure of frequency—they just show the 19 top idioms. Second, the idioms they infer have too specific values, such as particular names of classes, methods and even variables. Such idioms show common source code fragments instead of syntactic constructs. Moreover, they include neither aggregated values nor features of AST nodes. Some idioms manage to combine different types of statements such as `if` and `try/catch` because they do not generalize all the syntactic constructs involved in that idiom (i.e., they use specific values of the source code).

5.3. Data visualization

We reduced to two dimensions the 12 datasets with the 4 algorithms described in Section 4.4. To have an estimate of how much information is lost in the dimensionality reduction, we computed the explained variances for the PCA algorithm. The explained variance is defined as the ratio of the two principal component eigenvalues to the total

Table 4
Most frequent syntactic constructs found in Java source code.

Dataset	Construct	Most frequent values
Statements	AST node type	Method invocation (27.5%), variable definition (26%), `return` (14.7%), `if` (12.3%), assignment (9.7%), `throw` (2.1%), `try` (1.7%), `catch` (1.5%), postfix `++` statement (0.9%), `for` (0.8%)
	Parent AST node	Method definition (67.2%), `if` (17%)
Type definitions	AST node type	`class` (94%), `interface` (3.4%), `enum` (2.6%)
	Visibility	`public` (84.8%), package (15.2%)
	Modifiers	`final` (1.5%), `abstract` (2.2%), `strictfp` (0%), `static` (3.6%)
	Extends	24.4% of the types extend another class
	Implements	30.5% of the types implement one or more interfaces
	Annotations	38.1% of the types have annotations
	Default package	15.5% of the types are defined in the default package
	Generics	1.3% of the types are generic
	Nested classes	16.2% of the types are nested classes
	`static` block	0.37% of the types include a `static` block
Field definitions	Visibility	`private` (71.2%), `public` (23.1%), package (4.2%), `protected` (1.5%)
	Modifiers	`final` (46.7%), `static` (45.6%), `volatile` (0.07%), `transient` (0.15%)
	Initial value	No initial value (53%), `int` literal (13.7%), constructor invocation (10.8), method invocation (10.4%)
	Type	Reference type (33.4%), `int` (20.2%), `String` (13.4%), generic type (13.4%), `boolean` (6.2%)
	Annotations	99.1% of the fields have no annotations
	Visibility + `static` + `final`	`private` (43.7%), `public static final` (21%), `private static final` (18.8%)
Expressions	AST node type	Variable (36.4%), member selection (19.17%), method invocation (13.1%), `String` literal (5.2%), `int` literal (4.8%), comparison expression (4.5%), `null` (3%), `new` (2.7%), arithmetic expression (2.7%), `boolean` literal (2.4%)
Method definitions	Visibility	`public` (84.1%), `private` (8.4%), package (5.5%), `protected` (2%)
	Modifiers	`final` (0.97%), `static` (4.9%), `abstract` (5.5%), `strictfp` (0%), `native` (0.61%), `synchronized` (0.1%), `@Override` (20.4%)
	Generic methods	0.21% of the methods are generic
	`throws` clause	5.3% of the methods have `throws`
	Return type	`void` (37%), reference type (23.2%), `String` (10.2%)
	Parameters	48.4% of the methods have no parameters
	Annotations	68.3% of the methods have no annotations
	Constructor	8% of the methods are constructors
	Method overloading	12.6% of the methods are overloaded
Types	AST node type	Reference type (64.3%), `int` (14.1%), `String` (13.9%), `boolean` (4.2%), `double` (1.4%), `long` (0.7%), `char` (0.6%), `byte` (0.4%), `float` (0.4%), `short` (0.1%)
	Built-in types	21.2% of the types are built-in (`String` is not considered built-in)
	Parent AST node	Method parameter (24.2%), local variable definition (22.2%), method return type (19.6%), constructor invocation (17.1%), field definition (12.8%)
	Generic types	9.3% of the types used are generic
	Array types	1.6% of the types used are arrays
Programs	Class percentage	72.4% of the projects only contain classes
	Interface percentage	78.2% of the projects do not define any interfaces
	`enum` percentage	86.2% of the projects do not define any enumerations
	Default package	49% of the projects define types in the default package

eigenvalues. It represents the information explained by the two features obtained after applying PCA.

Fig. 4 presents the two-dimensional visualizations of the programs dataset after applying the four dimensionality reduction algorithms. The programs dataset is the one that showed the highest explained variance (99.75%)—i.e., the one with the highest information kept after reducing the dimensions. This dataset is not balanced (Table 2) and most samples represent beginner programs. For Linear PCA and NMF, no clusters with sufficient points can be clearly identified; experts and beginners are not separated either. Clustering and program separation by expertise is better performed by t-SNE. Kernel PCA is able to separate many experts, but clusters cannot be clearly identified.

Performing the analysis described in the previous paragraph with the 12 datasets, we detected that t-SNE is the visualization that provides the best clustering and program separation by expertise—all the figures are available at Ortin et al. (2022). Fig. 5 shows the t-SNE visualization of all the datasets but programs. Field definitions, types and type definitions are the datasets with the clearest identification of clusters with close syntactic constructs (Euclidean distance). They also appear to be the best ones at separating experts from beginners. Heterogeneous datasets do not identify any cluster, but there are regions with many points belonging to the same level of expertise.

The visualization of homogeneous and heterogeneous syntactic constructs provides us with more information than the n-TED AST-similarity measure used by Yin et al. (2015). n-TED has only shown

