Suggested Blocks: Using Neural Networks To Aid Novice Programmers In App Inventor

Maja Svanberg

Submitted in Partial Fulfillment of the Prerequisite for Honors in Computer Science under the advisement of Franklyn Turbak

May 2018

© 2018 Maja Svanberg
Acknowledgements

Many hours have gone into researching and writing these pages. Solving the big problems, like “Why is nothing working?”, and “What do I do for the next 20 hours until my thing is done?”, wouldn’t have been possible if it wasn’t for the set of people I have been lucky to meet in the past few years. In particular, I owe thanks to the following people:

Thank you to my family, mom, dad, sister, brother-in-law, grandmothers, aunt, uncle, cousins, and dogs, for keeping me grounded in knowing that at the end of the day, it doesn’t matter what I do, since you would never leave me alone even if I asked you to. Special thank you to those of you who live with me. I couldn’t have asked for a better family.

Thank you to friends, near and far, and to the many communities at Wellesley who have supported me and shaped me into the person I am.

Thank you to my Wellesley hostess, Victoria Lia ’00 (and of course, Gary!), whose presence in my Wellesley experience always succeeded in giving me perspective.

Thank you to the professors who taught some life-changing courses while I have been at Wellesley, especially Eni, Sravana, Rhys, Ada, Takis, Stella, Ben, Jean, Randy, Lyn, Professor Bidart, Professor Kerr, Casey Pattanayak, Professor Diesl, Professor Volič, and Professor Trenk.

Special thanks to Sravana Reddy, for giving me the idea that prompted this thesis, and to Scott, for helping me with the question “Why is nothing working?”.

Thank you also to Professor Sipser, Professor Gupta, Professor Williams, and Professor Vogan at MIT, for your teaching.

Thank you to David Bau, Mark Sherman, Mike Tissenbaum, and Paul Medlock-Walton at MIT, for your time spent talking to me about this project.

Thank you to Jeff Schiller, also MIT, for providing me with the dataset that made this project possible.

Thank you to Benjamin Xie, University of Washington, for contributing to the ai2_summarizer.py. And thank you for, two years ago, asking a college sophomore if they knew anything about machine
learning, which prompted them to start learning about it.

Thank you, Eni, for encouraging me to apply myself, and giving me the courage to execute and explore, instead of just imagining.

Thank you Isabelle Li ’20, for sharing your in-depth knowledge about the prolific dataset, and your consistent willingness to offer your help and advice.

Thank you to the Office of the Provost and the Wellesley College Computer Science department for generously providing the funding that made this thesis possible.

Thank you to past and current members of my thesis committee, Eni, Brian, Robbie, and Orit.

Lyn,

Thank you for letting a first year who had never taken your classes do research with you.
Thank you for being a soundboard throughout my time at Wellesley.
Thank you for taking the risk in advising me, knowing that I tend to do my own thing.
Thank you for juggling ideas with me, however unrelated or unconventional they may be.
Thank you for continuing to remind me to explain myself.

Thank you for your undying effort in aiding the learning and progress of, not just me, but all students you teach.

Finally, thank you for always being excited. It’s really infectious.

We did it!
Abstract

MIT App Inventor is a programming environment in which users build Android applications by connecting blocks together. Because its main audience is beginner programmers, it is important that users are given the proper guidance and instruction to successfully become creators. In order to offer this help, App Inventor provides text-based tutorials that describe the workflow of example programs to users. However, studies have shown that out-of-context help such as tutorials has little to no effect on learning, and when given the choice, users prefer in-context hints and suggestions. In order for users to overcome some of the barriers with self-training, we need to provide them with relevant information and in-context suggestions. Therefore, I am introducing Suggested Blocks, a data-driven model that leverages machine learning to provide users with relevant suggestions of which blocks to include in their programs.

In this project, I focused on developing the neural networks to power a suggested blocks system. Using original apps from real App Inventor users, I developed a set of experiments to discover plausible vector representations of the data, including tree traversals, n-grams, tree structures, as well as different network architectures to generate the best possible block suggestions for the users. The objective is not only to be accurate, but to provide suggestions that are sensible, relevant, and most importantly, educational. When simulating the best model on reconstructing an original project from a novice user, suggesting only 10 blocks at a time, the user would be able to drag-and-drop 60% of her blocks straight from the Suggested Blocks drawer. Overall, the results show promise for a future implementation of a Suggested Blocks system.
Chapter 1

Introduction

1.1 Blocks Programming Languages

When using a Blocks Programming Language (BPL), programmers are able to write their programs by connecting blocks, not unlike putting together pieces of a jigsaw-puzzle. The shapes and colors of the blocks act as visual cues to the functionality and structure of the program. BPLs use visual features of the blocks to alleviate the syntactic frustrations and cognitive load that distract novice programmers from focusing on more central concepts, thereby allowing them to build their first apps faster [Bau+17].

Being able to browse through the library of blocks is said to be one of the key features to the success of blocks programming [WW15]. After a block has been selected, it can be dragged and dropped onto the workspace, effectively writing code. In many BPLs the library is categorized into drawers by properties of the block-types. The color of the drawer and the blocks within it is a visual cue that makes memorization of the location of the block easy (Fig. 1-1). When you click on a drawer, a fly-out menu opens in the workspace where the relevant blocks are displayed (Fig. 1-2).

Apart from the procedures drawer, the collections of blocks in each drawer are statically generated, hard-coded in JavaScript by the App Inventor implementers. In the procedures drawer, blocks used to define a procedure are statically generated, but procedure call blocks only appear once a procedure has already been defined (Fig. 1-3, Fig. 1-4). While drawers are helpful, users spend up to 16% of their working time searching for blocks [Rod+17].

Another visual cue is the nubs, notches, plugs, and sockets, that connect the blocks. The nubs at the bottom of statement blocks fit into the notches at the top of other statement blocks, allowing a
vertical composition that signifies their order of execution. Plugs at the left end of expression blocks fit into the sockets at the right end of statement blocks, allowing for a transfer of a value from the plug to the socket. These two different pairings visually distinguish statements from expressions, and thereby prevent syntactic errors that are possible in text-based languages. These BPL features, along with the natural language description of each block, help lower the barriers to programming [WW15] and thereby democratize the access to the creation of digital tools. The cognitive load placed on the novice user as they build the program is structured to be more intrinsic to the concepts compared to syntactic details and language features. This is done without reducing the inherent cognitive load: programming concepts [Swe10]. Removing distractions allows the user to stay focused on the high-level creation aspects of the program. Rather than struggling with syntax, the user is allowed
Blocks Programming Environments (BPEs) that leverage BPLs are many. This paper will focus on MIT App Inventor [Appa], that is built on top of Blocky. Other notable BPEs include Snap! [Sna], Looking Glass [Loo], LogoBlocks [Log], Scratch [Scr], StarLogo [Sta], and Alice [Ali].

1.2 MIT App Inventor

MIT App Inventor is an online BPE that allows users to create Android smartphone apps. On its website, it states that it seeks “to democratize software development by empowering all people,”

Figure 1-3: The procedure drawer when the workspace is empty.

Figure 1-4: The procedure drawer after two procedures, proc1 and proc2 have been defined and named.
especially young people, to move from technology consumption to technology creation” [Appb] and this idea was reiterated upon by Wolber et al. in “Democratizing Computing with App Inventor” [WAF15]. It does so by its use of Blockly, as well as creating a stronger bridge between programming and creating tools for the user’s direct community. With MIT App Inventor, 6.3 million users have created 25 million apps, and the number of monthly active users is over 0.3 million [Appc].

Its interface consists of two parts: the Designer Editor and the Blocks Editor. In the Designer Editor, the user drags components onto a blank screen, controlling the user interface of their apps. Components can be either visual, e.g., a Button or a Vertical Arrangement, or non-visual, e.g., a Location Sensor. Visual components are able to let the user interact with the program, i.e., you can click a Button, but sometimes they only serve to arrange the layout of the user interface, e.g., a Vertical Arrangement. Non-visual components represent features of the phone and allow for the programmer to access for example sensors, memory, or internet connectivity.

In the Blocks Editor, the user employs blocks to give functionality to the components they added in the Designer Editor. Here, blocks are organized into drawers. The user drags-and-drops the desired blocks from the drawers onto the workspace to define the functionality of the components. When finished, the app can be downloaded and installed on smartphones with just the click of a button, or uploaded to the gallery to share with the community.

Because of the presence of components in App Inventor and because each component comes with its set of possible actions, the number of block-types in App Inventor is virtually unlimited, in contrast with other BPEs. For example, App Inventor has thousands of possible block-types, but in for example Scratch, this number is in the 100s. This poses possibilities as well as challenges for the users and implementers of App Inventor.

1.2.1 The Need for Democratization of Programming

Recently, there have been multiple news stories about the lack of gender diversity in the technology industry. Sometimes the stories even shed light on adversity toward efforts to make the sphere more inclusive of women and minorities. From the sexist Google Manifesto [Ore17] to Siri defaulting as female [Hil11], society has been experiencing the consequences of the cultural as well as gender uniformity among influencers in this powerful industry. Meanwhile, the pipeline into the industry, computer science college programs, is male-dominated. In 2015, the percentage of American CS bachelor’s degrees awarded to women was 18% [ES16a] [ES16b]. Research into gender in technology shows that this is a downward spiral — perception of one’s own “maleness” is a strong indicator
of how one assesses their fit in tech [LAY16]. The perceived fit is in turn correlated with retention in college level CS classes. We as a society need to work even harder to break the spiral, and one approach toward that is to focus on CS education and democratize not only programming but creation through programming.

The way the end-user perceives their confidence, support, and motivation within a programming environment differ between the average woman and the average man [BB04]. Therefore, an environment can be made to cater to either male-pattern or female-pattern learners. For example, women are less likely to be motivated by a productized, monetizable end-product [BB04], and are more likely to respond negatively to criticism [Rob91]. Further, being able exploring possibilities through tinkering has shown to be more beneficial to men than women [Bec+06]. I discuss this further in Sec. 1.3.3. If we want to reach women when we democratize programming, it is important to take their needs into consideration when designing the environments in which they learn.

1.3 Barriers To Success in Online Learning Environments

1.3.1 Lack of Social Context and Guidance

Questions have been raised as to whether or not online learning environments alone are successful at teaching computational concepts. A 2015 study [LK15] evaluated the learning outcomes of beginners when using the interactive tutorial platform CodeAcademy [Cod], the puzzle-oriented game Gidget [Gid], and the open-ended creative canvas feature of Gidget (not unlike App Inventor’s canvas) [Lee+14]. While the former two showed success in teaching fundamental CS concepts, working on the canvas environment did not. It even showed a slight decline in test results over time, as the participants seemed discouraged by the task. This highlights a need for supplemental material when learning to use a canvas environment, which could be done with both online and offline help.

In situations where a social context was given, canvas environments have proven to be successful in teaching basic programming skills. In a 2008 study by Maloney et al., a group of children and older mentors, all without previous programming experience, used Scratch for a summer [Mal+08]. The results showed that participants were able to pick up and develop an understanding of programming concepts and use these to create basic games. The skills developed progressively during the course of the study and included loops, variables, and user interaction. Here, the importance of social connection and presence of mentorship is highlighted as motivation for students to develop their skills. The study was able to engage a rather even gender ratio, suggesting that girls were not
discouraged by the tasks.

### 1.3.2 Lack of In-Context Help

Prospective creators may not have access to social situations where programming is the focus. If we want to democratize programming, we need to build effective tools to enable users to self-train. For this purpose, App Inventor provides 31 tutorials [Appd]. These tutorials range from simple four-block programs to advanced tutorials showing users both how to leverage the hardware of the phone, as well as teach them programming concepts. Further, a number of independent sources provide their own similar educational material to users [Ima] [Pur] [Uta].

However, both literature within learning theory and novice programming agree that in order for novices to make use of instructional material, it needs to be presented to them in a convenient form. According to Carrol et al. in *The Minimal Manual*, users are not inclined to read instructional material but are rather interested in taking action. They tend to skip crucial material if it does not pertain directly to the problem they are solving [Car+87]. In a study of how users interact with educational resources, Hnin et al. confirmed this hypothesis in the context of the BPE Looking Glass [HIK17] [Loo]. When given access to documentation, code snippet suggestions, puzzles, and tutorials, users were drawn to in-context resources (Fig. 1-5). They only spent time on tutorials in the last quarter of the time building their project, potentially to suggest that users only went there when other resources were exhausted. This poses a big limitation to how help is currently provided within App Inventor, as App Inventor provides documentation [Appe] and tutorials [Appd] but not suggestions or puzzles.

However, tutorials might still carry a lot of value. Hnin discussed a limitation of in-context help: it does not allow for the same in-depth learning that tutorials and documentation does. What it primarily does is that it lowers the bar to entry. However, literature has shown that when the bar is low for entry into an online learning environment (such as a MOOC), a lot more people enroll. The same mechanic, a low bar for entry, is also the reason behind high attrition rates in these environments [Moo04]. This leaves us with an unanswered question: If we mitigate some of the initial learning difficulties using a limited in-context help, would that just mean that the BPE transfers superficial learning? Or, assuming that the help is able to guide the novice to a first app, would that boost their confidence and give them the courage to tackle the next challenges more head on?
1.3.3 Marginalization of Learning Styles

As mentioned in Sec. 1.2.1, there is a difference in learning styles between individuals. Further, this difference sometimes lies along gender divides. If we were to create an environment that caters more to male-pattern learning, this would have an adverse effect on women [BB04]. In order to address this potential problem, and counteract the lack of diversity in the technology industry, we need to understand how this notion manifests within the environments we are creating. In an ideal scenario, we are able to build environments that support and encourage women to program, without creating unnecessary disturbances that would drive men away from the platforms.

Confidence and self-perceived ability are factors identified by Beckwith et al. that impact the success of programmers. Analyzing HackerNews [Hac] comments, Barik concluded in 2017 that “fun”, as a motivator in programming, can be approached through different lenses, including art, tinkering, playground, and spontaneity [Bar17]. He described these in relation to software engineering, and not necessarily novice programmers. However, tinkering and play in software engineering is gendered — women do not necessarily find the same enjoyment as men do [Bec+06] [Bec+05]. Women tend to stay within a sphere of comfort when building, taking time to understand what is going on. While this makes them on average better at debugging than men [Bec+06] and might be the reason why women perform better on gamified tasks [McL+17], in an open canvas environment, this puts them...
at a disadvantage. What gamification might be providing to the users is a confirmation of ability, which for women would raise confidence. Here, we would want to incorporate features into the environment that encourage users to tinker.

While there is no difference in ability when it comes to actually learning new features, lower confidence cause women to make more mistakes since they are not getting the help they need [Bec+05]. This is aggravated by the fact that women respond more negatively and strongly to evaluative criticism as compared to men [Rob91]. This is another challenge that novice women face when trying to self-train themselves to program. This points to a need for encouragement in order for women to succeed. Positive feedback in addition to error messages could not only help with confidence but, if given properly, could also encourage tinkering and exploring on their own.

