Computers and Education Open 10 (2026) 100333

Contents lists available at ScienceDirect

Computers and Education Open

journal homepage: www.sciencedirect.com/journal/computers-and-education-open

Learning personalization in block-based programming languages using clustering and static code analysis

Tatiana Person [a], Juan Antonio Caballero-Hernández [a,*], Cristóbal Romero [b], Iván Ruiz-Rube [a], Juan Manuel Dodero [a]

[a] University of Cadiz, Department of Computer Science, Av. Universidad de Cádiz, 10 Puerto Real (11519), Spain
[b] University of Córdoba, Department of Computer Science and Artificial Intelligence, Edif. C2 Albert Einstein, Campus de Rabanales Córdoba (14071), Spain

ARTICLE INFO

Keywords:
Learning personalization
Block-based languages
Machine learning
K-Means
Hierarchical Agglomerative

ABSTRACT

Personalized learning in programming education aims at adapting instructional strategies to the diverse needs of students, particularly novice programmers. However, traditional approaches often fail to accommodate individual learning styles or emphasize clean coding practices. This study proposes a data-driven method to personalize programming education by analyzing large-scale datasets from block-based Visual Programming Language (VPL) projects created by novice programmers. Using static code analysis and machine learning, the approach examines the coverage of programming concepts and code quality metrics to identify patterns in student performance, enabling adaptive learning pathways. This approach is implemented in BlocklyMining, a tool that integrates static code analysis, quality assessment through SQALE, and machine-learning-based clustering. The study applies K-Means and Hierarchical Agglomerative Clustering to 215,244 MIT App Inventor projects, evaluating cluster quality using the Silhouette Coefficient (SC) and Davies–Bouldin Index (DBI). Results show that the clustering algorithms effectively group projects, thereby facilitating the generation of personalized learning recommendations tailored to novice programmers' skill levels.

1. Introduction

Computer programming is an increasingly essential skill, not only for computer science students, but also for individuals in a wide range of disciplines [1,2]. As programming becomes integral to various fields, educators face the challenge of making coding accessible to a broader audience. Block-based Visual Programming Languages (VPLs) address this challenge by providing visual representations of complex programming language structures, making it easier for users without programming experience to understand these concepts [3]. VPLs are more accessible to beginners owing to higher usability and less rigid syntax than textual languages, while also providing a more enjoyable and engaging programming experience [4]. In this context, adapting instruction to individual learner needs is a plausible next step.

Personalized learning has provided significant advantages to students, primarily by improving learning outcomes and has long been a key objective in educational technology [5]. Studies indicate that it can improve academic performance by allowing students to progress at their own pace and in ways that align with their individual learning styles [6]. Furthermore, personalized learning fosters greater engagement and motivation, which in turn promotes deeper understanding and retention of knowledge. Moreover, using data-driven insights, educators can better understand the strengths, weaknesses, and preferences of students, enabling them to provide more targeted instruction and support [7]. In programming education, this entails considering not only which concepts are covered but also how code is written.

The recommendation of different learning strategies for different learners in computer programming has been a significant issue [8]. Traditional approaches to personalization in computer programming learning often rely on standardized instructional designs, which, while theoretically optimized, do not necessarily align with the diverse needs and learning preferences of individual students. This misalignment can be particularly challenging for novice programmers, who may benefit from a more adaptive educational approach tailored to their unique learning styles and difficulties. Additionally, an often overlooked aspect of computer programming education is the teaching of clean coding practices—a critical element in professional software development [9]. Despite their importance, the principles of writing clean and maintainable code are seldom emphasized in the early stages of programming education.

* Corresponding author.
E-mail addresses: tatiana.person@uca.es (T. Person), juanantonio.caballero@uca.es (J.A. Caballero-Hernández), cromero@uco.es (C. Romero), ivan.ruiz@uca.es (I. Ruiz-Rube), juanma.dodero@uca.es (J.M. Dodero).

https://doi.org/10.1016/j.caeo.2026.100333
Received 29 May 2025; Received in revised form 8 January 2026; Accepted 13 January 2026
Available online 14 January 2026

This study proposes a novel data-driven approach to customizing programming education by analyzing large-scale datasets from novice block-based programming projects. Our approach examines patterns in programming concept coverage and code quality to predict and adapt learning pathways that better align with learners' needs. A core aspect of this approach involves the use of code quality metrics, obtained through static code analysis, to guide the learning process. Static code analysis helps identify potential software defects before they cause problems, allowing programmers to address them proactively [10]. These metrics offer valuable insight into coding practices, allowing a structured integration of clean coding principles into the learning process.

This approach aims to create more effective and relevant learning experiences for novice programmers by combining learner data analysis with clean code metrics. The data-driven customization enhances learner engagement and learning outcomes while also equipping educators with actionable data to refine instructional strategies, bridging the gap between theoretical learning and practical coding proficiency. Consequently, this perspective supports connecting static analysis and clustering with personalized guidance.

Based on these insights, the following hypothesis is presented: *the use of static code analysis, enhanced by machine learning techniques, provides a personalized learning experience for users of block-based VPLs*. Therefore, an investigation is conducted to determine whether machine learning can group applications according to various characteristics to offer personalized learning guidance to VPL users. This guidance is operationalized by mapping static-analysis metrics to cluster-defined proficiency levels and then to concrete, block-level recommendations presented in-app (MIT App Inventor), thus producing adaptive learning pathways for novice programmers.

The rest of this paper is structured as follows. Section 2 provides background information on personalized learning and its applications in programming education. Section 3 presents the proposed method for data-driven learning customization. Section 4 describes the validation of the method and the dataset used in this study. Section 5 details the experimentation and results. Finally, Section 6 discusses the conclusions and future work.

2. Personalized learning in block-based programming languages

Personalized learning in programming is based on adapting instruction to precisely match students' individual needs and preferences. This involves modifying the teaching methods, content, and learning experience according to the prior knowledge, skills, and learning style of each student [8]. This approach acknowledges that students differ in terms of their previous experience, interests, and aptitudes, aiming to provide a more engaging and effective learning experience [11].

The concept of personalized learning in programming emerged in the early 2000s, when initial studies explored how technology could be used to tailor instruction to individual learners. One study focused on the use of personal robots as tools for teaching programming to students [12]. This study found that personalization played a crucial role in students' success in introductory programming courses. In addition, it demonstrated that effective interventions could be implemented to improve self-efficacy during instruction.

Another study investigated the incorporation of software agents with varying levels of knowledge into the learning process to assist educators in creating personalized learning experiences [13]. This study highlighted the importance of coordinating teaching and learning efforts to support personalized learning in online environments.

The development of innovative learning approaches and tools has also driven personalized learning in programming. An example is an interactive and collaborative learning platform designed to teach Java programming anytime and anywhere [11]. This platform aims to spark students' interest in programming, improve the effectiveness of programming education, and keep students engaged in their computer science studies.

A key aspect of personalized learning is the use of recommendation systems, which automatically generate tailored suggestions to adapt learning experiences to individual students. These systems analyze the data of the learners to model their profiles, group them based on similarities, and establish relationships between the learners and the learning objects [14]. Building an automated recommendation system requires defining key parameters of the learning model, such as learning styles, prior knowledge, learning paths, performance, and social interactions. Various recommendation techniques, including ontology-based approaches, collaborative filtering, and machine learning algorithms like K-Nearest Neighbor and K-Means, enhance recommendations by identifying patterns in learner behavior [15,16]. Alongside these methods, feedback mechanisms help adjust recommendations over time, ensuring a more adaptive and personalized learning experience [17].

Block-based VPLs such as Scratch and MIT App Inventor have gained prominence as educational tools that facilitate programming instruction [18]. Scratch,[1] developed by the MIT Lifelong Kindergarten Group in 2002, was originally designed for students aged 8 to 16 but has since been adopted in higher education [19]. It enables users to create and share visual content, including games and animations. MIT App Inventor,[2] initially developed by Google in 2010 and now maintained by MIT, is widely used for teaching introductory computer science, particularly to non-engineering students. Since the release of MIT App Inventor 2 in 2013, its block-based syntax has been built on Blockly, a JavaScript library developed by Google in 2012.

Just like textual programming languages, programs developed using VPLs can also contain errors and exhibit deficiencies in their code structure (known as code smells). Static code analysis, which originated with tools like Lint to detect errors in C code [20], has significantly evolved, incorporating artificial intelligence and machine learning techniques [21]. Static analysis helps improve code quality, ensuring compliance with coding standards and best practices by identifying defects early [22]. Different approaches exist, including metric-based, change-based, and architecture-based analysis, all often used in the industry despite scalability challenges in handling large codebases. One widely adopted method to assess software quality is Software Quality Assessment based on Lifecycle Expectations (SQALE) [23,24], which evaluates software components based on attributes such as maintainability and reliability throughout the development lifecycle.

A literature review was conducted to find studies on block-based VPLs that personalize the user learning experience. First, a search was carried out in several recognized digital libraries, including IEEE Xplore Digital Library, ACM Digital Library, and Springer. The search terms used were "block-based programming language" and "personalized learning" applying the following criteria:

- *Off topic*: Publication dated before 1986, the year the first VPL was developed.
- *Duplicated*: Work whose content is similar to or the same as one already included.
- *Out of scope*: Work whose content is not relevant to this study.
- *Unsupported language*: Work published in a language other than English or Spanish. These are the authors' working languages and cover the vast majority of publications in computing education indexed in the searched databases.
- *Included*: Work whose content is relevant to this study.

In total, 176 relevant publications were found. These publications were analyzed according to the established selection criteria. Table 1 presents the query used in each library, along with the number of works found and the number selected for this search. After the analysis, only 2 publications met the criteria and were selected, while 174 were discarded. Table 2 details the selected publications, including their year of publication and other relevant information.

Table 1
Search results: Block-based language and personalized learning.

Library	*Query*	*Results*	*Selected*
IEEE Digital Library	Abstract: block-based programming language AND Abstract: "personalized learning"	1	0
ACM Digital Library	Abstract: block-based programming language AND Abstract: "personalized learning"	129	0
Springer	block-based AND programming AND language AND "personalized learning"	48	2

Table 2
Study selection: Block-based language and personalized learning.

