Classification of Computer Programs in the Scratch Online Community

by

Lena Abdalla

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2020

Author . . . .
Department of Electrical Engineering and Computer Science
January 27, 2020

Certified by . . . .
Andrew Sliwinski
Research Affiliate
Thesis Supervisor

Accepted by . . . .
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Classification of Computer Programs in the Scratch Online Community

by

Lena Abdalla

Submitted to the Department of Electrical Engineering and Computer Science
on January 27, 2020, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Scratch is a graphical programming platform that empowers children to create computer programs and realize their ideas. Although the Scratch online community is filled with a variety of diverse projects, many of these projects also share similarities. For example, they tend to fall into certain categories, including games, animations, stories, and more. Throughout this thesis, I describe the application of Natural Language Processing (NLP) techniques to vectorize and classify Scratch projects by type. This effort included constructing a labeled dataset of 873 Scratch projects and their corresponding types, to be used for training a supervised classifier model. This dataset was constructed through a collective process of consensus-based annotation by experts. To realize the goal of classifying Scratch projects by type, I first train an unsupervised model of meaningful vector representations for Scratch blocks based on the composition of 500,000 projects. Using the unsupervised model as a basis for representing Scratch blocks, I then train a supervised classifier model that categorizes Scratch projects by type into one of: “animation”, “game”, and “other”. After an extensive hyperparameter tuning process, I am able to train a classifier model with an F1 Score of 0.737. I include in this paper an in-depth analysis of the unsupervised and supervised models, and explore the different elements that were learned during training. Overall, I demonstrate that NLP techniques can be used in the classification of computer programs to a reasonable level of accuracy.

Thesis Supervisor: Andrew Sliwinski
Title: Research Affiliate

Acknowledgments

I begin by thanking God (Allah) for blessing me with completing this thesis, for all of the endless blessings He has bestowed upon me in my life, those I realize and those I don't, and for the ability to thank Him in the first place. Alhamdulilah (All Praise is due to Allah).

It would be foolish to think that anyone makes it anywhere in life on their own. I am so grateful for so many people, too numerous to list here, and am at a loss for words to truly express that gratitude.

I would like to thank my supervisor, Andrew Sliwinski, for his *incredible* support throughout this thesis. Thank you for being so patient, encouraging, supportive, flexible, available, consistent and understanding. You have been instrumental in all aspects of this thesis, whether it be the technical, conceptual, or writing parts. *You truly are a great teacher.* Thank you! I would also like to thank Karishma Chadha for being my co-mentor on this thesis. Thank you for saving me countless times, whether it was with writing scripts to handle Scratch projects, or running an annotation session with the team, or your thoughtful insight and feedback throughout this project. Thank you for being so encouraging, friendly, and such a wonderful and pleasant person to work with and be around.

I would also like to thank everyone at LLK and the Scratch Team for being so welcoming and supportive throughout my time at LLK. Whether it was a heartfelt “How's everything going?”, a welcoming smile, an “Is there any way that I can help?”, or the *endless* hours annotating Scratch projects for my thesis, it really made all the difference! This project is as much mine as it is yours. I would be remiss if I did not give a special shout out to the *superstars* that put in *so* many hours annotating Scratch projects for my thesis – you know who you are and you guys are amazing.

I would like to thank my parents, although “thank you” always seems insufficient. Thank you, Mama, for all of the hard work that no one sees you do, for always being a source of love, care, and service to us, and for always knowing what I needed, even when I didn't. Thank you, Pops, for always being a source of support, guidance, wisdom, and, most importantly, friendship. For always giving me space to grow and mature, and for instilling in me the concept of hard work, of which this thesis is a product. Everything I've accomplished and will accomplish goes back to you both.

I would like to thank my wonderful brothers, Ali and Sasha. Thank you, Ali, for being an endless source of support, upliftment, laughter, wisdom, and criticism (when needed). For always being only a phone call away, for sharing many of my life's troubles with me, and for making me laugh in the moment, and laugh *again* at past jokes that are so funny, they become timeless. Thank you for being my bestest friend. Thank you, Sasha, for truly being the big brother I always needed, for completely taking care of so many of my burdens, and for always amazing me with your incredible life and adulting skills. For always being a text away, for answering my endless specific questions, for doing so much for us that goes unnoticed, and for always reminding me of the important things in life when I need it.

I would like to thank my many, many friends, cousins, and mentors who have been beacons of light in my life. I am *so* grateful for the fact that you are *too many* to enumerate here, but know this: your support, encouragement, and overall friendship have influenced and uplifted me in so many ways. I have and continue to learn so much from you. I would be remiss not to specifically mention three of my closest friends, who I am forever indebted to: Maha Shady, Jana Saadi, and Rowaa Abdalla. Thank you for being my *sisters*.

I end by thanking and praising God again.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

"How can we accurately classify Scratch projects by their type using machine learning techniques?"

Throughout this paper, I discuss the use of machine learning techniques to classify computer programs which were created using the Scratch graphical programming language. To accomplish this task, I created a taxonomy of computer programs shared in the Scratch online community, and developed a novel application of both unsupervised and supervised Natural Language Processing (NLP) techniques. This work resulted in a system that is able to meaningfully distinguish between computer programs using our taxonomy.

Scratch is an online platform, featuring a block-based graphical programming editor, that enables children to create rich, interactive media, ranging from games to animations to stories. In addition, Scratch is a social community, where kids can share their projects, collaborate on ideas, and receive feedback on their creations [10]. Users, called “Scratchers,” snap blocks of code together like LEGO pieces to create a program (see Figure 1-1) [37]. In short, Scratch empowers children to create projects based on their passions together with peers in a playful spirit [38]. With over 50 million users registered, and 48 million projects shared in the online community to date [11], Scratch reaches and engages a wide population.

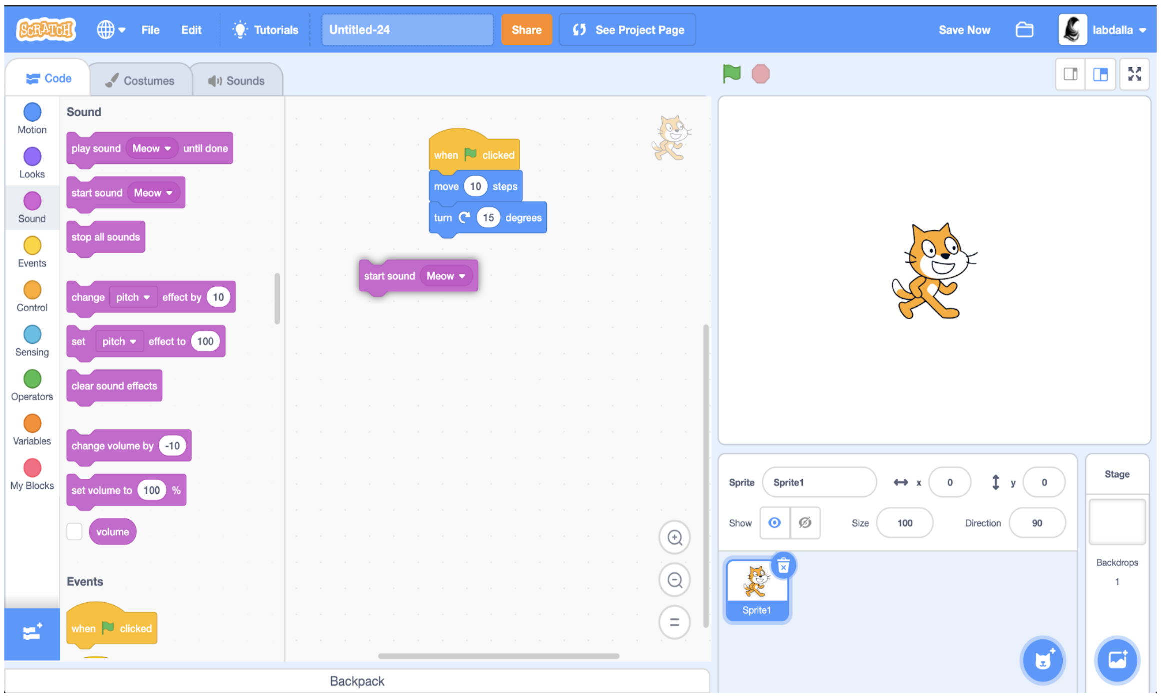
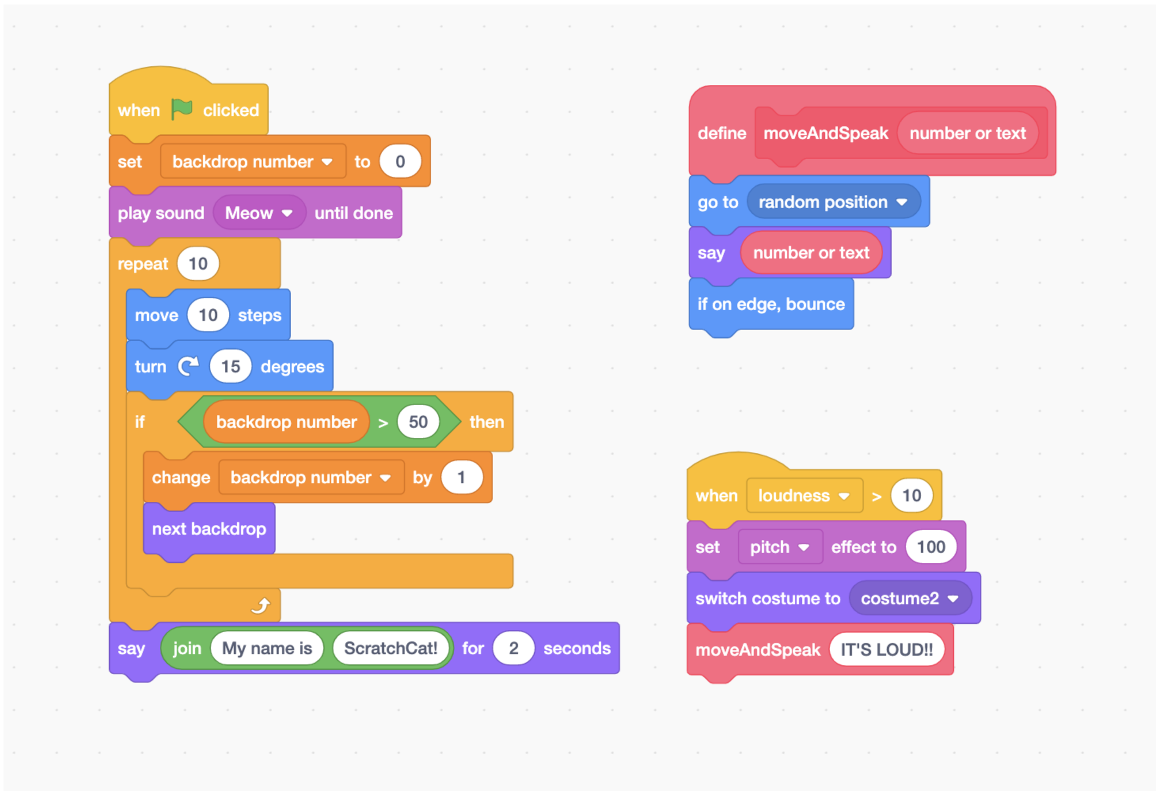


Figure 1-1: Scratch programming editor page. Users drag and drop blocks of code into the editor to create a program.

In Scratch, blocks have different shapes and can be arranged in a myriad of ways (as shown in Figure 1-2). Much like a LEGO brick [37], the shape of each Scratch block contains affordances and constraints that dictate how it can be attached with other blocks. Some blocks are “hat” blocks and can only be used to start a procedure, with no blocks being attached on top of them. Other blocks have an oval shape, meant to indicate that they can be attached as arguments to other blocks. Some blocks have an elongated hexagonal shape, meant to indicate that they represent conditions which evaluate to booleans. Blocks can be arranged into different “stacks”, which are contiguous sequences of blocks. Blocks can also “nest” other blocks, such as in the case of `control_repeat` and `control_if` blocks. Blocks can also be inputs to other blocks, such as in the case of the `operator_join` block being an input to the `looks_say` block.



```
when clicked
set backdrop number to 0
play sound Meow until done
repeat 10
  move 10 steps
  turn 15 degrees
  if backdrop number > 50 then
    change backdrop number by 1
    next backdrop
say join My name is ScratchCat! for 2 seconds
```

```
define moveAndSpeak number or text
go to random position
say number or text
if on edge, bounce
```

```
when loudness > 10
set pitch effect to 100
switch costume to costume2
moveAndSpeak IT'S LOUD!!
```

Figure 1-2: Different block arrangements in the Scratch editor. Blocks have different shapes and can be arranged in a variety of ways.

Blocks are also color-coded to indicate the block category they belong to. For example, all blue blocks belong to the `Motion` category, and include blocks that involve the motion of a character or "sprite" in the program. The different block categories are listed on the left-hand side of the editor.

1.1 Categorizing Scratch Projects

Scratch projects tend to have some commonalities; for example, the type of a project may fall into different categories, including (but not limited to): a game, an animation or a story. Seeking to extract these regularities, and possibly more, prompts us to explore the field of machine learning and apply it to this problem. In this work, we are interested in exploring the field of statistical classification to help us distinguish between different project types.

Currently, the Scratch online community leverages user-defined "tags" to group projects by category. Unfortunately, less than 2% of shared projects actually contain any tags. Within that 2%, however, it's evident that Scratchers are already thinking about project groupings, as they include project tags such as `“#game”`, `“#platformer”`, `“#animation”`, `“#tutorial”`, and `“#coloring contest”`. Given this rich project taxonomy from a Scratcher's perspective, as well as the Scratch Team's perspective on content curation, a new classification mechanism that could be applied to all projects may be called for.

1.2 Why Classify Scratch Projects?

Having a robust mechanism for classifying Scratch projects can be useful in a variety of ways.

First, it can help us improve a user's project discovery experience in Scratch. By developing a robust way of classifying Scratch projects, we would be establishing the foundation for a robust project recommendation algorithm that could suggest projects from the community to a user. Currently, there is an evident disparity in the wealth of attention on the Scratch platform, with only a few users receiving most of the attention in terms of likes, favorites, and views on their projects. Thus, a project recommendation algorithm aimed at redistributing attention across the platform could be useful in tackling this attention disparity issue. Although this work does not explore the development of such a recommendation algorithm, I believe a classification mechanism could pave the way for such important future work.

Second, having a way to classify Scratch projects could prove useful for the Scratch Team and empower them with better methods for analyzing Scratch projects. The ability to automatically divide projects by their type or tag a project with its type could help the Scratch Team uncover trends on the platform, such as estimates of how many projects of each type are being created.

Lastly, better project classification techniques can enhance the project search mechanism on Scratch, providing users with more targeted search results. For example, if a user searches for an “animation” on the platform, they could receive projects that have been categorized as animations by a robust classifier model, as opposed to projects that simply contain the string “animation” in the title or description.

1.3 Outline

In **Chapter 2 (Background)**, I offer background on the Scratch platform as a programming language and a community, and detail the motivation for this work. **Chapter 3 (Related Work)** surveys some of the recent work related to training word vector representations, and applying machine learning techniques on both text and software code corpora. In **Chapter 4 (Methods)**, I describe the methods utilized throughout this project, including constructing the dataset of Scratch projects and their corresponding types, vectorizing Scratch projects by training meaningful vector representations of Scratch blocks (unsupervised learning), categorizing Scratch projects by training a classifier neural network model (supervised learning), and the process of finding the optimal set of hyperparameters (hyperparameter tuning). In **Chapter 5 (Evaluation)**, I evaluate the unsupervised and supervised models’ accuracy, as well as explore the different elements that were learned during the training process. Finally, **Chapter 6 (Conclusion)** concludes this paper by recounting its contributions, and suggesting future directions for improving and utilizing this work.

Chapter 2

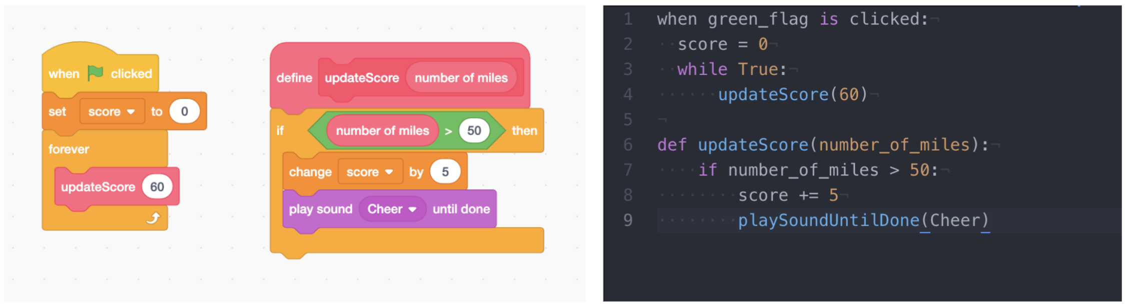
Background

Scratch is a visual drag-and-drop programming language and online community originally developed by the Lifelong Kindergarten Group at the MIT Media Lab. Since its release in 2007, Scratch has been used by millions of children all around the world to create and share interactive stories, games, and animations. Through Scratch children develop the ability to not only "read" the digital world around them, but also "write" it – developing essential skills and creative confidence [37].

The design of Scratch is rooted in the "4 P's" of creative learning: projects, passion, peers, and play. **Projects** form the primary activity in Scratch, with users creating projects (i.e., computer programs) and then sharing them in the online community. **Passion** energizes young people to work harder and longer on projects that are personally meaningful to them. **Peers** form the backbone of the Scratch community, because, as founder of Scratch Mitchel Resnick puts it, "Creativity is a social process, with people collaborating, sharing, and building on one another's work." And, finally, **play** is at the core of the Scratch experience, where kids are encouraged to explore, tinker, and experiment with computational concepts, as a means of realizing their creative potential [38].

2.1 Scratch as a Programming Language

Scratch is a turing-complete programming language which includes features that are common to many other languages including variables, lists, parallelism, recursion, and dynamic typing. By leveraging these features, children can program their own games, animations, stories, simulations, and more.



```
when [flag] clicked
set score to 0
forever
    updateScore 60

define updateScore (number of miles)
if <(number of miles) > 50> then
    change score by 5
    play sound Cheer until done
```

```
when green_flag is clicked:
  score = 0
  while True:
      updateScore(60)

def updateScore(number_of_miles):
    if number_of_miles > 50:
        score += 5
        playSoundUntilDone(Cheer)
```

Figure 2-1: Equivalent programs created using the Scratch programming language (left) and pseudo (Python) code (right).

The design of the Scratch programming language is built on three core principles: **low floor**, **high ceiling**, and **wide walls**. A **low floor** enables novices to get started with creating projects easily, while a **high ceiling** enables experts to create more advanced projects, and **wide walls** enable users to create a wide array of diverse projects [38]. The Scratch Team believes that the diversity of projects that are possible on the platform is key to attracting a diverse user base, including many children who are not traditionally represented in computer science.

The Scratch design and development team have explored the dimensions of computational thinking that Scratchers might engage with through active participation with the platform [19]. This includes **computational concepts**, such as *loops*, which provide a method to execute a specific stack of blocks a number of times, *parallelism*, which provide a mechanism to execute multiple stacks of blocks simultaneously, and *conditionals*, which provide a way to execute blocks only if certain conditions are satisfied, thereby allowing for the possibility of multiple outcomes. In addition to these concepts, children become familiar with **computational practices.** Such practices include *being incremental and iterative*, – writing a little bit of code, trying it out, then making changes as a result – *testing and debugging*, – such as experimenting with different use cases, and soliciting help from others as needed – and *abstracting and modularizing* – starting with an initial idea and then breaking up the different aspects of the code between sprites and stacks of code. Lastly, children develop **computational perspectives** about themselves and their surroundings. Such perspectives include: *viewing computation as a means for self-expression* – feeling empowered to create and express one's ideas using computation – *connecting and creating with others* – being able to collaborate with others on projects as well as having an audience to create projects for – and *questioning the "taken-for-granted"* – thinking about how technologies work and how one can tinker with them.

2.2 Scratch as a Community

In addition to being a programming language, Scratch is also an online community where children can share their creations and interact with each other's projects. From the outset, the Scratch online community was intended to be an essential part of the platform, where children can gain inspiration for new ideas and receive feedback on their projects [37].

Across the platform, Scratchers are encouraged to engage with their peers in a variety of ways, including commenting on each others' projects and profiles, "favoriting" and "loving" each others' projects, joining and contributing to themed project studios with fellow Scratchers, and engaging in forums centered around a specific topic. All of these features provide avenues for Scratchers to form connections with, and learn from, their peers.

Moreover, once a Scratcher shares a project to the community, others are able to interact with that project and explore the code. They can learn new programming practices by examining the code, and can "remix" the project to tinker with it further.

Scratchers collaborate across the platform in a myriad of ways. For example, they offer complementary skills when working together on projects, they contribute to subcommunities, and they offer their skills as services to the community [37].

As Scratchers share their creations in the online community, they learn from and with their peers in many ways, whether it's by way of receiving comments, exploring code, remixing projects, or collaborating on ideas with others.

2.3 Project Taxonomy

Projects in Scratch also tend to fall into certain categories of **type**. Such types include games, animations, slideshows, simulations, and many others. Based on anecdotal evidence from the Scratch Team, games and animations tend to be the most common project types in Scratch, with many sub-varieties of each (e.g., platform games, arcade games, role-playing games). Worthwhile to note is the subjective nature of assigning such categories to a project's type. For instance, defining what a game is and is not differs from person to person, making a generic, all-inclusive definition of a game difficult to arrive at (see Figure 2-2). A portion of this project included spending time trying to construct concise yet comprehensive definitions for the different identified types – namely, "game", "animation", and "slideshow" – but we couldn't arrive at a set of criteria that would accommodate all subcategories of each type, and that would clearly separate between the categories. Expectedly, this difficulty to come up with comprehensive definitions merely reflected the larger debate that exists today around defining some of these very categories, for instance, games [44]. Still, although a consensus is difficult to reach for what each category absolutely is and is not, there is an intuitive categorizing of the projects that many Scratchers and Scratch Team members assign, albeit potentially different from one person to another. One way to reconcile this inability to absolutely categorize projects is to softly categorize them by way of votes, thereby accounting for the different perspectives through consensus.

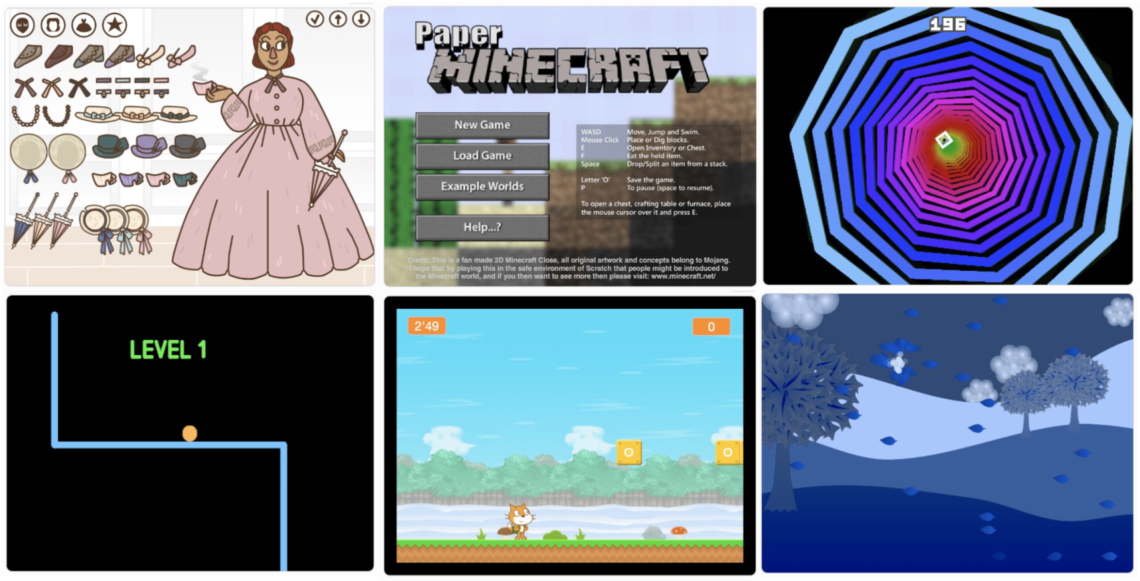


Figure 2-2: Thumbnails of some of the popular games in Scratch.

2.4 Motivation

Peer learning is central to the learning experience in Scratch. Unfortunately, there is a significant disparity in the distribution of attention across the system, with a few popular users getting a lot of the attention. According to current internal Scratch user data, although the average view count for all projects shared in the last year is as high as ~ 13.8, 50% of Scratchers get less than 2 views on their projects, indicating that a relatively small number of users dominate the wealth of attention on Scratch. Similarly, while the average love count for those projects is 0.68, 50% of Scratchers actually get no "loves" on their projects. This disparity in attention proves an obstacle for Scratchers to fully engage in peer learning and reap the benefits associated with it. According to internal Scratch analysis, when looking at a Scratcher's retention (i.e. the amount of time they remain engaged in the platform after their first visit), their retention rate steadily increases with the number of loves and favorites they receive on their **first shared project**. Such a correlation implies that peer recognition and feedback can play a significant role in a Scratcher's engagement on the platform.

