applied sciences

MDPI

Article

Source Code Analysis in Programming Education: Evaluating Learning Content with Self-Organizing Maps

Marko Jevtić, Saša Mladenović and Andrina Granić *

Faculty of Science, University of Split, Ulica Ruđera Boškovića 33, 21000 Split, Croatia; marko.jevtic@pmfst.hr (M.J.); sasa.mladenovic@pmfst.hr (S.M.)
* Correspondence: andrina.granic@pmfst.hr

Abstract: Due to the everchanging and evergrowing nature of programming technologies, the gap between the programming industry's needs and the educational capabilities of both formal and informal educational environments has never been wider. However, the need to learn computer programming has never been greater, regardless of the motivation behind it. The number of programming concepts to be taught is increasing over time, while the amount of time available for education and training usually remains the same. The objective of this research was to analyze the source codes used in many educational systems to teach fundamental programming concepts to learners, regardless of their prior experience in programming. A total of 25 repositories containing 3882 Python modules were collected for the analysis. Through self-organization of the collected content, we obtained very compelling results about code structure, distribution, and differences. Based on those results, we concluded that Self-Organizing Maps are a powerful tool for both content and knowledge management, because they can highlight problems in the curriculum's density as well as transparently indicate which programming concepts it has successfully observed and learned to recognize. Based on the level of transparency exhibited by Self-Organizing Maps, it is safe to say that they could be used in future research to enhance both human and machine learning of computer programming. By achieving this level of transparency, such an Artificial Intelligence system would be able to assist in overall computer programming education by communicating what should be taught, what needs to be learned, and what is known.

Keywords: content management; computer programming education; source code analysis; self organizing maps; artificial intelligence; neural networks

Citation: Jevtić, M.; Mladenović, S.; Granić, A. Source Code Analysis in Programming Education: Evaluating Learning Content with Self-Organizing Maps. *Appl. Sci.* **2023**, *13*, 5719. https://doi.org/10.3390/app13095719

Academic Editor: Andrea Prati

Received: 27 March 2023
Revised: 28 April 2023
Accepted: 1 May 2023
Published: 5 May 2023

1. Introduction

The Fourth Industrial Revolution is anticipated to foster a rise in skills gaps throughout diverse industries. The relentless and expeditious progressions in artificial intelligence (AI), robotics, and other nascent technologies have led to a paradigm shift in the job market, necessitating a rapid transformation in the skills essential to perform them [1].

Before the pandemic, a degree alone was insufficient to secure a good job. The pandemic has caused a shift in recruitment patterns. The "new normal" has led to companies seeking individuals with strong technical and soft skills, problem-solving abilities, creativity, confidence, excellent communication skills, good academic records, and innovative thinking. As a result, there is a high demand for these skills to get a new or safeguard an existing job, emphasizing the need to acquire these competencies to enhance future prospects [2].

Many recent software engineering graduates often face difficulties when beginning their professional careers due to a misalignment of skills learned during their formal education at the university level with what is expected and needed in the industry [3–5].

A literature search suggests that Java and Python are the most commonly used programming languages for teaching at the undergraduate level. A significant percentage

(88%) of surveyed schools use one of four languages (Java, Python, C++, and C) for their first two programming courses, with Python growing in popularity [6]. Java remains the most popular choice for second programming courses. At the same time, research has suggested that scripting languages like Python may be more suitable for teaching novices than system programming languages like Java and C++ [7].

One way to cope with this trend could be to continuously learn and stay up-to-date with the latest technologies and tools in the industry. This can be achieved through various means such as taking online courses, attending conferences and workshops, and participating in online communities.

As time progresses, the trend is clear: the quantity of programming-related concepts that need to be taught tends to increase, while the duration of education and training typically remains constant.

In formal education, it is possible that this could lead to a more condensed curriculum and a need for more efficient teaching methods [8,9].

Concentrating on establishing a solid understanding of fundamental programming concepts can aid in the adaptation to new technologies and tools with greater ease.

Students can establish a solid understanding of fundamental programming concepts by studying computer science fundamentals, such as data structures, algorithms, programming languages, and software engineering principles. They can also practice coding exercises and work on programming projects to reinforce their knowledge and skills.

Additionally, during the onboarding procedure in firms, new employees can seek guidance and feedback from experienced programmers or they can give and ask for help by participating in coding communities to further solidify their understanding of new concepts.

The final product of programming is the source code, based on programmers' mental models of programs [10], which can be used to analyze the concepts employed in the program and provide feedback to curriculum designers. This study aims to examine the source codes utilized in numerous educational systems for teaching fundamental programming concepts to beginners without prior programming experience. Self-Organizing Maps are a valuable tool for managing both content and knowledge, as they can identify issues in curriculum density and clearly indicate which programming concepts have been effectively recognized and learned.

The primary contribution of this research is to demonstrate that Self-Organizing Maps (SOMs) can serve as a valuable tool for identifying issues related to curriculum density. The findings indicate that SOMs can effectively pinpoint areas in the existing curriculum that require modifications to enhance student learning in programming. Furthermore, the application of objective methods, such as SOMs, introduces transparency in the evaluation of curricula, which is typically guided by grades and student perceptions.

2. Background

This section presents the outcomes of the literature review.

2.1. Literature Review

The study conducted a literature search utilizing three specific search terms, namely "self-organizing maps", "programming", and "learning". The most relevant citation databases in the research field, namely the Web of Science, Scopus, and IEEEXplore, were employed for this purpose. Through a systematic analysis of the abstracts retrieved, the literature search revealed a body of literature that indicated the potential of self-organizing maps (SOMs) to facilitate the effective recognition and acquisition of programming concepts.

