Identifying Common Errors in Open-Ended Machine Learning Projects

James Skripchuk
North Carolina State University
Department of Computer Science
Raleigh, North Carolina, USA
jmskripc@ncsu.edu

Yang Shi
North Carolina State University
Department of Computer Science
Raleigh, North Carolina, USA
yshi26@ncsu.edu

Thomas Price
North Carolina State University
Department of Computer Science
Raleigh, North Carolina, USA
twprice@ncsu.edu

ABSTRACT

Machine learning (ML) is one of the fastest growing subfields in Computer Science, and it is important to identify ways to improve ML education. A key way to do so is by understanding the common errors that students make when writing ML programs, so they can be addressed. Prior work investigating ML errors has focused on an instructor perspective, but has not looked at student programming artifacts, such as projects and code submissions to understand how these errors occur and which are most common. To address this, we qualitatively coded over 2,500 cells of code from 19 final team projects (63 students) in an upper-division machine learning course. By isolating and codifying common errors and misconceptions across projects, we can identify what ML errors students struggle with. In our results, we found that library usage, hyperparameter tuning, and misusing test data were among the most common errors, and we give examples of how and when they occur. We then provide suggestions on why these misconceptions may occur, and how instructors and software designers can possibly mitigate these errors.

CCS CONCEPTS

• **Social and professional topics → Computing education**; • **Computing methodologies → Machine learning**.

KEYWORDS

Computer science education, Machine learning education, Data science

ACM Reference Format:
James Skripchuk, Yang Shi, and Thomas Price. 2022. Identifying Common Errors in Open-Ended Machine Learning Projects. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE 2022), March 3–5, 2022, Providence, RI, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3478431.3499397

1 INTRODUCTION

Machine Learning (ML) Education Research is a growing facet of the established field of CS Education Research. ML Education is distinct because it combines traditional CS skills such as programming with specific technical knowledge about how to build and apply ML models, along with the use of advanced APIs and knowledge from statistics on how to analyze and interpret data.

The demand for ML engineers has grown significantly over the past decade, and educational institutions are investing heavily in ML faculty and curricula [14]. However, the rapid growth of ML also requires the pedagogy and knowledge behind ML education to grow as well. Mistakes in ML models and improper use of biased data can create disparate outcomes for end-users of ML systems, with effects ranging from social injustices [5, 7] to medical misdiagnosis [18]. In order to improve ML pedagogy, educators and researchers need to better understand how students *learn ML*, in particular what errors they frequently make and in what contexts they make these errors. This understanding can help us develop better instruction, assignments, assessments, and tools to support students. For example, if many students make the same specific mistakes when selecting hyperparameters, such as using hand-picked values without performing hyperparameter tuning, instructors can address this through examples, pointers to appropriate API calls, and explicit feedback on inappropriate design choices.

However, most existing ML Education research has been limited to the *instructor's* perspective, largely consisting of interviews and surveys asking practicing instructors about what troubles they perceive students to have [13, 21]. While this prior work gives important insight into high-level challenges that *instructors are already aware of*, it is unclear how well these perceptions reflect the mistakes students commonly make, and how these mistakes affect the validity of students' ML projects. Prior work in CS education shows that instructor's beliefs about what students struggle with during programming do *not* always match empirical data on student errors [2]. Therefore, concrete analysis of students' programming artifacts is essential for researchers to improve ML curricula and build support tools.

To address this, we manually analyzed over 28,000 lines of student code from open-ended ML course projects in order to identify *what* errors students are making and *how frequently they occur.* We use "error" as an overarching term, referring not only to compilation and programming errors, but also to misconceptions regarding ML models, data analytics, and stylistic errors. We focus on an open-ended final project, where students had to independently construct a full ML pipeline, showcasing their applied knowledge of ML. Our research question is as follows:

RQ: What are the most common errors students make when programming an open-ended ML project?

