An approach for measuring creativity of Scratch projects based in games with specific genre

Daniel Escobar Morales
Escuela Técnica Superior de Ingeniería de Telecomunicación
Universidad Rey Juan Carlos (Madrid, Spain)
daniesmor@gsyc.urjc.es

Abstract

In this work, we propose a novel method to measure creativity in a set of Scratch projects through the use of deep learning models, specifically architectures based on Encoder-only Transformers. We represent each project as a list of scripts, which in turn are represented as a sequence of blocks, and the model generates contextual embeddings that reflect the structure of the project. During training, the Triplet Loss function is applied to distinguish repetitive patterns within a specific project genre. In the evaluation, creativity is measured by comparing the cosine similarity of specific embeddings with reference projects within the same genre.

Key Words: *computational creativity, evaluation, creativity in Scratch projects, Triplet Loss for creative evaluation*

Work In Progress

Introduction

Creativity in software development is a difficult concept to measure, especially when analyzing the creative value of code sequences. This project explores an innovative approach to evaluate creativity in Scratch projects using Encoder-only Transformer architectures, a widely used architecture in natural language processing.

The main objective is to represent code scripts as sequences of blocks that make up the projects of a specific subgenre, where each block reflects a functional unit of the project. Using embeddings generated by a Transformer, trained with a Triplet Loss function, the creativity of a project is evaluated by comparing its embeddings with those of reference projects, whose embeddings were generated during the training phase, both in a local context (scripts) and at a global level (the entire sprite).

Through this methodology, it is possible to evaluate creativity by comparing the cosine similarity of the contextual embeddings of a specific project with the reference projects, in order to measure how much it follows conventional patterns within the genre it belongs to. With these procedures, we aim to develop an automated system capable of identifying repetitive patterns within a specific project subgenre, providing a useful tool for researchers and professionals in the field.

Fundamentals

Embeddings in Language: Imagine two speakers of Spanish. One uses the language conventionally, while the other plays with the language, using common expressions in unexpected ways or combining words to create a new nuance of meaning. Both speakers might use the same words, but the way they combine them and the context in which they are employed could differ.

If we were to generate embeddings for the sentences of both speakers, we could observe that, although the individual words might have similar embeddings, the complete sentences would generate different embeddings due to their unique context and combination. This deviation from expected patterns in language use could be seen as a sign of creativity.

Similarly, in programming, especially in the context of projects like Scratch, one programmer might follow conventional patterns when combining blocks of code, while another might introduce variations in how these blocks are used or combined, creating new solutions or unexpected effects.

By generating embeddings for code scripts, we are essentially capturing not only the individual blocks but also how these blocks interact and combine to create a final outcome. A script that follows a common pattern will likely generate an embedding that is close to the reference embeddings, which represent the expected behavior based on the model's training. However, a script that introduces variations, even while using similar blocks, will generate an embedding that deviates from these reference embeddings. This deviation can be interpreted as an indicator of creativity, much like how we would view a speaker who plays with language.

To gain insight into the following topic, the following pseudocode fragments are presented:

Algorithm 1 Find Maximum and Count Occurrences (Inefficient)

```
1: Initialize maximum to a very small value
2: for each number in the list do
3:     if number > maximum then
4:         Assign number to maximum
5:     end if
6: end for
7: Initialize counter to 0
8: for each number in the list do
9:     if number == maximum then
10:        Increment counter by 1
11:    end if
12: end for
13: Return maximum and counter
```

Work in Progress

Algorithm 2 Find Maximum and Count Occurrences (Efficient)

```
 1: Initialize maximum to a very small value
 2: Initialize counter to 0
 3: for each number in the list do
 4:     if number > maximum then
 5:         Assign number to maximum
 6:         Assign counter to 1
 7:     else if number == maximum then
 8:         Increment counter by 1
 9:     end if
10: end for
11: Return maximum and counter
```

Work in Progress

Both code snippets perform the same function: given a list of numbers, they seek to find the maximum number and count how many times it appears in the list. Although both snippets achieve the same goal and use the same instructions (statements and loops), one of them makes more efficient use of computational resources.

The first algorithm traverses the list twice: in the first pass, it finds the maximum number, and in the second pass, it counts how many times it appears. This approach has a time complexity of *O(2n)*, as it performs two iterations over the list.

In contrast, the second algorithm iterates through the list only once. In each iteration, it checks if the current number is greater than the stored maximum; if so, it updates the maximum and resets the counter. If the number is equal to the stored maximum, it simply increments the counter without needing to reassign the maximum. This approach has a time complexity of *O(n)*, making it clearly more efficient than the first algorithm.

It is easy for someone who is just starting to program or who does not have a sufficient grasp of the discipline to fall into poor practices like those in Algorithm 1. For this reason, we will use Algorithm 1 as a reference to complete this task of finding and counting maximums.

Now, let's assume that both algorithms are processed through a model capable of extracting the context of each instruction present in each algorithm and interpreting each as a vector in a two-dimensional space that represents the semantic relationship between the different instructions. The result would be something like this:

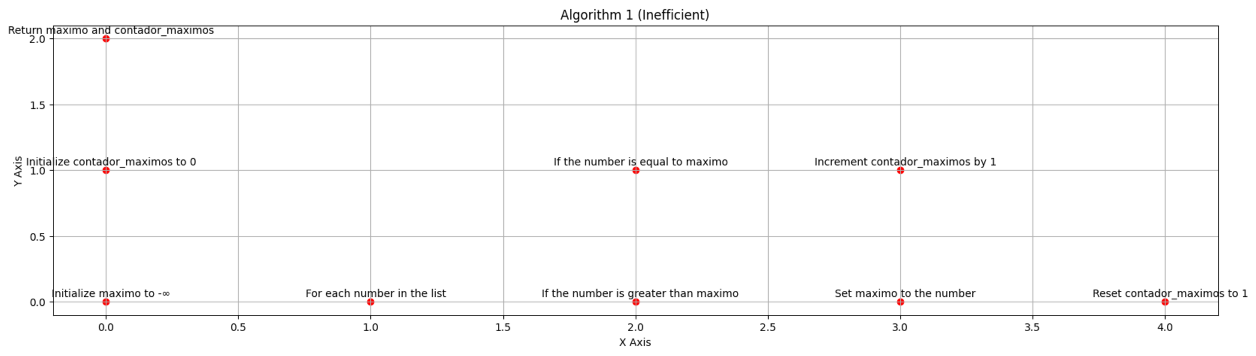


Figure 1: Algorithm 1 Example Embeddings

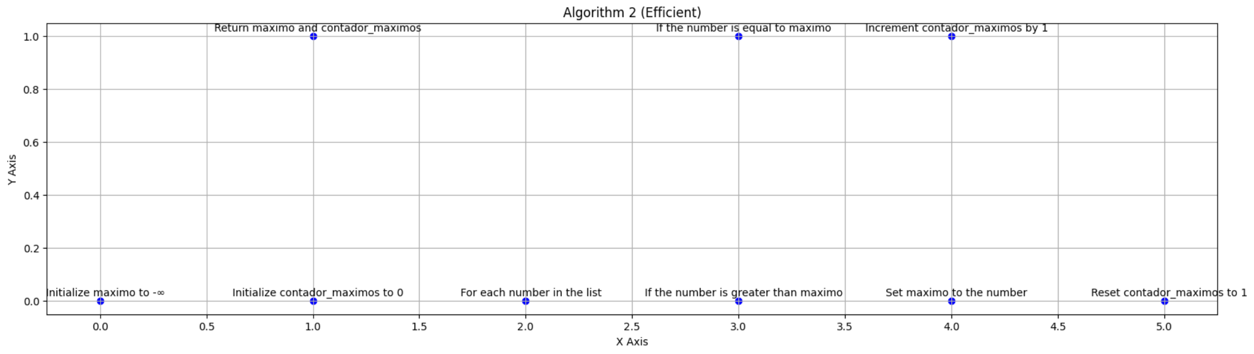


Figure 2: Algorithm 2 Example Embeddings

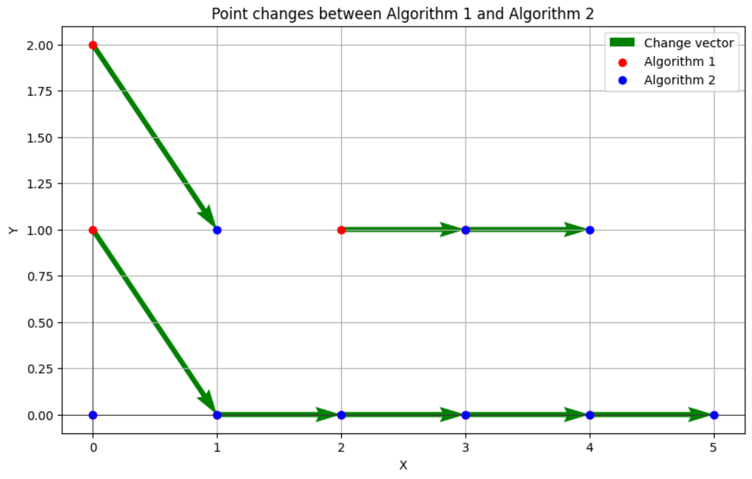
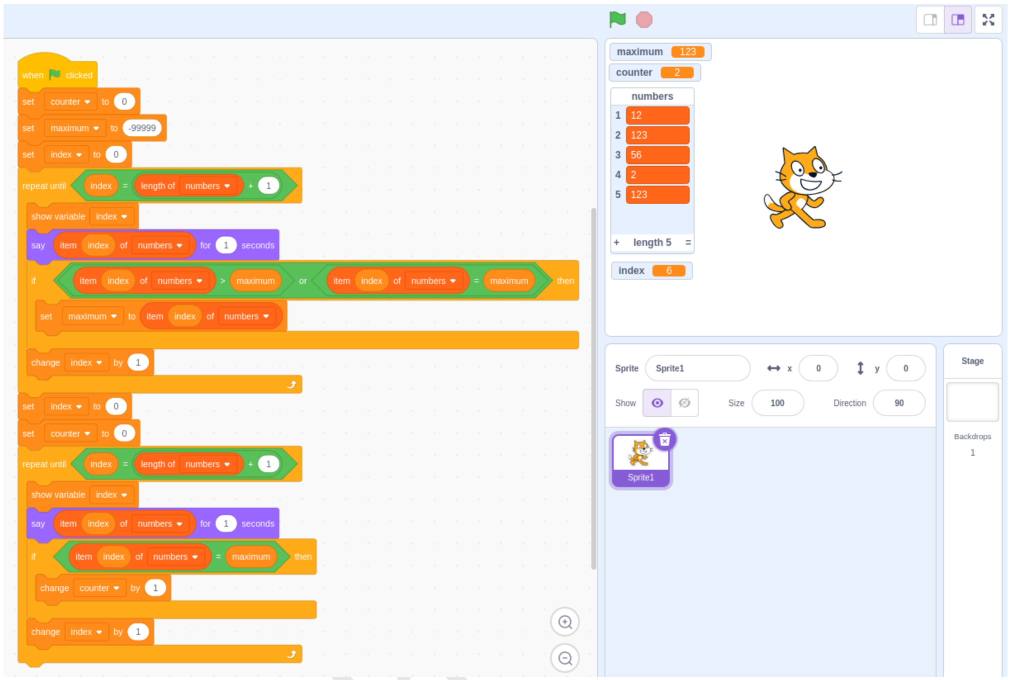


Figure 3: Point Changes between Algorithm 1 and Algorithm 2

If we consider all the change vectors present in the graph 3 and measure their magnitudes, we obtain a total of 8.82 scalar units. This change could be interpreted as an increase in computational efficiency by 8.82 points, but in reality, this is not necessarily the case. What these units of change actually indicate is the difference relative to a reference vectorial set. As mentioned earlier, this reference set provides contextual information about typical programming patterns and habits of a specific project genre.

This change score could represent how creative someone has been in their programming methodology, understanding creativity as the ability to test and experiment with new paradigms and implementations compared to usual behaviors.

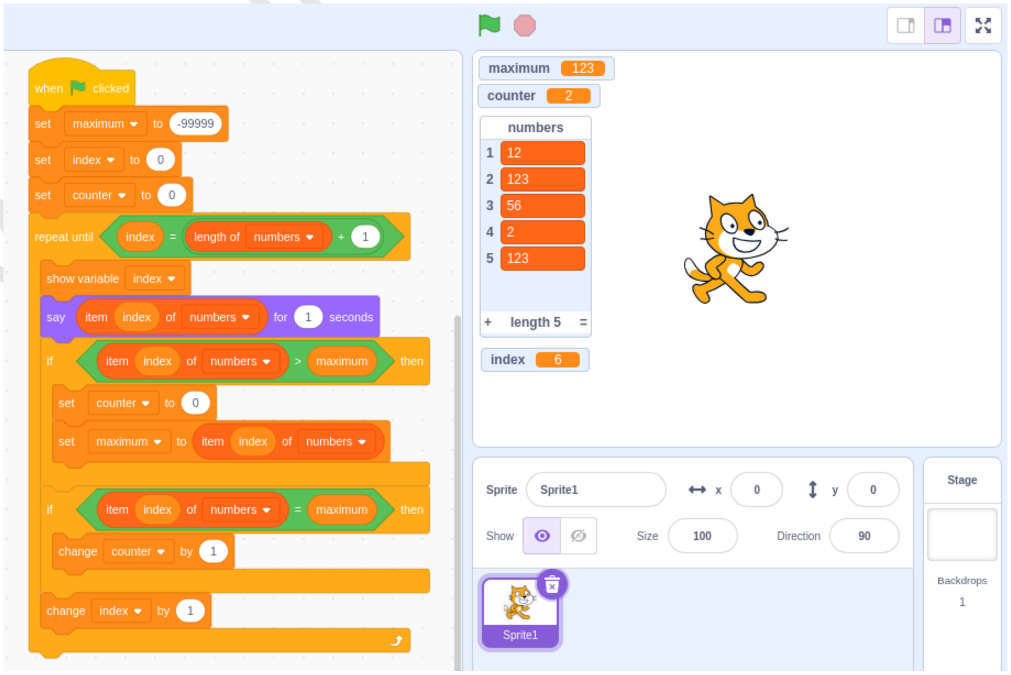
For more illustrative purposes, both algorithms were brought to the MIT *Scratch* application. Thanks to Chrome's Developer Tools, we can access the performance statistics of both programs and see more graphically how one is indeed more efficient than the other.



```
when [flag] clicked
set counter to 0
set maximum to -99999
set index to 0
repeat until <(index) = ((length of numbers) + 1)>
    show variable index
    say (item (index) of numbers) for 1 seconds
    if <<(item (index) of numbers) > (maximum)> or <(item (index) of numbers) = (maximum)>> then
        set maximum to (item (index) of numbers)
    change index by 1
set index to 0
set counter to 0
repeat until <(index) = ((length of numbers) + 1)>
    show variable index
    say (item (index) of numbers) for 1 seconds
    if <(item (index) of numbers) = (maximum)> then
        change counter by 1
    change index by 1
```

```
maximum 123
counter 2
numbers
1 12
2 123
3 56
4 2
5 123
+ length 5 =
index 6
```