The literature highlights the flexible applications of self-organizing maps (SOMs) in various fields. In the domain of education, SOMs have been used in intelligent tutoring systems to provide tailored tuition to learners based on their preferred learning styles [11]. Furthermore, SOMs have demonstrated the potential to identify important elements of object-oriented programming, such as classes and objects, by detecting common features

in software code [12]. SOMs can also aid in the analysis of completed programming code, such as computer games, to identify the development of computational thinking skills [13].

SOMs have also been used in different fields, such as student clustering based on academic grades [14] and course recommendations for e-learning systems [15]. Additionally, SOMs have been used to group procedures with similar properties by identifying common features in software code [12,16]. Furthermore, a study demonstrated that the variation of SOMs can identify algorithms implemented as programs by converting source code into syntax trees and computing similarities between them [17]. In conclusion, the studies suggest that SOMs can effectively analyze and cluster programming code, making them a valuable tool in this domain.

2.2. Research Question

The aim of this research was to use an SOM to give an answer to our research question of whether or not a self-organization machine learning technique can be used to achieve transparency. The SOM is just a representative of that technique.

The goal was set to reach the level of transparency in which each neuron of the trained SOM represents at least one programming concept OR a class of programming concepts. The purpose of the training was to categorize source codes and cluster them based on their structure and complexity.

2.3. Google Colab

To run our experiment, we used the Google Colab collaborative coding tool. It ensures cross-team collaboration and consistency in results reproduction as well as effortless management of available computer resources.

2.4. GitHub

We used GitHub as the source of training data. GitHub is an internet service for open-source code hosting and version control and is currently the largest source code host [18].

2.5. Abstract Syntax Tree

To eliminate subjective differences in source code, we used the Abstract Syntax Tree (AST) to represent the programming concepts used in an objective and fully automated manner. An AST is a tree data structure representing the abstract syntactic structure of the source code written in a formal language [19], which was Python in our case [20].

2.6. Self-Organizing Map

A self-organizing map (SOM) is a neural network used for dimensionality reduction that typically reduces higher dimensional data into a two-dimensional representation. Unlike many other artificial neural networks, SOM uses competitive learning, rather than error-correction learning, during training [21].

The following formula shows the weight calculation for each SOM neuron [22]:

$$W_v(s+1) = W_v(s) + \theta(u, v, s) * \alpha(s) * (D(t) - W_v(s))$$

where:

- W_v is the current weight vector of node v
- s is the current iteration
- u is the index of the best matching unit (BMU) in the map
- v is the index of the node in the map
- $\theta(u, v, s)$ is the neighboring function used to restrain learning based on the distance
- $\alpha(s)$ is a learning decline due to iteration progress
- t is the index of the target input data vector in the input data set D
- $D(t)$ is a target input data vector

The quantization error is used to measure the distance between the final SOM value and the original input value. It is calculated with the following formula [23,24]:

$$QE = 1/N\Sigma_{i=1}^{N}||\phi(x_i) - x_i||$$

where N is the number of input vectors and $\phi : D \mapsto M$ is the mapping from the input space D to the SOM M.

The topographic error measures how well individual data points are mapped to the SOM's nodes following the formula [23,24]:

$$TE = 1/N\Sigma_{i=1}^{N}t(x_i)$$

$$t(x) = \begin{cases} 0 & \text{if } \mu(x) \text{ and } \mu'(x) \text{ are neighbors} \\ 1 & \text{otherwise} \end{cases}$$

$\mu(x)$ and $\mu'(x)$ are the best-matching unit and the second-best matching unit, respectively. The process of training a SOM, presented in Figure 1, consists of these four steps:

1. Initialization—a vector of the same dimensionality as the input vector is assigned to each neuron.
2. Competition—SOM nodes compete, and the best matching unit is considered a winner. The winner node's vector values are updated to learn from the input vector and get closer to it.
3. Cooperation—The winner node cooperates with its neighbours to propagate the change it underwent.
4. Adaptation—Neighboring nodes adapt to the change on the winner node, but the magnitude of this change decreases with time and grid distance from the winner node.

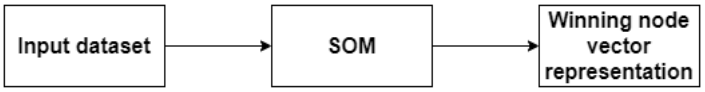

Figure 1. The process of SOM utilization.

The algorithm for the last three steps of the training process can be viewed in Figure 2.

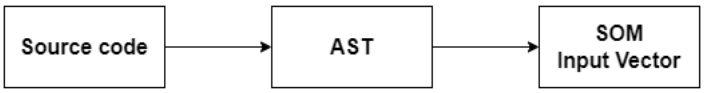

Figure 2. The process of source code transformation with the input vector for the SOM network.

The benefit of using a SOM is that it keeps the structural information of the training data set (programming code) intact. The main drawback is that a value for each dimension of the input vector is needed for each member of the input data set in order to generate a SOM. Since we provide either 0 or 1 for each dimension, this drawback does not affect our research. Another problem is that every SOM is different and finds different similarities among the sample vectors.

Thus far, we have not found any references in research on this type of source code analysis with a focus on the extraction of the programming concepts from the content. SOMs are typically used for visualization and exploratory data analysis [25–27], but they are rarely, if ever, used on the source code.

3. Methodology

3.1. Data Collection

By filtering data on the GitHub web page, our focus was on source codes written in Python programming language. Since Python is a general-purpose language, we had to further narrow our search by defining the categories of interest, which are

- Python project
- Problem-solving
- Object-oriented programming
- Artificial Intelligence
- Data Science
- Python tutorial

These categories cover three levels of programming education, from introductory programming (Python tutorial, Problem-solving) to intermediate programming (Python project, Object-oriented programming) and narrow domain programming (Artificial Intelligence, Data Science).

Even though we relied on GitHub to be the source of our training data, since it can provide a large data set of open data, we set our focus on a rather closed subset of narrow-domain programming code, focusing on topics of educational interest, topics which are currently taught at many universities worldwide, including the Faculty of Science at the University of Split.