Criterion	*Publications*	*Frequency*
Off topic	0	0%
Out of scope	174	98.86%
Unsupported language	0	0%
Duplicated	0	0%
Included	2	1.14%
Total	**118**	**100%**

Next, the following question was posed in *Scite AI Assistant*[3]: *Tell me which are the existing block-based programming tools to develop apps that supports personalized learning*. In this case, three relevant studies were returned. The first study [25] mentions the ALAS pedagogical architecture, used to create mobile applications that guide students in their learning process. Although it includes a case study with MIT App Inventor, its primary focus is on creating applications that guide teachers, making instructors the main target audience. The second study [26] presents an architecture for student self-management of learning content, but focuses on the use of Learning Management Systems. Its main focus is on managing and organizing learning content, not specifically on block-based programming environments. The third study [27] describes a classroom case study applying personalized learning, but it is not directly related to block-based programming environments. In this case, although all studies provided by *Scite AI Assistant* are related to personalized learning, they do not focus on its specific application in block-based VPL environments.

Across the two searches mentioned, we found two studies that address the application of personalized learning in block-based environments. The first study [28] presents a chatbot with an automated assessment module for student projects in Scratch. The chatbot applies evaluation criteria predefined by the teacher and, based on the obtained score, suggests the next practice activity while providing the necessary learning materials. This process is tailored to each student's profile according to the knowledge they demonstrated in previous tasks. The second one [29] focuses on projects created with ScratchJr, a mobile application designed for children aged 5 to 7. It compares children's behavior when developing applications at school and at home and proposes a personalized learning approach to align parents and teachers in guiding children. However, no specific software solution is implemented for this purpose.

Our review focused on peer-reviewed publications indexed in major digital libraries and did not systematically include gray literature or platform-curated resources. Notably, MIT App Inventor research portal[4] aggregates studies and materials that provide insights into user behavior and feedback mechanisms within App Inventor. In addition, we restricted sources to English and Spanish, potentially omitting work published in other major academic languages (e.g., Chinese, Portuguese, Russian). While these resources were outside our predefined search scope, they may contain relevant evidence on personalization. Future work should incorporate this portal and related gray-literature sources into a broader review to complement the peer-reviewed evidence base.

Personalized learning in the Scratch environment has been quite limited, with only two studies identified. Furthermore, this methodology has not yet been applied to MIT App Inventor. These studies highlight the importance and potential of personalized learning in visual programming environments, suggesting that exploring similar techniques in MIT App Inventor could enhance students' learning experiences by adapting to their individual needs and skills. Additionally, none of the reviewed studies have implemented personalized learning through machine learning clustering techniques, which represents a clear gap to address. As these environments allow users to create applications on diverse topics, recommendations should be generated automatically to personalize the experience. To achieve this, developing a machine-learning-based recommendation system is crucial. These observations frame the design choices we operationalize in the proposed method.

3. Data-driven method for personalized learning in block-based programming languages

This section presents a method designed to support novice programmers using block-based VPLs, to foster continuous improvement in their applications through personalized learning. Users enhance their applications based on two main sources, namely (1) the SQALE score obtained from static code analysis and (2) the classification of each application according to a set of criteria. The SQALE score indicates whether the application needs to address identified *code smells*,[5] while classification aims to identify concepts the programmer may need to learn.

The proposed approach is inspired by the Design Science Research (DSR) category of methods [30], which focuses on creating artifacts (e.g., a software artifact) to address an identified research problem. In this study, the software artifact integrates static code analysis features and clustering algorithms for classification to provide personalized learning recommendations. The BlocklyMining tool has been developed to support this method, as described in Section 4.

The research design includes an iterative approach in which users refine their applications based on automated feedback, aligning with the core principles of DSR experimentation. The general methodology described below consists of several stages and involves key stakeholders, as shown in Fig. 1.

- *Researchers*: Experts in static code analysis and personalized learning. They define the static analysis metrics and classification criteria for machine learning algorithms. In addition, they evaluate results, identify user behavior patterns, and propose new metrics or classification criteria.
- *Static analysis metric developers*: Programming experts are responsible for implementing and refining the static code analysis metrics defined by researchers.
- *Machine learning algorithm developers*: Machine learning specialists who implement algorithms that are tailored to the chosen criteria, providing personalized learning recommendations.
- *Users*: Novice programmers use block-based VPLs to create applications that are analyzed for quality and learning progression.

[1] https://scratch.mit.edu
[2] https://appinventor.mit.edu
[3] https://scite.ai/assistant
[4] MIT App Inventor Research: https://appinventor.mit.edu/explore/research

[5] A code smell is a surface indication of a deeper problem in the design or structure of computer programming code that may hinder maintainability or scalability.

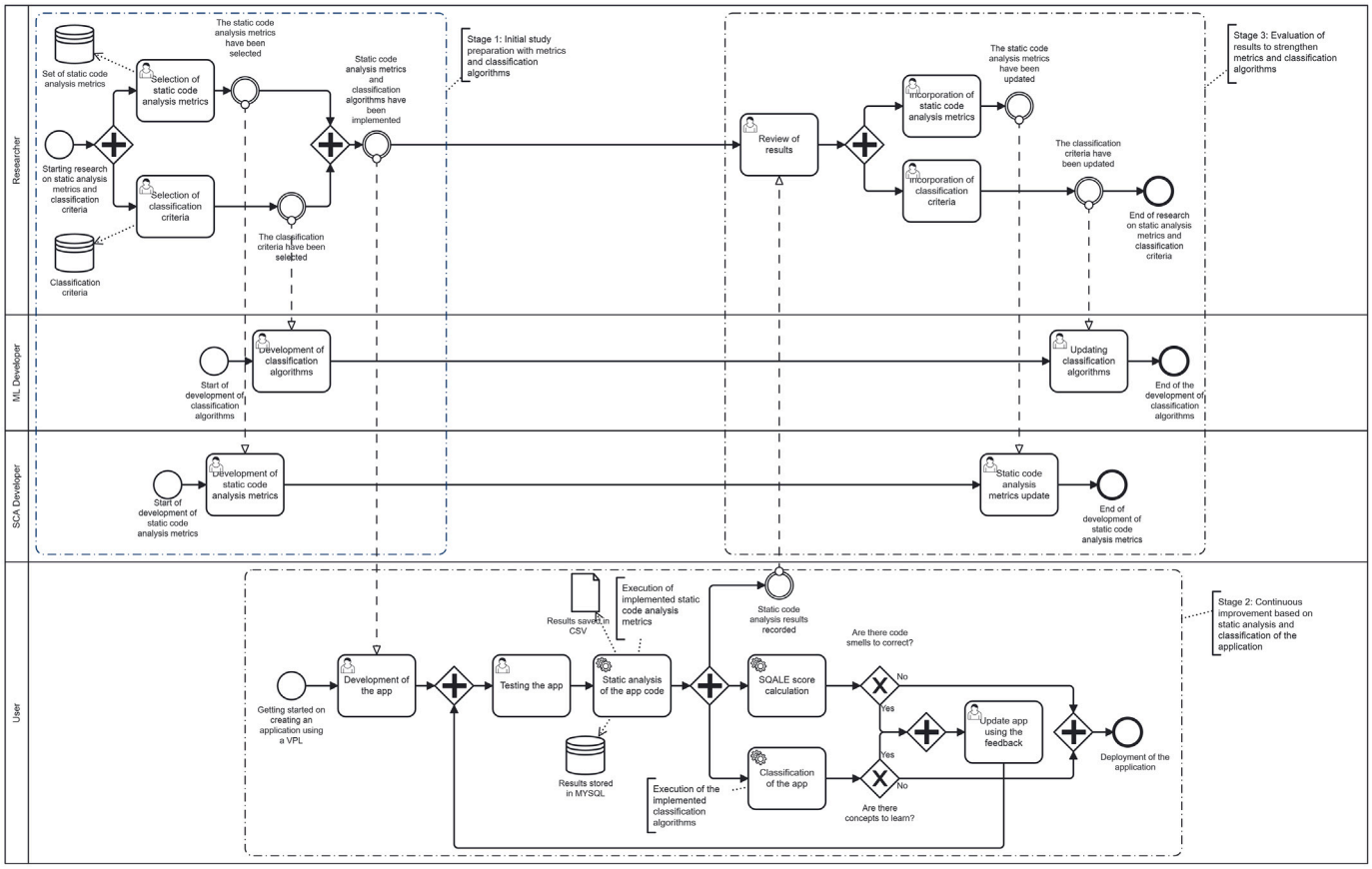

Fig. 1. Stages of the method for applying static code analysis as part of programming learning.

The proposed method is flexible and follows an iterative approach, allowing novice programmers to create and refine multiple versions of their applications, which are then subjected to static code analysis. The method also enables developers to integrate new analysis metrics and clustering algorithms for classification on identified learning patterns. The method consists of three main stages:

- *Initial study preparation*: The researchers define static code analysis metrics and classification criteria. The developers then implement these metrics and provide the machine learning algorithms for classification.
- *Continuous improvement through static analysis and classification*: Users develop applications using a block-based VPL, designing both the interface and behavior. The applications undergo automatic static analysis, receiving the SQALE score based on identified code smells. The clustering algorithms are then used to categorize the application according to a set of predefined criteria. Users iteratively refine their applications based on detected issues and classification feedback.
- *Evaluation and refinement*: The researchers analyze the results to identify new patterns of user behavior, leading to the proposal of additional metrics or classification criteria. Developers update the tool accordingly, integrating new metrics and machine learning algorithms.

The proposed method directly addresses the identified gap in personalized learning within visual programming environments. By integrating static code analysis with machine-learning-based classification, this approach enables automated, data-driven recommendations tailored to each user's learning needs.

Each recommendation provides a detailed explanation of the corresponding metric and its use within MIT App Inventor. This information is accessible directly through the platform's interface. By describing both the concept behind the metric and the functionality of the relevant blocks, the system helps users integrate clean coding practices effectively into their projects. Moreover, a visual demonstration illustrates how to identify and utilize these blocks, ensuring clearer guidance for learners.

Operationally, we translate the analysis into adaptive guidance in five steps: (1) Metric extraction: obtain static indicators of concept usage and code quality; (2) Learner profiling: derive project levels that reflect usage patterns; (3) Gap detection: identify underused or missing concepts per level; (4) Recommendation mapping: link each gap to targeted, concept-level guidance and examples; and (5) Delivery and iteration: present recommendations in the IDE and re-analyze projects to update levels and next steps.

Recommendations are authored and validated by the Researchers as part of the study design: they define the metric signals that indicate conceptual gaps and the rule-based mapping from each gap to a concrete recommendation (explainer, example snippet, refactoring hint). The unsupervised clustering assigns projects to learner levels where these rules are applied. Metric/Machine Learning developers implement the extraction, clustering, and rule engine to integrate the guidance into the MIT App Inventor interface. End-users receive the in-IDE guidance and iterate on their projects, closing the feedback loop.