Figure 4: Algorithm 1 (Inefficient) in Scratch Environment

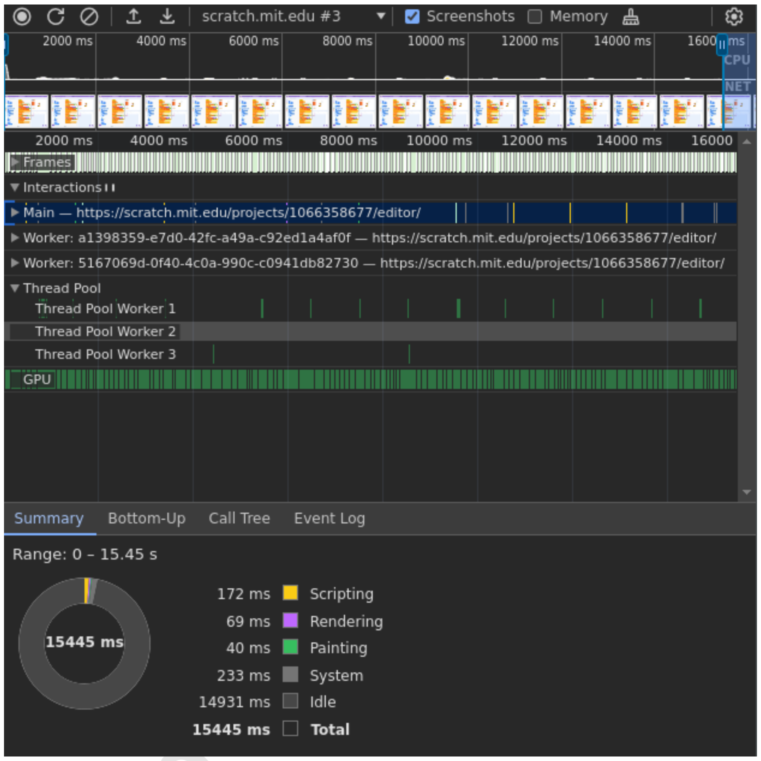


```
when [flag] clicked
set maximum to -99999
set index to 0
set counter to 0
repeat until <(index) = ((length of numbers) + 1)>
    show variable index
    say (item (index) of numbers) for 1 seconds
    if <(item (index) of numbers) > (maximum)> then
        set counter to 0
        set maximum to (item (index) of numbers)
    if <(item (index) of numbers) = (maximum)> then
        change counter by 1
    change index by 1
```

```
maximum 123
counter 2
numbers
1 12
2 123
3 56
4 2
5 123
+ length 5 =
index 6
```

Figure 5: Algorithm 2 (Efficient) in Scratch Environment

The following analysis covers certain real-time statistics of both programs.

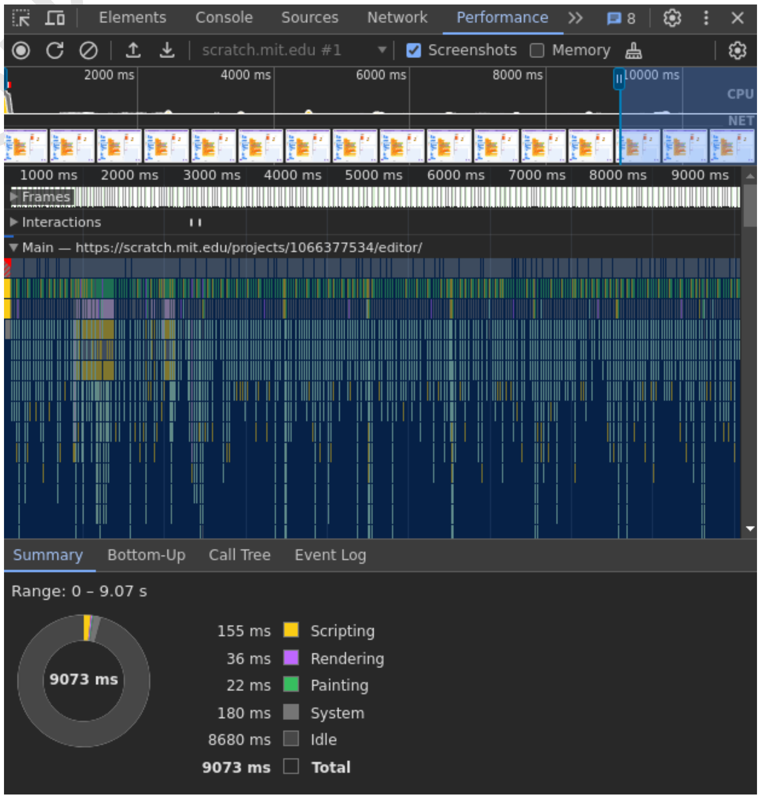

scratch.mit.edu #3
Screenshots
Memory
CPU
NET
Frames
Interactions
Main — https://scratch.mit.edu/projects/1066358677/editor/
Worker: a1398359-e7d0-42fc-a49a-c92ed1a4af0f — https://scratch.mit.edu/projects/1066358677/editor/
Worker: 5167069d-0f40-4c0a-990c-c0941db82730 — https://scratch.mit.edu/projects/1066358677/editor/
Thread Pool
Thread Pool Worker 1
Thread Pool Worker 2
Thread Pool Worker 3
GPU
Summary
Bottom-Up
Call Tree
Event Log
15445 ms

Range: 0 – 15.45 s

172 ms	🟨	Scripting
69 ms	🟪	Rendering
40 ms	🟩	Painting
233 ms	⬜	System
14931 ms	⬜	Idle
15445 ms	⬜	**Total**

Figure 6: Algorithm 1 (Inefficient) Performance

Work in



Range: 0 – 9.07 s

155 ms	🟨	Scripting
36 ms	🟪	Rendering
22 ms	🟩	Painting
180 ms	⬜	System
8680 ms	⬜	Idle
9073 ms	⬜	**Total**

Figure 7: Algorithm 2 (Efficient) Performance

While these analyses were performed in a very rudimentary manner, it is evident that there is a notable difference in execution times between the two algorithms, and given that both serve a similar purpose, this translates into a loss of efficiency. There are cases where creativity not only plays a role in innovation but also in not compromising efficiency through this increase in creativity.

Now, imagine that the reference *embeddings* were obtained from the efficient algorithm 2, and that our specific *embeddings* were obtained from the inefficient algorithm 1. When compared, we would again obtain 8.82 units of change. But now this change is not to *"more efficient"* but rather represents the use of patterns that compromise computational efficiency and introduce redundancy. In this case, we could assert that the creativity is not justified.

The methodology explained below covers the obtaining of contextual embeddings for the "Action" subcategory within the category of games in Scratch projects. These embeddings will be used as references to be compared with the embeddings obtained from a specific project within that same subcategory. Following a similar procedure to the one already explained, it will be possible to establish how creative that project has been.

Work in Progress

Methodology

Data Tokenization: To process the lists, each code block is associated with a numerical identifier derived from a vocabulary containing all the blocks used in the project scripts, sorted in decreasing order according to their frequency of appearance. This tokenization process converts the blocks into numerical representations, which are then transformed into tensors to be processed by the deep learning model.

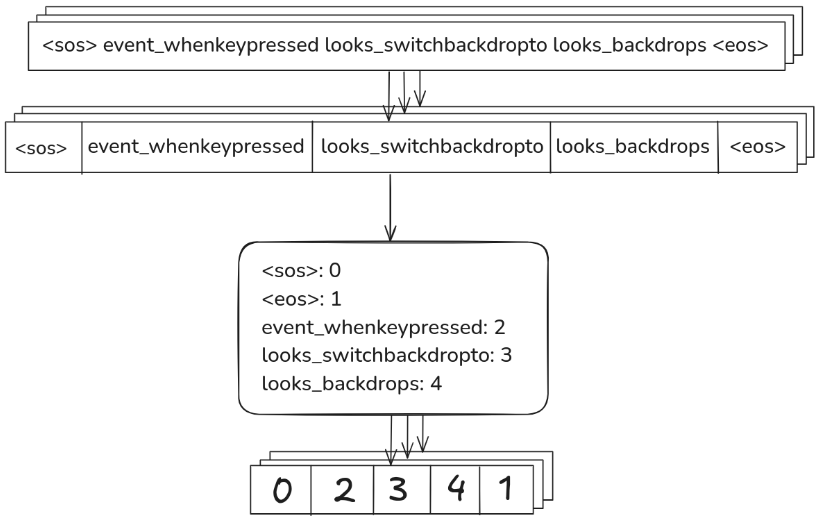


Figure 8: Esquema de tokenización de scripts

Once the tokenization process, visually summarized in figure 20, is completed, a maximum length for the sequences is set, meaning a maximum number of blocks that can be processed by the model. If the sequence exceeds this length, it will be truncated; if it is shorter, it will be padded with special *<pads>* tokens.

