Identifying original projects in App Inventor

Eni Mustafaraj and **Franklyn Turbak** and **Maja Svanberg**
Department of Computer Science
Wellesley College, Wellesley, MA
emustafa, fturbak, msvanberg@wellesley.edu

Abstract

Millions of users use online, open-ended blocks programming environments like App Inventor to learn how to program and to build personally meaningful programs and apps. As part of understanding the computational thinking concepts being learned by these users, we want to distinguish original projects that they create from unoriginal ones that arise from learning activities like tutorials and exercises. Given all the projects of students taking an App Inventor course, we describe how to automatically classify them as original vs. unoriginal using a hierarchical clustering technique. Although our current analysis focuses only on a small group of users (16 students taking a course in our institution) and their 902 projects, our findings establish a foundation for extending this analysis to larger groups of users.

Introduction: Distinguishing Original and Unoriginal Blocks Programs

Blocks programming environments, in which programs are constructed by connecting fragments shaped like puzzle pieces, are increasingly becoming a way that beginners are exposed to programming and that hobbyists, scientists, and other "casual" programmers write programs. Examples of these environments include App Inventor, Scratch, Blockly, Snap!, Pencil Code, and Alice. Through courses and extracurricular activities like Code.org's *Hour of Code*, blocks programming environments have become a popular way to introduce programming and computational thinking to tens of millions of people of all ages and backgrounds.

By lowering various barriers, blocks environments facilitate "making stuff" involving computation, without starting with traditional text-based languages (Bau et al. 2017). For example, Scratch enables computational newbies to create animations and games (Resnick et al. 2009), and App Inventor empowers beginners to create their own mobile apps for Android devices (Wolber, Abelson, and Friedman 2015).

The first two authors have used App Inventor to teach app development, computational thinking, and introductory programming since 2009 in the context of classes at our institution and a variety of workshops for both faculty and students. App Inventor has powerful programming mechanisms (e.g., event handlers, lists, loops, and procedures) for controlling key features of mobile devices (camera, location sensor, speech recognizer, voice recorder, local and web-based databases, etc.). But in our experience, users often have trouble making apps that work as desired. Their apps contain many bugs, and even aspects that work are sometimes implemented in overly complex and roundabout ways.

We are currently analyzing large datasets of App Inventor projects to see which anecdotal observations are borne out in the data. Our long-term goal is to use learning analytics to identify difficulties encountered by blocks programmers and to alleviate these difficulties by improving the programming environments and their associated pedagogies.

The open-ended nature of App Inventor and lack of information about its users presents many challenges for our research. App Inventor collects no demographic data on users other than what is provided in an optional survey completed by only a small percentage of users. For most users, we have no information on their gender, age, geographic location, programming background, etc. We also don't know whether any of their projects were created in collaboration with others, or as part of a class or other coordinated activity.

One of the thorniest problems we have encountered is distinguishing original from unoriginal projects. While learning App Inventor, users often create (or simply upload) many projects that are nearly identical to *global tutorials* from a few popular App Inventor websites or what we will call *local examples* = tutorials, exercises, and other constrained activities done in the context of classes, MOOCs, hackathons, etc. We consider these projects to be *unoriginal* because users are either following detailed instructions or solving problems in highly constrained contexts, neither of which illustrates what users can design and build on their own or in groups. In contrast, *original* projects are those in which users develop an app based on their own ideas and current programming skills. If we want to understand what App Inventor users are learning about programming and what misconceptions they have, we need to focus on their original projects and filter out the unoriginal ones.

We have tried out various techniques to identify which projects are minor variations of global tutorials found on App Inventor websites. Our preliminary results indicate that attempting to identify tutorials by project names is inaccu-

rate; instead, structural comparisons with known tutorials based on project content is needed.

The problem of how to detect *local examples* is more vexing since they are, by definition, local to a class, and at the very least require knowing which students are taking a class together. As described later, we have developed a way to automatically discover in our dataset collections of users who appear to be groups of students taking the same class.