Not only do environments regulate feedback systems and notifications, but the way they are designed also affect how they are able to motivate users. For example, women are more encouraged by collaboration and connections with others, as compared to creating an end product from which they could drive personal profit [BB04]. In terms of App Inventor, publishing their app and seeing the number of downloads rise would be less meaningful than sharing the app with their communities and seeing an impact. This is a huge strength for App Inventor as a platform, which emphasizes the power that the environment holds.

However, it is not only when it comes to the design of programming environments that male-pattern learners are more benefited. In their 1992 paper “Epistemological Pluralism and the Revaluation of the Concrete”, a reflection of classroom culture, Turkle and Papert describe the female learning patterns as soft, as opposed to hard male learning patterns [TP92]. Soft learning patterns are typically deployed in the humanities, in the close way literature teaches us to engage with meaning. Programming, on the other hand, is taught with distance and objectivity, alienating soft learners. While the harder school of thought is dominant in computer science education, one begins to wonder how a softer learning style would be able to break into the field without having to compromise their way of thinking and identity. Self-training could be a way to allow these users to learn without the imposition of objectivity to how they are learning. However, with the confidence, motivation, and support issues that face self-training women, the question remains, how can we make sure that we integrate gender-inclusive designs into our self-training and BPEs.
1.4 Crowd-Sourcing Blocks Recommendations

In order to address these barriers, I am introducing Suggested Blocks, a crowd-sourced, in-context tool, that suggests blocks to the user based on the current state of the project. The design of suggested blocks is another drawer in the blocks editor, where at any point in the production the user can open the drawer to receive intelligent suggestions on which block to add next (Fig. 1-6, Fig. 1-7).

The Suggested Blocks tool leverages one of the most important resources of our time, data. The data used consists of projects created by App Inventor users that have created at least 20 projects each (so-called “prolific” [LTM17] users). The environment takes the data, transforms it, and trains a machine learning model to extract overarching patterns from the data. It approximates a function to predict which block the user is likely to want to insert into the program, and serves this to the user in the Suggested Blocks drawer. The objective of this project is to develop a practical machine-learning-based algorithm for populating the Suggested Blocks drawer based on the current contents.
of the App Inventor workspace.

1.5 Why Suggested Blocks?

Having a high-quality intelligent system give block suggestions is an attempt at addressing the barriers discussed earlier in this chapter: (1) lack of in-context help, (2) lack of social context and guidance, and a (3) marginalization of learning styles.

Apart from errors and warnings, App Inventor relies mainly on tutorials as instructional material for self-training novice users. Only recently have tutorials been semi-integrated into the environment with the side panel App Building Guides [Appa]. However, when a user is trying to figure out a way to start a creative project or extend an already existing one, finding this knowledge in the current infrastructure requires leaving the environment. By adding a passive suggested blocks system that is reading the state of the project, we can steer the user in a direction other users have gone before.
While this could have been done with active suggestions, or an intervening warning feature, these tend to be disruptive in cases where the user is already receiving real-life help [Tis17]. In fact, these systems can be despised enough to make it into popular culture. The Microsoft Office Assistant, commonly known as *Clippy*, was available in Microsoft Office in the late 90s and early 2000s. It was removed, but has made its return as a meme where applications ironically add Clippy advice in their user interfaces [Cli] [Mey15]. Suggested Blocks, however, would be a passive actor in the background, that is hardly noticeable when suggestions are unwanted.

Since we know that women are less prone to tinkering with unfamiliar features, but should also be encouraged to do so, I suggest we find a way to bring familiarity and trust to what users do. Elevating a smaller number of blocks, from the thousands of readily available to a user, might mitigate the sense of unfamiliarity and make tinkering and creating easier for this demographic. Although this would likely correlate with being able to make an accurate prediction algorithm, let us assume that building follows a pattern. If so, suggesting a smaller number of blocks could help users make relevant choices, reducing the risk of bugs and, subsequently, perceived failure. Block suggestions could be confidence boosting and speed up the rate of development.

While challenges remain to all three barriers, block suggestions could be a small step on the path toward making self-training in BPEs more viable. The first step toward knowing whether a data-driven approach could be helpful in an open-ended environment is to develop a viable algorithm. A viable algorithm would be an algorithm that not only gives suggestions based on the data but gives suggestions that are sensible and relevant: Sensible with respect to that the blocks suggested can actually be added to the project; Relevant in the sense that the blocks suggested could be aligned with a user’s hypothetical goals. Since it is an open-ended environment, i.e., we do not know the end-goals of the user, relevance is loosely defined and we need to look at each project from several different perspectives when doing the evaluation.

Because the purpose of this study is to improve the user experience, it is clear that a user study would be needed to prove that an algorithm is truly helpful to programmers. Before we can get there, we need to find the algorithm with the highest potential of being helpful. The set of potential algorithms is infinite, and we need to confine ourselves to a subset of algorithms. Strong candidates when it comes to such a subset are neural networks. They are good for actual implementation with their quick prediction time and have gained large recognition in recent years for their ability to recognize complex patterns [Cho17]. Therefore, I will be exploring the potential of neural networks trained on App Inventor data to be a viable algorithm for a user study on suggested blocks system.
Chapter 2

Related Work

2.1 Blockly Shadow Blocks

In App Inventor, sometimes the sockets of a block are pre-filled (Fig. 2-1). This is meant to make coding faster because there is no need to figure out which block-type fills the socket and what appropriate values for them would be. It also removes the need to drag in several blocks instead of one. However, sometimes these blocks are not the ones the user would want. If they want to add another element, for example, a variable, procedure call, or an operator, it takes an effort to remove the default blocks and replace them.

![Figure 2-1: The for_each block in App Inventor comes with pre-filled integer values.](image)

To mitigate these issues, in Fall 2015 the developers of Blockly launched the shadow block feature [Sha]. They are pre-filled semi-transparent blocks (Fig. 2-2), that aim to balance the effort it takes to drag-and-drop the default block, and the effort it would take to remove it from its default location.
The user is able to either click the shadow block to activate it or drag another block into the socket, automatically removing the shadow block. Further, they communicate to the user the default value, as well as the expected input type of a socket.

Figure 2-2: Blockly repeat_k_times block, with prefilled math_number shadow block with the value 10. Image taken from [Sha].

2.2 Intelligent Tutoring Systems

Intelligent Tutoring Systems (ITSs) are becoming more prevalent in different programming languages. An ITS is any system that emulates a human tutor by aiding a student in their progress while the student is working independently. In 2017, Price et al. called for a stronger presence of ITSs in BPLs [PB17]. They suggested using data-driven approaches as these scale to new and larger problems, whereas using heuristics does not.

A heuristics approach would involve having a human agent identify the rules that are necessary for the ITS to be good for a learner, i.e., hard-coding the suggestions. This might involve both quantitative and qualitative looks at data, as well as the inclusion of learning theory into the process. A key benefit of heuristics is that for each suggestion given, it is easy to backtrace and understand why the suggestion was given, providing the user with valuable insight. It is also easy to target specific bad habits or mistakes of novices, e.g., when a procedure is defined but never called, or a chunk of code is copy-pasted 20 times instead of used within a loop.

A data-driven approach, on the other hand, involves writing a program that reads data and draws insights to present in the form of suggestions. It has the benefit that it notices examples not necessarily discovered by humans and, as stated above, is scalable in a way that using heuristics is not. In order to make up for the benefits of heuristics, however, it is important that the data the algorithm draws from is sound so that unsound programming habits are not perpetuated. The data
also needs to be sizable enough as to not overfit to the existing projects by capturing patterns that do not generalize to other datasets.

I will now present different approaches to building ITSs that aim to solve different problems across both text-based and block-based platforms.

### 2.2.1 HelpMeOut

As a response to novices not knowing what to do with error messages, in 2010, researchers at UC Berkeley created HelpMeOut [Har+10]. HelpMeOut is a crowdsourced recommender system for text-based languages. It works by drawing examples of how to respond to error messages and fix bugs from change-tracking of other programming projects. When a user encounters a similar error, the model returns a code example of how to avoid the error. The model has shown to provide a useful fix in 47% of the cases, but suggested unuseful fixes in 25% of the cases and returned nothing otherwise. These numbers were not compared to approaches to debugging such as searching StackOverflow or Google. The authors claim that the success rate will rise the longer the application is deployed, since not only can users submit their own fixes, but it also tracks what fixes are helpful and not, through reinforcement learning. Reinforcement learning is allowing a machine learning model to determine its ideal behavior while interacting with its environment [KLM96].

HelpMeOut is scalable across platforms and languages, and when the study was published, it had been implemented in the two programming environments Processing and Arduino [Har+10]. However, this is mainly to suggest solutions to error messages and does not explicitly display the possibilities of the environment and language to the user, as Suggested Blocks would.

Examining Fig. 2-3, we can see that even though the suggestion may fix the error message itself, it might perpetuate bad programming habits. Assuming that the user tried to set the entire array to 0, and not just the first 200 entries, a conditional statement is not needed. Instead, the suggestion should have changed `i<200` to `i<myArray.length`. Hence, while the exception was avoided, this is a prime example of what happens when data patterns are bad.

### 2.2.2 Looking Glass Semi-Automated Suggestions

Ichinco et al. of Washington University in St Louis are developing *Semi-Automatically Generated Code Snippets* for the BPE *Looking Glass*. Looking Glass enables its users to create animations. While allowing the user to work on an independent goal in this open-ended environment, the ITS identifies when users are struggling with a concept, and intervenes with a suggestion to get them
onto the right track. The hardship is identified by comparing the state of the project to a set of heuristics, identifying the intent behind the code, i.e., what behavior the user is trying to invoke, but not achieving. The suggestions are given as English sentences, e.g., “Make a character jump multiple times”. When a suggestion is chosen, it is accompanied by several snippets of code, as well as the animations they generate. The user can then return to the workplace, and choose whether to incorporate the suggestion into their code [Ich17].

When the system was tested in a small user study with novices, within the first 50 minutes of coding, 80% of users were given at least five suggestions. From these, they incorporated, on average, 0.8 new concepts into their own code [IK18].

2.2.3 CommunityCommands

In AutoCAD [Aut], a text-based design and programming environment, the user types text commands into an interpreter, where they are translated into a graphic representation. The commands are finite, but many of them are more or less unused. Instead, users were taking detours around commands that they did not know, spending unnecessary time on simple tasks. CommunityCommands was built as a recommendation system, aiming to diversify the commands that users use, and thereby make programming more streamlined and effective, a goal it has shown to be largely successful at [Mat+09] [Li+11]. The authors, Matejka et al., suggest that it be implemented in a UI where the
users are able to toggle the suggestions. Successful suggestions are defined to be novel and useful, where novelty is defined as the user not having used that command before, and usefulness is defined in terms of relevance to the current project.

It is implemented using Collaborative-Filtering [SM95], an algorithm that finds the nearest neighbor of a user, and recommends to the user items their neighbors enjoyed, i.e., ranked highly, but the user has not tried. But, instead of incorporating rankings, CommunityCommands simply suggest commands that other, similar users have tried that our user has yet to [Mat+09].

A key difference from App Inventor is that AutoCAD does not provide drawers of commands. While part of the educational goal of Suggested Blocks would be to provide users with some new blocks, non-novel blocks would not be unnecessary suggestions. They would just bring relevant blocks to the forefront, and the users would not have to spend time searching for it, adding to the time not spent on coding.

2.2.4 iSnap

In a classroom setting, students often work on the same problem, approaching similar solutions. Holding help room and office hours, instructors help students reach different solutions based on what path they have chosen to solve the problem. Automating the process of providing such feedback, Price et al. have developed the extension iSnap for the blocks programming environment Snap! [PDL17]. It is data-driven and works by comparing the tree structure of the current student program to a set of existing solutions. On request from the user, the algorithm uses the edit distance to the closest solution in the set and suggests a sequence of edits that will bring the user closer to that solution (Fig. 2-4). The hints then update continuously as the user makes the changes toward it. The user can disable hints anytime. In general, users were willing to follow the hints, especially as they approached the solutions. Only 2 of the 32 participants used all hints given to them, which indicates that the users still spent a significant amount of time working independently and reasoning their way to the solution. Price brings up the key limitation to data-driven systems: they do not communicate why a suggestion was given to the user. The example he uses was that users, although hinted not to, used conditionals with trivially true statements. This speaks of a need for more explanation, and perhaps that users do not automatically trust hints.

iSnap works when students are given an assignment, and there are clear solutions that can be defined. My recommendation system wishes to aid in open-ended creation and not just class work. A similar “find-closest solution” algorithm is therefore not feasible.
Figure 2-4: When a student clicks the “hint bubble” above, the below window opens to suggest an added block. Image taken from [PDL17].

### 2.2.5 Hour of Code Hint Generation

Not unlike iSnap, using a data-driven approach Piech et al. at Stanford University have developed a hint-generating system for solving problems given during Hour of Code [Pie+15a] [Hou]. They not only had access to final versions of solutions but also recorded intermediate snapshots every time the students tested their code. However, instead of basing the suggestion on the difference between the current state of the project, a so-called *partial solution*, and the final solutions, the authors developed a notion of a *desirable path* to the solution. They state that students who are less likely to program like experts had the partial solutions that were furthest from the desirable path, while learners who were able to finish the programs never strayed to far from it. Therefore, they were able to draw the conclusion that uncommon behaviors represent mistakes. Extrapolating this
to Suggested Blocks tells me that while I will not be able to tell the user to remove a block from their project, judging correctness and function of a given project might partially follow under this assumption. This is, of course, complicated by all users not striving toward the same final solution.

### 2.3 Deep Learning On Computer Programs

#### 2.3.1 Precondition and Postcondition

*Knowledge Tracing* (KT) is the attempt to model and predict future success on tasks based on previous performance. It often acts as a foundation for ITSs, or to track progress in MOOCs or similar environments. Similar to any foundation for an ITS system, they can be either hard-coded or data-driven, and both approaches have promise. However, limitations of KT include a binary classification of understanding and of success, which could be alleviated by student descriptions instead of pure annotations. For the data-driven approaches, while Bayesian probability models have shown great success at these tasks [YKG13], recent studies using Recurrent Neural Networks (RNNs) have been able to outperform earlier models [Pie+15b], and shown success with embeddings of blocks programs [Wan+17b][Wan+17a]. This has paved the way for Deep Knowledge Tracing to become the new state of the art. RNNs are well known for their ability to do prediction tasks based on time-series data [Cho17], working on all datasets that can be vectorized. However, since App Inventor does not automatically store sequential data, for my project, I am unable to use RNNs in this manner. Notable is that they are using a neural network to preprocess the written code. They train this network by comparing the input state of the program, *precondition*, to the corresponding output, *postcondition*, effectively capturing the behavior of the program instead of coding style [Pie+15c].