We collected 19 final group project submissions from undergraduate and graduate students in an introductory ML class and inductively coded their programs to develop an inventory of common errors that occurred across projects. We identified 18 distinct types of errors that were present in student projects, as well as 3 stylistic or efficiency issues, which may be a source of future errors in code [8]. We report the frequency at which these errors occur and present examples of common errors pulled from students' code. We suggest reasons these misconceptions may occur and how they can be addressed. After this, we suggest implications of the results for ML education and research.

2 RELATED WORK

2.1 Machine Learning Education

Due to the newness of ML Education Research, the current research in this topic is sparse. Shapiro et al. put forth a research agenda on what questions ML education researchers should be asking and suggestions for how they could go about this research [19]. While these are high level questions, we put forth our research to serve as a concrete base to build more elaborate and in-depth research studies.

Sulmot et al., in an effort to establish ML Pedagogical Content Knowledge (PCK), performed a series of interviews with various ML instructors who were teaching non-majors [20, 21]. In doing so, they organized common ML learning goals into the Structure of Observed Learning Outcomes (SOLO) taxonomy, a framework for mapping concepts to a student's level of understanding. Goals at base of the SOLO taxonomy require understanding of a single relevant aspect of a topic (e.g. describing a single ML algorithm), while the higher goals within the taxonomy involve relating aspects to each other (e.g. model evaluation) and generalizing knowledge (e.g. developing a model from scratch to solve a problem). Their results suggested that the higher a learning goal was within the SOLO taxonomy, the more instructors stated that the topic was difficult to teach and for students to understand. Another main result they discovered was that, at least in the opinion of the instructors, "student's struggle less with actual algorithms and more with the design decisions around them". In this paper we verify these results through our analysis of students' submissions of an open-ended final project, where these "design decisions" are key to success, and fully determined by the student.

Kross and Guo performed interviews with data scientists who "teach in a diverse variety of settings across industry and academia" [13]. While covering a host of topics such as student motivations and communication, a prominent result pertinent to this study is how professors teach data-analytic workflows. In these workflows, instructors "emphasized teaching as a more disciplined data-analytic workflow" and "provide [students] the skill to write more robust and reproducible scientific code". On this point, Kross and Guo discuss library usage, the importance of teaching data-analytic programming (such as dataframes), and the need for authentic tools and learning environments.

An overall theme with previous ML education work is that it is *not grounded in student data*, and is focused from an instructor perspective. In addition to this, prior ML education research focuses on how ML is taught to non-majors. While this is important, our work focuses on a population of CS majors, which have their own associated challenges and a higher technical standard.

2.2 Misconception Identification

Prior work in CS Education has focused on identifying student misconceptions and errors by analyzing student artifacts. These studies have focused on a variety of topics ranging from CS1[3] to Algorithms [6], but no prior work has done so in a ML course. By creating a concrete list of misconceptions, this research not only helps to provide a general understanding of common student behaviors, it also serves as a basis for many types of future research. For example, by closely understanding misconceptions, it can be used to build a concept inventory (CIs) - a multiple choice diagnostic assessment. By mapping known misconceptions to the wrong answers on a question, a CI allows instructors to not only see what misconceptions students have, but also to measure learning gains with a pre- and post- test. With clear definitions, it is also possible to create tools to automatically detect misconceptions within student code, allowing for either real-time support or post-hoc analysis of student behavior [16].

3 METHODOLOGY

Our goal in this work was to identify and describe common errors students make when programming in open-ended ML projects. To do so, we collected data from a ML course and manually analyzed the students' entire project code, to identify and quantify the ML errors that occurred.

3.1 Context

Data was collected from final project submissions from a single course at a R1 university on the east coast. The course in question was an upper level machine learning and data analytics course that was cross-listed as both an upper-level undergraduate and graduate course. The course included 122 students (64 undergraduate; 58 graduate), of whom 63 consented to data collection. This course was primarily designed for CS majors and minors, though a small number of non-CS graduate students were also enrolled. Demographic data was not collected.