In the case of analyzing the global context, each script follows the explained tokenization process, with the difference that this time each script will be separated from the next by the special

<SCRIPT_END> token.

Data Preprocessing: The dataset used in this study consists of a detailed breakdown of each block present in the scripts of projects from the "Action" genre. Each script is broken down into logical blocks representing functional units of the code. To implement the *Triplet Loss* function during the training phase, it is essential to have three types of sequences: *Anchor*, *Positive*, and *Negative*. For creating the *Anchor* and *Positive* sequences, the projects from the "Action" genre were divided into two groups evenly. The resulting sequences are as follows:

- **Anchor**: This list contains the scripts from the first group of "Action" projects. The scripts in this list act as anchors during training.
- **Positive**: This list contains the scripts from the second group of "Action" projects. The scripts in this list are designed to be similar to those in the *Anchor* list.
- **Negative**: This list contains scripts from a different genre, in this case, "Storytelling". These scripts serve as negative examples that should be as distant as possible from the scripts in the *Anchor* and *Positive* lists.

These three lists are fundamental for implementing the *Triplet Loss* function, which optimizes the distance between the sequences in the *Anchor* and *Positive* lists and maximizes the distance between the *Anchor* list and the *Negative* list, which will be explained in more detail later.

To enrich the context of each block, its context is analyzed across different attention ranges. The first range covers the context of the script to which the block belongs; the second range covers all the scripts that make up a project. However, due to technical limitations, the global range has been limited to the sprite level, processing all the scripts present in each sprite that makes up the project. In this way, the general context and relationships between scripts can be extracted to understand how they interrelate within the sprite. For this reason, these three sequences will be generated in both *local context* and global context.

In summary, using the *local context*, the input to the *Transformer* would be a sequence of blocks from an individual script. If we use the *global context*, the input to the *Transformer* would be all the scripts present in each sprite that makes up each project.

Work in Progress

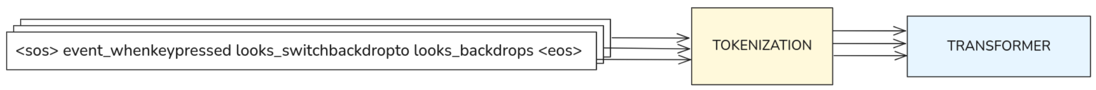

<sos> event_whenkeypressed looks_switchbackdropto looks_backdrops <eos>
TOKENIZATION
TRANSFORMER

Figure 9: Secuencias en *contexto local*

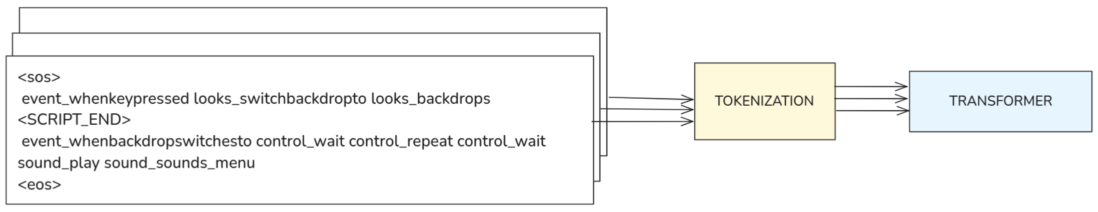

<sos>
event_whenkeypressed looks_switchbackdropto looks_backdrops
<SCRIPT_END>
event_whenbackdropswitchesto control_wait control_repeat control_wait
sound_play sound_sounds_menu
<eos>
TOKENIZATION
TRANSFORMER

Figure 10: Secuencias en *contexto global*

La figura 9 muestra de forma visual y aproximada como es el proceso de creación de las entradas del modelo para un *contexto local*, mientras que la figura 10 muestra este mismo proceso para un *contexto global*.

Arquitectura del trasnformer: El modelo utilizado en este estudio es un Transformer, que consta de un encoder, utilizado para capturar las relaciones complejas entre los elementos de entrada. Dado que estamos interesados en evaluar la creatividad a través de la similitud de secuencias, el modelo se centra únicamente en el encoder, evitando el uso del decoder tradicional. Este encoder toma como entrada secuencias de bloques de código (scripts) para el *contexto local*, y una secuencia de scripts para el *contexto global*, previamente tokenizadas. Los embeddings representan de manera numérica cada bloque de código en un espacio vectorial continuo.

Work in Progress

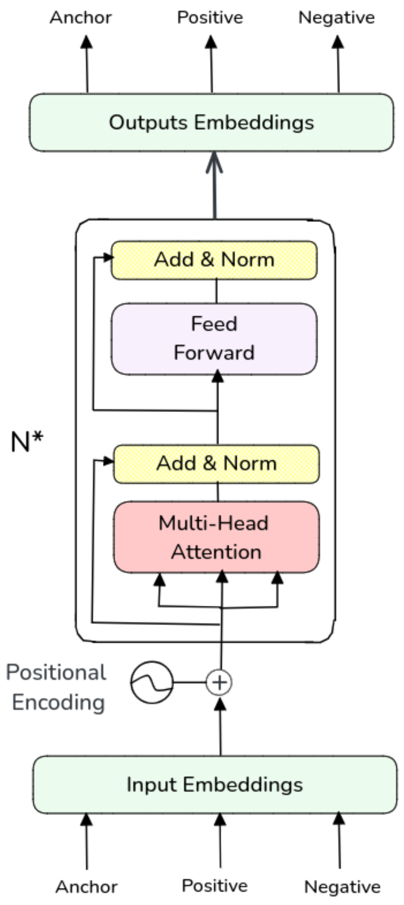


Figure 11: Encoder-only scheme

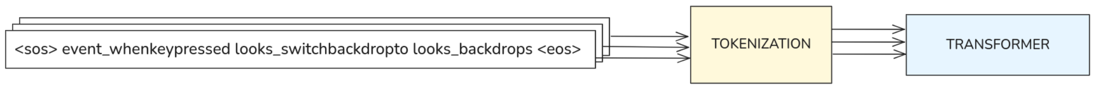
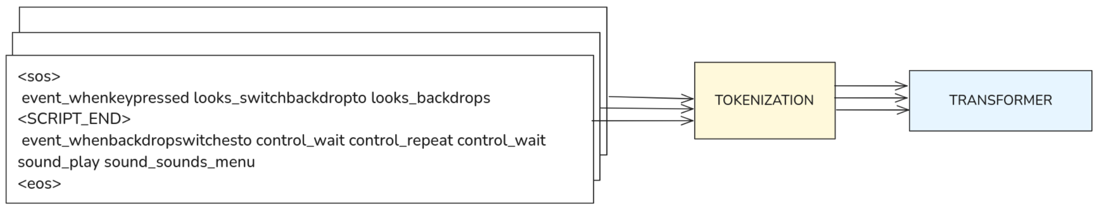

<sos> event_whenkeypressed looks_switchbackdropto looks_backdrops <eos>
TOKENIZATION
TRANSFORMER

Figure 12: Sequences in *local context*



```
<sos>
 event_whenkeypressed looks_switchbackdropto looks_backdrops
<SCRIPT_END>
 event_whenbackdropswitchesto control_wait control_repeat control_wait
sound_play sound_sounds_menu
<eos>
```

Figure 13: Sequences in *global context*

Figure 20 visually and approximately shows the process of creating model inputs for a *local context*, while figure 20 shows the same process for a *global context*.

Transformer Architecture: The model used in this study is a Transformer, consisting of an encoder used to capture the complex relationships between input elements. Since we are interested in evaluating creativity through sequence similarity, the model focuses solely on the encoder, avoiding the use of the traditional decoder. This encoder takes as input sequences of code blocks (scripts) for *local context*, and a sequence of scripts for *global context*, previously tokenized. The embeddings numerically represent each code block in a continuous vector space.