4. Method validation

This section presents the evaluation framework for the proposed method, detailing both the implementation of the BlocklyMining[6] support tool and the dataset used for validation. The tool is described

[6] https://github.com/TatyPerson/BlocklyMining

in terms of its core services, while the dataset overview highlights its origin, structure, and filtering criteria. To support transparency, we provide installation, configuration, and execution details for BlocklyMining in Appendix, including tool versions, dependency list, and the command-line used in our experiments.

4.1. BlocklyMining implementation

The BlocklyMining tool extracts data from applications' source files by applying the implemented static code analysis metrics. MIT App Inventor was selected as the VPL to implement our tool due to its widespread adoption among novice programmers. Additionally, the scarcity of previous research on personalized learning in MIT App Inventor highlights the need for further exploration in this area. The availability of a big dataset containing real user projects, provided by MIT App Inventor researchers, has facilitated the implementation of the data-driven method.

BlocklyMining was implemented in Python and exposes a command-line interface and service modules so that experiments can be executed in batches over exported projects. In this study, we used BlocklyMining 1.4.0, Python 3.10, and Ubuntu 22.04 LTS. Installation, configuration, and execution details are provided in Appendix. By default, results are written as CSV in the target directory. A typical offline run over a directory of MIT App Inventor projects is launched as follows:

```
python BlocklyMining.py --source=appinventor
    --path=/projects
```

BlocklyMining follows a client–server architecture. When a user creates a project in MIT App Inventor, the corresponding *.aia* project file is sent to BlocklyMining for static code analysis. Then, the system performs a static code analysis and classifies the project using machine learning algorithms. Based on this classification, a recommendation is generated and sent back to the MIT App Inventor environment, guiding the necessary improvements to reach the next classification level. The system also assigns an SQALE-based quality score and communicates it to the user.

During the experiments conducted for this study, the tool was executed to process project files independently rather than as a real-time interaction with MIT App Inventor. However, the system architecture has been designed for its future deployment as an autonomous service. The architecture of BlocklyMining is depicted in Fig. 2 using the C4 model notation [31]. It contains three core services, namely *Static Code Analysis Service*, *Quality Measurement Service* and *Personalized Learning Service*.

The *Static Code Analysis Service* extracts and evaluates code metrics from MIT App Inventor projects. The service provides an insight into project structure, potential issues, and programming practices in a visual programming environment. First, *Element-based* metrics quantify the presence of different elements in applications, such as the number of screens, blocks, event handlers, loops, and variables. This data helps assess application complexity and understand user preferences in component usage. Second, *code smell* metrics detect potential issues in the code, such as duplicated or unreachable code, overly long functions, poorly named variables and unused variables. Finally, *SonarQube-inspired* metrics were integrated to enhance the analysis,[7] focusing on cyclomatic and cognitive complexity, excessive nesting, magic numbers, uninitialized variables, and infinite loops.

The *Quality Measurement Service* assigns a quality score to projects based on the SQALE model [23], enabling the analysis of code smells and errors in applications built with MIT App Inventor. To achieve this, a set of metrics has been defined to identify potential issues in the code, such as block duplication, excessively long functions, unused variables, or overly nested control structures. These metrics help detect poor coding practices and assess the project's technical debt.

Following the approach used in SonarQube, technical debt is calculated by assigning a correction factor to each metric and multiplying it by the number of detected occurrences. This provides a quantitative measure of the effort required to fix the identified issues. In addition, the system categorizes the impact of each code smell or error on the application functionality, establishing different severity levels. Then, the tool assigns a maintainability score based on the project's technical debt percentage, classifying it into levels ranging from A (low technical debt) to E (very high technical debt). This feature provides users with a clear evaluation of code quality and guidance on necessary improvements to enhance maintainability.

The *Personalized Learning Service* employs machine learning algorithms to classify projects and provide personalized learning recommendations. As discussed in Section 2, developing an automated recommendation system requires careful consideration of several key aspects such as learning model parameters, recommendations and feedback mechanisms. Each recommendation appears in a side panel within the blocks editor, combining a concise concept explanation, a visual cue to the exact blocks to add or refactor, and a minimal working example that learners can copy and adapt. For the present validation, recommendations were generated by first defining cluster-to-action mappings and later applying them in the BlocklyMining tool.

4.2. Metrics and quality scoring

In this study, an expert in programming education manually selected the metrics. The selection process prioritized metrics directly related to fundamental programming concepts typically introduced during the early stages of learning, such as control structures (conditionals and loops), event-driven programming, functional decomposition, and variable management. From the three available categories of metrics in BlocklyMining (Element-based, Code Smell and SonarQube-Inspired ones), only Element-based metrics were considered. This decision was made because Element-based metrics offer a direct and interpretable measure of the programming constructs commonly used by students, avoiding the added complexity and abstraction introduced by Code Smell and SonarQube-Inspired metrics, which are more suited for advanced code quality assessments.

Within the Element-based metrics of Table 3, 7 out of 9 metrics were selected (i.e., M3 to M9). Metrics M1 (i.e., number of screens) and M2 (i.e., total number of blocks) were excluded, as they provide aggregate information that does not specifically target particular programming concepts. Instead, the selected metrics emphasize distinct programming skills, facilitating a clearer interpretation of students' abilities and supporting the goal of providing actionable learning recommendations. The code employed for manual Element-based metrics is shown in Listing 1.

```
X = np.array(df[['NumberOfDefinedFunctionalBlocks',
'NumberOfUsedFunctionalBlocks','NumberOfEventsBlocks',
'NumberOfConditionalBlocks','NumberOfLoopsBlocks',
'NumberOfGlobalVariables','NumberOfLocalVariables']])
```

Listing 1: Manual selection of metrics

4.3. Clustering and evaluation metrics

K-Means was selected because the dataset lacks labels and the goal is unsupervised profiling; K-Means is a well-established baseline in educational recommenders for grouping users and driving personalized suggestions [14]. Its centroid-based partitions are computationally efficient at this scale and yield interpretable profiles (via feature means) that map cleanly to actionable guidance. Its implementation is shown in Listing 2. An Elbow curve is generated to determine the optimal number of clusters, and its implementation is shown in Listing 3.

[7] https://docs.sonarqube.org/latest/user-guide/metric-definitions/

Table 3
Usage metrics of MIT App Inventor VPL elements.

ID	*Metric*	*Description*
M1	NumberOfScreens	Number of screens in the application.
M2	NumberOfTotalBlocks	Total number of blocks in the application.
M3	NumberOfDefinedFunctionalBlocks	Number of functional blocks defined in the application.
M4	NumberOfUsedFunctionalBlocks	Number of function calls belonging to the components available in MIT App Inventor.
M5	NumberOfEventsBlocks	Number of event blocks in the application.
M6	NumberOfConditionalBlocks	Number of conditional blocks (*if/else*, *switch*) in the application.
M7	NumberOfLoopsBlocks	Number of loop blocks (*for*, *while*, *do-while*) in the application.
M8	NumberOfGlobalVariables	Number of global variable definition blocks in the application.
M9	NumberOfLocalVariables	Number of local variable definition blocks in the application.

```
kmeans = KMeans(n_clusters=4).fit(X)
centroids = kmeans.cluster_centers_
preds = KMeans(n_clusters=4).fit_predict(X)
```

Listing 2: K-Means classification with K = 4

```
Nc = range(1, 20)
kmeans = [KMeans(n_clusters=i) for i in Nc]
score = [kmeans[i].fit(X).score(X) for i in
    range(len(kmeans))]
plt.plot(Nc, score)
plt.xlabel('Number of Clusters')
plt.ylabel('Score')
plt.title('Elbow curve')
plt.show()
```

Listing 3: Elbow curve for selecting the optimal K

Then, a Hierarchical Agglomerative Clustering approach was also included to validate the results obtained with K-Means. This alternative method provides a different classification perspective while maintaining the unsupervised nature of the analysis. The implementation code is shown in Listing 4.

```
agglom = AgglomerativeClustering(n_clusters = 4,
    linkage = 'ward')
preds = agglom.fit_predict(X)
```

Listing 4: Hierarchical Agglomerative classification with K = 4

A group-based recommendation strategy was adopted, leveraging the classification structure generated by K-Means and Hierarchical Clustering. This method allows recommendations to be tailored to users based on their cluster assignment. Euclidean distance was selected as the similarity measure, since it is the internal metric used by both K-Means and Hierarchical Agglomerative Clustering for classification.

Davies–Bouldin Index (DBI) and Silhouette Coefficient (SC) were used to assess the quality of the clustering results. Unsupervised clustering goes beyond descriptive statistics by uncovering latent usage profiles. Using DBI and SC provides quantitative evidence of cluster separation and compactness, supporting principled selection of K and more robust interpretations [32]. DBI and SC were computed using the reference implementations in *scikit-learn* for Python: `davies_bouldin_score`[8] and `silhouette_score`.[9] DBI measures the effectiveness of clustering for a given K value by averaging the similarity between each cluster and its most similar counterpart. SC evaluates the clustering structure using the silhouette method, which is defined as the difference between the average distance to elements in the nearest cluster and the average intra-cluster distance, divided by the maximum of these two distances.

[8] https://scikit-learn.org/stable/modules/generated/sklearn.metrics.davies_bouldin_score.html

[9] https://scikit-learn.org/stable/modules/generated/sklearn.metrics.silhouette_score.html

4.4. Dataset

The dataset used to validate the proposal was obtained from MIT on June 28th, 2021. It consists of 500,000 MIT App Inventor projects created by users, with a total size of 10.39 GB. The data is organized into numbered folders, each representing a project identified by a unique ID. Each project folder contains an *extraction.properties* file (empty due to privacy reasons) and two subfolders: *src* and *youngandroidproject*. The source code for each project's screens is located in *src/appinventor/masked/ProjectID*, where BKY files are stored. BKY files are XML representations of the blocks that implement the behavior of each screen. To ensure user privacy, project folders have been anonymized by replacing the original creator's name with the label "masked". Additional project-related information, such as the application version code, is stored in the *youngandroid* folder.

For the analysis, we excluded projects with fewer than 70 defined blocks across all screens. This threshold operationalizes prior work in App Inventor that models learners' "depth of capability" via program size/variety, such as the number of distinct block types used per project, showing that richer projects tend to reflect more advanced capability and progression [33]. In mobile, multi-screen, event-driven apps such as those built with App Inventor, projects naturally accumulate more blocks than in other VPLs as Scratch-like environments; thresholds suited to Scratch would therefore admit many tutorial-style fragments. We also considered that novices often overproduce blocks for simple behaviors, inflating counts without broader conceptual coverage; the 70-block cutoff mitigates this by ensuring a minimum functional substance so that static analysis and clustering capture meaningful behavioral patterns. Moreover, although our target population is novice programmers, we aim to extract stable usage patterns and generate actionable recommendations, which benefits from filtering out incomplete or trivial artifacts.

