OneLabeler: A Flexible System for Building Data Labeling Tools

Yu Zhang
yu.zhang@cs.ox.ac.uk
University of Oxford
Oxford, The United Kingdom

Yun Wang
wangyun@microsoft.com
Microsoft Research Asia
Beijing, China

Haidong Zhang
haidong.zhang@microsoft
Microsoft Research Asia
Beijing, China

Bin Zhu
binzhu@microsoft.com
Microsoft Research Asia
Beijing, China

Siming Chen
simingchen@fudan.edu.cn
Fudan University
Shanghai, China

Dongmei Zhang
dongmeiz@microsoft.com
Microsoft Research Asia
Beijing, China

ABSTRACT

Labeled datasets are essential for supervised machine learning. Various data labeling tools have been built to collect labels in different usage scenarios. However, developing labeling tools is time-consuming, costly, and expertise-demanding on software development. In this paper, we propose a conceptual framework for data labeling and OneLabeler based on the conceptual framework to support easy building of labeling tools for diverse usage scenarios. The framework consists of common modules and states in labeling tools summarized through coding of existing tools. OneLabeler supports configuration and composition of common software modules through visual programming to build data labeling tools. A module can be a human, machine, or mixed computation procedure in data labeling. We demonstrate the expressiveness and utility of the system through ten example labeling tools built with OneLabeler. A user study with developers provides evidence that OneLabeler supports efficient building of diverse data labeling tools.

CCS CONCEPTS

• **Human-centered computing** → **Interactive systems and tools**; • **Information systems** → **Multimedia information systems**.

KEYWORDS

data labeling, framework, toolkit, interactive machine learning, visual programming

ACM Reference Format:
Yu Zhang, Yun Wang, Haidong Zhang, Bin Zhu, Siming Chen, and Dongmei Zhang. 2022. OneLabeler: A Flexible System for Building Data Labeling Tools. In *CHI Conference on Human Factors in Computing Systems (CHI '22), April 29-May 5, 2022, New Orleans, LA, USA.* ACM, New York, NY, USA, 22 pages. https://doi.org/10.1145/3491102.3517612

CHI '22, April 29-May 5, 2022, New Orleans, LA, USA

ACM ISBN 978-1-4503-9157-3/22/04…$15.00
https://doi.org/10.1145/3491102.3517612

1 INTRODUCTION

Labeled datasets play an essential role in supervised machine learning. Modern neural networks are generally trained with large labeled datasets. Data labeling typically involves human annotators. Many data labeling tools have been built to enable annotators to label data objects of various types for various learning tasks (e.g., classification [36, 53], segmentation [2]) in different application domains (e.g., image [19, 64], text [18, 62], video [21, 43]). These tools are generally built for a specific labeling task. Different types of labeling tasks require individually built labeling tools.

Building customized labeling tools for various applications is challenging. Programming a labeling tool generally requires cross-disciplinary knowledge on interaction techniques and visual design, algorithmic techniques such as active learning and semi-automatic labeling, and software development skills to implement and compose all these modules into a labeling tool. It is time-consuming and costly. Although many labeling tools have been built, most of them are monolithic applications with limited usage scenarios and are not designed to support further editing and customization. Adapting an existing tool to fulfill a new labeling task with new requirements is generally difficult.

Meanwhile, labeling tools for different labeling tasks share commonalities. To alleviate the burden of building labeling tools, we advocate a modular composable design that extracts the commonalities among different labeling tools to enable easy customization and extension. In this paper, we present a conceptual data labeling framework with a modular composable design. We have identified eight types of common modules through inductive coding of modules in existing data labeling tools. The framework consists of common conceptual modules and constraints in composing these modules. In the framework, a data labeling tool is modeled as a graph denoting its workflow. In the graph, modules are nodes, and the execution order of the modules is encoded with directed edges.

Based on the framework, we develop OneLabeler, a system for building data labeling tools. OneLabeler is designed with reusability and flexibility in mind. It enables visual programming to compose and configure software modules. Developers can build a data labeling tool by creating a workflow graph using built-in implementations of the conceptual modules. In a created labeling tool, interface modules and algorithm modules can be instantiated as human, machine, or mixed computation procedures. The constraints on composing modules are embedded in a static program checker to check the workflow graph and verify the feasibility of the created

labeling tool. A labeling tool built with OneLabeler can be exported as an installer of the tool for sharing.

To demonstrate the expressiveness of the proposed framework and system, we present a case study of ten tools built with OneLabeler. These tools cover various usage scenarios, including a set of typical labeling tools for different data types and labeling tasks, a classification tool for a customized webpage data type, a machine-aided multi-task text labeling tool, a mixed-initiative image classification tool, and prototyping an interactive machine learning system for chart image reverse engineering. To examine OneLabeler's usability, we conduct a user study in which developers are asked to accomplish four tasks on building data labeling tools with OneLabeler. The study results indicate that OneLabeler is easy to learn and use, enabling developers to efficiently build different data labeling tools.

This paper has the following major contributions:

- We propose a conceptual framework that illuminates common conceptual modules and composition constraints in data labeling.
- We develop the OneLabeler system based on the framework to support easy building of diverse data labeling tools.
- We conduct an extensive case study to demonstrate the capability of OneLabeler in creating diverse data labeling tools, as well as a user study to validate the usability of OneLabeler.

OneLabeler's source code can be found at https://github.com/microsoft/OneLabeler.

2 RELATED WORK

2.1 Data Labeling Tools

Due to the importance of labeled data for supervised machine learning, various labeling tools have been proposed for different applications. For example, LabelMe [59] is a labeling tool for image object detection, wherein an annotator can label objects through bounding boxes and polygons. VoTT [49] supports bounding box and polygon annotation in images and video frames. Labelbox [38] enables annotation for classification, segmentation, and object detection in images and videos. VIA [24] allows annotators to label spatial regions and temporal segments in images, audios, and videos. These tools require intensive human labeling efforts.

Various techniques focus on reducing human efforts, typically through integration of machine assistance for semi-automatic labeling. ISSE [13] algorithmically suggests refined segmentation to help annotators separate sound into its respective sources by painting on time-frequency visualizations. Fluid Annotation [2] uses a pre-trained model to propose a set of possibly-overlapping segments to help annotate image segmentation. V-Awake [29] focuses on time series segmentation. It uses LSTM to assign tentative labels and visualizes the model information to help annotators diagnose and correct model predictions.

Another family of techniques commonly used in labeling tools is active learning that focuses on prioritizing user efforts. It has been used for labeling documents [36, 62] and geometric objects [85]. In a similar vein, Deng et al. propose a strategy to select data objects to label that maximize annotation's utility-to-cost ratio in multi-label classification [22]. While active learning focuses on algorithmic selection strategies, the selection may also involve user feedback [53]. For example, projection scatterplots with iso-contours visualizing label uncertainty have been used to assist users in data selection [43].

Aside from semi-automatic labeling and active learning addressing algorithmic aspects, novel interaction and visual design are proposed to support efficient labeling. Clustering is frequently used in data labeling tools [19, 65]. Similar data objects that likely share the same label can be grouped to batch their labeling [64]. MediaTable [21] combines automatic content analysis and a tabular interface providing *focus+context* in image and video labeling. Choi et al. [18] highlight keywords that imply document sentiment, identified by the attention mechanism of LSTM, to aid annotators.

Another branch of study focuses on designing tasks assigned to annotators. A prominent example is gaming with a purpose (GWAP) [69] in which annotators are instructed to play games to implicitly contribute data labels. Techniques in this category, such as ESP game [68] and Peekaboom [70], focus on the gameplay design that strives to motivate annotators. Another example is embedding labeling tasks in human/bot verification [71, 77]. A large body of work on crowdsourcing annotations relates to task design, which typically focuses on task assignment schemes to collect high-quality labels with low cost and latency [15].

Despite the diversity in labeling tools, common themes exist, such as using semi-automatic labeling and active learning to save annotation efforts. These commonalities lay the foundation for our efforts described in Section 4 to summarize shared conceptual modules across data labeling tools.

2.2 Workflows in Data Labeling

Research efforts have also been directed to design patterns for labeling tools, typically in the form of generic workflows composed of conceptual modules. Settles' active learning survey describes a typical pool-based active learning cycle with algorithmic query selection, human annotation, and model training [61]. Wang and Hua survey the use of active learning in multimedia data annotation and retrieval, and summarize three schemes: conventional active learning, multiple-instance active learning, and large-scale interactive annotation [73]. Höferlin et al. introduce a workflow that extends the conventional active learning workflow with user selection and model manipulation [31]. Bernard et al.'s workflow integrates interactive visualization components [6]. Zhang et al. propose a workflow that features algorithmic sampling and default labeling [85].

Similar to the literature on generic workflows, we aim to summarize design patterns in labeling tools. Instead of a single workflow, our solution is a generative framework for building workflows. We believe that a single workflow cannot be optimal for all usage scenarios. Meanwhile, our framework builds on existing workflows, as they are included in the corpus for summarizing common modules through coding (in Section 4).

Our work falls into the family of toolkit research in HCI [39] that contributes techniques for building new tools. Examples in this body of research related to our work are the ones that assist the development of visual interfaces [8, 48, 56]. Meanwhile, to the best of our knowledge, there exists no literature on extensible systems for building data labeling tools. A relevant thread of work that also

aims to save the cost of developing labeling tools is data labeling platforms and systems that support multiple labeling applications. Examples include the project templates of Amazon Mechanical Turk [1] and Prodigy [26]. While the template-based approach can address a number of common use cases, extending to new usage scenarios requires implementing new templates from scratch, and reuse of software modules is not supported. LabelStudio [67] also provides a collection of templates but with more flexibility. While it allows the user to use active learning and pre-labeling in the labeling interface, customizing interface modules in LabelStudio is difficult. OneLabeler also aims to promote reuse as these systems but focuses on reuse at the module level instead of template level. Additionally, building on the conceptual framework, OneLabeler supports a more extensive coverage of data labeling techniques.

3 DESIGN REQUIREMENTS

Various scenarios require deploying different labeling tools to collect labels, but developing an effective labeling tool is generally challenging and costly. We see an opportunity to alleviate developers' efforts through modular composable design for reuse, as there are commonalities in labeling tools. We aim to develop a flexible system to enable easy building of various data labeling tools. To better understand pain points in developing labeling tools, we interviewed three experts with expertise in developing labeling tools as an informal pilot study.

3.1 Pilot Study

The three experts we interviewed have developed labeling tools for a company-owned research institute, two with the machine learning (ML) background (E1, E2) and one with the HCI background (E3). They have 8 to 12 years of programming experience. E1 has developed labeling tools for image object detection and binary classification. E2 has developed text labeling tools for relation detection between tables and documents. E3 has developed an image labeling tool for multi-label classification. The study started with a structured interview with questions concerning participants' experience of developing labeling tools. Example questions we asked include "Why did you decide to implement a data labeling software instead of using existing ones?" and "What functionalities do you think are critical for data labeling software?" Then, we conducted a walkthrough demonstration of an early prototype of OneLabeler with rudimentary functionalities.

Through the interview, we observed that no participant ever used public labeling software in their projects due to their application-specific requirements. E1 mentioned that he needed to integrate a customized rule to forbid labeling the same data object twice in an image, which could not be achieved in existing tools. E2 mentioned that the text datasets for the labeling tools he developed were stored in different types of structured data (e.g., document, table, webpage) and required different types of structured labeling. We conclude that customized labeling tools are needed to address application-specific requirements.