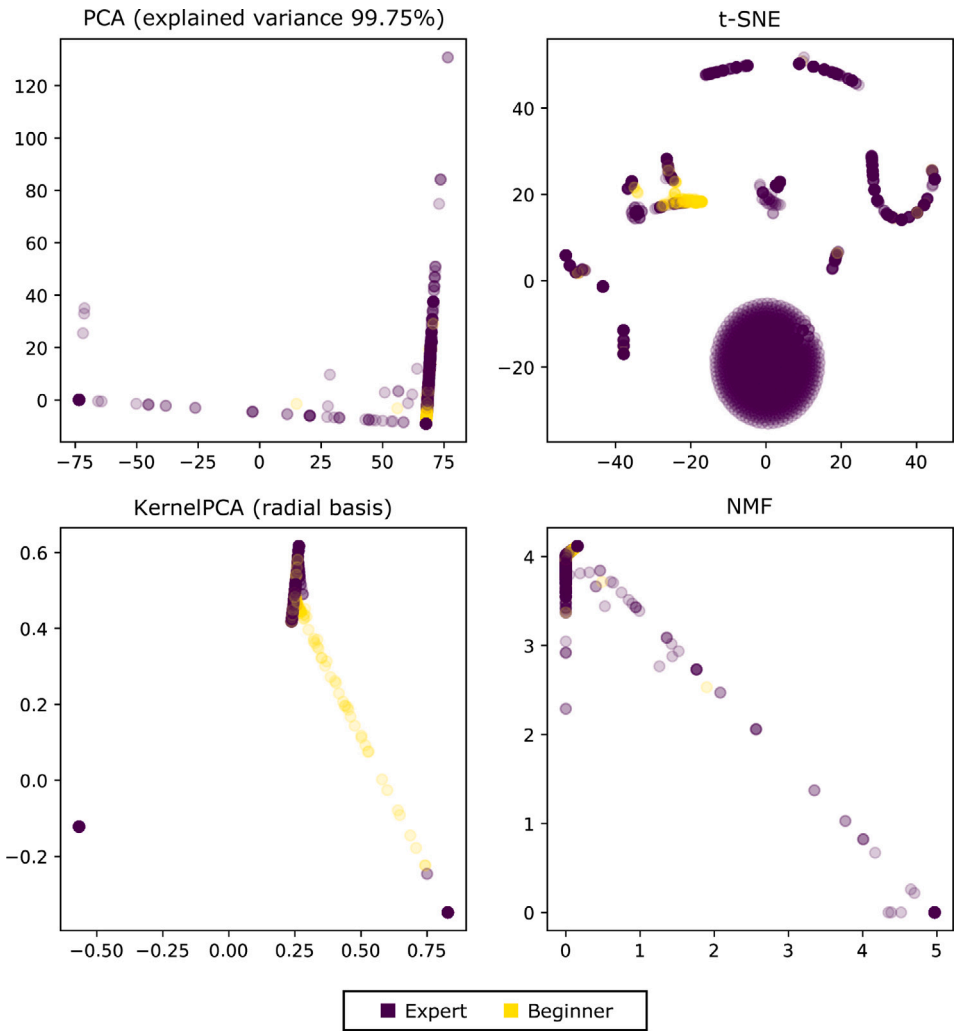

Fig. 4. Visualization of the four dimensionality reduction algorithms used for the programs dataset.

benefits to detect clusters of small code fragments, namely different implementations of the same function (Choudhury et al., 2016). On the contrary, we visualize groups of seven similar types of syntactic constructs and five different combinations of bigger ASTs.

5.4. Logistic regression

We built as many logistic regression models as datasets. Table 5 shows the classification performance of each model. We can see how the types written by a programmer are the syntactic constructs that hold lesser information about their expertise. The second worst classifier is method definition. Notice that the method body is not included in the dataset; when it is included (heterogeneous 3), the F_1-score obtained is 98%. The rest of the classifiers have a minimum F_1-score of 90.1%. Heterogeneous datasets have the highest performance because they aggregate more features (programs is an exception, because it is a highly unbalanced dataset).

Table 5 also shows the accuracies of the models built by Ortin et al. to classify syntactic constructs (discussed in Section 2). For all the homogeneous datasets but method and type definitions, our logistic regression models provide higher accuracy. The same occurs for the first three heterogeneous datasets, while we provide lower accuracy for the two last ones. While there are no important differences in the accuracy of both approaches, the logistic regression models provide much more interpretability than the complex classifiers obtained by Ortin et al. (see Section 2).

Table 5
Performance of the 12 logistic regression models. The column "Accuracy (Ortin et al., 2020)" shows the performance of similar classifiers in the related work of Ortin et al. discussed in Section 2.

	Dataset	Accuracy	Accuracy (Ortin et al., 2020)	Precision	F_1-score
Expressions	0.8593	0.9484	0.7812	0.8841	0.9152
Statements	0.8393	0.9565	0.8101	0.8595	0.9054
Types	0.7182	0.8528	–	0.7455	0.7956
Method definitions	0.8184	0.9837	0.8499	0.8223	0.8958
Field definitions	0.8427	0.9242	0.8344	0.8824	0.9028
Type definitions	0.9032	0.9282	0.9620	0.9353	0.9317
Programs	0.9710	0.9710	0.9401	0.9710	0.9710
Heterogeneous 1	0.9504	0.9531	0.8830	0.9678	0.9604
Heterogeneous 2	0.9763	0.9782	0.9136	0.9827	0.9805
Heterogeneous 3	0.9759	0.9769	0.9519	0.9831	0.9800
Heterogeneous 4	0.9823	0.9833	0.9945	0.9877	0.9855
Heterogeneous 5	0.9846	0.9851	0.9958	0.9894	0.9872

As mentioned in the methodology, the β_i coefficients of the logistic regression models provide us with information about the influence of the features. Higher β_i values are associated with higher probability of being an expert programmer. Zero β_i coefficients represent no influence on programming expertise. Lower negative β_i values represent higher probability of beginner.

