Measuring the Programmatic Intricacy of App Inventor Projects Grouped by Functionality

Benjamin Xie, Isra Shabir, and Hal Abelson
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts, USA
Email: {bxie, ishabir, hal}@mit.edu

***Abstract*—MIT App Inventor is a web service that enables users with little to no previous programming experience to create applications for mobile devices using a visual blocks language. We analyze a random sample of projects to better understand the capability of using App Inventor to create Android applications of varying functionality. We group projects of similar functionality and measure the programmatic intricacy of these groups using multiple metrics. Our results depict qualitative differences in the learnability of the App Inventor service to implement apps of particular functionality. Our discussion addresses these findings in the context of App Inventor's learning resources.**

I. Introduction

MIT App Inventor is an environment that leverages a blocks-based visual programming language to create mobile apps for Android devices. An App Inventor project consists of a set of components and a set of blocks which give the components functionality. Components include items seen on the app (e.g. buttons, text boxes, images, drawing canvas) as well as non-visible items (e.g. camera, accelerometer, speech recognizer, GPS location sensor).[1] The app's functionality is programmed in Blockly, a visual blocks-based programming language. For example, Figure 1 shows blocks for an app to prevent texting while driving. When a text is received, a predefined message is sent back to the person who sent the original text[2]

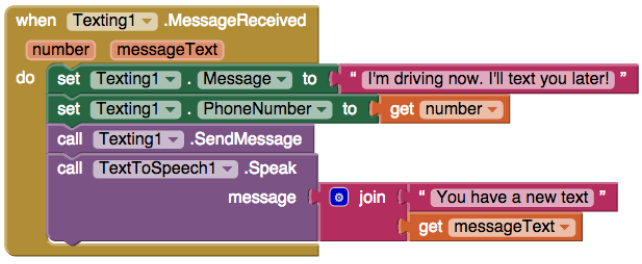

Fig. 1. Blocks for an App Inventor 2 project that automatically responds to texts received with a predefined message.

There have been two major versions of App Inventor. App Inventor Classic (also known as App Inventor 1) was released in 2009 and ran the blocks editor from a separate Java application. In late 2013, App Inventor 2 (AI2) was released; the blocks editor now runs in a web browser as a Javascript program.[6] Although both versions of App Inventor are currently still in use, as of February 2015, 90% of App Inventor users use App Inventor 2. For this reason, this research focuses on App Inventor 2 data.

App Inventor is taught at both grade school and college level, to a broad audience. Reports on courses taught depict App Inventor being used to create very diverse apps. These apps range from apps that discourage texting while driving to apps that track school buses to apps that organize community service cleanups.[5] The pattern we see is that App Inventor enables "situated computing." This quarter-century old concept suggests that the convergence of computing, connectivity, and content enables users to use computing to bridge the gap between intentions and actions. App Inventor enables users to leverage their mobile devices to solve everyday problems they encounter. [7]

App Inventor also has copious resources for self-learners, typically in the form of self-contained tutorials. The App Inventor resources page includes 26 tutorials ranging from basic to intermediate to advanced difficulty. Most tutorials involve creating an entire functioning app from start to finish. Each tutorial typically focuses on either a new component or additional functionality for a previously introduced component.[3]

To date, over 3.1 million users from 195 countries have created over 8.2 million apps with App Inventor. There are over 100,000 weekly active users.[2]

A. Objective

We seek to better understand what users create with App Inventor and determine what kinds of apps the App Inventor environment lends itself towards creating. We define metrics to measure the intricacy of the projects, measuring the proficiency of the blocks code used to program the apps. From this, we determine how easy it is to create an app of a certain functionality as well as the extent of capabilities App Inventor provides to implement certain functions.

We analyze a random sample of projects and group them based on the components used in the app. To measure intricacy, we look at both the number of unique blocks in projects as well as the existence of more advanced programmatic structures, such as procedures, lists, and loops. From this, we evaluate how well suited App Inventor environment is to create apps with different functionalities and behaviors.

In this paper, we explain our technical approach to extracting information from raw project data, filtering and grouping projects, and comparing the intricacy of groups given our metrics. We then discuss our findings in the context of the App Inventor service and its teaching resources.