Developing a data labeling tool is time-consuming. It takes 7 to 30 man-days for an experienced engineer to develop one according to the participants' experience. Regarding the three labeling tools he had developed, E2 commented: *"... the interface modules for displaying data points are similar. They all display the table to be annotated, and the supported interactions for annotating the table are similar."* This comment implies that labeling tools share commonalities, but facilitating reuse requires a careful software design. Like other software development, iteration is needed for labeling tools. E1 commented: *"The customized rules typically need to go through many iterations as the ML project progresses."* Similarly, E3 said, *"I didn't know the label categories at the beginning and was not sure about the labeling task that I needed to carry out. Thus, I need to be able to refine the data labeling interface correspondingly as I gradually get more annotations."* Finishing a functional labeling tool is not the end of development, as it frequently needs to undergo further editing. We conclude that building labeling tools is time-consuming and needs iterations. Meanwhile, there is an opportunity to save the development cost, as labeling tools share commonalities.

While E1 and E2, both from the ML background, regarded the annotation interface design and implementation of annotation interaction as the hardest part, E3 from the HCI background regarded algorithm modules as the hardest to implement. We conclude that grasping all required techniques for implementing a data labeling tool can be difficult for labeling tool developers since it may involve cross-disciplinary knowledge. Visual programming can help reduce the skill barrier on labeling tool development in this case [35].

All of the three experts said they would be happy to use OneLabeler if their labeling tasks were supported. They worked on label tasks of image, video, and text. We conclude that the system should build in various modules to support different label tasks for good coverage of usage scenarios.

3.2 Design Principles and Requirements

The interview confirms the need for a system to support easily building and modifying labeling tools for different labeling tasks and application-specific requirements. To achieve this goal, we settle on the following design principles for the system:

- **DP1: Modular composable design and reuse at multiple levels.** To exploit shared commonalities among labeling tasks and to facilitate reuse and modification, the system should be based on a modular composable design to enable building labeling tools with composable primitives. Primitives should have low coupling. Reuse low-level implementations as composable primitives, while reuse high-level implementations as editable templates. These reuses can significantly reduce the development time and efforts.
- **DP2: Ease building and guide the development process.** To reduce the skill barrier, the system should facilitate easy building of labeling tools through visual configuration and guiding developers towards feasible solutions.

Based on the design principles, we propose the following specific requirements for our OneLabeler system. **R1**, **R2**, and **R3** meet **DP1**, while **R4** and **R5** meet **DP2**.

- **R1: Unified module APIs.** Software modules (i.e., interface modules and algorithm modules) should implement unified APIs. Each API should represent a family of common modules in labeling tools. Unified APIs allow substituting a software module in a labeling tool or extending the system with a new module without changing other modules (i.e., separation of concerns). In

this way, unified APIs make it easier to extend and customize software modules.

- **R2: Module composition.** The system should support composition of modules to allow a rich space of labeling tools to be created (i.e., good coverage of usage scenarios) with a small number of primitives. Especially, algorithm and interface modules should be composable, as labeling tools typically feature joint efforts of humans and machines.
- **R3: Reuse modules and tools.** The system should build in common software modules to be reused as primitives and example composed tools to be reused as boilerplate to scaffold building of labeling tools, which can significantly ease building commonly used labeling tools and enable extending and customizing built-in label tools to fit application-specific requirements.
- **R4: Visual programming.** The system should support visual programming to enable developers to reuse software modules and compose workflows with no or minimal textual coding. Labeling tool development requires cross-disciplinary knowledge. A developer may not be proficient with every part of the required technology stacks. Visual programming can reduce the development barrier in such scenarios. It has become a common practice in enterprise machine learning services (e.g., Microsoft Azure[1] and Amazon Web Services[2]). Moreover, the diagram in visual programming provides an overview of the developed system that assists the development process.
- **R5: Static checking.** The system should support static checking to detect infeasible configurations and recommend their fixes to guide editing actions during workflow creation. It enables a developer to spot and localize bugs at an early stage without running the program and guides the developer to reach a feasible solution quickly.

A critical step to fulfill these design requirements is exploitation of commonalities in various labeling tools. This is discussed in the next section, where we identify common APIs (**R1**), common software modules for reuse (**R3**), and common constraints for composing modules (**R2**, **R5**).

4 DECOMPOSING LABELING TOOLS

To facilitate fast and easy development of labeling tools, our system should build on a modular design. To design APIs with expressiveness and a suitable level of abstraction, we consult decomposition of labeling tool modules in the literature. In the following, we describe the common conceptual modules and states identified through inductive coding of the literature. Each module, together with its input and output states, defines a unified API for a family of techniques (**R1**). For each module, we give examples of its instances that can serve as software modules in labeling tools (**R3**). An instance of the module corresponds to an implementation satisfying the API. The constraints in composing the modules in building labeling tools are described at the end of the section. The constraints guide the module composition (**R2**) and facilitate static checking (**R5**).

[1]https://azure.microsoft.com/services/machine-learning
[2]https://aws.amazon.com/sagemaker/

4.1 Methodology

To identify common conceptual modules, we use data labeling flowcharts in the literature as the dataset and summarize common themes of modules and states of labeling tools. The rationale is that flowcharts depict cautious decomposition of software into executable modules by the authors. Commonalities in the decompositions likely imply modules with high cohesion and low coupling.

We start with 76 papers on labeling tools and methods published between 2003 and 2021, collected from venues in human-computer interaction, visualization, multimedia, and machine learning. The full list of the papers is provided in the supplementary material. For each paper, we extract flowchart figures describing data labeling workflows, which can be application-specific or generic. We discard the papers without flowchart figures, and obtain a dataset of 36 flowchart figures extracted from 33 papers (2 of them contain multiple flowchart figures).

We conduct an inductive coding of the extracted 36 flowcharts with two stages. In the first stage, we extract the tagged phrases from each flowchart and categorize them into "module" (human, machine, and mixed computation process) and "state" (input and output). The phrases form preliminary sets of module codes and state codes for that flowchart. For example, Crayons [27] supports image pixel-level segmentation with a flowchart containing five phrases. Four of them ("train", "classify", "feedback to designer", and "manual correction") are categorized as preliminary codes for modules because they describe actions and are placed inside blocks in the flowchart figure, while one of them ("interactive use") is excluded as it is a modifier of another phrase.

In the second stage, we group the preliminary codes collected from all the flowcharts into themes. The grouping is not mutually exclusive. We further remove the themes outside the scope of data labeling (e.g., "model understanding" is removed), and merge themes when necessary to synthesize a final code for each theme. The final module/state codes are generated from representative preliminary codes and normalized into short noun phrases describing an action/variable. For example, "classify" in Crayons [27] is finally grouped into the theme "default labeling", and this theme also serves as its final code.

4.2 Results

After the first stage of the coding process, we obtain 188 and 163 preliminary codes for module and state, respectively. For each coded flowchart figure, the number of preliminary module codes ranges between 1 and 11 ($mean = 5.22$, $SD = 2.25$), and the number of preliminary state codes ranges between 0 and 10 ($mean = 4.53$, $SD = 2.61$). After the second stage, we identify five final codes for states (sorted by frequency as marked):

- **Data Objects**: the list of entities to be labeled (59/163).
- **Labels**: the list of annotations assigned to entities (57/163).
- **Samples**: an entity subset annotators handle at a time (18/163).
- **Model**: one or multiple machine-learned models (18/163).
- **Features**: the list of feature representations of entities (13/163).

Similarly, we identify eight final codes for modules: **interactive labeling**, **data object selection**, **model training**, **feature extraction**, **default labeling**, **quality assurance**, **stoppage analysis**, and **label ideation**, which are to be introduced in details in the next

subsection. The occurrences of the final codes in the 36 flowcharts are summarized in Appendix A (Table 2 and Table 3), and the detailed coding results are included in supplementary material.

During the coding process, we have excluded several themes of states and modules regarded as irrelevant to data labeling (see Appendix A.1). For example, understanding the machine-learned model does not directly benefit labeling, and thus we exclude the "model understanding" theme, whose occurrences include the phrase "understand" in Höferlin et al. [31]'s flowchart. Similarly, the "data collection" theme refers to the process of collecting or enlarging the dataset to be labeled. Its occurrences include the phrase "video crawler" in Hua and Qi [33]'s flowchart. We exclude it because of the low frequency (4 occurrences) and the ambiguity of whether it should be regarded as a part of data labeling or a preparation step before data labeling.

Combining the modules and states finally devised from the coding process, we further identified the common APIs in data labeling tools, as depicted by Fig. 1. To ensure that every API has an output, we add two additional states "categories" (output of "label ideation") and "stop" (output of "stoppage analysis"). API inputs are optional, as a software module implementing an API may utilize some or even none of the inputs available. These API definitions are integrated in OneLabeler (described in Section 5).

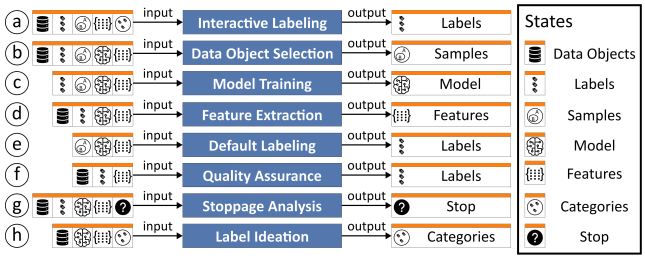

Figure 1: Common APIs in data labeling tools.

4.3 Common Modules in Labeling Tools

The conceptual modules (sorted by frequency as marked) and their instances as software modules are described below. An instance can be an algorithm (i.e., a machine computation process), an interface module involving human computation, or a mixed computation process.

- **Interactive Labeling** (49/188) is the core module where annotators carry out labeling tasks in an interface (Fig. 1a). Different instances of this module typically vary in the design of task, interface, and interaction. *Task design* concerns the task that annotators are instructed to carry out. Typical tasks include label assignment and correction tasks for the concerned data type (e.g., image, text) and label task type (e.g., classification, segmentation). Alternatives include GWAP instructing annotators to play a game [68–70]. When strong labels (e.g., segmentation) are needed, using weakly supervised learning may allow annotators to provide weak labels (e.g., classification) that are then algorithmically compiled to strong labels, potentially improving labeling efficiency [10]. Additionally, various research efforts in crowdsourcing concern task scheme design [16]. *Interface design* typically concerns data objects' layout such as grid matrix and hierarchical layout [19, 81]. Tasks requiring fine-grained editing may prefer displaying one data object each time [2, 59]. *Interaction design* concerns interaction techniques facilitating efficient labeling such as batch edit [64] and graph cut [9].
- **Data Object Selection** (34/188) determines the order for data objects to be selected and labeled by annotators (Fig. 1b). Instances include active learning strategies that select informative data objects first [62]. The selection may also use clustering algorithms to group similar data objects [19, 45], which may enable annotators to assign the same label in one go. Annotators may involve in deciding which data objects to label [14, 43].
- **Model Training** (21/188) trains/updates a learning model (that may serve as input to other modules) with newly gathered labels (Fig. 1c). Variations of model training may concern what model is trained and how the training is conducted. Although any predictive model may serve the purpose, semi-supervised learning and transfer learning techniques better match data labeling scenarios, where unlabeled data objects are abundant (useful for semi-supervised learning), and the cold start issue is prominent (alleviated by transfer learning). Model training may be conducted by training from scratch or by incremental update methods in online learning.
- **Feature Extraction** (20/188) turns data objects into feature representations, typically vectors (Fig. 1d), facilitating other modules that cannot work with raw data objects (e.g., model training). Algorithmic instances of feature extraction can be handcrafted (e.g., HoG), unsupervised (e.g., PCA), or supervised (e.g., LDA). It is also possible to involve annotators in feature extraction [17].
- **Default Labeling** (17/188) assigns tentative labels to data objects, simplifying annotators' work from creating labels to verification and correction (Fig. 1e). It may be facilitated by models trained with the model training module, pre-trained models, or rules.
- **Quality Assurance** (6/188) reviews label quality and corrects erroneous labels (Fig. 1f). Algorithmic relabeling may be used to suggest suspicious labels for annotators to verify [12, 51, 88]. Annotators may also exploratorily review labels and search for potentially mislabeled data objects to correct [4, 29, 75]. In mission-critical applications, quality assurance may require going through all the labels one by one [86].
- **Stoppage Analysis** (4/188) decides whether to keep assigning tasks to annotators or stop (Fig. 1g). A common criterion is to check if all the data objects have been labeled once. Alternative criteria may decide the stoppage time by empirical measures of the label quality [22] or confidence [87] and stability of models [7, 84] trained with the partially labeled dataset.
- **Label Ideation** (3/188) develops the label categories used for labeling (Fig. 1h). It may appear as an interface widget allowing an annotator to create new categories ad hoc. More structured ways to generate categories may involve algorithmic assistance (e.g., topic modeling) in an interface that supports users iteratively propose, verify, and refine categories [28, 37].