The course content focused on supervised and unsupervised learning, the ML pipeline (data, preprocessing, feature transformation, model training, hyperparameter tuning, model evaluation), and specific ML models and approaches. These included decision trees, k-NN, Bayesian networks, ensemble methods, regression, multilayer perceptron networks, support vector machines, as well as various clustering methods (k-means, hierarchical, DBSCAN). The lecture content focused on these ML concepts from a theoretical perspective. and only briefly introduced ML *programming* in one introductory lecture.

Students learned ML programming and APIs and through 4 group homework assignments (2-3 students per group). Homeworks consisted of theoretical questions as well as practical programming assignments in Python. Programming problems asked students to apply processing and ML approaches (e.g. transformation, feature selection, model training, evaluation) to simple datasets using existing functions from the scikit-learn (`sklearn`) machine learning library. Students were also asked to implement some classical

ML algorithms by hand (e.g. K-means, evaluation metrics). The homeworks were written as scaffolded Jupyter Notebook files, the industry standard for ML computing. The homeworks often included example code to introduce new API functions and linked to the documentation for more information. They included unit tests (both public and hidden) that would automatically grade most problems, so students knew when they had completed the problems correctly. Students were assessed using homeworks and exams, as well as a final project (our main object of study).

3.2 Final Project

The final project was an open-ended machine learning project, where the instructor required students to "explore an interesting data mining problem of your choice in the context of a real-world data set". Projects were evaluated on the novelty of the problem formulated, the rigor and breadth of experimentation, and the clarity of the submitted final report. Students were allowed to work in groups of three or four, or individually. Graduate students were not allowed to work with undergraduate students, and were given more extensive project requirements. Students were encouraged, but not required, to use Jupyter notebooks for their final project.

The project covered the second half of the course (8 weeks) and was broken down into 3 milestones: a proposal (at 2 weeks), a midway report (at 4 weeks) and a final report. At each phase, students submitted a written report in academic style and received feedback from TAs. At the end of the project, they submitted their code. This study focuses on analysis on students' code, rather than the report; therefore, we scoped our analysis to the program code only. In the class, students' project code was not directly graded (as it could be quite extensive), but it was submitted as proof that the students implemented the models described in their report. As a result, students' code was not organized with the intention of being run to reproduce their results, and while the code was generally complete, it sometimes contained isolated sections with errors (discussed in Section 4.3). Therefore our analysis reveals how students write ML code without the pressure of it being directly graded – arguably a more authentic context than graded coursework – which may affect the generalizability of our results (discussed in Section 5).

3.3 Scope

Due to the time-intensive nature of our qualitative analysis, we intentionally limited the scope of our analysis to a single classroom. Therefore, it is possible that the errors we report are the result of specific teaching decisions in that classroom and may not generalize to other instructors, courses, or projects. However, given the lack of literature on ML errors, we believe our results serve as an important starting point, helping to direct instructors and researchers to locations where students are more likely to need assistance. We therefore emphasize the errors themselves, which could happen in other contexts, and not their relative magnitudes, which may vary. We note that similar limitations are common in literature on common misconceptions and errors [9].

3.4 Thematic Analysis

We chose to identify ML errors in a bottom-up manner, to empirically determine which errors manifested in students' submitted code. Typically an "error" is defined as a student action during problem solving that leads them away from a correct solution [23]. For open-ended projects, we defined an error as an action which would lead to:

- The inability to complete the ML pipeline (e.g. misusing a library)
- Incorrect conclusions about the results, such as the performance of a model (e.g. underestimating error)
- Incorrect interpretation of a model (e.g. interpreting parameters of a degenerate model)
- Inefficient programming[1].

Three researchers participated in the coding process, two graduate students and one instructor. All coders had experience in teaching and designing ML courses. We followed the thematic analysis coding process outlined in Braun et al., which provides a rigorous qualitative method of identifying themes within data [1]. This method has been used similarly in analyzing student code and providing results to identify misconceptions [9].