In an effort to better understand trends in peer learning, the Scratch team views users as potentially falling into multiple categories of engagement. Of these categories, those relevant to this thesis are: “active” and “engaged”. An “active user” is one who **creates or edits** a project during a given period of time. An “engaged user” is one who **shares** a project during a given period of time. According to internal Scratch user data, when looking at users who joined Scratch in 2019, around 79% of them created projects during that period, but only around 15% shared those projects. This disparity of about 64% can be due to many factors, many of which are related to the social anxiety associated with sharing content to a wide audience.

At the same time, Scratchers have reported that seeing other sophisticated projects on the platform can intimidate novice users, potentially causing them to feel incompetent, and to become demotivated to create their own projects.

Given the positive impact of social connection on a Scratcher’s retention, as well as the overcoming of self-esteem obstacles, it seems worthwhile to try to redistribute the wealth of attention on Scratch. Towards that end, we believe that a project recommendation algorithm aimed at redistributing attention across the platform is called for. The algorithm would suggest projects from the community to a user, thereby increasing the visibility of some projects, and enabling the process of social feedback to positively impact learning.

A key component of any such project recommendation algorithm is its ability to distinguish between different types of projects, such as “game”, “animation”, “slideshow”, and “other”. If an algorithm is able to make that distinction, then it can fetch more targeted examples of projects, making for a more personalized experience for the Scratcher. Thus, developing a robust classification model that is able to determine the type of a project (up to a satisfactory confidence threshold) would enhance the “intelligence” of the recommendation algorithm, and could lead to better recommendations.

Chapter 3

Related Work

3.1 Word Representations

The problem of finding meaningful and effective vector representations for English words has been a long-standing problem in the Natural Language Processing (NLP) field. Traditional vector representations of words include the Term Frequency–Inverse Document Frequency (TF–IDF) measure and one-hot encoding. The TF–IDF value is meant to capture how important a word is to a document in the corpus, and is directly proportional to the number of times a word is repeated across the document and inversely proportional to the number of documents in which that word appears [36]. Thus, common words such as "the" are not weighted disproportionately highly. One-hot encoding is an N-dimensional vector representation of a word in a vocabulary of size N, where only one element in the vector is 1 (namely, at the position corresponding to that word's index) and all other elements are 0 [9].

Recently, it has been found that continuous vector representations of words obtained from training large *unsupervised* datasets of English text provide meaningful high-quality word representations [31]. Mikolov et. al first showed in 2013 that such word representations, trained using the `word2vec` model architecture, perform well on syntactic and semantic word similarity tasks, indicating their meaningful nature. For instance, semantic relations between similar pairs of words are successfully captured by this method, such as in the example of "*Man* is to *King* as *Woman* is to *Queen*".

There are multiple types of `word2vec` architectures, including the Continuous Bag-of-Words (CBOW) model architecture, and the Skip-gram model architecture [31]. In both architectures, the most likely word(s) are predicted given their context. In the CBOW architecture, the most likely *current* word, $w(t)$, is predicted given its surrounding context, which includes some of the words that come prior to it and some of those that come after it in the sentence. In the Skip-gram architecture, the most likely *surrounding* words are predicted given the current word, $w(t)$ (which can be thought of as the "flip" of the CBOW architecture). In both cases, the vector representations correspond to the weight matrix values that are optimized and tuned for these prediction tasks during training.

Word embeddings are *dense low-dimensional* vector representations of words [23]. Their dense and low-dimensional nature make them better suited for use with neural network tasks, as it leads to better generalization power and lower computational requirements [22]. Moreover, using pre-trained word embeddings to represent text has been shown to improve the accuracy of many downstream NLP tasks, as demonstrated by Pennington et. al [34], and Turian et. al's works [41]. In our case, these improvements imply that pre-training word representations on an unsupervised dataset of Scratch projects and using those representations for our downstream project classification task could prove to be advantageous.

Within the area of learning distributed word representations, Pennington et. al developed the GloVe model for learning global word vectors [34]. Their model merges global matrix factorization methods, which are useful for capturing the global co-occurrence statistics of the training corpus, with local context window methods (such as `word2vec`), which are better at uncovering the important linear substructures in the corpus, drawing upon the benefits of both approaches. The model performs better than baseline models on word analogy and similarity tasks, and results in a vector space that contains meaningful substructure [34].

3.1.1 fastText

Facebook's fastText library [2] offers an API for efficient text representation and classification that is based on the `word2vec` approach, but that also varies in a few important ways. Like the original CBOW and Skip-gram models, the fastText model predicts words given their contexts. However, unlike those vanilla models, the fastText model also takes into account the morphology of words, representing each word as a "bag of character n-grams". This "subword" information is typically ignored by the traditional CBOW and Skip-gram models, since each word is treated as a distinct unit and its sub-structure is not taken into account [18].

In the fastText model, each word is represented using its character n-grams. For instance, if the word in question is "where", and $n = 3$, the character n-grams (or 3-grams, to be precise) would include:

$$\begin{gathered}<\text{wh}\\ \text{whe}\\ \text{her}\\ \text{ere}\\ \text{re}>\end{gathered}$$

In the fastText model, a range of n-grams corresponding to different lengths are extracted, with the word itself (shown below) also being included as part of these n-grams [18].

$$<\text{where}>$$

A vector representation is learned for each of the character n-grams as well as the word n-gram. The final vector representation for the word in question is then the sum of these n-gram vector representations. This setup allows for encoding misspelled or fake words, since it simply represents a word using its character n-grams [3]. Moreover, words that have similar sub-word sequences will share character n-gram sequences and their vector representations. This benefit is especially relevant to our work, since blocks can share sub-words, such as those within the same category of blocks or those that have similar functionalities. For instance, `motion_movesteps` and `motion_gotoxy` share the sub-word `motion`, while `looks_setsizeto` and `motion_setx` share the sub-word `set`. Sharing the vector representations of these sub-words between these blocks would thus align their vector representations, which is desirable.

The fastText library is intended to provide an architecture that can be quickly and efficiently trained. Moreover, the fastText architecture was evaluated on word similarity and analogy tasks and was found to achieve state-of-the-art performance in many contexts [18]. In addition to training meaningful word representations, the fastText library also offers an efficient text classification architecture that performs comparably to other deep learning classifier networks [27]. All of these qualities make it a suitable library for our use case of encoding Scratch blocks and projects using low-dimensional vector representations, and then using those representations for project classification. However, given that Scratch projects are not originally encoded in textual format (as English sentences are, for example), we are somewhat "misusing" the fastText architecture to suit our needs. To get around this limitation, I develop a "textification" process that transforms a Scratch project into a body of text, by encoding blocks and transitions between them in a special way. Please refer to Chapter 4 (Methods) of this thesis for an in-depth description of the project textification, vectorization, and classification processes.

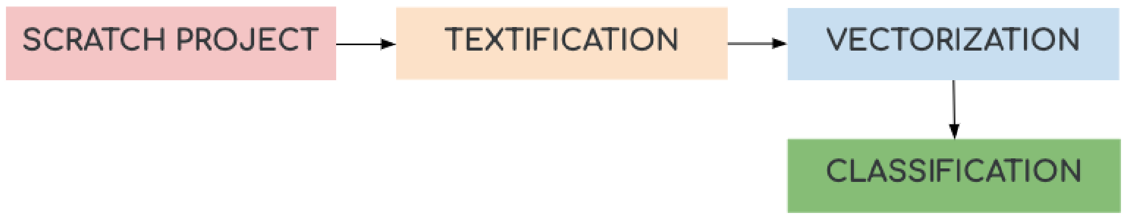


Figure 3-1: Diagram describing the high level process of this work. Each Scratch project undergoes textification, vectorization, and, finally, classification.

Although a Scratch project is not naturally encoded as a body of text, its structure is amenable to being transformed into a textual format. For instance, Scratch blocks have “opcodes”, which describe the functionality of each block. An example of an opcode is `motion_movesteps`, which denotes a block that causes a character to move a specified number of steps in the Scratch program. These opcodes tend to be meaningful and typically correspond to the block’s functionality, so a natural choice is to encode blocks by their opcodes. In this way, we would be somewhat describing the program’s functionality in the form of words. Besides the functionality of blocks, the structure and arrangement of these blocks in the Scratch program can also be encoded using special textual symbols. Much like English words in a sentence, the hope is that training embeddings on these “bodies of text” representing Scratch programs would capture syntactic and semantic relationships between blocks.

3.2 Machine Learning on Software Code

There has been much recent work regarding machine learning applications on software programs and code, with this field garnering the name “Big Code” [15]. Naturally, due to the rich and defined structure of code, as well as the plethora of programs and written code widely available nowadays, applying machine learning techniques to learn from such data is a field that has been (and continues to be) widely explored.

Similar to natural languages, programming languages serve as a medium of communication [15]. They have rich and strict syntactic rules and a complex grammar, lending themselves nicely to prediction and comprehension tasks. Not only that, but similar to natural language, it turns out that the use of programming languages (i.e. code written by humans) tends to be repetitive and predictable, thereby making way for statistical language modeling [26]. In 2012, Hindle et. al showed that this hypothesis holds empirically by applying natural language machine learning techniques to written software code, yielding impressive results [26].

In a similar vein, White et. al demonstrated the use of deep learning to construct software language models, proposing a feed-forward, Elman recurrent neural network for the task [43]. Using a corpus of over a 1,000 Java projects from Github, they showed that their deep neural network outperforms the baseline n-gram model, when considering model perplexity (PP) as the primary metric for comparison. Furthermore, their model also did better than the baseline n-gram model at the software engineering task of code suggestion.

In terms of determining appropriate program features for learning, it appears that hand engineering the features would be too time-consuming and ad hoc, and may yield suboptimal results [33]. In contrast, using a neural network to determine the important features seems like a worthwhile endeavor; besides the advantage of not having to curate the features by hand, a deep neural network is also able to model complex nonlinear features and functions [33]. Moreover, prior insight on the data is not required for neural nets [33], allowing room for learning from preliminary experiments with neural network architectures to glean such insight.

Although code has a very rich structure and syntax, the task of finding good representations for software programs has proven to be a non-trivial problem. In the context of neural networks, Peng et. al suggest that feeding random program embeddings without any pre-training results in vanishing or exploding gradients during backpropagation, yielding poor learning [33]. Instead, they propose a "coding criterion", which uses deep neural networks to learn program vector representations of the nodes of the program's abstract syntax tree (AST). In essence, each node's representation is a combination of its children nodes' representations. This serves as a pre-training method to construct meaningful initializations of program vectors that are then fed into a neural network for learning. In a subsequent paper, they show how to use Convolutional Neural Networks (CNNs) over trees constructed using the above "coding criterion", where they use tree-based kernels to infer program's structural details [32].

There has been recent work in the area of finding meaningful vector representations for software code. Recently, Alon et al proposed the code2vec approach, which aims to learn continuous distributed vector representations of code, or "code embeddings" [17]. These code embeddings are meant to capture the semantic properties of the code snippet. To achieve this goal, they use an attention-based model, and represent the code snippet as a “bag of paths”. First, these syntactic paths between leaf nodes are extracted from the code snippet’s abstract syntax tree (AST), and “path-contexts” are constructed by including the *values* of the leaf nodes (in addition to the path itself). Then, given this bag of “path-contexts” corresponding to the code snippet in question, distributed vector representations are learned for each path-context jointly along with how best to aggregate these embeddings (using attention) into a single code embedding that would represent the code snippet [17].

Alon et. al evaluate their approach with a method name prediction task, which predicts the method name corresponding to the method’s body, using its distributed vector representation, and show that their model is both successful in carrying out this task and surpasses previous works’ model accuracies. Moreover, method name vectors are also learned during the training process, and are shown to capture semantic similarities and analogies [17].

Another attempt at learning feature embeddings of programs was recently undertaken by Piech et. al in their paper *“Learning program embeddings to propagate feedback on student code”.* They propose a method to represent programs as linear mappings from a precondition space to a postcondition space: given that a precondition is satisfied in an environment, their method aims to learn features of the program in question, that would help in predicting the postcondition (i.e. outcome of executing the program) in the environment [35]. Their basic model for learning the function of a program is to learn a *nonlinear* feature transformation of the precondition and postcondition vectors, and then linearly relate those feature vectors using a “program embedding matrix”, that represents the program in question. Moreover, they are interested in propagating automatic feedback to ungraded programs, using only a few human graded and annotated programs. Thus, this active learning problem can be thought of as an N binary classification task, but since it involves more than just considering the function of a program, the authors use an RNN architecture that resembles the structure of the program’s abstract syntax tree, to refine the program embedding matrix [35].

Beyond representing software programs as real-valued vectors, there has been recent research exploring their representation with graphs. Allamanis et. al explore a hand-designed approach for constructing program graphs that capture the syntactic and semantic details of a program [16]. Composed of a set of nodes, node features, and directed edges, the graph is primarily based on the program's abstract syntax tree (AST). As such, the graph contains syntax nodes and tokens, edge relations between children nodes to capture their ordering, as well as graph structures to represent the control and data flow of the program. As an example, `LastRead`, `LastWrite` and `ComputedFrom` edges between variable nodes are included to capture the semantic structure of the code. Armed with this program graph, the authors then use Gated Graph Neural Networks (GGNN) to learn representations of the nodes as state vectors. The two tasks they use to evaluate their model are "VarNaming" and "VarMisuse" tasks, aimed at predicting variable names using their usage context, and predicting the right variable that should be used at a specific program location, respectively. The learned node representations from the GGNN thus allow the authors to infer context and usage representations of variables and "empty slots", to aid in these tasks [16].

Another common application of analyzing source code is detecting vulnerabilities in programs. Russell et. al propose a system to detect vulnerabilities at the functional-level using an ensemble classifier. To construct the dataset, they use static analyzers to provide the labels for the code vulnerability of the functions. However, their method promises to offer more than the static and dynamic analysis of rule-based tools. Source code tokens are embedded as matrices and then a Convolutional Neural Network is used for extracting features from the representations. Finally, learned features are then fed into a Random Forest classifier. They also experiment with neural networks for classification, but surprisingly, found that using a Random Forest classifier worked best [39].

Li et. al propose another way to detect code vulnerabilities, namely using "code gadgets" to represent software programs. These are groups of semantically-related lines of code that are not necessarily adjacent in the program. These features are encoded into vectors, after which they're fed into a Bidirectional Long Short-Term Memory (BLSTM) neural network for training. In addition to constructing the VulDeePecker [28] system to detect vulnerabilities, they contribute a set of guidelines for vulnerability detection using deep learning, including program representation, appropriate granularity of a representation, and selection of appropriate neural network architectures [28].

Chapter 4

Methods

4.1 Constructing the Dataset

A significant portion of this project involved constructing a dataset of Scratch projects and their corresponding type, in order to be able to use supervised learning techniques to classify Scratch projects. The resulting dataset consisted of 873 projects and their labels, obtained using consensus-based annotation by experts.

The task of annotating Scratch projects by their type requires knowledge of the Scratch platform and familiarity with Scratch projects. Thus, in order to get high-quality annotations, such a task could not have been opened up to the general public (through services such as Amazon Mechanical Turk [1] or Figure Eight [5]). Instead, this task was better suited for members of the Scratch team to fulfill, given their extensive expertise and familiarity with Scratch projects. The requirement of a specialized group of people to carry out the annotations meant that there was a smaller pool of annotators, which led to the annotation process taking a longer period of time.

To construct the dataset of Scratch projects for annotation, the following methodology was used:

- A total of 9,937 projects were pulled from the Scratch online community at random

- The projects had the following restrictions:
  - Their "language" field is "english"
  - Projects must have at least 25 blocks
  - Projects must have at least 25 views on the platform
  - Projects must not include the string "Untitled" in their title (which is a default string that is added to a project's title when it gets saved on Scratch without a title being specified)
  - Projects must be created after January 1st, 2012 (after the launch of Scratch 2.0)
  - Projects must not have more than 500 sounds and 500 costumes (to eliminate projects that would take too long to load)

  These restrictions were enforced to ensure that the projects were more "established" or "finished" projects (and not simply starter or "test" projects), were released after Scratch 2.0, and were not too large. Large projects take a long time to load on the Figure Eight system and would considerably slow down the workflow for annotators. Including large projects in the dataset would have resulted in an inconvenient and time-consuming process for annotators who were trying to go through as many projects as they could in a limited amount of time. In an effort to make the process as smooth as possible for annotators, we opted to remove large projects from the supervised dataset.

- Of these 9,937 projects, a random subsample of around 2000 projects were chosen.

- Around 39 handpicked animations and 47 handpicked slideshows were manually added to the dataset, to ensure that there was a base level of those types of projects in the dataset. Given that the dataset was randomly pulled from the Scratch backend, and there's currently no way to specify the type of Scratch projects to pull, adding in these handpicked animations and slideshows was a way of ensuring there were enough of these types of projects for the classifier model to subsequently train on. Games tend to be a very common project type in Scratch, and were represented well in the dataset.

I utilized the Figure Eight platform to gather annotations for our dataset of Scratch projects. I first set up a "job" in the platform that included our uploaded dataset of Scratch projects, where each "row" in the job represented a specific Scratch project. Each row is accompanied by a set of questions that "contributors" must answer. In our case, the "contributors" were members of the Scratch team and the job was only accessible to them.

At the start of this work, we had determined that the general categories of projects include the following: "game", "animation", "slideshow" or "other". Games and animations are among the most popular project types on the Scratch platform, and are general enough to include many sub-varieties. We chose to include "slideshow" as a separate category in the beginning because it was a category of projects that clearly differed from games and animations, but was distinct enough to garner its own category. As we'll see in the Section 4.3 – Classifying Scratch Projects later, we ended up lumping projects tagged with the "slideshow" label into the "other" category, as there ended up being relatively few "slideshow" projects in the supervised dataset and likely not enough for training an accurate classifier model.

Our end goal for the annotation phase was to obtain high-quality annotations for the category of each project – i.e. "game", "animation", "slideshow" or "other". However, given that each category can be defined subjectively and can include a wide range of projects, asking the right questions that would help guide contributors to arrive at an accurate label for each project's type was paramount. To that end, we experimented with a few iterations of these questions to explore which set was suitable.

For the first few iterations, we included some starting questions that were meant to guide the contributors to focus on specific qualities of the project, to help narrow down the contributor's thought process while categorizing projects. We consulted the Scratch Team to determine which qualities of a project were important in differentiating one type from another, and we based the first two annotation questions on those qualities (shown below). We then asked the contributor to categorize the project in the third and final annotation question. Below I list the different iterations of questions, with the changes between iterations marked in bold.

The **first** iteration of questions used included the following:

1. Is there user interaction in this project?
   - This project requires user interaction.
   - This project includes some user interaction, but it's optional.
   - This project has no user interaction.
2. Does the user move through the project on a (primarily) variable or predefined path?
   - The path of this project is primarily predefined and is the same regardless of user action.
   - The path of this project is primarily variable and that variability depends on user action.
   - The path of this project is primarily variable but the variability does not depend on any user action.
3. What category best describes this project?
   - Game
   - Animation
   - Slideshow
   - Other

The **second** iteration of questions used included the following:

1. Is there user interaction in this project?
   - This project requires user interaction.
   - This project includes some user interaction, but it's optional.
   - This project has no user interaction.
2. Does the user move through the project on a (primarily) variable or predefined path?
   - **The path of this project is primarily predefined.**
   - **The path of this project is primarily variable.**
3. What category best describes this project?
   - Game
   - Animation
   - Slideshow
   - Other

The **third** iteration of questions used included the following:

1. Is there user interaction in this project?
   - This project requires user interaction.
   - This project includes some user interaction, but it's optional.
   - This project has no user interaction.
   - **Could not answer the questions for this project.**

2. **Would the creator of this project be able to define all of the outcomes of this project? Or would they only be able to define the user's behavior?**

    - **The creator of this project would be able to define both the user's behavior in the project and all of the outcomes of this project.**
    - **The creator of this project would only be able to define the user's behaviour in the project.**
    - **Could not answer the questions for this project.**

3. What category best describes this project?

    - Game
    - Animation
    - Slideshow
    - Other
    - **Could not answer the questions for this project.**

The **fourth** and final iteration of questions used was:

1. What category best describes this project?

    - Game
    - Animation
    - Slideshow
    - Other
    - Could not answer the questions for this project.

For each iteration, we incorporated the Scratch Team's feedback when making changes and adding new questions. However, after many discussions and trial runs, it became apparent that the starting questions caused confusion and were counter-productive to their original aim. Although these questions tried to get at a certain intuition, getting their phrasing right was a significant challenge. As a result, we decided to remove them altogether, and keep the third question as the only annotation question, giving rise to the final iteration shown above. Please refer to Figure 4-1 for an example of how the project and corresponding question appeared for this final iteration.

The Figure Eight platform offers a few different job settings that enable high-quality annotations. In any Figure Eight job, there are two types of questions: “work” and “test” questions.

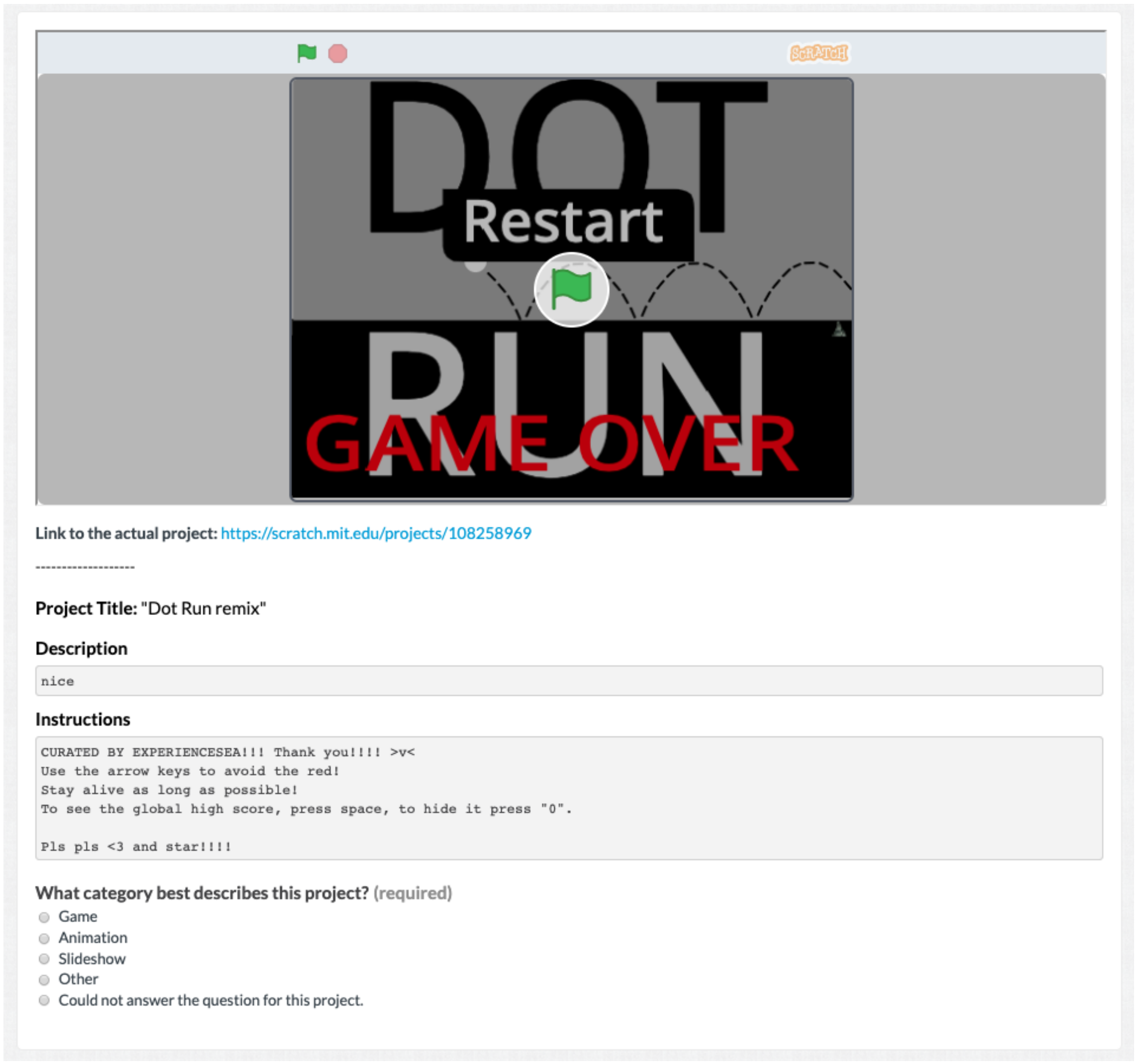
Work questions are simply the rows of the dataset that the job owner wishes to collect annotations for. These questions form the crux of the job, since annotating these rows is the main task that the job owner wants to crowdsource. Test questions are meant to verify whether contributors understand the task at hand, and whether they are consistently producing high-quality annotations. The job owner chooses certain rows in the dataset to become “test questions”, and specifies the accepted answers for those questions. These questions are then presented to the contributor throughout the job in a hidden manner, where they are indistinguishable from work questions. Each contributor has a running accuracy score that is updated as they work through the job. This score is based on whether their annotations fall within the accepted range for test questions. The platform allows for the job owner to specify a minimum accuracy score that contributors must maintain as they work through the job [6]. In our case, we set the minimum accuracy score to 80%.