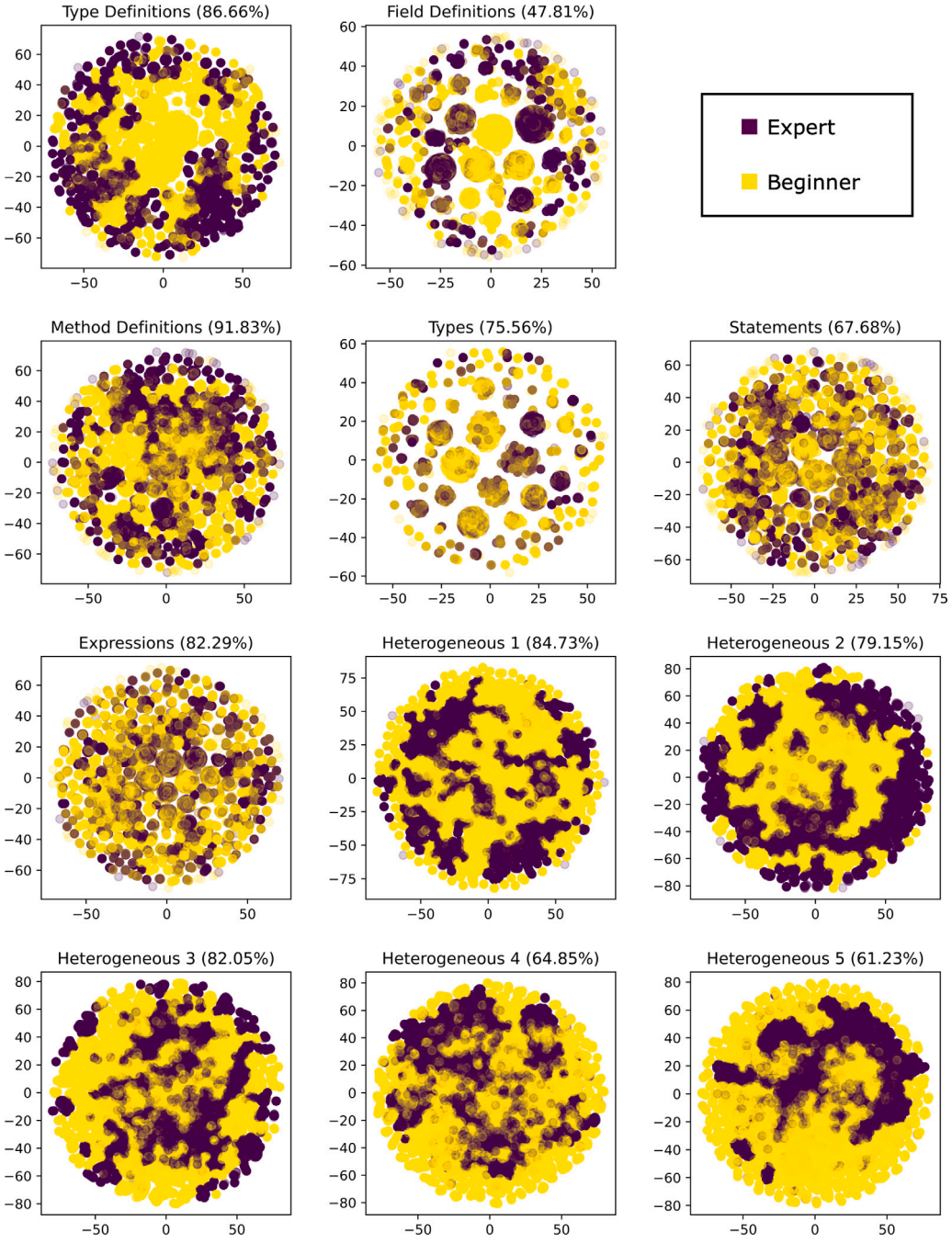

Fig. 5. Visualization of the remaining datasets after t-SNE dimensionality reduction. Values between parenthesis represent the explained variance after PCA.

Table 6 shows the highest and lowest β_i coefficients for all the datasets (those greater than 1 and lower than −1). We do not include information that is redundant (e.g., complementary values) or has been previously discussed in the outliers analysis (Section 5.1). The β_i coefficients of the heterogeneous models did not add new information not previously inferred from the homogeneous ones.

As the depth and height of the ASTs representing an expression grow, it increases the probability to be written by an expert. `float` and `long` literals (e.g., `2.3F` and `3L`), `new` array initializers (`new int[]{1,2,3}`) and assignments as expressions are much more probable for experts. The use of `instanceof` is most common in beginners, sometimes denoting the wrong use of types and the absence of polymorphism. They also use boolean literals as child nodes of comparison expressions, where they are actually not necessary (e.g., `boolVariable==true`).

As in expressions, higher height and depth of statements ASTs mean higher probability of expert programmer. The `assert`, `synchronized` and `continue` statements and statements inside lambda expressions are also more common in code written by experienced programmers. Empty statements (i.e., just `;`) and constructor invocations as statements (e.g., `new MyClass();`) are more likely written by beginners. The former is because they write more ; than necessary. The latter is because they use constructors as methods, not just as object state initializers.

When defining methods, experts are more likely to write more inner classes, exceptions in `throws` clauses, annotations and generic

Table 6
Most significative β coefficients of the logistic regression models.

Dataset	Feature	β
Expressions	Height inside the whole expression	10.97
	Depth inside the whole expression	2.35
	`float` literal	2.30
	`long` literal	2.22
	`new` array initializer	1.30
	Use assignments as expressions	1.17
	`instanceof` expression	−1.75
	Use boolean literals as child nodes of comparison expressions	−2.34
Statements	Height of the statement inside the method body	15.05
	`assert` statement	3.50
	`synchronized` statement	2.86
	Statement inside a lambda expression	2.38
	`continue` statement	1.67
	Constructor invocation as a statement	−1.04
	Use `;` as an empty statement	−1.35
Methods	Number of inner classes	8.71
	Number of exceptions in `throws`	6.25
	`synchronized` method	4.07
	Number of annotations	3.85
	`final` method declaration	3.39
	Number of generic variables	3.34
	`native` method declaration	2.70
	`public` method declaration	−1.08
	Number of local variables	−1.13
Types	Height of the type inside the whole type	90.58
	Number of generic variables	6.35
	`short` type	5.56
	`long` type	2.34
	`byte` type	1.08
Fields	Number of annotations	2.28
	`long` field type	2.22
	`short` field type	1.38
	The field is initialized with a method invocation	1.14
	`final` field declaration	1.14
	Field name with lowercase letters (no camel-case)	−1.09
Type defs.	Number of annotations	12.47
	Number of interfaces implemented by a class	6.48
	Number of fields	5.40
	Number of `static` nested types	3.87
	Number of methods	3.67
	`final` class declaration	3.58
	Type named with all capitals	−1.33
	`public` types	−2.24
	Type name camel-cased, first char lowercased	−2.46
	Number of overloaded constructors	−3.61
Programs	Percentage of enums	8.74
	Percentage of types defined in the default package	3.50
	Percentage of interfaces	−9.79

variables. They also use more `synchronized`, `final` and `native` method qualifiers. On the contrary, a higher number of local variables are associated with beginners—instead of using bigger expressions, they write simpler ones and store their values in local variables. Students also declare methods as `public` when it is not necessary.

For types, all the significant β_i coefficients are associated with experts: the height and depth of the type structure, generic variables, and the use of `short`, `long` and `byte` types (they are barely used by students). In field definitions, the number of annotations and generic variables, `long` and `short` types, the `final` field qualifier, and field initializations with method invocation (e.g., `int field = obj.method();`) are more likely written by experts. Beginners are more likely to write no camel-case field names, all lowercased.