Through the coding process, we observe that interactive labeling is by far the most common module in labeling tools. It appears 49 times among the 188 codes extracted from the 36 flowcharts, meaning that on average more than one module in each flowchart

is related to interactive labeling. Any flowchart that strictly depicts a data labeling workflow with human annotators would need to mention it at least once.

4.4 Module Composition Constraints

Labeling tools can be built with instances of the conceptual modules as building blocks. However, not all compositions of software modules produce a valid labeling tool. A labeling tool is represented by a flowchart that contains module nodes, i.e., nodes denoting implementation of common modules, as well as initialization, decision, and exit nodes. Assuming that states of a labeling tool are stored globally, modules fetch inputs from and register outputs to corresponding global states.

The constraints for composing software modules into a valid labeling tool are listed as follows:

- **Valid Flowchart**: The graph should satisfy graph-theoretic constraints that generally hold for a flowchart [34]. For example, all the nodes should be reachable from the initialization node.
- **Input Initialized**: For each possible walk on the graph, a node that represents a software module should not be visited until its input parameters are all initialized.
- **No Redundancy**: After a module is visited, it should not be revisited until at least one of its inputs has changed its value. After a module is visited, its output(s) should be used by a module.
- **Involve Labeling**: Interactive labeling should exist in all walks of the graph to ensure it depicts a workflow of a labeling tool.

The first two constraints ("valid flowchart" and "input initialized") are derived from the rationale that the flowchart should represent a correct program. The third constraint concerns efficiency of the software. Labeling tools often require heavy human computation (e.g., manual annotation) and heavy machine computation (e.g., model training), making performance optimization a critical issue. The fourth constraint ensures that the represented software is a labeling tool. These constraints translate to rigid propositions (in Appendix B) and are integrated in OneLabeler's static checking function to validate user-created labeling workflows.

5 ONELABELER

Based on the requirements described in Section 3, we propose the OneLabeler system for building labeling tools. OneLabeler enables developers to visually program (**R4**) data labeling tools by composing software modules into a workflow (**R2**). A created labeling tool can be exported as an installer. The conceptual modules identified in Section 4 inform the API design for data labeling modules in OneLabeler (**R1**). A developer can reuse a collection of built-in modules and templates (**R3**) or customize on demand. OneLabeler's visual programming environment supports static checking (**R5**) of user-created workflows to assist debugging and guide towards feasible solutions.

5.1 System Architecture

5.1.1 Module. The eight types of conceptual modules (interactive labeling, data object selection, model training, feature extraction, default labeling, quality assurance, stoppage analysis, and label ideation) are integrated into OneLabeler as eight API definitions (**R1**). An API can be implemented with either an algorithm or interface module. While an algorithm module can be automatically executed, the execution of an interface module requires human intervention. In the following, we use the data object selection module as an example for illustration.

Defining a conceptual module as API: As shown in Fig. 1b, the data object selection module conceptually depicts a function that returns a dataset subset, given data objects, labels, features, model, and samples. The inputs are optional. Schematically, it translates to an API definition in OneLabeler as:

```
class DataObjectSelection {
  /** Input states. */
  props: ['dataObjects', 'labels', 'samples', 'model', 'features'];
  /** Output state's setter, evoked in .render or .run. */
  methods: ['setSamples'];
  /** Whether the execution of the module is blocking or not. */
  blocking: boolean;
  /** (Optional) Only needed for an interface module. */
  /** Whether to render the module when not executed. */
  persistent?: boolean;
  /** (Optional) Interface module implementation. */
  render?: Function;
  /** (Optional) Algorithm module implementation. */
  run?: Function;
}
```

Executing a module implementation: Conceptually, executing a module implementation is to send it the input states and wait for the output states to be returned. For example, the data object selection API defined above can be implemented with an algorithm (e.g., an active learning strategy) or a user interface module (e.g., a projection scatterplot as Fig. 6B$_1$ where an annotator can select manually with lasso). When the API is implemented as an algorithm, the execution is a straightforward function call. When the API is implemented as an interface, the execution is regarded as a function call that has a side effect of altering the user interface. The function returns when a callback function setting the returned states is triggered by user interaction in the interface. For example, a projection scatterplot (as Fig. 6B$_1$) is regarded as a function that returns the selection after the user finishes a lasso selection in the scatterplot. Data object selection takes samples as an optional input because when implementing it with a projection scatterplot, the scatterplot may need to highlight the previous samples.

Input and output access: Each module fetches inputs and returns outputs through getters and setters exposed by a global data model. We refer to this storage model as a *blackboard model* as all the modules read from the blackboard and write on the blackboard. OneLabeler and labeling tools generated with OneLabeler follow the model-view-controller (MVC) design pattern[3]. The seven states of labeling tools in Fig. 1 are stored globally and managed by the data model. The data fetching and storage are configurable in labeling tools built with OneLabeler. It allows multiple annotators to use installations of a labeling tool that have data access from the same database, enabling simultaneous annotation by multiple annotators.

5.1.2 Workflow. To enable developers to create complex tools with a small number of primitives, OneLabeler supports composition of software modules into a workflow declaring a labeling tool (**R2**).

[3]To disambiguate, the (data) "model" here refers to the part of software responsible for managing states (e.g., states in Fig. 1), instead of a machine learning "model".

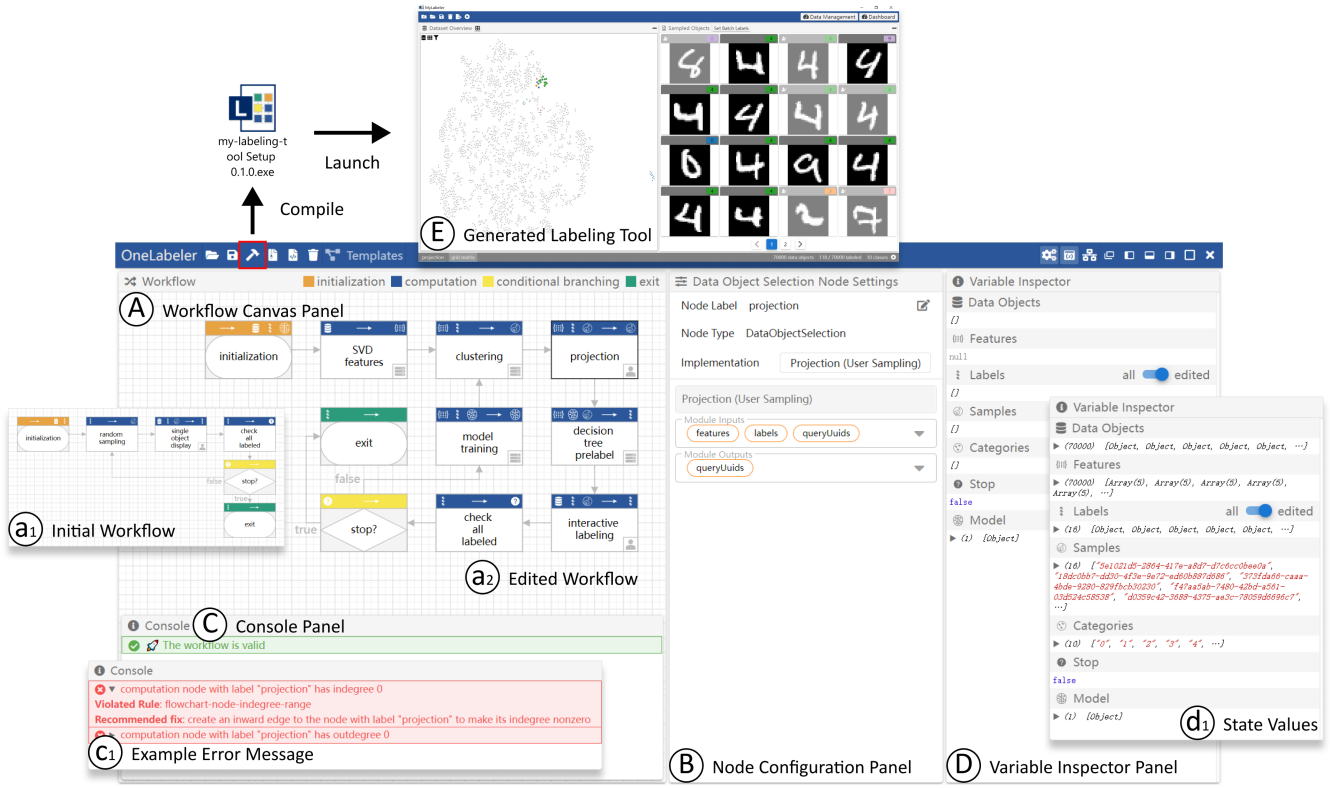

Figure 2: OneLabeler's visual programming interface: (A) Workflow canvas panel for a developer to create and compose nodes, as well as editing the visual layout of the workflow; (B) Node configuration panel for configuring parameters of a selected node; (C) Console panel for showing static checking result of whether the workflow denotes a valid labeling tool; (D) The variable inspector panel showing state values; (E) The generated data labeling tool by OneLabeler according to the workflow.

Defining a workflow: Within OneLabeler, a labeling tool is declared by its workflow graph with modules being nodes and execution order specified by directed edges. The graph can be either created in OneLabeler through visual programming (see Section 5.2), or uploaded to OneLabeler through a JSON file storing a list of nodes and edges. A valid workflow graph has to satisfy constraints described in Section 4.4. Unlike data flow systems [48, 79], directed edges in OneLabeler only define execution order without implying data transmission. Each module fetches inputs and returns outputs through the global data model. Thus, for each directed edge, the source node's output is not necessarily the target node's input.

Executing a workflow: To execute a workflow, OneLabeler traverses the graph representing the workflow. The traversal starts from the initialization node in the graph. OneLabeler recursively visits subsequent nodes following directed edges. When a node corresponding to a module is visited, OneLabeler supplies the module's required inputs by reading the blackboard model, and executes the module as a function. When the module is an interface module, the interface is rendered in a window (e.g., Fig. 6B$_1$ is a rendered window for data object selection using interactive projection). By default, OneLabeler waits for the function to return. After the function returns, OneLabeler registers the return value to the global data model and then visits the next node. Alternatively, the module can be optionally configured to be non-blocking, in which case OneLabeler visits the next node before the function returns. When the workflow is faithfully executed, the labeling tool's interface may constantly be changing because, after execution, an interface module will no longer show up until the next visit. The "persistent" option addresses this issue of the rapid context switch. For a "persistent" interface module, the interface is persistently rendered in the window even when the node is not executed. When a decision node is visited, the decision criterion is checked, and the edge to follow is chosen accordingly. The execution of the labeling workflow terminates when the exit node is visited.

5.2 Visual Programming of Workflows

OneLabeler enables developers to build labeling tools through visual programming of the labeling tools' workflows (**R4**). The navigation toolbar provides import/export and compilation functions through button clicking (highlighted in Fig. 2 with red). The labeling tool's workflow can be imported/exported in the JSON format to facilitate reuse and sharing. A developer can use the built-in templates provided in the template menu as a boilerplate to scaffold development. When the workflow is finalized, the developer can click the compile button on the navigation bar to compile the workflow into an installer of the corresponding labeling tool.