At the start of every job, the contributor begins in Quiz Mode, which comprises of a few test questions that the contributor must fully answer correctly. This phase is intended to ensure that all contributors have an understanding of the task at hand to begin with. Once the contributor successfully completes Quiz Mode, they then transition to Work Mode, where they are presented with a specified number of work and test questions per “page” of work [6]. In our case, each page of work contained four questions in total: one hidden test question, and three work questions.

The Figure Eight platform presents each row in the job to a specified number of contributors, and aggregates their responses into one final annotation. This aggregation process resembles a weighted voting scheme, where the responses are weighed based on the accuracy levels of the contributors, and the highest scoring response is chosen as the final annotation. The resulting score of the final annotation represents its "confidence score" [7].

The Figure Eight platform also allows the job owner to specify the desired number of "judgements" to be collected per row, the maximum number of "judgements" to be collected per row, and the minimum confidence score for each row. Each unique contributor's response for a specific row is considered a "judgement". In our case, we set the desired number of judgements per row to be 3, the maximum number of judgements per row to be 5, and the minimum confidence score per row to be 0.8. These settings mean that the platform will collect a total of 3 judgements for each row by default, unless the confidence score for that row is below the minimum (0.8), in which case two additional judgements are requested, for a total of up to 5 judgements per row. In summary, these settings allow the platform to "dynamically" collect more judgements for each row in an effort to reach contributor consensus on an annotation [6].

Test questions form an essential part of the benefits offered by the Figure Eight platform and ensure that contributors maintain a base level of accuracy. As a result, each page of work must include at least one hidden test question. The implication of this requirement is that the job owner must specify as many test questions as their desired number of pages of work. Choosing appropriate test questions can be a tedious and time-consuming process. Fortunately, the Figure Eight platform allows job owners to convert completed rows into test questions – after a specific number of rows have been completed in the job – as a way of facilitating that process [6].



Link to the actual project: https://scratch.mit.edu/projects/108258969

Project Title: "Dot Run remix"

Description

```
nice
```

Instructions

```
CURATED BY EXPERIENCESEA!!! Thank you!!!! >v<
Use the arrow keys to avoid the red!
Stay alive as long as possible!
To see the global high score, press space, to hide it press "0".

Pls pls <3 and star!!!!
```

What category best describes this project? (required)

- ○ Game
- ○ Animation
- ○ Slideshow
- ○ Other
- ○ Could not answer the question for this project.

Figure 4-1: Example project and corresponding question(s) from the Figure Eight system, as it would appear for a contributor.

Throughout the annotation phase, we faced a recurring tradeoff between the quality and quantity of annotations. In order to deliver high-quality annotations, contributors needed to spend time interacting with each project until they felt confident in their categorization of its type. On the other hand, in order to get many projects annotated, we needed contributors to go through a large amount of projects in a short amount of time. Given that many of the Scratch Team members had other important work responsibilities to fulfill, there were only so many annotations we could gather in the limited time frame that we had.

As a result, the annotated dataset was relatively small in size, coming out to 873 annotated projects with a confidence of greater-than or equal to 80%.

4.2 Vectorizing Scratch Projects

Once I had the labeled dataset of Scratch projects and their types, the next step was to vectorize them in an effective manner, in order to feed these vectors as input to the classifier neural network. As discussed earlier in Chapter 3 (Related Work), the most promising method for vectorizing these projects was via unsupervised training of word embeddings on a random selection of 500,000 Scratch projects. Given the recent success of using word embeddings to represent natural language words and sentences in boosting the performance of several machine learning tasks, we anticipated that embeddings would also work as a better representation mechanism for Scratch blocks and projects.

In lieu of vectorizing projects using traditional methods such as TF-IDF or one-hot encoding, we decided to focus on training block and project embeddings on a large unsupervised dataset of Scratch projects. Through this unsupervised training phase, we hoped to train efficient and meaningful embeddings, that would adequately represent Scratch blocks and projects, and sufficiently capture the relationships between the different Scratch blocks. Given that our labeled dataset of Scratch projects was small in size, we hoped to overcome this limitation and boost the accuracy and performance of our classifier neural network by ensuring that we represented Scratch projects in an efficient and meaningful manner. Since the process of training embeddings is an unsupervised task and doesn't require any project labels, we can leverage the virtually unlimited amount of projects in the Scratch community and use an arbitrarily large dataset of Scratch projects for unsupervised training. Training embeddings using a large dataset is likely to result in more meaningful and useful embeddings that make the downstream task of project type classification easier.

At a high level, the overall training pipeline for this project involved the following parts (shown in Figure 4-2):

- **Unsupervised training**: training a large unsupervised dataset of 500,000 Scratch projects to extract high-quality "word" embeddings.

- **Supervised training**: using the aforementioned pre-trained embeddings as a representation mechanism for "words", training a supervised classifier model on the labeled dataset of Scratch projects.

- **Evaluation**: evaluating the resulting classifier model by calculating the F1 score on the test dataset.

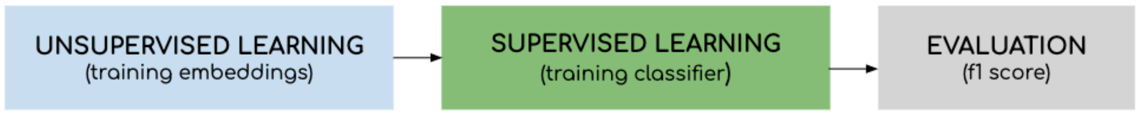


Figure 4-2: Diagram showing the training pipeline for this work.

4.2.1 Methodology

I used the fastText library to train word embeddings on the large unsupervised corpus of Scratch projects. The fastText library is meant for "efficient *text* classification and representation learning" [2], so a preliminary step before training word embeddings on our dataset is to transform Scratch projects into textual form.

Textifying Scratch Projects

To transform a Scratch project into a textual document, I encoded each block via its opcode and designated special symbols for specific "transitions" between blocks. For instance, the first block from Figure 4-3 would be encoded by its opcode `event_whenflagclicked`, and the transition from this block to the next block would be encoded by the `_NEXT_` symbol.

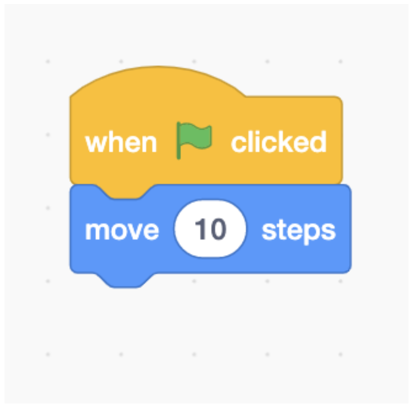


Figure 4-3: A stack of blocks consisting of the `event_whenflagclicked` and `motion_movesteps` blocks.

As discussed in the Introduction, blocks can be arranged in a variety of ways in Scratch, as shown by Figure 1-2. Blocks can be arranged into different "stacks", which are contiguous sequences of blocks. Blocks can also "nest" other blocks, such as in the case of `control_repeat` and `control_if` blocks. Blocks can also be inputs to other blocks, such as in the case of the `operator_join` block being an input to the `looks_say` block.

See Appendix A.1 for a mapping of transitions to their encodings, as well as a list of all blocks used in the corpus of 500,000 projects.

Rather than have the model infer the different transitions between blocks, I chose to explicitly encode some of the common transitions in an effort to make the learning process easier for the unsupervised model. To that end, I explicitly encoded the following transitions (shown in Table 4.1):

Table 4.1: Symbols to Transitions

`_STARTSTACK_`	beginning of a new stack
`_ENDSTACK_`	end of a stack
`_STARTNEST_`	beginning of nesting
`_ENDNEST_`	end of nesting
`_STARTINPUT_`	beginning of input
`_ENDINPUT_`	end of input
`_NEXT_`	next

Moreover, in order to ensure that there were enough regularities in the dataset for the model to learn them effectively, I opted to remove verbose artifacts – such as menu options or variable names – and replace them with generic symbols denoting their presence. Keeping the original verbosity level of these artifacts would have resulted in an enormously large vocabulary and the model will likely not have learned the patterns effectively.

Table 4.2 shows examples of such artifacts:

Table 4.2: Symbols to Artifacts

`numtext_input`	numeric or textual input
`_VAR_`	variable
`_LIST_`	list
`menu_option`	a chosen menu option
`_MENU_`	dropdown menu
`_NUMTEXTARG_`	numeric or textual argument*
`_BOOLARG_`	boolean argument*
`procedures_definition`	custom procedure definition**
`procedures_call`	custom procedure call

**These symbols are only used with* `procedures_definition` *block.*

**`procedures_definition` *is analogous to a software function.*

Once the encoding of blocks and transitions were determined, I then wrote a Javascript command-line program that executed the following steps:

- Converts the project into Scratch 3.0 version
- Traverses each stack of blocks and encodes the blocks and transitions accordingly

This script was implemented recursively, in order to ensure that arbitrary levels of nesting and input arrangements are correctly encoded. Accounting for all cases of block arrangements and handling the different block encodings in the Scratch back-end, as well as writing and debugging this script was a time-consuming process and constituted a significant portion of this thesis.

Training the Embeddings

Once I had transformed the projects into textual form, I was able to utilize the fastText library to train word embeddings on the dataset of Scratch projects.

The fastText library offers both a Python-based and a command-line based tool, with the latter containing more offerings and wider functionality. As a result, I utilized the fastText command line based tool during both the unsupervised and supervised training processes.

The fastText API offers options for training either CBOW or Skip-gram unsupervised models. As discussed in Chapter 3 (Related Work), the CBOW (Continuous Bag-of-Words) model learns to predict the most likely *current* word, $w(t)$, given its surrounding context (the words before and after it in the sequence). The Skip-gram model learns to predict the most likely *surrounding* words given the current word, $w(t)$.

I created a command-line based Python script called `fasttext_helper.py` that interfaced with the fastText API, and streamline the unsupervised and supervised training processes. This script utilized the `subprocess` [12] API to spawn shell commands, and had the following structure:

- The script takes in a user-created `settings` file that contains the arguments (i.e. hyperparameters) to the fastText unsupervised / supervised training command.
- The script parses that `settings` file and extracts the hyperparameters from it. See Section 4.4 (Hyperparameter Tuning) below for more details on the different hyperparameters.
- The script constructs the fastText unsupervised / supervised training command with those hyperparameters as arguments.

- The script spawns a shell command containing that fastText unsupervised / supervised training command, which kicks off the training process.

In terms of the choice of which type of unsupervised model is best suited for our purposes, CBOW or Skip-gram, it has been reported that CBOW tends to work slightly better for frequent words in the corpus, and takes less time to train [14], [13]. Skip-gram, on the other hand, tends to have higher accuracy for rare words, such as misspelled words, and takes longer to train. Given that our vocabulary is quite constrained, consisting of predefined blocks and transition symbols with few out-of-vocabulary words, training an unsupervised CBOW model seems to be the better choice.

In Scratch, there are two types of blocks: core and extension blocks. Core blocks represent core functionality that would be essential for most Scratch programs, including categories such as Motion, Looks, and Control, and are present on the Scratch editor by default. Extension blocks, on the other hand, represent blocks that offer extra specialized functionality, including categories such as Pen, Music, Video Sensing, and must be manually included into the Scratch editor by the user. Naturally, core blocks tend to be used more often in Scratch projects, and extension blocks less so. In a sense, extension blocks could end up being considered "out-of-vocabulary" words in our corpus if they are not seen during training. In an effort to avoid this situation, I increased the size of the unsupervised dataset in an attempt to cover a large amount of blocks. The larger our unsupervised dataset is, the better the chance we have at including as many blocks as possible in the training corpus.

Initial experiments comparing the accuracy of CBOW and Skip-gram models trained on our unsupervised corpus of Scratch projects demonstrated that CBOW works better. Due to time constraints, I thus opted to focus on training CBOW models during the hyperparameter tuning process (refer to Section 4.4 (Hyperparameter Tuning) below for more details).

4.3 Classifying Scratch Projects

Once I had trained word embeddings on the large *unsupervised* dataset of Scratch projects, I was then able to train a supervised classifier model on our smaller *labeled* dataset of Scratch projects, using these trained word embeddings as the representation mechanism for the supervised dataset.

4.3.1 Methodology

I utilized fastText's text classification API to train a supervised model on this dataset. I used the `fasttext_helper.py` Python script described earlier (see Training the Embeddings in Section 4.2.1 above) to interface with the fastText command-line tool for this supervised training portion.

Dataset Pre-processing

To begin, I executed some dataset pre-processing on the labeled dataset, which consisted of the following steps:

1. Removing any duplicate entries from the dataset.
2. Assigning the final labels.
3. Lumping the Slideshow projects into the Other category.
4. Textifying the projects using the textification script described earlier in Textifying Scratch Projects in Section 4.2.1.
5. Formatting the dataset to include the textified projects along with their labels in the format that fastText requires (i.e. a text file containing all projects in the dataset, where each line contains the project label first, encoded as `__label__game`, for example, and the textified project second).
6. Shuffling and splitting the dataset into training and test sets. The training set comprised roughly 80% of the full dataset and the test set roughly 20%.

Step 2 of assigning the final labels using a confidence threshold filtering mechanism and consisted of the following methodology:

- First, a threshold value for the confidence level is determined.
- For all project labels with a confidence level *above* this threshold, the label is kept as is.
- For all project labels with a confidence level *below* this threshold, the label is changed to "Other", and the confidence level is set to 1.

I chose a threshold value of 0.7.

I implemented step 3 of lumping the Slideshow projects into the Other category, as there ended up being only 46 Slideshows in the labeled dataset (and only 30 after filtering by the threshold confidence level). Such a low amount was likely not enough for training an accurate classifier model.

Training the Supervised Classifier

Once I pre-processed the labeled dataset, I commenced supervised training on it using the `fasttext_helper.py` Python script.

4.4 Hyperparameter Tuning

The end goal of the entire machine learning pipeline described above is to classify Scratch projects by type as accurately as possible. To that end, I tuned the hyperparameters of both the unsupervised and supervised training processes to optimize for the downstream performance of the classifier model, as measured by its F1 score on the (supervised) test dataset.

The hyperparameters tuned in the *unsupervised* training included the following [4]:

- type of unsupervised model to train, either `skipgram` or `cbow`
- dimension of the trained embeddings
- minimum number of word occurrences
- minimum length of the character n-grams
- maximum length of the character n-grams
- number of epochs for unsupervised training
- learning rate for unsupervised training

I explain each hyperparameter below and the reasoning behind our choice of its value range.

For the type of unsupervised model to train, I decided to only focus on training a CBOW model, since, as mentioned previously, this model type was better suited for our problem and demonstrated better accuracy on a few initial experiments.

The dimension of the trained embeddings refers to the desired length of the resulting vector representing each "word" in our corpus. The range of dimensions I considered was: `[50, 64, 128, 175, 200]`. I started with a dimension of 50 as a relatively "small" embedding size, ended with a dimension of 200 as a relatively "large" embedding size, and randomly chose a few values in between that were relatively evenly spaced out. The intuition was to experiment with a few values that each differed from the previous one in a non-trivial way.

The minimum number of word occurrences refers to the minimum number of times a word must appear in the corpus before an embedding is trained for it. The range we chose for this hyperparameter was: `[1, 5]`. We chose these values based on fastText default values. 1 and 5 seemed like reasonable values to experiment with, where a value of 1 indicates that each word in the corpus will have an embedding trained for it, and a value of 5 indicates that only words that appeared five times will have corresponding embeddings. Ultimately, the two situations we are experimenting with are the situation where all words, including infrequent words, receive a corresponding trained embedding, and the situation where only more frequent words receive a corresponding trained embedding.

The minimum and maximum length of character n-grams refers to the minimum and maximum length of substrings of the original word to use when representing that word. The resulting set of substrings (or "n-grams") will in fact include substrings of *all* lengths between the minimum and maximum length specified. For example, a minimum length of 1 and a maximum length of 5 implies that the following n-gram lengths are extracted from the original word: `[1, 2, 3, 4, 5]`. Once all the n-grams are extracted, they are collectively considered to represent the original word (along with the full word n-gram itself), and a vector representation is learned for each n-gram. The resulting word vector representation is a combination of its character n-gram vector representations. The range we chose for the minimum and maximum length of character n-grams is: `[(min: 1, max: 5), (min: 1, max: 8), (min: 1, max: 10)]`. We carefully chose these values based on the length of meaningful words in our corpus. Our corpus contains block opcodes – representing the "title" of a block – and transition symbols – signifying the presence of a special transition between one block and another. For instance, the following sequence might appear in our corpus:

```
_STARTSTACK_ looks_nextcostume _NEXT_ motion_gotoxy
_STARTINPUT_ numtext_input _ENDINPUT_ _STARTINPUT_ numtext_input
_ENDINPUT_
```

The size of the smallest *meaningful* substring in this sequence is 1. An example of such a substring is `"x"` in the `motion_gotoxy` opcode. `"x"` is a meaningful substring because it represents a coordinate of a physical location in the Scratch program. The size of the longest meaningful substring in this sequence is 5. An example of such a substring is `START` in the `_STARTSTACK_` transition symbol. Thus, an appropriate range for the minimum and maximum length of character n-grams would be: `(min: 1, max: 5)`. The length of the longest substring in our corpus is 10. An example of such a substring is `procedures`, found in `procedures_definition` and `procedures_call` opcodes. We generally wished for the model to always pick up on size-5 substrings, since the `START` substring is shared among a few transition symbols (e.g. `_STARTSTACK_ ; _STARTINPUT_ ; _STARTNEST_`) and is generally very meaningful for the model to learn. As a result, we picked a few values to experiment with for the *maximum* length that were roughly evenly spaced out – namely, `[1, 5, 8]`.

The number of epochs refers to the number of passes through the full training set during the training process. The learning rate represents the step size used to update the learned parameters during training (i.e. during the stochastic gradient descent procedure). The number of epochs and the learning rate hyperparameters are typically inversely related. Setting the number of epochs to a high number (e.g. 50) implies that the learning rate should be set to a low number (e.g. 0.01), and vice versa. The values we chose to experiment with for these hyperparameters were:

```
 (epoch:  5, lr:  0.1),
(epoch:  10, lr:  0.05),
(epoch:  25, lr:  0.01),
(epoch:  50, lr:  0.01))
```

The hyperparameters tuned in the *supervised* training included the following [4]:

- minimum number of word occurrences
- number of epochs for supervised training
- learning rate for supervised training
- size of *word* n-grams to consider
- pre-trained word vectors

Many of the hyperparameters in the supervised training are similar to those described in the unsupervised training section above. The only new hyperparameters in this case are the size of *word* n-grams to consider, and the pre-trained word vectors. The former refers to the size of word n-gram "window" around the current word to consider during training. The values we chose to experiment with for the size of word n-grams were: `[1, 5, 10]`. The fastText default value for this hyperparameter was 1. However, since our work involves categorizing Scratch projects, and the type of a Scratch project likely depends on the interaction between blocks and their neighboring blocks, it seemed befitting to experiment with a few values for the size of neighboring word n-grams to consider.

The pre-trained word vectors refer to the word embeddings trained in the unsupervised learning step, to be used as the initial representation mechanism for blocks and symbols in the subsequent supervised learning step.

For the supervised training case, we experimented with slightly different values for the number of epochs and learning rate, as compared to the unsupervised training case. The values we experimented with here were:

```
(epoch: 5, lr: 0.1),
(epoch: 10, lr: 0.1),
(epoch: 25, lr: 0.05),
(epoch: 50, lr: 0.05)
```

The fastText default value for the supervised learning rate is 0.1 (in contrast, the unsupervised learning rate default value is 0.05). We chose to experiment with this value of 0.1 and the smaller value of 0.05, while accordingly varying the number of epochs.

I implemented a grid search [23] to tune the hyperparameters for both the unsupervised and supervised training processes. I iterated through all combinations of the unsupervised and supervised hyperparameters, training models at each setting, and evaluated the performance of the supervised classifier model on the test set. I wrote Python scripts that implemented these steps.

During the process of unsupervised hyperparameter tuning, I elected to train all the unsupervised models on a smaller unsupervised dataset consisting of 10,000 projects, in lieu of the larger 500,000 project dataset. Training only one unsupervised model on the larger dataset takes a non-trivial amount of time, and repeatedly training many unsupervised models during the hyperparameter tuning process (to account for all the different hyperparameter combinations) would have taken prohibitively long. As a result, I opted to use the smaller 10,000 project dataset to get a ballpark estimate of the top 10 performing hyperparameter combinations. Then, I took these hyperparameter combinations and trained unsupervised and supervised models on the larger 500,000 project dataset to get more precise accuracy metrics and determine which set of hyperparameters resulted in the best accuracy. Moreover, to further speed up the unsupervised training process during hyperparameter tuning, I used the hierarchical softmax loss function in lieu of the (default) negative sampling loss function. Negative sampling loss tends to result in more accurate models for frequent words [14]. Again, once I found the best performing hyperparameter combinations, I trained a new unsupervised model using negative sampling loss function.

Although this workaround of using the smaller dataset for hyperparameter tuning is not guaranteed to optimize all hyperparameters, it is a reasonable compromise for gaining the resulting speedup. Moreover, given the constrained nature of our corpus in terms of vocabulary and syntax, this workaround can provide a reasonable estimate for the optimal hyperparameter settings.

Moreover, throughout the hyperparameter tuning process, I evaluated the accuracy of the resulting supervised model on the **test set** by calculating its F1 Score. In the end, I chose the hyperparameter configuration that yielded the highest score on the test set. The ideal scenario would be to use a **validation set** during the hyperparameter tuning process while keeping the test set untouched throughout the entire training phase, so as to avoid overfitting and implicitly optimizing for the test set. In the current configuration, we are effectively finding the optimal set of hyperparameters that would result in the best performance on the test set. This practice could lead to generalization issues, as it is difficult to confirm whether the model truly generalizes to unseen data samples (from the test set), if those samples were used during the training phase in one way or another. Unfortunately, the classification dataset is too small (873 projects) to split into training, validation, and test sets.

The best hyperparameter configuration I found was:

Unsupervised Hyperparameters:

- type of unsupervised model to train: `cbow`
- dimension of the trained embeddings: `64`
- minimum number of word occurrences: `5`
- minimum length of the character n-grams: `1`
- maximum length of the character n-grams: `5`
- number of epochs for unsupervised training: `5`
- learning rate for unsupervised training: `0.1`

Supervised Hyperparameters:

- minimum number of word occurrences: `5`
- number of epochs for supervised training: `50`
- learning rate for supervised training: `0.05`
- size of word n-grams to consider: `1`

Additional testing suggests that further tuning might result in improvements. In future work, additional hyperparameter tuning on the larger 500,000 dataset could be conducted to see the full range of improvements possible (albeit consuming more computing resources).

Chapter 5

Evaluation

5.1 Unsupervised Training Evaluation

I evaluate and analyze the unsupervised model in this section. In particular, I will examine both word and project embeddings.

In the sections below, I will refer to the concept of "similarity" often. Word similarity is a well-established and well-defined concept – it's usually straightforward to judge whether two words are similar in their semantics or not. This is not immediately the case for Scratch blocks, as it is often harder and more ambiguous to determine whether two Scratch blocks are "similar" or what the metrics for similarity should be in the first place. Two Scratch blocks can be similar in terms of functionality (though likely not *exactly the same* in that regard), or similar in terms of appearing in similar contexts or similar in terms of falling within the same category (e.g. `Motion` category or `Looks` category). Although there isn't currently a robust conception of how two blocks might be similar, I will attempt to explore the aforementioned notions when assessing similarity below.

5.1.1 Evaluating Word Embeddings

Nearest Neighbors

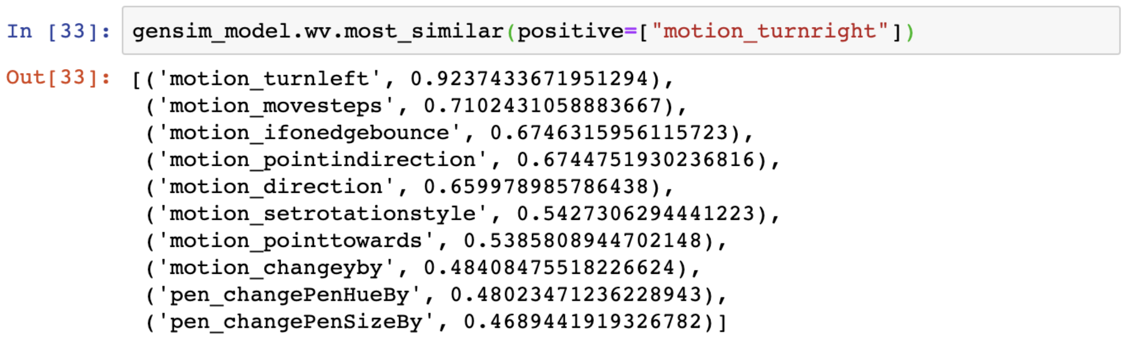
I examine the nearest neighbors of a few select blocks, based on their embeddings, and intuitively assess whether these are reasonably “similar” to the original block.