A Jupyter Notebook is an interactive programming environment consisting of independently executable "cells" of code. We annotated each cell in the notebook with any codes that applied. Some projects had multiple notebooks within them, and each notebook within a project was coded individually. If a code was not localized to a specific cell, the code was instead applied to the entire file. For pure python files, we coded the entire file at once and made no divisions between lines of code.

First, the three coders open coded 2 projects, working together to identify codes and develop initial definitions. Afterwards, the rest of the coding procedure was completed by the two graduate students. The two coders open coded 2 additional projects individually, then reconvened to compare their results and began standardizing codes and definitions. They then performed one more round of open coding on 2 projects, before reconvening and creating a closed codebook with precise definitions for each code.

After the closed codebook was created, the coders then went back and re-coded the projects that they had already open-coded, and then met to calculate their inter-rater reliability, refine code definitions, and resolve any disagreements. Inter-rater reliability (Cohen's Kappa) was calculated by constructing binary vectors at the project level: 1 indicating that a code was present in a project, 0 if not. Inter-rater reliability is calculated and reported at the project level, since we were more concerned with the presence/absence of a given error in a project than the exact set of cells responsible for the error.

This process of individual closed coding, code refinement, and disagreement resolution was repeated until an inter-rater reliability score using Cohen's Kappa of 0.8 (very good) was reached, which occurred after 10 projects in total were closed coded. Finally, the two coders evenly split the work (based on the amount of code) between the remaining 3 projects and closed coded them individually.

[1]This is not an "error" in the traditional sense, since it would not lead to an incorrect solution, but we included them in our initial coding to get a better understanding of support students may need.

4 RESULTS & DISCUSSION

We report our results at the project-level, as detailed above. Table 1 shows a frequency count of how many projects a certain error was present in. When mentioning specific examples, we reference a project by ID (e.g. P07). We first review some of the most common errors and inefficiencies, before discussing less frequent (but still notable) errors.

4.1 Categories of Errors

We list our coded errors, inefficiencies, and their respective frequencies in Table 1. Every error code is assigned categories and sub-categories to facilitate the analysis of student submissions.

Programming Errors consist of syntax and logical errors that are not specifically localized to machine learning. While these errors can be specific to machine learning APIs, they do not indicate fundamental misconceptions of how models work or the machine learning pipeline. This also includes *Code Smells*, which are detailed more later. *All* 19 assignments had Programming errors present in them.

Data Processing errors are errors that occur before one begins training and evaluating a model. These errors take place during the beginning of the machine learning pipeline, such as during EDA, pre-processing, feature selection, and any times where a student is wrangling data before training a model. These errors occurred within 12/19 assignments.

Finally, **Training and Evaluation Errors** occur during the training and evaluation process. This includes setting up the hyperparameters of the model, and the student evaluating their results. These errors occurred within 17/19 of the assignments.

4.2 Hyperparameter Errors

Two of the largest conceptual errors we found reinforce the theme that students have trouble using hyperparameters. ML experts agree that from both a theoretical and practical standpoint, tuning hyperparameters correctly is essential for a high performing model [15, 17]. Sulmont et al. mentions that instructors noticed that optimizing parameters was difficult for students, and that "students believe that they do not have to do much work after implementation" [20]. They further state that "This may relate back to students' preconceptions of the power of machine learning from sensationalist media, believing that machines do all of the work." An unclear understanding of how hyperparameters are rigorously tuned can further cement the misconception that ML models are simply black boxes that obtain high performance though unknown means.

Lack of Systematic Hyperparameter Tuning (14/19)

The most frequent error in model training and evaluation, as well as second most frequent error overall, was the lack of systematic hyperparameter tuning. The frequency of this error is notable, given that one of the project requirements was justifying the hyperparameters chosen, suggesting students may have had misconceptions about how to do so correctly.