Once cloned, code repositories were organized into directories named after the aforementioned categories. Since some code repositories contain non-Python source code, it was imperative to extract only the code that can be parsed by Python's native AST parser, i.e., *.py and *.ipynb source code files.

3.2. Data Preparation

The process of data collection and preparation was performed in accordance with the methodology described in Algorithm 1. Prior to organizing the collected data using a Self-Organizing Map (SOM), the data was parsed and transformed into a binary vector utilizing a custom Python Abstract Syntax Tree (AST) Transformer

Algorithm 1 Datacollection and preparation

```
git clone the selected repository
for all *.py and *.ipynb files within the cloned repository do
    if file has *.ipynb extension then
        convert the file to Python source code
    end if
    convert Python source code into the AST representation using the native Python AST parser
    parse the AST to form the SOM input vector
end for
add metadata (source repository, category, number of files)
```

The first problem we had to solve was to determine the best vector representation of the collected source code. Initially, the vector consisted of frequency values for each Python AST node. However, such a metric did not represent the complexity of the source code due to the fact that the large quantity of a single AST node in a source code does not indicate a more complex code. As an alternative, a binary vector was used to aggregate the indicator values, representing the existence or lack of an AST node within a source code.

The prepared data set consists only of syntactically correct Python source code, which was ensured due to the fact that the AST transformer would not be able to parse an incorrect syntax tree.

Given that there are n available AST nodes within the *ast* native Python module at any time, the resulting vector would be

$$u = (u_0, ..., u_{n-1}) \in \{0, 1\}^n$$

3.3. Data Analysis

The vector produced in the subsequent step served as input data to the SOM network. Table 1 shows the configuration settings of our SOM network.

Table 1. SOM configuration.

Configuration Field	Configuration Value	Explanation
Number of columns	15	Number of weeks within a semester.
Number of rows	10	Number of semesters in the entire CS program at the Faculty of Science.
Input length	114	Number of Python's AST nodes.
Neighborhood function	Gaussian	Gaussian neighborhood function was used to weigh the neighborhood of the winner node.
Sigma	1.5	Spread of the neighborhood function.
Learning rate	0.5	Initial learning rate.
Activation distance	Euclidean	The Euclidean distance was used to make a decision on which neurons are activated during each step of the training process.

The best matching unit using the Euclidean distance is determined by finding the neuron or unit in a neural network that has the smallest Euclidean distance to a given input vector.

The Euclidean distance between two vectors is the square root of the sum of the squared differences between their corresponding elements.

For example, suppose we have a neural network with three output units and we want to find the best matching unit for an input vector $x = (x_1, x_2, x_3)$, we first calculate the Euclidean distance between x and the weight vectors of each output unit $w_1 = [w_{i_1}, w_{i_2}, w_{i_3}]$, where $i = 1, 2, 3$. The Euclidean distance between x and w_i is given by $d_i = \sqrt{\sum_{j=1}^{3}(x_j - w_{i,j})^2}$ We then select the output unit with the smallest Euclidean distance as the best matching unit, that is the unit i that minimizes d_i. $Bestmatchingunit = argmin_i d_i$.

The concept of the Euclidean distance is used to group the features according to the best neuron or winning neuron with the Best Matching Unit (BMU) approach.

Once the input vectors have been produced, the training of the SOM begins and lasts until all training data have been utilized. The SOM training process is elaborated in Algorithm 2. The result of the training is a rectangular grid of nodes or neurons, where each neuron is assigned a weight vector W_s.

Algorithm 2 The SOM training process

```
for all vectors in the training data set do
    Provide the vector as input to the SOM
    Find the best matching unit within the SOM
    Find the neighbors of the best matching unit
    Update the best matching unit and its neighbors
end for
```

3.4. SOM Test and Evaluation

To test the SOM and evaluate its results, we used the data set compiled at the Faculty of Science, University of Split, in the scope of the introductory-level programming course where we used Python programming language. Since our teaching program has been re-accredited and the course has been declared as the core or compulsory course, the validity of the test data set has been proven.

Appendixes A.1–A.3 show examples of source codes used for the testing and evaluation of the SOM results.

4. Results

Table 2 shows the amount of cloned GitHub repositories and the amount of parsed Python modules for each category. Additionally, Table 3 showcases the results of training data aggregations.

Table 2. Quantity of collected data.

Code Category	Number of Repositories	Number of Python Modules
Python project	2	153
Problem solving	2	113
Object-oriented programming	7	297
AI	7	2306
Data science	2	179
Python tutorial	5	834

Table 3. Results of training data aggregations.

	Tree Depth				Node Count			
Category	**Min**	**Max**	**Mean**	**Median**	**Min**	**Max**	**Mean**	**Median**
AI	1	119	10.532	11.0	1	17307	919.138	534.5
Data Science	1	22	8.849	10.0	1	8893	787.888	581.0
OOP	1	22	9.252	9.0	1	253,206	1613.66	124.0
Problem solving	6	18	11.283	12.0	15	690	187.752	152.0
Python project	1	17	8.071	9.0	1	5594	169.104	46.0
Python tutorial	1	22	9.486	9.0	1	8343	497.285	227.0

Figure 3 shows us that there is a significant difference in the complexity of source code within the categories observed in our research. There are three reasons for this. The first reason lies in the fact that scientific programming, covered by AI and Data Science categories, tends to result in Interactive Python notebooks, which hold the entire application within a single file. The second reason is due to the problem's complexity. Educational programming problems, covered by object-oriented programming and problem-solving categories, tend to solve an oddly specific problem, and the code is usually held within a single module for self-sustainability. The third reason takes root in software development practices, which can be observed within the Python project. In software development, there is a tendency to keep things simple and separate concerns, which means that there are usually more Python modules found in such source codes.