Block 1: `motion_turnright`

Figure 5-1 shows the nearest neighbors of the `motion_turnright` block. As shown, the closest block to it is the `motion_turnleft` block, which is expected since those two blocks share the same exact functionality, only differing in which direction to turn the character in the Scratch program. The similarity measure for this block is higher than the remaining blocks by a noticeable margin.

The remaining blocks that are deemed similar to the `motion_turnright` block are notable. A good portion of these blocks involve some sort of behavior that affects or involves the direction of the character in the program, including `motion_ifonedgebounce`, `motion_pointindirection`, `motion_direction`, `motion_setrotationstyle`, `motion_pointtowards`. It is remarkable that the model was able to learn this "direction" relation between these blocks, even though the word "direction" isn't present in all the opcodes nor do all the opcodes share the same word.

The second most similar block to the `motion_turnright` block is the `motion_movesteps` block. Although their functionalities are not the same, with the `motion_movesteps` block indicating that the character move a certain number of steps in the current direction, these two blocks are often used together.



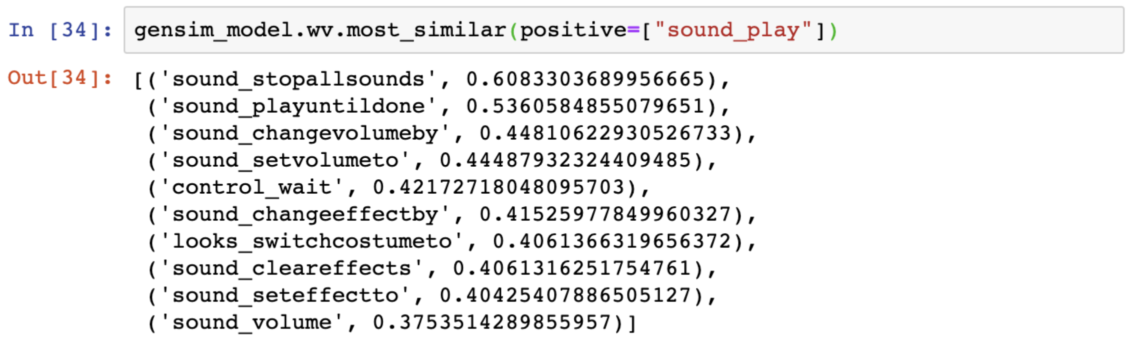
```
In [33]: gensim_model.wv.most_similar(positive=["motion_turnright"])

Out[33]: [('motion_turnleft', 0.9237433671951294),
          ('motion_movesteps', 0.7102431058883667),
          ('motion_ifonedgebounce', 0.6746315956115723),
          ('motion_pointindirection', 0.6744751930236816),
          ('motion_direction', 0.659978985786438),
          ('motion_setrotationstyle', 0.5427306294441223),
          ('motion_pointtowards', 0.5385808944702148),
          ('motion_changeyby', 0.48408475518226624),
          ('pen_changePenHueBy', 0.48023471236228943),
          ('pen_changePenSizeBy', 0.4689441919326782)]
```

Figure 5-1: Nearest neighbors of the `motion_turnright` block, ordered by the similarity level of their embeddings.

Block 2: `sound_play`

Figure 5-2 shows the nearest neighbors of the `sound_play` block. The model was able to correctly identify the `sound_playuntildone` block as one of closest blocks to the `sound_play` block. The model deems the `sound_stopallsounds` block as the most "similar" block to the `sound_play` block, although the similarity level of 0.608 is not significantly high. This is reasonable since the former block's functionality is essentially the opposite of the latter's – i.e. stopping sounds as opposed to playing sounds. The model was able to learn that these functionalities are significantly related. Lastly, most of the blocks in the nearest neighbors list are part of the Sound category of blocks, indicating that the model at least learned that these blocks share this feature in common. Of course, since the model takes into account sub-word information as well when representing a block, it picks up on the sub-word `“sound”` that is shared between these blocks.



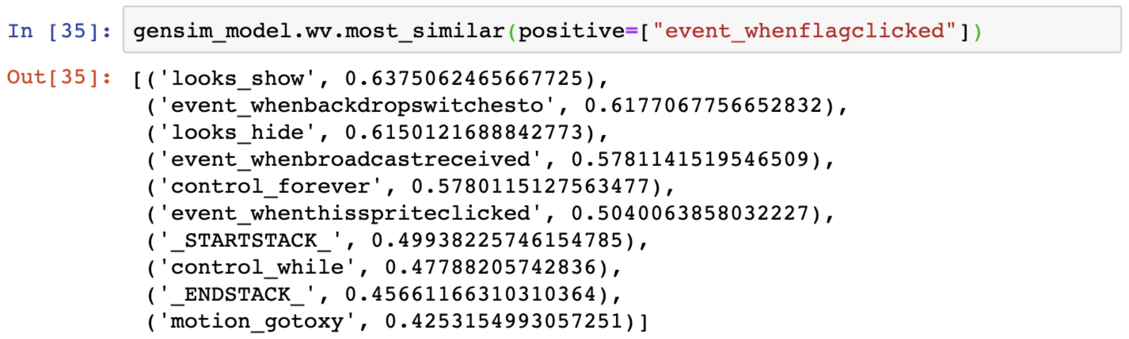
```
In [34]: gensim_model.wv.most_similar(positive=["sound_play"])

Out[34]: [('sound_stopallsounds', 0.6083303689956665),
          ('sound_playuntildone', 0.5360584855079651),
          ('sound_changevolumeby', 0.44810622930526733),
          ('sound_setvolumeto', 0.44487932324409485),
          ('control_wait', 0.42172718048095703),
          ('sound_changeeffectby', 0.41525977849960327),
          ('looks_switchcostumeto', 0.4061366319656372),
          ('sound_cleareffects', 0.4061316251754761),
          ('sound_seteffectto', 0.40425407886505127),
          ('sound_volume', 0.3753514289855957)]
```

Figure 5-2: Nearest neighbors of the `sound_play` block, ordered by the similarity level of their embeddings.

Block 3: event_whenflagclicked

Figure 5-3 shows the nearest neighbors of the event_whenflagclicked block. The model identified that the looks_show, looks_hide, and control_forever blocks are related to the event_whenflagclicked block, which is promising because those three blocks tend to appear right after the event_whenflagclicked block often. Moreover, the model learned that other "hat" (header) blocks, such as the event_whenbackdrop-switchesto, event_whenbroadcastreceived, and event_whenthisspriteclicked blocks, are similar to the event_whenflagclicked block. This is significant because those blocks can only be placed at the top of a stack, so they share a "structural" quality, and they are triggered when certain events occur (such as the backdrop changing, a broadcast being received, etc.), so they share a "functional" quality. Lastly, the model identified that the _STARTSTACK_ and _ENDSTACK_ symbols are similar to event_whenflagclicked block, which makes sense because those symbols demarcate the different stacks and the event_whenflagclicked block is often the starting block in a stack.

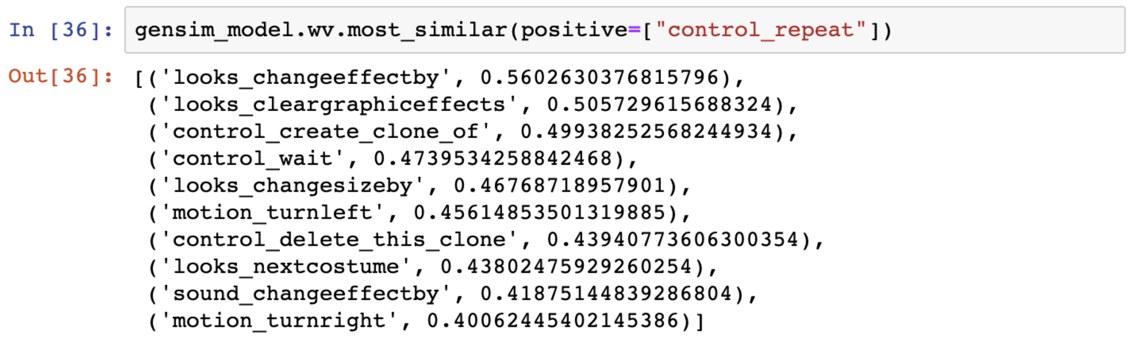

```
In [35]: gensim_model.wv.most_similar(positive=["event_whenflagclicked"])

Out[35]: [('looks_show', 0.6375062465667725),
          ('event_whenbackdropswitchesto', 0.6177067756652832),
          ('looks_hide', 0.6150121688842773),
          ('event_whenbroadcastreceived', 0.5781141519546509),
          ('control_forever', 0.5780115127563477),
          ('event_whenthisspriteclicked', 0.5040063858032227),
          ('_STARTSTACK_', 0.49938225746154785),
          ('control_while', 0.47788205742836),
          ('_ENDSTACK_', 0.45661166310310364),
          ('motion_gotoxy', 0.4253154993057251)]
```

Figure 5-3: Nearest neighbors of the event_whenflagclicked block, ordered by the similarity level of their embeddings.

Block 4: **`control_repeat`**

Figure 5-4 shows the nearest neighbors of the `control_repeat` block. In this case, many of the blocks deemed similar by the model are not immediately intuitive. For instance, one would expect that the `control_repeat_until` and `control_forever` be listed in the nearest neighbors, since these blocks share similar looping functionality with the `control_repeat` block. However, the model doesn't seem to have picked up on that. However, many of the blocks listed as nearest neighbors are still reasonable, since they are blocks that are typically nested inside the `control_repeat` block, such as the `looks_changeeffectby`, `control_wait`, `motion_turnleft`, `sound_change-effectby`, and `looks_nextcostume` blocks, among others. Lastly, since the model utilizes sub-word information while training, it expectedly lists some of the Control category blocks in the nearest neighbors list – namely, the `control_create_clone_of`, `control_wait`, and `control_delete_this_clone` blocks.



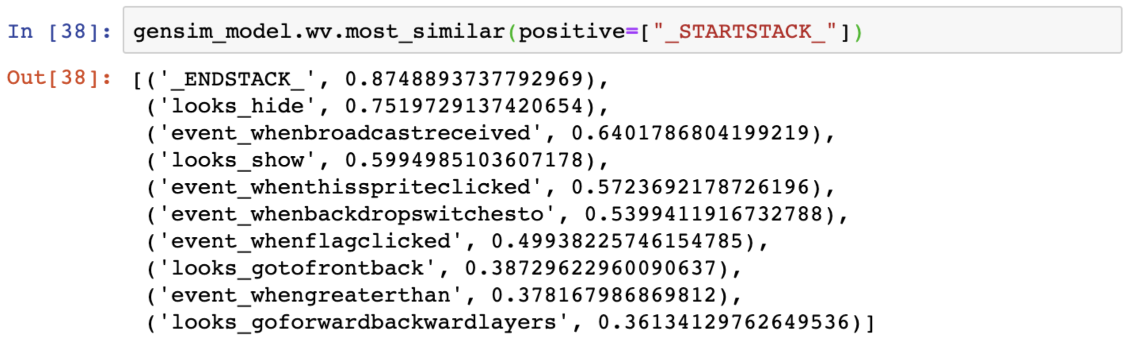
```
In [36]: gensim_model.wv.most_similar(positive=["control_repeat"])

Out[36]: [('looks_changeeffectby', 0.5602630376815796),
          ('looks_cleargraphiceffects', 0.505729615688324),
          ('control_create_clone_of', 0.49938252568244934),
          ('control_wait', 0.4739534258842468),
          ('looks_changesizeby', 0.46768718957901),
          ('motion_turnleft', 0.45614853501319885),
          ('control_delete_this_clone', 0.43940773606300354),
          ('looks_nextcostume', 0.43802475929260254),
          ('sound_changeeffectby', 0.41875144839286804),
          ('motion_turnright', 0.40062445402145386)]
```

Figure 5-4: Nearest neighbors of the `control_repeat` block, ordered by the similarity level of their embeddings.

Block 5: `_STARTSTACK_`

Figure 5-5 shows the nearest neighbors of the `_STARTSTACK_` symbol. Expectedly, the model picks up on the similarity between the `_STARTSTACK_` and `_ENDSTACK_` symbols, which both share the functionality of demarcating stacks, and share the sub-word `STACK`. The model assigns a high similarity measure of 0.875 corresponding to these two blocks. Quite a few of the other blocks in the nearest neighbors list are `Event` category "hat" (header) blocks. This is expected since those types of blocks can only come at the start of a stack, and most stacks begin with them (so they often appear right after the `_STARTSTACK_` symbol).



```
In [38]: gensim_model.wv.most_similar(positive=["_STARTSTACK_"])

Out[38]: [('_ENDSTACK_', 0.8748893737792969),
          ('looks_hide', 0.7519729137420654),
          ('event_whenbroadcastreceived', 0.6401786804199219),
          ('looks_show', 0.5994985103607178),
          ('event_whenthisspriteclicked', 0.5723692178726196),
          ('event_whenbackdropswitchesto', 0.5399411916732788),
          ('event_whenflagclicked', 0.49938225746154785),
          ('looks_gotofrontback', 0.38729622960090637),
          ('event_whengreaterthan', 0.378167986869812),
          ('looks_goforwardbackwardlayers', 0.36134129762649536)]
```

Figure 5-5: Nearest neighbors of the `_STARTSTACK_` block, ordered by the similarity level of their embeddings.

Cosine Similarity Between Block Pairs

I examine the cosine similarity between pairs of “similar” blocks and pairs of “different” blocks. The cosine similarity is a similarity measure between two vectors that evaluates whether they point in approximately the same direction [25]. Cosine similarity is often used as a similarity measure for word vectors. The cosine similarity measure is:

$$\cos(a, b) = \frac{(a \cdot b)}{||a|| * ||b||}$$

The higher the cosine similarity measure, the higher the similarity between the two vectors, with a value of 1 indicating perfect similarity [25].

Similar Blocks

Pair 1: `motion_turnright` & `motion_turnleft`

```
Cosine similarity:  0.9237434
```

These two blocks have a high cosine similarity (close to 1), as expected. Since they share a few sub-words (`motion`, `turn`), and, as a result, share very similar functionality, it is a good sign that the model picked up on this similarity.

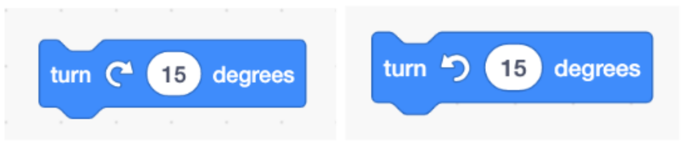


Figure 5-6: The `motion_turnright` (left) and `motion_turnleft` (right) blocks.

Pair 2: `control_repeat & control_forever`

`Cosine similarity``: -0.04653213`

These two blocks have a low similarity score. This is surprising since these two blocks share similar looping functionalities and share the sub-word `"control"`, but the model doesn't learn this correlation. This could perhaps be due to the fact that although these blocks share the looping functionality, that isn't reflected in the opcode, in that there are no shared sub-words that would indicate this similarity in functionality. Another reason for this low similarity might be that these blocks aren't typically used interchangeably in similar contexts within Scratch.

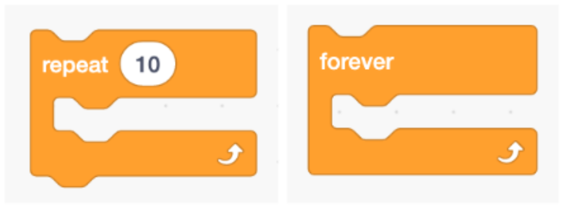


Figure 5-7: The `control_repeat` (left) and `control_forever` (right) blocks.

Pair 3: `operator_gt & operator_lt`

`Cosine similarity``: 0.9350381`

As expected, these two blocks have a high similarity score (close to 1). Given that they share practically the same opcode (with the exception of a single letter), the model learns that these two blocks should share similar representations. Moreover, it is likely that they are often used in very similar contexts, giving rise to a high similarity measure between their embeddings.

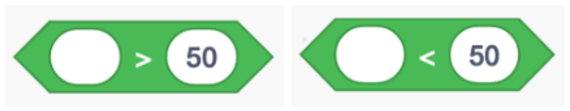


Figure 5-8: The `operator_gt` (left) and `operator_lt` (right) blocks.

Pair 4: `data_setvariableto` & `data_changevariableby`

`Cosine similarity``: 0.5607882`

These two blocks share a relatively low similarity measure. Although they share some sub-words (`data`, `variable`), their somewhat shared functionality of modifying variables is not reflected in the opcode, seemingly the reason why the model doesn't pick up on this similarity. Although these blocks can conceivably be used interchangeably in similar contexts, perhaps this interchangeability doesn't occur enough throughout the training corpus for the model to pick up on.

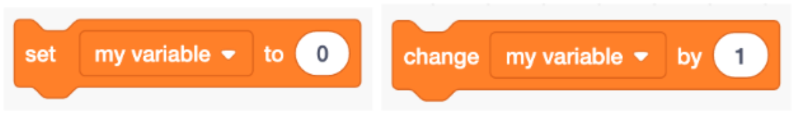


Figure 5-9: The `data_setvariableto` (left) and `data_changevariableby` (right) blocks.

Pair 5: `control_if` & `control_if_else`

`Cosine similarity``: 0.5709473`

These two blocks have a relatively low similarity measure. Although they share a few significant sub-words (`control`, `if`), this is not enough for the model to learn the similarity in their conditional functionality.

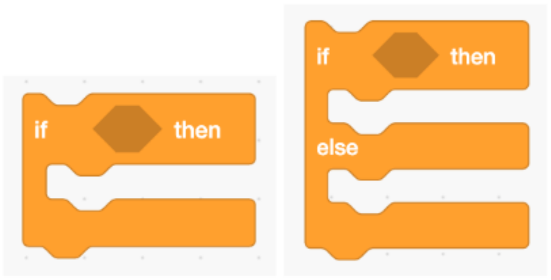


Figure 5-10: The `control_if` (left) and `control_if_else` (right) blocks.

Pair 6: `motion_turnright` & `motion_ifonedgebounce`

`Cosine similarity:` `0.67463166`

These two blocks share a relatively medium similarity score, given that they share similar "rotation" functionalities, but don't share many sub-words to indicate that. The model seems to have picked up on this similarity, although it is hard to determine this conclusively.

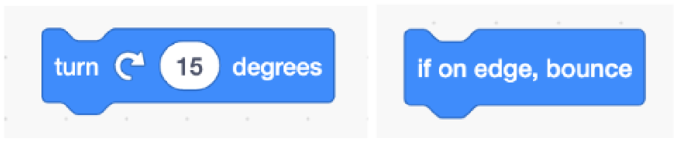


Figure 5-11: The `motion_turnright` (left) and `motion_ifonedgebounce` (right) blocks.

Pair 7: `event_broadcast` & `event_whenbroadcastreceived`

`Cosine similarity:` `0.637749`

These two blocks are "complementary" blocks, where the `event_broadcast` block broadcasts a message, and the `event_whenbroadcastreceived` is triggered when that message is received. Thus, since they share complementary functionalities, we expect the model to learn this, to some extent. This is a hard relationship to decode because these blocks don't tend to appear consecutively, but rather, are typically found in separate stacks. This distance between them would make this relationship difficult to pick up. Consequently, we consider the similarity score (0.638) of these two blocks to be relatively high.

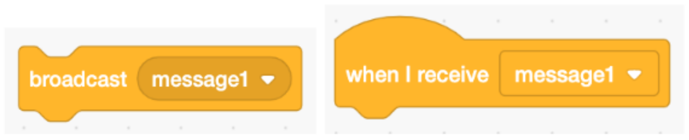


Figure 5-12: The `event_broadcast` (left) and `event_whenbroadcastreceived` (right) blocks.

Pair 8: `_STARTSTACK_` & `_ENDSTACK_`

```
Cosine similarity:  0.8748894
```

As a sanity check, our model should have learned that these two symbols are similar, given that they share the `STACK` sub-word, and demarcate the beginning and end of each stack, so they occur in each other's contexts often. We see that the model has learned this similarity, assigning the high score of 0.875.

As we can see from the previous examples, the use of sub-word information for finding optimal representations of blocks and symbols is very useful, since these sub-words tend to reflect the similarity in functionality for blocks and symbols.

Different Blocks

As a further check, we ensure that our model assigns low similarity scores for blocks that are considered to be "different".

Pair 1: `control_create_clone_of` & `sensing_askandwait`

`Cosine similarity:` `-0.3848171`

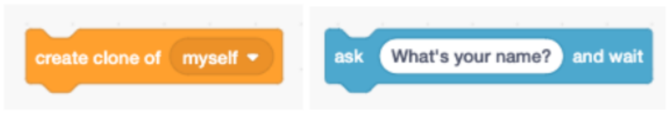


Figure 5-13: The `control_create_clone_of` (left) and `sensing_askandwait` (right) blocks.

Pair 2: `operator_join` & `motion_setrotationstyle`

`Cosine similarity:` `-0.20272619`

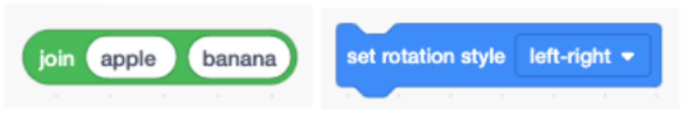


Figure 5-14: The `operator_join` (left) and `motion_setrotationstyle` (right) blocks.

Pair 3: `operator_add` & `looks_nextcostume`

`Cosine similarity:` `-0.3173863`

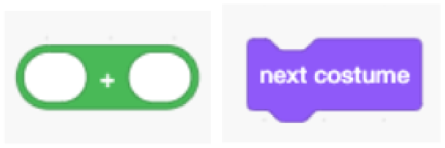


Figure 5-15: The `operator_add` (left) and `looks_nextcostume` (right) blocks.

For all of the block pairs above, each pair share no meaningful sub-words and no functionality. Moreover, the last pairs cannot be effectively used interchangeably in the same locations within a Scratch program, since their shapes are different and don't allow for that. As expected, the model deems all of these pairs as "different", giving them low (and negative) similarity scores.

Comparing Similarities of Embeddings

I create a heatmap matrix to visualize the (cosine) similarities between a few select blocks, shown in Figure 5-16. Darker colors correspond to a high similarity between the corresponding blocks in that row and column. The diagonal of this matrix is the darkest since it represents the similarity of each block with itself. Note that the scale of cosine similarities goes down to below -0.3.

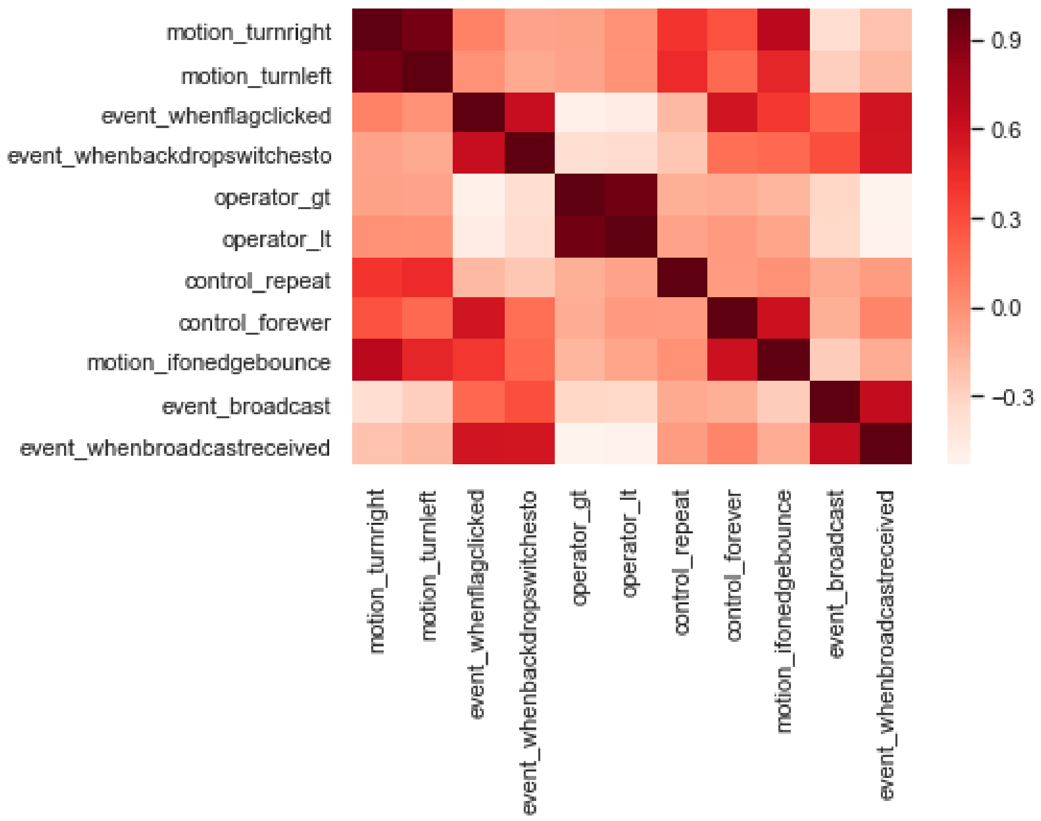


Figure 5-16: Heatmap matrix showing the cosine similarities between select blocks. The shade of each square in the matrix represents the cosine similarity of the Scratch block in that row and the Scratch block in that column. Darker shades indicate higher similarity.

t-SNE Visualization

Closest Words Visualization

I create t-SNE plots [29] to visualize the closest set of blocks and symbols to a specific block. t-SNE is an efficient dimensionality reduction technique that allows us to simplify the data embeddings to fewer dimensions. In this case, I collapse the data to 2 dimensions. I choose to focus on the `motion_turnright`, `procedures_definition`, and `data_setvariabletolook`, and look at their 20 closest words. I use these t-SNE plots both as an explorative way to uncover some of the relationships between different blocks, and as a way to confirm relationships we expect to exist.