Many simply failed to specify hyperparameters, leaving the libraries to use default values, without engaging in any tuning process. This is problematic, as default hyperparameter values can lead to very poor estimates of model performance [8]. Other students specified hyperparameter values in their code, but there was no evidence of a systematic tuning procedure, such as grid search. For example P11 left a comment on their final model, stating *"this is the best model hyperparameters after many runs!"*, without specifying which values were tested. This suggests the student may have "played around" with values until they achieved a better result, without a systematic tuning process. One group seemed to create their own system for tuning, by tuning one hyperparameter at a time, fixing the value, and moving on to the next. While this likely did improve model performance, it still indicates a lack of knowledge about existing, efficient hyperparameter tuning approaches (discussed below).

Implications: Instructors should explain the need for a rigorous approach to hyperparameter tuning and explicitly cover methods and corresponding libraries. Once hyperparameters are introduced, ensure that all example code includes hyperparameter tuning such that it is an expected part of the pipeline. From a tooling perspective, one possible solution would be to make it more apparent to the user when a given model has default hyperparameters that could be tuned. By contrast, in `sklearn`, most hyperparameters can be skipped without warning, replaced with default values, and users may be unaware unless they reference the documentation.

Use of Test Data Outside of Model Evaluation (7/19)

The role of a test dataset is to stand in for unseen data, about which a model will have to make predictions when used in production. It is therefore imperative that the test data not be used to shape the model in any way, including feature transformation, hyperparameters selection, and model training [11]. We tagged projects with this code when students used test data incorrectly in this way. This could occur when students used an explicit holdout test dataset (e.g. to select hyperparameters), or by using a validation fold to shape the model during cross-validation.

```
scores = {}
for k in range(1,25):
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(X_train, y_train)
    y_pred = knn.predict(X_test)
    scores[k] = metrics.accuracy_score(y_test, y_pred)
```

Figure 1: Incorrectly searching for the optimal k in a k-NN classifier by evaluating accuracy on the testing set. Argmax of `scores` was chosen as the final k. (Code reduced for brevity)

The most common inappropriate use of test data was to tune hyperparameters. This error is exemplified by code found in P20 in Figure 1. The use of an explicit validation set, or cross validation, for hyperparameter tuning was explicitly taught in the course, suggesting that this skill was difficult to internalize and put into practice.

Another, more subtle way that test data was misused was when students used the whole dataset (training and test) to transform features (e.g. dimensionality reduction, scaling, etc.). For example in Figure 2, P07 fit a `MinMaxScaler` to their whole dataset, rather than fitting to the training dataset only and then transforming both training and test data. This could introduce subtle bias on the evaluation of outliers in the test data. More importantly, since `sklearn`'s

Table 1: Student Errors and Inefficiencies - Categories and Frequencies

Errors	**Category**	**Subcategory**	**Frequency**
Lack of Systematic Hyperparameter Tuning	Training and Evaluation	Hyperparameters	14
Use of Test Data Outside of Model Evaluation	Training and Evaluation		7
Confusing Classification with Regression	Training and Evaluation	Evaluation	6
Unjustified Feature Selection	Data Processing	Features	5
Using Poor Performing Model	Training and Evaluation	Evaluation	5
Improper Use of Dimensionality Reduction	Data Processing		3
Unjustified Feature Transformation	Data Processing	Features	2
Interpreting Model with Poor Performance	Training and Evaluation	Evaluation	2
Failure to Address Correlated Features	Data Processing		2
Unused Analysis	Data Processing		2
Not Verifying Results of Operation	Programming		2
Inappropriate Operation for Data Type	Data Processing		2
Misunderstanding of Ensemble Learning	Training and Evaluation		1
Incorrect Data Sampling	Data Processing		1
Hyperparameter Tuning with Training Data	Training and Evaluation	Hyperparameters	1
Inconsistent Train/Test Splits	Data Processing		1
Tokenized at the Wrong Granularity	Data Processing		1
Tuning Non-Hyperparameter	Training and Evaluation	Hyperparameters	1
Inefficiencies	**Category**	**Subcategory**	**Frequency**
Code Smell	Code Smells		19
Not Using Library	Programming	Efficiency	14
Duplication of Work	Programming	Efficiency	3

transformers, including `MinMaxScaler`, explicitly supports a separate `fit` and `transform` functions, it suggests a lack of knowledge about this paradigm.