B. Previous Work

Previously, analysis of App Inventor Classic has been done mostly at a user level.[4] Some of the notable findings include:

- 30% of projects have no blocks and therefore are not functional.
- Nearly 50% of users do not have a single block or component in their projects.
- 53% of users only have 1 project.
- A large number of procedures are only called 0 or 1 time.

This indicated that a large number of App Inventor Classic projects were never completed. It was theorized that a major contributing factor is the usability of the service. Whereas App inventor 2 is a single-page web service, App Inventor Classic required the deployment of an external Java service to program the app. This motivated the significant changes in the blocks language for App Inventor 2 to focus on usability.

II. Technical Approach

We extract features from a random sampling of App Inventor 2 projects. We group projects based on functionality by considering the components they contain and measure intricacy by the total number of unique blocks in the project as well as the existence of ”multiplier” structures. We then look at each group and the distribution of the intricacy of projects in each group.

A. Data Source

Our source data is 5,228 random App Inventor 2 projects. We randomly selected projects according to their project ID, a sequentially increasing number (so a project with a lesser project ID corresponds to a project made earlier). The 5,228 projects selected come from 5,174 users, suggesting that randomization was accomplished. Given the size of the dataset and the randomness of the selection, we assume that the sample is representative of all App Inventor 2 projects. We use Pandas, a python data analysis library, for most of our data processing

Of the 5,228 projects sampled...

- At leasts 20% (1,031 projects) are recreations of tutorials. These recreations of the step-by-step tutorials were found by matching project names. We only look at the 25 tutorials from the App Inventor website, although there exist many other tutorials made by other groups and individuals.[3] These tutorials are included in our dataset.
- 16% (845 projects) define procedures yet only 13% (670 projects) call the procedures multiple times and use the procedures appropriately.

We also find that 21% (1,107 projects) are Certainly Non-Functional (CNF), or are guaranteed to be incomplete apps that have no behavior. If an app is without components, then there is nothing the user can interact with or for the app to do, so it cannot have functionality. In order for an app to have functionality, it must have at least two blocks: One to handle an event and one to respond to an event. Figure 2 shows an example of a simple action from two blocks. No functionality can occur with less than two blocks. We say an app is CNF if it either has no components or has fewer than two blocks.

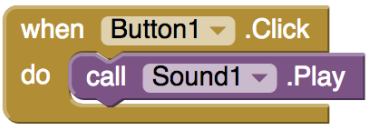

Fig. 2. The simplest app behavior requires at least two blocks: An event handler and a resulting action. Here, a sound is played when a button is pressed.

B. Feature Extraction

We focus primarily on quantitative features to enable us to run clustering algorithms. In particular, we look at the number of each kind of component in a project, as well as the number of each type of block. The type of a component or block is provided in the source code of the project.

Features Extracted from Projects:

- Project Name
- User Name (anonymized)
- Number of Screens in Project
- Number of Components by Type
- Number of Blocks by Type

C. Grouping Projects

We group projects according to the components they contain. The palette in App Inventor organizes components by functionality and places each group in its own ”drawer”. Because the palette neatly organizes components into drawers of similar functions, we use it to define our groups. (Figure 3).

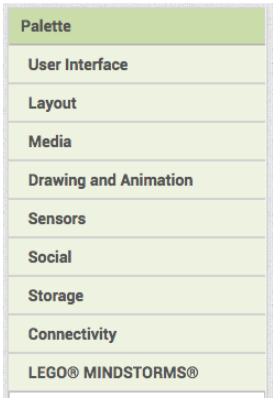

Fig. 3. The palette in App Inventor 2 groups components into categories. These categories are used to group projects by functionality.

We follow the palette drawers to define our groups, with two notable changes: Disregarding the ”Layout” component drawer entirely as well as the sound and clock/timer components in our consideration of functionality groups.

Layout components were removed as they are extraneous and irrelevant when grouping by functionality. Layout components enable users to change the visual display of apps and nothing further. Since we group projects by their functionality and layout components add no additional behavior to apps, they are disregarded.