#### 2.3.2 Tree-Based Convolutional Neural Networks

Instead of investigating pre- and postconditions of a program, Mou et al. focused their efforts on using a neural network to examine the structure of the programs [Mou+16]. They came up with the, at the time, novel idea of a Tree-Based Convolutional Neural Network (TBCNN) to leverage the Abstract Syntax Tree (AST) structure of programming languages in detecting poor programming styles and bugs (Fig. 2-5). Basically, the Convolutional Neural Network (CNN), *slides* over each node and its children, and feeds the information further into the network until it reaches the fully connected layers, and finally, the output layer, see Sec. 4.2 for a closer description of a CNN. CNNs
are frequently used in image analysis [Cho17] and capture local patterns in the input. By encoding the AST of computer programs into a format that was readable by a CNN, they were able to detect the use of Bubble Sort with an 89% accuracy, compared to the previous state-of-the-art bag-of-trees SVM model that got a 77% accuracy. For classification of functionality, the TBCNN obtained 94% accuracy, up 5% compared to the previous state-of-the-art bag-of-trees neural network model. The benefit of this lies, at least partly, therein that TBCNNs require no manual extraction of the useful tree-structures, as opposed to an n-gram model. Rather, these are detected automatically and given the appropriate power to minimize the loss.

![Figure 2-5: Illustration of how Mou et al. constructed their tree based convolutional neural network classifier to predict behavior of computer programs using their abstract syntax trees. Image form [Mou+16].](image)

Another advanced deep learning model, the Recursive Neural Network (RNN), is usually more discussed when it comes to using tree structures for prediction [Chi09]. However, due to the relatively rigid rules of programming languages [Mou+16], and that RNNs require more computing power than CNNs [Chi09], CNNs are likely still better suited for doing predictions based on programs.
Chapter 3

The Dataset: App Inventor

Programs from Prolific Users

The dataset was collected on March 16th, 2016 and consists of all projects from the 46,320 users who had 20 or more projects at the time, so-called prolific users [LTM17]. We have no identifying information in these projects other than which projects belonged to the same user. The users have all been anonymized, and are indexed using a random enumeration. In total, there are 1,546,056 projects. The mean number of projects per user is 33.4, while the maximum is 634 (see Tab. 3.1).

<table>
<thead>
<tr>
<th></th>
<th>Total</th>
<th>Per User</th>
<th>Per Project</th>
<th>Per Screen</th>
<th>Per Handler</th>
</tr>
</thead>
<tbody>
<tr>
<td>Users</td>
<td>46,320</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>Projects</td>
<td>1,546,056</td>
<td>33.4</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>Screens</td>
<td>2,075,398</td>
<td>44.8</td>
<td>1.3</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>Handlers</td>
<td>9,095,318</td>
<td>196.4</td>
<td>5.9</td>
<td>4.4</td>
<td>n/a</td>
</tr>
<tr>
<td>Components</td>
<td>17,414,131</td>
<td>376.0</td>
<td>11.3</td>
<td>8.4</td>
<td>n/a</td>
</tr>
<tr>
<td>Blocks</td>
<td>64,896,526</td>
<td>1401.0</td>
<td>42.0</td>
<td>31.3</td>
<td>7.1</td>
</tr>
</tbody>
</table>

Table 3.1: Total counts, as well as intersection of mean values of Users, Projects, Screens, Handlers, Components and Blocks.

The projects are stored as .aia files, which in turn contain one property file, as well as two files for each screen. One contains XML representations of the blocks in the screen, and the other contains a JSON representation of the components. Note that since data was collected as a snapshot in time, we can assume that many projects were unfinished.

As opposed to choosing projects from random users, by choosing projects from prolific users I hope that their extensive use of App Inventor would have a correlation with stronger programming skills. Li et al. showed in 2017 that this set of prolific users had a slightly higher tendency to declare
and call procedures compared to a set of random users [LTM17]. Potentially, this would reduce the bad programming patterns users might perpetuate as a group, and raise the chances that a network trained using this data provides helpful suggestions. If the users are also more experienced, they might exhibit more uniform programming habits, as explained by Piech et al. [Pie+15a]. However, Li et al. also showed that the prolific dataset contained more tutorials and group exercises than a dataset of randomly selected [LTM17]. This means that a lower percentage of the projects are truly open-ended, which might skew the recommendations toward a subset of more frequent programming patterns.

The top structure of App Inventor programs are the screens. Screens in App Inventor can be more or less considered independent of each other, except when using components like databases. Each screen has its own set of components and blocks independent from the other screen(s).

Apps built in App Inventor are reactive to outside stimuli, and therefore, programs are structured into handlers. Handlers are function calls that are triggered when the user interacts with the interface, e.g., when a Button is clicked. Handlers are parent blocks of all their child blocks, the body of the handler. We can see in Fig. 3-1 that most screens have rather few handlers. Non-handler blocks that are not connected to a handler are called orphan blocks, and are not considered at all when building the training data, as they do not impact the behavior of the program. They are, effectively, comments.

![Figure 3-1: Histogram over the distribution of number of handlers per screen. Rightmost bin is inclusive of all values ≥ 20.](image-url)
The number of blocks and number of components per screen both follow a pattern similar to the number of handlers. Most projects contain rather few blocks and components, but there is a tail that reaches quite far, especially with blocks, as seen in Fig. 3-2 and Fig. 3-3.

Although App Inventor has an, in practice, unlimited number of blocks and components, because users are able to submit extensions [Aie], this data only contains 1936 different block-types and 105 component-types. I have chosen the block-types to be inclusive of the component-type a generic block refers to. As a result, `Label.SetWidth` and `Button.SetWidth` count as two separate block-types. However, if there are two buttons, a block referring to `Button1` is not distinguished from one referring to `Button2`.

It is worth noting that some blocks are used a lot more frequently than others. This is because certain blocks have a more versatile use, such as numbers (`math_number`), text fields (`text`), and blocks used to get and set variables (`lexical_variable_get`, `lexical_variable_set`), but also because users expect certain behaviors, e.g., clicking a button would trigger a certain behavior (`Button.Click`). An example of an uncommon block might be a `BluetoothClient.HighByteFirst`, that perhaps only a subset of the advanced users to deploy. The most common block, `text`, accounts for 13.6% of all blocks. The top-3 and top-5 most common blocks account for 33.6% and 42.8% of all blocks, respectively (Tab. 3.2). This gives me baselines for what accuracy a model would have to produce to be able to consider successful, since a model that always returns the most common block.
blocks would have these accuracies. However, this also tells me that the problem has an uneven class distribution. The more common blocks will be trained using more examples, and getting a model that distinguishes between them will be a lot easier than getting the model that does well on the less common blocks. And the uncommon blocks are many, almost one-third of all blocks occur less than ten times over the entire dataset.

Although the baseline of top-k blocks leaves a lot of room for improvement, we are left with consequences, or possibilities, of this class imbalance. As seen in Tab. 3.3, a block suggestion system would be able to obtain an accuracy above 85% by only ever suggesting the top-74 block-types, and looking at Tab. 3.2, we only ever need to choose from the top-8 to get above 50% – but we would never suggest a single boolean operator, which are integral to many projects. Clearly, in order to claim that a system is educational, we need to make sure that it suggests a wider distribution of blocks.

Upon manual inspection of the projects of a subset of users in the test dataset, it was noted that many projects were incomplete, in that they had empty sockets, or did not have any sensible behavior. This was far more frequently occurring than one project per user among the users sampled for investigation. While version control, or group projects, are approaches that drive progress while often leaving these incomplete files behind, in some cases it appears these projects have simply been
<table>
<thead>
<tr>
<th>Rank</th>
<th>Name</th>
<th>Count</th>
<th>Percentage</th>
<th>Cum. Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>text</td>
<td>8,820,095</td>
<td>13.6%</td>
<td>13.6%</td>
</tr>
<tr>
<td>2</td>
<td>lexical_variable_get</td>
<td>6,759,716</td>
<td>10.4%</td>
<td>24.0%</td>
</tr>
<tr>
<td>3</td>
<td>math_number</td>
<td>6,223,542</td>
<td>9.6%</td>
<td>33.6%</td>
</tr>
<tr>
<td>4</td>
<td>Button.Click</td>
<td>3,727,515</td>
<td>5.7%</td>
<td>39.3%</td>
</tr>
<tr>
<td>5</td>
<td>Label.SetText</td>
<td>2,253,750</td>
<td>3.5%</td>
<td>42.8%</td>
</tr>
<tr>
<td>6</td>
<td>lexical_variable_set</td>
<td>2,045,783</td>
<td>3.2%</td>
<td>46.0%</td>
</tr>
<tr>
<td>7</td>
<td>global_declaration</td>
<td>1,777,802</td>
<td>2.7%</td>
<td>48.7%</td>
</tr>
<tr>
<td>8</td>
<td>controls_if</td>
<td>1,653,636</td>
<td>2.5%</td>
<td>51.3%</td>
</tr>
<tr>
<td>9</td>
<td>TextBox.GetText</td>
<td>1,449,108</td>
<td>2.2%</td>
<td>53.5%</td>
</tr>
<tr>
<td>10</td>
<td>logic_boolean</td>
<td>1,322,909</td>
<td>2.0%</td>
<td>55.5%</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>64,896,526</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.2: Most common blocks, counts and percentages.

<table>
<thead>
<tr>
<th>Block Count</th>
<th>Types</th>
<th>% of Types</th>
<th>Blocks</th>
<th>% of Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>More than 1,000,000</td>
<td>13</td>
<td>0.7%</td>
<td>39,292,373</td>
<td>60.5%</td>
</tr>
<tr>
<td>Between 999,999 and 100,000</td>
<td>61</td>
<td>3.2%</td>
<td>16,089,922</td>
<td>24.8%</td>
</tr>
<tr>
<td>Between 99,999 and 10,000</td>
<td>222</td>
<td>11.5%</td>
<td>8,100,092</td>
<td>12.5%</td>
</tr>
<tr>
<td>Between 9,999 and 1,000</td>
<td>311</td>
<td>16.1%</td>
<td>1,261,442</td>
<td>1.9%</td>
</tr>
<tr>
<td>Between 999 and 100</td>
<td>353</td>
<td>18.2%</td>
<td>137,183</td>
<td>0.2%</td>
</tr>
<tr>
<td>Between 99 and 10</td>
<td>370</td>
<td>19.1%</td>
<td>13,461</td>
<td>0.02%</td>
</tr>
<tr>
<td>Less than 10</td>
<td>606</td>
<td>31.3%</td>
<td>2,053</td>
<td>0.00003%</td>
</tr>
</tbody>
</table>

Table 3.3: Frequency distribution of block-types by number of occurrences in the dataset. Note that almost one third of the block-types appear less than 10 times out of 64,896,526 blocks total.
abandoned by their creators. Therefore, when building machine learning models trying to predict which block to add next, we need to give some consideration to the fact that the training data already contains missing elements.

In addition to the project data, my advisor had access to an in-environment survey users were asked to fill out between Dec, 2013 and Aug, 2016, that asked about demographic information, including previous programming experience and motivation behind using the environment. With help from the App Inventor team, we were able to map these to users in our dataset. I was given access to anonymized responses, that allowed me to build a more holistic picture of the types of users I had in my dataset.
Chapter 4

Representing the Data for Machine Learning

Neural networks are a type of classifier. There are many different kinds of neural networks, and the simplest version is the Deep Neural Network (DNN), also known as a Fully Connected Neural Network (FCNN). Given an input vector of a certain length \( a \), a neural network multiplies the input through a set of 2D vectors and outputs a 1D vector of appropriate length. This output length could be either the number of different classes to be predicted or one if the problem is a binary classification or regression. A binary classifier would project this one number onto a range between 0 and 1, and return the closest integer, while a regression would return the number.

In simplified terms, an already trained DNNs consists of a sequence of matrices with activation functions, such that when the input is multiplied through the sequence, the result is the output. The activation functions are needed, because otherwise, the matrices would be equivalent to a “collapsed” matrix, i.e., multiplying their product with our input would give the same result as doing it sequentially, which would only be a linear classifier. The most common activation function for intermediate layers, which is also used in this paper, is the Rectified Linear Unit (ReLU), defined as \( y = \max(0, x) \) \[\text{Cho17}\]. Despite the simplicity of ReLU, DNNs with as few as one hidden layer, i.e., two matrices between input and output, are universal function approximators \[\text{HSW89}\]. For the output layer, instead of ReLU, softmax is used, which normalizes the output values to be between 0 and 1, where the sum of all values is 1. This allows the output to be interpreted as probabilities.

However, neural networks for any problem in the world do not grow on trees, and in order for the models to be of use, we need to train them to do what we want them to do. A network is randomly
initialized with small floating point numbers as weights for all entries of each matrix. Then, in simple terms, input vectors are fed to the network, and their predictions are compared to annotations (ground truth data). If the predictions do not match the annotations, the network reconfigures itself by changing its weights, i.e., the values of its matrix entries. This process of changing the weights is called backpropagation, and is carried out by a specified optimizer. Backpropagation is a gradient descent problem aiming to minimize a loss function as provided to the neural network by the programmer. The loss function has been chosen to reflect the problem at hand, e.g., a classification problem has a different loss function from a regression. An appropriate loss function for a classification problem would be a cross-entropy function, while one might choose mean-squared error for a regression problem. As the loss decreases, i.e., the better the network is trained, performance generally increases. In our case, performance means accuracy, which is defined as the percentage of correct predictions on the data being measured, over the total number of predictions [Cho17].

An example of a more advanced model is a Convolutional Neural Network (CNN). CNNs are notable for their contribution to the advancements of computer vision, and often use images as inputs. Compared to DNNs, instead of consisting of a simple chain of matrices and activation functions, CNNs are simplified as using a “sliding window”, where a smaller matrix slides over the input. As opposed a fully connected layer, one large matrix, this sliding window is called a convolutional layer. Each of the results from the window is pooled using, e.g., a max or average function, and the results thereof are either put through another convolutional layer or a fully connected layer. This captures relative patterns in the data since, at each convolutional layer, the result is based on the values with relative proximity to each other. The final layer, or layers depending on chosen architecture, are fully connected. Similar to DNNs, CNNs are trained with backpropagation.

Overfitting is the problem of learning the peculiarities of a dataset rather than overarching patterns that would generalize to new examples. Overfitting is detected when performance metrics are significantly higher on the training dataset, compared to any other dataset. There are a few ways to combat overfitting. First of all, we train the network on a large, randomly selected training set. We also determine how many times, epochs, the network should iterate over its input by tracking the accuracy of a validation set, and stop training when the accuracy of the validation set is no longer improving. When the model sees the validation set, it does not train, it just predicts. To avoid overfitting to our validation data as well, we hold back a third data set, the test set. While overfitting to the validation data would mostly affect how many epochs we train the model for, this might have slight implication on the final result of the model. Therefore, we hold out the test data just make sure that our final accuracy value is independent of any information we used to train and
Figure 4-1: Example of the results from feeding a partially completed project to a trained neural network. The example was constructed by taking the tutorial HelloPurr and removing the handler block Button.Click. The returned top-5 blocks can be found in Tab. 4.1. This example was generated by using the data representation described in Sec. 4.1.1.