After this filtering step, the dataset used in the study comprised 215,244 projects. The BlocklyMining tool was then applied to extract relevant metrics, generating CSV and SQL files containing the analysis results.

The code of each project was statically analyzed to calculate the SQALE score, identifying potential code smells. The clustering algorithms were used to categorize the developed applications, based on an approach that selects the key metrics reflecting fundamental block-based programming concepts, such as functional blocks, event blocks, control structures or variable counts.

5. Experimentation

In this section, two case studies will be presented, in which the K-Means and Hierarchical Agglomerative algorithms will be applied to the dataset described in the previous section. For both algorithms, the manually selected metrics of the dataset will be used. The main objective of this analysis is to detect novice programmers' behavioral patterns, to provide them with personalized recommendations that can improve and strengthen their programming skills. Through this approach, it is intended to guide these programmers in their learning process, providing them with relevant and effective resources and

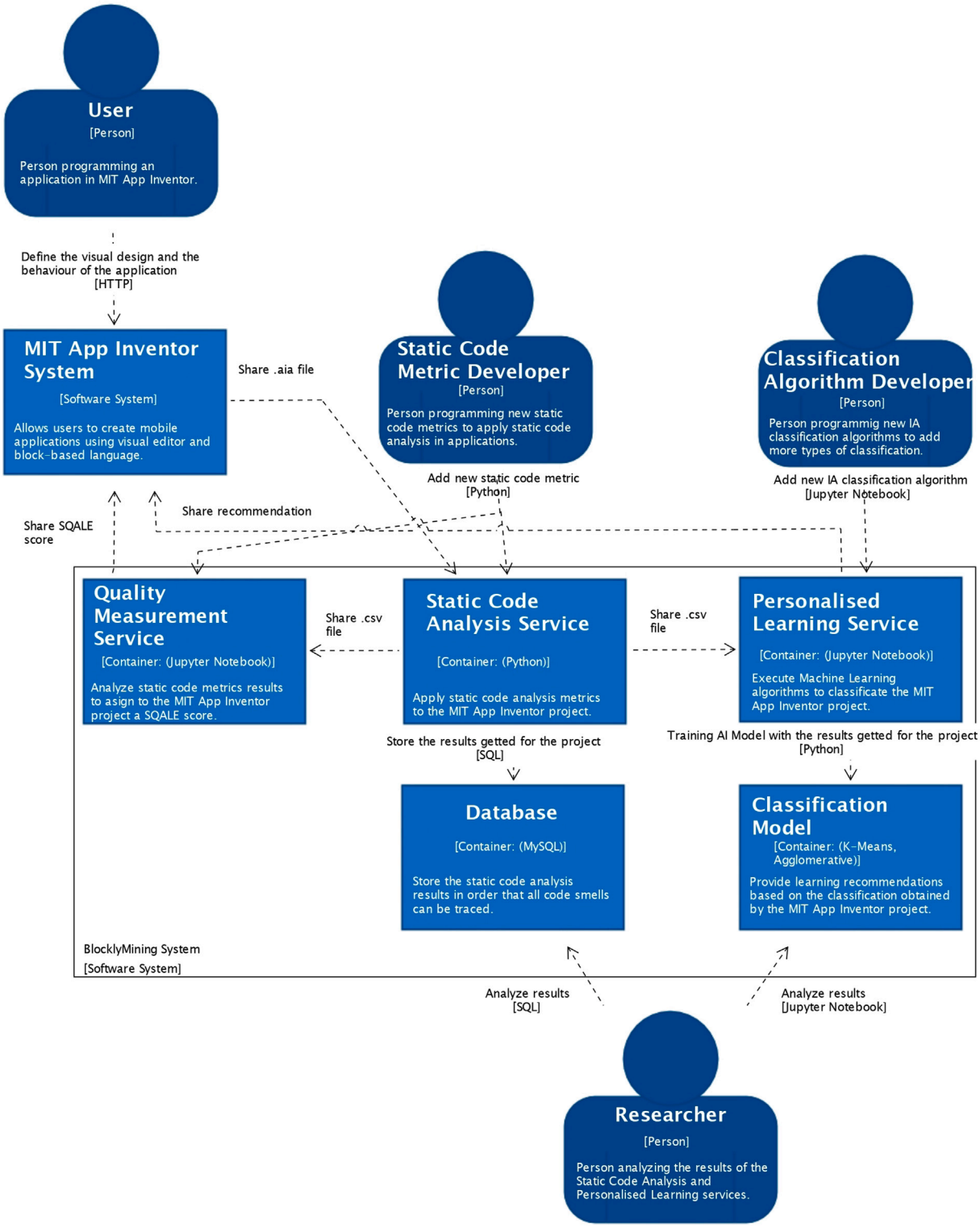

Fig. 2. Architecture diagram of the BlocklyMining tool.

exercises for their growth in programming. All preprocessing and analysis steps are documented in Appendix to reproduce BlocklyMining execution with any equivalent corpus of MIT App Inventor projects.

We executed BlocklyMining in *offline* batch mode over exported `.aia` projects. For clustering we used the manually selected element-based metrics M3-M9 (described in Table 3, Section 4). Because these features are non-negative counts, we applied z-score standardization to prevent scale bias across metrics; missing values did not occur (all metrics are computed deterministically from BKY files). We evaluated two unsupervised algorithms with Euclidean distance: K-Means (k-means++ initialization, `n_init`=20, `max_iter`=300, `random_state`=42) and Hierarchical Agglomerative Clustering with Ward linkage.

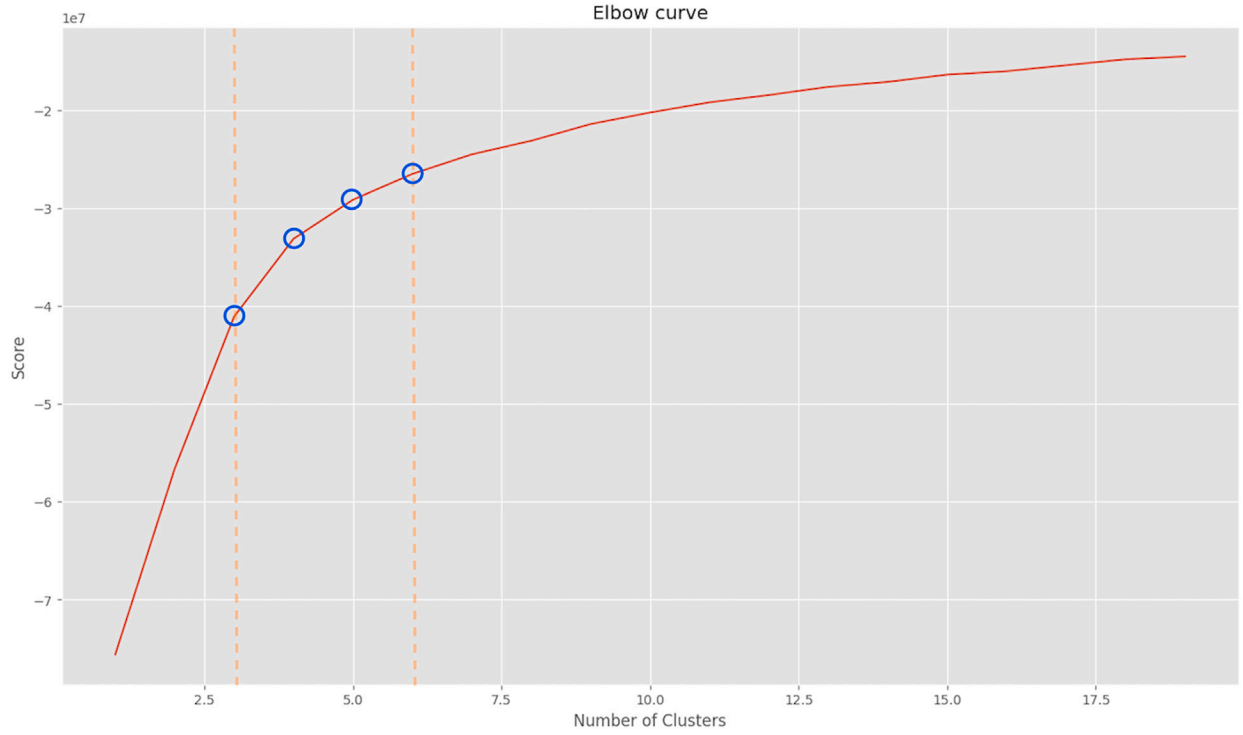

Fig. 3. Elbow curve of K-Means inertia across candidate K. The circled points (K = 3–6) mark the elbow region; $K \approx 3$ shows the largest marginal gain, with diminishing returns beyond $K \approx 6$. Dashed lines mark the considered range: $K \in [3, 6]$.

Table 4
Classification using the K-Means algorithm with K = 6 for the model based on manually selected metrics.

Cluster	*M3*	*M4*	*M5*	*M6*	*M7*	*M8*	*M9*
0	2.04	13.33	8.52	3.88	0.17	2.96	0.17
1	60.4	639.12	381.4	259.8	3.48	195.8	10.08
2	0.68	3.32	5.83	3.15	0.29	3.42	0.35
3	5.52	65.69	38.22	13.42	0.75	9.5	1.82
4	3.05	28.31	17.28	7.49	0.48	4.75	0.43
5	15.92	172.55	165.66	49.52	1.72	24.08	1.63

Table 5
Classification using the K-Means algorithm with K = 5 for the model based on manually selected metrics.

Cluster	*M3*	*M4*	*M5*	*M6*	*M7*	*M8*	*M9*
0	0.85	4.77	6.08	3.2	0.26	3.15	0.3
1	10.05	126.39	103.32	34.36	1.5	17.5	1.81
2	2.53	17.51	10.76	4.74	0.24	3.62	0.23
3	4.19	47.8	27.78	10.92	0.67	7.85	1.04
4	53.1	579.21	417.28	239.38	3	173.86	8.69

Table 6
Classification using the K-Means algorithm with K = 4 for the model based on manually selected metrics.

Cluster	*M3*	*M4*	*M5*	*M6*	*M7*	*M8*	*M9*
0	0.89	5.05	6.13	3.24	0.25	3.12	0.3
1	46.08	513.27	393.81	216.05	4.35	145.89	7.08
2	2.61	18.73	11.48	4.93	0.26	3.85	0.25
3	5.05	60.31	36.98	13.82	0.76	9.26	1.36

Table 7
Classification using the K-Means algorithm with K = 3 for the model based on manually selected metrics.