Block 1: `motion_turnright`

Figure 5-17 shows the t-SNE plot for the `motion_turnright` block. Many of the blocks in the set of closest words are `motion` category blocks, confirming our expectation that blocks within the same category have closer embeddings. Moreover, many of these `motion` category blocks involve direction-related functionality, such as `motion_pointtowards`, `motion_pointindirection`, `motion_direction`, `motion_setrotationstyle`, and `motion_ifonedgebounce`. Moreover, it is interesting to see that a few `pen` category blocks are included in the set of closest words.

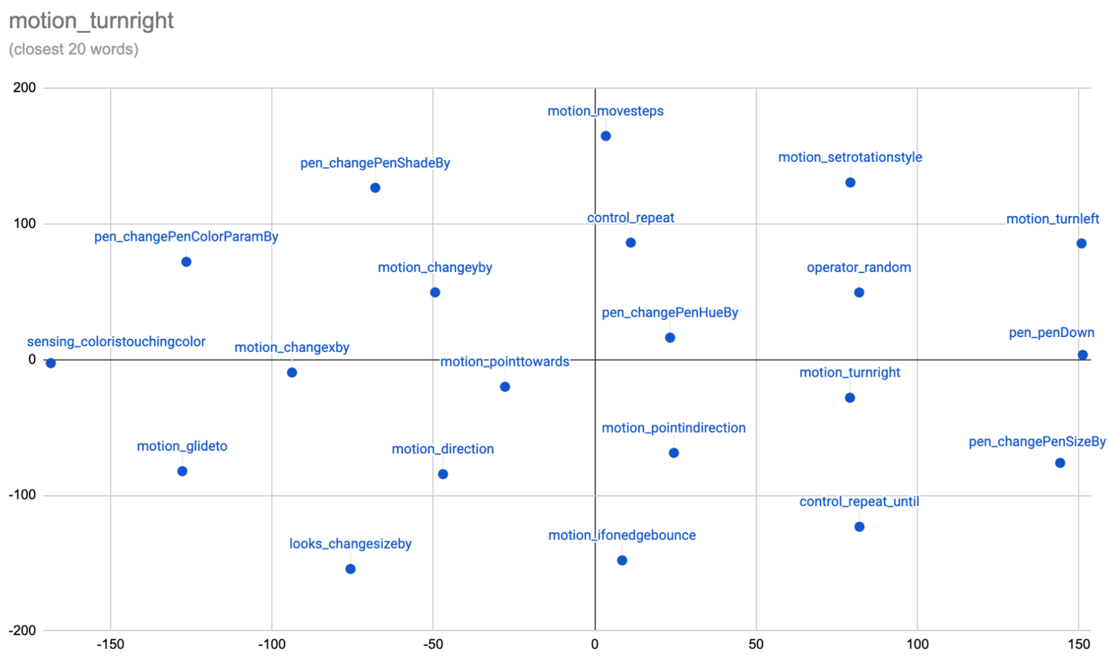


Figure 5-17: t-SNE plot showing the closest 20 words to the `motion_turnright` block.

Block 2: `procedures_definition`

Figure 5-18 shows the t-SNE plot for the `procedures_definition` block. This block corresponds to defining a new custom procedure in the program, much like a function in software code. Unlike other block categories, the `procedures` category only contains two blocks in total, the `procedures_definition` and `procedures_call` blocks, meaning won't be multiple blocks from the same category in the t-SNE plot. There are quite a few `data` category blocks present in the set of closest words, particularly list-type blocks, such as `data_insertatlist`, `data_replaceitemoflist`, `data_itemnumoflist`, and more, implying that definitions of new procedures tend to include list manipulations often. Moreover, the `_NUMTEXTARG_` and `_BOOLARG_` are symbols that denote the presence of arguments for the custom procedure, of either numerical, textual, or boolean types. These symbols only appear in stacks beginning with the `procedures_definition` block, so, as expected, they are present in its t-SNE plot as well.

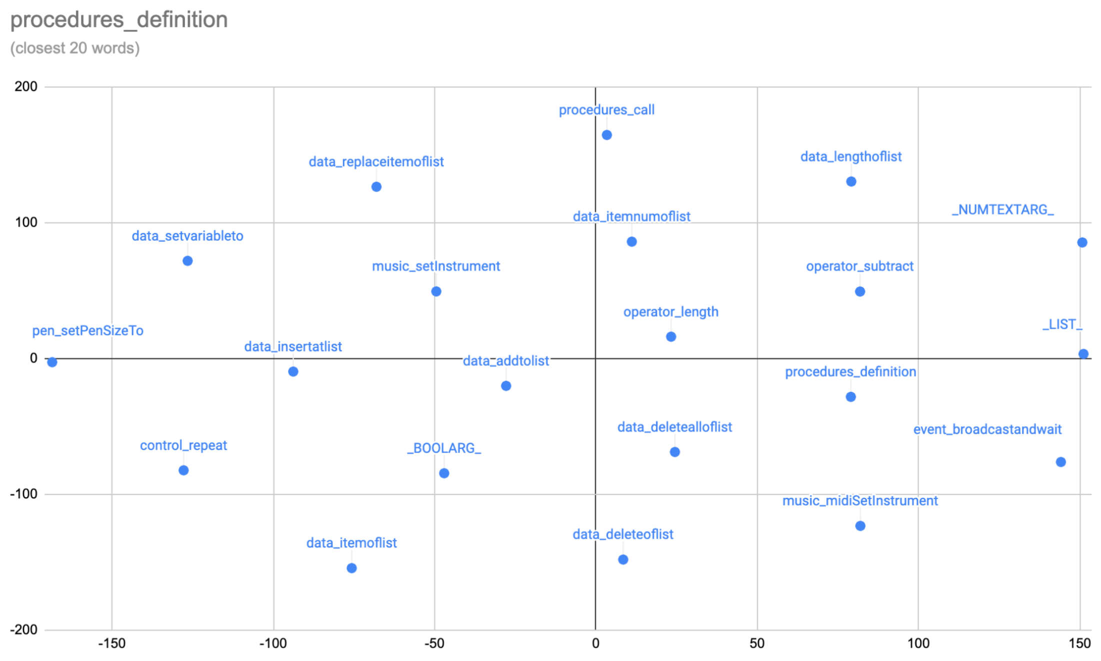


Figure 5-18: t-SNE plot showing the closest 20 words to the `procedures_definition` block.

Block 3: **`data_setvariableto`**

Figure 5-19 shows the t-SNE plot for the `data_setvariableto` block. As expected, many of the `data` category blocks are present in the set of closest words. Moreover, many `operator` category blocks are also present, such as `operator_add`, `operator_multiply`, `operator_round`, and `operator_join` blocks, suggesting that variable manipulations tend to involve mathematical and lexical operations. Lastly, the `_VAR_` symbol, used to denote the use of a variable, is also present in the plot, since the `data_setvariableto` block employs and manipulates variables.

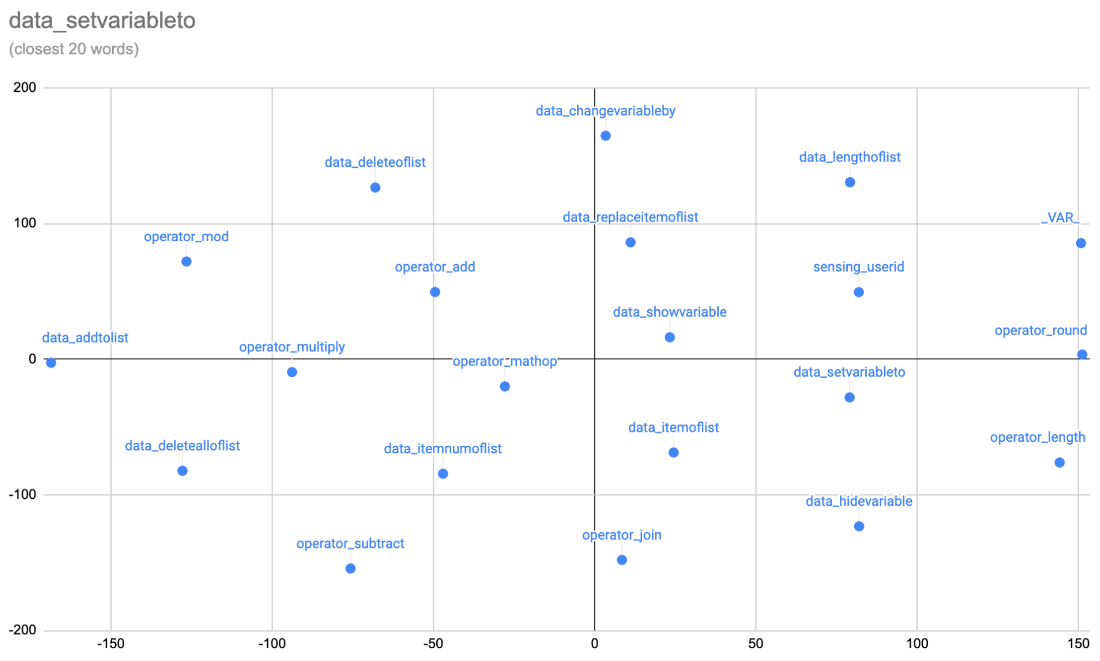


Figure 5-19: t-SNE plot showing the closest 20 words to the `data_setvariableto` block.

Clusters of Embeddings

I create a t-SNE plot with 2 dimensions containing all the blocks in our corpus, and include it in the Appendix (Section B.1). As expected, blocks of the same category tend to cluster close together, such as the `data` category blocks on the top left of the plot, the `pen` category blocks on the bottom left of the plot, and the `sound` category blocks on the top right of the plot. Moreover, it is interesting to note which category clusters are found near each other. A few of the hardware extension categories, such as `LEGO MINDSTORMS EV3`, `LEGO Education WeDo`, `Micro:bit`, `LEGO BOOST`, and `Vernier Go Direct`, are found near each other towards the middle left of the plot. In addition, the points corresponding to pairs of blocks that have "sister" functionalities are found near each other on the plot, such as `ev3_motorTurnClockwise` and `ev3_motorTurnCounterClockwise` (middle of the plot), `motion_setx` and `motion_sety` (top middle of the plot), and `data_showlist` and `data_hidelist` (top left of the plot).

Please refer to the Appendix (Section B.1) for this t-SNE plot containing all blocks in our corpus.

Heatmap Visualizations of Embeddings

Visualizing Word Embedding Similarities

I create and compare heatmaps of various block embeddings to visualize the similarity in their values at each dimension. I look at triplets of blocks at a time, choosing two blocks that are similar (in terms of cosine similarity), and a third one that is different from the pair. It is important to note that two embeddings do not need to have the similar values at each dimension in order to have a high cosine similarity. This evaluation method is merely included to offer a different way of visualizing similarity, if any exists.

Triplet 1:

Similar:

- `motion_turnright`
- `motion_turnleft`

Different:

- `looks_changesizeby`

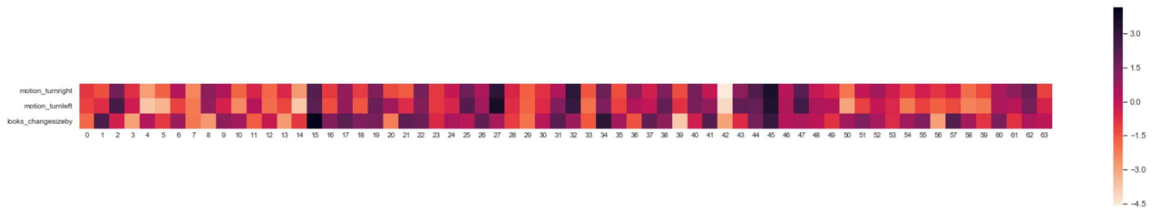


Figure 5-20: Heatmap visualization of the `motion_turnright`, `motion_turnleft`, and `looks_changesizeby` embeddings.

The `motion_turnright` and `motion_turnleft` embeddings have a higher cosine similarity measure (0.924) than either the `motion_turnright` and `looks_change-sizeby` (0.411), or the `motion_turnleft` and `looks_changesizeby` (0.335) embeddings.

Figure 5-20 shows the heatmap visualization for these blocks. The `motion_turn-right` and `motion_turnleft` embeddings share similar values at quite a few dimensions (in terms of how small / large the values are, and subsequently, the shading level of each square) as compared to the `looks_changesizeby` embedding. Although this isn't an exact science, it is meant to give a rough intuition for whether the embeddings for similar blocks line up at different dimensions.

Triplet 2:

Similar:

- `operator_gt`
- `operator_lt`

Different:

- `event_whenkeypressed`

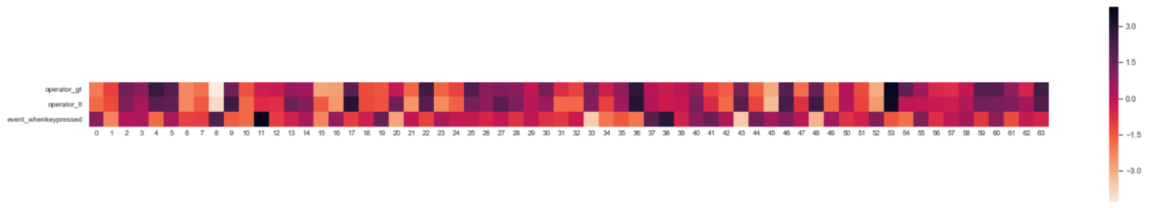


Figure 5-21: Heatmap visualization of the `operator_gt`, `operator_lt`, and `event_whenkeypressed` embeddings.

The `operator_gt` and `operator_lt` embeddings have a higher cosine similarity measure (0.935) than either the `operator_gt` and `event_whenkeypressed` (-0.321), or the `event_whenkeypressed` and `operator_lt` (-0.344) embeddings.

Figure 5-21 shows the heatmap visualization for these blocks. The `operator_gt` and `operator_lt` embeddings share similar values at more than half the dimensions as compared to the `event_whenkeypressed` embedding, and visually seems to line up overall.

Triplet 3:

Similar:

- `looks_seteffectto`
- `looks_changeeffectby`

Different:

- `sound_seteffectto`

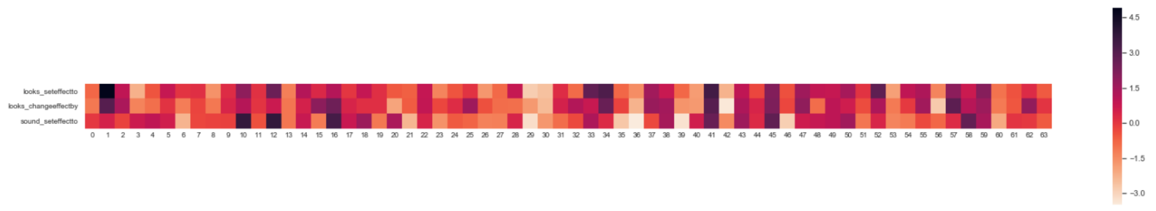


Figure 5-22: Heatmap visualization of the `looks_seteffectto`, `looks_changeeffectby`, and `sound_seteffectto` embeddings.

The `looks_seteffectto` and `looks_changeeffectby` embeddings have a higher cosine similarity measure (0.810) than either the `looks_seteffectto` and `sound_seteffectto` (0.594), or the `sound_seteffectto` and `looks_changeeffectby` (0.519) embeddings. As an aside, although the `looks_seteffectto` and `sound_seteffectto` opcodes share more sub-words (`set`, `effect`, `to`) than `looks_seteffectto` and `looks_changeeffectby` opcodes (`looks`, `effect`), the first pair have a much higher cosine similarity score. This may be due to the fact that the functionalities of the `looks_seteffectto` and `looks_changeeffectby` blocks are very similar and they can consequently be used interchangeably in many of the same contexts.

Figure 5-22 shows the heatmap visualization for these blocks. Although the `looks_seteffectto` and `looks_changeeffectby` have a high similarity score, their embeddings don't seem to significantly line up more than with the `event_whenkeypressed` embedding.

5.1.2 Evaluating Project Embeddings

Project embeddings can be derived from block embeddings by using the fastText API, by computing a normalized average on the block embeddings (in the same way that a sentence embedding is derived from word embeddings) [8].

I conduct a few evaluation experiments on these project embeddings.

Nearest Neighbors

I randomly choose a few projects from our training corpus and examine their nearest neighbors within the same corpus. I qualitatively evaluate whether these projects display any similarities to the original project. Since the block and project embeddings only take into account the code of the program, and not the title or description, I only focus on perceived similarities between the structure of the code between these projects.

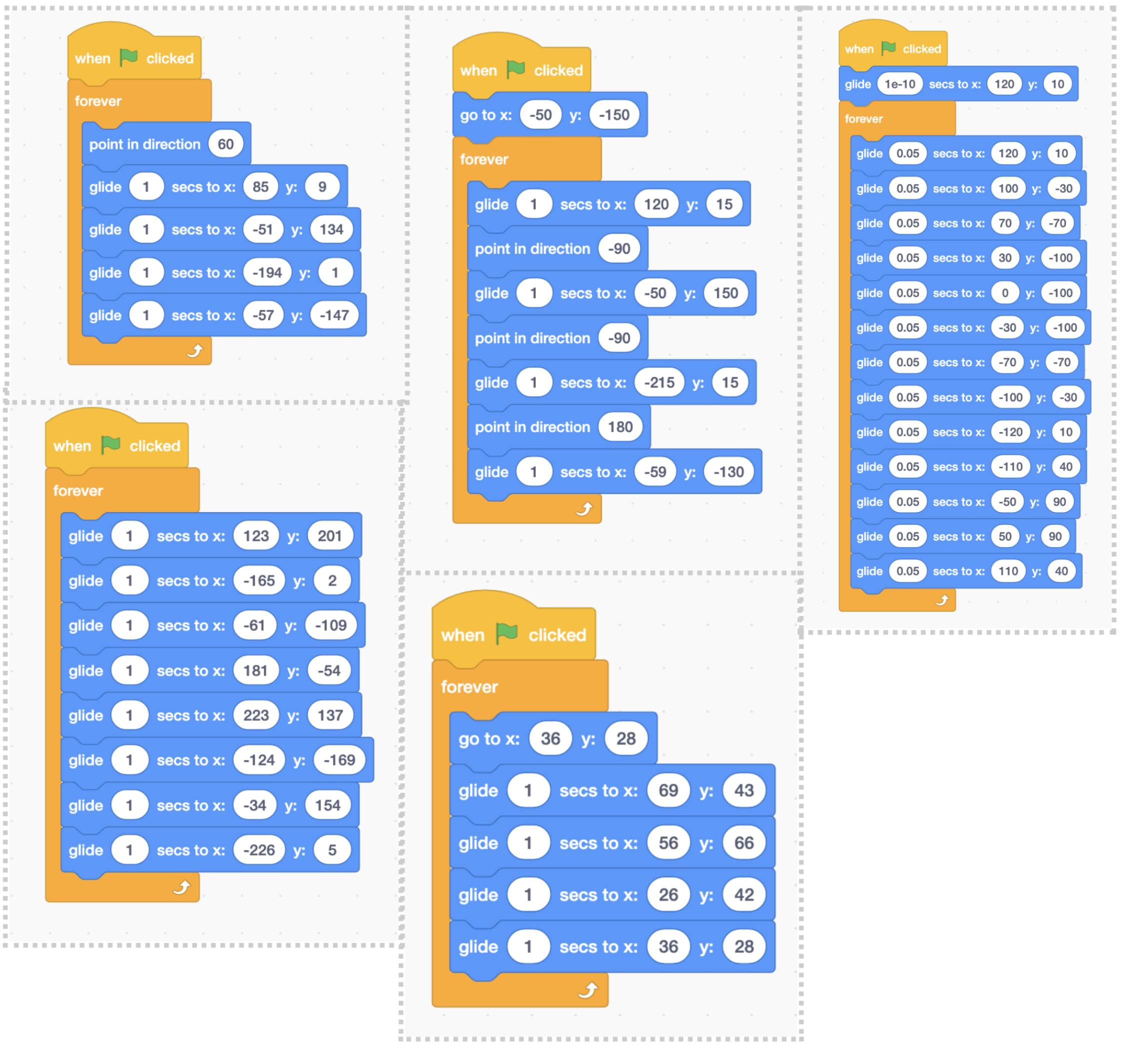
Example 1:

Chosen project: `https://scratch.mit.edu/projects/311242236/`

The nearest neighbors for this project all exhibited similar code structure to it. Each project contained a `control_forever` loop with `motion_glidesecstoxy` blocks nested inside of it, as shown in Figure 5-23. Although one of the projects was remixed from the same parent project as the original project, the other nearest neighbors were not related to the original project in this way.

The links to the **nearest neighbor projects** are listed below:

- `https://scratch.mit.edu/projects/309248329/`
- `https://scratch.mit.edu/projects/1362473/`
- `https://scratch.mit.edu/projects/206398363/`
- `https://scratch.mit.edu/projects/304297420/`



```
when ⚑ clicked
forever
    point in direction (60)
    glide (1) secs to x: (85) y: (9)
    glide (1) secs to x: (-51) y: (134)
    glide (1) secs to x: (-194) y: (1)
    glide (1) secs to x: (-57) y: (-147)
```

```
when ⚑ clicked
go to x: (-50) y: (-150)
forever
    glide (1) secs to x: (120) y: (15)
    point in direction (-90)
    glide (1) secs to x: (-50) y: (150)
    point in direction (-90)
    glide (1) secs to x: (-215) y: (15)
    point in direction (180)
    glide (1) secs to x: (-59) y: (-130)
```

```
when ⚑ clicked
glide (1e-10) secs to x: (120) y: (10)
forever
    glide (0.05) secs to x: (120) y: (10)
    glide (0.05) secs to x: (100) y: (-30)
    glide (0.05) secs to x: (70) y: (-70)
    glide (0.05) secs to x: (30) y: (-100)
    glide (0.05) secs to x: (0) y: (-100)
    glide (0.05) secs to x: (-30) y: (-100)
    glide (0.05) secs to x: (-70) y: (-70)
    glide (0.05) secs to x: (-100) y: (-30)
    glide (0.05) secs to x: (-120) y: (10)
    glide (0.05) secs to x: (-110) y: (40)
    glide (0.05) secs to x: (-50) y: (90)
    glide (0.05) secs to x: (50) y: (90)
    glide (0.05) secs to x: (110) y: (40)
```

```
when ⚑ clicked
forever
    glide (1) secs to x: (123) y: (201)
    glide (1) secs to x: (-165) y: (2)
    glide (1) secs to x: (-61) y: (-109)
    glide (1) secs to x: (181) y: (-54)
    glide (1) secs to x: (223) y: (137)
    glide (1) secs to x: (-124) y: (-169)
    glide (1) secs to x: (-34) y: (154)
    glide (1) secs to x: (-226) y: (5)
```

```
when ⚑ clicked
forever
    go to x: (36) y: (28)
    glide (1) secs to x: (69) y: (43)
    glide (1) secs to x: (56) y: (66)
    glide (1) secs to x: (26) y: (42)
    glide (1) secs to x: (36) y: (28)
```

Figure 5-23: Snippets of code from the nearest neighbor projects for Scratch project: `https://scratch.mit.edu/projects/311242236/`. Each dashed box corresponds to code from one of the projects.