Work in Progress

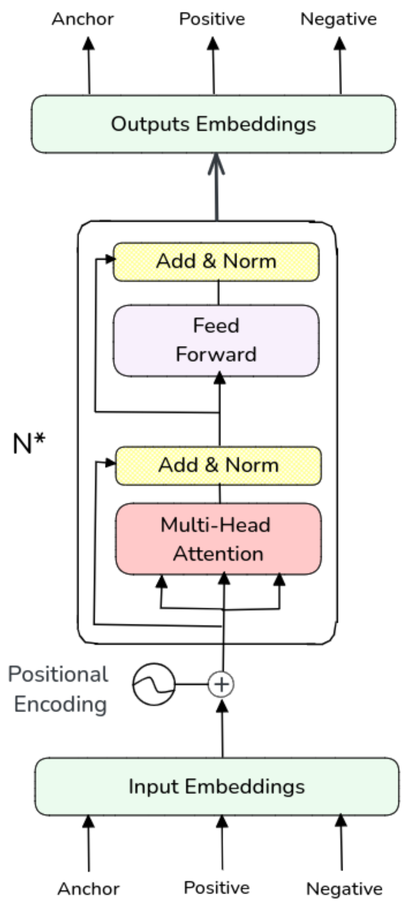


Figure 14: Encoder-only scheme

This architecture, shown in figure 20, is used to obtain embeddings for *local* and *global* contexts, meaning that each block has both a local representation within a script and a global representation within the context of the entire sprite to which it belongs, including all scripts present in it. Therefore, two identical architecture models have been used to analyze different contexts and attention ranges. It is worth noting that for the dimensionality of the resulting embeddings, $d_{model} = 512$ is used, with a sequence length of 128 tokens for both *local* and *global* contexts.

Hyperparameter Study: Starting with the maximum sequence length, the length of scripts in terms of the number of code blocks present in each one has been analyzed. The evaluation of block lengths revealed that 0.014% of the scripts are composed of fewer than 128 blocks, for the anchor, positive, and negative datasets. Based on this analysis, the *seq_len* value has been set to 128.

This *seq_len* value is chosen to capture all blocks present in the vast majority of scripts while adhering to common standards used in Transformer models. Using a *seq_len* of 128 allows the model to process the maximum length of scripts observed in our data without truncating relevant information, while maintaining compatibility with typical Transformer model parameters, which commonly use values in this range.

Choosing a *seq_len* of 128 ensures that the model can handle sequences of varying lengths while preserving the integrity of the code block information, which is essential for accurate creativity evaluation in the context of script analysis. The following figures provide a view of the number of blocks in each script analyzed for this article within the *local context*:

Work in Progress

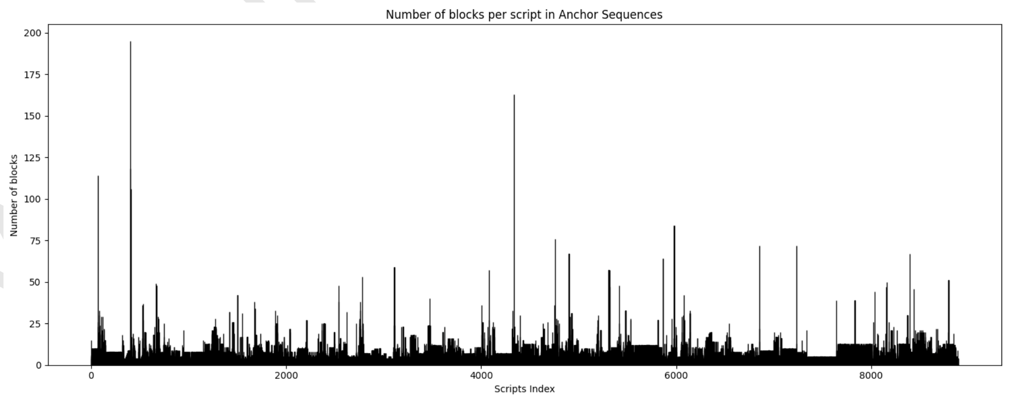


Figure 15: Amount of blocks per Script in Anchor Sequence (Local context)

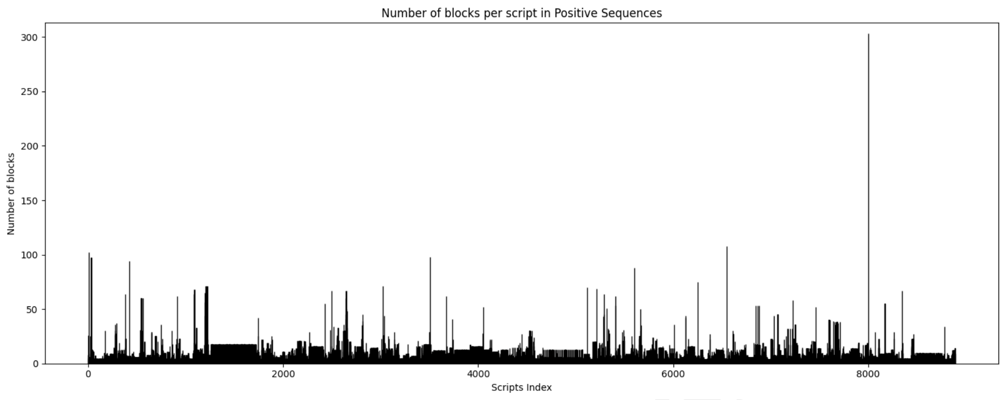


Figure 16: Amount of blocks per Script in Positive Sequence (Local context)

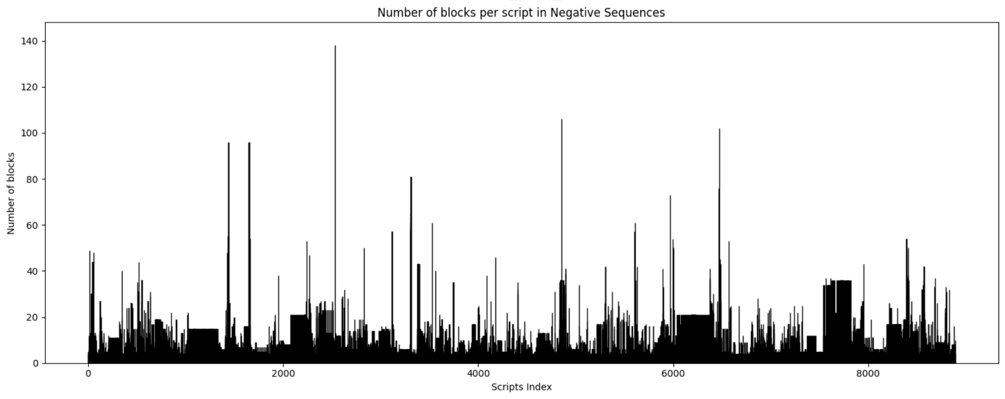


Figure 17: Amount of blocks per Script in Negative Sequence (Local context)

Increasing the attention range to the global context, the analysis methodology has been repeated, but this time focusing on the *global context*, i.e., the number of blocks per sprite rather than per script. The analysis reveals that only 1.47% of sprites contain more than 128 blocks. This extremely low percentage justifies the decision to keep the *seq_len* value at 128.