In this paper, we investigate this research question: **given a collection of all App Inventor projects from students in a class, can we automatically classify such projects as either being *unoriginal* (i.e., based on global tutorials or local examples) or *original* (i.e., projects that students created on their own or in small groups)?**

We studied this problem using the 902 projects created by all 16 students in a CS0 App Inventor course taught by the second author at our institution in Fall, 2015. We manually labeled the 902 projects as being global tutorials, local examples, or original projects. We then developed an algorithm that (1) represents App Inventor projects as feature vectors and (2) uses hierarchical clustering based on similarities between feature vectors to automatically classify projects as unoriginal or original. The classification determined by our algorithm matches our ground truth labels to a high degree of accuracy, and the clusters found by the algorithm align well with particular global tutorials and local examples.

Related Work

The work most closely related to ours is the automated analysis of App Inventor projects by (Xie, Shabir, and Abelson 2015; Xie and Abelson 2016). They automatically extract project summaries including types of components and blocks from project datasets to deduce skills and concepts users are learning over time. In (Xie, Shabir, and Abelson 2015), they filter out projects classified as tutorials, but this is determined only by the *names* of the projects, not their *contents*. This work inspired us to find more accurate ways to identify unoriginal projects.

Classifying students or their programs is a topic that has interested the research community for years, although not with the same goals as our current research. Recently, Piech et al. used clustering of snapshots from the same assignment to build a model that captured the different stages a student goes through while learning to program (Piech et al. 2012). Their analysis used a fine-grained data collection process that currently doesn't exist in App Inventor, but might become a reality in a near future (Sherman and Martin 2015).

App Inventor: An Overview

MIT App Inventor is a browser-based blocks programming environment for creating mobile apps for Android devices. It is popular: over 4.5 million registered users have created nearly 17 million app projects, and there are over 340 thousand active monthly users. It has a broader age distribution of users than Scratch, and the range of compelling mobile apps is so large that everyone can find classes of apps to be passionate about. Some users are drawn to App Inventor by the desire to learn programming or the lure of entrepreneurship (App Inventor apps can be sold in the Google Play Store). But many just want to make an app for themselves, their family and friends, or their community.

One of the first apps created by many App Inventor users is `TalkToMe` (Figure 1), the subject of the first two video tutorials for beginners at the MIT App Inventor website[1]. This app has two behaviors: (1) when a button is pressed, it will speak aloud whatever text has been entered into a text box; and (2) when the the device is shaken, it will speak the phrase "Stop Shaking me!"

The app is specified in two parts within an App Inventor browser window. In the Designer window (part of which is shown in Figure 1(a)), the user drags and drops the components of an app from a palette into the representation of the screen. Components include visible user interface elements, such as (in this case): a `TextBox1` component entering the text to be spoken and a `Button1` component that causes the text to be spoken when pressed. Components can also be object-like entities that add functionality to the app, in this case: (1) a `TextToSpeech1` component that "speaks" a string; and (2) an `AccelerometerSensor1` component that can detect when the device is being shaken.

The behavior of an app is specified in the Blocks Editor (Figure 1(b)). In this window, the user drags blocks from a blocks palette organized by functionality into a 2D workspace and connects them. Some blocks (such as `when Button1.Click`) are *event handlers* that are triggered by an event involving the component. Other blocks include *methods* that operate on a component (such as `call TextToSpeech1.Speak`), component property getters (such as `TextBox1.Text`) and setters, and *built-in blocks* like the string block for "Stop Shaking me!".

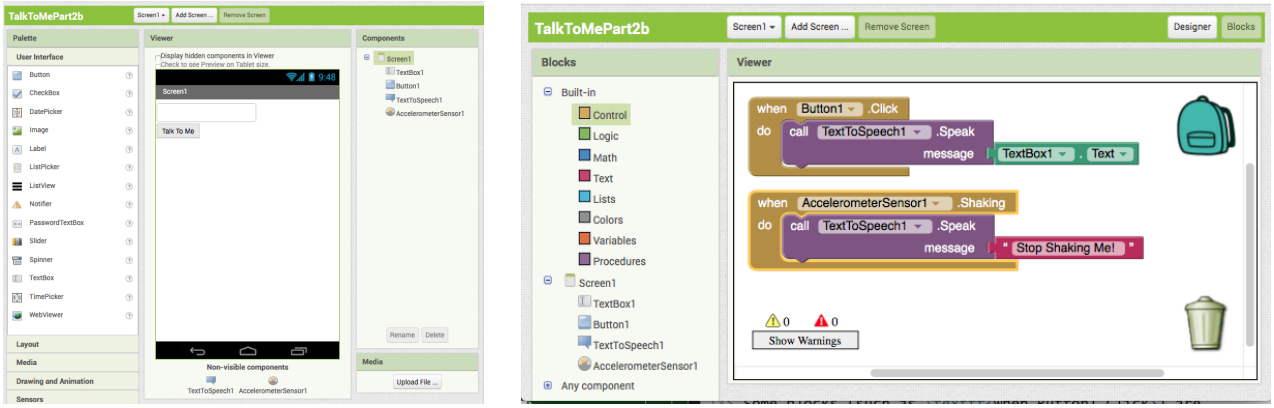
The capabilities of App Inventor go far beyond this simple example. For example, App Inventor apps can take and display pictures and videos, record and play sounds, send and receive text messages, make phone calls, scan barcodes, use GPS location data, and access, store, and process information saved persistently on the device or on the web.