The sound and clock components were removed to improve the differentiation between groups without detracting from the purpose of the app. The sound component (media drawer) plays a sound whenever the user specifies. Examples include playing a "meow" when an image of a cat is pressed to playing a famous speech in a historical quiz app. The clock (sensor) component enables apps to keep track of time. Uses for this are varied, ranging from keeping time in a stopwatch app to keeping time to periodically moving a sprite in a game app. The components certainly offer functionality to apps, but the component alone gives little affordance as to the intended behavior of the app. Because the sound and clock/timer components have very general uses and are often used, they do not help differentiate groups and are also excluded in the consideration of functionality groups.

There are eight groups. Basic apps only contain User Interface components (and possible Layout components, but they are ignored). *Media*, Drawing, Sensor, Social, Storage, Connectivity, and Lego contain at least one component in the respective group in the palette. This categorization allows for overlap, as projects that contain components from multiple palette groups may be placed into multiple groups. In example, a project that uses both bluetooth (connectivity) and twitter (social) component would be labelled both Connectivity and Social. The exception is the Lego group, which we deemed to be an exclusive group because of the specificity of the components. Since Lego components are solely for integration with Lego Mindstorms, they produce the most specific apps. So if a project contains a Lego component, it is grouped as only Lego, regardless of other components it may contain.

Reiterating, Basic and Lego groups are disjoint from other groups and each other. Other groups may overlap. Table 1 provides a description of each group, the condition for a project to be in that group, and example components from that group.

While we used components to group projects, we use the blocks that provide components functionality to measure project intricacy.

TABLE I. FUNCTIONALITY GROUPINGS

Group Name	Description of App Functionality	Condition	Example Components
Basic	Only basic user interface components and nothing further	Only User Interface Components	Button, Image, Label, Notifier, Textbox
Media	Playing/recording of audio or video	At least 1 media component (excluding "sound")	Camera, TextToSpeech, VideoPlayer
Drawing	Use screen as canvas for drawing	At least 1 drawing component	Canvas, Ball, ImageSprite
Sensor	Response to phones' sensors	At least 1 sensor component (excluding "clock')	Accelerometer Sensor, LocationSensor, NearField (NFC)
Social	Communication via phone or web	At least 1 social component	Texting, Twitter, PhoneCall
Storage	Saving information	At least 1 storage component	TinyDB, FusiontablesControl, File
Connect	Networking with other apps and phones	At least 1 connectivity component	ActivityStarter, BluetoothClient, Web
Lego	Control Lego Mindstorm kits	At least 1 lego components	NxtDrive, NxtLightSensor

D. Measuring Programmatic Complexity

We define the "intricacy" of an App Inventor project as a measurement of the proficiency and quality of the implementation, as evidenced by the blocks used to program it. Intricate projects tend to exhibit good coding practices, such as code reuse through procedures and loops. They also tend to use advanced structures such as lists. A more intricate app tends to either implement more complex components or uses components in a more advanced manner.

Of particular focus in our measure of intricacy is code reuse. In example, consider the case where two functionally similar projects exist and Project A copies the same code in three locations whereas Project B defines a procedure and calls the procedure three times. We argue Project B exhibits strong coding practices as it leverages code reuse in the form of procedures and therefore is a more intricate project. Project A has a greater number of blocks, but Project B has a slightly greater number of unique blocks with the blocks to initialize and call a procedure. A project that appropriately uses a procedure or for-loop rather than copies and pastes blocks (creating redundant code) would show evidence of greater intricacy, even if the resulting app in either case has identical functionality. We define the procedure, loop, and list blocks to be multiplier structures, as they increase the intricacy by "multiplying" the functionality of the the blocks contained in the procedure, loop, or list.

We define two metrics for programmatic intricacy in App Inventor: The number of unique blocks and the existence of different multiplier structures in the blocks code. We choose the number of unique blocks instead of the total number of blocks so the measure of intricacy is not affected by redundant code. We check projects for the existence of multiplier structures to account for advanced constructs that suggest more intricate code.

Note that we try to ensure that the constructs are used properly. In example, a project is only considered to have a procedure if it is not only initialized, but also called multiple times as procedures are supposed to be. A full description of the conditions to have a multiplier structure counted in a project is shown in Table 2.

TABLE II. MULTIPLIER STRUCTURES

Structure	Condition
Procedure	At least one procedure initialized and called multiple times
Loop	At least one loop block created
List	At least one list initialized

Our two metrics of the number of unique blocks and the existence of advanced, multiplier structures specifically addresses the issue of redundant coding when measuring the intricacy of a project's implementation.