Our SOM, configured as specified in Table 1 produced both distributional and differential data, representing the distribution of programming concepts and the differences between them, respectively. A visualization of those data can be seen on Figure 4.

Figure 5 depicts that one SOM neuron will be activated for 98% of the AST nodes when our test codes consist of just one concept. This shows that SOMs consider such code to be trivial in complexity and ultimately very similar.

The following two outputs come from SOM, after their weight vector has been expressed with Python's abstract syntax:

- Module Assign Constant Name Store
- Module Assign Expr Call Constant Name Load Store

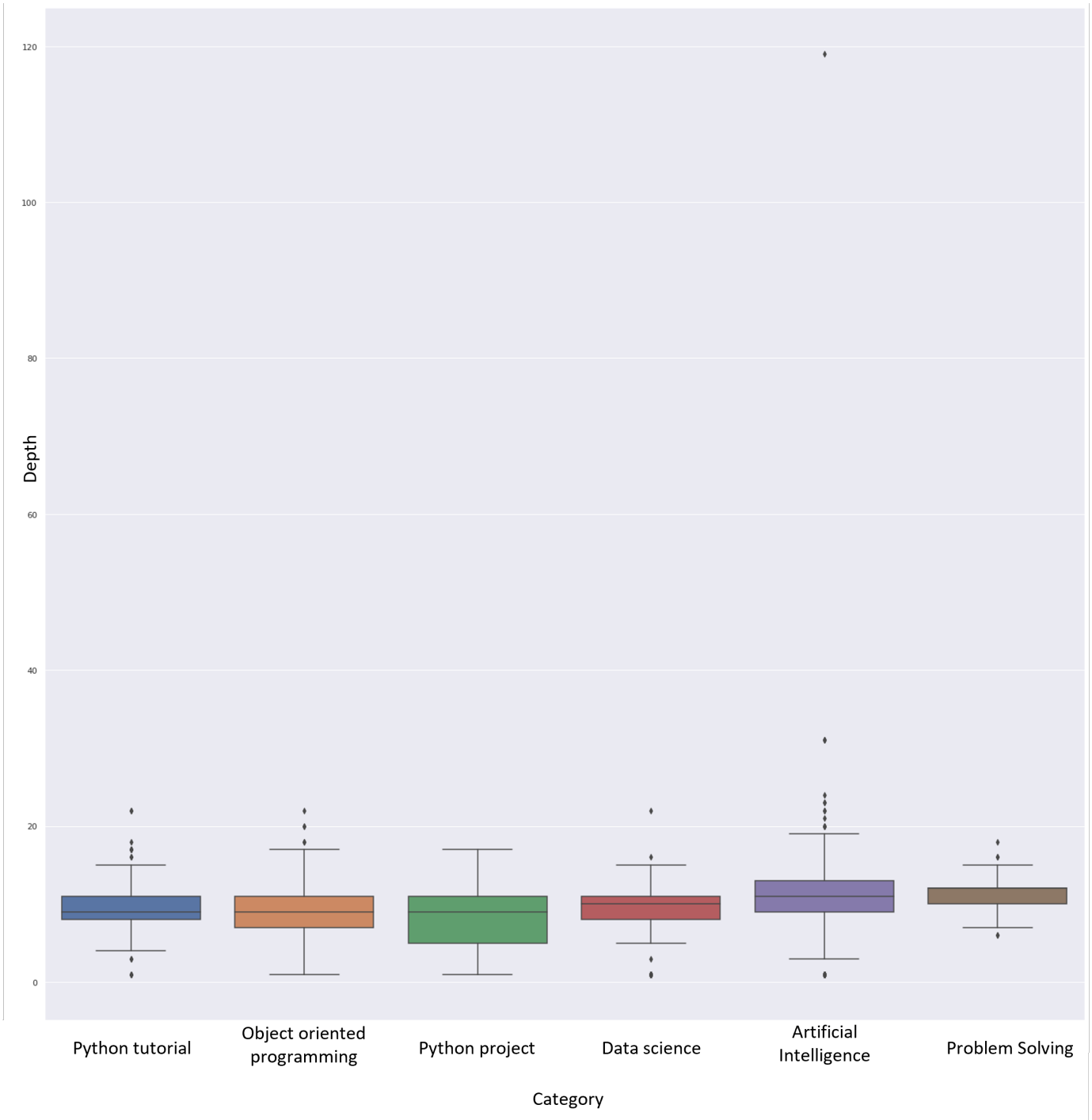

Figure 3. Visualization of training data aggregations.

It can be observed that we successfully trained the SOM to make a distinction between the following two programming concepts:

- Using a variable to store a value
- Reusing the variable to utilize the stored value

Figure 6 shows us that training resulted in a relatively high quantization error (2.025) and a relatively low topographic error (0.2).

The following source codes provide a solution to the same problem: calculating and printing the sum of list elements.