App Inventor project source code files are automatically saved in the cloud, but can be exported as `.aia` files. Each of these is a zipped collection of files that includes a project property file, media files used by the app, and, for each screen in the app, a JSON representation of its components and an XML representation of its blocks.

Data and Labeling

Our project dataset is 902 App Inventor `.aia` project source files created by all 16 students taking the second author's Wellesley course *CS117 Inventing Mobile Apps* in Fall 2015. These files, provided to us by the MIT App Inventor team (with consent of the students) include additional timestamp information indicating (1) when the project was created and (2) when it was last modified. During the course students worked in pairs on various pair programming projects, and sometimes created new gmail accounts for this work; unfortunately, we do not not have access to the projects created in these other accounts. On average, the

[1] `http:appinventor.mit.edu`



(a) Designer (b) Blocks Editor

Figure 1: App Inventor `TalkToMe` tutorial app

students created 56.4 projects (min=41, max=72, std=8.2), within the time period August 31 to December 23, 2015.

Our goal was to classify the student projects as being unoriginal (closely related to global tutorials or local examples) or original (all other projects, including nontrivial extensions to global tutorials or local examples).

We began by collecting a set of 80 *global tutorial apps* from pedagogical websites for App Inventor. This includes video and written tutorials from `appinventor.mit.edu` and `appinventor.org`, including a set of 14 so-called Maker Cards from `appinventor.mit.edu` that were distributed to students in the course. For each tutorial, we downloaded or created an App Inventor project file. In some cases, where a tutorial had multiple parts or suggested extensions, we created a separate project file for each such part/extension. For example, the `TalkToMe` tutorial has two videos, and the second video has two extensions to the app created in the first video, so we named these projects `TalkToMePart1`, `TalkToMePart2a`, and `TalkToMePart2b`.

We also collected projects for all the tutorials, exercises, and example App Inventor programs that were presented in the CS117 class. These are the *local example apps*; there were 46 of them.

Since the second author was familiar with all course materials and student work, he manually examined all 902 projects, comparing them to the global tutorial and local example apps. Each student project that closely matched a global tutorial or local example was labeled GLOBAL or LOCAL along with which tutorial/example project it matched. All 46 local tutorials appeared in the student dataset, but only 40 of the 80 global tutorials appeared. In some cases, a student project was deemed to be a nontrivial extension to a tutorial or example, and was labeled BEYOND a particular tutorial or example. A few (26) of the projects contained zero blocks (they only had the layout of the app, not its behavior); these were labeled EMPTY and removed from further consideration.

Most remaining projects were related to one of five creative projects students were assigned. These increased in complexity from the first assignment, in which students built a very simple app using a few randomly dealt components, to the open-ended final assignment, in which students designed and built from scratch an app of their choosing that had to incorporate a web database. These projects were labeled COURSE PROJECT n, with n in the range 1 to 5.