<table>
<thead>
<tr>
<th>Rank</th>
<th>Block Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Button.Click</td>
<td>0.991</td>
</tr>
<tr>
<td>2</td>
<td>Button.LongClick</td>
<td>0.002</td>
</tr>
<tr>
<td>3</td>
<td>Button.TouchDown</td>
<td>0.002</td>
</tr>
<tr>
<td>4</td>
<td>Form.BackPressed</td>
<td>0.001</td>
</tr>
<tr>
<td>5</td>
<td>Button.LostFocus</td>
<td>0.001</td>
</tr>
</tbody>
</table>

Table 4.1: The most common blocks from the example in Fig. 4-1. Here, based on the values, it is clear that Button.Click would be the most appropriate suggestion. Less obvious examples would have less distinct values.

Aside from detecting overfitting, there are ways to regularize the model and combat overfitting before it occurs, and allow for the models to be trained longer. One way of doing this is to include drop-out layers after each, or some, activation function(s). When training, these layers randomly drop out, i.e., set to 0, a subset of the nodes in the hidden layers (the exact number can be specified when compiling the model). At test or validation time, nothing is dropped.

As an example, Fig. 4-1 and Tab. 4.1 display the behavior we desire when training the models. When given a project, the model creates an array where each index corresponds to a block-type. It then returns the blocks that correspond to the top-k maximum value indices.
4.1 Bag-of-Words Representations

The simplicity of the DNN puts a lot of restrictions on how we are able to feed our data into the models. As mentioned earlier in this chapter, DNNs require a fixed-width one-dimensional input vector. Further, because we never compare values to each other, or process them in smaller groups like in a CNN, the model is not able to pay attention to relative ordering of input. Instead, it only finds global patterns. In order to make this work, bag-of-words has become a standard way of representing complex data in vector format [Skl]. In my case, bag-of-words means that an index of the input vector represents the count of one of my selected features. I have chosen to include both components and blocks in the input vector. However, while DNNs restrict me to bag-of-words approaches, it is not clear how I should structure my data and annotations to get the best model possible for a suggestion system. This section will investigate and focus on different sampling methods for the annotations that I will use to train the networks.

4.1.1 Fully Uniform Sampling

The data and annotations for Fully Uniform Sampling are created by choosing one block at a time and using the components and non-selected blocks to predict the selected block (Fig. 4-2). This way, not only can a value be predicted for an empty socket, but a parent block could also be predicted by its orphaned children, as seen in the second example of Fig. 4-2. Note that this approach makes no assumptions about the order in which the programmer inserted the blocks into the workspace. Rather, the annotations are uniformly selected from the screen. The bag-of-words approaches used here give an input dimensionality of 2044 and an output dimensionality of 1936, where 1936 is the number of block-types, and 108 is the number of component-types.

4.1.2 Depth First Search Sampling

As opposed to the uniform selection, if we are able to assume that users tend to add blocks in a certain order, we could leverage that and perhaps get more relevant predictions. For example, assuming that programs are built in a top-down manner in which the parent blocks are added to a workspace before children-blocks, handlers might be more likely to be suggested earlier in a project, and “leaf blocks”, having a plug and no sockets, would be suggested later. Since I do not have our data annotated with the order of insertion, I infer the order based on a depth-first search (DFS) tree traversal.

The blocks would be predicted without replacement. As opposed to predicting with replacement
(uniform representation), I am using the components and all the preceding elements in a DFS traversal to predict the next element. In other words, in the example from Fig. 4-3, the components would predict `Button.Click`, those four elements would predict `Sound.Vibrate`, and so on. Where there are multiple handlers, handlers will be added sequentially to the prediction based on their placement in the workspace, sorted first top-down, and then left-to-right.

However, it is a strong assumption that users would program in this manner. And even if so, studies have shown that a top-down programming style is preferred by men, and creating a recommendation system on this basis could have consequences for users who prefer a non-linear style [BB04].

Figure 4-2: Example of how a small project is broken down into input and output annotations.

Figure 4-3: Ordering of blocks in DFS traversals.
4.1.3 Breadth First Search Sampling

While DFS sampling makes a lot of assumptions that we are unable to verify, there might be aspects of top-down ordering that are desirable compared to a strictly uniform sampling. Being able to predict parent blocks before their children could give some top-down function to the predictions. Therefore, I also try breadth-first traversal, BFS, where levels are defined as in Fig. 4-5, which represents the same handler as in Fig. 4-4. In order to not distinguish between different blocks at the same level, I do a uniform sampling at each level as follows: For each handler, I remove the entire handler and let all other handlers and components predict the parent block. Then, I add the parent to the input. I also add all “first-generation” children to the parent, and one by one, with replacement, remove one of the children and let all other blocks predict the removed block. I then do this, level by level, including all parent and grandparent blocks in the input to predicting each child, until no more levels remain. Then, we remove another handler and repeat until all blocks have been predicted. This is meant to mitigate a relative ordering of handlers and sibling-blocks, while still maintaining top-down programming.

![Figure 4-4: Example of handler in a user project.](image)

4.1.4 Bigrams

Also referred to as *Bag-of-Trees*-representation when applied to tree structures [Mou+16], n-gram representations capture part of the structure and order of the input. They are constructed the same way as bag-of-words (also called “one-gram” or “unigram” representations), but instead of counting occurrences of single words, they count occurrences of pairs of words. For example, in a project with three blocks, the handler `Button.Click`, which triggers a `Sound.Vibrate` with the value `math.number`, a unigram would have three counts corresponding to each of the blocks. A two-gram (bigram) could also count the connections `Button.Click|Sound.Vibrate` and
Sound.Vibrate|math_number. There are different ways of defining what a pair should be, and I experiment with two of them. One is bigram based on adjacency, analogous to how bigrams are used in Natural Language Processing, and the other is bigrams based on which handler a block is a part of. Of course, including bigrams increases the number of possible features to give the network by a polynomial factor.

Note that my bigram representations also include individual blocks, as these are needed to capture common, and significant, events like inserting a single handler without connecting it to some other block.

**Adjacency-bigrams**

Considering adjacency, we define each bigram as the presence of a block being directly connected to another. The small three-block project as described above would be an example of an adjacency-bigram representation. Another can been seen in Fig. 4-6.

In total, there are 69,014 different adjacency-bigrams in the dataset. They are distributed with a long tail - while almost half of the bigrams are of the most common 90 bigram types, 60% of the types occur less than 10 times across the entire dataset, accounting for 0.2% of the total count (Table 4.3). This is similar to the distribution of block-types. However, as it is not computationally feasible to keep a sparse input vector of almost 70,000, I am restricting my experiments to include the 3890 most common bigram, components, and unigrams. This amounts to 108 components, 3,321 bigrams, and 659 unigrams. I use uniform sampling as described in section Sec. 4.1.1. The only difference is
that as opposed to only using components and unigrams, I use components, unigrams, and bigrams.

A list of the most common adjacency-bigrams can be found in Tab. 4.2, and a histogram of the distribution and counts in Tab. 4.3.

<table>
<thead>
<tr>
<th>Rank</th>
<th>Name</th>
<th>Count</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>text_join</td>
<td>text</td>
<td>1,612,164</td>
</tr>
<tr>
<td>2</td>
<td>lists_create_with</td>
<td>text</td>
<td>1,573,856</td>
</tr>
<tr>
<td>3</td>
<td>controls_if</td>
<td>math_compare</td>
<td>883,098</td>
</tr>
<tr>
<td>4</td>
<td>global_declaration</td>
<td>math_number</td>
<td>843,125</td>
</tr>
<tr>
<td>5</td>
<td>lists_select_item</td>
<td>lexical_variable_get</td>
<td>745,313</td>
</tr>
<tr>
<td>6</td>
<td>math_random_int</td>
<td>math_number</td>
<td>733,461</td>
</tr>
<tr>
<td>7</td>
<td>math_compare</td>
<td>math_number</td>
<td>727,029</td>
</tr>
<tr>
<td>8</td>
<td>math_compare</td>
<td>lexical_variable_get</td>
<td>726,037</td>
</tr>
<tr>
<td>9</td>
<td>LabelSetText</td>
<td>text</td>
<td>689,010</td>
</tr>
<tr>
<td>10</td>
<td>controls_openAnotherScreen</td>
<td>text</td>
<td>666,187</td>
</tr>
</tbody>
</table>

Table 4.2: Ten most common adjacency-bigrams, counts and percentages.

Figure 4-6: Same handler as in Fig. 4-4, divided into all possible bigrams.

**Handler-bigrams**

Because of the nature of App Inventor, the function of the programs is hugely dependent on which sequence of blocks is triggered by each event. In addition to the adjacency-bigrams, where I consider the direct adjacency of blocks, I also create a representation based on *handler-bigrams*, where each block is considered in relation to the event that triggered it, instead of just on its own. The example with the three blocks, containing the handler *Button.Click*, which triggers a
Table 4.3: Frequency distribution of adjacency-bigram types by number of occurrences in the dataset.

<table>
<thead>
<tr>
<th>Bigram Count</th>
<th>Types</th>
<th>% of Types</th>
<th>Bigrams</th>
<th>% of Bigrams</th>
</tr>
</thead>
<tbody>
<tr>
<td>More than 100,000</td>
<td>90</td>
<td>0.1%</td>
<td>27,827,292</td>
<td>49.6%</td>
</tr>
<tr>
<td>Between 99,999 and 10,000</td>
<td>614</td>
<td>0.9%</td>
<td>17,470,825</td>
<td>31.1%</td>
</tr>
<tr>
<td>Between 9,999 and 1,000</td>
<td>2,474</td>
<td>3.6%</td>
<td>7,922,543</td>
<td>14.1%</td>
</tr>
<tr>
<td>Between 999 and 100</td>
<td>6,814</td>
<td>9.9%</td>
<td>2,197,034</td>
<td>3.9%</td>
</tr>
<tr>
<td>Between 99 and 10</td>
<td>17,079</td>
<td>24.7%</td>
<td>559,607</td>
<td>1.0%</td>
</tr>
<tr>
<td>Less than 10</td>
<td>41,943</td>
<td>60.8%</td>
<td>117,310</td>
<td>0.2%</td>
</tr>
</tbody>
</table>

Table 4.4: Ten most common handler-bigrams, counts and percentages.

<table>
<thead>
<tr>
<th>Rank</th>
<th>Name</th>
<th>Count</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Button.Click</td>
<td>text</td>
<td>4,011,277</td>
</tr>
<tr>
<td>2</td>
<td>Button.Click</td>
<td>lexical_variable.get</td>
<td>2,146,817</td>
</tr>
<tr>
<td>3</td>
<td>Button.Click</td>
<td>math_number</td>
<td>2,141,937</td>
</tr>
<tr>
<td>4</td>
<td>global_declaratiion</td>
<td>text</td>
<td>1,622,440</td>
</tr>
<tr>
<td>5</td>
<td>Button.Click</td>
<td>TextBox.GetText</td>
<td>1,276,044</td>
</tr>
<tr>
<td>6</td>
<td>procedures_defnoreturnd</td>
<td>lexical_variable.get</td>
<td>1,187,436</td>
</tr>
<tr>
<td>7</td>
<td>Button.Click</td>
<td>Label.SetText</td>
<td>1,149,761</td>
</tr>
<tr>
<td>8</td>
<td>global_declaratiion</td>
<td>math_number</td>
<td>987,798</td>
</tr>
<tr>
<td>9</td>
<td>Button.Click</td>
<td>lexical_variable.set</td>
<td>964,024</td>
</tr>
<tr>
<td>10</td>
<td>procedures_defnoreturnd</td>
<td>math_number</td>
<td>765,931</td>
</tr>
</tbody>
</table>

Sound.Vibrate with the value math_number, would give the bigrams Button.Click|SoundVibrate and Button.Click|math_number. In total, there are 33,243 different handler-bigrams present in the dataset. I restrict the input dimensionality the same way I restricted the dimensionality of adjacency-bigrams. Like for the adjacency-bigrams, a list of the most common adjacency-bigrams can be found in Tab. 4.4, and a histogram of the distribution and counts in Tab. 4.5.

4.2 Advanced Machine Learning Data Representations

As discussed briefly at the end of Sec. 2.3 and at the beginning of this chapter, there are multiple more advanced machine learning models that are able to consider not only human-identified features but
also the structure and context of its input data. In order to fully leverage the power of neural networks, I constructed a *multimodal* version of the TBCNN suggested by Mou et al. in 2016 [Mou+16]. This aims to capture the tree structure, but instead of looking at connections the way bigram models do it, it considers the entire tree at once. Here, I use a fully uniform sampling, removing and predicting one block at a time, with replacement.

### 4.2.1 Multimodality

Multimodality means either a multi-input or multi-output (multi-headed) model. Multimodal models are able to assume multiple sources of input, which are able to have multiple shapes and take multiple paths through the model. For my purposes, if I am to preserve the tree structure of the blocks, I need a way to incorporate components into the data representation in order to maintain as much relevant information as possible into my prediction, like visualized in Fig. 4-7. However, if I am not to look at the relative placement of components in the interface, i.e., the design, ordering of components would have less impact on block prediction.

![Figure 4-7: Module configuration of the AICNN.](image)

### 4.2.2 Representing The Tree

App Inventor stores block programs as XML trees. A visualization of parent-child relationships in these trees can be seen in Fig. 4-4 and Fig. 4-5. While Mou et al. rewrote a convolutional kernel to work directly on the tree structure [Mou+16], I have chosen a different approach, where I instead encode the tree into a format readable by standard CNNs. As CNNs are commonly used on data with dimensions of fixed lengths, I am able to write the program into a three-dimensional space,
where the plane represents the placement of each block in the tree, the *structure embedding*. Each position in the plane contains a vector representing the block. When encoding these relationships onto a plane grid, I let each column correspond to the depth of the block in the tree and I let each row correspond to the path to a leaf. Note that for each new handler is placed on level 0.

**Embedding the structure**

All encodings of structure, need to be onto a grid of the same size, no matter how large or small each project is. This creates a problem, as the deepest project has 30 levels, and the “leafiest” project has 236 leaves. This would already give us a dimensionality of 7080, without considering the encoding of the blocks. This makes it unfeasible, and we have to prune the trees or the dataset. Therefore, I remove all screens from the dataset where the contents exceed the fixed dimensions of the plane, $10 \times 30$, where 10 is the highest number of levels, and 30 is the highest number of leaves. This affects 9.7% of projects, 1 component-type, and 823 block-types. As one might argue that only predicting using partial projects might skew the data, these have been left out of the training, test, and validation sets completely. Perhaps also, for educational purposes, it might be more desirable for projects of that size to be suggested to delete a few blocks or use abstraction, instead of more blocks. The embedding of the handler seen in Fig. 4-4 can be found in Fig. 4-8.