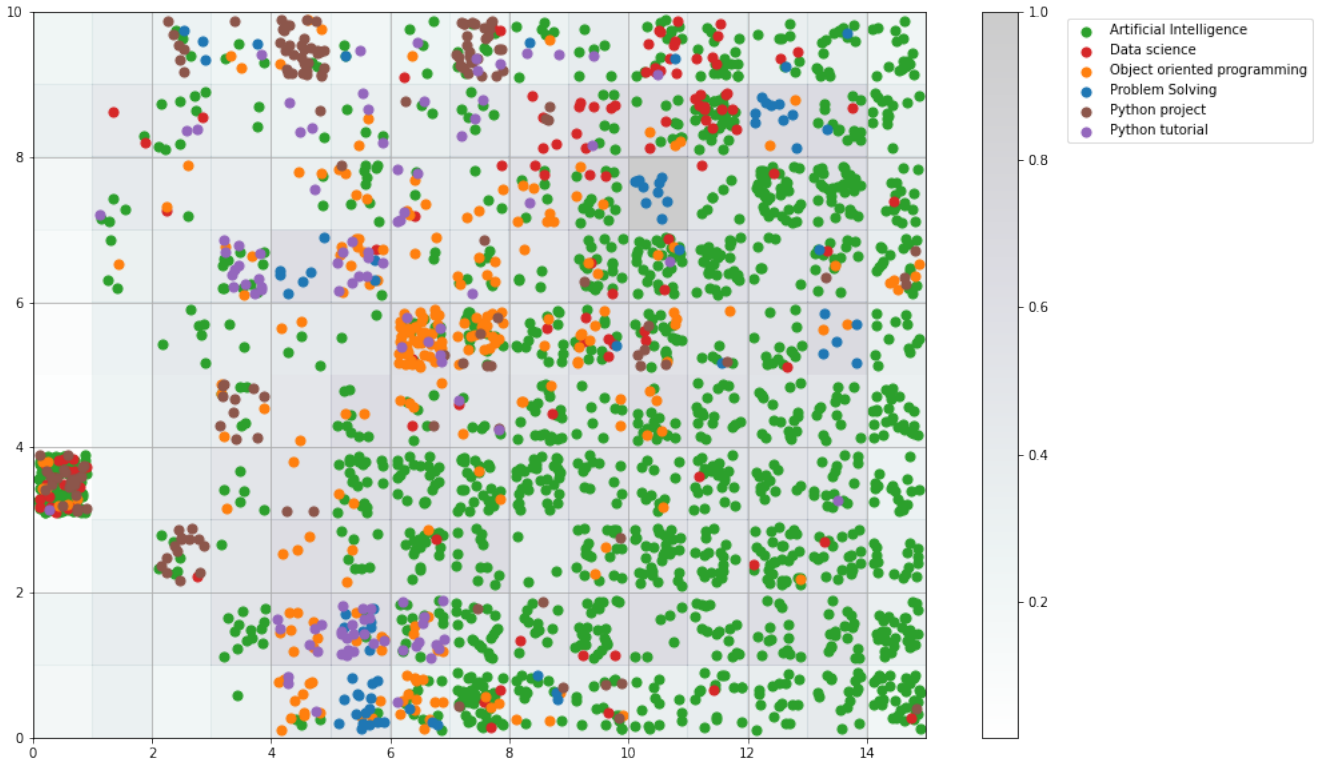

Figure 4. Code distribution within the SOM based on categories.

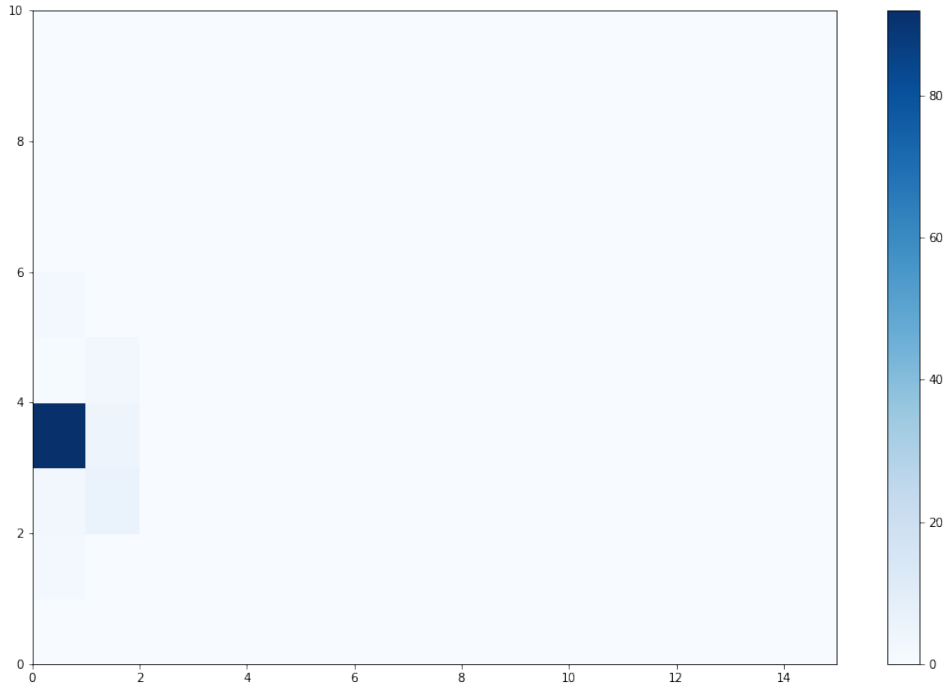

Figure 5. SOM activation when facing individual AST nodes.

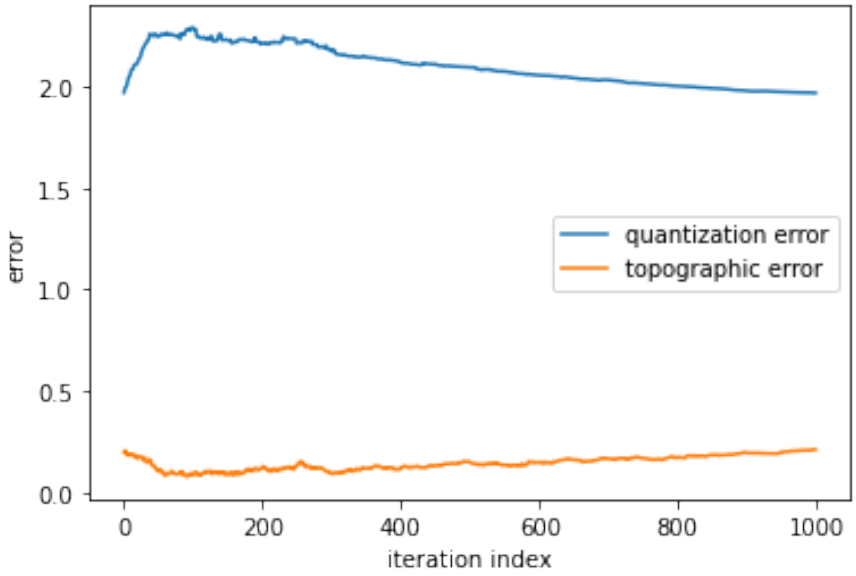

Figure 6. Quantization and topographic error of the trained SOM.