When defining a new type (method, class or enum) the number of annotations, interfaces implemented by a class, fields, `static` nested types and methods increase the probability of being an expert. The same occurs when the class is declared `final`. On the other hand, beginners are more likely to write `public` types (they do not hide types outside the package) and choose type names not starting with a capital letter or with all letters capitals. If the class has many implementations of constructors, it is more likely it was written by a beginner (they create unnecessary constructor implementations).

For programs (Java projects), experts use more enumerations. They are also more likely to write types in the default package. After speaking to the lecturers of the year-1 programming courses used to create the dataset, they told us that they forbid the use of the default package. Therefore, the percentage of types in the default package feature is not actually associated with the programming expertise. The same occurs with the percentage of interfaces: as we mention, some assignments consist in implementing various interfaces.

5.5. Classification rules

We ran the three supervised machine learning algorithms described in Section 4.6 to mine classification rules, obtaining 2521 rules (164 inferred with decision trees, 1921 with RIPPERk and 573 with IREP[1]). To filter too specific rules with lower information, we only consider those with minimum support of 5% and 3 or fewer conditions in the antecedent. Likewise, the minimum confidence is 95%. We do not include in the analysis those rules that only combine the features already extracted from the logistic regression models (Table 6).

A rule r_1 is more general than r_2 when all the instances fulfilling the antecedent of r_2 are also fulfilled by the antecedent of r_1 (e.g., $a \Rightarrow c$ is more general than a AND $b \Rightarrow c$). When we find two rules, r_1 and r_2, that fulfill the conditions in the previous paragraph, and r_1 is more general than r_2, then we only consider r_1 since it provides more general information (higher support with lower but sufficient confidence).

The previous filtering process is aimed at discussing only the new relevant information, not discussed before. All the rules and the source code used to produce them are available for download at Ortin et al. (2022).

Table 7 shows the classification rules obtained, which are self-explanatory. All the rules taken from the homogeneous datasets (1 to 9) represent syntactic constructs written by experts. As with the logistic regression models, there are more syntactic patterns for experts than for novice programmers—most of the constructs used by beginners are also used by experts.

The classification rules mined from the heterogeneous datasets (10 to 13) combine information from different syntactic constructs. For example, rule 11 classifies as beginner the code that has no `final` fields, declared in class with no annotations, and defined in a program that contains no type in the default package.

There are notable differences between the rules mined with our approach and the one described in Losada et al. (2022). The association rule mining approach followed by Losada et al. retrieves a huge number

[1] The sum of the number of rules extracted with the three methods (2658) is not the same as the total number of rules (2521) because some of the rules are repeated.

Table 7
Classification rules obtained after the filtering process. Support values are computed considering the number of instances of the dataset the rule was extracted from. DT stands for Decision Tree.

	Rule			Support	Confidence	Algorithm
1	Field named with snake-case convention	⇒	Expert	8.63%	92.17%	IREP
2	Field is `final` AND not `public` AND not `static`	⇒	Expert	8.63%	92.17%	RIPPER*k*
3	Method named with camel-case convention AND number of parameters ≥ 3	⇒	Expert	28.75%	93.33%	DT
4	First parameter of a method is a reference (arrays and strings are not considered as references)	⇒	Expert	31.75%	91.11%	IREP
5	Return type of a method is a reference	⇒	Expert	8.99%	91.01%	IREP
6	Type not defined in the default package AND has 1 or more annotations	⇒	Expert	25.54%	99.22%	DT
7	Percentage of `static` fields in a type ≤ 20% AND that type is not `public`	⇒	Expert	12.01%	95.46%	IREP
8	Type defined in the default package AND number of constructors ≤ 1 AND it is `final`	⇒	Expert	5.90%	99.87%	RIPPER*k*
9	Type defined with one or more annotations AND has `extends`	⇒	Expert	9.76%	99.56%	RIPPER*k*
10	Type defined with no annotations AND its program has less than 0.18% of enums	⇒	Beginner	24.82%	98.8%	DT, IREP
11	Field is not `final` AND it is defined in a class with no annotations AND the program does not define any type in the default package	⇒	Beginner	21.39%	90.45%	IREP
12	Type defined with no annotations AND no snake-case name AND its program has less than 0.18% of enums	⇒	Beginner	33.79%	99.53%	DT
13	Method returning a reference type AND written in a program with more than 1.94% of enums	⇒	Expert	5.15%	99.20%	IREP

of rules with obvious information (e.g., if the percentage of classes in a program is 100%, the percentage of enumerations is 0%) (Losada et al., 2022). Since association rules require all the features to be binary, the numeric features must be discretized, producing highly multidimensional datasets. Such a huge number of features involves association rules with too many conditions in the consequent, hard to understand. Another drawback of the highly sparse data is the low support of the extracted rules. The result is that the rules mined with our approach are easier to understand, provide more valuable information and have higher support. In fact, none of the rules in Table 7 are detected by Losada et al. using the same Java projects.

5.6. Clustering

We run k-means against the 12 datasets, following the method described in Section 4.7. The kneedle algorithm found the number of clusters depicted in Table 3. ANOVA and Tukey's HSD tests were computed to see if there were statistical differences among clusters for the values of each variable. We show an example in Fig. 6, where mean values and 95% confidence intervals of some features are depicted for the 6 clusters found in the type definitions dataset. This is the information inferred from the analysis of the clusters retrieved from the syntactic constructs of type definitions (Fig. 6):

- Cluster 3 in Fig. 6 represents a low number of instances (1.51%) with the highest number of methods, constructors, nested types and inner types. It represents classes with high complexity, which would probably be better redesigned.
- Cluster 4, with less than 2% instances, holds simple classes with the lowest number of annotations, implemented interfaces and fields, but with the highest number of `static` methods. Helper and utility classes are included in Cluster 4.
- Cluster 6 represents classes holding configuration data (the lowest number of methods and the highest number of `static` fields).
- Cluster 5 may be characterizing classes to instantiate Data Transfer Objects (DTOs), because they hold the greatest number of fields.
- Types with the highest number of annotations belong to Cluster 2, representing code written by experts (98.61% of its instances).
- Finally, Cluster 1 is the group with the most instances (62.63%). It defines types with a low number of `static` members, and an average number of methods, constructors, fields, annotations, and nested and inner classes. It groups the collection of common classes.