III. Results

We group the projects by functionality and show the distribution. Next, we show the distribution of the number of unique blocks in projects for each group. Finally, we show the fraction of projects within each group that use certain multiplier structures.

A. Grouping

After grouping projects by functionality, we find that 77.3% of projects can be categorized into exactly one group. The remaining projects were categorized across multiple groups. The 4,121 projects resulted in 5,369 categorizations into groups; on average, a project fits into 1.3 groups.

Due to simplicity and the availability of learning resources, the largest groups are the Basic and Drawing groups, each containing around 24% of the projects. Since basic apps are simple to create as they only include user interface components, it follows that there is a large amount of them. And when looking at the App Inventor tutorials, we see that over half of the beginner tutorials most users start with involve the creation of a drawing app. So, this suggests that the large amount drawing apps users create are in the form of tutorials and projects that extend from tutorials.

The Lego group is the smallest, containing only 28 (0.5%) projects. One likely reason is because there is an additional hardware requirement in the form of Lego Mindstorm kits to use an app grouped as Lego. Another is that there exists no official tutorial for Lego projects. The full breakdown is shown in Figure 4.

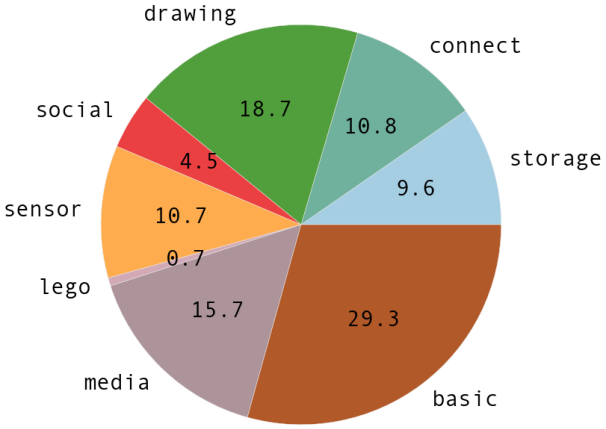

Fig. 4. Functionality Groups by Size. 77.3% of projects are categorized into exactly one group, with the others categorized across multiple groups

B. Number of Unique Blocks

We plot the distribution of the number of unique blocks in each group and compare these subsets of projects to each other and the entire set of projects.

Each subset and the entire set of projects exhibits a right skew, suggesting that each group contains a few outlier projects that have a significantly greater number of unique blocks and are likely well-developed projects.

Storage apps have the greatest median number of unique blocks, the widest distribution, and contains the project with the most number of unique blocks, suggesting that it has the greatest capability to create intricate apps.

The short distribution of the Lego group suggests its capabilities are limited. Despite this narrow distribution, it has a wide lower quartile (lower whisker). This wide lower quartiles suggests that even a simple project involving Lego components require more unique blocks to create. Developing these projects is not as intuitive and therefore more difficult to create.

Figure 5 and accompanying Table 3 shows the plot of the number of unique blocks of projects within each group, as well as the distribution for all projects ("All" in Figure 5). We reiterate that 23% of projects fit into multiple groups, so one project can be represented in multiple plots. This is most evident in the outliers. In example, the greatest outlier is a password keeper app with 56 unique blocks in it; it is categorized as a Storage, Connect, and Media app because it has components of each of those types.