Cluster	*M3*	*M4*	*M5*	*M6*	*M7*	*M8*	*M9*
0	1.28	7.77	7.11	3.52	0.24	3.26	0.28
1	43.25	498.3	380.48	202.83	4.02	138.38	6.55
2	3.69	38.69	23.22	9.33	0.57	5.93	0.72

5.1. Clustering methodology and evaluation metrics

First, the Elbow curve shown in Fig. 3 is analyzed to determine the optimal value of K. This value determines the number of groups into which the data will be organized. We selected a candidate range for K using the Elbow method. The curve of K-Means inertia exhibits a clear inflection near $K \approx 3$, after which the marginal reduction in inertia drops sharply and the slope visibly flattens for K between 3 and 6. Practically, this corresponds to the first point where adding clusters yields rapidly diminishing returns (small first differences in inertia). We therefore treated $K \in [3, 6]$ as the candidate range to examine in subsequent analyses.

The centroids derived from the execution of the K-Means algorithm are described below for each cluster in Table 4 when considering K = 6, in Table 5 for K = 5, in Table 6 with K = 4, and in Table 7 when using K = 3.

The centroid values obtained for each group after running the Hierarchical Agglomerative algorithm are presented in Table 8 with K = 6, in Table 9 with K = 5, in Table 10 with K = 4, and in Table 11 with K = 3.

Next, Table 12 shows the distribution of projects in each group, considering all analyzed values of K. In addition, Table 13 provides the calculations that evaluate the quality of clusters for different values of K. The best results in both tables are highlighted in green. In this case, DBI and SC are used to evaluate the quality of the classifications obtained.

It is evident that the K-Means algorithm offers superior performance, evidenced by the optimal results obtained in terms of DBI and

Table 8
Classification using the Hierarchical Agglomerative algorithm with K = 6 for the model based on manually selected metrics.

Cluster	*M3*	*M4*	*M5*	*M6*	*M7*	*M8*	*M9*
0	55.92	594.96	426.41	245.15	3.22	182.22	9.33
1	1.64	10.77	8.31	3.4	0.13	2.33	0.16
2	4.74	59	35.27	15.85	0.77	11.65	1.56
3	2.61	25.23	16.48	6.93	0.42	4.87	0.38
4	0.73	2.61	4.75	3.33	0.39	4.22	0.42
5	16.41	187.56	134.15	39.4	1.83	21.33	1.66

Table 9
Classification using the Hierarchical Agglomerative algorithm with K = 5 for the model based on manually selected metrics.

Cluster	*M3*	*M4*	*M5*	*M6*	*M7*	*M8*	*M9*
0	5.22	64.31	39.36	16.82	0.82	12.05	1.56
1	1.64	10.77	8.31	3.4	0.13	2.33	0.16
2	55.92	594.96	426.41	245.15	3.22	182.22	9.33
3	2.61	25.23	16.48	6.93	0.42	4.87	0.38
4	0.73	2.61	4.75	3.33	0.39	4.22	0.42

Table 10
Classification using the Hierarchical Agglomerative algorithm with K = 4 for the model based on manually selected metrics.

Cluster	*M3*	*M4*	*M5*	*M6*	*M7*	*M8*	*M9*
0	1.25	7.28	6.78	3.37	0.24	3.14	0.28
1	5.22	64.31	39.36	16.82	0.82	12.05	1.56
2	55.92	594.96	426.41	245.15	3.22	182.22	9.33
3	2.61	25.23	16.48	6.93	0.42	4.87	0.38

Table 11
Classification using the Hierarchical Agglomerative algorithm with K = 3 for the model based on manually selected metrics.

Cluster	*M3*	*M4*	*M5*	*M6*	*M7*	*M8*	*M9*
0	3	30.97	19.84	8.38	0.48	5.92	0.56
1	1.25	7.28	6.78	3.37	0.24	3.14	0.28
2	55.92	594.96	426.41	245.15	3.22	182.22	9.33

SC coefficients. In particular, the most appropriate value of K is 3. However, it is noted that in the classification performed by K-Means with K = 3, the vast majority of the projects (93.23%) are grouped in cluster 0, while cluster 1 comprises a minimum number of projects (0.02%), and cluster 2 contains the remaining projects (6.75%).

In contrast, the choice of K = 4 in K-Means results in a more diverse grouping: cluster 0 contains 71.07% of the projects, cluster 1 only 0.02%, cluster 2 26.88% and cluster 3 the remaining 2.04%. This variation in the distribution of clusters is the reason why the classification obtained with K-Means and K = 4 is also examined.

5.2. K-Means (K=3)

An analysis of the classification results presented in Table 14 illustrates the formation of three clusters (0, 1 and 2). Cluster 1 stands out as having remarkable values for the metrics: M3 (*NumberOfDefinedFunctionalBlocks*), M4 (*NumberOfUsedFunctionalBlocks*), M5 (*NumberOfEventsBlocks*), M6 (*NumberOfConditionalBlocks*), M7 (*NumberOfLoopsBlocks*), M8 (*NumberOfGlobalVariables*) and M9 (*NumberOfLocalVariables*). In addition, two other clusters (0 and 2) were formed, which are similar in their characteristics, as they both have remarkable values for the metrics: M3 (*NumberOfDefinedFunctionalBlocks*), M4 (*NumberOfUsedFunctionalBlocks*), M5 (*NumberOfEventsBlocks*), M6 (*NumberOfConditionalBlocks*) and M8 (*NumberOfGlobalVariables*). The

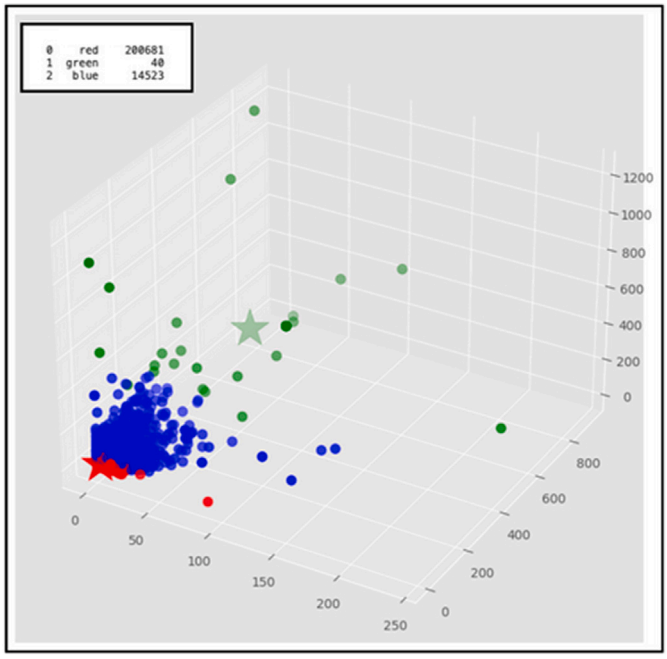

Fig. 4. Visual representation of the 3-cluster classification performed by the K-Means algorithm using manually selected metrics.

main difference between the two clusters (0 and 2) lies in the average values obtained for each of the metrics, noting that cluster 2 tends to have higher average values in all the metrics compared to cluster 0. The composition of these clusters is as follows: cluster 0 includes 200,681 projects (93.23%), cluster 1 consists of 40 projects (0.02%) and cluster 2 includes 14,523 projects (6.75%).

In addition, Fig. 4 depicts a graph illustrating how the projects are distributed in each of the clusters, providing an additional visual representation of the classification performed by the K-Means algorithm with a value of K = 3. In this figure, projects classified in cluster 0 are shown in red, projects classified in cluster 1 are shown in green, and projects classified in cluster 2 are shown in blue. The axes of the graph represent the three-dimensional position assigned by the K-Means algorithm to each of the projects according to the metrics selected for the study.

With this classification, it is possible to group projects into different levels according to the programming elements used in each of them. In this case, the process makes it possible to distinguish between initial levels, where users use fewer types of blocks, and more advanced levels, where they demonstrate a greater use of blocks. The levels can be defined as follows:

- Level 1: Projects that use all elements except loops and local variables (cluster 0).
- Level 2: Projects that use the same type of elements as Level 1 projects, but make greater use of these elements (cluster 2).
- Level 3: Projects that use the same elements as levels 1 and 2, including the use of loops and local variables (cluster 1).

Based on this information, it can be concluded that defining local variables and using loops are more complex tasks for novice MIT App Inventor programmers. It is important to note that cluster 1, which shows complete mastery of all elements, contains only 40 projects, indicating that the definition and use of local variables and the use of loops are less mastered skills by novice programmers compared to the use of other elements.

Table 12
Classification of K-Means and Hierarchical Agglomerative algorithms for different values of K, using manually selected metrics.

Algorithm	*K value*	*C0*	*C1*	*C2*	*C3*	*C4*	*C5*
K-Means	6	80.434	25	115.504	2.678	16.457	146
Hierarchical Agglomerative	6	27	109.615	3.314	20.092	82.053	143
K-Means	5	147.111	391	61.599	6.114	29	
Hierarchical Agglomerative	5	3.457	109.615	27	20.092	82.053	
K-Means	4	152.968	37	57.853	4.386		
Hierarchical Agglomerative	4	191.688	3.457	27	20.092		
K-Means	3	200.681	40	14.523			
Hierarchical Agglomerative	3	23.549	191.688	27			

Table 13
SC and DBI results for classification of K-Means and Hierarchical Agglomerative algorithms for different values of K, using manually selected metrics.

Algorithm	*K value*	*SC*	*DBI*
K-Means	6	0.29	1.07
Hierarchical Agglomerative	6	0.22	1.16
K-Means	5	0.37	0.99
Hierarchical Agglomerative	5	0.22	1.15
K-Means	4	0.39	0.94
Hierarchical Agglomerative	4	0.51	1
K-Means	3	0.66	0.80
Hierarchical Agglomerative	3	0.59	0.85

Table 14
Features per cluster created by K-Means algorithm and K = 3 using manually selected metrics.