Similar analyses were undertaken for all the clusters obtained (Ortin et al., 2022). We summarize here those that imply new information, not discussed before:

- Fields initialized with reference values (not including strings or arrays) and with the highest number of annotations are clustered together. 90% of their instances are expert syntactic constructs.
- `private`, `final`, `static` fields with uppercased names comprise a cluster of 96% expert samples.
- Most fields (53.13%) are not `final`, not initialized and camel-cased, written by both experts and beginners.
- One cluster of method definitions represents only 0.05% of the methods with too dense bodies: the highest number of local variables and statements.
- The clustering algorithm groups those expressions with the greatest height and weight in the same cluster, with 100% of their instances written by experts.
- AST representing types with the greatest height and not representing a primitive type are grouped together (all of them coded by experts).
- One heterogeneous cluster represents programs with the highest percentage of classes and fields per class, probably representing data management applications.
- The programs composed of a high number of enumerations and types with multiple annotations, written by experts (98.61%), are grouped together by the clustering algorithm.
- A cluster is found for programs that require the highest number of enums and interface types, interfaces implemented by classes, and methods, fields, inner and nested classes per type definition. This group represents complex programs and 100% of the instances were written by experts. Similar clusters are found in multiple heterogeneous datasets.

Out of the 66 clusters found by k-means, 21 and 5 of them hold more than 90% syntactic constructs written by, respectively, experts and beginners. Thus, programming expertise seems to be a characteristic intrinsic in many syntactic constructs, since 39.4% of the clusters created by the unsupervised machine learning algorithm have the same level of expertise. Likewise, experts write more particular syntactic constructs than novice programmers. As discussed before, this is because most of the constructs used by beginners are also used by experts.

An important benefit of our method compared to the clustering approach proposed by Choudhury et al. is the easier interpretation of the

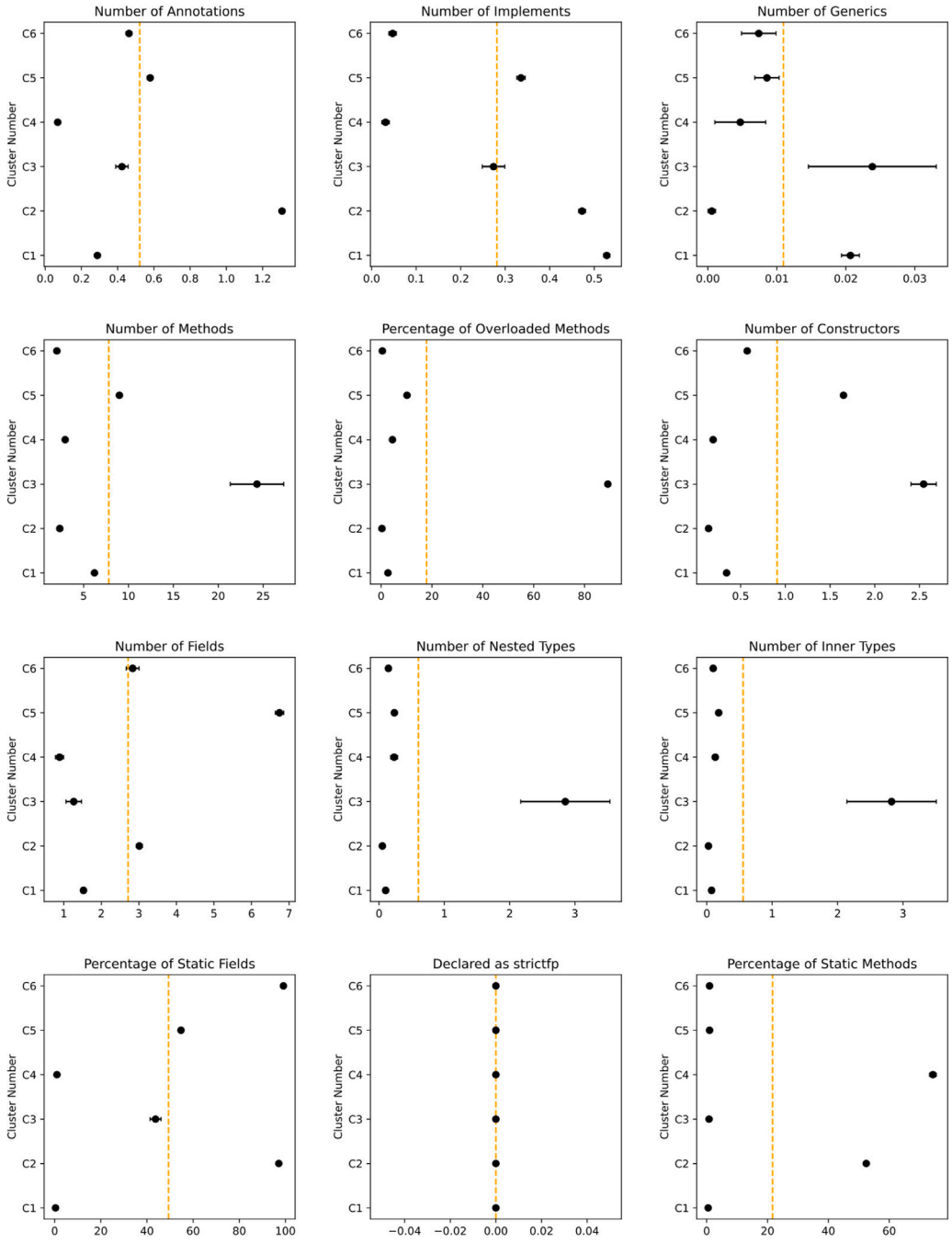

Fig. 6. Distribution of feature values per cluster for the type definitions dataset (inter-cluster analysis). The dashed line represents the average value for the whole dataset. Dots describe the average value for one cluster, and whiskers the 95% confidence intervals. C_n denotes Cluster n.

retrieved clusters. Their clustering system is based on n-TED, a numeric measure of similarity between ASTs (Choudhury et al., 2016). After finding the clusters, they have to manually analyze the ASTs in each cluster to find their common patterns (and also the differences among clusters). In our case, that process is automatically done by running ANOVA and Tukey's HSD tests. Unlike n-TED, our feature engineering method to translate ASTs into datasets is highly interpretable, since the features (columns) in the datasets represent characteristics of the syntactic constructs. Another limitation of the proposal of Choudhury et al. is that it has only been proven to be beneficial in short pieces of code implementing the same problem (Section 2).

6. Discussion

We have shown a mechanism to analyze the syntactic constructs of Java code. An important discussion is to what extent our proposal could be applied to other programming languages. To that aim, we analyze the five steps of our method and how they depend on a particular programming language.