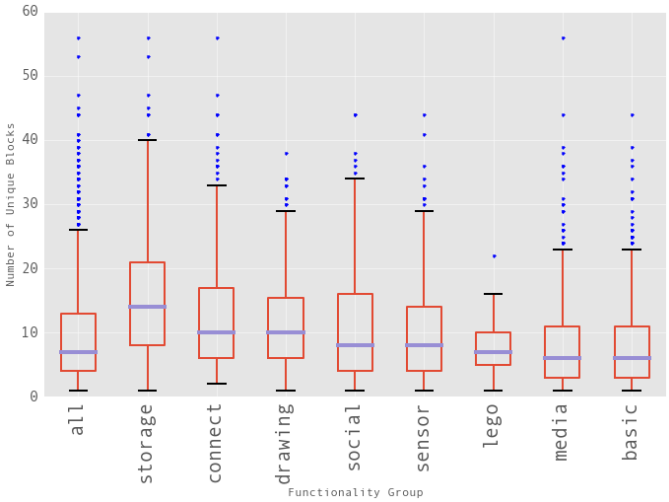

Fig. 5. Distribution of Number of Unique Blocks by Functionality Group

TABLE III. Summary Statistics for the Number of Unique Blocks by Group

	basic	lego	media	storage	connect	social	sensor	draw
mean	7.30	7.86	7.71	15.39	12.39	10.12	8.86	10.67
std. dev.	5.71	4.86	6.47	9.27	8.53	8.62	6.91	6.39
min	1	1	1	1	2	1	1	1
max	44	22	56	56	56	44	44	38

C. Multiplier Structures

To visualize the existence of multiplier structures in each group, we plot the proportion of projects within each group that use procedures, loops, and lists. The results are depicted in Figure 6.

When looking at all projects across all groups ("All" in Figure 6), we see that the proportion of projects that use

procedures and the proportion that use lists are approximately equal; the proportion of projects that use loops is about half compared to the other two. This relationship between proportion of multiplier structures does not hold true across all groups, suggesting that some multiplier structures are better suited for certain groups. We argue that the use of the loop and list structures vary by functionality of the app and therefore by group, but procedures tend to be more general in purpose.

Projects involving storage heavily use lists as temporary storage before moving information to a database or table as most projects categorized as storage tend to do.

Drawing, Sensor, and Media apps' functionalities tend not to require the use of loops. Drawing apps often involve a user interacting with a canvas, such as using their finger to draw a picture. Sensor apps often respond to a sensor on the phone, such as responding to the shaking of the phone. Media apps often involve the recording or playing of audio or video. All three groups' functionalities often do not require iterating, so the proportion of loops evident in those groups is lower.

Because procedures only organize blocks and enable other blocks to be called easily, we argue its use is more general across groups. Whereas lists and loops perform a specific function, the functionality of a procedure is defined by the user and is therefore more general. We explore this further in the Discussion section.

It is of note that the proportion of procedures existing in the Drawing group is abnormally high. Tutorials may in part explain this, as some popular beginning tutorials involve procedures, such as a "Whack-a-Mole" tutorial that uses a procedure to randomly move the mole the user is trying to whack. Users creating apps similar to these tutorials that contain procedures may only use the procedure in the manner described in the tutorial.

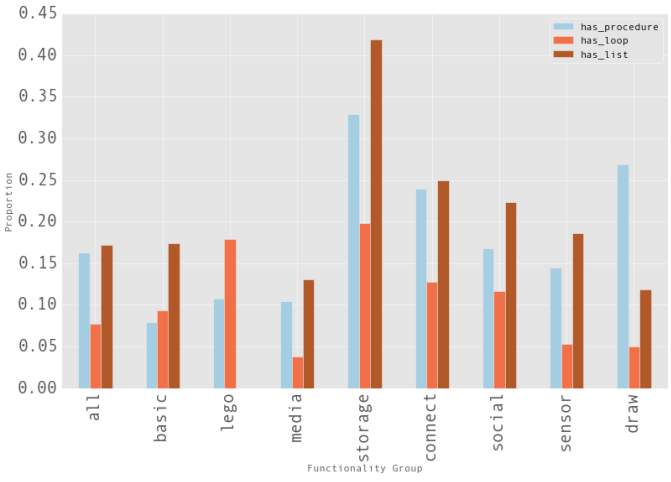

Fig. 6. Proportion of Multiplier Structures by Functionality Group

IV. Discussion

We critique our metrics of measuring intricacy and go on to analyze the intuitiveness of App Inventor to create different types of apps. We then tie this discussion on learnability back to App Inventor tutorials.

A. Analysis of Metric

When measuring the intricacy of projects, our challenge is in ensuring that the groups we categorize our projects in do not bias our metrics. That is, certain apps do not have greater or lesser intricacy just because they are in a certain group and perform a certain functionality. We argue that our metrics for the number of unique blocks and the proportion of projects in a group with procedures are our best metrics.