It is important to note that, compared to blocks per script, the length range has increased when considering sprites, which is an expected observation given the greater complexity and diversity of projects in the sprite context. Despite this increase in range, the percentage of sprites exceeding 128 blocks remains insignificant. Therefore, using a *seq_len* of 128 continues to be an appropriate and practical choice, allowing efficient sequence length management without truncating relevant information. The following figures provide a view of the number of blocks in each sprite analyzed for this article within the *global context*:

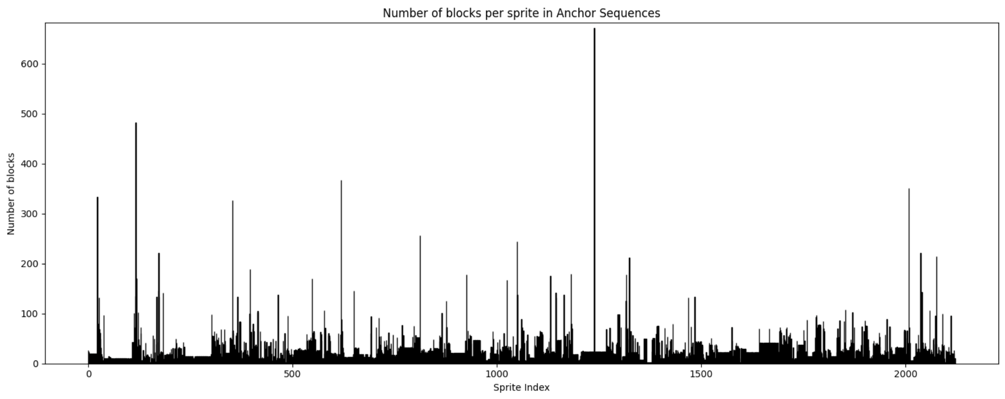


Figure 18: Amount of blocks per Sprite in Negative Sequence (Global context)

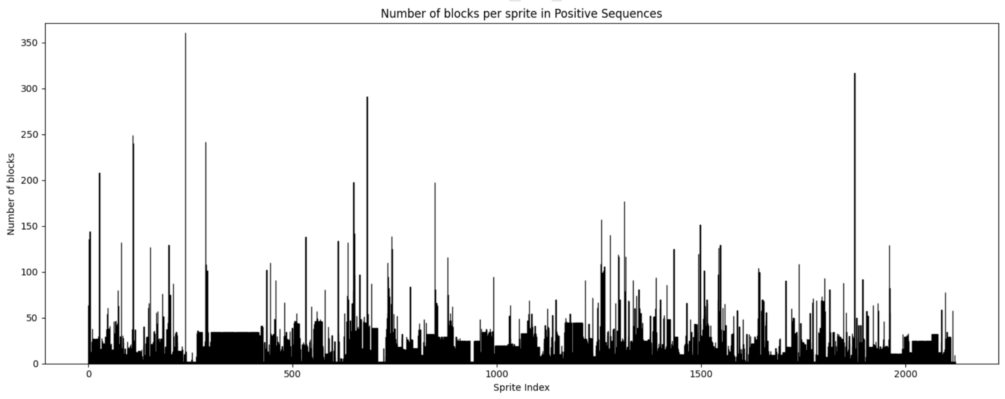


Figure 19: Amount of blocks per Sprite in Positive Sequence (Global context)

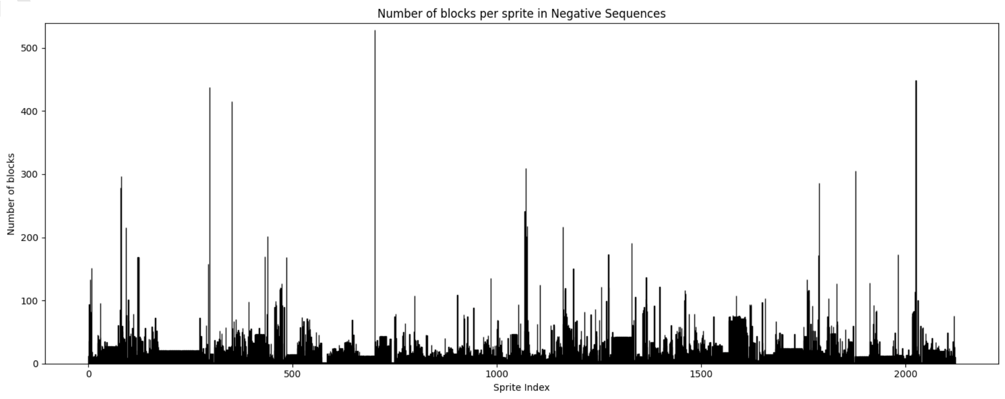


Figure 20: Amount of blocks per Sprite in Negative Sequence (Global context)

For the batch size, a value of 16 has been chosen due to the technical limitations of the development environment. For other model hyperparameters, the values established in the study by Vaswani et al. (2017) [2], which introduced the Transformer model, have been used. These values include common configurations such as the number of encoder layers, the number of heads in multi-head attention, and the size of the feedforward layers. This choice is based on the proven effectiveness of these parameters in the literature and their widespread adoption in Transformer implementations for various natural language processing tasks. Although additional adjustments may be needed to optimize the model for the specific domain of this study, it is considered that the established parameters provide a solid foundation for initial experimentation.

Implementation of Triplet Loss: To recognize patterns of a specific genre, we adopt the Triplet Loss approach, as expressed in equation **??**. This loss function aims to minimize the distance between embeddings of related sequences (anchor and positive) while maximizing the distance with an unrelated sequence (negative). This allows the model to learn representations that reflect semantic similarity between different genres. Thus, with anchor and positive being scripts present in projects of the same genre, and negative being scripts present in projects of a different genre, Triplet Loss can be used to find certain programming patterns that characterize projects of a particular subgenre. The Triplet Loss function is given by the following formula:

$$L = \max(0,\ d(a,p) - d(a,n) + \alpha) \tag{1}$$

where $d(a,p)$ represents the distance between the anchor and the positive, $d(a,n)$ the distance between the anchor and the negative, and α is a margin that ensures a minimum separation between the distances.

Training Process: During training, each *batch* contains a set of sequences (anchor, positive, and negative). Each sequence can be a script, in *local context*, or all the scripts that make up a sprite, in *global context*. Thus, for each *batch* in each epoch, the *encoder* runs three times, once for each sequence, generating the corresponding *embeddings* for the anchor, positive, and negative. These *embeddings* are then compared using the *Triplet Margin Loss* function defined in the following equation 1, which calculates the distances between the three vectors. Over multiple epochs, the model adjusts its parameters to ensure that embeddings derived from projects of similar genres are closer in the vector space, and those from unrelated genres are farther apart. As previously discussed, the context of each block is measured both locally and globally, requiring the training process to be divided into two separate processes according to the attention range using two different instances of the model.

At the end of the training process, the model will have learned to generate *embeddings* that capture the structure and internal patterns of the code for a specific genre.

Work in Progress

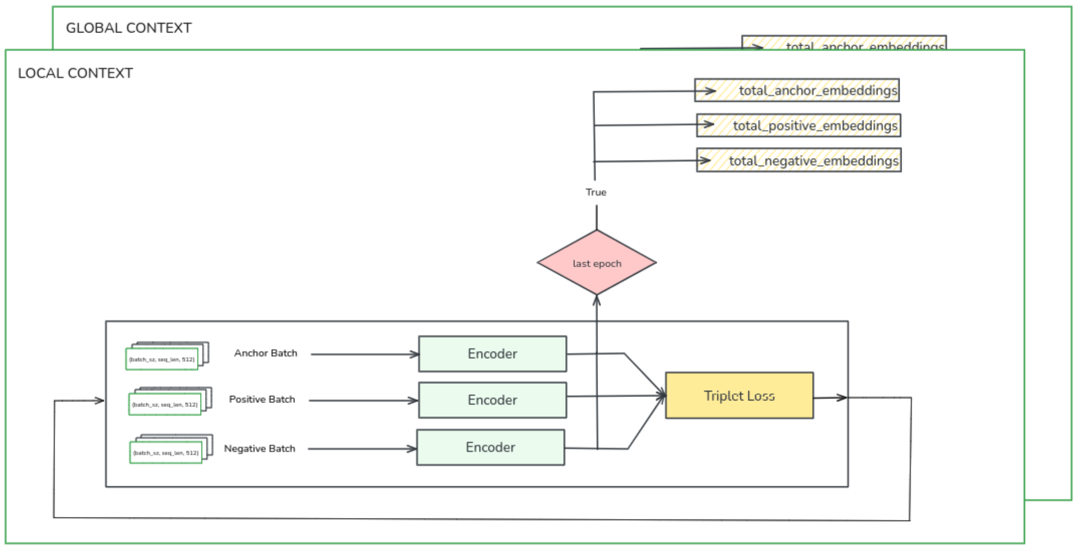


Figure 21: Epoch Flow for *Global* and *Local* Contexts

The scheme 21 is used for the training process in both *local* and *global* contexts, and the lists of embeddings are concatenated similarly for both contexts.