Cluster	*M3*	*M4*	*M5*	*M6*	*M7*	*M8*	*M9*
0	1.28	7.77	7.11	3.52	0.24	3.26	0.28
1	43.25	498.3	380.48	202.83	4.02	138.38	6.55
2	3.69	38.69	23.22	9.33	0.57	5.93	0.72

The BPMN diagram shown in Fig. 5 details the recommendations made at each level, which is represented as a task or activity in the diagram. The connections between levels indicate the recommended flow of progress from one to the next. For example, to reach level 2, learners first complete level 1 and then address the task of intensifying the use of the programming elements embedded in the application. Concretely, for level 1, where loops and local variables are largely absent, the system recommends introducing iteration (e.g., a minimal "for each" pattern over a list) and refactoring globals into locals within event handlers, with a short before/after example. For level 2, where these constructs appear only sporadically, the guidance shifts to consolidation: expand the use of loop constructs across repeated patterns and systematically replace remaining globals with scoped locals, reinforcing best practices through small, in-place refactorings.

Finally, Fig. 6 provides a demonstration of the recommendation given to a user who has reached level 1. The recommendation includes a detailed explanation of how to create a new screen in MIT App Inventor application. This screen is set up as a place to integrate new functionality into the application, thereby increasing the use of additional programming elements. By offering this guide, novice programmers are guided toward developing more sophisticated applications that can incorporate a greater amount of functionality.

Table 15
Features per cluster created by K-Means algorithm and K = 4 using manually selected metrics.

Cluster	*M3*	*M4*	*M5*	*M6*	*M7*	*M8*	*M9*
0	0.89	5.05	6.13	3.24	0.25	3.12	0.3
1	46.08	513.27	393.81	216.05	4.35	145.89	7.08
2	2.61	18.73	11.48	4.93	0.26	3.85	0.25
3	5.05	60.31	36.98	13.82	0.76	9.26	1.36

5.3. K-Means (K=4)

An analysis of the classification results presented in Table 15 reveals the identification of 4 clusters that are significantly different from each other. Each of these clusters is characterized by particular values in the metrics considered. Cluster 0 stands out because it has values for the metrics: M4 (*NumberOfUsedFunctionalBlocks*), M5 (*NumberOfEventsBlocks*), M6 (*NumberOfConditionalBlocks*) and M8 (*NumberOfGlobalVariables*). Cluster 1 differs by presenting notable values for the metrics: M3 (*NumberOfDefinedFunctionalBlocks*), M4 (*NumberOfUsedFunctionalBlocks*), M5 (*NumberOfEventsBlocks*), M6 (*NumberOfConditionalBlocks*), M7 (*NumberOfLoopsBlocks*), M8 (*NumberOfGlobalVariables*), and M9 (*NumberOfLocalVariables*). Cluster 2 is notable for having remarkable values for the metrics: M3 (*NumberOfDefinedFunctionalBlocks*), M4 (*NumberOfUsedFunctionalBlocks*), M5 (*NumberOfEventsBlocks*), M6 (*NumberOfConditionalBlocks*), and M8 (*NumberOfGlobalVariables*). Finally, cluster 3 is characterized by significant values for the metrics: M3 (*NumberOfDefinedFunctionalBlocks*), M4 (*NumberOfUsedFunctionalBlocks*), M5 (*NumberOfEventsBlocks*), M6 (*NumberOfConditionalBlocks*), M8 (*NumberOfGlobalVariables*) and M9 (*NumberOfLocalVariables*).

The composition of these clusters is as follows: Cluster 0 contains 152,968 projects (71.07%), Cluster 1 contains 37 projects (0.02%), Cluster 2 contains 57,853 projects (26.88%), and Cluster 3 contains 4386 projects (2.04%).

In addition, Fig. 7 presents a graph representing the distribution of projects in each of the clusters, providing an additional visual representation of the classification performed by the K-Means algorithm with a value of K = 4. In this figure, the projects classified in cluster 0 are represented with red color, the projects classified in cluster 1 with green color, the projects classified in cluster 2 with blue color and the projects classified in cluster 3 with light blue color. The axes of the graph represent, as in the previous study, the three-dimensional position assigned by the K-Means algorithm to each of the projects according to the metrics selected for the study.

This classification divides the projects into different levels according to the programming elements used in each of them, similar to the classification previously studied with K = 3. In this way, it is possible to distinguish again between initial levels, where users use a smaller variety of blocks, and more advanced levels, where a wider use of blocks and programming functionalities can be observed. The levels can be characterized as follows:

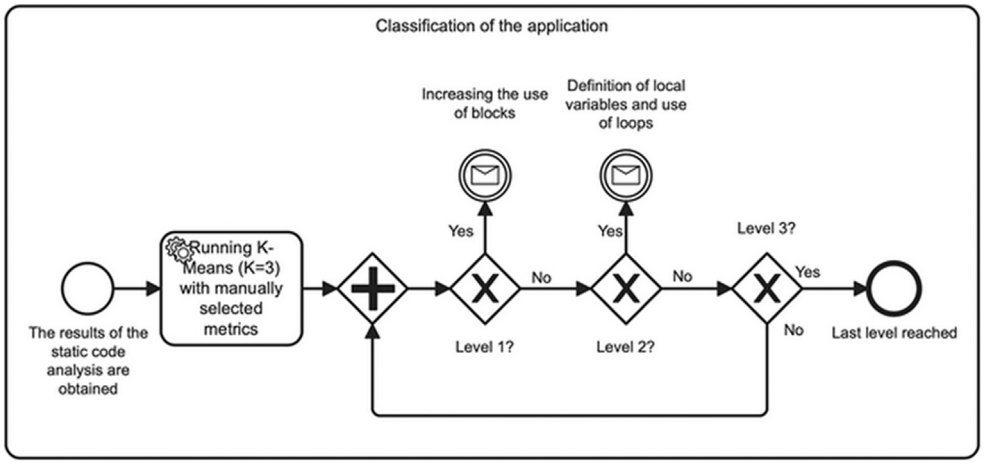

Fig. 5. Classification process and recommendations with K-Means (K = 3) using manually selected metrics.

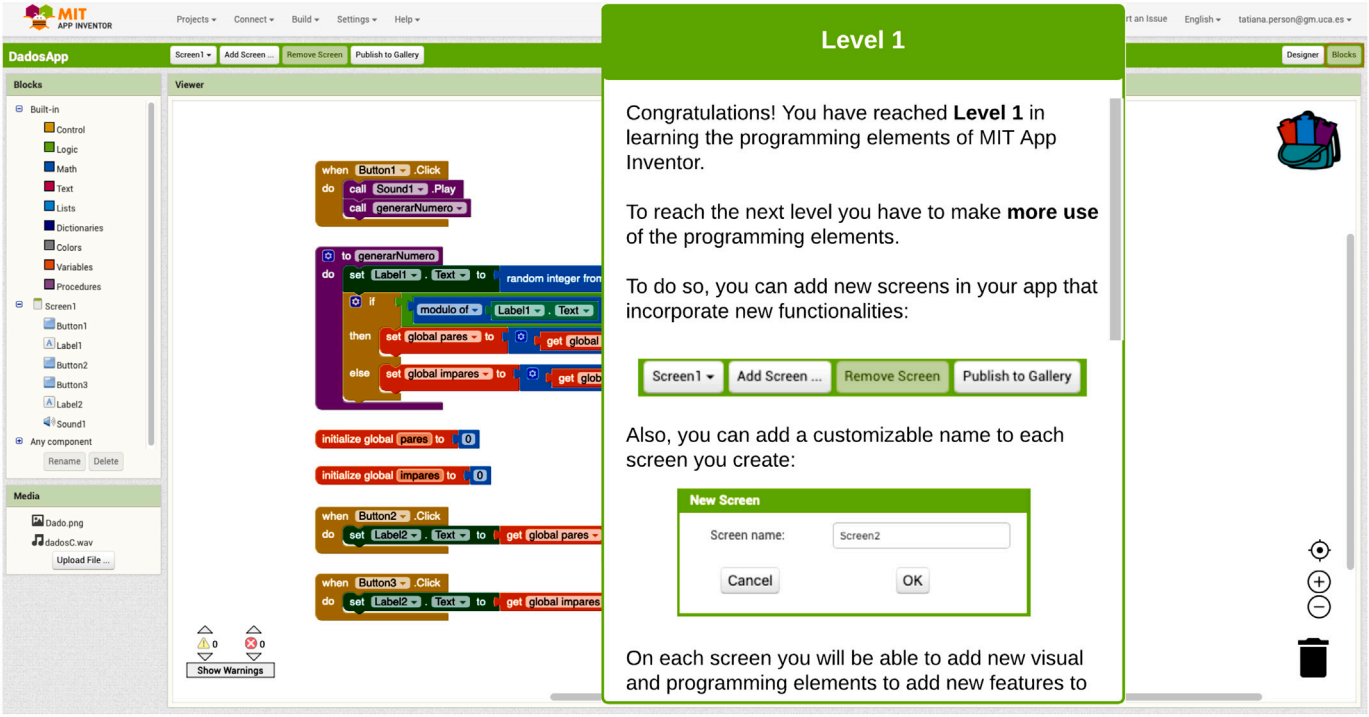

Fig. 6. Demonstration of project with level 1 recommendation in MIT App Inventor (K-Means with K = 3 using manually selected metrics).

- Level 1: Projects that use MIT App Inventor's predefined functions and events, use conditional blocks, and define global variables (cluster 0).
- Level 2: Projects with the same characteristics as the Level 1 projects, including their own user-defined functions (cluster 2).
- Level 3: Projects that have the same characteristics as Level 2 projects and use local variables (cluster 3).
- Level 4: Projects that include all characteristics of the previous levels and use loops (cluster 1).

From the extracted results, the most challenging activities for novice programmers in MIT App Inventor are, in terms of difficulty, the definition of functions, the definition of local variables and the use of loops. It is interesting to note that cluster 1, which shows full use of all these programming elements, contains only 37 projects. This figure suggests that the ability to use loops is less widely adopted by novice programmers compared to other skills, a finding that is consistent with the observations made in the previous classification with K = 3.

The BPMN diagram shown in Fig. 8 summarizes the personalized recommendations associated with each skill level. Fig. 9 illustrates an example provided to a user at level 3. This recommendation offers a clear and structured explanation of how to use the different types of loop constructs available in MIT App Inventor. By combining a textual description of each loop's purpose and behavior with a visual guide that highlights the corresponding blocks in the VPL interface, the system bridges the gap between conceptual understanding and practical implementation.