5.2.1 Visual Programming and Configuration. In OneLabeler, a data labeling tool is declared by its workflow. To enable developers to visually configure the workflow without textual programming (**R4**), OneLabeler provides a visual programming interface (Fig. 2). Within the interface, the developer can build a labeling tool by interactively configuring its workflow.

Adding a node: Within the workflow canvas panel (Fig. 2A), the developer can add nodes through the right-click menu on the canvas. The menu provides module nodes and control nodes. The module nodes belong to one of the eight types of conceptual modules (in Section 4). The control nodes can be initialization, decision, and exit nodes. After creating a module node, to make it executable, the developer needs to configure the implementation used for the node, as described below.

Composing nodes: Two nodes can be composed by creating a directed edge to specify the execution order. To create an edge between two nodes, the developer can hover on a node serving as the source, which shows the link ports of the node, drag the link port of the node, and release on the link port of another node serving as target.

Configuring a node: To configure implementation details of a node, the developer first needs to select a node in the workflow canvas panel through clicking. The node configuration panel (Fig. 2B) shows the details for the selected node. The user can select the built-in implementation of the node in the "implementation" menu in the configuration panel. The parameters of the selected implementation can be configured on demand. For example, for the data object selection module (as shown in Fig. 2B), the user can configure which selection method to use (e.g., active learning or interactive projection) and the parameters of the selected implementation (e.g., how many data objects to sample each time by the active learning strategy). Additionally, outputs of the initialization node are configurable. If a state is configured as initialization's output, it will be initialized to a non-empty value when initialization is executed.

5.2.2 Static Checking. To guide a developer smoothly towards a feasible solution (i.e., a workflow that depicts a valid labeling tool), we use the classic idea of static program analysis [3] in software debugging (**R5**). The constraints (in Section 4.4) allow OneLabeler to check the graph-theoretic properties of the workflow created by the developer as a proxy of the feasibility of the created labeling tool. Besides, we add practical constraints on the graph data structure (e.g., node id should be unique) and on module configuration (i.e., an implementation should be chosen for the module). OneLabeler integrates these constraints into a checker to statically validate the created labeling tool. OneLabeler notifies the developer of identified violations of the constraints in the console panel (Fig. 2C), allowing the developer to identify potential mistakes before using the created labeling tool. Hovering/Clicking an error message in the console panel highlights/selects the node(s) and edge(s) involved in the error in the workflow canvas panel. Clicking the triangular expand button on the error message shows the error code and the recommended way(s) of fixing the error (Fig. $2c_1$). For example, when a new module node is just created by the developer, there is no edge connecting it to other nodes. In this case, the "valid flowchart" rule (in Section 4.4) is violated, since this node has indegree and outdegree zero, meaning it is not used. The error messages *computation node with label "projection" has indegree 0* and *computation node with label "projection" has outdegree 0* displayed in the console panel remind the developer to resolve this issue. OneLabeler continuously validates the workflow as the developer edits it. The messages are ranked by severity. The low severity errors are hidden from the console panel until high severity ones are resolved to avoid overwhelming the developer with many error messages. The developer can arrive at a feasible solution, i.e., a labeling tool that works, by iteratively resolving the error messages.

5.2.3 Labeling Tool Preview. To help the developer debug the built labeling tool, OneLabeler provides a real-time preview of the built tool. As the developer finishes the workflow, the developer can try out the preview to examine if it meets the requirement. During the process, the developer can inspect the state values (e.g., Fig. $2d_1$) of the labeling tool through the variable inspector (Fig. 2D). OneLabeler also enables the developer to manipulate the control flow of the preview for debugging. Specifically, the developer can conduct single-step debugging for a node or force the control flow to start from a node. The control flow manipulation functions can be selected in the node's right-click menu.

5.3 Built-in Modules

OneLabeler provides a collection of built-in implementations for the conceptual modules. The user can configure the implementations on demand. Moreover, for interactive labeling modules, the user can configure three dimensions (i.e., data type, label task type, and interface design) to derive various combinations.

- For **interactive labeling**, as it is the most important module in data labeling, OneLabeler splits it into three design dimensions: data type, label task type, and interface design. OneLabeler builds in the following implementations for the three dimensions:
  - **Data type**: image, text, video, audio, point cloud.
  - **Label task type**: single-/multi-label classification, freeform text annotation, object detection (for image), segmentation (for image and point cloud), text/temporal span tagging (for text, audio and video), span relation (for text, audio and video).
  - **Interface design**: single object display (e.g., Fig. $5B_1$) and grid matrix with editable layout (e.g., Fig. $6B_2$).

  The three dimensions are identified following the rationale that the interface for interactive labeling needs to show data object details (which depends on the **data type**), provide the interaction for annotating labels (which depends on the **label task type**), and layout the data objects following an **interface design**. The data types are chosen to cover the data types in the 33 coded papers as well as common data types in machine learning benchmark datasets as indexed by *Papers With Code*[4], including image, text, video, audio, point cloud, and sequential data. We have implemented all these data types except sequential data as it is concerned in only one paper [40]. Similarly, the built-in label task types cover all the tasks in the 33 papers except for video object tracking with one occurrence [31]. The two most common interface designs (i.e., grid matrix and single object display) in the 33 papers are already built-in, while others (i.e., thumbnail projection, table with metadata, and thread layout)

[4] https://github.com/paperswithcode

are excluded for now. Those excluded from the current built-in can be implemented in the future.

- For **data object selection**, OneLabeler builds in algorithmic selection and interactive selection implementations. For the algorithmic selection, OneLabeler provides active learning techniques (including three entropy-based methods [11, 41, 76], least confidence, and smallest margin), clustering-based techniques (selection by ranking cluster labels, distance to cluster centroids, and density estimation). For interactive selection, OneLabeler provides interactive configurable data projection methods (e.g., Fig. 6B$_1$). The projection can be configured to be a scatterplot or heatmap of a T-SNE/PCA/LDA projection of the feature values, where the user can select data objects to label.
- For **model training**, OneLabeler currently builds in several classic supervised/semi-supervised learning algorithms, including decision tree, support vector machine, logistic regression, restricted Boltzmann machine, and graph-based label propagation. OneLabeler can be easily extended to support models provided by libraries that follow the scikit-learn library's API design.
- For **feature extraction**, OneLabeler builds in three techniques for image feature extraction, including a bag of features technique (handcrafted features describing color, edge, and texture), truncated SVD of raw images (unsupervised), and LDA projection of raw images (supervised). For text feature extraction, non-negative matrix factorization of tf-idf is provided.
- For **default labeling**, OneLabeler builds in model prediction, where the model can be the ones trained by the model training module(s) or provided by the developer through a prediction API. OneLabeler also provides a rule-based default labeling method for text span labeling based on part-of-speech (POS) tagging.
- For the other three conceptual modules with relatively low frequencies (in Section 4), OneLabeler builds in basic implementations for the time being. For **quality assurance**, modules for interactive labeling can be reused for the annotator to go through data objects one by one. For **stoppage analysis**, OneLabeler builds in a criterion based on the sample rate specifying the rate of data objects to be labeled by annotators before stopping. For **label ideation**, OneLabeler provides an interface widget where label categories can be dynamically added, and for each label, the applicable label task type can be specified.

A developer can craft a labeling tool's workflow in the visual programming interface from scratch with the modules described above. Additionally, OneLabeler builds in a collection of predefined template workflows, covering combinations of built-in data types and label task types. Specifically, the built-in templates include image classification, image segmentation, text classification, text span tagging, video classification, video temporal segmentation, audio classification, audio temporal segmentation, point cloud classification, and point cloud segmentation. The built-in templates are chosen to cover the five built-in data types, each data type with one coarse label task (i.e., classification) and one fine-grained label task (e.g., segmentation). A developer can start with a predefined template and tailor it towards specific needs. For example, a developer may start from a template workflow shown in Fig. 2a$_1$ and add nodes to support default labeling and mixed-initiative sampling and corresponding connections, resulting in a revised workflow shown in Fig. 2a$_2$. The workflow (Fig. 2a$_2$) and the resulting data labeling tool (Fig. 2E) is almost the same as the ones detailed in Section 6.4 except for minor differences in the configuration for interactive projection and cluster sampling batch size.

5.4 Customization

OneLabeler's modular architecture supports extensions for module implementations and workflow templates (**R1**). If the built-in implementations do not meet a developer's needs, the developer can provide customized implementations. OneLabeler provides a command-line interface (CLI) to scaffold customization. Using the CLI, the developer can choose a type of customization as introduced below, and OneLabeler will create corresponding template files for the developer to start with (see our code repository at https://github.com/microsoft/OneLabeler for details on the CLI).

5.4.1 General Purpose Module Customization. To add a customized module to OneLabeler, a developer needs to fill the data structure as specified by the API definition in Section 5.1.1. For an algorithm module, a developer needs to implement the "run" function that includes the code for running the algorithm and registering the result through setters (e.g., "setSamples") when the algorithm returns. For an interface module, a developer needs to implement the "render" function that includes the code for rendering the module and registering the result through setters in the callback function of user interaction events. For a customized algorithm module implemented as an algorithm server, OneLabeler allows the module to be directly registered in the visual programming interface.

5.4.2 Customization for Interactive Labeling. OneLabeler internally represents a built-in interactive labeling module with three submodules: data type, label task type, and interface design. Customizing interactive labeling modules can be done by following the instructions above for general-purpose customization or by implementing only a customized submodule. To customize the data type, a developer needs to provide a display function defining how to display a single data object of this data type in the interface design. This display function takes as input the data object's data structure, together with the width and height of the display area for it. For example, to extend OneLabeler to support webpage labeling, a developer may specify that a data object of the webpage data type should be displayed with the HTML iframe tag (see details in Section 6.2). To customize the label task type, a developer needs to provide a display function defining the interaction widgets (e.g., buttons and menus). These widgets are appended to the toolbar in the interface design to support the labeling interactions. For the label task type that requires altering the visual appearance of the data object (e.g., image segmentation requires displaying a segmentation mask), a developer additionally needs to provide a render function defining how the created annotation of a data object should be displayed. To customize the interface design, a developer needs to implement a display function defining the annotation interface. The display function of the annotation interface takes the display function of data type declarations and toolbar and rendering function of label task type declarations as input.

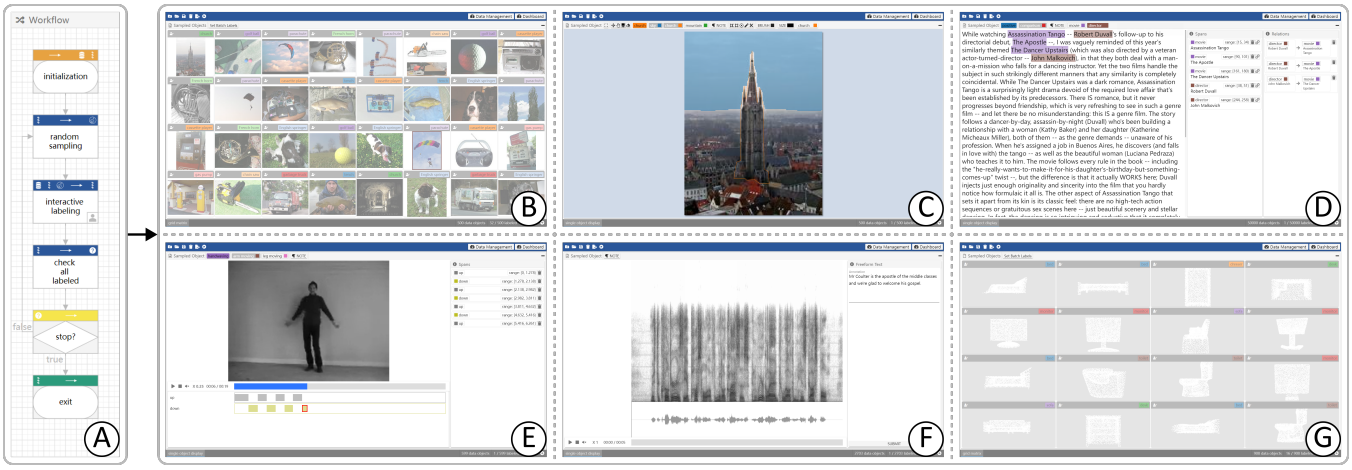

Figure 3: A gallery of labeling tools built with OneLabeler using a single workflow (shown in Ⓐ) but different configurations for the interactive labeling module. The configurations differ in data type, label task type, and interface design. The labeling tools ⒸⒹⒺ all support classification, multi-label classification, and freeform text annotation, while additionally supporting label task types that are specialized to the corresponding data type (e.g., object detection for image). (A) A minimal data labeling template that can be used to generate the six labeling tools ⒷⒸⒹⒺⒻⒼ, with some modifications to node configurations for each labeling tool; (B) An image classification tool; (C) A multi-task image labeling tool (additionally support object detection, and segmentation); (D) A multi-task text labeling tool (additionally support span tagging, and span relation annotation); (E) A multi-task video labeling tool (additionally support span tagging); (F) An audio freeform text annotation tool; (G) A point cloud classification tool.