In this way, we enrich the contextual information of each block by merging and subsequently weighting both *embeddings*. To obtain the reference embeddings, since each block can appear in multiple contexts, generating different embeddings in each occurrence, an aggregation process has been implemented to obtain a unique representation per block. To achieve this, we iterated over the sequences, extracting their embedding and storing it in a data structure that indexes each unique block present in the training sequences, along with all the embeddings of its different contexts.

For each block, the mean of all its contextual embeddings is calculated, producing an embedding vector of the same dimension (in our case, 512 dimensions), which captures the average of the contextual representations of the block in question.

Work in Progress

Evaluation (WORK IN PROGRESS): To evaluate a specific project, the following steps are taken: First, the sequence of code blocks from the project is taken, and its *local* and *global* embeddings are generated. Then, these embeddings are merged and weighted to obtain a unique representation of the project. This process involves combining the contextual embeddings generated for each block throughout its appearance in different contexts.

Once the resulting embedding for the evaluated project is obtained, it is compared with the reference embeddings generated during the training phase. The reference embeddings are obtained by averaging the contextual embeddings of each block in the training sequence, as detailed above. Thus, for each block, the mean of all its contextual embeddings is calculated, resulting in an embedding vector of the same dimension (512 dimensions in our case), which captures the average of the contextual representations of the block in question.

For comparison, similarity metrics, such as cosine similarity 2 and Euclidean distance 3, are used to quantify the distance between the embedding of the evaluated project and the reference embeddings for each block present in both the reference projects and the specific project. Since the reference projects will always represent greater diversity due to their number, it is very likely that some blocks from the reference projects may not be found in the specific project. These metrics allow us to identify how similar (or creative) a project is compared to others.

Cosine Similarity:

$$\text{Cosine Similarity}(x, y) = \frac{x \cdot y}{\|x\|\|y\|} \tag{2}$$

Euclidean Distance:

$$d(x, y) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2} \tag{3}$$

In both formulas, $\vec{x}$ is the embedding vector of a specific project and $\vec{y}$ is the embedding vector of the reference projects.

Results

To perform a preliminary test as an initial approach to the validation process, a *dataset* consisting of a total of 624 Scratch files in *sb3* format has been used. Of this set of projects, 303 are categorized under the genre *Action*, while the remaining 321 belong to other genres such as *Storytelling*, *Quiz*, *Puzzle*, or unspecified.

Following the methodology previously described in the article for the training phase, high-dimensional *embeddings* defining the vector distributions between different Scratch blocks for the *Action* genre were obtained. Although this approach to measuring programmatic relationships within a genre can be applied to other genres, it is the most equitable way to establish programmatic reference behaviors for a specific genre.

For the visualization of the reference embeddings obtained in the training phase, a t-SNE (t-Distributed Stochastic Neighbor Embedding) representation was used, which is a dimensionality reduction and visualization algorithm particularly useful for exploring high-dimensional data.