For levels 1 and 2, where gaps consistently appear in loops and local variables, recommendations prioritize introducing simple loop patterns (e.g., a minimal "for each" over a list) and refactoring globals into scoped locals within event handlers using brief before/after examples. In level 1 specifically, the recommendations system also addresses the lack of user-defined procedures by proposing a small procedure template to encapsulate repeated logic. For level 3, the focus narrows to iteration fluency: targeted loop templates and small in-place

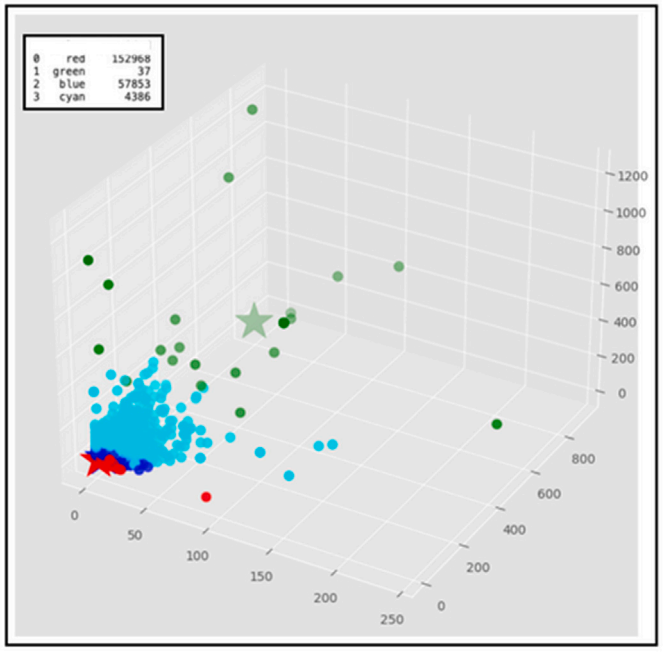

Fig. 7. Visual representation of the classification into 4 clusters performed by the K-Means algorithm using manually selected metrics.

refactorings that integrate loops into existing behaviors without changing the app's intent. These recommendations guide the user toward incorporating new programming constructs and reinforce learning by contextualizing abstract concepts in real, actionable examples within the user's development environment.

This example illustrates the underlying approach of the recommendation system: to deliver concrete, skill-based suggestions tailored to the user's programming level. These recommendations are not generic, but directly align with the concepts the user has not yet mastered. By integrating them within the development environment, the system promotes incremental learning and supports the acquisition of more complex programming skills in a guided and efficient manner.

5.4. Limitations and implications for learning

Our validation relies on large-scale static analysis and unsupervised clustering rather than user studies observing learning outcomes directly. As such, the evidence presented concerns patterns in code artifacts (e.g., concept usage and maintainability signals) rather than causal effects on learning. To keep the analysis reproducible, we report observable outcome proxies that the pipeline already exposes, such as profile transitions and metric deltas (e.g., increased use of loops and local variables; reductions in SQALE-derived technical debt) after project edits, while acknowledging that these indicators do not, by themselves, establish learning gains. In addition, in this case study, the gap-to-action mappings for generating new recommendations are prepared after each BlocklyMining editing session. Finally, any gap identified in prior work reflects the scope of our literature search and should be interpreted within those boundaries.

Despite these limits, the results have clear implications for personalized learning in block-based environments. Cluster profiles act as data-driven learner states; gap detection over these profiles pinpoints underused concepts; and the IDE-embedded messages operationalize next steps with concise explainers and block-level examples. This pipeline provides a practical mechanism to deliver adaptive guidance at scale and makes the pathway to summative evaluation explicit. Future studies could complement these proxies with classroom deployments and mixed-methods evidence to assess perceived usefulness and learning outcomes, but such work lies beyond the scope of the present analysis.

6. Conclusions and future work

Personalized learning in computer programming has long been a key objective in educational technology. Traditional instructional designs often fail to meet the diverse needs of individual students, presenting a significant challenge for novice programmers. Moreover, early programming education rarely emphasizes clean coding practices, which are essential for professional software development. Personalized learning allows students to progress at their own pace, enhances engagement, deepens understanding, and improves knowledge retention by leveraging data-driven insights for targeted instruction.

This paper presented a method for customizing programming education by analyzing large-scale datasets from block-based projects created by novice programmers. Block-based VPLs are popular for their visual representation and ease of use, which help beginners grasp complex concepts more intuitively compared to text-based languages with stricter syntax. The presented method combines learner data analysis with clean code metrics to examine patterns in programming concept coverage and code quality, thereby predicting and adapting learning pathways to better suit individual needs.

To our knowledge and within our search scope in Section 2, machine-learning-based clustering has not been applied to personalize learning in block-based environments. Considering that such environments aim at providing the freedom to create applications without predefined constraints, integrating machine learning to automatically generate recommendations based on individual proficiency is beneficial.

The proposed method follows an iterative DSR that enables novice programmers to refine their applications using the SQALE score to identify code issues and machine learning classification to detect areas for improvement. BlocklyMining is a tool developed to support this method through three core services, namely static code analysis, SQALE model-based quality evaluation and machine-learning-based applications' classification.

Validation of the research hypothesis–*the use of static code analysis, enhanced by machine learning techniques, provides a personalized learning experience for users of block-based VPLs*–was carried out on 215,244 MIT App Inventor projects, using K-Means and Hierarchical Agglomerative Clustering algorithms. Several iterations were performed with different numbers of clusters and several evaluation metrics.

The classification algorithms were applied to a selected set of metrics that capture fundamental block-based programming concepts. Consistent with the Elbow curve, which showed an inflection between $K = 3$ and $K = 6$, the optimal K and the best-performing algorithm were determined by evaluating clustering quality using SC and DBI. K-Means Clustering with $K = 3$ yielded the most favorable results (SC = 0.66, DBI = 0.80), although $K = 4$ was also considered due to the increased diversity of projects in some clusters (SC = 0.39, DBI = 0.94). This approach prioritizes interpretability, as each selected metric directly corresponds to specific programming concepts, facilitating the understanding of the clustering-based classification. The method effectively groups projects into meaningful clusters, enabling the generation of personalized learning recommendations based on the skill levels demonstrated by novice programmers.

Our method explores the use of machine learning for personalized learning in block-based VPL environments offering an adaptive mechanism to guide novice programmers in enhancing both their code quality and conceptual understanding. Through an iterative process, the approach provides targeted feedback that enables users to progressively refine their applications while mastering programming concepts. This systematic and automated approach not only improves learning experiences, but also establishes a scalable foundation for implementing personalization in block-based VPLs.

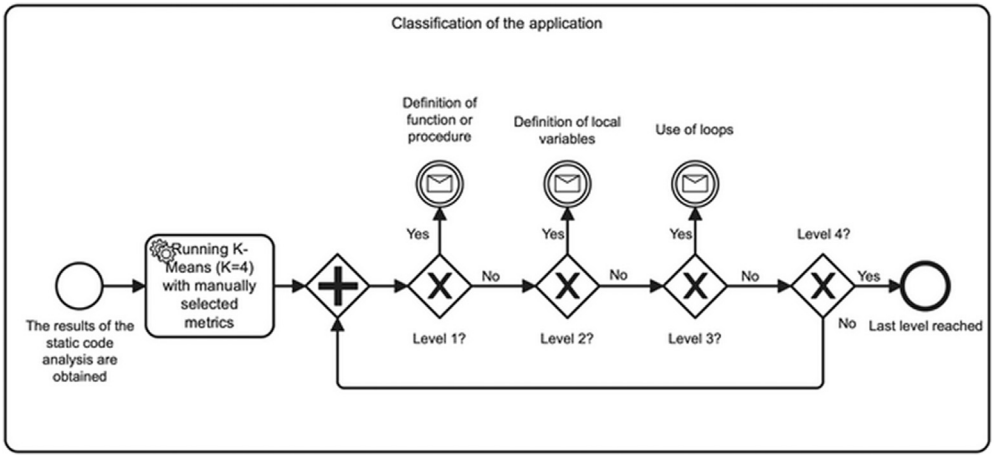

Fig. 8. Classification process and suggestions with K-Means (K = 4) using manually selected metrics.

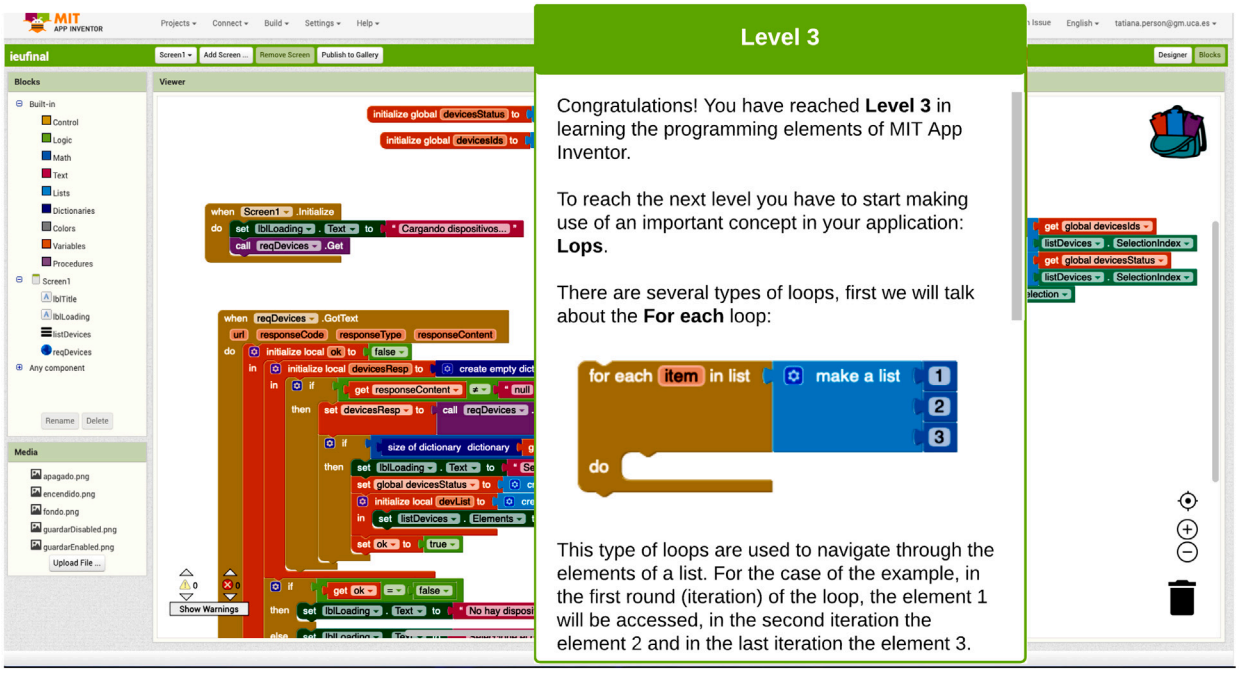

Fig. 9. Demonstration of project with level 3 recommendation in MIT App Inventor (K-Means with K = 4 using manually selected metrics).