Implications: All of these errors share a common theme: the unbiased evaluation of an ML model is threatened by the model observing and evaluating data and features that the model would not have in production. This "data precognition" is a subtle, yet important error that novices encounter when first learning ML. We generalize this theme into a class of errors we call **Precognition Errors**. Precognition Errors would imply that a student has some misconception about when an ML model should have access to certain data and features. These errors can lead to poor model design, as well as poor estimates of model performance. Explicit care is needed on understanding when to use training, testing and validation data – otherwise subtle biases can be introduced into the final model.

4.3 Code Smells

The most frequent errors in our dataset are not specifically related to machine learning at all, but are actually traditional code smells that can be present in any programming project. For our analysis, we consider code smells that are similar to what other CS Education researchers have looked for in simple student projects [12]. These include things such as redundant code, magic numbers, and dead code. Code smells were present in all 19 projects.

A common issue we saw was that students submitted notebooks where a variable was used that was not defined. This is a unique quirk of notebook style computing, which is non-linear compared to traditional programming. Since the code was not directly graded,

```
min_max_scaler = preprocessing.MinMaxScaler()
x_scaled = min_max_scaler.fit_transform(X)
scaled_X = pd.DataFrame(x_scaled)

#train and test data
train_X, test_X, train_Y, test_Y = train_test_split(X,
↪  Y, test_size=0.3)
```

Figure 2: An erroneous way of scaling data before splitting into training and test sets, which allows information about the test set distribution to shape the model's feature transformation.

it was not required for students to submit code that can reproduce their results. Therefore, while a student can compute useful results and document it within their report, they also could overwrite their code multiple times before submitting their project. It is entirely possible for someone to create a variable in cell `A`, use that variable for a calculation in cell `B` and get a result, and later delete cell `A` with it's variable definition. This is one of the pain points identified in prior work that users face when switching to notebook computing. [4].

Implications: One of the most appealing parts of notebooks is that they facilitate *literate programming*, and allow the user to intermix code and discussion for ease of sharing, communication, and reproducible results. Future work could study if code smells have the same detrimental effects in notebook programming as they do in traditional programming, and what effects they have on the reproducibility of results.

4.4 Not Using Libraries

The *Not Using Libraries* code was present in 14/19 projects. As part of homework assignments, students were asked to implement ML algorithms by hand (e.g. K-means) before being introduced to the library. However, during their final project, they copied and pasted their handwritten code instead of using the official `sklearn` libraries. Of course, we can not blame the students for copying correct code, but machine learning libraries are usually optimized and evaluated much more rigorously than handwritten implementations.

Another common behavior is students failing to take advantage of dataframe operations. Some students would iterate and manipulate data using `for` loops instead of using the built in APIs provided for dataframes. As stated by Kross and Guo [13], teaching data-analytic workflows is essential in ML. However, many students encouter dataframe-style programming for the first time in ML courses, and in the absence of explicit instruction may fall back on well-understood iterative/imperative style. Mastery of this paradigm is important for efficient work as a data scientist; not only are these grouping operations more powerful and concise than iteration, but also the creators of these languages and libraries have optimized these operations to be significantly faster than simple iteration.

Implications: Our findings suggest that students may rely on their own code over that of a more reliable library, especially if they have implemented the functionality previously. Students might be disadvantaged in the future by relying on traditional programming paradigms if they are not comfortable with data-analytic workflows. Learning to use APIs is a difficult but teachable skill that is rarely taught explicitly in CS classrooms [10, 22]. In many introductory ML courses, not only do students learn a large amount of theory, but they are exposed to dataframe-based programming for the first time. How should instructors balance the need to teach conceptual understanding versus practical API usage, and is there actually a measurable learning benefit to implementing ML models by hand? From a tooling perspective, efforts to make dataframe operations easier (such as autocompletion, visualization, or possibly GUIs for beginners) would possibly show improved student comprehension.