Work in Progress

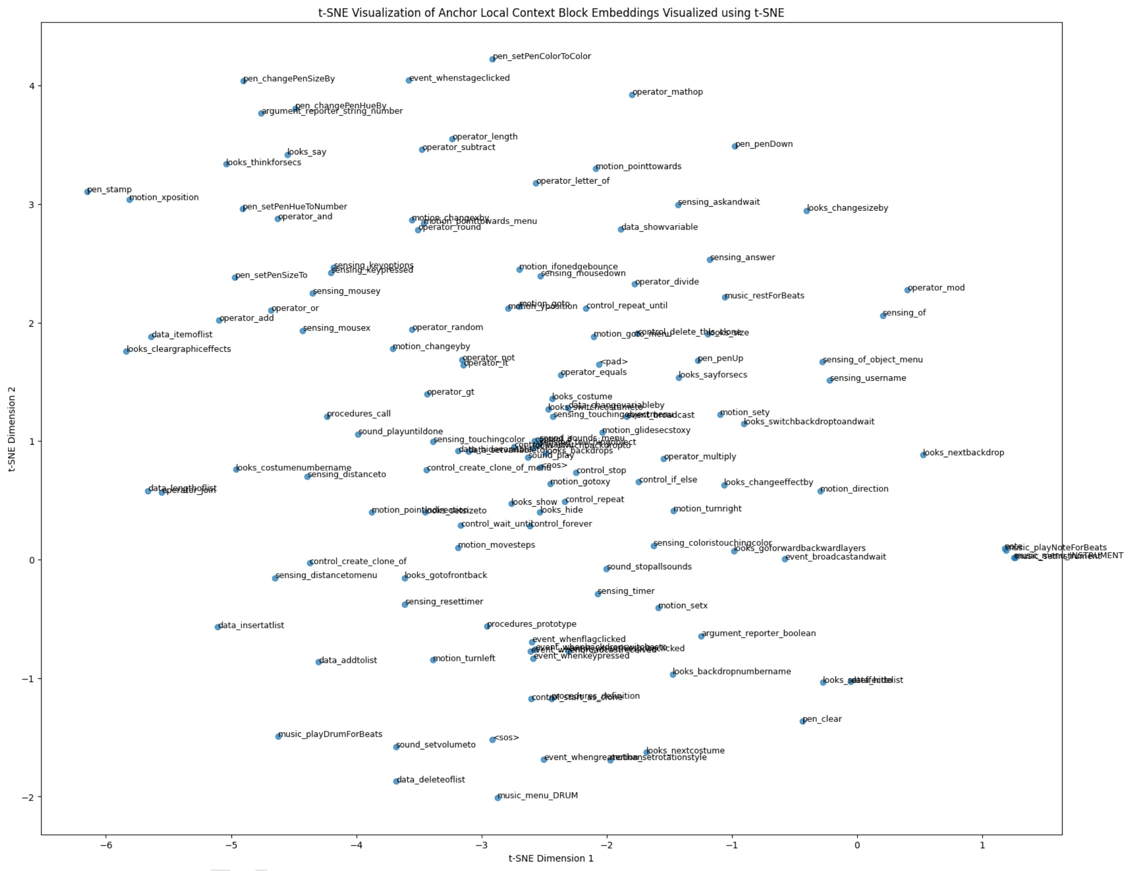


Figure 22: Anchor *local context* embeddings for the "Action" genre

Work

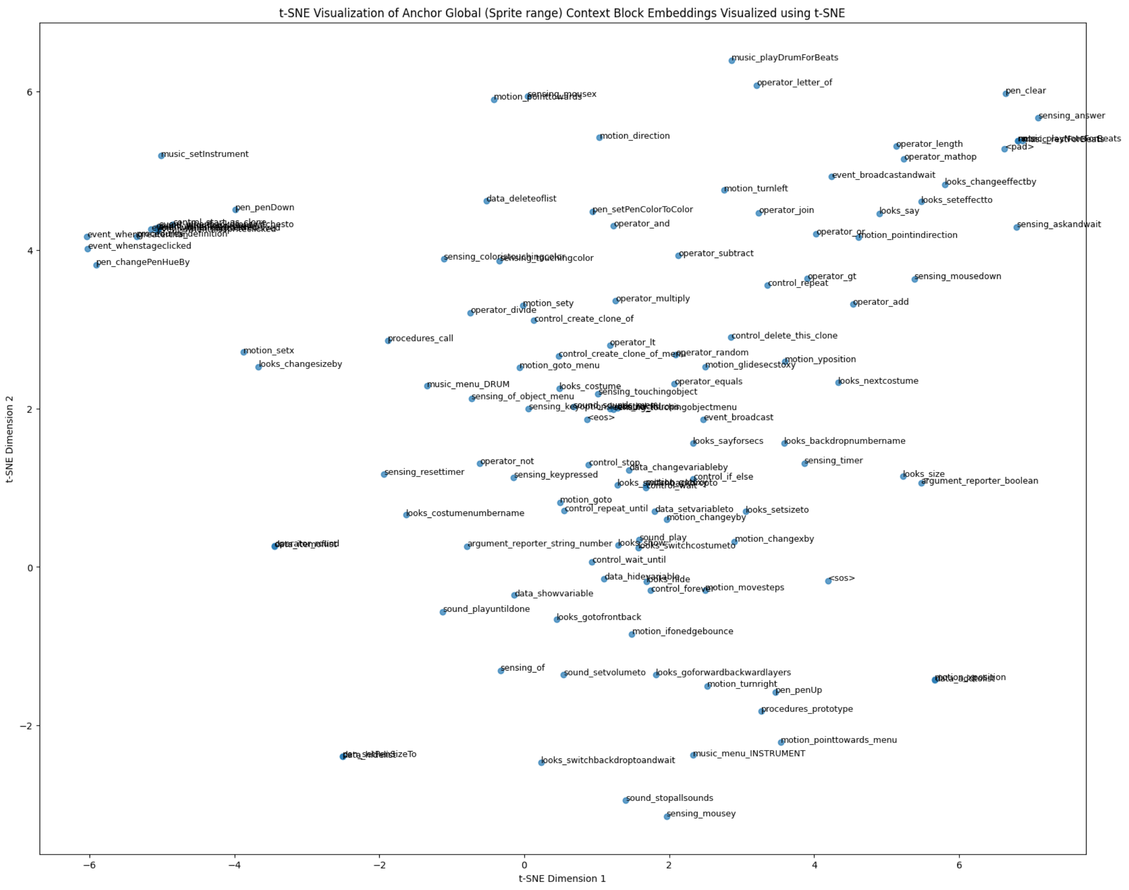


Figure 23: Anchor *global context* embeddings for the "Action" genre

To combine both contexts, an analogous procedure was used, where for each unique block present in each of the contexts, a weighted average of the *local context* embeddings with the *global context* embeddings was calculated. This resulted in a data structure similar to the existing one, where the global embeddings capture, for each block, both the local context at the script level and the *global context* at the sprite level.

The graphs illustrate how blocks are grouped or separated based on their contextual embeddings, providing a visual insight into the semantic relationships learned by the model.

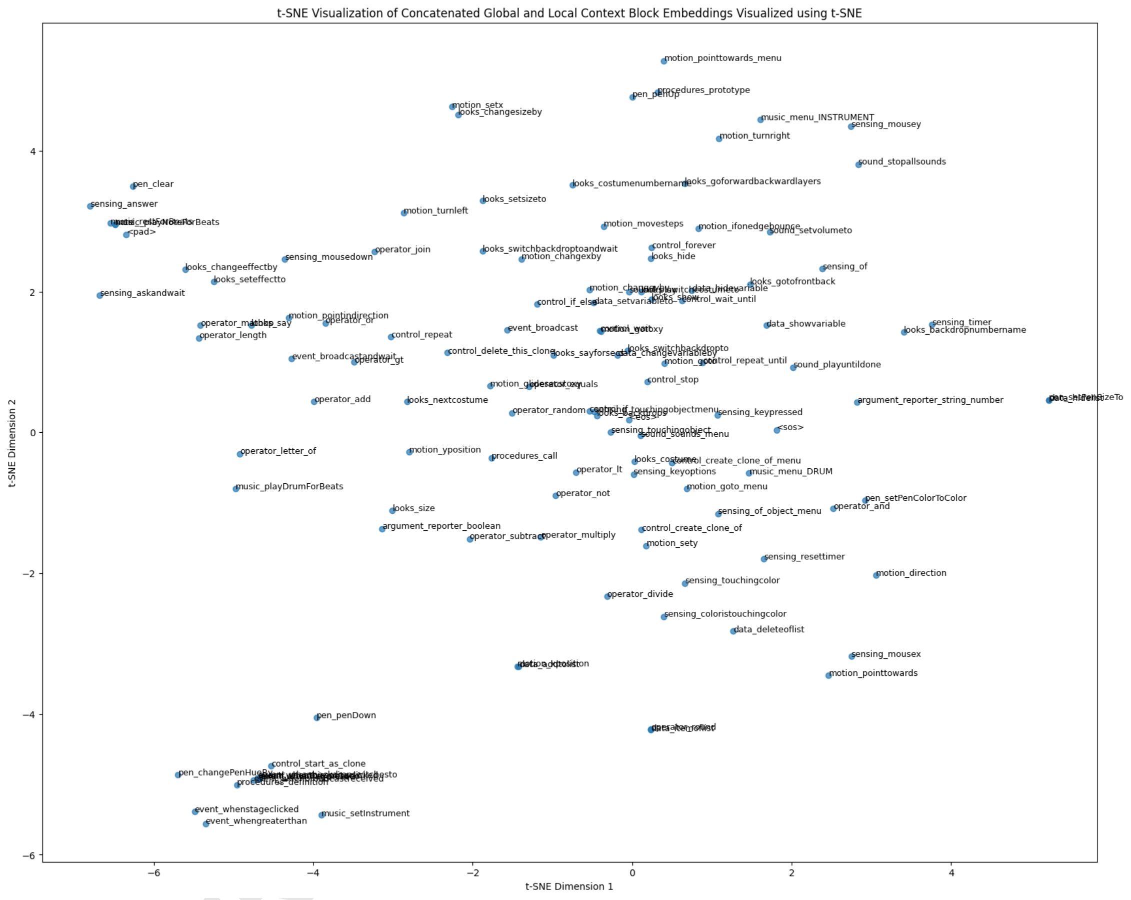


Figure 24: Weighted *global* and *local context* anchor embeddings for the "Action" genre

Discussion

It is important to highlight that throughout the article, the specificity of each genre has been emphasized, as certain practices that may be relatively common in a given genre can turn out to be highly complex in another. This is the case for genres such as *shooters*, where the technical demands to develop a game are not as high as in more narrative genres, where the nature of the genre does not require high complexity in programming, often leading to repeated behaviors.

By examining the *global context* embedding graph of the anchor 24, it is possible to observe the semantic and programmatic relationship between different blocks. Small groups of blocks that are very close to each other are highlighted, which could indicate a specific and recurring pattern in the "Action" genre.

In this article, a dataset has been used that, although categorically distinguishes between "Action" and "No Action" genres due to the nature of the projects, presents certain ambiguities in this classification. Consequently, the projects included in the "Action" category do not adequately represent the general characteristics of this genre and, in general, are "very simple".

References

[1] J. Moreno-León, G. Robles, and M. Román-González, "Dr. Scratch: Análisis Automático de

Proyectos Scratch para Evaluar y Fomentar el Pensamiento Computacional," *Revista de Educación a Distancia (RED)*, no. 46, 2015. [Online]. Available: https://revistas.um.es/red/article/view/240251.

[2] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is All You Need," in *Proc. 31st Advances in Neural Information Processing Systems (NeurIPS 2017)*, Long Beach, CA, USA, 2017, pp. 5998–6008.

[3] F. Schroff, D. Kalenichenko, and J. Philbin, "FaceNet: A unified embedding for face recognition and clustering," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 815-823.

[4] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2013.

[5] J. McCormack and T. Gifford, "Evaluating creativity in computational systems," *Journal of Artificial Intelligence Research*, vol. 69, pp. 399-423, 2020.

[6] L. van der Maaten and G. Hinton, "Visualizing Data using t-SNE," *Journal of Machine Learning Research*, vol. 9, no. 11, pp. 2579-2605, Nov. 2008.

[7] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," arXiv preprint arXiv:1810.04805, 2018.

Work in Progress