![Figure 4-8](image)

Figure 4-8: Spacial encoding of Fig. 4-4 onto 2-dimensional grid. Each row corresponds to a leaf, and its path, and each column corresponds to a level of the original tree.

**Embedding the types**

All neural networks need numeric input, so we need to change the string representation of the blocks into vector representations. In other words, each string in Fig. 4-8 needs to be converted into a vector to be given to the models. The naive way to do this is to take their *one-hot* representations, where each of the blocks is mapped to an index of a fixed-size vector. One-hot is a representation where a vector is all zeroes, except for the index corresponding to the type, which is one [Cho17]. When choosing how to encode the third axis of the location of each block on the grid, we could use one-hot embeddings. Since this vector would have to be 1936 long, it would scale the already large
space by a lot, which is not feasible. Therefore, we need to fit information about the block into a smaller space, by dimensionality reduction.

A modern algorithm that allows for that is Word2Vec [Mik+13] [RS10]. Word2Vec is a common tool that is used by a range of different entities, from music recommendations [Spo] to biology [Ng17]. It works by calculating the likelihood of each, in this case, block-type to show up next to all other blocks-types, and using one-hot representations of size 1936, trains a neural network with one hidden layer of size $h$, and an output layer of size 1936. The output used for training is the vector of the probabilities of words being adjacent to the input word. In the end, the embedding for each word is obtained by multiplying the input vector with just the hidden layer. This produces a vector of length $h$. Similar blocks have similar vectors.

I create this block to embedding mapping with $h = 5$, and for each block in the grid mention above, I replace the text with the corresponding vector and the empty slots with the 0 vector. Now, I have a $10 \times 30 \times 5$ three-dimensional matrix, that I feed to the convolutional module.

### 4.2.3 The Convolutional Module

Without going into too much detail of the inner workings of CNNs, the overarching ideas give the intuition behind why they have managed to become so powerful. While fully connected networks detect patterns on a global level when making their predictions, CNNs, because of the convolution and pooling, detect local patterns and bring these to the forefront when making the predictions [Cho17]. Convolution can, as stated in Sec. 2.3.2, be thought of as sliding a window across the grid, using the weights of the window to draw insight about what is seen through the window. Pooling then consists of sliding a window across the convolutions, choosing one convolution per window by either a maximum, or average. This serves to preserve the most desired local patterns in the following layers. This is repeated so that the patterns detected become more and more complex, and finally, the throughput is flattened, and fed into a fully connected layer which is able to output the prediction. Just like a DNN, CNNs use activation functions to achieve non-linearity and are trained through backpropagation.

So instead of deciding by myself which features are most important, e.g., a certain n-gram or count, I make the more advanced network in hope to preserve as much of the tree structure as possible, and be able to draw information from the preserved structure. It is not possible to reconstruct a program from only block counts, but in an embedding such as the one I am proposing, although interface design cannot be inferred, and the Word2Vec would have to be reversed, it would
be possible to recreate the program using the representation.
Chapter 5

Experiments

In this chapter, I will describe in-depth the technical aspects of going from App Inventor projects created by users to trained neural network models, as well as the performance of these networks in terms of predictive power.

5.1 Framework and Technologies

The first challenge I faced when starting to build the first model was finding an easy way to map the .aia files that represent the projects into the vectors I describe in Sec. 4.1.1. I needed to find all the block- and component-types present in the dataset, and map each of them to its own index in a vector. Afterward, I had to process the data to fit into the representations.

As I am an avid Python user, I used Python in my machine learning coursework, and the language is well suited for working with data processing with its extensive libraries, it was my language of choice. On my local machine, I use Python 2.7.11. For preprocessing, I mainly used the libraries scikit-learn [Ped+11], numpy [Oli06], as well as object serialization packages JSON and pickle, and os. In addition, I was lucky to already have access to an App Inventor summarizer program, that I contributed to together with my advisor Lyn Turbak, and Benjamin Xie, at the time MIT M.Eng. student, in 2015. The summarizer scrapes the .aia files into JSON dictionaries, which include overarching information about a project, as well as the counts of block- and component-types. For the more complex data representations, I was able to tweak the summarizer to also return information about the structure of the program. I use these summaries as the basis for my data processing.

With all the data easily available in summaries, the next step is to transform them into matrix format. Instead of manually creating the block to index mapping myself, I took advantage of
scikit-learn’s module CountVectorizer. CountVectorizer is normally used for natural language processing, and takes input strings consisting of words, or tokens, separated by natural language delimiters such as spaces, commas, or punctuation. I represented each project as a string of tokens (in this case, component and block names) separated by spaces and let the vectorizer transform the sentences into NumPy matrices. For exact details on all these steps, the example of how to build the input data for the uniform model can be found in appendix A.

Although the CountVectorizer has a parameter-setting that counts bigrams, because the projects have a tree structure rather than a linear one, the vectorizer would not have been able to process the bigrams in the way I intended. Therefore, I preprocessed all bigrams, representing each as a token, and when using the vectorizer, treated them as if they were unigram tokens. As mentioned in Sec. 4.1.4, I limited the dimensions of the bigrams to 3890, using CountVectorizer’s built-in max_features parameter.

For the block data of the convolutional model, instead of using a vectorizer, I preprocessed the blocks into their two-dimensional matrix grid, and then applied Word2Vec [ˇRS10] on each element in the matrix. The components were processed separately using a vectorizer.

Because there were over 64 million blocks in the dataset, each block is predicted once, and each instance consists of two vectors with a dimension of around 2000 at least (input and annotations), the data is rather large, and each process to go from summary to matrix is time-consuming. The size of each processed dataset was between 30 GB, for the uniform data, and 250 GB, for the convolutional. On my 2016 MacBook Pro with a 2.9 GHz Intel Core i5, getting the data from summary to matrix format would take up to 15 hours.

Just because of the sheer size of the data, and the computational demands of backpropagation, it proved to be infeasible to train any model locally on my computer. After having tried experimenting with truncating the data in various ways, I was able to figure out a way to train with un-truncated data on my computer, using generating functions. A generating function is a Python function that returns an iterator that allows us to access parts of the data, and not load the entire dataset into working memory. However, running 30 epochs of one uniform experience would take three months to completion, as estimated by Keras. After receiving generous funds from the Office of the Provost and the Computer Science department, I solved this problem by utilizing cloud computing with a GPU to drastically reduce processing time. Because of a large initial discount, and my personal familiarity with the platform from an internship, I chose Google Cloud Platform and its Computing Engine to carry out my experiments. I used an Nvidia Tesla P100 GPU with a 250 GB disk, although I changed the size of the disk as the content on it shrunk or expanded. This was able to bring down
the computation time to about 20 hours for the most computationally intensive experiment, the convolutional model. However, because of the costs involved with each computing hour on the cloud service, I still did the preprocessing, and all analysis and visualization on my local machine, using the packages numpy, matplotlib and pandas. Because of the time required to transfer the files between the machines, I ended up using only half of the convolutional data, 125 GB, which on its own took over 24 hours to transfer to the cloud. In total, running all 15 experiments, outlined in Tab. 5.1, ended up costing $710, not including the $300 free trial credit I received when opening my Google Cloud account.

On my new cloud machine, I installed Python 3.6.4, together with the packages mentioned above. Further, I installed TensorFlow [Aba+15], the low-level neural network framework developed at Google in order to run TensorFlow with the GPU, together with CUDA 8.0 and cuDNN, which allow TensorFlow to access the GPU accelerator. I also installed Keras, a high-level Python package that provides a layer of abstraction to TensorFlow, while maintaining a lot of flexibility and choices regarding architecture. In order to define top-$k$ accuracy for $k \neq 1$, I used the package functools.

Using generating functions to allow the machine to train on all the data without exceeding memory limits, I trained each model using the over 46,000 users project data, setting aside 1000 users as validation data, and another 1000 users as test data.

5.2 Experiments

As is common in machine learning, I did some experiments to find which model would best suit the problem I had at hand. In total, this study considers 15 different experiments, 7 of which are using the unigram model with uniform sampling, and 3 of which are using the adjacency-bigram model. An overview of these experiments can be found in Tab. 5.1.

As there is a multitude of variables that come into play when building a neural network, and they all might have an impact on the final powers of the network, it was important to reduce the number of controlled variables I was working with. Unless otherwise specified, by recommendation of Chollet, the creator of Keras, in “Deep Learning with Python” [Cho17], I only work with ReLu as activation functions (see Ch. 4), and RMSProp as optimizers (an optimizer is responsible for the update of the network), with the standard settings. Because I was working with a multiclass problem, I used categorical cross-entropy as my loss function, and soft-max on my output layer. Multiclass means that the number of possible predictions is greater than one, where I am looking to predict one out of 1936 classes, i.e., block-types. My batch-size is defined in terms of the yield from my generating
Table 5.1: High-level descriptions of the 15 experiments, by order of execution. Experiment 14 was carried out with a defect version of the intended experiment, see Sec. 5.2.1.

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Data Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Uniform</td>
</tr>
<tr>
<td>2</td>
<td>Uniform</td>
</tr>
<tr>
<td>3</td>
<td>Uniform</td>
</tr>
<tr>
<td>4</td>
<td>Uniform</td>
</tr>
<tr>
<td>5</td>
<td>Uniform</td>
</tr>
<tr>
<td>6</td>
<td>Uniform</td>
</tr>
<tr>
<td>7</td>
<td>Uniform</td>
</tr>
<tr>
<td>8</td>
<td>DFS Traversal</td>
</tr>
<tr>
<td>9</td>
<td>BFS Traversal</td>
</tr>
<tr>
<td>10</td>
<td>Adjacency-Bigram</td>
</tr>
<tr>
<td>11</td>
<td>Adjacency-Bigram</td>
</tr>
<tr>
<td>12</td>
<td>Adjacency-Bigram</td>
</tr>
<tr>
<td>13</td>
<td>Buggy Handler-Bigram</td>
</tr>
<tr>
<td>14</td>
<td>Convolutional</td>
</tr>
<tr>
<td>15</td>
<td>Handler-Bigram</td>
</tr>
</tbody>
</table>

function. This, in turn, was dependent on the size of the data.

What remains to be experimented with is (1) the number of hidden layers, (2) the size of the layers (3) whether or not to add drop-out the layers. However, regarding the size, Chollet suggests to not have any hidden layer be smaller than the output layer, or larger than the input layer, as the smallest layer would cause an information bottleneck.

5.2.1 Validation Process

Because of financial and time constraints, decisions had to be made in order determine which experiments were worthwhile. Since the structures of the data representations are in many ways similar, to some extent we are solving similar problems over and over again. The input and output dimensions will be similar, and the way the data behaves is in some regards also similar. Therefore, when noticing a pattern in earlier models, e.g., that 3 layers with size 2000 and added dropout worked best for uniform (Tab. 5.2) this configuration was given preference in later experiments.

Some data representations very quickly showed that they held very little statistical power compared to earlier experiments. In addition, exploring all configurations for the uniform data only caused a difference of 7 percent units between the best and worst models in earlier experiments. Relative to the difference in performance between models, this showed little variance. The convolutional, DFS, and BFS representations were therefore only used to train one network each. However, this turned out to be a hasty decision as BFS performed very well in later stages of the evaluation.
Finally, certain experiments required significantly more resources than others. Because of the decreased sparsity of the data for the convolutional network, and more complex architecture, it was not feasible to repeat that experiment. Further, in order to reduce the runtime of the training to only 20 hours for that model, the learning rate was doubled, from 0.001 to 0.002. Similarly, the additional complexity of Experiment 12 made the architecture of Experiment 11 a more suitable candidate for the first handler-bigram experiments. However, as the experiment had to be repeated because of the bug described in Sec. 5.2.1, and at that point, additional funding had been granted for the project, Experiment 15 was run with the architecture form Experiment 12.

Validation accuracy was recorded once per epoch, i.e., once per pass over the entire training dataset. It is calculated by making one pass over the validation data and computing the accuracy, as defined in Sec. 5.1. The maximum validation accuracy from each of the 15 experiments are described in this section in Tab. 5.2.

**Handler-Bigrams Bug**

After running Experiment 14, I discovered that the data representation did not look as intended. Instead of counting the handler Component.Action|subBlock as one unit, a bug caused the bigram to split up into Component and Action|subBlock. This caused the component count to be incremented, and also merged certain blocks, e.g., Button.Click|subBlock and Canvas.Click|subBlock both generated the feature Click|subBlock. Because the experiment showed high accuracy, the model was kept, but another experiment, Experiment 15, was launched where this bug was fixed.

**5.3 Analysis and Results**

In this section, I consider only the best performing models for each data representation, as determined by the validation accuracy. Since when developing a model, we risk overfitting to the validation data by making a cutoff on its maximum, I had set aside a test data set that is independent of any data the models have been evaluated or trained on in the experimental phase. In this section, I first define some important performance metrics, namely top-\(k\) accuracy, precision, recall, F1-score, correlation, and range, and then carry out analysis of the models on the test data, trying to see if I am able to gain some insight about the way the models work.
<table>
<thead>
<tr>
<th>Data Representation</th>
<th>Hidden Layers</th>
<th>Functions</th>
<th>Max Validation Accuracy</th>
<th>Epochs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Uniform</td>
<td>[2000, 2000, 2000]</td>
<td>[relu, relu, relu, softmax]</td>
<td>72.1%</td>
<td>50</td>
</tr>
<tr>
<td>3 Uniform</td>
<td>[2000]</td>
<td>[relu, softmax]</td>
<td>66.0%</td>
<td>35</td>
</tr>
<tr>
<td>5 Uniform</td>
<td>[1950, 1950, 1950]</td>
<td>[relu, relu, relu, softmax]</td>
<td>70.7%</td>
<td>35</td>
</tr>
<tr>
<td>6 Uniform</td>
<td>[2050, 2050, 2050]</td>
<td>[relu, relu, relu, softmax]</td>
<td>71.2%</td>
<td>35</td>
</tr>
<tr>
<td>8 DFS Traversal</td>
<td>[2000]</td>
<td>[relu, softmax]</td>
<td>33%</td>
<td>35</td>
</tr>
<tr>
<td>9 BFS Traversal</td>
<td>[2000, 2000]</td>
<td>[relu, relu, softmax]</td>
<td>45%</td>
<td>35</td>
</tr>
<tr>
<td>13 Convolutional</td>
<td>[(64, (3, 3)), (2,2), (64, (3, 3)), (2,2), (64, (3, 3)), (2,2), 1024], [100,100,100]</td>
<td>[relu, MaxPooling, relu, MaxPooling, relu, MaxPooling, relu], [relu, relu, relu], [softmax]</td>
<td>42%</td>
<td>25</td>
</tr>
</tbody>
</table>

Table 5.2: Configurations and maximum top-1 validation accuracy for all experiments, by order of execution. Each experiment corresponds to the development of one model. Models were trained until they converged, the number of epochs is recorded in the rightmost table. Example of the progression of validation accuracy for the uniform models can be seen in Fig. 5-1. Each number in the arrays of layers represent the size of a layer, and the length of the array represents the number of hidden layers. The functions represent the activation function of each layer, including the softmax function for the output layer.
Figure 5-1: Plots of the training (dots) and validation (lines) accuracies for top 1 blocks, all networks trained on uniform sampled bag of blocks. As it was clear that Experiment 1 and 2 converge at around epoch 25, subsequent experiments were only trained for 35 epochs total. Experiment 7, due to time constraints, was only trained for 25 epochs, however, also showed convergence.
5.3.1 Definitions

The following metrics, unless otherwise specified, are taken from the sci-kit learn library [Ped+11].