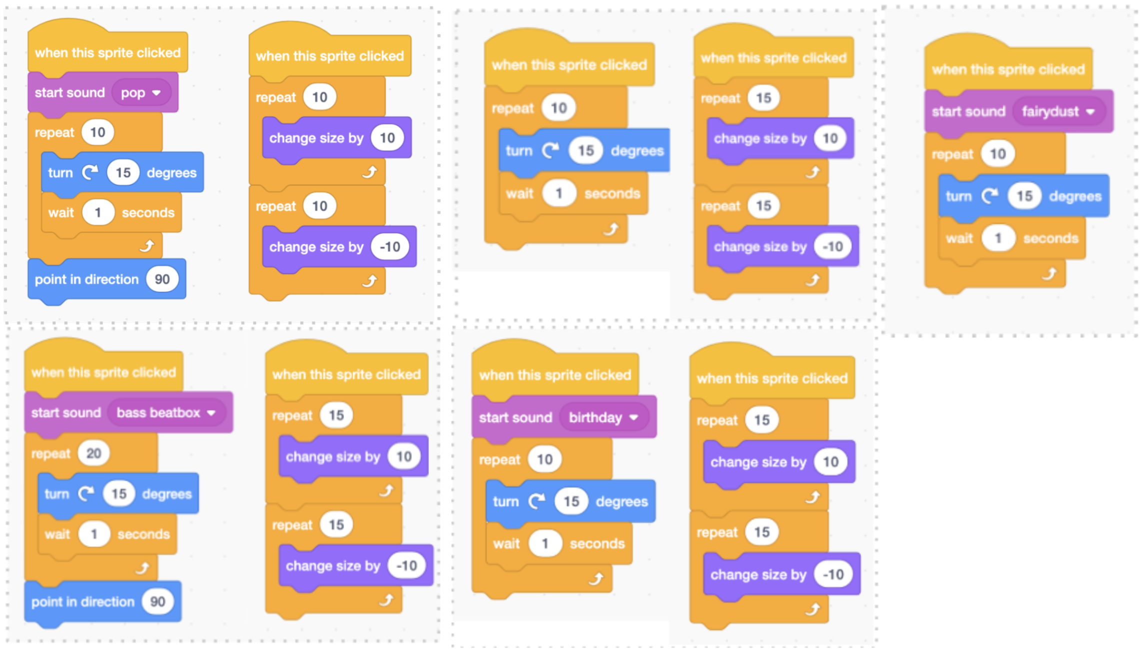
Example 2:

Chosen project: `https://scratch.mit.edu/projects/223682613/`

The nearest neighbors for this project also exhibited similar code structure to it. In this case, all of the projects involved animating a name in Scratch, based on one of the Scratch tutorials, so they expectedly shared similar code, as shown in Figure 5-24.

The links to the **nearest neighbor projects** are listed below:

- `https://scratch.mit.edu/projects/269937962/`
- `https://scratch.mit.edu/projects/250085163/`
- `https://scratch.mit.edu/projects/146961249/`
- `https://scratch.mit.edu/projects/79984870/`



```
when this sprite clicked
start sound pop
repeat 10
    turn ↻ 15 degrees
    wait 1 seconds
point in direction 90
```

```
when this sprite clicked
repeat 10
    change size by 10
repeat 10
    change size by -10
```

```
when this sprite clicked
repeat 10
    turn ↻ 15 degrees
    wait 1 seconds
```

```
when this sprite clicked
repeat 15
    change size by 10
repeat 15
    change size by -10
```

```
when this sprite clicked
start sound fairydust
repeat 10
    turn ↻ 15 degrees
    wait 1 seconds
```

```
when this sprite clicked
start sound bass beatbox
repeat 20
    turn ↻ 15 degrees
    wait 1 seconds
point in direction 90
```

```
when this sprite clicked
repeat 15
    change size by 10
repeat 15
    change size by -10
```

```
when this sprite clicked
start sound birthday
repeat 10
    turn ↻ 15 degrees
    wait 1 seconds
```

```
when this sprite clicked
repeat 15
    change size by 10
repeat 15
    change size by -10
```

Figure 5-24: Snippets of code from the nearest neighbor projects for Scratch project: `https://scratch.mit.edu/projects/223682613/`. Each dashed box corresponds to code from one of the projects. Each stack of blocks may not be for the same characters (i.e. sprites) within a program.

Crafted Project Examples and Their Similarities

I handcraft a few projects from scratch and measure the cosine similarity between their embeddings as well as show a heatmap visualization comparing their embeddings.

Simple Programs:

Figure 5-25 shows the two handcrafted Scratch programs. The only differing blocks are the `motion_movesteps` and `motion_turnright` blocks.

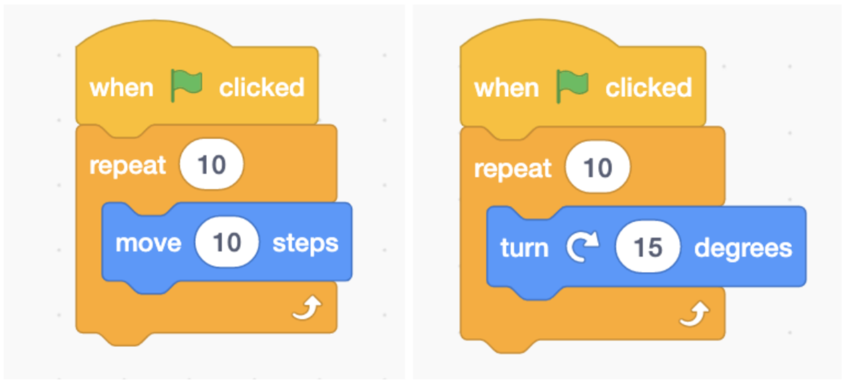


Figure 5-25: A pair of simple Scratch programs that are very similar, with the only differing blocks being the `motion_movesteps` (program on the left) and `motion_turnright` (program on the right) blocks.

Cosine Similarity:

The cosine similarity of these two programs is expectedly high (close to 1).

```
Cosine similarity:  0.9897882
```

Heatmap visualization:

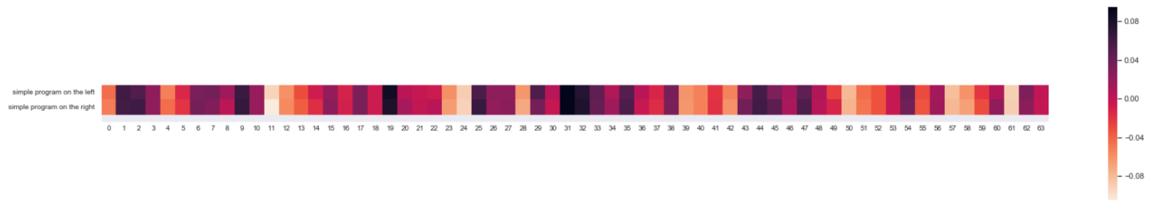
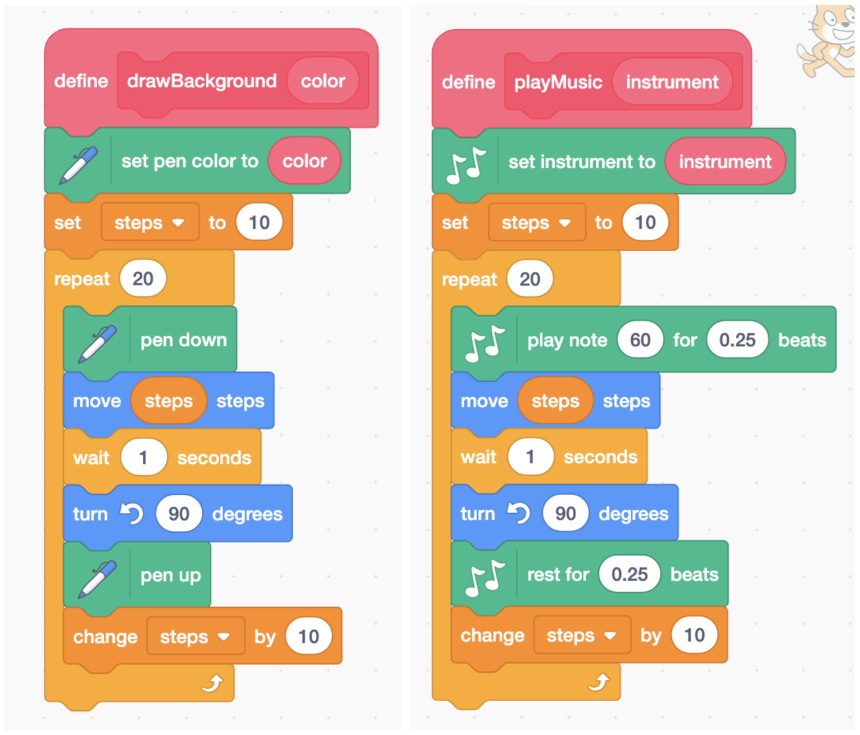


Figure 5-26: Heatmap visualization for the pair of (simple) Scratch programs shown in Figure 5-25.

It is evident from the heatmap visualization in Figure 5-26 that these two project embeddings share similar values in most of the dimensions. This is expected, since these programs only differ by one block in the program.

Complex Programs:

Figure 5-27 shows the two handcrafted Scratch programs. The main differing blocks are the extension blocks used, with the `pen` extension blocks in the program on the left and the `music` extension blocks in the program on the right.



```
define drawBackground (color)
set pen color to (color)
set [steps ▾] to (10)
repeat (20)
    pen down
    move (steps) steps
    wait (1) seconds
    turn ↺ (90) degrees
    pen up
    change [steps ▾] by (10)
end
```

```
define playMusic (instrument)
set instrument to (instrument)
set [steps ▾] to (10)
repeat (20)
    play note (60) for (0.25) beats
    move (steps) steps
    wait (1) seconds
    turn ↺ (90) degrees
    rest for (0.25) beats
    change [steps ▾] by (10)
end
```

Figure 5-27: A pair of more complex Scratch programs that are very similar, with the main differing blocks being the `pen` extension blocks (program on the left) and the `music` extension blocks (program on the right).

Cosine Similarity:

The cosine similarity of these two programs is expectedly high (close to 1).

```
Cosine similarity:  0.9795549
```

Heatmap visualization:

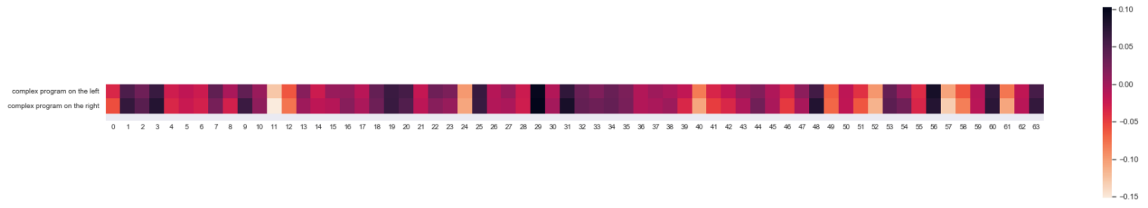
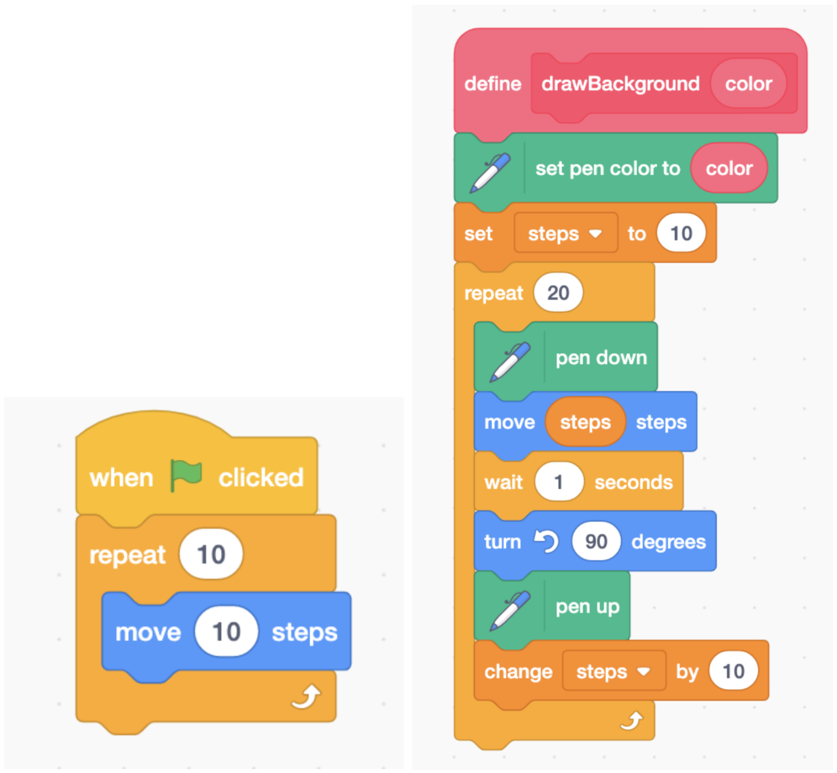


Figure 5-28: Heatmap visualization for the complex pair of Scratch programs shown in Figure 5-27.

Although a little less apparent than in the example of the simple programs, it is still evident from the heatmap visualization in Figure 5-28 that these two project embeddings share similar values in many of the dimensions. This is expected, since these programs only differ by a few blocks.

Across Program Types:

Figure 5-29 shows a cross-pairing of the previous programs, with a simple program on the left and a more complex program on the right. There are many differing blocks between these two programs.



```
when [flag] clicked
repeat 10
    move 10 steps
```

```
define drawBackground color
set pen color to color
set steps to 10
repeat 20
    pen down
    move steps steps
    wait 1 seconds
    turn ↺ 90 degrees
    pen up
    change steps by 10
```

Figure 5-29: A cross-pairing of (different) Scratch programs with many differing blocks.

Cosine Similarity:

The cosine similarity of these two programs is relatively lower than the two previous examples of similar programs, however, note that this similarity score is still considered objectively high.

```
Cosine similarity:  0.71526206
```

Heatmap visualization:

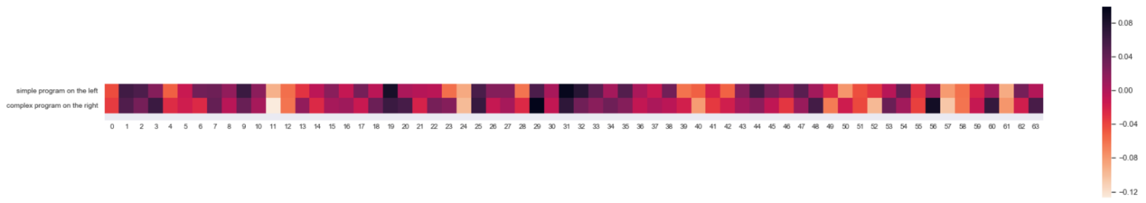


Figure 5-30: Heatmap visualization for the (differing) Scratch programs shown in Figure 5-29.

It is evident from the heatmap visualization in Figure 5-30 that these two project embeddings do not share as many similar values in the different dimensions as the previous examples of similar projects do (simple and complex pairs). This is expected, since these programs differ in many blocks as well as in their complexity.

5.2 Supervised Training Evaluation

5.2.1 Quantitative Evaluation

Accuracy Metrics:

The supervised model achieved an F1 score of 0.737, precision of 0.737, and recall of 0.737, as is shown by Table 5.1. For the "game" category, the supervised model achieved an F1 score of 0.811, precision of 0.789, and recall of 0.833. For the "animation" category, the supervised model achieved an F1 score of 0.832, precision of 0.800, and recall of 0.867. For the "other" category, the supervised model achieved an F1 score of 0.442, precision of 0.500, and recall of 0.395. The model did pretty well for the "game" and "animation" categories, achieving high F1 scores, and struggled quite a bit with the "other" category. This difference in performance can be attributed to the fact that the "other" category is essentially a blend of many different sub-categories by nature, as all projects that aren't games and animations get lumped into "other". As a result, we expect that it was difficult for the model to learn the commonalities between projects labeled as "other", especially with the small size of the supervised dataset.

Table 5.1: Accuracy Metrics for Supervised Classification on Test Set

	Overall	Game	Animation	Other
Precision	0.737	0.789	0.800	0.500
Recall	0.737	0.833	0.867	0.395
F1 Score	**0.737**	**0.811**	**0.832**	**0.442**

Confusion Matrix:

I create a confusion matrix [21] for the supervised model that demonstrates the number of projects that are predicted to belong to one category when they, in reality, belong to another. This confusion matrix is shown in Figure 5-31. As foreshadowed by their high F1 scores, the model does well in predicting true animations and games. The model evidently struggles with the “other” category, categorizing actual “other” projects practically uniformly at random between the three categories (see the last row of the matrix).

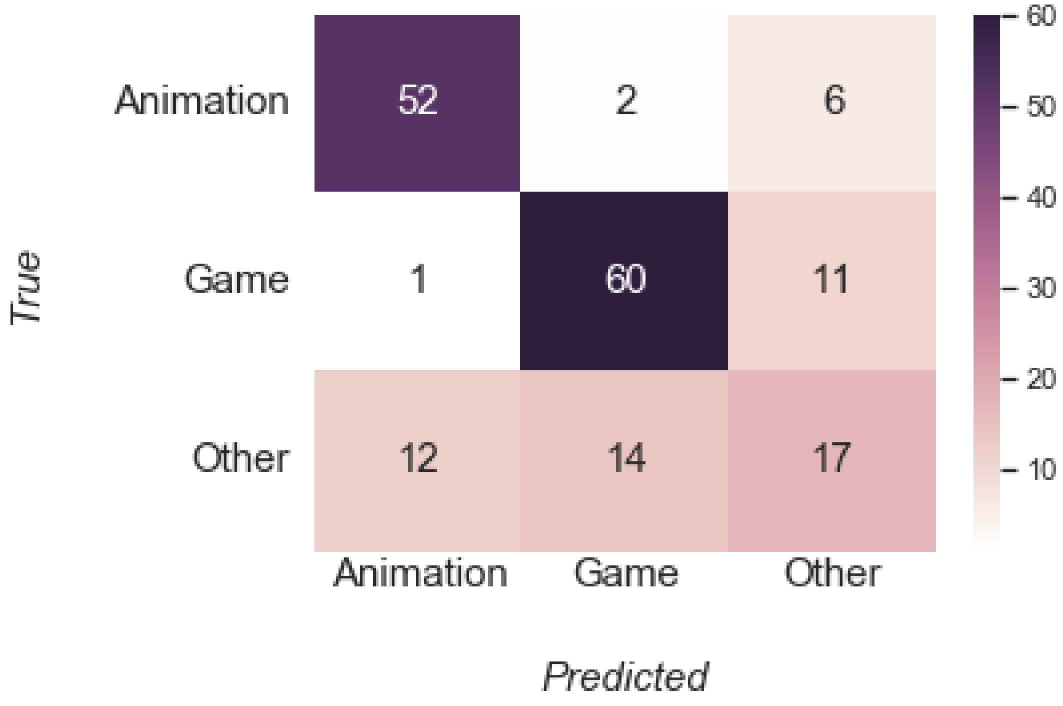


Figure 5-31: Confusion matrix for the supervised classifier model.

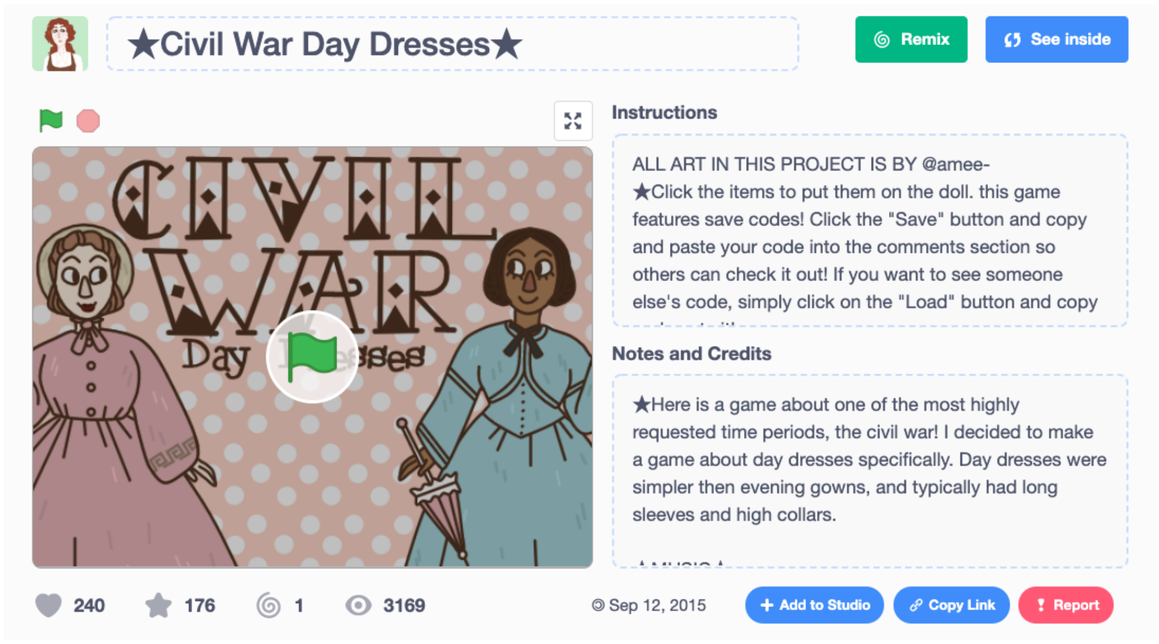
5.2.2 Qualitative Evaluation

Example Predictions for Select Projects

I examine the model predictions for a few select projects in each category. The model does well on most examples, with only two mistakes out of twelve.

Games

Example 1:



★Civil War Day Dresses★

Instructions

ALL ART IN THIS PROJECT IS BY @amee-
★Click the items to put them on the doll. this game features save codes! Click the "Save" button and copy and paste your code into the comments section so others can check it out! If you want to see someone else's code, simply click on the "Load" button and copy

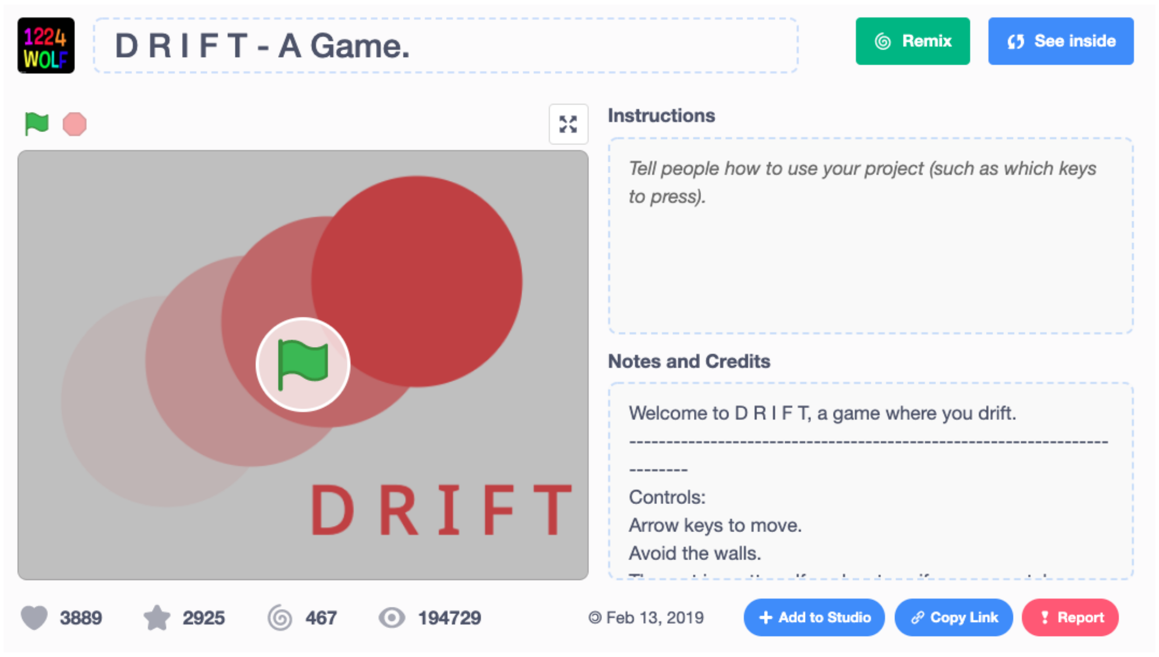
Notes and Credits

★Here is a game about one of the most highly requested time periods, the civil war! I decided to make a game about day dresses specifically. Day dresses were simpler then evening gowns, and typically had long sleeves and high collars.

Figure 5-32: This Scratch project can be found at: `https://scratch.mit.edu/projects/75395146`.

Model prediction: `game with probability 0.575`

Example 2:



Instructions

Tell people how to use your project (such as which keys to press).

Notes and Credits

Welcome to D R I F T, a game where you drift.

Controls:

Arrow keys to move.