Based on the source code provided in Appendix A, Table 4 shows us that the SOM acts as a sieve of sorts, with the code structure being a distinguishable feature that impacts which of the SOM's neurons are the best fit to represent the code. Only one neuron can be the winner, and that decision is based on the conceptual density of the source code in our example.

Table 4. Different nodes activated for different solutions to the same problem.

Source Code	Coordinates of the Winner Node
Appendix A.1	(3, 6)
Appendix A.2	(1, 6)
Appendix A.3	(5, 5)

Table 5 shows the number of SOM nodes that contain each of the fundamental programming concepts, as well as Python AST nodes we associate with each of the programming concepts.

Table 5. The abundance of programming concepts within the SOM.

Programming Concept	Python AST Node	Abundance
Value assignment	Assign	84%
Function definition	FunctionDef	71.33%
Branching	If	62.67%
Loops	For, While	42.67%
Class	ClassDef	34.67%

Table 6 shows the memory consumption in the scope of our experiment. Included in the table is the size of the input data, both after collection and after transformation into the input vector form. Aside from those values, the table includes the sizes of the exported values obtained after the SOM was trained.

Table 6. The memory consumption.

Data Type	Size
Collected source code	3.81 GB
SOM input vector	3.61 MB
SOM in the memory	142.92 KB
Exported trained SOM	141 KB

Table 7 shows the execution times of the key steps of the experiment. SOM self-reflection is the step in our experiment in which the SOM network is requested to self-explain what it has learned post-training. SOM classification is the step in which we used a single item from the test data set in order to classify it using the trained SOM network.

Table 7. Execution times.

Experiment Step	Duration
SOM training	2.9 s
SOM self-reflection	1.44 s
SOM classification	362 µs

5. Discussion

Since technology is everchanging, educational environments have a problem keeping up with the industry's tempo; thus, a system for data collection and the analysis of bleeding-edge topics and technologies would be beneficial. The everchanging nature of technology, however, might lead to a large number of new variables being unrepresented in the training data set, which would lead to the overfitting of any neural network.

In our research, a SOM was used as a tool for programming content self-organization. We used the SOM's sensitivity to the data structure, which some perceive as its major disadvantage, to our advantage. While the SOM was not able to predict or generate anything from the data, it was able to learn what the code consists of and highlight the problems of the input data set in relation to the time frame represented by the dimensions of the SOM.

For the purposes of this research, we perceived AST nodes to be basic programming concepts that are taught in many programming courses, both formally and informally. Thus, our binary input vector represents knowledge of the existence of all fundamental programming concepts within a source code. This was achieved through AST traversal via a custom-made AST transformer.

The output of each SOM neuron is a vector of the same dimensionality as the input vector. In this way, each neuron is able to communicate which programming concept it represents. Aside from that, our trained SOM can be used to classify new Python code after it has been transformed into an SOM-compatible input vector.

Once faced with a source code, the SOM can declare which neuron is a winner. In Table 4, we can see that using a for loop (Appendix A.1) or a built-in function (Appendix A.2) to calculate a sum of integers produces results that are conceptually much closer to each other, while implementing a recursion to do the same thing (Appendix A.3) is in the vicinity of those solutions, but conceptually produces a different solution, slightly distanced from both, although being much closer to the for loop solution (Appendix A.1).

A high quantization error is expected, because it indicates that SOM is not large enough to hold all the different types of programming code. The low topographic error indicates that SOM neurons can be considered representative of all the different programming code conceptual classes found within the input data set by the SOM.

Aside from that, due to the established transparency of our SOM, it has been proven that the differences in the code structure are driving the clustering process successfully.

Every SOM neuron was able to use Abstract grammar to communicate what activates it, i.e., what it has learned. However, this approach to content management is not fully automated, since it is not a generative AI model able to learn new technologies from syntax, documentation, and source code.

The purpose of this system is not to create a system that manages content better than a content manager, be it an educator or a programming expert; rather, it is necessary to design a system that can assist content managers to manage the content in an unbiased way. To satisfy this purpose, future research goals will be to ensure that the concepts to be taught are represented by the data set. AI transparency continues to be the main goal. The learning process should be transparent to all users (content managers, educators, and learners) as well as the auto-generated content (feedback) that drives the process.

When a source code is represented abstractly as a graph, the nodes of the graph represent programming concepts in a language-agnostic but paradigm-aware way. Most of the concepts represented by the SOM are actually patterns for using fundamental programming concepts, i.e., programming competencies, with the purpose of solving specific class(es) of programming problems.

The abundance of programming concepts reported in Table 5 shows that the SOM can be used to analyze which programming concepts are fundamental and the order in which they should be taught. What makes the programming code complex is the combination of the fundamental programming concepts, in other words, the ability to use various fundamental programming concepts in tandem to solve a problem.

Limitations of This Research

The main limitation comes from the fact that not all programming concepts can be represented by the AST nodes. By limiting ourselves to only those nodes that are part of the native AST Python library, we were able to train the SOM to be sensitive to only those source code features represented by AST nodes.

Aside from that, our research is limited to the programming concepts that are present within the source codes collected from GitHub. However, there are many other unrepresented concepts that also impact the content self-organization.