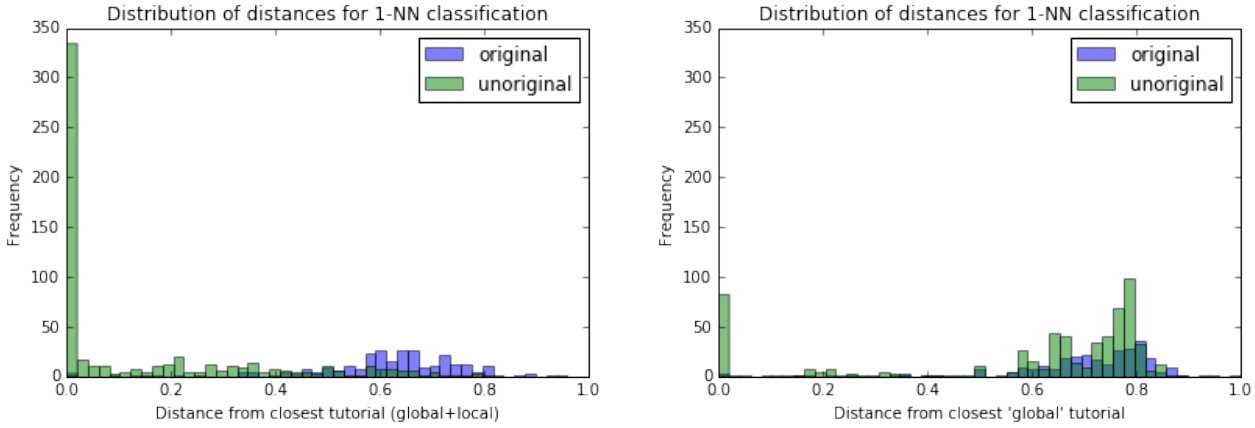
Finally, the small number of remaining projects were labeled as TESTING projects, because they tended to test certain components, often ones not covered in class.

In terms of originality, we consider projects labeled GLOBAL or LOCAL to be **UNORIGINAL** and those labeled CLASS PROJECT, TESTING, or BEYOND to be **ORIGINAL**. Of the 876 nonempty projects in the dataset, only 280 (32%) were labeled as **ORIGINAL**; the remaining 596 (68%) were **UNORIGINAL**. The fact that a large majority of projects in the dataset were **UNORIGINAL** underscores the need to filter them out when analyzing student work for understanding, misconceptions, etc.

Feature Representation and Distance

We developed a Python program to summarize each `.aia` source file as a JSON file containing, for each screen in the app, the types and frequencies of the components and blocks used in the screen. As shown in Figure 1(b), the behavior of an app is specified by blocks. For blocks that operate on a component (e.g., event handlers, methods, and component property getters and setters), we only distinguish blocks by the type of the component and not the particular component name. For example, an app with a `.Click` handler for a `StartButton` and `StopButton` is considered to have two occurrences of the `Button.Click` block. But the property getter that extracts the current string contents of a label (`Label.Text`) is considered different from the one that returns the string in a text box (`TextBox.Text`). Blocks that specify values like numbers or strings are distinguished only by their types and not their values. While there are more than one thousand distinct block types in App Inventor, our dataset of 902 student projects, 40 global tutorials, and 46 local examples uses only 347 different block types.

In a program, the order and nested organization of blocks



(a) (b)

Figure 2: Distribution of distances using 1-NN classification of student projects relative to (a) both global tutorials and local examples and (b) only global tutorials. Without knowledge of local examples, many unoriginal projects would appear to be original based just on their 1-NN distance.

matters, just like the order of words gives meaning to a document. However, relying on the successful bag-of-words model that discards order to focus on presence and frequency of words in a document, we experimented with a unigram model in which all block types are independent. Our experiments with various combinations of features for determining similarity between App Inventor projects showed that representing a project as a vector of the 347 block type frequencies (disregarding components) was adequate for our needs. Because of this decision, two projects with the same blocks that differ only in details of their user interface are considered the same. Intuitively, the originality of a project is captured by the kind and number of blocks that are not common in unoriginal projects.

Our sample of 988 projects (902 from students, 40 global tutorials, and 46 local examples) has an average of 65.9 total blocks per project (median=49, std=68.2, min=0, max=557). For unique block types, the statistics are: 20.3 unique blocks in average (median=19, std=13.2, min=0, max=71). So, on average, 20 of the 347 slots in a project feature vector are non-zero.