5.4.3 Template Customization. To extend OneLabeler with a customized workflow template, a developer can either provide a TypeScript/JavaScript file declaring the workflow as an object or an equivalent JSON file (see Appendix C for an example).

5.5 Workflow Compiler

OneLabeler enables a workflow to be compiled into an installer of the specified labeling tool through button-clicking in the visual programming interface. The compilation is conducted by the backend of OneLabeler that holds a mirror of OneLabeler's source code. Upon receiving the compilation request, the backend hard-codes the workflow in the request into the source code mirror. Then, the backend filters out from the mirror the dead-code irrelevant of the labeling tool declared by the workflow (e.g., the built-in modules of OneLabeler not used by the labeling tool and the code responsible for OneLabeler's workflow editing/compilation). The modified and filtered version of OneLabeler's source code is then compiled into the declared labeling tool's installer using Electron Packager[5]. The installer installs the labeling tool as desktop software.

6 CASE STUDY

OneLabeler builds in various implementations of common modules in data labeling tools as introduced in Section 5, which can directly lead to a wide range of data labeling tools with appropriate configurations. To demonstrate the effectiveness of OneLabeler in supporting the development of data labeling tools, this section presents ten sample data labeling tools built with OneLabeler.

We first introduce a gallery of six basic labeling tools based on built-in data types and label tasks. Without changing the topology of a simple workflow template in Fig. 3A, the six example tools can be directly generated and can reproduce major functionalities of popular open-source labeling tools that cover various data types (including image, text, video, audio, and point cloud) and labeling tasks (such as classification, multi-label classification, freeform text annotation, segmentation, span tagging, and span relation).

Moreover, following the API of modules as described in Section 4, developers can also extend the capabilities of OneLabeler by providing new implementations as plugins. To further demonstrate the expressiveness and extensibility of OneLabeler, we showcase four more examples of advanced data labeling and interactive machine learning applications that integrate additional algorithm and interface modules, requiring more complicated workflows.

6.1 A Gallery of Basic Labeling Tools

In Fig. 3, we present six examples of basic labeling tools without machine assistance, based on the same workflow template in Fig. 3A. With corresponding modifications to the configurations of the interactive labeling module, concerning data type, label task type, and interface design, we can achieve specific data labeling tools that target various tasks such as classification (Fig. 3B), segmentation (Fig. 3C), span tagging (Fig. 3D) in different application domains including image (Fig. 3B and 3C), text (Fig. 3D), video (Fig. 3E), audio (Fig. 3F), and point cloud (Fig. 3G). In Fig. 3, we load public datasets to the created labeling tools, including Imagenette[6] (Fig. 3B

[5]https://github.com/electron/electron-packager

[6]https://www.tensorflow.org/datasets/catalog/imagenette

and Fig. 3C), IMDb reviews [47] (Fig. 3D), KTH Action [60] (Fig. 3E), LibriSpeech [54] (Fig. 3F), and ModelNet [74] (Fig. 3G).

The example tools can reproduce the capabilities of some existing data-labeling systems in public use. For example, the labeling tool in Fig. 3C supports image classification, multi-label classification, freeform text annotation, object detection (with polygon or bounding box), and segmentation. These functionalities cover the major functionalities of LabelImg [44] (for object detection with bounding box) and LabelMe [59] (for object detection with polygon).

Similarly, using the text labeling tool in Fig. 3D, major features of Doccano [50] can be reproduced. Doccano supports labeling of image classification and multiple text-related label tasks, including named entity recognition, sentiment analysis, translation, and text to SQL. With the text labeling tool in Fig. 3D, named entity recognition can be done by span tagging, sentiment analysis can be done by classification, while translation and text to SQL can both be done by freeform text labeling.

6.2 Customizing a Webpage Data Type

OneLabeler can be extended with customized data types. A customized data type can be implemented as a submodule of the interactive labeling module (Section 5.4). Customizing a data type requires specifying how a single data object should be displayed. For example, to add a new "webpage" data type, one can supply the following JavaScript code (by putting the code file in the data type folder of OneLabeler system). The API design of data type declaration is compatible with Vue.js[7], a popular front-end development framework, enabling developers to utilize their previous programming experience to develop OneLabeler's plugins.

```
/** The declaration of webpage data type. */
export default {
  /** The label of the data type to appear in the menu. */
  label: 'webpage',
  /** Inputs to the display function. */
  props: ['dataObject', 'width', 'height'],
  /** Render a webpage using HTML iframe element. */
  render: (h, { props }) => (
    h('iframe', {
      attrs: {
        width: props.width,
        height: props.height,
        src: props.dataObject.src,
      },
    })
  ),
}
```

With the corresponding data type declaration, OneLabeler can support a new data type with tasks independent of data types, such as classification, multi-label classification, and freeform-text annotation. For example, using the webpage data type, we can build a basic webpage classification tool as shown in Fig 4. This tool origins from the same workflow as Fig. 3A except that the data type is set to the newly customized webpage data type.

Developers can further develop data-type-specific label tasks with additional code. For example, span tagging is originally targeted at text, video, and audio data types. By specifying how to select a text "span" in HTML text nodes inside a webpage, the developer can make span tagging applicable to webpages.

[7]https://vuejs.org/guide/extras/render-function.html

Figure 4: A classification tool for the customized webpage data type.

6.3 Machine-Aided Multi-Task Text Labeling

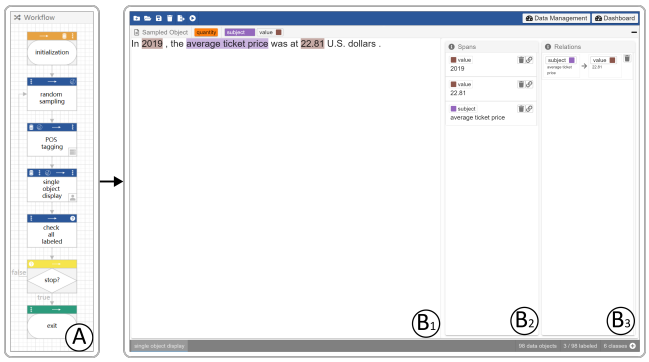

Figure 5: Building a machine-aided multi-task text labeling tool: (A) User-configured workflow for the tool. The interface of generated multi-task text labeling tool: (B_1) A labeling panel showing the sentence for the annotator to conduct interactive labeling; (B_2) A list of created named entities; (B_3) A list of created entity relations.

Here we introduce a configured text labeling tool that incorporates machine assistance. This tool is motivated by a data labeling scenario concerned in Cui et al. [20]'s supervised technique that learns from a human-annotated text dataset. The data objects are quantitative statements. The label tasks require annotating the statement type (i.e., document classification), the subject and value words in each sentence (i.e., span tagging), and the correspondence between subject and value words (i.e., span relation detection). In this usage scenario, the three types of text labels described above are required to be assigned to sentences of quantitative statements. Therefore, we demonstrate the creation of a tool that enables annotators to provide three types of labels: document classification, span tagging, and span relation detection.

In this application, the concerned data objects are text (i.e., strings), and the label tasks are to (1) assign one of four class categories ("proportion", "quantity", "change", and "rank") to the text to mark what type of statement it belongs to, (2) extract named entities from text and associate with one of two class categories ("subject" and "value"), and (3) link pairs of named entities if they are related subject and value.

Semi-automatic labeling [2, 29, 85] is a common technique in the literature for efficient annotation. To incorporate this idea in the labeling tool, we can start from the simple labeling workflow template as shown in Fig. 3A, add a default labeling module, and configure it to be implemented with the built-in POS tagger. Specifically, the developer may decide to use a predictive model to detect tentative text spans that belong to the subject and value words. In this way, the annotator may save many efforts when the predictive model is accurate, as the annotator only needs to delete false positive detections and create the missing false negative spans. This functionality can be achieved with a POS tagger built inside OneLabeler that automatically detect numeric values. Moreover, if the developer is not satisfied with the built-in POS tagger, the developer can also implement a customized default labeling module. Appendix D shows the needed code for the developer to customize a default labeling module for text span detection with a noun detector based on POS tagging.

Fig. 5A shows the visually programmed workflow for this labeling tool. The resulting workflow of the generated labeling tool starts from a random sampling of data objects (**Data Object Selection**). To save the efforts of annotating text spans, a machine-learned POS tagger is applied to extract phrases denoting values from the sentence (**Default Labeling**). Within a labeling panel that displays a single sentence, the user can edit its class category (Fig. 5B_1), use brush to create named entities (Fig. 5B_2), and link entities to create entity relation annotation (Fig. 5B_3) (**Interactive Labeling**).

6.4 Mixed-Initiative Image Classification

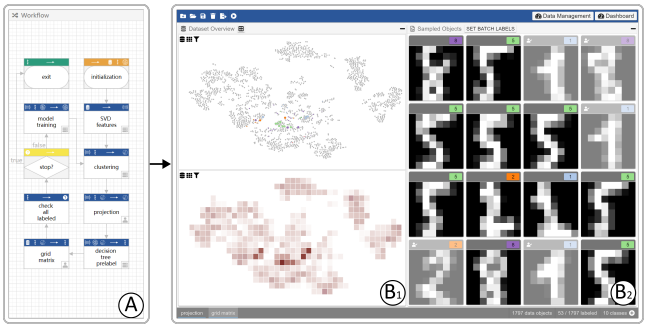

Figure 6: Building a mixed-initiative image classification tool: (A) User-configured workflow for the tool. The interface of generated image classification tool: (B_1) A projection of data object features allowing annotators to use lasso selection for data object selection; (B_2) A labeling panel showing images for an annotator to conduct interactive labeling.

Image classification is a common data labeling application scenario. We demonstrate that OneLabeler allows fast prototyping of an image classification tool that features mixed-initiative sampling (with clustering and interactive projection) and default labeling (with decision tree).

To support efficient annotation, aside from semi-automatic labeling mentioned above, cluster-based labeling (e.g., [19, 65]) is also a common technique that involves algorithm and interface design. In the labeling tool, the developer may decide to incorporate these two ideas by integrating: (1) default labeling so that annotators only need to do correction (i.e., semi-automatic labeling), and (2) mixed-initiative sampling for selecting similar data objects so that multiple data objects sharing the same label may be selected together, allowing one label to be assigned to them simultaneously. The mixed-initiative sampling may be accomplished by joining a data object selection module implemented with a clustering algorithm and a projection view that enables annotators to initiate cluster selection [5, 43].

To build a labeling tool that incorporates these two ideas, the developer may first start from the simple labeling workflow template shown in Fig. 3A. The developer can further edit the workflow and leverage OneLabeler's built-in interface and algorithm modules for data object selection and default labeling.