Even though we used the predetermined size of the SOM to our advantage to depict the density of programming concepts within our SOM, we were also limited by this because we could not provide an answer to the question of how big a SOM we would need to truly contain the knowledge. SOM should have a growth quality to allow it to expand itself upon reaching a certain level of intranodal density.

6. Conclusions

6.1. Answer to the Research Question

We answered our research question by showing that an SOM aided by AST parsing (transformation) can be used to achieve the necessary level of transparency. One of the goals of this system was to assist in the creation of a concept map supplemented by educational content. The concept map was derived from the SOM neural network once it had been labeled using the names of the AST nodes to form custom competency names. Each competency could be represented by a graph (more specifically, an AST). However, this merits further study.

At the moment, the language for achieving transparency is limited to Python's AST abstract grammar, which means that the output vector of each neuron can be transformed into a set of AST nodes, forming a primitive but effective way of communicating what has been machine learned by the SOM.

This research paper shows us that through self-organization and the use of SOMs, we can analyze source codes, which are the most common type of content in programming education. As a workaround to the potential problem of ending up with an unmaintainable training data set, our focus in future research will be to create data sets that focus on one

programming concept or a group of similar programming concepts to be taught. We plan to use more complex source codes for evaluation.

6.2. Future Research

The goal of this system is not to serve as a replacement for any of the existing automatic AI systems that work with programming code, but rather, it can be used as a supplement to ensure the transparency of AI system's knowledge. Currently, our SOM produces educationally-valuable metadata as an output, because it indicates where the content should reside in the curriculum based on its structural complexity. Aside from that, a SOM can be perceived as a way of extracting knowledge from the content by forming a neural representation of abstract programming concepts, thus providing a representation of programming knowledge. In this research, we formed a rudimentary knowledge base by using a dictionary to store programming concepts as *key: value* pairs. Future research should revolve around making this knowledge base more sophisticated and scalable. Since K-means and Self-Organizing Maps (SOMs) are popular clustering methods, it would be valuable to compare them in the context of this research. K-means is widely used because it is fast and easy to implement. However, it has significant limitations. In K-means, clusters are formed based on the movement of individual nodes that are not directly related to each other. On the other hand, SOMs are artificial neural networks that arrange nodes in a grid and establish direct relationships between them. SOMs can be useful for visualizing high-dimensional data and creating categories within data sets. K-means forms clusters through the use of centroids and cluster size, while SOMs use a geometric approach that seems more appropriate for curriculum design and analysis. The importance of our findings lies in the fact that, with this level of established transparency within a SOM, it is possible to establish a value system from educational content as well as a glossary of programming concepts that could be used to express programming needs, prerequisites, and what is known and unknown to a learner. Our future work will also focus on using the SOM to measure how well can it reinforce programming education. Before integrating a SOM or a similar self-organizing system in a learning environment, it would be beneficial to test its abilities within a learning simulation. NetLogo is one of the options for simulating a learning process since it has a solid interoperability layer with the Python programming language, which we used in this research. Through a simulation, we would establish a hybrid AI system consisting of agent-based, machine-learning, and knowledge-based approaches to represent content, learning, and knowledge management, respectively.

In our future research, we plan to use this semi-automatic approach to content management in cases where people, as well as AI systems, have to be trained or educated about programming. In subsequent studies, the potential for utilizing the present research to provide students with feedback on their work will be explored. Furthermore, the feasibility of employing self-organizing maps for the automated generation of program code, while maintaining transparency, will be investigated. Aside from that, orienting towards the reduction of the quantization error by using a growing SOM or a similar technique with the ability to grow based on a heuristic is also necessary.

Author Contributions: Conceptualization, M.J.; Formal analysis, M.J.; Investigation, M.J. and S.M.; Methodology, M.J., S.M. and A.G.; Writing—original draft, M.J., S.M. and A.G.; Writing—review & editing, S.M. and A.G. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Research data are available upon submitted request to the corresponding author via email.

Acknowledgments: We are grateful to many members of the Python Open Source community that share their code on GitHub under licenses that allow us to reuse the data for research and educational purposes.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

AI	Artificial Intelligence
AST	Abstract Syntax Tree
BMU	Best matching unit
SOM	Self-Organizing Map

Appendix A. Examples from the Test Data Set

Appendix A.1. Sum of Integers Using a for Loop

```
numbers = [1, 2, 3, 4, 5]
sum_of_numbers = 0
for number in numbers :
sum_of_numbers += number
print(sum_of_numbers )
```

Appendix A.2. Sum of Integers Using a Built-In Function

```
numbers = [1, 2, 3, 4, 5]
print(sum(numbers))
```

Appendix A.3. Sum of Integers Using a Recursion

```
def sum_of_list(l, n):
if n == 0:
return l[n];
return l[n] + sum_of_list(l, n-1)

numbers = [1, 2, 3, 4, 5]
print(sum_of_list(numbers, len(numbers) - 1))
```