To determine distance between project feature vectors, we experimented with various metrics, including Euclidean and Jaccard. Jaccard *similarity* determines the ratio of the intersection of features divided by the union of features. Jaccard *distance* is 1 minus the similarity. So entities that are the same have a Jaccard distance of 0, while those with nothing in common have Jaccard distance of 1. We found that the Jaccard distance metric worked well on projects, and it worked best when the block type feature vector used frequency counts rather than binary values (where a `1` indicates at least one block of a given type). However, the `jaccard` option in the Python `scipy` library `spatial.distance.pdist` did not correctly handle frequency counts, so we had to supply a function for correctly computing it.

In our following presentation, it is assumed that projects are represented as feature vectors based on block types with counts, and distances are calculated via the Jaccard metric.

Classification, Clustering, or Both?

Our first approach for distinguishing between original and unoriginal projects was to apply the k-NN classification algorithm with k=1. Given that the set of tutorials is known, we can calculate the distance of every project to these known tutorials and then assign the label of the closest tutorial to a project. The problem with this approach is that by having only one labeled example per class, it's very difficult to establish the hyperspace boundaries between the different classes (with each tutorial being a class on its own). It also depends on the set of known tutorials.

For most App Inventor users, we know only the global tutorials. For our particular experiment, we also happen to know the 46 example projects that were local to the class. This knowledge makes a huge difference. For our dataset with 86 known tutorials (40 global and 46 local), the distributions of distances for original and unoriginal projects to the closet tutorials are mostly distinct, as shown by the histogram in Figure 2(a). However, if we repeat the same 1-NN classification, but only using the set of the known global tutorials, the two classes of original and unoriginal projects are all in the same space, making it hard to distinguish between the two categories (Figure 2(b)).

When classifying student projects by distance between known tutorials, we miss an important source of information: the similarity of projects to other projects. Even if we don't have a priori knowledge of the local examples used in class, when many projects belonging to different students are close together, this strongly suggests that they are unoriginal near-copies of global tutorials or local examples.

This can be revealed through clustering. Consider the dendrogram in Figure 3 generated by applying hierarchical clustering to a subset of (1) 58 projects chosen because their names resembled the names of some global tutorials and (2) 7 of the actual tutorials. In this case, hierarchical clus-

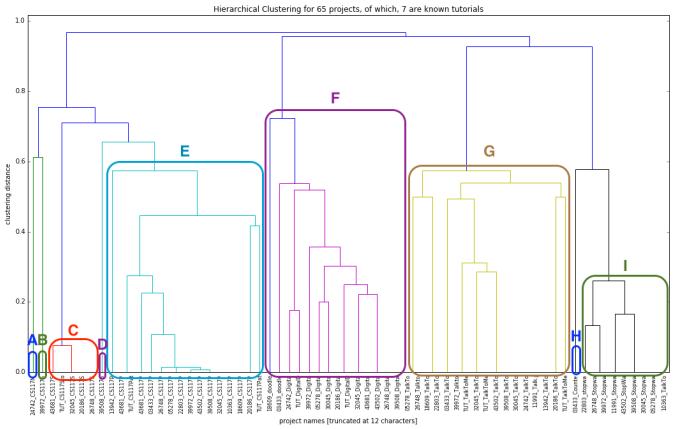


Figure 3: Annotated dendrogram showing the clustering of 58 student projects and 7 known tutorial projects.

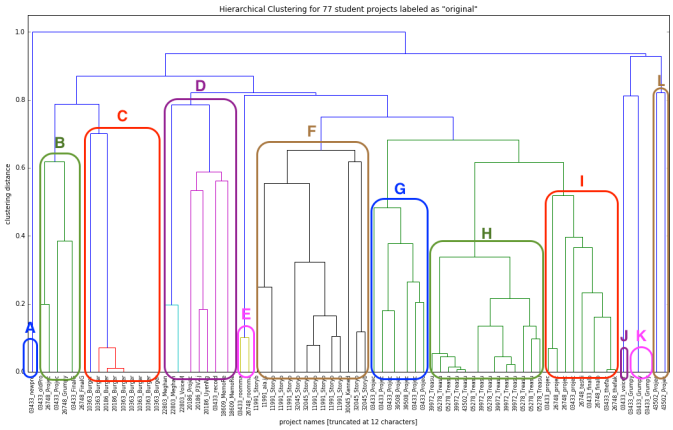


Figure 4: Annotated dendrogram showing the clustering of 77 student projects labeled as **ORIGINAL**.