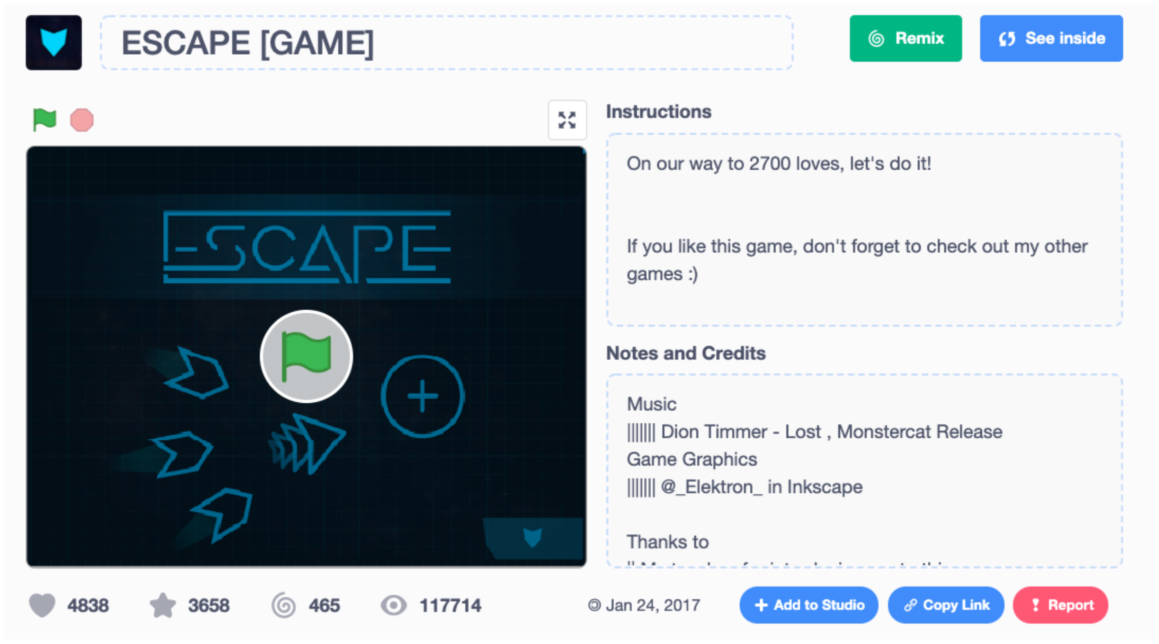
Avoid the walls.

3889 2925 467 194729 Feb 13, 2019

Figure 5-33: This Scratch project can be found at: `https://scratch.mit.edu/projects/286136792`.

Model prediction: `game with probability 0.799`

Example 3:


ESCAPE [GAME]

Instructions

On our way to 2700 loves, let's do it!

If you like this game, don't forget to check out my other games :)

Notes and Credits

Music
|||||| Dion Timmer - Lost , Monstercat Release
Game Graphics
|||||| @_Elektron_ in Inkscape

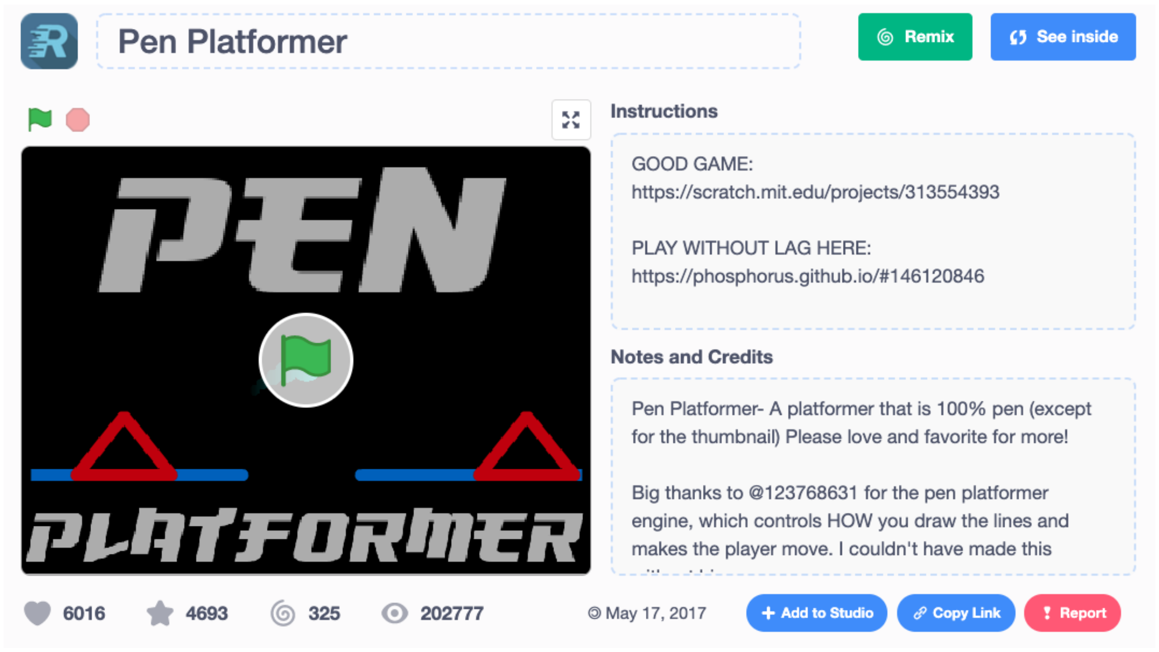
Thanks to

4838 3658 465 117714 Jan 24, 2017

Figure 5-34: This Scratch project can be found at: `https://scratch.mit.edu/projects/141677189`.

Model prediction: `other with probability 0.477`

Example 4:



Instructions

GOOD GAME:
https://scratch.mit.edu/projects/313554393

PLAY WITHOUT LAG HERE:
https://phosphorus.github.io/#146120846

Notes and Credits

Pen Platformer- A platformer that is 100% pen (except for the thumbnail) Please love and favorite for more!

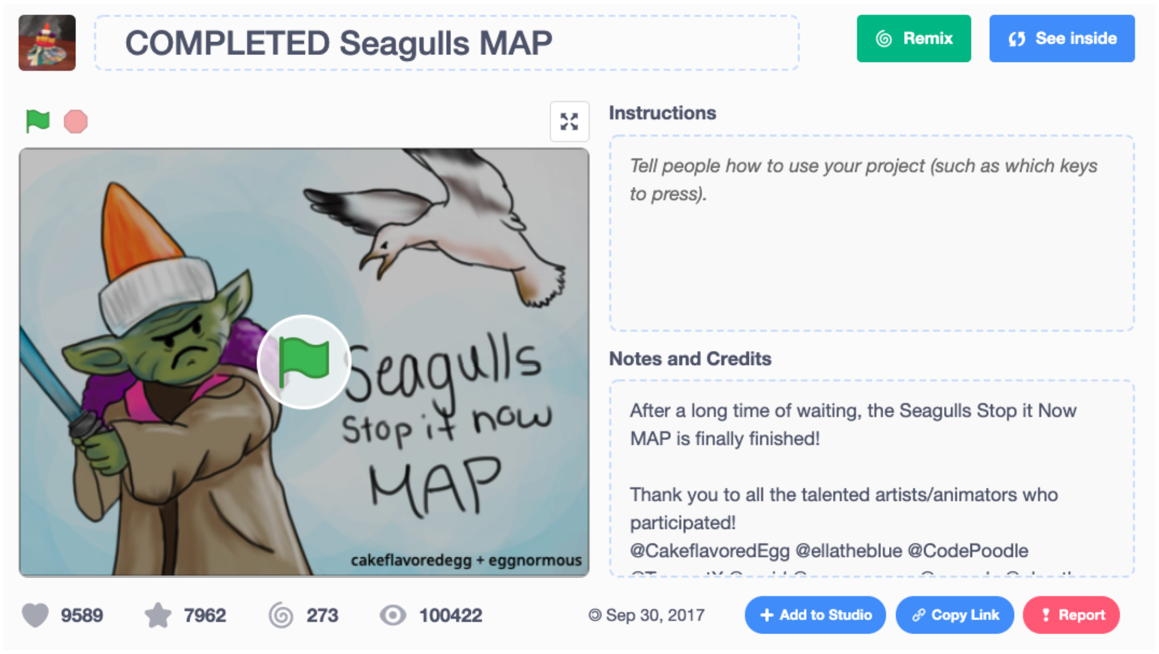
Big thanks to @123768631 for the pen platformer engine, which controls HOW you draw the lines and makes the player move. I couldn't have made this

Figure 5-35: This Scratch project can be found at: `https://scratch.mit.edu/projects/146120846`.

Model prediction: `game with probability 0.513`

Animations

Example 1:



Instructions

Tell people how to use your project (such as which keys to press).

Notes and Credits

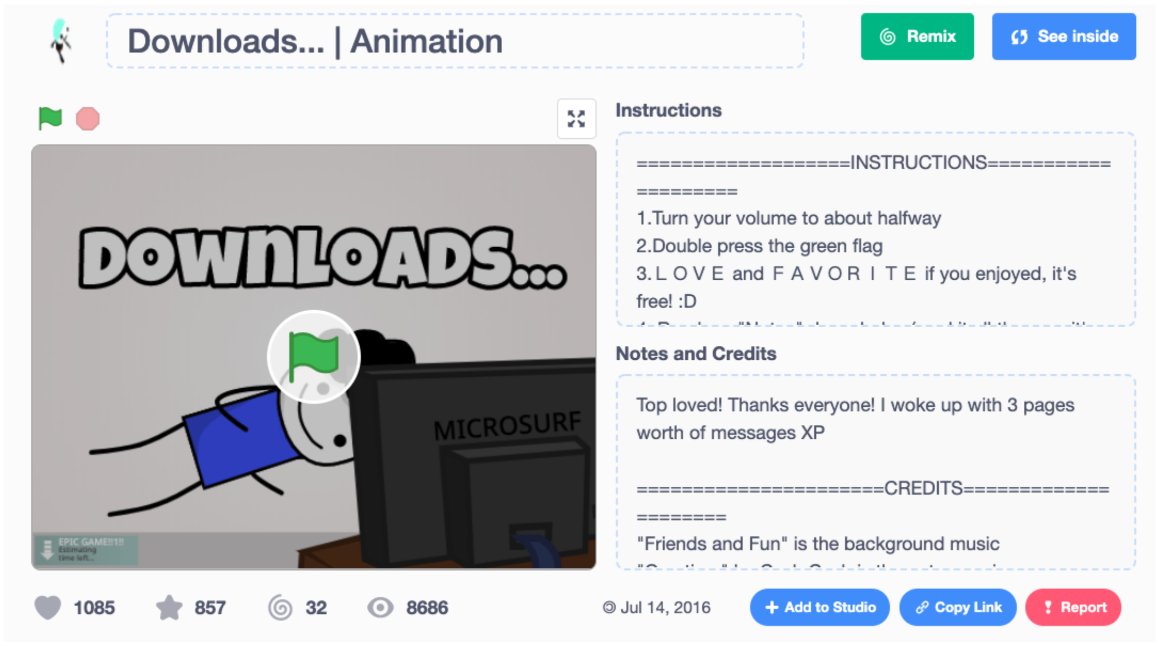
After a long time of waiting, the Seagulls Stop it Now MAP is finally finished!

Thank you to all the talented artists/animators who participated!
@CakeflavoredEgg @ellatheblue @CodePoodle

Figure 5-36: This Scratch project can be found at: `https://scratch.mit.edu/projects/169401431`.

Model prediction: `animation` `with probability 0.845`

Example 2:



===================INSTRUCTIONS====================
1.Turn your volume to about halfway
2.Double press the green flag
3.L O V E and F A V O R I T E if you enjoyed, it's free! :D

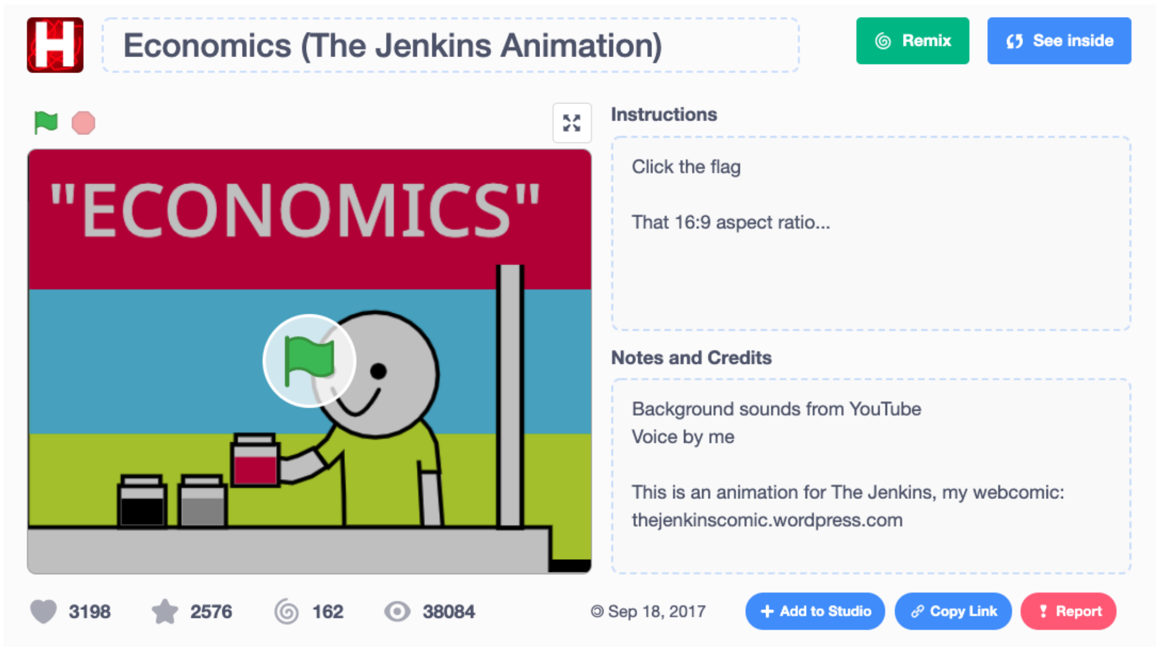
Top loved! Thanks everyone! I woke up with 3 pages worth of messages XP

=====================CREDITS=====================
"Friends and Fun" is the background music

Figure 5-37: This Scratch project can be found at: `https://scratch.mit.edu/projects/116260245`.

Model prediction: `animation with probability 0.901`

Example 3:


Economics (The Jenkins Animation)

Instructions

Click the flag

That 16:9 aspect ratio...

Notes and Credits

Background sounds from YouTube
Voice by me

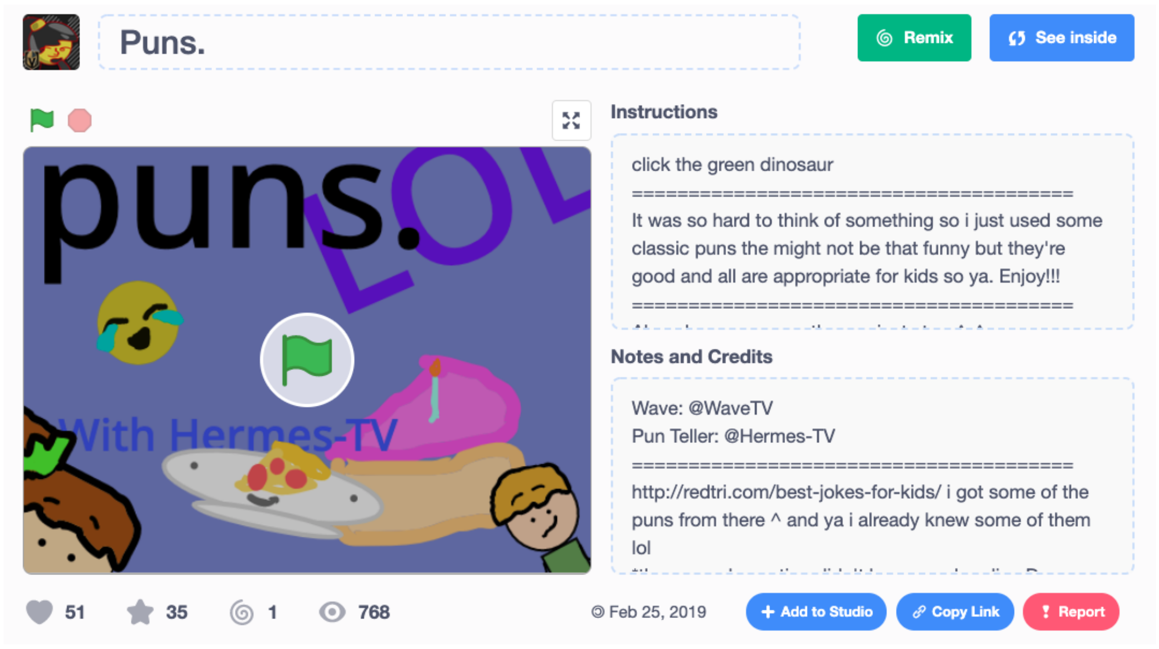
This is an animation for The Jenkins, my webcomic:
thejenkinscomic.wordpress.com

3198 2576 162 38084 Sep 18, 2017

Figure 5-38: This Scratch project can be found at: `https://scratch.mit.edu/projects/174103769`.

Model prediction: `animation with probability 0.536`

Example 4:



Instructions

click the green dinosaur
==
It was so hard to think of something so i just used some classic puns the might not be that funny but they're good and all are appropriate for kids so ya. Enjoy!!!
==

Notes and Credits

Wave: @WaveTV
Pun Teller: @Hermes-TV
==
http://redtri.com/best-jokes-for-kids/ i got some of the puns from there ^ and ya i already knew some of them lol

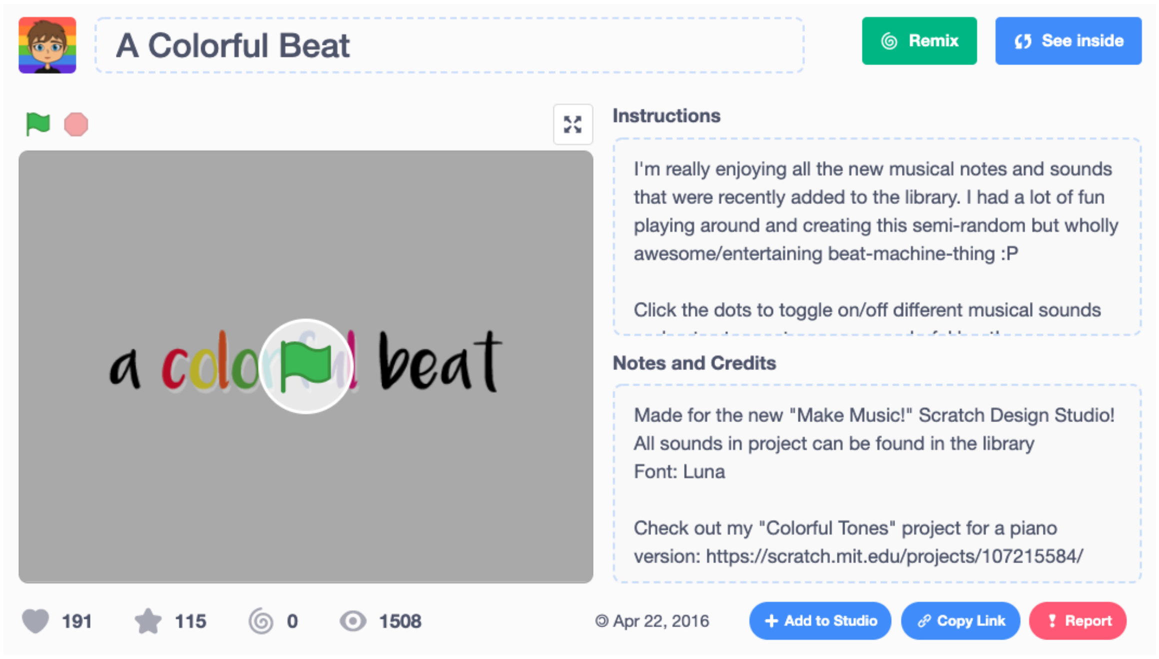
Feb 25, 2019

Figure 5-39: This Scratch project can be found at: `https://scratch.mit.edu/projects/288182786`.

Model prediction: `animation` `with probability 0.986`

Other

Example 1:



Instructions

I'm really enjoying all the new musical notes and sounds that were recently added to the library. I had a lot of fun playing around and creating this semi-random but wholly awesome/entertaining beat-machine-thing :P

Click the dots to toggle on/off different musical sounds

Notes and Credits

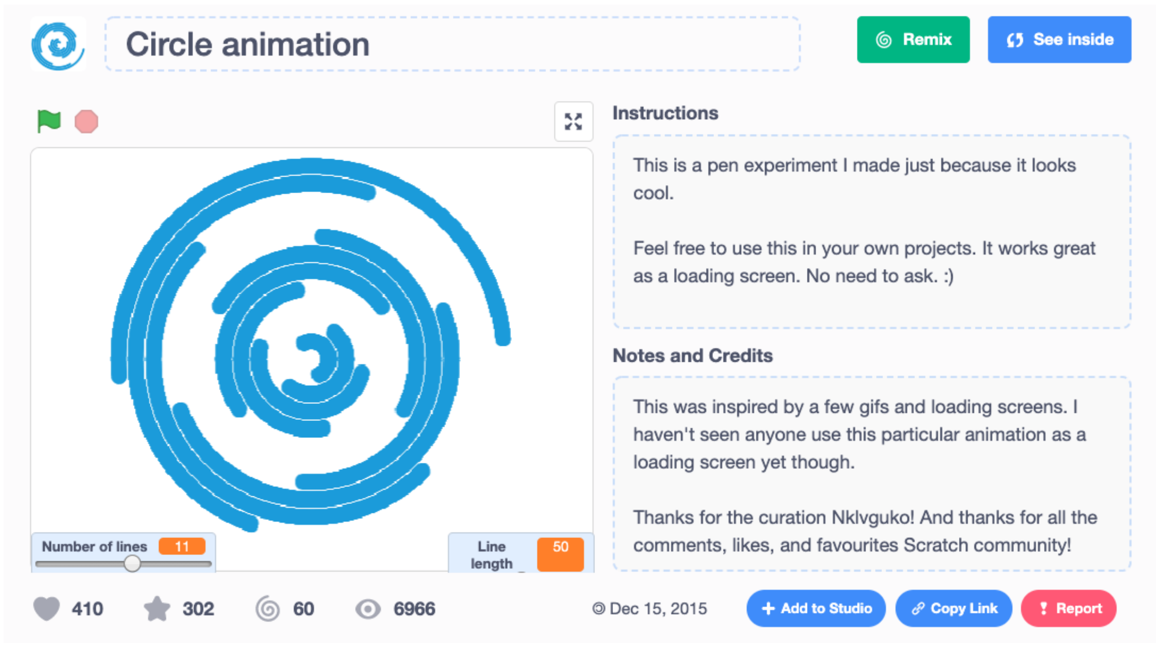
Made for the new "Make Music!" Scratch Design Studio!
All sounds in project can be found in the library
Font: Luna

Check out my "Colorful Tones" project for a piano version: https://scratch.mit.edu/projects/107215584/

Figure 5-40: This Scratch project can be found at: `https://scratch.mit.edu/projects/106553892`.

Model prediction: `other with probability 0.578`

Example 2:



Instructions

This is a pen experiment I made just because it looks cool.

Feel free to use this in your own projects. It works great as a loading screen. No need to ask. :)

Notes and Credits

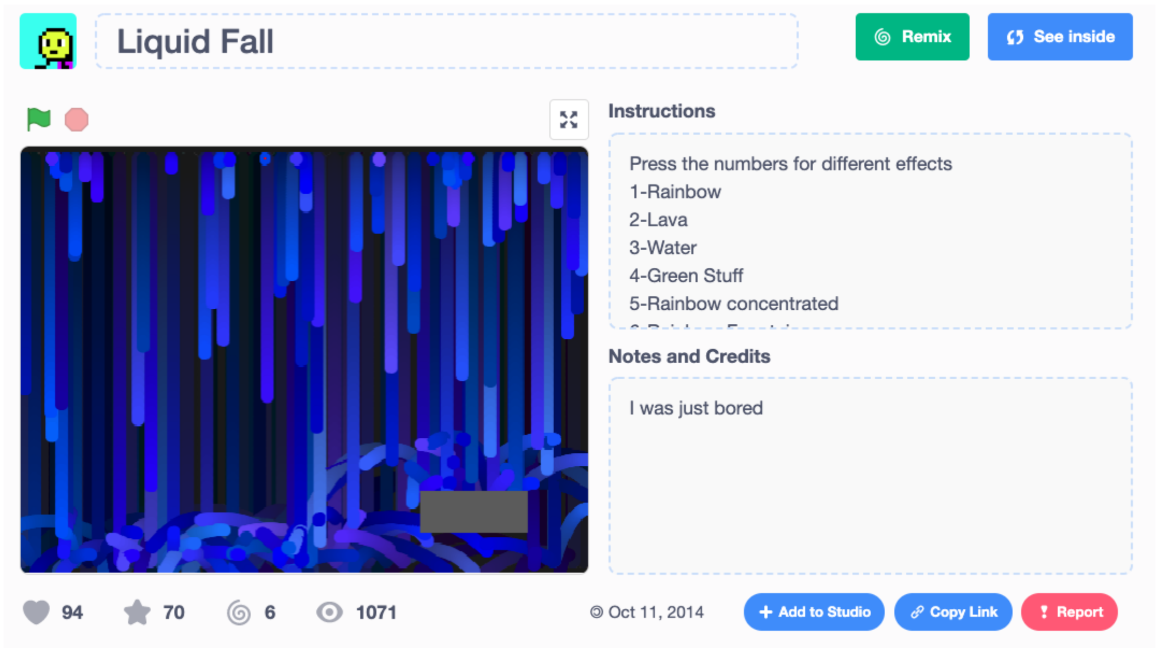
This was inspired by a few gifs and loading screens. I haven't seen anyone use this particular animation as a loading screen yet though.