We define a feature engineering system to transform tree structures into n-dimensional datasets. Although it is a manual process, the following two procedures can be followed to apply our method to any programming language:

1. Identification of the homogeneous syntactic constructs. The different groups of homogeneous syntactic constructs in a programming language should be pinpointed. Examples of such constructs for most programming languages are programs, type definitions, module definitions and expressions—some pure functional languages do not provide statements. A dataset should be created for each homogeneous construct.
2. Feature definition. The features of each of the previous homogeneous constructs (dataset) should be defined (Table 1 and Tables A.1 to A.6 show how we did it for Java). When there are different types of AST nodes in a homogeneous construct (e.g., different types of statements), a *category* feature should be defined: it is a nominal feature whose values are the names of the different AST nodes (e.g., `IfStatement` and `WhileStatement`). The parent's and children's categories (AST node names) should also be included as features. Finally, we should add one feature per each common element in the homogenous construct. For example, the `static` and `final` binary features are included in method (Table A.3) and field definitions (Table A.4).

Once the homogeneous datasets are generated, the remaining three steps of our method are language agnostic:

3. The heterogeneous datasets are created by applying drill-down operations to the homogeneous datasets (Section 3.1).
4. After creating the homogeneous and heterogeneous datasets, the methodology in Section 4 describes how to use different machine learning algorithms to mine the datasets.
5. The results produced by the algorithms of the previous step are interpreted as described in Section 5.

The previous steps could be applied to any programming language. Step 2 is the one that has stronger dependencies on the programming language to be analyzed. However, it is a straightforward process if we have access to the AST structures used to represent programs (e.g., the Python's `ast` module included in its standard library (Python, 2022)).

7. Conclusions

The syntactic tree-structured information of a programming language can be analyzed with supervised and unsupervised machine learning algorithms to infer valuable information about how programmers use that language. We define a feature engineering process to translate ASTs into a collection of homogeneous and heterogeneous n-dimensional datasets. Then, interpretable machine learning algorithms are run to analyze the syntactic constructs.

By applying this method to the Java programming language, we document the syntactic constructs mostly (and barely) used by programmers. We also rank the most important syntactic features used by expert and novice programmers. Classification rules allow combining distinct features of heterogeneous syntactic constructs to document the intrinsic expertise of such constructs. We also visualize in two dimensions the structure of 12 datasets with more than 17 million instances and analyze different clusters of programs with similar syntactic constructs. With such analyses, we realized that some groups of constructs represent different kinds of applications and, occasionally, an inherent level of complexity.

This article summarizes the information extracted with the selected machine learning algorithms. All the results, source code and the datasets created in our research work are freely available for download at https://www.reflection.uniovi.es/bigcode/download/2022/java-patterns

8. Future work

In this work, we analyze syntactic constructs of source code. However, programs also enclose semantic information commonly represented with graphs, such as control flow graphs and program, call and class dependency graphs (Rodriguez-Prieto et al., 2020). We plan to enrich our datasets with semantic information to be mined. The classifiers could also be improved by adding semantic data. To do that, we will extend our compiler plugin so that it creates semantic representations. The feature engineering process should also be adapted to deal with cycles in graphs.

Graph neural networks have been used in many scenarios where models are trained with graph data (Wu, Cui, Pei, Zhao, & Song, 2022). In fact, GNNs have already been used to create predictive models from source code (Allamanis, 2022). However, such predictive models act as black boxes to classify programs, being hard to interpret. In the last few years, there have been many efforts to explain the prediction mechanisms of these GNNs with tools such as GNNExplainer, XGNN and PGExplainer (Li, Zhou, Verma, & Chen, 2022). An interesting work would be to create GNN models with syntactic and semantic information, and analyze those models with the existing explainability tools. The results could be compared with the ones presented in this paper.

This article proposes a collection of analyses for syntactic constructs and applies those analyses to the Java programming language. Section 6 discusses how those analyzes could be applied to other programming languages. In the future, we would like to conduct that study with other languages of different paradigms, and compare them with Java.

CRediT authorship contribution statement

Francisco Ortin: Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Resources, Data curation, Writing – original draft, Writing – review & editing, Visualization, Supervision, Project administration, Funding acquisition. **Guillermo Facundo:** Software, Investigation, Resources, Data curation, Writing – review & editing,Visualization. **Miguel Garcia:** Software, Validation, Investigation, Resources, Data curation, Writing – review & editing, Project administration, Funding acquisition.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Francisco Ortin reports financial support was provided by Spain Ministry of Science and Innovation.

Data availability

All the results, source code and the datasets created in our research work are freely available for download at https://www.reflection.uniovi.es/bigcode/download/2022/java-patterns

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, Spain through its support to official research groups (GR-2011-0040).

Appendix. Structure of the homogeneous datasets

Table 1 defined the structure (features) of the expressions dataset. In this appendix, we include the same information for statements (Table A.1), types (Table A.2), method (Table A.3), field (Table A.4) and type definitions (Table A.5), and programs (Table A.6).

Table A.1
Features of the statements dataset.

Name	Type	Description
Category	Nominal	Syntactic category of the current node (e.g., If, Return and Throw).
1st, 2nd and 3rd child	Nominal	Syntactic category of the corresponding child node.
Parent node	Nominal	Syntactic category of the parent node.
Role	Nominal	Role played by the current node in the structure of its parent node.
Height	Integer	Distance (number of edges) from the current node to the root node in the enclosing type (class, interface or enumeration).
Depth	Integer	Maximum distance (number of edges) of the longest path from the current node to a leaf node.
Expertise	Nominal	Beginner or Expert.

Table A.2
Features of the types dataset.

Name	Type	Description
Category	Nominal	Syntactic category of the current node (e.g., Int, String and Reference).
Primitive	Binary	Whether the type is primitive (built-in).
Parent node	Nominal	Syntactic category of the parent node.
Role	Nominal	Role played by the current node in the structure of its parent node.
Height	Integer	Distance (number of edges) from the current node to the root node in the enclosing type (class, interface or enumeration).
Number of generics	Integer	The number of type variables in the generic type (0 = non-generic).
Number of dimensions	Integer	The number of dimensions in an array type (0 = non-array).
Expertise	Nominal	Beginner or Expert.

Table A.3
Features of the method definitions dataset.