tering does a remarkable job of clustering projects that we had labeled in the same way. We have annotated the diagram with clusters A through I for expository purposes. Each such annotated cluster corresponds to a labeling for a particular global tutorial or local example. For example, cluster I corresponds to the Maker Card `Stopwatch` tutorial, and even includes the `TalkToMe` project of user 10363, which is poorly named and is in fact really a stopwatch project. Cluster H corresponds to the Maker Card `Counter` tutorial, which has a subset of blocks in the `Stopwatch` tutorial. Cluster C corresponds to the tutorial for a much more accurate local stopwatch tutorial presented to students when they were working on their final projects. Cluster E corresponds to the starter and solution files for a local GPS path tracking tutorial presented in class. Clusters F and G are, respectively, many versions of the global `DigitalDoodle` and `TalkToMe` tutorials. The two leftmost projects in cluster F were labeled as nontrivial extensions to `DigitalDoodle`, so it makes sense that they join the cluster as singletons high in the tree.

Each of the non-singleton annotated clusters contains projects from several students in the class, making the clusters reveal their nature as a learning activity as opposed to an original individual or pair project.

From observing the clustering of student projects labeled as **ORIGINAL** (see dendrogram in Figure 4), we can gain more intuition about how a hybrid process of classification and clustering could work. We have annotated the dendrogram with lettered clusters for the sake of exposition. Clusters A, B, and K correspond to different versions of the same original game project done by a pair of students. Clusters C, E, F, G, H, I, J, and L correspond to different individual and pair projects. Cluster F has 13 projects created by students 05278 and 39972, but there is also one copy of their project in the folder of student 43502 that apparently was shared with this other student. The student partners have a balanced number of iterative versions of the projects (7 and 6). Meanwhile, Cluster H has 14 projects created mostly by the two students 11991 and 32045, with one interjection by student 30045. Differently from Cluster F, one of the students in this cluster has generated more versions (9 versus 4). These examples suggest that clusters of many similar projects created mostly by a small number of users can be automatically classified as original projects, because they indicate sequential progress on a project. Since App Inventor doesn't have version control, users save versions manually, which indicates how a project progressed over time.

Cluster D is interesting because it includes the work of four students from the third original project in the course, which was an individual project. These students did not work together, but their projects are lumped together because this particular original project was much more constrained than the others: students were required to implement an app that recorded voice memos and had an interface for saving, displaying, playing, and deleting the voice memos. While the apps had different interfaces and numbers of screens, they ended up using many similar blocks and so were clustered together by our Jaccard distance metric that emphasizes block similarity. Because cluster D contains the projects of many students, our approach would incorrectly classify it as an unoriginal activity as opposed to a collection of similar original individual projects that it really is.

A Hybrid Approach: Classification through Clustering

A project is unoriginal if it is very similar to one of the existing global or local tutorials. However, since often not all these tutorials are known a priori, a project will be deemed unoriginal if it is similar to projects from a large number of different users. Thus, once a relatively tight cluster of projects from different users has been created, all of the projects can be classified as **UNORIGINAL**.

A project is original if it is not similar to any cluster of known unoriginal projects. Such a project might often occur within a cluster of other original projects. Thus, if we discover a cluster of many projects contributed by a small number of users, we can classify all of them as **ORIGINAL**.

What makes this classification interesting and different

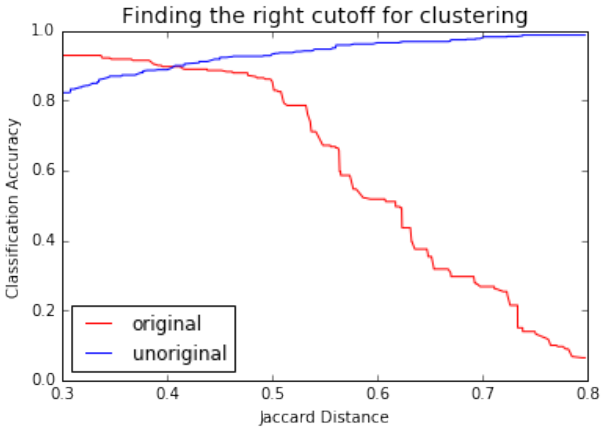


Figure 5: The accuracy for the **ORIGINAL** class is highest for small Jaccard distances and declines as this distance increases, because **ORIGINAL** projects get merged into clusters where the majority is composed of **UNORIGINAL** projects.

from other classification problems is the fact that it relies on meta-information outside the projects: the users, number of users in a cluster, and similarity to other projects.