To support default labeling, the developer adds a default labeling module and uses model prediction as its implementation. To facilitate model prediction, a model training module is needed where the developer may choose a decision tree as the trained model. Additionally, feature extraction is required as a decision tree is not an end-to-end model, and the developer may choose SVD as the feature extraction implementation. To support mixed-initiative sampling, the developer may first add a data object selection module and choose clustering by k-means as an algorithmic implementation for it so that data objects are sampled cluster by cluster, potentially grouping similar data objects. To allow annotators to involve in the selection process, the developer can add another data object selection module implemented with interactive projection.

Fig. 6A shows the finalized visually programmed workflow of this labeling tool and the resulting labeling tool with the UCI handwritten digits dataset [23] loaded. In this application, the concerned data objects are images of handwritten digits, and the task is to assign digit labels (0, 1, ..., 9) to the images.

The resulting workflow of the generated labeling tool starts from singular value decomposition (SVD) as the feature extraction method for processing images (**Feature Extraction**). Mixed-initiative sampling is accomplished by sampling data objects by clustering and then allowing the annotator to revise the samples if needed (**Data Object Selection**). Specifically, a clustering with k-means is run to group and sort data objects where a batch of 16 data objects is sampled each time. The interface displays a projection of the data objects, allowing the annotator to use lasso selection to manually sample data objects (Fig. 6B_1). A decision tree is used to assign default labels for the sampled data objects by machine and annotator (**Default Labeling**). Within a labeling panel with grid matrix layout, the user can edit the labels of sampled data objects (Fig. 6B_2) (**Interactive Labeling**). Each time the user confirms the current samples are correctly labeled, the interim decision tree model is updated to make the default labeling more accurate (**Model Training**).

6.5 Interactive Machine Learning for Chart Image Reverse Engineering

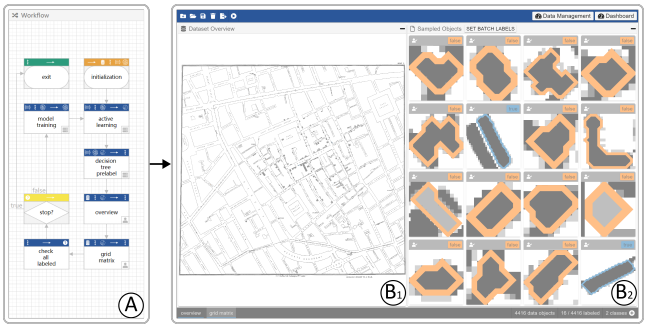

Figure 7: Building an interactive machine learning system for visualization reconstruction: (A) User-configured workflow for the system. The interface of the generated system for visualization reconstruction: (B_1) An overview image that allows the annotator to conduct interactive labeling and overview the visualization reconstruction result; (B_2) A labeling panel showing the geometric objects for the annotator to conduct interactive labeling.

Apart from enhancing conventional labeling processes with algorithm and interface modules, OneLabeler can also support fast prototyping of interactive machine learning systems with customized interactive labeling interfaces. We take the system for visualization reconstruction proposed by Zhang et al. [85] as an example. Their system aims to extract the dataset generating visualizations, and involves reverse engineering chart images by machine-aided classification of geometric objects. We demonstrate the reproduction of the system by OneLabeler.

Following the workflow figure in their paper, we construct a data labeling workflow to reproduce the system. In this application, the concerned data objects are geometric objects (e.g., polygons), and the task is to assign boolean labels (true detection, false detection) to the objects. The geometric object is not a common data type, and OneLabeler has not built-in support for it. Thus, the developer needs to implement a customized data-type plugin that specifies the import handler and display handler for geometric object instances. In the import handler, the image processing algorithm code by Zhang et al. [85] is evoked for extracting geometric objects from images. In the display handler, we write a simple script for displaying polygons.

Active learning selects informative data objects according to label entropy-based criteria. The label entropy is computed from label distribution estimated by a label propagation model (**Data Object Selection**). A decision tree is used to assign default labels (**Default Labeling**). The user can label geometric objects in the grid matrix panel (Fig. $7B_2$) or in the overview image (Fig. $7B_1$). The overview image with extracted polygons embedded is provided to OneLabeler as a plugin for interactive labeling (**Interactive Labeling**). In the model update phase, the label propagation model used for entropy computation and the decision tree model used for default labeling are both updated (**Model Training**).

Fig. 7 shows the visually programmed workflow for this system and the interface of the resulting system reproduced with similar features as in the literature. The reproduced system mimics not only the interface design of the original system (an overview plus an annotation panel) but also major algorithm modules for machine-aided labeling (e.g., active learning, algorithmic default labeling). OneLabeler has the potential to accelerate the research on interactive machine learning by allowing researchers to build research prototypes more efficiently. The customization required writing in total around 200 lines of TypeScript code for plugins (excluding the image processing code provided in the literature [85]). Specifically, around 50 lines of code are needed to implement the data type plugin to support polygon as a customized data type. Around 150 lines of code are needed to implement the overview image that embeds the extracted polygons. In contrast, the front-end of the original system [85] contains around ten thousand lines of code.

7 USER STUDY

We have conducted an in-lab user study to evaluate the usability of OneLabeler for prospective developers of data labeling tools.

7.1 Experimental Design

Participants: We recruited eight participants with programming experience between 9 to 12 years and different levels of experience in data labeling. Five of them (P1, P2, P3, P6, P8) had developed data labeling tools. One (P4) had an ongoing research project that required developing labeling tools. The other two (P5, P7) had participated in labeling tool design and conducted labeling tasks as annotators. The programming expertise of the participants varies. All of them could write back-end or algorithm code using Python or C♯. Three (P1, P2, P8) also had experience in front-end development using JavaScript.

Procedure: The study lasted around two hours for each participant. In each session, we started from asking about the participant's experience on programming and data labeling (~ 15 minutes); next, we gave training on the OneLabeler system (~ 30 minutes); then we asked the participant to build labeling tools using OneLabeler (~ 50 minutes); and finally, we collected feedback on OneLabeler usage experience (~ 15 minutes).

In training, we showed a documentation website on OneLabeler's usage. The documentation introduces basic concepts in OneLabeler and its visual programming functionalities, including workflow editing interactions, static checking, and the labeling tool preview.

We designed the following four tasks for each participant to accomplish with OneLabeler:

- Task 1: Build an image segmentation tool similar to that shown in Fig. 3C without using any built-in template.
- Task 2: Build an image classification tool similar to Fig. 3B based on the workflow built for Task 1.
- Task 3: Build a machine-aided image classification tool similar to Fig. 6A based on the workflow built for Task 2.
- Task 4: Either reproduce the participant's own labeling tool or build a webpage classification tool similar to that shown in Fig. 4.

We designed the first three tasks to evaluate the overall OneLabeler usability. Specifically, Task 1 evaluates how a user builds new workflows from scratch, which reflects the understanding of basic

Table 1: Summary of task completion status and time (in minutes).

Participant	Task 1	Time	Task 2	Time	Task 3	Time	Task 4	Time
P1	complete	9	complete	5	complete with hint (state initialization)	13	complete	12
P2	complete	13	complete	4	complete	17	complete	11
P3	complete with hint (parameter)	15	complete with hint (parameter)	4	complete	12	partially complete	14
P4	complete	29	complete	5	complete with hint (linting message)	25	complete	4
P5	complete	16	complete	3	complete	14	partially complete	12
P6	complete with hint (parameter)	15	complete	9	complete with hint (state initialization)	25	partially complete	22
P7	complete	9	complete	5	complete	18	complete	4
P8	complete	8	complete	8	complete	13	complete	11

concepts and usage of OneLabeler. Task 2 evaluates how a user adapts an existing workflow to a new usage scenario. Task 3 evaluates how a user builds complex bespoke workflows. Moreover, we designed Task 4 to further understand if OneLabeler can be used to build a custom labeling tool for real-world usage. It is an open-ended task. A participant could either reproduce a labeling tool the participant had built before or extend OneLabeler with custom modules to fulfill the desired labeling scenario. For each task, we provided the participants a textual specification of the desired functionality of the labeling tool (see Appendix E).

7.2 Results

7.2.1 Task Completion. Table 1 summarizes task completion status and time for the four tasks. For the first three tasks (T1, T2, T3), all the participants were able to complete them. Out of the 24 trials in total (8 participants × 3 tasks), the participants could finish the tasks correctly in 18 trials without hints from the experimenter. Four participants (P1, P3, P4, P6) needed hints in six trials to make their results exactly the same as the specification given in the instructions. For Task 4, three of the participants (P4, P6, P7) chose to reproduce their own labeling tools: both P4 and P7 developed a text labeling tool similar to Fig. 3D, and P6 built a tool for pairwise comparison of images. The other five participants selected the alternative to build a predefined webpage classification tool. Five out of the eight participants (P1, P2, P4, P7, P8) completed Task 4 independently. The other three participants (P3, P5, P6) completed Task 4 after receiving help from the experimenter. The experimenter helped P6 with the coding part as P6's tool required creating a new data type for image pairs, while P6 was unfamiliar with web development skills needed for the customization. P3 and P5 also had no web development experience. As a result, they could not complete the coding part when building the webpage classification tool. After the experimenter helped them develop the required customized module, they constructed the workflow to achieve the desired labeling tool.

7.2.2 Usability. As new users, all the participants could use OneLabeler to accomplish the assigned tasks to build labeling tools within a short time. This indicates that OneLabeler is easy to learn and easy to use overall. For most participants, building a new workflow from scratch (Task 1) took less than 16 minutes, adapting an existing workflow to a new usage scenario without adding or removing a node (Task 2) took less than 10 minutes, and adding machine assistance to an existing workflow (Task 3) took around 15 minutes. For Task 4, which requires coding, participants with web development experience (P1, P2, P8) accomplished the task independently and efficiently within 15 minutes. In the interview, all the participants commented that OneLabeler was flexible and comprehensive.

During the tasks, all the participants had utilized the static checking function to resolve occurred issues, indicating the usefulness of static checking. During the tasks, participants looked up the documentation website several times. For example, P1, P2, and P6 looked up the definition of the conceptual modules, while P5 and P7 looked up the detailed illustration for the error code "no-uninitialized-inputs" (the Input Initialized rule in Section 4.4). It indicates that the documentation website provides helpful information and that OneLabeler requires some learning.

7.2.3 Potential Improvements. Through the user study, we have identified several opportunities to improve OneLabeler's usability. The completion time of the tasks was affected by misoperations. The fastest participant (P8) spent only 8 minutes on Task 1, while P4 spent 29 minutes on the same task. The reason for P4's lengthened completion time in Task 1 was that P4 clicked the browser's Refresh button by mistake, which caused the previous progress to be lost, and thus P4 had to rebuild the workflow. Similarly, P6 refreshed the webpage and spent 25 minutes on Task 3. This indicates a potential usability improvement by supporting automatic saving of user progress. Additionally, P1 suggested that OneLabeler should provide a function of refining the workflow layout with one click to save developer's efforts in manually arranging nodes.

In the first three tasks, some participants needed hints from the experimenter to make their workflows consistent with the specification. Three of the hints (in Task 1 for P3, Task 1 for P6, Task 2 for P3) were given due to wrong parameter choices in their workflows. For example, the initial workflows created by both P3 and P6 in Task 1 used the default value 48 for the number of data objects sampled by random sampling instead of setting it to 1 as required in the textual specification of the task. These mistakes were caused by carelessness as they forgot to change the configuration from the default value. In real-world usages, these mistakes caused by carelessness may be less common, as the specifications for real-world usages are driven by the developer's own needs, and such mistakes would be easy for the developer to spot. Nevertheless, it may be beneficial to reduce such mistakes via improving the documentation to stress the necessity to try out the labeling tool preview and carefully check if it functions as expected.

The other three hints (in Task 3 for P1, P4, P6) were given as the participants did not recognize that the "model" state needs to be initialized before use. In this case, OneLabeler's static checking provided multiple recommended fixes, as "model" can be initialized either by declaring it as an output of the initialization node (described in Section 5.2.1) or by adding a module that outputs model. The participants either took a wrong path to fix it by adding unnecessary modules, deviating from the specification (P4), or recognized

that they needed to initialize the state, but did not realize that the initialization could be done in the initialization node and got stuck (P1 and P6). This suggests further improvements to provide more understandable error messages to help developers locate and fix errors more effectively.