Accuracy

Top-k accuracy has been already been used and defined earlier in this paper as

\[
\text{Accuracy} = \frac{\# \text{Success}}{\# \text{Total}},
\]

where \# Success means the number of times the annotation of an instance was a member of the top-k blocks returned by the model, and \# Total is the total number of instances in the test data set. Accuracy is defined across the entire dataset, independent of which classes, i.e., blocks, are being predicted.

Precision, Recall, and F1-score

While accuracy operates across the entire dataset, other metrics are typically used to track performance on a class-by-class basis (i.e., block-type-by-block-type). For these purposes, we define precision, recall, and F1-score.

In relation to a single class, when making a prediction, there are four potential outcomes. Either we correctly predict an element to be of that class (true positive), we wrongly predict an element to be of that class (false positive), we wrongly predict an element not to be of that class (false negative), or we correctly predict an element not to be of that class (true negative). Fig. 5-2 visualizes these differences. For example, the large circle is what the model predicts to be a certain class, say Button.Click. Inside these, we will have both annotations that actually are Button.Click (solid circles), and those that are not (unfilled circles). These are the true positives (the Button.Clicks we predicted as such) and false positives (the other blocks that were still predicted Button.Click. Then, for all the blocks that were predicted to be anything but Button.Click, we will likely have those that, correctly, were not (true negatives), but also those that were (false negatives).

Precision and Recall are defined using these four categories to measure different aspects of performance. They are both in the range between 0 and 1, and are somewhat analogous to accuracy. Studying the following definitions,
Figure 5-2: Graph defining the notions of true/false positives/negatives. These are used in calculating explaining precision and recall. Image taken from [Wal14].

\[
\text{Precision} = \frac{tp}{tp + fp}, \quad \text{Recall} = \frac{tp}{tp + fn},
\]

(5.2)

the numerator for the two measurements are the same. However, while precision penalizes the score for each false positive, recall penalizes each false negative. In other words, precision measures, in what percentage of cases where \texttt{BlockTypeX} was predicted was \texttt{BlockTypeX} the actual result? On the other hand, recall measures, in what percentage of the cases where \texttt{BlockTypeX} was the the actual result was \texttt{BlockTypeX} predicted? This relation is further illustrated in Fig. 5-3.

However, neither precision nor recall on their own can testify to the usefulness of a classifier. If we predict 100% of the instances will be a certain class, there are no false negatives, so we are guaranteed 100% recall. If all instances in a dataset would be annotated as a particular class, but we
only correctly predict one instance as that class, there are no false positives, and precision would be 100%. Neither of these would necessarily be good classifiers. Therefore, we combine precision and recall into an F1-score, that penalizes both false positives as well as false negatives. It is defined to be

\[ F1 = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \]  

which also is a floating point number between 0 and 1. In general, a high F1-score means that the classifier works well for a class, both in terms of precision and recall.

In this project, however, we have 1936 classes. Therefore, I will also report on the combined F1-score, precision, and recall of the classes, which is a weighted average of each individual class, by block-type frequency.

**Pearson Correlation**

The Pearson correlation is a standard correlation metric between two variables, \(X\) and \(Y\). I am using the Pandas implementation of it [Sou17]. Correlation ranges between -1 and 1, where -1 indicates a perfect negative correlation, 0 no correlation, and 1 a perfect positive correlation. It can be defined as
\[ \rho = \frac{\text{cov}(X,Y)}{\sigma_x \sigma_y}, \]  

(5.4)

where \( \text{cov} \), covariance, is the degree to which the variables move in tandem, and \( \sigma \) is the standard deviation of each of the variables. To analyze the way my models behave with more common versus uncommon blocks, I use the correlation between the F1-score and how frequently a block occurs in the annotations.

**Range**

Attempting to measure the variability and diversity of the block suggestions that are given, I calculate this range as a percentage of the domain size,

\[ \text{Range} = \frac{\# \text{ different block-types in predictions}}{\# \text{ different block-types in annotations}}. \]  

(5.5)

Note that the number of different types in the annotations is 1033. This is lower than the number in the entire dataset, 1936, as not all blocks occur in the test data.

### 5.3.2 Results and Discussion

The overall accuracy results, as seen in Tab. 5.3, were not unlike the validation results. The three bigram models outperform the unigram and convolutional models, with little distinction among them. This is expected, since more features in the input give the networks more data to base their predictions on. However, there are some interesting patterns that arise among the unigram models. While the uniform model accuracy range from around 73% to 96% between \( 1 \leq k \leq 20 \), and the DFS model from 32% to 64%, the BFS model goes from 43% to 94%. That is to say, while the BFS model might not be the strongest for smaller \( k \)s, it shows a strong upward trend; more on this in Sec. 6.2.2. Together with the uniform model, the results from BFS, especially for large \( k \), are in fact comparable to those of the bigram models.

Not unexpectedly, the patterns of the F1-scores, precision, and recall, closely follow those of the accuracy per model. We summarize the metrics for each model by using a weighted (by frequency) average of the scores for each class. These can be found in Tab. 5.4. However, considering the correlation between F1-score and type frequency, as well as the range, more patterns start to arise about the behavior of the different model groups. From the range, we can write off the convolutional
Table 5.3: Top 1, 3, 5, 10, and 20, accuracies for the best performing experiment of each model, ordered by Top 1 accuracy.

and DFS models as simply not working. A good suggestion systems should be able to return more than 51% and 10% of the total block-types. This is especially true for the convolutional model, since this percentage is boosted by the total number of block-types in the dataset being limited due to the truncation described in Sec. 4.2. Considering range, both the uniform and BFS models outperform all the bigram models; the uniform even does so by a large margin. Referring back to Tab. 3.3, we know that as many as 31.3% of the block-types have less than 10 blocks total. This means that the uniform model, especially, with its range of 84%, is able to predict these very uncommon blocks with a high frequency, which might not necessarily be true for the bigram models. The correlation between F1-score and type frequency reinforces this story — that in relation to unigram models, bigram models might slightly overfit to more frequent blocks, or at least get less common blocks confused.

![](image)

Table 5.4: Summary Statistics for test data, ordered by F1-score. The F1-score, precision, and recall, have been combined using the weighted average of each class.

The scatter plots in Fig. 5-5 visualize the correlations between F1-score and type frequency. Fig. 5-4 seeks to explain what patterns we can see. The yellow arrow shows an approximate fitting of the correlation. Despite rather big variance in the two variables, there is a positive correlation between how common a block is to how well the model could predict those blocks. If the blue dots were more closely aligned to the arrow, the correlation would be stronger, and if they were less
Figure 5-4: Annotated scatterplot of the correlation F1-score and block-type frequency of the results from the uniform model. The marked regions represent common but poorly predicted blocks (green), common and well predicted blocks (purple), and uncommon but well predicted blocks (red). The arrow represents a (manually) fitted line that attempts to visualize the correlation. The circled dots are `BarcodeScanner.UseExternalScanner` (left), and `Button.Click` (right).

Closely aligned, the correlation would be weaker. If the best-fitted arrow were pointing downward, the correlation would be negative.

Further, there are three marked areas, one green, one red, and one purple, where some interesting things happen. The green area is where common blocks that the model struggles to predict would be. However, since this is a rather good model, it is sparsely populated, apart from a few outlier block-types. The purple area, together with the red area, is where ideally, the blue dots would be densely crowded, since these represent blocks across the domain of frequencies with high F1-scores. The red area, especially, is full of, “success stories”. Block-types in this area all occur less than 10 times, but still have a rather high F1-score. These category tend to include the “generic” blocks, that work as an abstraction for using `Component.Action` blocks in loops and procedures. The reason behind their success might be that, assuming they are used by more advanced users, these users show similar programming patterns otherwise, that the models are able to extract. This assumes that the “desirable path” theory mentioned in Sec. 2.2.5 holds for open-ended App Inventor projects.

There are two dots that have been circled serve as examples of blocks with different type frequencies. These represent `Button.Click` with a frequency of 31,458 (orange) and `BarcodeScanner.SetUseExternalScanner` with a frequency of two (bright yellow).

Understanding what the scatterplots mean, we can start comparing them to each other. First, we see that the convolutional (Fig. 5-5f) and DFS traversal, Fig. 5-5g, simply put, fail to predict a
lot of different block-types, and the ones that are predicted tend to be more common types. The
uniform and BFS traversal graphs are not too dissimilar from each other. However, across the
spectrum of type frequencies, uniform performs better. This can be thought of as all the blue dots
being shifted upward from Fig. 5-5e to Fig. 5-5d, with no visible setbacks for any range of frequency.
However, comparing uniform to the bigram models, Fig. 5-5a through Fig. 5-5c, while the upward
trend continues very strongly among the more common blocks, as seen in the visibly empty lower
right quadrants, the top left corners are notably more empty than that of the uniform model. This
confirms that the bigram models, across the board, are worse than the uniform model at predicting
less common block-types.

The reason for this likely lies in the way the bigram models were constructed. As discussed in
Sec. 4.1.4, the number of possible features for the bigrams well exceeded what would be computa-
tionally feasible, and the dimensions were restricted. However, this also restricted which unigram
blocks were included, meaning that adding a less common block to a project might not do anything
at all to the input data. While this allowed to include more bigrams which were helpful to predict
more common blocks from common projects, we see the adverse effects in the correlation between
block frequency and F1-score.

Although precautions were taken to follow the machine learning framework for testing, the tests
that were carried out in this section do not necessarily correspond to how these models would perform
as suggestion systems. Since the models are tested on the data representations they were trained
with, and these are all different, comparing the accuracies is not necessarily fair. For example, the
uniform model has generally more information than BFS and DFS from which to make its prediction,
although the input vector indices correspond to the same features across the three models. Therefore,
the results from this chapter serve as proof of predictive powers rather than showing which model is
inherently better at giving suggestions. The next chapter attempts to place these models in a more
practical setting with a common denominator, a simulation, and evaluate them on their intended
task, suggesting blocks to users.
Figure 5-5: Scatter plots of the relationship between frequency of a block-type and F1-score. Each blue dot represents a type of block. Only types that were in the prediction range, with F1-score greater than 0, are included. Note that the x-axis uses a logarithmic scale.
Chapter 6

Simulating the Suggestions

In developing any intelligent agent, we have to make sure that it serves the goals of humans and not the other way around. AI must fulfill the purpose it is set out for, and improve the user experience. Minimizing a loss function does not necessarily mean that the model provides the best possible suggestions it could, the same way maximizing grades does not necessarily mean maximizing learning. It relies on the assumption that better accuracy equals more educational suggestions. Therefore, trying to build a better experience cannot solely rely upon evaluation on algorithmically generated data, but humans need to be involved in the process. There are many ways numbers can fail us. For example, we cannot be certain, that even with a highly accurate model the suggestions given would not be confusing instead of helpful.

Bridging the gap between tests and a user study, this chapter describes a simulation of what suggestions the models would give while a project was being built, trying to measure how the models perform in the context of actual App Inventor projects.

6.1 Simulation

In order to assess the ability of the models to act not only as pattern finders to answer the question “Which block is missing?”, I trace the performance of each model through the building of a single test project. This aims to bridge the gap between the evaluations in Sec. 5.3.2 and a future user study, as it is dependent on the order a user would actually build the block in, even though it is not interactive. Earlier, the models have only been tested on the data it was constructed on, while here, they are brought together to see how they behave on the same input. I consider how well the models predict the block the user includes at each drawer access, through direct accuracy, and
how it manages to suggest blocks that appear later in the project, through accumulative accuracy. Further, I look at the blocks that are suggested but not used in the project — are they sensible, could they be inserted? Are they relevant, do they drive the project forward? If the models succeed on these two points, this would benefit the users in terms of cognitive load and how much time users spend looking for blocks, as discussed in Sec. 1.1.

Because of time and resource constraints, I only carry out this experiment using one project. Therefore, the results should be interpreted as exploratory rather than assertive.

The project chosen belongs to one of the users in our test dataset. I have chosen to call her Alice. Using data from the in-environment survey mentioned at the end of Ch. 3, we have been able to identify some of Alice’s characteristics. She is a woman with no programming experience outside App Inventor, who is using App Inventor “to make apps for herself and her friends”. Alice is between the ages of 35 and 44, and at the time the data was collected, she had 22 projects in her repository. Judging by the project names, it appears that she has completed a few tutorials. Many of her projects are closely related — she appears to want to write apps to keep track of orphaned kittens. She makes multiple attempts of this app, with several different versions each, interwoven with tutorials on databases and lists. Her projects on average contain 98 blocks, ranging from 11 to 229. The project I chose, a working version of the Kitten Tracker, was her 17th project out of 22. The interface of the Kitten Tracker is functionally designed, allowing for access and update to a database (Fig. 6-1). It had 1 screen, 72 blocks total, of 28 different block-types (Fig. 6-2). It was chosen after investigating several of Alice’s projects, many of which were incomplete, or in other ways broken, and finding that the Kitten Tracker was a working app. Further, because Alice had multiple projects with multiple screens, it was important that the app only had one screen, since the models have no notion multi-screen projects.

In order to simulate the user experience building this project, my advisor, Lyn Turbak, who had not at this point seen the output of any model, built the project block by block while recording the progress. He was given an .aia file containing the components of the project, as well as a screenshot of the completed blocks workspace. Although an expert in the field, he made conscious effort to build the project the way he imagines a typical user with a moderate level of App Inventor experience would build it.

While working on this simulation, it occurred to us that inserting blocks from the drawers is just one of many ways that blocks are added to a project. The users are able to add blocks by:

- Dragging-and-dropping blocks from the drawers.
• Accessing `lexical_variable_get` and `lexical_variable_set` blocks through a fly-down menu when hovering over the name of a `global_declaration` block (Fig. 6-3).

• Copy-pasting blocks using the built-in short cuts `ctrl+c` and `ctrl+v`. Doing this copies not only one block, but all of its children.

• Searching for blocks using the keyboard, “typeblocking”. When a few characters are typed, a menu of possibilities appears.

• After inserting a property block for a component in whichever way, it is possible to edit which component it is referring to (Fig. 6-4). However, the user can only do so within the specified component-type, which will not change the block-type.

Lyn used dragging-and-dropping from the drawers, copy-pasting, and the fly-down approach when building this project. Imagining how a user would program, he built the first occurrence of the sequence of three statements inside `btnReset.Click` when creating the procedure, and later moved them to the procedure declaration.