Name	Type	Description
Visibility	Nominal	Public, protected, package or private.
Abstract, static, final, strictfp, native and synchronized	Binary	Whether the method includes the corresponding modifier.
Default implementation	Binary	Whether the method in an interface has a default implementation.
Has override	Binary	Whether the method has an @Override annotation.
Number of parameters	Integer	The number of parameters in that method.
Number of generics	Integer	The number of type variables in the generic method (0 = non-generic).
Number of throws	Integer	The number of exceptions declared in the throws clause (0 = no-throws).
Number of annotations	Integer	The number of annotations used in the method.
Number of statements	Integer	The number of statements in the method body.
Number of local variables	Integer	The number of local variables defined in the method body.
Naming convention	Nominal	The naming convention used for the method identifier (Lower, Upper, CamelLow, CamelUp or SnakeCase).
Constructor	Binary	Whether the method in a constructor.
Return type	Nominal	The type returned by the method (e.g., Int, String and Reference).
Number of inner classes	Integer	The number of inner classes defined in the method body.
Number of overloaded methods	Integer	The number of overloaded method implementations for the given method name.
Type of the 1st, 2nd and 3rd parameter	Nominal	Syntactic category of the corresponding types.
Expertise	Nominal	Beginner or Expert.

Table A.4
Features of the field definitions dataset.

Name	Type	Description
Visibility	Nominal	Public, protected, package or private.
Static, final, volatile and transient	Binary	Whether the field defines the corresponding modifier.
Number of annotations	Integer	The number annotations used in the field.
Naming convention	Nominal	The naming convention used for the field identifier (Lower, Upper, CamelLow, CamelUp or SnakeCase).
Initial value	Nominal	The expression used to initialize the field (syntactic category of expression; None = no-initialization).
Type	Nominal	The type of the field (e.g., Int, String and Reference).
Parent node	Nominal	Syntactic category of the parent node.
Expertise	Nominal	Beginner or Expert.

Table A.5
Features of the type definitions dataset.

Name	Type	Description
Category	Nominal	Syntactic category of the current node (class, interface or enumeration).
Public	Binary	Whether the type is public.
Abstract, static, final and strictfp	Binary	Whether the type includes the corresponding modifier.
Has extends	Binary	Whether the type extends another type.
Number of annotations	Integer	The number annotations used in the type definition.
Number of implements	Integer	The number interfaces directly implemented by the type.
Number of generics	Integer	The number of type variables in the generic type (0 = non-generic).
Number of methods, constructors, fields, static blocks, and (static) nested and inner types	Integer	The number methods, constructors, fields, static blocks, and (static) nested and inner classes defined in the type.
Naming convention	Nominal	The naming convention used in the type identifier (Lower, Upper, CamelLow, CamelUp, SnakeCase or Anonymous).
Inner class, nested class	Binary	Whether the type is an inner or nested class.
Percentage of overloaded methods, static fields and static methods	Real [0–100]	Percentage of overloaded method static fields and static methods defined in the type.
Default package	Binary	Whether the type is defined in the default package.
Expertise	Nominal	Beginner or Expert.

Table A.6
Features of the programs dataset.

Name	Type	Description
Percentage of classes, interfaces and enums	Real [0–100]	Percentage of classes, interfaces and enumerations defined in the program.
Code in default packages	Binary	Whether the project has types defined in the default package.
Code in packages	Binary	Whether the project has types defined in packages.
Number of types in packages	Integer	The number of types defined in packages.
Number of types default package	Integer	The number of types defined in the default package.
Expertise	Nominal	Beginner or Expert.

References

Abdi, H., & Williams, L. J. (2010). Tukey's honestly significant difference (HSD) test. In *Encyclopedia of research design* (pp. 1–5). SAGE.

Aggarwal, K., Salameh, M., & Hindle, A. (2015). *Using machine translation for converting python 2 to python 3 code: Technical Report PeerJ PrePrints.*

Allamanis, M. (2022). Graph neural networks in program analysis. In *Graph neural networks: foundations, frontiers, and applications* (pp. 483–497). Springer.

Allamanis, M., Barr, E. T., Ducousso, S., & Gao, Z. (2020). Typilus: Neural type hints. In *Proceedings of the 41st ACM SIGPLAN conference on programming language design and implementation* (pp. 91–105).

Allamanis, M., & Sutton, C. (2014). Mining idioms from source code. In *FSE 2014, Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering* (pp. 472–483). New York, NY, USA: Association for Computing Machinery.

Andrew, W. A., & Jens, P. (2002). *Modern compiler implementation in Java*. Cambridge University Press, ISBN 0–521–58388–8.

Ankerst, M., Breunig, M. M., Kriegel, H.-P., & Sander, J. (1999). OPTICS: Ordering points to identify the clustering structure. *ACM Sigmod Record*, *28*(2), 49–60.

Arakelyan, S., Hakhverdyan, A., Allamanis, M., Hauser, C., Garcia, L., & Ren, X. (2022). NS3: Neuro-symbolic semantic code search. arXiv preprint 2205.10674.

Barone, A. V. M., & Sennrich, R. (2017). A parallel corpus of Python functions and documentation strings for automated code documentation and code generation. arXiv preprint 1707.02275.

Baxter, I., Yahin, A., Moura, L., Sant'Anna, M., & Bier, L. (1998). Clone detection using abstract syntax trees. In *Proceedings of the international conference on software maintenance* (pp. 368–377). Washington DC: IEEE Computer Society.

Bhatia, S., & Singh, R. (2016). Automated correction for syntax errors in programming assignments using recurrent neural networks. arXiv preprint 1603.06129.

Bhoopchand, A., Rocktäschel, T., Barr, E., & Riedel, S. (2016). Learning Python code suggestion with a sparse pointer network. arXiv preprint 1611.08307.

Bloch, J. (2008). *Effective Java: A programming language guide* (2nd Revised edition (REV)). Addison-Wesley Longman, Amsterdam.

Bock, H.-H. (2007). Clustering methods: a history of k-means algorithms. In *Selected contributions in data analysis and classification* (pp. 161–172). Springer.

Breiman, L. (1984). *Classification and regression trees*. Routledge.

Cabrera, A. F. (1994). Logistic regression analysis in higher education: An applied perspective. In *Higher education: handbook of theory and research. Vol. 10* (pp. 225–256).

Choudhury, R. R., Yin, H., & Fox, A. (2016). Scale-driven automatic hint generation for coding style. In *Proceedings of the 13th international conference on intelligent tutoring systems* (pp. 122–132). Berlin, Heidelberg: Springer-Verlag.

Cohen, W. W. (1995). Fast effective rule induction. In A. Prieditis, & S. Russell (Eds.), *Machine learning proceedings 1995* (pp. 115–123). San Francisco (CA): Morgan Kaufmann.