We implemented this hybrid two-step clustering and classification as follows: we performed hierarchical clustering using the generalized Jaccard distance metric and the `average` linkage method. From our visual inspection of several dendrograms depicting subsets of the data, we had noticed that unoriginal projects tend to cluster together at small distance values, while original projects were merged only at large distances. We decided that all singleton projects with a distance higher than 0.8 from all other projects are **ORIGINAL**. Then, flattening the dendrogram at different cutoff distances between 0.3 and 0.8 we applied (appropriately) the classifications **UNORIGINAL** or **ORIGINAL** to clusters based on the rules described above.

Results of Classification

Given that we know the real labels, we can estimate the accuracy of such an automatic classification at different distance levels. Our labeled dataset is unbalanced: 280 original projects and 596 unoriginal projects. Given that we care more about correctly predicting the **ORIGINAL** class, we will calculate the accuracy for each label separately. The results are represented by the two lines in Figure 5. At the distance 0.4, the two accuracies are both at 89%. Further increasing the distance means that some singleton, original projects start getting merged into clusters where the majority is **UNORIGINAL**. This result indicates that by cutting off the hierarchical clustering at distance 0.4, we get a high accuracy of labeling for both classes (89%). In future work, we will consider different ways of performing this process that takes into account other metadata about the projects, for example, their timestamps.

Ongoing and Future Work

This work is based on data from students in a known class. Is it possible for us to automatically discover other classes of students in our App Inventor datasets? In other work (in progress), we are exploring ways to discover which App Inventor users are likely to be students taking a class together using creation timestamps for their projects. This is based on the observation that two students in the same class will be more likely to create projects at the same time (as part of classroom activities) than two arbitrarily chosen users. Clustering users based on this idea, we have been able to discover candidate classes of students, including the exact members of Wellesley's Fall 2015 CS117 App Inventor course.

In other work, we are also: identifying projects "in the wild" as being minor variations of particular global tutorials; determining which user projects appear to be a sequence of versions of a single original app project; and investigating the components, blocks, and programming patterns that users typically employ in their original projects.

One application of this line of work would be creating a teacher dashboard for App Inventor instructors. Automatically classifying all student projects as global tutorials, local examples, and original projects would help teachers better understand the participation and progress of their students. And visualizing the concepts and misconceptions in the original projects would provide detailed information for highlighting areas in which students might need guidance.

Acknowledgments

This work was supported by Wellesley College Faculty Grants and by NSF grant DUE-1226216.

References

Bau, D.; Gray, J.; Kelleher, C.; Sheldon, J. S.; and Turbak, F. 2017. Learnable programming: Blocks and beyond. *Communications of the ACM*. To appear.

Piech, C.; Sahami, M.; Koller, D.; Cooper, S.; and Blikstein, P. 2012. Modeling how students learn to program. In *43rd ACM Technical Symposium on Computer Science Education*, 153–160. ACM.

Resnick, M.; Maloney, J.; Monroy-Hernández, A.; Rusk, N.; Eastmond, E.; Brennan, K.; Millner, A.; Rosenbaum, E.; Silver, J.; and Silverman, B. 2009. Scratch: programming for all. *Communications of the ACM* 52(11):60–67.

Sherman, M., and Martin, F. 2015. Learning analytics for the assessment of interaction with App Inventor. In *IEEE Blocks and Beyond Workshop*, 13–14.

Wolber, D.; Abelson, H.; and Friedman, M. 2015. Democratizing computing with App Inventor. *GetMobile: Mobile Computing and Communications* 18(4):53–58.

Xie, B., and Abelson, H. 2016. Skill progression in MIT App Inventor. In *IEEE Symposium on Visual Languages and Human-Centric Computing*, 213–217.

Xie, B.; Shabir, I.; and Abelson, H. 2015. Measuring the usability and capability of App Inventor to create mobile applications. In *3rd International Workshop on Programming for Mobile and Touch*, 1–8.