I have made the decision only to predict the blocks that were added through the drawers. This is motivated by the fact that unless the user is accessing the drawers, the user would not see any suggestions and the assumption that a Suggested Blocks drawer would not be habit-forming enough to make copy-paste obsolete. By creating this simulation, I also assume that users follow the pattern
Figure 6-2: Blocks that make up Kitten Tracker.

Figure 6-3: Fly-down menu that appears when hovering over a global declaration name.

that Lyn uses when building it. While it could be interesting to see what would be suggested for
the blocks that are copy-pasted, for my purposes, I am unsure that these blocks should be given the
same weight as those that are dragged-and-dropped from the drawers. Further, a copy-paste often
includes multiple blocks due to all children of the highlighted block also being copied. Therefore,
not including these reduces the level of complexity of my models, allowing for easier interpretation
of the results.

I then studied the recording. Every time before a block was about to be inserted from a drawer,
I recorded the current state of the project and added an instance where the current state was to
predict the block to be inserted. In total, from the 72 blocks, 31 were either copy-pasted or accessed
using the fly-down menu in the globaldeclaration. This gave me a total 41 instances to test my
networks on.
6.2 Analysis

I hard-coded the results into input and annotations for the best models of each of the seven different data representations. I analyzed them in terms of accuracy from a variety of different angles. Not only is it important that the blocks used are suggested, but at what stage in the project are correct suggestions given, and what kind of blocks are suggested, but not included? This section consists of two parts: (1) the definitions of the metrics I am using, and (2) the results, including a discussion of the findings.

6.2.1 Definitions

Direct Accuracy

Direct accuracy answers the question: at the moment the user inserted Block A, would the user have been able to drag it straight from the “Suggested Blocks” drawer? It is indeed when the user is about to insert the block, that we want it to be in the drawer. However, this gives a more narrow look of the project and doesn’t consider ideas that could have sprung from accessing the drawer at an earlier time, or alternative paths the user could be taking to build the project.

I calculate direct top-k accuracy the same way I have been calculating accuracy earlier in this process. k refers to the number of blocks the drawer would have to contain in order to include the block.

Accumulative Accuracy

As opposed to direct accuracy, accumulative accuracy seeks to answer the question: at any point before block A was inserted, had block A been suggested? For example, assume Lyn had yet to
insert a \texttt{Button.Click} into the program. If \texttt{Button.Click} is suggested, but Lyn chooses to insert a different block this time, this would count as a success, because if he would have chosen to program in a different order, this would have been a great suggestion. Therefore, this is a more generous definition of accuracy than direct accuracy. The metric assumes that the model generates $k$ blocks at each access. Knowing that our, according to the experimental results in seen in Fig. 5-5, Tab. 5.4 and Tab. 5.3, best models are order-agnostic (uniformly sampled as opposed to sampled using a traversal), for our purposes, the order of insertion \textit{should not} matter too much when it comes to the power of the networks to deliver accurate predictions.

\textbf{Correlation between accuracy and state of project}

Another interesting factor would be to ask the question at which state of the projects, e.g., just starting out or when almost finished, the models are able to give the best results. I, therefore, computed the correlation between success and failure (0, 1) and the number of blocks already in the workspace. In table Fig. 6-8, I am representing this for each possible $k$. A positive correlation means that, for that $k$, the model will give more accurate suggestion the more blocks the user have in the workspace. A negative one would mean that the suggestions that are given when the user is just starting out are more accurate than the later ones. The Pearson correlation is described in Sec. 5.3.1.

\textbf{6.2.2 Results and Discussion}

Compared to the results from Sec. 5.3.2, the results from the simulation were in many ways surprising. There seem to be some common patterns that could be used to explain the differences in relative performance of the models. However, a lot is open to speculation.

In general, the performance of the models in terms of accuracy is dismal compared to earlier experiments. As can be seen in Tab. 6.1, the highest top-1 direct accuracy is 19.5%, compared to 89.9% from the experiments. While some drop is expected as the earlier experiments were run with the same configuration of the data as the models were trained on, this is quite drastic. Perhaps, this shows that predicting \textquote{Which block was removed?} is not directly analogous to which block should be suggested, and the models were strongly overfitting to answer that question.

However, it is interesting to consider the change in relative ordering of models. While Tab. 6.1 shows uniform as the best data representation across the board, there is fluctuation in the relative ranks of the other models between the different values of $k$. Since it is not clear what $k$ should be
Table 6.1: Top 1, 3, and 5, direct accuracies for the best performing experiment of each model, ordered by Top 1 accuracy.

<table>
<thead>
<tr>
<th>Model</th>
<th>Top 1</th>
<th>Top 3</th>
<th>Top 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uniform</td>
<td>19.5%</td>
<td>39.0%</td>
<td>46.3%</td>
</tr>
<tr>
<td>Adjacent-bigrams</td>
<td>17.1%</td>
<td>22.0%</td>
<td>22.0%</td>
</tr>
<tr>
<td>Buggy Handler-bigrams</td>
<td>17.1%</td>
<td>19.5%</td>
<td>19.5%</td>
</tr>
<tr>
<td>BFS traversal</td>
<td>9.8%</td>
<td>22.0%</td>
<td>26.8%</td>
</tr>
<tr>
<td>Convolutional</td>
<td>5.9%</td>
<td>11.8%</td>
<td>14.7%</td>
</tr>
<tr>
<td>Handler-bigrams</td>
<td>4.9%</td>
<td>12.2%</td>
<td>14.6%</td>
</tr>
<tr>
<td>DFS traversal</td>
<td>0.0%</td>
<td>14.6%</td>
<td>19.5%</td>
</tr>
</tbody>
</table>

used, or even whether $k$ should be in the range between 1 and 5, Fig. 6-5 explores the top-$k$ accuracy for all models across all $k$ from 1 to 25. We can see that although both the adjacent- and buggy handler-bigram models start out very strongly in the range $1 \leq k \leq 3$, they are overtaken by the BFS traversal. Depending on what number $k$ would be most appropriate in order to not overwhelm our users with a large number of suggestions, we might actually not want to consider bigrams for this task. For larger $k$s, they do not even beat the baseline of always returning the $k$ most common blocks.

![Figure 6-5: Progression of direct accuracy over all $k \leq 25$, for all models, compared to the baseline of always returning the $k$ most common blocks.](image)

However, the overfitting might be a consequence of what was discovered in Fig. 5-5, where bigrams, although performing better in general, had lower scores for less common blocks. Because
of the input features being restricted to more common blocks, even with some knowledge of structure, they lack information on what to do when uncommon blocks are present. Therefore, the models struggle to predict these less-frequent blocks, which might be part of the issue as to why they do not perform well in the simulation. It would be interesting to explore what would happen if these models first and foremost would include all unigrams, with the addition of the most common bigrams.

Although the accuracy has dropped by a lot considering smaller $k$, there is promise for implementing a larger $k$. When $k = 10$, the uniform model gives a direct accuracy of 60%, meaning that in more than half the cases our user would have been able to insert the block straight from the Suggested Blocks-drawer. With $k = 25$, this number has risen to above 80% of the cases. However, this number is highly dependent on the order the user chose to insert the blocks. As we might also think that the content of the Suggested Blocks-drawer might affect the order to conform to the blocks being suggested, it is also interesting to consider the accumulative accuracy over different $k$s. A graph of accumulative accuracy over $1 \leq k \leq 25$ can be seen in Fig. 6-6.

![Figure 6-6: Progression of accumulative accuracy over all $k \leq 25$, for all models, compared to the baseline of always returning the $k$ most common blocks.](image)

The relative ordering of the models in Fig. 6-6 is similar to that of Fig. 6-5, however, there are stronger groupings of accuracies. Although the bigram models are fluctuating heavily, they converge toward the same accuracy, around 75% for larger $k$s. The convolutional and DFS traversal models are shown to, frankly, fail at their task, never even reaching an accuracy of 33%. Notably, and
excitingly, the uniform model is able to reach an almost 90% accuracy for $k = 10$, achieving 100% when $k = 21$. It would be worth investigating if a user who only selects blocks from the Suggested Blocks drawer would be able to build the Kitten Tracker with such a $k$.

On manual inspection of the results, part of what was causing the low results of the convolutional and DFS models, was that after about a midway point in the project, the suggestions became more or less static. The top-3 suggestions stayed the same — generic variable and value related blocks that have very little relevance to the specific working of the project. Meanwhile, the other models kept changing their suggestions in response to new additions of blocks in an expected way.

Despite the high accumulative accuracy, we can easily deduce that when $k = 25$ and we make over 40 accesses to the predictions, we are able to predict over 1000 different block-types, which is half the number of total block types, and more or less all block-types in the test data. This would likely give us a high accuracy even if we were to use a random generator as our predictor. In order to not overwhelm users, and be able to justify the relevance of blocks even if using a large $k$, the entire body of our predictions should not be that large in comparison to the number of of blocks we use. Looking at Fig. 6-7, we can see that our models suggest nowhere near the upper bound of 1000 different block-types, but rather remains in the range between 50 and 140 different types, even for $k = 25$.

![Figure 6-7: Progression of size of the set of predicted block-types for all $k \leq 25$, for all models, compared to the baseline of always returning the $k$ most common blocks.](image-url)
We should note, that just because a block-type was not included in the program, does not mean that it was a bad suggestion. Not only are there multiple ways to program that lead to the same result, but there is no way of knowing that the way this program works is what Alice wanted to achieve. While these metrics are interesting tools to understand what might be going on, they do not assure us of what would give good suggestions to users.

Not only is it interesting to consider how well the models perform on the test data, but it is also interesting to consider when in the project they do well. Therefore, I am looking at the correlation between accuracy and how many blocks have already been inserted into the project. If this value is positive, it means that the model got better at suggesting blocks, in terms of accuracy, as time went on. If it is negative, it did better in the first few blocks compared to the later. Neither positive nor negative is good or bad, rather they just convey information about how the models work. Fig. 6-8 displays a graph of the models for each $k$.

![Figure 6-8: Correlation between accuracy of the top-$k$ suggestions and how many blocks have already been inserted into the project.](image)

In terms of uniform, we see that when $k$ is very small, it is better at predicting later blocks, i.e., when there is more data, but for larger $k$, it is better at predicting earlier blocks. Interestingly, the two models with the highest performance, uniform and BFS traversal, show vastly different correlations. Although this experiment only uses one project, this might suggest that there are some fundamental differences in the functions that are approximated in each of these models. It is worth
investigating which of the models would be most useful as recommendation systems, do we want more accurate suggestions as the user inserts the first few blocks of the project, or later when the user might be struggling, or is uninspired?

It should be repeated, that this simulation was just a sample of how the models could perform on real-life data. While these results might open up to more questions, and allow for hypothesizing and speculation, they should not be seen as statistically significant. Rather, I am hoping that this demonstration could open up for, and inspire, future work.
Chapter 7

Conclusion

7.1 Summary

From a dataset consisting of projects from over 46,000 prolific users, I have built and evaluated 15 different machine learning models as candidates to become a back-end algorithm for a Suggested Blocks feature for App Inventor. To build these 15 different models, I used 7 different data representations. For the best performing model for each of the different representations, I did a more extensive analysis. I also simulated how accurate the suggestions would be when accessed during the building of an original user project.

The data representations included different n-gram models, different ways of constructing annotations, and a more advanced convolutional representation. I processed these using Python packages such as CountVectorizer and Word2Vec, and built the neural networks using Keras. I used a combination of performance metrics, visualizations, and correlations, to evaluate the models.

The results were, generally, promising for the possibility of using neural networks to power a Suggested Blocks feature. The best models in the experimental setting, the bigram models, gave an almost 90% top-1 accuracy, closely followed by the uniform model. This included a weak positive correlation between F1-score and block-type frequency, yet many uncommon block-types, ones that occur as few as ten times across the dataset, had good F1-scores.

However, the simulation showed that these results do not necessarily mean that the suggestions would be of that caliber. Despite a drop in maximum top-1 accuracy to around 20%, when returning a larger number of blocks at a time, such as 10, or 20, the suggestions generally included the blocks of the project. Here, the uniform and BFS models outperformed the other models by a margin,
possibly because of their stronger ability to suggest less common blocks.

While the results from the simulation cannot necessarily be considered absolute and future work remains, the results from this study give promise to the future implementation of a neural network based Suggested Blocks system for MIT App Inventor.

7.2 Future Work

My project is the first step towards a new App Inventor feature. However, much more investigation remains before the implementation of such a feature could be viable. This section is divided into two parts, Sec. 7.2.1, aiming to define the steps necessary to achieve a minimum viable product to be implemented in the environment, and Sec. 7.2.2, describing ways in which that product could be further improved upon.

7.2.1 Near Future

Different Models

In this project, shortcomings were discovered when evaluating the data, many of which could have been mitigated by building different models, while these might need even more computational power. Fixing these shortcomings could include investigating building a bigrams model using all unigram tokens, and then add as many bigrams as feasible. It could also include investigating the way a convolutional model could be improved, perhaps using a higher dimensionality for the Word2Vec transformation. It could even include metadata, such as survey data, the number of projects a user has made, or even be responsive to a user telling the model what they are trying to build — A game? A tool? A social network?

More Simulations

While we carried out one simulation of a user project as a proof of concept, we would need many more such simulations to confirm that the results obtained were not due to chance. Spending more time analyzing video content and transforming it into our given data representations would ensure that a model that was taken further, perhaps to a user study, would, in fact, be the model that is best suited for the task.

In relation to this, we developed the notion of investigating how these simulations would play out for different personas. From the survey mentioned in Ch. 3, we could extract a set of users and
analyze them from previous programming experience, gender, and purpose of using App Inventor, to assure that we fulfill the objectives that pertain to learning style, as mentioned in Sec. 1.3.3.

**Objective Look on the Suggestions**

While it is easy to define good suggestions in terms of accuracy, it is harder to define what a bad suggestion would be without either subject knowledge, or a personal investment, such as using them to build a project. Therefore, in addition to more simulations, an idea would be to crowd-source this knowledge from a range of App Inventor experts, who could evaluate a set of suggestions, give feedback on what should be there, and what should not, as well as presenting the suggestions to users while they are building their projects. Depending on their views, we might consider a “sensibility filter” that would remove bad suggestions.

**Implementation**

Having done more of the research described above, a model would be selected to be the back-end system of the Suggested Blocks feature. However, with this, some design decisions would have to be made that do not necessarily follow from this project’s research. For example, when a Component.Action-block is suggested, do we specify the component it belongs to if there are more than one of that type? This would be the case with any app that has more than one button — when Button.Click is suggested, which of the Buttons should it refer to? Would the user be able to specify this with a drop-down block? Should this be inferred based on the content of the workspace? Should copies of the block be made for each of the components?