7.2.4 Future Usage Scenarios. According to the participants, existing data labeling tools usually are not suitable for their own tasks, mainly due to the following three reasons:

(1) **Unconventional labeling tasks**: For example, P1 needed to annotate visual objects with their attributes in visualizations stored as vector and bitmap images. There exist no mature labeling tools for this labeling task.
(2) **Requirement of algorithm support**: To save annotation costs, a labeling tool may integrate algorithm support. The suitable algorithm largely depends on the usage scenario. For example, P3 developed an auto-tracking function to infer bounding boxes of data objects given the bounding box annotations of previous frames. While P3 found an existing labeling tool that provides basic auto-tracking support, P3 could not adapt it to meet the requirement, because it was closed source.
(3) **More control over software changes**: The data labeling requirements can change over time. For example, P2 said *"Even if there exists a piece of software that temporarily meets my requirements, I would be diffident when using it. I may need to abandon it at some point when the requirements can no longer be accommodated by the existing software."*

All the participants agreed that OneLabeler could fill these gaps with its build-in features and customization support that enable efficient building and low-level control on the implementation.

8 DISCUSSION

OneLabeler is an attempt to improve the efficiency of building data labeling tools. The design of OneLabeler follows the rationale that accelerating the building requires promoting reuse of software modules, which in turn requires identifying commonalities in existing data labeling tools. Following this rationale, OneLabeler builds on a system architecture based on common data labeling modules in the literature.

On the expressiveness of conceptual modules: Although there is no guarantee that the primitive modules (i.e., the API definitions) in OneLabeler are capable of representing all possible variations in data labeling tools, there is evidence that they are expressive enough to cover a wide range of interesting variations. Firstly, the coding process that leads to the modules provides evidence that the modules are expressive enough, as they conceptually can represent at least the labeling-related modules in the 33 related papers. Moreover, the case study provides additional evidence that OneLabeler can generate diverse labeling tools in action.

Build in more module implementations: A clear direction to enhance OneLabeler is to extend it with more built-in modules. We aim to provide more built-in options for the lower frequency modules currently underexplored (e.g., quality assurance, stoppage analysis, and label ideation). Especially, while quality assurance is not frequently mentioned in the papers we collected, we expect it is important in real-world applications. Additionally, while the conceptual modules of OneLabeler are sufficient to produce diverse data labeling tools, they may not capture all common techniques related to data labeling emerging in all the related fields. Meanwhile, the coding process presented in Section 4 may be reapplied to a larger corpus of literature to improve the coverage of conceptual modules. Through this iterative process of extending the expressive power, we move gradually towards the ultimate goal of making OneLabeler *feature complete*, with various built-in implementations (algorithms or interfaces) available for multifarious real-world usages.

On orthogonality of conceptual modules: While OneLabeler builds on primitive modules, the conceptual modules in the proposed framework are not strictly orthogonal. For example, both the quality assurance module and the interactive labeling module output labels. OneLabeler does not and should not forbid developers to use an implementation of quality assurance for the interactive labeling purpose. We believe that enforcing strict orthogonality in the API definitions is unnecessary. For example, if the interactive labeling module and the quality assurance module were merged into a single module for strict orthogonality, the discoverability of interactive labeling implementations and quality assurance implementations would both decrease. A developer who wants to configure the module into an implementation of interactive labeling would have to pick the implementation out of all the interactive labeling implementations and quality assurance implementations, which is cumbersome.

On discoverability of modules: As the number of built-in module implementations increases, the options provided by OneLabeler may become too many for novice users to discover. We address this potential issue by allowing new implementations plugged into OneLabeler and built-in modules deleted from OneLabeler, as each module implementation in OneLabeler's source code is an individual script file. In this way, we can provide multiple versions of customized distributions of OneLabeler, each with a different set of built-in modules suited for different developers. In addition, OneLabeler can provide advanced search and recommendation functionalities in the module configuration panel to alleviate this issue in the future.

Extension to crowd labeling scenarios: In this work, we have not explicitly considered the usage scenario of crowd labeling workflows. Our framework may accommodate some aspects of crowd labeling workflows, e.g., the task scheme design in crowdsourcing [42] is related to the task design in interactive labeling and post-processing of crowdsourced labels is related to quality assurance. However, others are not captured by the conceptual framework. To support crowdsourcing, OneLabeler needs to build in additional interface and algorithm modules, such as for administrator's tasks, including monitoring label progress, assessing annotators' reliability, and data management. OneLabeler's states also need to be extended to accommodate new modules' inputs and outputs. To integrate with existing infrastructures, we also need to investigate how to distribute a labeling tool created with OneLabeler on crowdsourcing platforms.

Exploring other usages of OneLabeler: While OneLabeler is designed for building labeling tools, as demonstrated in Section 6.5, we envision that OneLabeler may be extended to support interactive machine learning applications other than data labeling. For example, OneLabeler may be extended to interactive model training if additional interface modules are added to OneLabeler to allow end

users to edit the model. Another example is to apply OneLabeler to build customized information retrieval systems (e.g., photo management systems) by adding more modules for data object selection with information retrieval techniques. Additionally, OneLabeler is designed for building standalone labeling tools, while it may also be beneficial to explore building UI widgets for data labeling that can embed on other systems.

On Customization: Customization support is essential to ensure the flexibility of OneLabeler, as revealed by Task 4 in the user study. OneLabeler allows a developer to add customized implementations, as long as the inputs and outputs of the implementations are within OneLabeler supported states. Meanwhile, the customizability of OneLabeler may be extended in future work to support customization on workflow execution and UI appearance. Fine control on parallelizing execution of multiple modules, such as waiting for all or racing, cannot be achieved at the moment. Besides, each interface module currently appears as a window in OneLabeler's interface, and UI widgets outside a window are not customizable. As customization requires textual programming, OneLabeler provides CLI (Section 5.4) to assist customization development, where a developer can start coding based on template code rather than from scratch. To further reduce textual programming efforts required for customization, it is beneficial to build a marketplace for OneLabeler modules, where developers can share their customized modules for others to reuse. It may also be beneficial to further decompose common modules into submodules. Through subdivision, the flexibility for reuse increases, as a developer may customize a submodule instead of an entire module.

9 CONCLUSION

In this paper, we have proposed a conceptual framework for data labeling and the OneLabeler system based on the conceptual framework to support easy building of labeling tools for different usage scenarios. To build the framework, we have identified common states and modules by coding the literature and summarized constraints in composing the modules to build labeling tools. Each modular process can be instantiated as a human, machine, or mixed computation procedure. OneLabeler provides a visual programming interface that uses modules in the conceptual framework as building blocks. It builds in various implementations for reuse so that developers can create labeling tools with no code or less code. It provides static checking and preview functionalities to assist development and debugging. OneLabeler supports customization, allowing developers to extend its capability further. We have demonstrated OneLabeler's expressiveness through a case study of building ten labeling tools. We have further conducted a user study to evaluate its usability and collect feedback from potential users. The user-study results suggest that OneLabeler is easy to learn and enables potential users to build labeling tools efficiently.

ACKNOWLEDGMENTS

This work was conducted when Y. Zhang was an intern at Microsoft Research Asia. Y. Wang is the corresponding author. S. Chen was partially supported by Shanghai Municipal Science and Technology (Major Project 2018SHZDZX01, Sailing Program No.21YF1402900 and No. 21ZR1403300).

REFERENCES

[1] Amazon. 2005. Amazon Mechanical Turk. https://www.mturk.com/
[2] Mykhaylo Andriluka, Jasper R. R. Uijlings, and Vittorio Ferrari. 2018. Fluid Annotation: A Human-Machine Collaboration Interface for Full Image Annotation. In *Proceedings of the ACM International Conference on Multimedia*. ACM, 1957–1966.
[3] Nathaniel Ayewah, William Pugh, David Hovemeyer, J. David Morgenthaler, and John Penix. 2008. Using Static Analysis to Find Bugs. *IEEE Software* 25, 5 (sep 2008), 22–29.
[4] A. Bäuerle, H. Neumann, and T. Ropinski. 2020. Classifier-Guided Visual Correction of Noisy Labels for Image Classification Tasks. *Computer Graphics Forum* 39, 3 (jun 2020), 195–205.
[5] Jürgen Bernard, Marco Hutter, Matthias Zeppelzauer, Dieter Fellner, and Michael Sedlmair. 2018. Comparing Visual-Interactive Labeling with Active Learning: An Experimental Study. *IEEE Transactions on Visualization and Computer Graphics* 24, 1 (Jan. 2018), 298–308.
[6] Jürgen Bernard, Matthias Zeppelzauer, Michael Sedlmair, and Wolfgang Aigner. 2018. VIAL: a unified process for visual interactive labeling. *The Visual Computer* 34, 9 (2018), 1189–1207.
[7] Michael Bloodgood and K. Vijay-Shanker. 2009. A Method for Stopping Active Learning Based on Stabilizing Predictions and the Need for User-Adjustable Stopping. In *Proceedings of the Conference on Computational Natural Language Learning*. Association for Computational Linguistics, 39–47.
[8] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. 2011. D3: Data-Driven Documents. *IEEE Transactions on Visualization and Computer Graphics* 17, 12 (Nov 2011), 2301–2309.
[9] Yuri Boykov, Olga Veksler, and Ramin Zabih. 2001. Fast Approximate Energy Minimization via Graph Cuts. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 23, 11 (2001), 1222–1239.
[10] Steve Branson, Catherine Wah, Florian Schroff, Boris Babenko, Peter Welinder, Pietro Perona, and Serge Belongie. 2010. Visual Recognition with Humans in the Loop. In *Proceedings of the European Conference on Computer Vision*. Springer Berlin Heidelberg, 438–451.
[11] Klaus Brinker. 2003. Incorporating Diversity in Active Learning with Support Vector Machines. In *Proceedings of the International Conference on Machine Learning*. AAAI Press, 59–66.
[12] Carla E. Brodley and Mark A. Friedl. 1999. Identifying Mislabeled Training Data. *Journal of Artificial Intelligence Research* 11, 1 (July 1999), 131–167.
[13] Nicholas J. Bryan, Gautham J. Mysore, and Ge Wang. 2014. ISSE: An Interactive Source Separation Editor. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 257–266.
[14] Maya Cakmak and Andrea L. Thomaz. 2011. Mixed-Initiative Active Learning. In *Proceedings of ICML Workshop on Combining Learning Strategies to Reduce Label Cost*. 5 pages.
[15] Chengliang Chai, Guoliang Li, Ju Fan, and Yuyu Luo. 2020. Crowdsourcing-based Data Extraction from Visualization Charts. In *Proceedings of the IEEE International Conference on Data Engineering*. IEEE, 1814–1817.
[16] Joseph Chee Chang, Saleema Amershi, and Ece Kamar. 2017. Revolt: Collaborative Crowdsourcing for Labeling Machine Learning Datasets. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2334–2346.
[17] Justin Cheng and Michael S. Bernstein. 2015. Flock: Hybrid Crowd-Machine Learning Classifiers. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work & Social Computing*. ACM, 600–611.
[18] Minsuk Choi, Cheonbok Park, Soyoung Yang, Yonggyu Kim, Jaegul Choo, and Sungsoo Ray Hong. 2019. AILA: Attentive Interactive Labeling Assistant for Document Classification through Attention-Based Deep Neural Networks. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 1–12.
[19] Jingyu Cui, Fang Wen, Rong Xiao, Yuandong Tian, and Xiaoou Tang. 2007. EasyAlbum: An Interactive Photo Annotation System Based on Face Clustering and Re-ranking. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 367–376.
[20] Weiwei Cui, Xiaoyu Zhang, Yun Wang, He Huang, Bei Chen, Lei Fang, Haidong Zhang, Jian-Guan Lou, and Dongmei Zhang. 2020. Text-to-Viz: Automatic Generation of Infographics from Proportion-Related Natural Language Statements. *IEEE Transactions on Visualization and Computer Graphics* 26, 1 (jan 2020), 906–916.
[21] Ork de Rooij, Jarke J. van Wijk, and Marcel Worring. 2010. MediaTable: Interactive Categorization of Multimedia Collections. *IEEE Computer Graphics and Applications* 30, 5 (sep 2010), 42–51.
[22] Jia Deng, Olga Russakovsky, Jonathan Krause, Michael Bernstein, Alex Berg, and Li Fei-Fei. 2014. Scalable Multi-Label Annotation. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 3099–3102.
[23] Dheeru Dua and Casey Graff. 2017. UCI Machine Learning Repository. http://archive.ics.uci.edu/ml
[24] Abhishek Dutta and Andrew Zisserman. 2019. The VIA Annotation Software for Images, Audio and Video. In *Proceedings of the ACM International Conference on Multimedia*. ACM, 2276–2279.