Because App Inventor features component-specific blocks, the number of unique blocks in a project is not directly dependent on its functionality. App inventor is event-driven, meaning the programming of App Inventor involves responding to an action, or event, relating to a component. This is made easy in App Inventor as each component has its own unique blocks to handle events, get and set attributes of the component, and call component functions. In Figure 7, we see that the button component has a block to handle it being pressed, the sound component has a block to play the sound, and the canvas has a block to change its color. Because App Inventor features blocks unique to each component, using one component instead of another does not inherently change the number of unique blocks in a project.

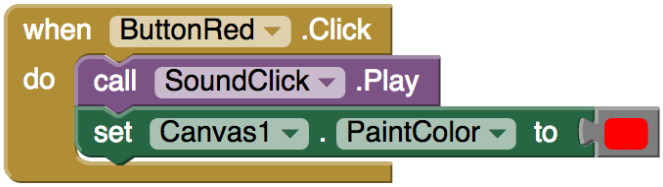

Fig. 7. Component-Specific Blocks. App Inventor features blocks that responding to events, getting/setting attributes, and component actions. For this reason, the number of unique blocks in a project is a suitable metric.

Whereas the proportion of loops and proportion of lists in projects can vary greatly depending on the functionality of it, the use of procedures are inherently not dependent on a certain functionality of the app. As mentioned in the results of the multiplier structures, we find that some functionalities rely more heavily on lists and loops than others. In example, drawing apps often have no use for loops but often use procedures to build reusable actions, such as erasing a drawing or resetting the score to a game. But the use of procedures is not dependent on functionality, as a procedure repeats whatever action the users defines when they create the procedure.

Because the number of unique blocks and proportion of projects with procedures do not depend on the functionality of projects, we focus primarily on these metrics in our analysis.

B. Learnability

A group is said to have high learnability if it does not require many different blocks to create a project. And if a group has high learnability, we expect many projects to be categorized into that group. We define a "learnability score" of a group as the ratio of the number of projects in the group divided by the the mean intricacy in Figure 8. We argue that the learnability score is influenced by the beginner tutorials.

It is not surprising that Basic apps have the greatest learnability score and are therefore the easiest to learn. This

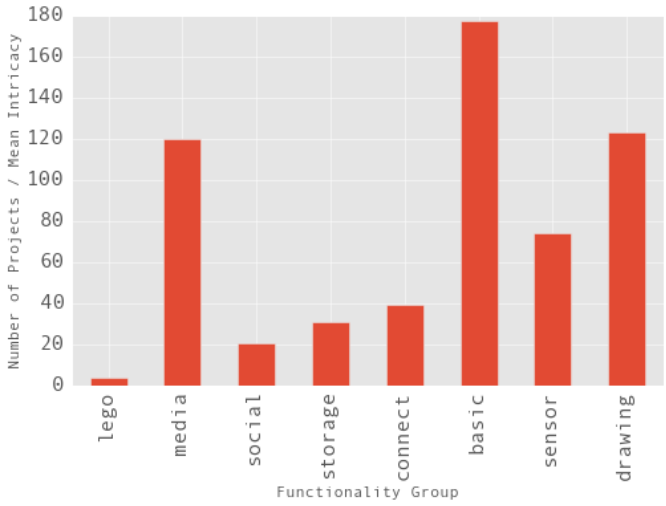

Fig. 8. Ratio of the Number of projects to mean intricacy, showing learnability of each group.

follows since we define the Basic group as only containing user interface components and no other types of components. The Drawing and Media groups also have high learnability scores. This is likely because the tutorials introduced earlier on create Drawing apps and Media apps. The linear nature of App Inventor's tutorials causes the learnability score to follow closely with the sequence of tutorials.

The close correlation with learning scores and the sequence of tutorials suggests that users build off of knowledge from only the tutorials they complete. On the App Inventor website, tutorials are displayed in linear order and have levels ranging from "Basic" to "Intermediate" to "Advanced". Table 4 shows what functionality groups tutorials fit into. Most users tend to start at the basic tutorials but the number of tutorials made until users creates their own original projects varies. It is safe to assume that the earlier a tutorial appears in this sequence, the more users will use it. We see Drawing, Media and Sensor apps appear as basic tutorials; they are also have the highest learnability score. There are no Lego apps, and the very low learnability score reflects that. And although there are several tutorials involving storage, they appear under the advanced section. The lower learnability score suggests both that these later tutorials are viewed less and that storage apps are more intricate.