4.5 Less Frequent Errors

We briefly summarize some of the less frequent errors, which are still notable. Some students **Confuse Classification with Regression** (6/19) during model training and evaluation. Students with this error tended to evaluate their models using *all* the metrics covered in the course without regards to validity: precision, recall, F1, MSE, RSE, etc. There might be a need to help students clearly define the type of their output variable and which evaluation metrics are valid.

Some students engaged in what we deemed to be **Unjustified Feature Selection** (5/19). Given their data, some students cut out features which were perfectly reasonable to be used in the model. In these cases, students removed features without any sort of justification present - either through comments or through the presence of rigorous feature selection methods such as analyzing feature correlation. In addition to this, we also noted students engineered features with no justification, and even proceeded to not even use the features that they engineered.

Lastly, one of the most concerning errors we've encountered was students **Using Poor Performing Models** (5/19). For example, a student might try different models, and the results for all of them would be very poor, either having an extremely low classification accuracy ($< 10\%$), or an absurdly high regression error ($\approx 3 * 10^{17}$). These students fail to identify why their models are not learning, and continue on in their project using the poor performing model as their final result with no justification. What is even more concerning, is that some students attempted to **Interpret a Model With Poor Performance** (2/19), and make inferences about the importance of features with faulty models. This indicates that some students might even have trouble understanding if their model performs well or even learns at all.

5 THREATS TO VALIDITY & FUTURE WORK

The main limitation of our research is the scope of our data. Our analysis is only from one project from one class during a single semester taught by a single instructor. In addition, the class was taught online when it is usually taught in person. These limitations could imply that the troubles we uncovered were a result of specific teaching decisions and are not generalizable. While this course gave an overview of introductory ML topics, some topics are not covered and thus any potential struggle with those topics is missing from our analysis. Another limitation is lack of context. Multiple students could work together on a single project, therefore the overall style of a project or even a single notebook could be muddied by multiple individuals working on the same codebase. Students were graded on their final report, not their submitted code: so it is possible that certain choices were justified within the report and justification was not present in the notebook.

However, due to our choice of codes and nature of qualitative analysis, we believe that these limitations are reasonable. While nonstandard techniques can be justified with the right context, these students are not experts. We highly doubt that any justification they provided would be of the technical rigor required. Due to the time-intensive nature of qualitative coding, we feel that analyzing a single class is suitable for exploratory research within this field. We are not claiming that these codes and their frequency is generalizable to the entire population of ML students.

We observed and codified errors by analyzing a large number of student programs, and hope that further studies use our codes as a guideline to see what actual *misconceptions* lead to these errors. A first step for future researchers would be to perform the same process on other ML courses and see if the relative frequencies and presence of codes generalizes to other course contexts and instructors. Another interesting result to see would be if students make the same misconceptions on formal assessments (e.g. homeworks and tests) compared to open-ended projects. This would help validate the theory that students have less trouble with the independent components of ML projects, and stringing them together into a cohesive whole.

REFERENCES

[1] Virginia Braun and Victoria Clarke. 2012. Thematic analysis. (2012).

[2] Neil CC Brown and Amjad Altadmri. 2017. Novice Java programming mistakes: Large-scale data vs. educator beliefs. *ACM Transactions on Computing Education (TOCE)* 17, 2 (2017), 1–21.

[3] Ricardo Caceffo, Pablo Frank-Bolton, Renan Souza, and Rodolfo Azevedo. 2019. Identifying and validating java misconceptions toward a cs1 concept inventory. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*. 23–29.

[4] Souti Chattopadhyay, Ishita Prasad, Austin Z Henley, Anita Sarma, and Titus Barik. 2020. What's Wrong with Computational Notebooks? Pain Points, Needs, and Design Opportunities. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. 1–12.