Thanks for the curation Nklvguko! And thanks for all the comments, likes, and favourites Scratch community!

Figure 5-41: This Scratch project can be found at: `https://scratch.mit.edu/projects/91593031`.

Model prediction: `other with probability 0.919`

Example 3:


Liquid Fall

Instructions

Press the numbers for different effects
1-Rainbow
2-Lava
3-Water
4-Green Stuff
5-Rainbow concentrated

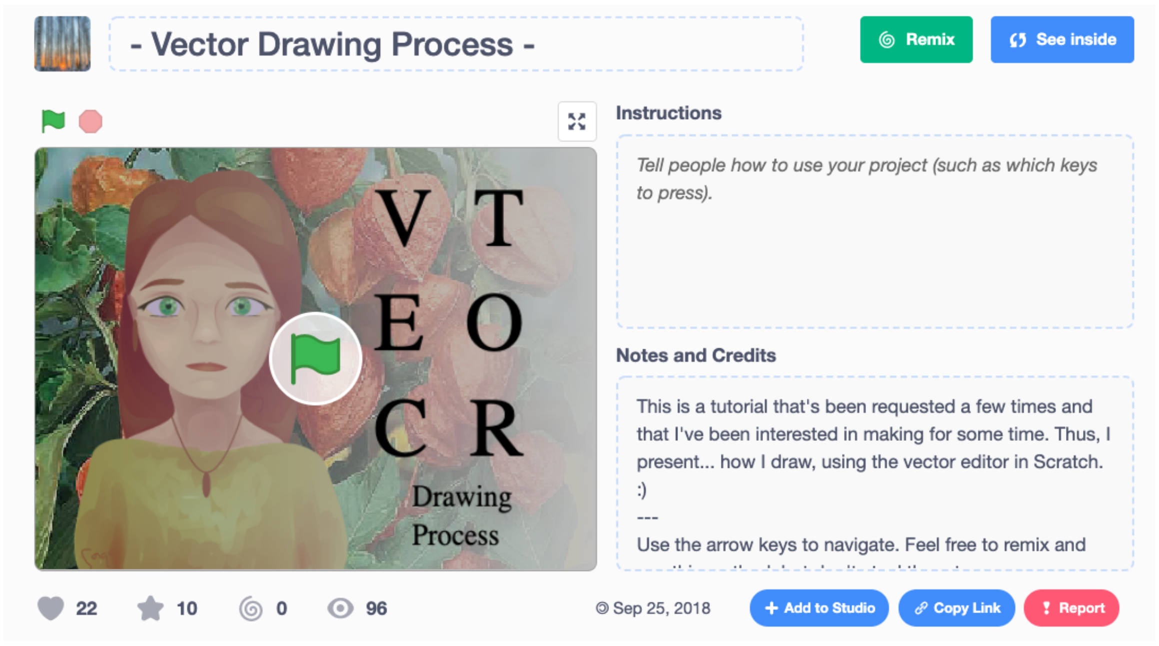
Notes and Credits

I was just bored

Figure 5-42: This Scratch project can be found at: `https://scratch.mit.edu/projects/29433528`.

Model prediction: `game` with probability 0.724

Example 4:



This is a tutorial that's been requested a few times and that I've been interested in making for some time. Thus, I present... how I draw, using the vector editor in Scratch. :)

Use the arrow keys to navigate. Feel free to remix and

Figure 5-43: This Scratch project can be found at: `https://scratch.mit.edu/projects/244219748`.

Model prediction: `other with probability 0.736`

Chapter 6

Conclusion

6.1 Contributions

Throughout this paper, I described the use of NLP techniques to vectorize and classify Scratch projects by type. This work has the following contributions:

- A labeled dataset of 873 Scratch projects and their corresponding types, with a confidence of greater than or equal to 80%, constructed through a process of consensus-based annotation by experts (i.e. staff of the Scratch Foundation).
- An unsupervised model of meaningful vector representations (i.e. "embeddings") for Scratch blocks based on the composition of 500,000 projects.
- A supervised classifier model that categorizes Scratch projects by type (into the categories of "animation", "game", and "other"), with an overall F1 Score of 0.737.
- An analysis of the unsupervised and supervised models, and an exploration of the elements learned during training.

Overall, I applied machine learning techniques to a new context, and demonstrated that NLP techniques can be used in the classification of computer programs to a reasonable level of accuracy.

6.2 Future Work

6.2.1 Engaging the Scratch Community For Annotation

The project annotation phase of this work drew upon the expertise of the Scratch Team for labeling Scratch projects by their type. Another community of people, namely the Scratch online community, have extensive experience and expertise with Scratch projects, and could be a promising pool of annotators to tap into as well. In addition, Scratchers may have different ways of thinking about project categorizations that the Scratch Team might miss. Combining the perspectives of both of these communities may result in a more robust and diverse set of project annotations, and eventually a better classifier model. Lastly, given that the classifier model could serve as the foundation for a project recommendation algorithm (see Section 6.2.6 - Recommendation Algorithm below), it would be essential to incorporate the perspectives of Scratchers in the project taxonomy, since they would be a primary end-user of the classifier model.

6.2.2 Overcoming Small Size of Labeled Dataset

Recent work has demonstrated effective methods for augmenting small datasets to improve the performance of text classifier models. Wei et. al. present "easy data augmentation techniques", such as "synonym replacement, random insertion, random swap, and random deletion" within a sentence while retaining and conserving the original label [42]. Their work has especially larger gains for smaller datasets. Although their work is specifically targeted towards text applications, we can experiment with using some of their methodology for overcoming the limitations of our small dataset. In a similar vein, active learning techniques aimed at gauging the value of unlabeled data samples by how informative they can be, and as a result, possibly improving the performance of a model with less data, could prove useful for our work [40]. All of these methods could be promising directions to apply to our work of training a classifier model on Scratch projects, where, as described earlier, labeling Scratch projects has proven to be a time- and resource-consuming process.

6.2.3 Hyperparameter Tuning on Larger Dataset

Given time constraints, the hyperparameter tuning process followed in this work was conducted on the smaller 10,000 project dataset, rather than on the larger 500,000 project dataset. In future work, more rigorous hyperparameter tuning on the larger 500,000 dataset can be conducted to see the full range of improvements possible (using more computing resources).

6.2.4 Expanding Scratch Project Encodings

This work encoded Scratch projects in a “linear” text-based method, traversing each stack of blocks and encoding blocks sequentially. Alternatively, we can explore encoding Scratch projects using graphical representations. A potential avenue to explore is to incorporate the Scratch program’s abstract syntax tree (AST) in the encoding, following the `code2vec` work [17]. In `code2vec`, the authors explored representing blocks of code using multiple “paths” between leaf nodes in the code’s AST [17]. Although software code and Scratch programs are not exactly the same, one can imagine defining what a “path” through a Scratch program’s AST would look like, and applying the methodology used in `code2vec` to explore whether better representations for Scratch programs are learned this way. Another avenue for improving the methodology for encoding Scratch projects is to incorporate information about the shapes of blocks. The shape of a Scratch block tends to enforce certain affordances, and imply certain limitations about how the block can be used and where it can be placed in the program. Since many Scratch blocks share similar shapes, this information would be useful for the model to pick up on so that it can learn the similarities between these similarly shaped blocks. A potential way for incorporating this information could be to encode each block as an “object” that has attributes (shape, inputs, menu options, etc).

6.2.5 Classifying by "Complexity"

In addition to classifying Scratch projects by type, we could also classify them by their level of complexity. There are a few notions of complexity to consider, but a goal of this future direction would be to develop an intuitive notion of a project's complexity, both from the software and perceptual perspective. From the software perspective, complexity measures such as McCabe's cyclomatic complexity (which evaluates the different paths through a program [30]), Halstead metrics (such as program vocabulary, length, difficulty, delivered bugs [24]) and data access complexity [20] (Card metric), among others, can be considered. In addition, we can explore a "blocks" complexity concept that will incorporate the complexity and sophistication level of the blocks used in the Scratch project as well as the complex usage of blocks together. From the perceptual perspective, complexity could entail how the project is perceived when interacted with. The number of moving parts, characters, features, scenes and concepts that the project introduces, among other things, can be examined. A part of this future direction will involve clearly defining and laying out the criteria for a project's complexity. Once such a concise definition is reached, we can build a labeled dataset of Scratch projects and their complexity levels, and train a classifier model on this dataset.

6.2.6 Recommendation Algorithm

To address the disparity of attention in the Scratch online community, we could explore ways to make Scratch projects visible to Scratchers in a natural, personalized, and sustainable way. An integrated project recommendation algorithm that will suggest projects to Scratchers is one way to achieve this goal. The classifier model trained as part of this work would naturally feed into this recommendation algorithm. Moreover, if a complexity classifier is also trained (see Section 6.2.5 – Classifying by "Complexity"), it can also serve as an engine for the recommendation algorithm. Armed with a way to classify projects by type and complexity, we can search our database of projects to pull "related" examples to surface to the user. The definition of what counts as a “related” project could include metrics such as:

- How closely a project relates to a Scratcher’s interests, as measured by their previous likes and favorites on the platform.
- How diverse a project’s type and complexity are from the Scratcher’s usual preference, as measured by their previous likes and favorites on the platform.
- How closely a project’s programming difficulty level resembles the Scratcher’s currently demonstrated programming skills.

A key priority of this recommendation algorithm would be to personalize the project discovery experience for a Scratcher. Of course, we wish to present the Scratcher with different uses of Scratch, to broaden their perspective on the power of Scratch, and the creativity it enables. However, in order to ensure our algorithm is sustainable and relevant, we must personalize our recommendation to the Scratcher and their interests. That way, the recommendations presented by our algorithm feel natural but contribute to the Scratcher’s learning implicitly.

An important aspect of designing the project recommendation algorithm will involve tailoring it to Scratchers and their learning goals. Thus, the design of this algorithm will likely require qualitative interviews with Scratchers to determine their preferences and goals for a project recommendation interface.

Appendix A

Tables

A.1 Mapping of Symbols to Transitions and Artifacts:

Table A.1: Symbols to Transitions

`_STARTSTACK_`	beginning of a new stack
`_ENDSTACK_`	end of a stack
`_STARTNEST_`	beginning of nesting
`_ENDNEST_`	end of nesting
`_STARTINPUT_`	beginning of input
`_ENDINPUT_`	end of input
`_NEXT_`	next

Table A.2: Symbols to Artifacts

`numtext_input`	numeric or textual input
`_VAR_`	variable
`_LIST_`	list
`menu_option`	a chosen menu option
`_MENU_`	dropdown menu
`_NUMTEXTARG_`	numeric or textual argument*
`_BOOLARG_`	boolean argument*
`procedures_definition`	custom procedure definition**
`procedures_call`	custom procedure call

* *These symbols are only used with* ***`procedures_definition`*** *block.*

*****`procedures_definition`*** *is analogous to a software function.*

A.2 All Blocks Used (in Corpus of 500,000 Projects) By Category:

Table A.3: Motion Category to Blocks

Motion:	motion_changexby
	motion_changeyby
	motion_direction
	motion_glidesecstoxy
	motion_glideto
	motion_goto
	motion_gotoxy
	motion_ifonedgebounce
	motion_movesteps
	motion_pointindirection
	motion_pointtowards
	motion_setrotationstyle
	motion_setx
	motion_sety
	motion_turnleft
	motion_turnright
	motion_xposition
	motion_yposition

Table A.4: Looks Category to Blocks

Looks:	looks_backdropnumbername
	looks_changeeffectby
	looks_changesizeby
	looks_changestretchby
	looks_cleargraphiceffects
	looks_costumenumbername
	looks_goforwardbackwardlayers
	looks_gotofrontback
	looks_hide
	looks_hideallsprites
	looks_nextbackdrop
	looks_nextcostume
	looks_say
	looks_sayforsecs
	looks_seteffectto
	looks_setsizeto
	looks_setstretchto
	looks_show
	looks_size
	looks_switchbackdropto
	looks_switchbackdroptoandwait
	looks_switchcostumeto
	looks_think
	looks_thinkforsecs

Table A.5: Sound Category to Blocks

Sound:	sound_changeeffectby
	sound_changevolumeby
	sound_cleareffects
	sound_play
	sound_playuntildone
	sound_seteffectto
	sound_setvolumeto
	sound_stopallsounds
	sound_volume

Table A.6: Events Category to Blocks

Events:	event_broadcast
	event_broadcastandwait
	event_whenbackdropswitchesto
	event_whenbroadcastreceived
	event_whenflagclicked
	event_whengreaterthan
	event_whenkeypressed
	event_whenthisspriteclicked

Table A.7: Control Category to Blocks

Control:	control_create_clone_of
	control_delete_this_clone
	control_forever
	control_if
	control_if_else
	control_repeat
	control_repeat_until
	control_start_as_clone
	control_stop
	control_wait
	control_wait_until

Table A.8: Sensing Category to Blocks

Sensing:	sensing_answer
	sensing_askandwait
	sensing_coloristouchingcolor
	sensing_current
	sensing_dayssince2000
	sensing_distanceto
	sensing_keypressed
	sensing_loud
	sensing_loudness
	sensing_mousedown
	sensing_mousex
	sensing_mousey
	sensing_of
	sensing_resettimer
	sensing_setdragmode
	sensing_timer
	sensing_touchingcolor
	sensing_touchingobject
	sensing_userid
	sensing_username

Table A.9: Operators Category to Blocks

Operators:	operator_add
	operator_and
	operator_contains
	operator_divide
	operator_equals
	operator_gt
	operator_join
	operator_length
	operator_letter_of
	operator_lt
	operator_mathop
	operator_mod
	operator_multiply
	operator_not
	operator_or
	operator_random
	operator_round
	operator_subtract

Table A.10: Variables Category to Blocks

Variables:	data_changevariableby
	data_hidevariable
	data_setvariableto
	data_showvariable
	data_addtolist
	data_insertatlist
	data_itemnumoflist
	data_itemoflist
	data_lengthoflist
	data_listcontainsitem
	data_replaceitemoflist
	data_deletealloflist
	data_deleteoflist
	data_hidelist
	data_showlist

Table A.11: Procedures Category to Blocks

Procedures:	procedures_call
	procedures_definition

Table A.12: Music Category to Blocks

Music:	music_changeTempo
	music_getTempo
	music_midiPlayDrumForBeats
	music_midiSetInstrument
	music_playDrumForBeats
	music_playNoteForBeats
	music_restForBeats
	music_setInstrument
	music_setTempo

Table A.13: Pen Category to Blocks

Pen:	pen_changePenColorParamBy
	pen_changePenHueBy
	pen_changePenShadeBy
	pen_changePenSizeBy
	pen_clear
	pen_penDown
	pen_penUp
	pen_setPenColorParamTo
	pen_setPenColorToColor
	pen_setPenHueToNumber
	pen_setPenShadeToNumber
	pen_setPenSizeTo
	pen_stamp

Table A.14: Video Sensing Category to Blocks

Video Sensing:	videoSensing_setVideoTransparency
	videoSensing_videoOn
	videoSensing_videoToggle
	videoSensing_whenMotionGreaterThan

Table A.15: Text to Speech Category to Blocks

Text to Speech:	text2speech_setLanguage
	text2speech_setVoice
	text2speech_speakAndWait

Table A.16: Makey Makey Category to Blocks

Makey Makey:	makeymakey_whenCodePressed
	makeymakey_whenMakeyKeyPressed

Table A.17: Micro:bit Category to Blocks

Micro:bit:	microbit_displayClear
	microbit_displaySymbol
	microbit_displayText
	microbit_getTiltAngle
	microbit_isButtonPressed
	microbit_isTilted
	microbit_whenButtonPressed
	microbit_whenGesture
	microbit_whenPinConnected
	microbit_whenTilted

Table A.18: LEGO MINDSTORMS EV3 Category to Blocks

LEGO MINDSTORMS EV3:	ev3_beep
	ev3_buttonPressed
	ev3_getDistance
	ev3_getMotorPosition
	ev3_motorSetPower
	ev3_motorTurnClockwise
	ev3_motorTurnCounterClockwise
	ev3_whenButtonPressed

Table A.19: LEGO BOOST Category to Blocks

LEGO BOOST:	boost_getMotorPosition
	boost_seeingColor

Table A.20: LEGO Education WeDo Category to Blocks

LEGO Education WeDo:	wedo2_getDistance
	wedo2_getTiltAngle
	wedo2_isTilted
	wedo2_motorOff
	wedo2_motorOn
	wedo2_motorOnFor
	wedo2_playNoteFor
	wedo2_setLightHue
	wedo2_setMotorDirection
	wedo2_startMotorPower
	wedo2_whenDistance
	wedo2_whenTilted

Table A.21: Vernier Go Direct Category to Blocks

Vernier Go Direct:	gdxfor_getTilt
	gdxfor_whenGesture
	gdxfor_whenTilted

A.3 Hyperparameter Values Tested:

Table A.22: Unsupervised Hyperparameters

type of unsupervised model:	skipgram
	cbow
dimension of embeddings:	50
	64
	128
	175
	200
min number of word occurrences:	1
	5
min / max length of the char n-grams:	(1, 5)
	(1, 8)
	(1, 10)
num. of epochs / learning rate:	(5, 0.1)
	(10, 0.05)
	(25, 0.01)
	(50, 0.01)

Table A.23: Supervised Hyperparameters

`min number of word occurrences:`	`1`
	`5`
`num.  of epochs / learning rate:`	`(5, 0.1)`
	`(10, 0.1)`
	`(25, 0.05`
	`(50, 0.05))`
`size of word n-grams to consider:`	`1`
	`5`
	`10`

Appendix B

Figures

B.1 t-SNE Plot for All Blocks:

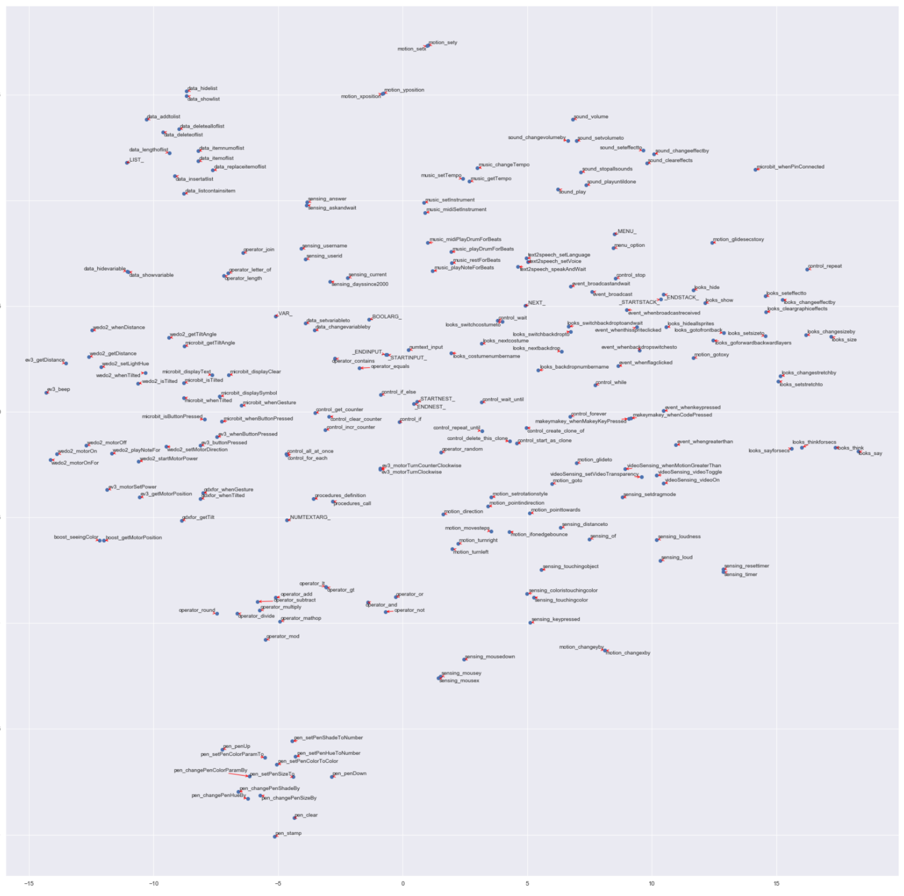


Figure B-1: t-SNE visualization of all the blocks and symbols in our corpus.

Bibliography

[1] Amazon Mechanical Turk. https://www.mturk.com/.

[2] fastText. https://fasttext.cc/.

[3] fastText FAQ page. https://fasttext.cc/docs/en/faqs.html/.

[4] fastText "List of options" page. https://fasttext.cc/docs/en/options.html.

[5] Figure Eight. https://www.figure-eight.com/.

[6] Figure Eight Success Center. https://success.figure-eight.com/hc/en-us.

[7] Figure Eight Success Center. How to Calculate a Confidence Score. https://success.figure-eight.com/hc/en-us/articles/201855939-How-to-Calculate-a-Confidence-Score?mobile_site=false.

[8] Github issue thread describing how sentence vectors are computed from word vectors. https://github.com/facebookresearch/fastText/issues/323#issuecomment-353167113.

[9] One-hot Encoder. Scikit-learn library. https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder.html.

[10] Scratch About page. https://scratch.mit.edu/about/.

[11] Scratch Statistics page. https://scratch.mit.edu/statistics//. Accessed: Jan 16, 2020.

[12] subprocess — Subprocess management. Python library. https://docs.python.org/3.7/library/subprocess.html.

[13] Tomas Mikolov's reply re: differences between CBOW and Skip-gram on Google Groups thread. https://groups.google.com/forum/#!searchin/word2vec-toolkit/c-bow/word2vec-toolkit/NLvYXU99cAM/E5ld8LcDxlAJ.

[14] word2vec. Google Code. https://code.google.com/archive/p/word2vec/.

[15] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4):1–37, 2018.

[16] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. *arXiv preprint arXiv:1711.00740*, 2017.

[17] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019.

[18] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics*, 5:135–146, 2017.

[19] Karen Brennan and Mitchel Resnick. New frameworks for studying and assessing the development of computational thinking. In *Proceedings of the 2012 annual meeting of the American educational research association, Vancouver, Canada*, volume 1, page 25, 2012.

[20] David N Card and Robert L Glass. *Measuring software design quality.* Prentice-Hall, Inc., 1990.

[21] Tom Fawcett. An introduction to roc analysis. *Pattern recognition letters*, 27(8):861–874, 2006.

[22] Yoav Goldberg. Neural network methods for natural language processing. *Synthesis Lectures on Human Language Technologies*, 10(1):1–309, 2017.

[23] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning.* MIT press, 2016.

[24] Maurice Howard Halstead et al. *Elements of software science*, volume 7. Elsevier New York, 1977.

[25] Jiawei Han, Micheline Kamber, and Jian Pei. Data mining concepts and techniques third edition. *Morgan Kaufmann*, 2011.

[26] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 837–847. IEEE, 2012.

[27] Armand Joulin, Edouard Grave, Piotr Bojanowski, and Tomas Mikolov. Bag of tricks for efficient text classification. *arXiv preprint arXiv:1607.01759*, 2016.

[28] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681*, 2018.

[29] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(Nov):2579–2605, 2008.

[30] Thomas J McCabe. A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320, 1976.

[31] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[32] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. Convolutional neural networks over tree structures for programming language processing. In *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[33] Hao Peng, Lili Mou, Ge Li, Yuxuan Liu, Lu Zhang, and Zhi Jin. Building program vector representations for deep learning. In *International Conference on Knowledge Science, Engineering and Management*, pages 547–553. Springer, 2015.

[34] Jeffrey Pennington, Richard Socher, and Christopher D Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014.

[35] Chris Piech, Jonathan Huang, Andy Nguyen, Mike Phulsuksombati, Mehran Sahami, and Leonidas Guibas. Learning program embeddings to propagate feedback on student code. *arXiv preprint arXiv:1505.05969*, 2015.

[36] Anand Rajaraman and Jeffrey David Ullman. *Mining of massive datasets*. Cambridge University Press, 2011.

[37] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, et al. Scratch: programming for all. *Communications of the ACM*, 52(11):60–67, 2009.

[38] Mitchel Resnick and Ken Robinson. *Lifelong kindergarten: Cultivating creativity through projects, passion, peers, and play*. MIT press, 2017.

[39] Rebecca Russell, Louis Kim, Lei Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul Ellingwood, and Marc McConley. Automated vulnerability detection in source code using deep representation learning. In *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 757–762. IEEE, 2018.

[40] Burr Settles. Active learning literature survey. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 2009.

[41] Joseph Turian, Lev Ratinov, and Yoshua Bengio. Word representations: a simple and general method for semi-supervised learning. In *Proceedings of the 48th annual meeting of the association for computational linguistics*, pages 384–394. Association for Computational Linguistics, 2010.

[42] Jason W Wei and Kai Zou. Eda: Easy data augmentation techniques for boosting performance on text classification tasks. *arXiv preprint arXiv:1901.11196*, 2019.

[43] Martin White, Christopher Vendome, Mario Linares-Vásquez, and Denys Poshyvanyk. Toward deep learning software repositories. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 334–345. IEEE, 2015.

[44] Ludwig Wittgenstein. *Philosophical investigations*. John Wiley & Sons, 2009.