TABLE IV. GROUPING OF APP INVENTOR TUTORIALS

Level	Functional Groups
Basic	Drawing(4), Media(2), Sensor(2), Basic
Intermediate	Drawing(5), Media(3), Storage, Social, Connect, Basic
Advanced	Storage(5), Sensor(4), Connect(2), Drawing(2)

From our learnability score, we see that users tend to build off of the knowledge they gain from tutorials found earlier. This suggests that from the tutorials, users learn to create a specific type of app but not enough for this knowledge to generalize to create apps of other types with App Inventor. This can be seen as both a need for more generalizable tutorials and evidence of the vast capabilities of App Inventor to develop apps with broad functionality.

V. CONCLUSION

We show that the number of unique blocks in a project as well as the proportion of projects that effectively use procedures both serve as sufficient measures of intricacy in the implementation of projects. By looking at the distribution of the number of unique blocks in projects of different groups, we are able to measure the capabilities for App inventor to develop apps of different functionalities. Our learning scores show that the learnability of specific functionalities of App Inventor closely relates to the functionalities in the basic, more popular tutorials. This suggests the amount of tutorials users complete are insufficient to provide them with enough generalizable knowledge of App Inventor to create original projects that are dissimilar from the tutorials they created.

We argue that breaking the sequentiality of the tutorials will better prepare users to deviate from tutorials and create more original apps. While we recognize that some functionality of App Inventor may inherently be more challenging than others, tutorials should still introduce more diverse components in the basic tutorials. The tutorials should also be displayed in such a way where users have a choice as to what to start with. We reiterate that our analysis is limited to the tutorials created by the MIT App Inventor team; the tutorials and curriculums made by other groups and individuals are out of the scope of this paper.

The existence of non-functional projects and recreations of tutorials in our dataset offer opportunities for future work. While we filtered out projects that were certainly non-functional by ensuring all projects had the minimum number of components and blocks, there still exist projects that do not have any functionality. Disregarding blocks that are not connected to other blocks and components with no programmed functionality would better filter out the non-functional projects. And while our findings suggest that the learnability score closely follows earlier tutorials, at least 20% of the projects analyzed are recreations of the tutorials. Analyzing projects that are truly original would result in more robust findings.

ACKNOWLEDGMENTS

This research is funded by the MIT EECS - Google Research and Innovation Scholarship.

We thank Jeffery Schiller (MIT App Inventor) for help collecting the data and Ilaria Liccardi (MIT) and Franklyn Turbak (Wellesley College) for helping guide the analysis.

REFERENCES

[1] Google. Blockly, 2015. URL https://developers.google.com/blockly/. Online; last accessed 10-April-2015.

[2] MIT App Inventor. Mit app inventor — explore mit app inventor, 2015. URL http://appinventor.mit.edu/explore. Online; last accessed 14-May-2015.

[3] MIT App Inventor. Tutorials for app inventor 2, 2015. URL http://a,ppinventor.mit.edu/explore/ai2/tutorialsd41d.html. Online; last accessed 14-May-2015.

[4] Johanna Okerlund and Franklyn Turbak. A preliminary analysis of app inventor blocks programs, 2013. URL http://cs.wellesley.edu/~tinkerblocks/VLHCC13-poster.pdf. Poster presented at the IEEE Symposium on Visual Languages and Human-Centric Computing (VLHCC 2013), Sept 17, San Jose, CA.

[5] Franklyn Turbak. Mobile computational thinking in appinventor 2, 2014. URL http://cs.wellesley.edu/~tinkerblocks/RIC14-talk.pdf. Slides of a talk given to CSTA-RI,April 10, 2014, Rhode Island College, RI.

[6] Franklyn Turbak, David Wolber, and Paul Medlock-Walton. The design of naming features in app inventor 2. *2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2014. doi: 10.1109/vlhcc.2014.6883034. URL http://dx.doi.org/10.1109/vlhcc.2014.6883034. Online; last accessed 29-April-2015.

[7] DWF Van Krevelen and R Poelman. A survey of augmented reality technologies, applications and limitations. *International Journal of Virtual Reality*, 9(2):1, 2010.