[5] Alexandra Chouldechova. 2017. Fair prediction with disparate impact: A study of bias in recidivism prediction instruments. *Big data* 5, 2 (2017), 153–163.

[6] Holger Danielsiek, Wolfgang Paul, and Jan Vahrenhold. 2012. Detecting and understanding students' misconceptions related to algorithms and data structures. In *Proceedings of the 43rd ACM technical symposium on Computer Science Education*. 21–26.

[7] Amit Datta, Michael Carl Tschantz, and Anupam Datta. 2014. Automated experiments on ad privacy settings: A tale of opacity, choice, and discrimination. *arXiv preprint arXiv:1408.6491* (2014).

[8] Preet Kamal Dhillon and Gurleen Sidhu. 2012. Can software faults be analyzed using bad code smells?: An empirical study. *Int J Sci Res Publ* 2, 10 (2012), 1–7.

[9] Yihuan Dong, Samiha Marwan, Veronica Catete, Thomas Price, and Tiffany Barnes. 2019. Defining tinkering behavior in open-ended block-based programming assignments. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. 1204–1210.

[10] Gao Gao, Finn Voichick, Michelle Ichinco, and Caitlin Kelleher. 2020. Exploring Programmers' API Learning Processes: Collecting Web Resources as External Memory. In *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 1–10.

[11] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. 2001. *The Elements of Statistical Learning*. Springer New York Inc., New York, NY, USA.

[12] Felienne Hermans and Efthimia Aivaloglou. 2016. Do code smells hamper novice programming? A controlled experiment on Scratch programs. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*. IEEE, 1–10.

[13] Sean Kross and Philip J Guo. 2019. Practitioners teaching data science in industry and academia: Expectations, workflows, and challenges. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. 1–14.

[14] Roberta Kwok. 2019. Junior AI researchers are in demand by universities and industry. *Nature* 568, 7752 (2019), 581–584.

[15] Gustavo A Lujan-Moreno, Phillip R Howard, Omar G Rojas, and Douglas C Montgomery. 2018. Design of experiments and response surface methodology to tune machine learning hyperparameters, with a random forest case-study. *Expert Systems with Applications* 109 (2018), 195–205.

[16] Joshua J Michalenko, Andrew S Lan, and Richard G Baraniuk. 2017. Data-mining textual responses to uncover misconception patterns. In *Proceedings of the Fourth (2017) ACM Conference on Learning@ Scale*. 245–248.

[17] Philipp Probst, Anne-Laure Boulesteix, and Bernd Bischl. 2019. Tunability: importance of hyperparameters of machine learning algorithms. *The Journal of Machine Learning Research* 20, 1 (2019), 1934–1965.

[18] Jonathan G Richens, Ciarán M Lee, and Saurabh Johri. 2020. Improving the accuracy of medical diagnosis with causal machine learning. *Nature communications* 11, 1 (2020), 1–9.

[19] R Benjamin Shapiro and Rebecca Fiebrink. 2019. Introduction to the special section: Launching an agenda for research on learning machine learning.

[20] Elisabeth Sulmont, Elizabeth Patitsas, and Jeremy R Cooperstock. 2019. Can You Teach Me To Machine Learn?. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. 948–954.

[21] Elisabeth Sulmont, Elizabeth Patitsas, and Jeremy R Cooperstock. 2019. What is hard about teaching machine learning to non-majors? Insights from classifying instructors' learning goals. *ACM Transactions on Computing Education (TOCE)* 19, 4 (2019), 1–16.

[22] Kyle Thayer, Sarah E Chasins, and Amy J Ko. 2021. A theory of robust API knowledge. *ACM Transactions on Computing Education (TOCE)* 21, 1 (2021), 1–32.

[23] Gavriel Yarmish and Danny Kopec. 2007. Revisiting novice programmer errors. *ACM SIGCSE Bulletin* 39, 2 (2007), 131–137.