**User Study**

As discussed at the beginning of Ch. 6, we need to make sure that the models serve its purpose — helping users. Therefore, with an initial implementation, we need to observe how users interact with the new Suggested Blocks drawer, and determine (1) whether users access the drawer, (2) whether those who do show any different behavior, (3) how usable the drawer is, and (4) how useful the users think the drawer is. This is perhaps the most important part of the project, and any implementation and launch of the feature in App Inventor would depend only on the results of this study.
7.2.2 Far Future

Reinforcement Learning

Assuming that a user study shows success, the feature could be launched to the above 6 million users of App Inventor. However, the development of the feature should not stop here. With the help of reinforcement learning, it could be made sure that the model continues to improve the longer time goes on.

Say that we define a measure of “success” and objective for each project. This could be uploading it to a gallery, downloading it and using it regularly on a phone, being shared across more than one device, or even how many times users access the suggested blocks drawer. The idea behind reinforcement learning is that any behavior the model is able to reproduce that leads to the objective is reinforced, similar to the way the models in this study were trained using the annotations. The longer the feature is running, and the more people use it, the better it will be at giving suggestions that benefit the user. It would give the model access to real life information, which I cannot imagine would not be vastly better than any data representation that could be reproduced computationally. In a way, the network would get a life of its own, and be closer to what most people would call “intelligent”. When a tutor suggests something to a student, it is likely not solely based on what other students have done, good or bad. It would be grounded in what would drive the current project to completion. A system built on reinforcement learning would capture exactly this. Suggested Blocks, as imagined in this paper, would be a great way to initialize a model that is then trained as users interact with it.
Bibliography


[LTM17] Isabelle Li, Franklyn Turbak, and Eni Mustafaraj. “Calls of the wild: Exploring procedural abstraction in app inventor”. In: Blocks and Beyond Workshop (B&B), 2017 IEEE. IEEE. 2017, pp. 79–86.


Appendix A

Python Files

The following Python files are examples of the framework and libraries I used to go from App Inventor summaries, to matrix-representations of the data, to trained models. Running the following sequence in the given order, starting from summaries, would output the uniform-trained model 7. The build_vectorizer.py and sentence_to_matrix.py files were used for all data representations, changing only the variable currTask. From summary_to_sentence_UNIFORM.py and remoteVersion7.py, most of the functions were kept the same, however changes were made to typesFromScreen, makeTruthStructure in summary_to_sentence_UNIFORM.py, as well as the configuration of the neural network architecture in remoteVersion7.py.

A.1 kitten_tracker_summary.json

Example structure of the summaries that are used to create the uniform model. Note that the other models used a modified summary that preserved the tree structure of the programs.

```json
{
  "**Project Name":"Kittens_TinyDB_Fusion",
  "**created":1430772026975,
  "**modified":1430937930460,
  "*Media Assets":[],
  "*Number of Screens":1,
  "Screen1":{
    "Blocks":{
      "*Top Level Blocks":{
        "Button.Click":3,
        "Form.Initialize":1,
        "ImagePicker.AfterPicking":1,
        "ListPicker.AfterPicking":1,
        "global_declaration":1,
```
"procedures_defnoreturn":1
},
"Active Blocks":{
  "Number of Block Types":29,
  "Number of Blocks":72,
  "Global Variable Names":{
    "nameList":7,
  },
  "Local Variable Names":{},
  "Procedure Names":{
    "resetForm":3,
  },
  "Procedure Parameter Names":{},
  "Strings":{
    "null":7,
    "Database Cleared":1,
    "Error — Incident ID already saved to DB":1,
    "Success — database updated":1,
    "nameList":2,
  },
  "Types":{
    "Button.Click":3,
    "DatePicker.GetText":1,
    "DatePicker.SetText":2,
    "Form.Initialize":1,
    "Image.GetPicture":1,
    "Image.SetPicture":3,
    "ImagePicker.AfterPicking":1,
    "ImagePicker.GetSelection":1,
    "ListPicker.AfterPicking":1,
    "ListPicker.GetSelection":4,
    "ListPicker.SetElements":2,
    "Notifier.ShowAlert":3,
    "TextBox.GetText":4,
    "TextBox.SetText":4,
    "TinyDB.ClearAll":1,
    "TinyDB.GetValue":4,
    "TinyDB.StoreValue":2,
    "controls_if":1,
    "global.declaration":1,
    "lexical_variable_get":5,
    "lexical_variable_set":1,
    "lists_add_items":1,
    "lists_create_with":3,
    "lists_is_in":1,
    "lists_select_item":3,
    "math_number":3,
    "procedures_callnoreturn":2,
    "procedures_defnoreturn":1,
    "text":12,
  }
},
"Orphan Blocks":{
  "Number of Block Types":0,
A.2 summary_to_sentence_UNIFORM.py

```python
'''Takes as input a set of directories containing ai2_summaries, and returns a set of .json files containing "sentence" truth representations of the input and annotations for the uniform sampling'''

import os
import json
import utils
import datetime

def findSummaries(dirName):
    '''takes a directory, and returns a list of JSON summaries from all subdirectories'''
    #print dirName
```
summaries = []
for source in os.listdir(dirName):
    source = os.path.join(dirName, source)
    if os.path.isdir(source):
        for project in os.listdir(source):
            if project.endswith('summary.json'):
                summaries.append(os.path.join(source, project))
return summaries

def allTypes(summaries):
    '''takes a list of summaries and returns a list of the tuples of (components,blocks)
    of all screens in the list.'''
    result = []
    for summary in summaries:
        with open(summary, 'r') as data_file:
            JSON = json.load(data_file)
            result.extend(typesFromProject(JSON))
    return result

def typesFromProject(JSON):
    '''For each project, return a list of
    the types of each screen.'''
    result = []
    # for each screen...
    for x in [key for key in JSON.keys() if key[0] != '*']:
        result.append(typesFromScreen(x, JSON))
    return result

def typesFromScreen(screen, JSON, top='non-top'):
    '''Return a tuple of two strings,
    the first one representing the components of a screen, and the second one
    representing the blocks. Each block is represented as a word, joined together
    by spaces.'''
    # make a list of the dictionaries with types of components, active and orphan blocks
    blockDict = []
    compDict = []
    try:
        blockDict = JSON[screen]['Blocks']['Active Blocks']['Types']
        except KeyError:
            pass
        except TypeError:
            pass
    compDict = JSON[screen]['Components']['Type and Frequency']
    blockString = dictToString(blockDict)
    compString = dictToString(compDict)
    return compString, blockString

def dictToString(dyct):
    ''' Turn dictionary where the keys are blocks/components
    and values are counts of the keys into string representations'''
    result = ''
    for key in dyct:
        # find the number of occurrences of key
n = dyct[key]
# text analyzer doesn’t work with dots, if there are dots, concatenate the two
# subwords
if '.' in key:
    key = ''.join(key.split('.'))
# append key to result n times
while n > 0:
    result += key + ' ' 
    n -= 1
return result[:-1]

def makeTruthStructure(projtypes):
    ''' Takes a list containing (components,blocks) tuples, and returns the truth
    structure X,y where entry i in X predicts entry i in y'''
    X = []
    y = []
    for screen in projtypes:
        types = screen[1].split()
        curr = []
        for i in range(len(types)):
            if types[i] not in curr:
                X.append(screen[0] + ' ' + ''.join(types[:i]+types[i+1:])))
                y.append(types[i])
            curr.append(types[i])
    return X, y

    ''' CODE IN THE NEXT SECTION WAS WRITTEN BY LYN TURBAK, ORIGINALLY FROM AI2_SUMMARIZER
    '"
    #-----------------------------------------------
    def readIndex():
        if not os.path.exists(lastIndexProcessedFilename):
            return -1
        else:
            with open(lastIndexProcessedFilename, 'r') as inFile:
                return int(inFile.readline().strip())

    def writeIndex(index):
        with open(lastIndexProcessedFilename, 'w') as outFile:
            outFile.write(str(index))

    def processNext(pDir):
        #utils.createLogFile(logname)
        nextIndex = readIndex() + 1
        while nextIndex <= lastIndexToProcess:
            processChunk(nextIndex, pDir)
            writeIndex(nextIndex)
            nextIndex += 1

    def processChunk(chunkNum, pDir):
        currentDir = os.path.join(pDir, utils.padWithZeroes(chunkNum,2))
        start = datetime.datetime.now()
        utils.logwrite('*** Start sentences.processChunk chunk {} at {}'.format(str(chunkNum), str(start)))
        utils.logwrite('fetching text...')
        project_summaries = findSummaries(currentDir)
        if len(project_summaries) == 0:
            return
        projtypes = allTypes(project_summaries)
A.3 build_vectorizer.py

```python
'''Creates vectorizers for the X and y corpuses of the preprocessed stringshaped .json files that have already been created using summary_to_sentence~currTask~.py.'''

import os
import json

```
from sklearn.feature_extraction.text import CountVectorizer
import cPickle
mainDir = '/Users/Maja/Documents/THESIS/mining/
currTask = 'uniform'
dirName = mainDir+currTask+'/

def make_corpus(directory, x_y):
    for source in os.listdir(directory):
        fileName = os.path.join(directory, source)
        if fileName.endswith(x_y+'.json'):
            print source,
            if currTask == 'conv':
                if x_y == 'X':
                    yield ' '.join(json.load(open(fileName))['comps'])
                else:
                    yield ' '.join(json.load(open(fileName)))
            else:
                yield ' '.join(json.load(open(fileName)))

if currTask == 'a_bigrams' or currTask == 'h_bigrams' or currTask == 'h_bigrams_bug':
    vectorizerX = CountVectorizer(stop_words='english', decode_error='ignore',
                                  max_features=3980)
else:
    vectorizerX = CountVectorizer(stop_words='english', decode_error='ignore')
vectorizerX.fit(make_corpus(dirName+'data/sentences', 'X'))
with open(dirName+'vectorizerX.pkl', 'wb') as fid:
    cPickle.dump(vectorizerX, fid)
print "wrote vectorizerX at " + dirName+'vectorizerX.pkl.'
print len(vectorizerX.get_feature_names())

vectorizery = CountVectorizer(stop_words='english', decode_error='ignore')
vectorizery.fit(make_corpus(dirName+'data/sentences', 'y'))
with open(dirName+'vectorizery.pkl', 'wb') as fid:
    cPickle.dump(vectorizery, fid)
print "wrote vectorizery at " + dirName+'vectorizery.pkl.'

A.4 sentence_to_matrix.py

'''Take the sentence data and the vectorizers, and preprocess it into
matrix format, that can then be sent to the cloud to train a neural network.'''
import os
import json
import cPickle
mainDir = '/Users/Maja/Documents/THESIS/mining/
currTask = 'uniform'
dirName = mainDir+currTask+'/

with open(mainDir+currTask+'/'+'vectorizerX.pkl', 'rb') as fid:
    vec_loaded = cPickle.load(fid)
with open(mainDir+currTask+'/'+'vectorizery.pkl', 'rb') as fid:
    vecy_loaded = cPickle.load(fid)

print "Input dimensions: " + str(len(vec_loaded.get_feature_names()))
print "Output dimensions: " + str(len(vecy_loaded.get_feature_names()))
print os.listdir(dirName+'data/sentences')

for source in os.listdir(dirName+'data/sentences'):
    fileName = os.path.join(dirName+'data/sentences', source)
    print fileName
    numChunks = 5 # this number was incremented to 20 for later data representations

# projects
if fileName.endswith("X.json"):
    X = vec_loaded.transform(json.load(open(fileName)))
    chunk_size = X.shape[0]/numChunks
    for i in range(numChunks-1):
        with open(os.path.join(dirName+'data', 'X', source[:-5] + str(i) + '.pkl'), 'wb') as f:
            cPickle.dump(X[i*chunk_size:(i+1)*chunk_size], f)
    with open(os.path.join(dirName+'data', 'X', source[:-5] + str(numChunks-1) +'.pkl'), 'wb') as f:
        cPickle.dump(X[(numChunks-1)*chunk_size:], f)

# annotations
elif fileName.endswith("y.json"):
    y = vecy_loaded.transform(json.load(open(fileName)))
    chunk_size = y.shape[0]/numChunks
    for i in range(numChunks-1):
        with open(os.path.join(dirName+'data', 'y', source[:-5] + str(i) + '.pkl'), 'wb') as f:
            cPickle.dump(y[i*chunk_size:(i+1)*chunk_size], f)
    with open(os.path.join(dirName+'data', 'y', source[:-5] + str(numChunks-1) +'.pkl'), 'wb') as f:
        cPickle.dump(y[(numChunks-1)*chunk_size:], f)


done importing, define funcs

def generator(xDir, yDir):
    xlist = sorted(os.listdir(xDir))[11:]
    ylist = sorted(os.listdir(yDir))[11:]
    assert len(xlist)==len(ylist)
    while True:
        for i in range(len(xlist)):
            X = open(os.path.join(xDir,xlist[i]), 'rb')

A.5 remoteVersion7.py

This file was named remoteVersion{experimentNum}.py as opposed to the “local” versions I had running on my local machine to test out the code.
y = open(os.path.join(yDir,ylist[i]), 'rb')
yield pickle.load(X, encoding='latin1').todense(),pickle.load(y, encoding='latin1').todense()
X.close()
y.close()

def set_aside_data(xList, yList):
    for i in range(len(xList)):
        with open(os.path.join(dirName+'X',xList[i]), 'rb') as x_not:
            if i == 0:
                X = pickle.load(x_not, encoding='latin1').todense()
            else:
                X = np.append(X, pickle.load(x_not, encoding='latin1').todense(), axis=0)
        with open(os.path.join(dirName+'y',yList[i]), 'rb') as y_not:
            if i == 0:
                y = pickle.load(y_not, encoding='latin1').todense()
            else:
                y = np.append(y, pickle.load(y_not, encoding='latin1').todense(), axis=0)
    return X,y

print ('set aside validation data')

sortX = sorted(os.listdir(dirName+'X'))
sortY = sorted(os.listdir(dirName+'y'))
valid_xList = sortX[6:1] # [6:] is to avoid .DS_STORE + test data
valid_yList = sortY[6:1] # [6:] is to avoid .DS_STORE + test data

validX, validY = set_aside_data(valid_xList, valid_yList)

print ('shape of X: ' + str(validX.shape))
print ('compile model')

model = Sequential()
model.add(Dense(2000, input_dim=validX.shape[1], activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(2000, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(2000, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(validY.shape[1], activation='softmax'))
top3_acc = functools.partial(top_k_categorical_accuracy, k=3)
top3_acc.__name__ = 'top3_acc'
top5_acc = functools.partial(top_k_categorical_accuracy, k=5)
top5_acc.__name__ = 'top5_acc'

model.compile(loss='categorical_crossentropy', optimizer='rmsprop',
metrics=['accuracy', top3_acc, top5_acc])

print ('fit model!!!!')

history = model.fit_generator(generator(dirName+'X', dirName+'y'),
    steps_per_epoch=45*10, 
    epochs=25,
    validation_data=(validX, validY))

print ('save model, history')

model.save('model7.h5')

with open('trainHistoryDict7.pkl', 'wb') as file_pi:
pickle.dump(history.history, file_pi, protocol=2)