As future work, the combination of BlocklyMining and MIT App Inventor is planned in order to provide visual suggestions and adaptions to programmers' learning process seamlessly integrated within the platform. Moreover, it can be used to evaluate user projects and assign grades based on SQALE and static code analysis results.

Future work could also explore how visual programming environments like MIT App Inventor might adapt to the growing influence of Large Language Models (LLMs) and prompt-based programming tools. LLMs could be used to automatically generate tailored feedback, suggest next steps based on the user's current project, or provide natural language explanations of programming concepts. This approach could go beyond predefined instructional templates, offering more flexible, real-time guidance adapted to each learner's progress and style. Combining the advantages of block-based programming with AI-assisted support could open new opportunities for personalized learning, while maintaining the accessibility that these platforms offer to novice users.

CRediT authorship contribution statement

Tatiana Person: Writing – original draft, Visualization, Software, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Juan Antonio Caballero-Hernández:** Writing – review & editing, Writing – original draft, Investigation. **Cristóbal Romero:** Writing – review & editing, Validation, Supervision. **Iván Ruiz-Rube:** Writing – review & editing, Validation, Supervision, Software, Project administration, Methodology, Conceptualization. **Juan Manuel Dodero:** Writing – review & editing, Supervision, Project administration, Funding acquisition.

Declaration of competing interest

The authors have declared no conflict of interest

Acknowledgments

This publication is part of the R&D&i Project PID2023-149674OB-I00 (GENIALLE), funded by MICIU/AEI/10.13039/501100011033 and ERDF, EU.

Appendix. Blocklymining installation

Environment. Python 3.10

1. **Clone the repository**

```
git clone https://github.com/TatyPerson/
BlocklyMining.git
cd BlocklyMining
```

2. **Create and activate a virtual environment**

```
python3.10 -m venv .venv
source .venv/bin/activate
```

3. **Install dependencies**

```
python -m pip install --upgrade pip
python -m pip install -r requirements.txt
```

Minimal `requirements.txt` (if not present):

```
mysql-connector-python==9.5.0
numpy==2.2.6
```

4. **Prepare the dataset**

   - Export MIT App Inventor projects as `.aia` files.
   - Unzip each `.aia` into its own subfolder, preserving the internal structure (e.g., `src/.../*.bky`, `youngandroidproject/...`).
   - (Optional) Batch unzip example (bash):

```
for f in *.aia; do unzip -q
"$f" -d "${f
```

   - Place all extracted project folders under a single directory, e.g., `/projects`.

5. **Run batch analysis (CSV output)**

```
python source/BlocklyMining.py --source
=appinventor --database=none --path
=/projects
```

The tool parses BKY (XML) and writes CSV results into `/projects`.

6. **(Optional) Persist results in MySQL**

   (a) Create the schema from `database-schema/BlocklyMiningDatabase.sql`.

   (b) Edit `source/ConfigurationParameters.cfg`:

```
[MYSQL_DATABASE]
MYSQL_HOST = localhost
MYSQL_USER = user
MYSQL_PASSWORD = password
MYSQL_DATABASE = BlocklyMiningDatabase
```

   (c) Run with MySQL mode:

```
python source/BlocklyMining.py --source
=appinventor --database=mysql --path
=/projects
```

References

[1] Wing JM. Computational thinking. Commun ACM 2006;49(3):33–5. http://dx.doi.org/10.1145/1118178.1118215.

[2] Güven I, Gulbahar Y. Integrating computational thinking into social studies. Soc Stud 2020;111(5):234–48. http://dx.doi.org/10.1080/00377996.2020.1749017.

[3] Rodríguez Corral JM, Ruíz-Rube I, Civit Balcells A, Mota-Macías JM, Morgado-Estévez A, Dodero JM. A study on the suitability of visual languages for non-expert robot programmers. IEEE Access 2019;7:17535–50. http://dx.doi.org/10.1109/ACCESS.2019.2895913.

[4] Kuhail MA, Farooq S, Hammad R, Bahja M. Characterizing visual programming approaches for end-user developers: A systematic review. IEEE Access 2021;9:14181–202. http://dx.doi.org/10.1109/ACCESS.2021.3051043.

[5] Bernacki ML, Greene MJ, Lobczowski NG. A systematic review of research on personalized learning: Personalized by whom, to what, how, and for what purpose(s)? Educ Psychol Rev 2021;33:1675–715. http://dx.doi.org/10.1007/s10648-021-09615-8.

[6] Harry A. Role of AI in education. Interdicip J & Hummanity (INJURITY) 2023;2(3).

[7] Dhananjaya GM, Goudar RH, Kulkarni AA, Rathod VN, Hukkeri G. A digital recommendation system for personalized learning to enhance online education: A review. IEEE Access 2024;12:34019–41. http://dx.doi.org/10.1109/ACCESS.2024.3369901.

[8] Gu P, Ma J, Chen W, Deng L, Jiang L. A personalized learning strategy recommendation approach for programming learning. In: Bao Z, Trajcevski G, Chang L, Hua W, editors. Database systems for advanced applications. Cham: Springer International Publishing; 2017, p. 267–74.

[9] Trisovic A, Lau MK, Pasquier T, Crosas M. A large-scale study on research code quality and execution. Sci Data 2022;9(1):60.

[10] Sözer H. Integrated static code analysis and runtime verification. Softw: Pr Exp 2015;45(10):1359–73. http://dx.doi.org/10.1002/spe.2287.

[11] Bhattacharya P, Guo M, Tao L, Fu Y, Qian K. A collaborative interactive cyber-learning platform for anywhere anytime java programming learning. In: 2011 IEEE 11th international conference on advanced learning technologies. 2011, p. 14–6. http://dx.doi.org/10.1109/ICALT.2011.12.

[12] McGill MM. Learning to program with personal robots: Influences on student motivation. ACM Trans Comput Educ 2012;12(1). http://dx.doi.org/10.1145/2133797.2133801.

[13] Leigh E, Spindler L. Simulations and games as chaordic learning contexts. Simul Gaming 2004;35(1):53–69. http://dx.doi.org/10.1177/1046878103252886.

[14] Raj NS, Renumol V. A systematic literature review on adaptive content recommenders in personalized learning environments from 2015 to 2020. J Comput Educ 2022;9(1):113–48.

[15] George G, Lal AM. Review of ontology-based recommender systems in e-learning. Comput Educ 2019;142:103642. http://dx.doi.org/10.1016/j.compedu.2019.103642.

[16] Lesan Sedgh MM, Latif A, Emadi S. A novel method for a technology enhanced learning recommender system considering changing user interest based on neural collaborative filtering. Data Sci Manag 2025;8(2):196–206. http://dx.doi.org/10.1016/j.dsm.2024.09.004.

[17] Erdt M, Fernández A, Rensing C. Evaluating recommender systems for technology enhanced learning: A quantitative survey. IEEE Trans Learn Technol 2015;8(4):326–44. http://dx.doi.org/10.1109/TLT.2015.2438867.

[18] Park Y, Shin Y. Comparing the effectiveness of scratch and app inventor with regard to learning computational thinking concepts. Electronics 2019;8(11). http://dx.doi.org/10.3390/electronics8111269.

[19] Cárdenas-Cobo J, Puris A, Novoa-Hernández P, Parra-Jiménez Á, Moreno-León J, Benavides D. Using scratch to improve learning programming in college students: A positive experience from a non-WEIRD country. Electronics 2021;10(10). http://dx.doi.org/10.3390/electronics10101180.

[20] Johnson SC. Lint, a C program checker. Bell Telephone Laboratories Murray Hill; 1977.

[21] Moser R, Pedrycz W, Succi G. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In: Proceedings of the 30th international conference on software engineering. ICSE '08, New York, NY, USA: Association for Computing Machinery; 2008, p. 181–90. http://dx.doi.org/10.1145/1368088.1368114.

[22] Sarıman G, Kucuksille EU. A novel approach to determine software security level using Bayes classifier via static code metrics. Elektron Ir Elektrotechnika 2016;22(2):73–80. http://dx.doi.org/10.5755/j01.eie.22.2.12177.

[23] Letouzey J-L, Ilkiewicz M. Managing technical debt with the SQALE method. IEEE Softw 2012;29(6):44–51. http://dx.doi.org/10.1109/MS.2012.129.

[24] Sadowski C, Aftandilian E, Eagle A, Miller-Cushon L, Jaspan C. Lessons from building static analysis tools at Google. Commun ACM 2018;61(4):58–66. http://dx.doi.org/10.1145/3188720.

[25] Jurayev TN. The use of mobile learning applications in higher education institutes. Adv Mob Learn Educ Res 2023;3:610–20. http://dx.doi.org/10.25082/amler.2023.01.010.

[26] Gueye ML, Expósito E. Education 4.0: Proposal of a model for autonomous management of learning processes. In: Service-oriented computing – ICSOC 2022 workshops. 2023, p. 106–17. http://dx.doi.org/10.1007/978-3-031-26507-5_9.
[27] Gerasimova E, Zorin SS, Kobeleva GA, Mamaeva EA. Designing a personalized educational model while working with digital technologies. P Sci Edu 2020;47:398–412. http://dx.doi.org/10.32744/pse.2020.5.28.
[28] Katchapakirin K, Anutariya C, Supnithi T. ScratchThAI: A conversation-based learning support framework for computational thinking development. Educ Inf Technol 2022;27:8533–60. http://dx.doi.org/10.1007/s10639-021-10870-z.
[29] Unahalekhaka A, Bers MU. Taking coding home: analysis of scratchjr usage in home and school settings. Educ Technol Res Dev 2021;69:1579–98. http://dx.doi.org/10.1007/s11423-021-10011-w.
[30] Peffers K, Tuunanen T, Rothenberger MA, Chatterjee S. A design science research methodology for information systems research. J Manage Inf Syst 2007;24(3):45–77. http://dx.doi.org/10.2753/MIS0742-1222240302.
[31] Brown S. The C4 model for visualising software architecture. Leanpub; 2023.
[32] Chicco D, Campagner A, Spagnolo A, Ciucci D, Jurman G. The Silhouette coefficient and the Davies-Bouldin index are more informative than Dunn index, Calinski-Harabasz index, Shannon entropy, and Gap statistic for unsupervised clustering internal evaluation of two convex clusters. PeerJ Comput Sci 2025;11:e3309. http://dx.doi.org/10.7717/peerj-cs.3309.
[33] Xie B, Abelson H. Skill progression in MIT app inventor. In: 2016 IEEE symposium on visual languages and human-centric computing (VL/HCC). 2016, p. 213–7. http://dx.doi.org/10.1109/VLHCC.2016.7739687.