References

1. Rotatori, D.; Lee, E.J.; Sleeva, S. The evolution of the workforce during the fourth industrial revolution. *Hum. Resour. Dev. Int.* **2021**, *24*, 92–103. [CrossRef]
2. Srivastava, S.; Tiwari, A. Preparing for the mission "Industry Demand 2030". The work of future and the skills for future workforce: Challenges & opportunities. *Spec. Ugdym.* **2022**, *1.43*, 6826–6834.
3. Garousi, V.; Giray, G.; Tuzun, E.; Catal, C.; Felderer, M. Closing the Gap between Software Engineering Education and Industrial Needs. *IEEE Softw.* **2020**, *37*, 68–77. [CrossRef]
4. Verma, A.; Lamsal, K.; Verma, P. An investigation of skill requirements in artificial intelligence and machine learning job advertisements. *Ind. High. Educ.* **2022**, *36*, 63–73. [CrossRef]
5. Meesters, M.; Heck, P.; Serebrenik, A. What is an AI engineer? An empirical analysis of job ads in The Netherlands. In Proceedings of the Proceedings of the 1st International Conference on AI Engineering: Software Engineering for AI, Pittsburgh, PA, USA, 16–24 May 2022; pp. 136–144. [CrossRef]
6. Siegfried, R.M.; Herbert-Berger, K.G.; Leune, K.; Siegfried, J.P. Trends Of Commonly Used Programming Languages in CS1 And CS2 Learning. In Proceedings of the 2021 16th International Conference on Computer Science & Education (ICCSE), Lancaster, UK, 17–21 August 2021. [CrossRef]
7. Warren, P. Teaching programming using scripting languages. *J. Comput. Sci. Coll.* **2001**, *17*, 205–216.
8. Kiesler, N.; Thorbrügge, C. A Comparative Study of Programming Competencies in Vocational Training and Higher Education. In *Proceedings of the 27th ACM Conference on on Innovation and Technology in Computer Science Education Vol. 1*; Association for Computing Machinery: New York, NY, USA, 2022; pp. 214–220. [CrossRef]
9. Scherer, R.; Siddiq, F.; Viveros, B.S. A meta-analysis of teaching and learning computer programming: Effective instructional approaches and conditions. *Comput. Hum. Behav.* **2020**, *109*, 106349. [CrossRef]

10. Heinonen, A.; Lehtelä, B.; Hellas, A.; Fagerholm, F. Synthesizing Research on Programmers' Mental Models of Programs, Tasks and Concepts—A Systematic Literature Review. *arXiv* **2022**, arXiv:2212.07763.
11. Martins, W.; de Carvalho, S.D., An Intelligent Tutoring System Based on Self-Organizing Maps—Design, Implementation and Evaluation. In *Intelligent Tutoring Systems*; Springer: Berlin/Heidelberg, Germany, 2004; pp. 573–579. [CrossRef]
12. Chan, A.; Spracklen, T., Discovering Common Features in Software Code Using Self-Organizing Maps. In *The State of the Art in Computational Intelligence*; Physica-Verlag HD: Heidelberg, Germany, 2000; pp. 3–8. [CrossRef]
13. Souza, A.A.; Barcelos, T.S.; Munoz, R.; Silveira, I.F.; Omar, N.; Silva, L.A. Self-organizing maps to find computational thinking features in a game building workshop. In Proceedings of the 2017 XLIII Latin American Computer Conference (CLEI), Cordoba, Argentina, 4–8 September 2017. [CrossRef]
14. Purbasari, I.Y.; Puspaningrum, E.Y.; Putra, A.B.S. Using Self-Organizing Map (SOM) for Clustering and Visualization of New Students based on Grades. *J. Phys. Conf. Ser.* **2020**, *1569*, 022037. [CrossRef]
15. WenShung Tai, D.; Wu, H.; Li, P. Effective elearning recommendation system based on selforganizing maps and association mining. *Electron. Libr.* **2008**, *26*, 329–344. [CrossRef]
16. Pedrycz, W.; Succi, G.; Musílek, P.; Bai, X. Using self-organizing maps to analyze object-oriented software measures. *J. Syst. Softw.* **2001**, *59*, 65–82. [CrossRef]
17. Zhu, G.; Zhu, X. The Growing Self-organizing Map for Clustering Algorithms in Programming Codes. In Proceedings of the 2010 International Conference on Artificial Intelligence and Computational Intelligence, Sanya, China, 23–24 October 2010. [CrossRef]
18. Kalliamvakou, E.; Gousios, G.; Blincoe, K.; Singer, L.; German, D.M.; Damian, D. The promises and perils of mining github. In Proceedings of the 11th working Conference on Mining Software Repositories, Hyderabad, India, 31 May–1 June 2014; pp. 92–101. [CrossRef]
19. Cui, B.; Li, J.; Guo, T.; Wang, J.; Ma, D. Code Comparison System based on Abstract Syntax Tree. In Proceedings of the 2010 3rd IEEE International Conference on Broadband Network and Multimedia Technology (IC-BNMT), Beijing, China, 26–28 October 2010; pp. 668–673. [CrossRef]
20. Balreira, D.G.; Silveira, T.L.d.; Wickboldt, J.A. Investigating the impact of adopting Python and C languages for introductory engineering programming courses. *Comput. Appl. Eng. Educ.* **2023**, *31*, 47–62. [CrossRef]
21. Kohonen, T. The self-organizing map. *Proc. IEEE* **1990**, *78*, 1464–1480. [CrossRef]
22. Varsta, M.; Heikkonen, J.; Millan, J.D.R. Context learning with the self-organizing map. In *Workshop on Self-organizing, Helsinki University of Technology, Espoo*; Citeseer: Princeton, NJ, USA, 1997.
23. Vettigli, G. MiniSom: Minimalistic and NumPy-Based Implementation of the Self Organizing Map. 2018. Available online: https://github.com/JustGlowing/minisom/ (accessed on 26 March 2023).
24. Kiviluoto, K. Topology preservation in self-organizing maps. In Proceedings of the International Conference on Neural Networks (ICNN'96), Washington, DC, USA, 3–6 June 1996; Volume 1, pp. 294–299. [CrossRef]
25. Schreck, T. *Visual-Interactive Analysis with Self-Organizing Maps-Advances and Research Challenges*; IntechOpen: London, UK, 2010. [CrossRef]
26. Fabiyi, S.D. A review of unsupervised artificial neural networks with applications. *Int. J. Comput. Appl.* **2019**, *181*, 22–26. [CrossRef]
27. Yusob, B.; Mustaffa, Z.; Shamsuddin, S. Preserving the topology of Self-Organizing maps for data Analysis: A review. In *Proceedings of the IOP Conference Series: Materials Science and Engineering*; IOP Publishing: Bristol, UK, 2020; Volume 769, p. 012004. [CrossRef]