Ester, M., Kriegel, H.-P., Sander, J., Xu, X., et al. (1996). A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the second international conference on knowledge discovery and data mining. Vol. 96* (34), (pp. 226–231).

Fürnkranz, J., & Widmer, G. (1994). Incremental reduced error pruning. In *Machine learning proceedings 1994* (pp. 70–77). Elsevier.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. M. (1994). *Design patterns: elements of reusable object-oriented software* (1st ed.). Addison-Wesley Professional.

GitHub (2022a). Copilot, your AI pair programmer. https://github.com/features/copilot.

GitHub (2022b). One million repositories. https://github.blog/2010-07-25-one-million-repositories.

GitHub (2022c). Where the world builds software. https://github.com/about.

Iyer, V., & Zilles, C. (2021). Pattern census: A characterization of pattern usage in early programming courses. In *Proceedings of the 52nd ACM technical symposium on computer science education* (pp. 45–51). New York, NY, USA: Association for Computing Machinery.

Jolliffe, I. T., & Cadima, J. (2016). Principal component analysis: a review and recent developments. *Philosophical Transactions of the Royal Society of London A (Mathematical and Physical Sciences)*, *374*.

Kaur, G. (2014). Association rule mining: A survey. *International Journal of Computer Science and Information Technologies*, *5*(2), 2320–2324.

Lee, D., & Seung, H. S. (2000). Algorithms for non-negative matrix factorization. *Advances in Neural Information Processing Systems*, *13*, 1–7.

Li, Y., Zhou, J., Verma, S., & Chen, F. (2022). A survey of explainable graph neural networks: Taxonomy and evaluation metrics. arXiv preprint 2207.12599.

Liu, F. T., Ting, K. M., & Zhou, Z.-H. (2008). Isolation forest. In *2008 Eighth IEEE international conference on data mining* (pp. 413–422). IEEE.

Losada, A., Facundo, G., Garcia, M., & Ortin, F. (2022). Mining common syntactic patterns used by Java programmers. *IEEE Latin America Transactions*, *20*(5), 753–762.

Moscovitz, I. (2022). IREP and RIPPER*k* Wittgenstein rule induction algorithms. https://pypi.org/project/wittgenstein.

Oracle (2022a). Java naming conventions. https://www.oracle.com/java/technologies/javase/codeconventions-namingconventions.html.

Oracle (2022b). JDK documentation - Java platform, standard edition tools reference. https://docs.oracle.com/javase/9/tools/javac.htm#JSWOR627.

Ortin, F., Escalada, J., & Rodriguez-Prieto, O. (2016). Big code: New opportunities for improving software construction. *Journal of Software*, *11*(11), 1083–1088.

Ortin, F., Facundo, G., & Garcia, M. (2022). Analyzing syntactic constructs of Java programs with machine learning (support material website). https://www.reflection.uniovi.es/bigcode/download/2022/java-patterns.

Ortin, F., Rodriguez-Prieto, O., Pascual, N., & Garcia, M. (2020). Heterogeneous tree structure classification to label Java programmers according to their expertise level. *Future Generation Computer Systems*, *105*, 380–394.

Peng, C.-Y. J., Lee, K. L., & Ingersoll, G. M. (2002). An introduction to logistic regression analysis and reporting. *The Journal of Educational Research*, *96*(1), 3–14.

Pradel, M., & Sen, K. (2018). Deepbugs: A learning approach to name-based bug detection. *Proceedings of the ACM on Programming Languages*, *2*(OOPSLA), 1–25.

Python (2022). ast – abstract syntax trees. https://docs.python.org/3/library/ast.html.

Qiu, D., Li, B., Barr, E. T., & Su, Z. (2017). Understanding the syntactic rule usage in Java. *Journal of Systems and Software*, *123*, 160–172.

Rodriguez-Prieto, O., Mycroft, A., & Ortin, F. (2020). An efficient and scalable platform for Java source code analysis using overlaid graph representations. *IEEE Access*, *8*, 72239–72260.

Satopaa, V., Albrecht, J., Irwin, D., & Raghavan, B. (2011). Finding a "kneedle" in a haystack: Detecting knee points in system behavior. In *2011 31st International conference on distributed computing systems workshops* (pp. 166–171). IEEE.

Schölkopf, B., Smola, A. J., & Müller, K. (1997). Kernel principal component analysis. In W. Gerstner, A. Germond, M. Hasler, & J. Nicoud (Eds.), *Lecture notes in computer science: vol. 1327, Artificial neural networks - ICANN '97, 7th international conference, Lausanne, Switzerland, October 8-10, 1997, Proceedings* (pp. 583–588). Springer.

SciKit-Learn (2022a). Class and function reference of scikit-learn. https://scikit-learn.org/stable/modules/classes.html.

SciKit-Learn (2022b). DBSCAN - density-based spatial clustering of applications with noise. https://scikit-learn.org/stable/modules/generated/sklearn.cluster.DBSCAN.html#sklearn.cluster.DBSCAN.

SciKit-Learn (2022c). Decistion tree classifier. https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html#sklearn.tree.DecisionTreeClassifier.

SciKit-Learn (2022d). Isolation forest algorithm. https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.IsolationForest.html?highlight=isolationforest#sklearn.ensemble.IsolationForest.

SciKit-Learn (2022e). K-Means clustering. https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html#sklearn.cluster.KMeans.

SciKit-Learn (2022f). Logistic regression classifier. https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html.

SciKit-Learn (2022g). OPTICS - ordering points to identify the clustering structure. https://scikit-learn.org/stable/modules/generated/sklearn.cluster.OPTICS.html#sklearn.cluster.OPTICS.

van der Maaten, L., & Hinton, G. E. (2008). Visualizing high-dimensional data using t-SNE. *Journal of Machine Learning Research*, *9*, 2579–2605.

Wu, L., Cui, P., Pei, J., Zhao, L., & Song, L. (2022). Graph neural networks. In *Graph neural networks: foundations, frontiers, and applications* (pp. 27–37). Springer.

Yin, H., Moghadam, J., & Fox, A. (2015). Clustering student programming assignments to multiply instructor leverage. In *Proceedings of the second (2015) ACM conference on learning @ scale* (pp. 367–372).

Zou, H., & Hastie, T. (2005). Regularization and variable selection via the elastic net. *Journal of the Royal Statistical Society. Series B. Statistical Methodology*, *67*(2), 301–320.