[25] Sara Evensen, Chang Ge, and Cagatay Demiralp. 2020. Ruler: Data Programming by Demonstration for Document Labeling. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. Association for Computational Linguistics, 1996–2005.

[26] Explosion. 2017. Prodigy. https://prodi.gy/

[27] Jerry Alan Fails and Dan R. Olsen. 2003. Interactive Machine Learning. In *Proceedings of the International Conference on Intelligent User Interfaces*. ACM, 39–45.

[28] Cristian Felix, Aritra Dasgupta, and Enrico Bertini. 2018. The Exploratory Labeling Assistant: Mixed-Initiative Label Curation with Large Document Collections. In *Proceedings of the ACM Symposium on User Interface Software and Technology*. ACM, 153–164.

[29] Humberto S. Garcia Caballero, Michel A. Westenberg, Binyam Gebre, and Jarke J. van Wijk. 2019. V-Awake: A Visual Analytics Approach for Correcting Sleep Predictions from Deep Learning Models. *Computer Graphics Forum* 38, 3 (2019), 1–12.

[30] Gaudenz Halter, Rafael Ballester-Ripoll, Barbara Flueckiger, and Renato Pajarola. 2019. VIAN: A Visual Annotation Tool for Film Analysis. *Computer Graphics Forum* 38, 3 (2019), 119–129.

[31] Benjamin Höferlin, Rudolf Netzel, Markus Höferlin, Daniel Weiskopf, and Gunther Heidemann. 2012. Inter-Active Learning of Ad-Hoc Classifiers for Video Visual Analytics. In *Proceedings of the IEEE Conference on Visual Analytics Science and Technology*. IEEE, 23–32.

[32] Steven C.H. Hoi and Michael R. Lyu. 2005. A Semi-Supervised Active Learning Framework for Image Retrieval. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, Vol. 2. IEEE, 302–309.

[33] Xian-Sheng Hua and Guo-Jun Qi. 2008. Online Multi-Label Active Annotation: Towards Large-Scale Content-Based Video Search. In *Proceedings of the ACM International Conference on Multimedia*. ACM, 141–150.

[34] Richard M. Karp. 1960. A Note on the Application of Graph Theory to Digital Computer Programming. *Information and Control* 3, 2 (jun 1960), 179–190.

[35] Caitlin Kelleher and Randy Pausch. 2005. Lowering the Barriers to Programming: A Taxonomy of Programming Environments and Languages for Novice Programmers. *Comput. Surveys* 37, 2 (jun 2005), 83–137.

[36] Kostiantyn Kucher, Carita Paradis, Magnus Sahlgren, and Andreas Kerren. 2017. Active Learning and Visual Analytics for Stance Classification with ALVA. *ACM Transactions on Interactive Intelligent Systems* 7, 3, Article 14 (Oct. 2017), 31 pages.

[37] Todd Kulesza, Saleema Amershi, Rich Caruana, Danyel Fisher, and Denis Charles. 2014. Structured Labeling for Facilitating Concept Evolution in Machine Learning. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 3075–3084.

[38] Labelbox, Inc. 2018. Labelbox. https://labelbox.com/

[39] David Ledo, Steven Houben, Jo Vermeulen, Nicolai Marquardt, Lora Oehlberg, and Saul Greenberg. 2018. Evaluation Strategies for HCI Toolkit Research. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 1–17.

[40] Fritz Lekschas, Brant Peterson, Daniel Haehn, Eric Ma, Nils Gehlenborg, and Hanspeter Pfister. 2020. Peax: Interactive Visual Pattern Search in Sequential Data Using Unsupervised Deep Representation Learning. *Computer Graphics Forum* 39, 3 (jun 2020), 167–179.

[41] David D. Lewis and Jason Catlett. 1994. Heterogeneous Uncertainty Sampling for Supervised Learning. In *Proceedings of the International Conference on Machine Learning*. Morgan Kaufmann Publishers Inc., 148–156.

[42] Guoliang Li, Jiannan Wang, Yudian Zheng, and Michael J. Franklin. 2016. Crowdsourced Data Management: A Survey. *IEEE Transactions on Knowledge and Data Engineering* 28, 9 (sep 2016), 2296–2319.

[43] Hongsen Liao, Li Chen, Yibo Song, and Hao Ming. 2016. Visualization-Based Active Learning for Video Annotation. *IEEE Transactions on Multimedia* 18, 11 (Nov. 2016), 2196–2205.

[44] Tzuta Lin. 2015. LabelImg. https://github.com/tzutalin/labelImg

[45] Dong Liu, Meng Wang, Xian-Sheng Hua, and Hong-Jiang Zhang. 2009. Smart Batch Tagging of Photo Albums. In *Proceedings of the ACM International Conference on Multimedia*. ACM, 809–812.

[46] Shixia Liu, Changjian Chen, Yafeng Lu, Fangxin Ouyang, and Bin Wang. 2019. An Interactive Method to Improve Crowdsourced Annotations. *IEEE Transactions on Visualization and Computer Graphics* 25, 1 (Jan. 2019), 235–245.

[47] Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. 2011. Learning Word Vectors for Sentiment Analysis. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*. Association for Computational Linguistics, 142–150.

[48] Gonzalo Gabriel Méndez, Miguel A. Nacenta, and Sebastien Vandenheste. 2016. iVoLVER: Interactive Visual Language for Visualization Extraction and Reconstruction. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 4073–4085.

[49] Microsoft. 2017. VoTT: Visual Object Tagging Tool. https://github.com/microsoft/VoTT

[50] Hiroki Nakayama, Takahiro Kubo, Junya Kamura, Yasufumi Taniguchi, and Xu Liang. 2018. doccano: Text Annotation Tool for Human. https://github.com/doccano/doccano

[51] Curtis Northcutt, Lu Jiang, and Isaac Chuang. 2021. Confident Learning: Estimating Uncertainty in Dataset Labels. *Journal of Artificial Intelligence Research* 70 (may 2021), 1373–1411.

[52] Allard Oelen, Markus Stocker, and Sören Auer. 2021. Crowdsourcing Scholarly Discourse Annotations. In *Proceedings of the International Conference on Intelligent User Interfaces*. ACM, 464–474.

[53] Jose Gustavo S. Paiva, William Robson Schwartz, Helio Pedrini, and Rosane Minghim. 2015. An Approach to Supporting Incremental Visual Data Classification. *IEEE Transactions on Visualization and Computer Graphics* 21, 1 (Jan. 2015), 4–17.

[54] Vassil Panayotov, Guoguo Chen, Daniel Povey, and Sanjeev Khudanpur. 2015. Librispeech: An ASR corpus based on public domain audio books. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*. IEEE, 5206–5210.

[55] Alexander Ratner, Stephen H. Bach, Henry Ehrenberg, Jason Fries, Sen Wu, and Christopher Ré. 2017. Snorkel: Rapid Training Data Creation with Weak Supervision. *Proceedings of the VLDB Endowment* 11, 3 (Nov. 2017), 269–282.

[56] Donghao Ren, Tobias Hollerer, and Xiaoru Yuan. 2014. iVisDesigner: Expressive Interactive Design of Information Visualizations. *IEEE Transactions on Visualization and Computer Graphics* 20, 12 (Dec. 2014), 2092–2101.

[57] Tim Rietz and Alexander Maedche. 2021. Cody: An AI-Based System to Semi-Automate Coding for Qualitative Research. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, Article 394, 14 pages.

[58] Olga Russakovsky, Li-Jia Li, and Li Fei-Fei. 2015. Best of both worlds: Human-machine collaboration for object annotation. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, 2121–2131.

[59] Bryan C. Russell, Antonio Torralba, Kevin P. Murphy, and William T. Freeman. 2008. LabelMe: A Database and Web-Based Tool for Image Annotation. *International Journal of Computer Vision* 77, 1-3 (2008), 157–173.

[60] Christian Schüldt, Ivan Laptev, and Barbara Caputo. 2004. Recognizing Human Actions: A Local SVM Approach. In *Proceedings of the International Conference on Pattern Recognition*, Vol. 3. IEEE, 32–36.

[61] Burr Settles. 2009. *Active Learning Literature Survey*. Technical Report. University of Wisconsin–Madison.

[62] Burr Settles. 2011. Closing the Loop: Fast, Interactive Semi-supervised Annotation with Queries on Features and Instances. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 1467–1478.

[63] Xindi Shang, Donglin Di, Junbin Xiao, Yu Cao, Xun Yang, and Tat-Seng Chua. 2019. Annotating Objects and Relations in User-Generated Videos. In *Proceedings of the International Conference on Multimedia Retrieval*. ACM, 279–287.

[64] Bongwon Suh and Benjamin B. Bederson. 2007. Semi-Automatic Photo Annotation Strategies Using Event Based Clustering and Clothing Based Person Recognition. *Interacting with Computers* 19, 4 (July 2007), 524–544.

[65] Jinhui Tang, Qiang Chen, Meng Wang, Shuicheng Yan, Tat-Seng Chua, and Ramesh Jain. 2013. Towards Optimizing Human Labeling for Interactive Image Tagging. *ACM Transactions on Multimedia Computing, Communications, and Applications* 9, 4, Article 29 (Aug. 2013), 18 pages.

[66] Yuandong Tian, Wei Liu, Rong Xiao, Fang Wen, and Xiaoou Tang. 2007. A Face Annotation Framework with Partial Clustering and Interactive Labeling. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, 1–8.

[67] Maxim Tkachenko, Mikhail Malyuk, Nikita Shevchenko, Andrey Holmanyuk, and Nikolai Liubimov. 2020. Label Studio: Data labeling software. https://github.com/heartexlabs/label-studio

[68] Luis von Ahn and Laura Dabbish. 2004. Labeling Images with a Computer Game. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 319–326.

[69] Luis von Ahn and Laura Dabbish. 2008. Designing Games with a Purpose. *Commun. ACM* 51, 8 (Aug. 2008), 58–67.

[70] Luis von Ahn, Ruoran Liu, and Manuel Blum. 2006. Peekaboom: A Game for Locating Objects in Images. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 55–64.

[71] Luis von Ahn, Benjamin Maurer, Colin McMillen, David Abraham, and Manuel Blum. 2008. reCAPTCHA: Human-Based Character Recognition via Web Security Measures. *Science* 321, 5895 (aug 2008), 1465–1468.

[72] Guotai Wang, Maria A. Zuluaga, Wenqi Li, Rosalind Pratt, Premal A. Patel, Michael Aertsen, Tom Doel, Anna L. David, Jan Deprest, Sébastien Ourselin, and Tom Vercauteren. 2019. DeepIGeoS: A Deep Interactive Geodesic Framework for Medical Image Segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 41, 7 (jul 2019), 1559–1572.

[73] Meng Wang and Xian-Sheng Hua. 2011. Active Learning in Multimedia Annotation and Retrieval: A Survey. *ACM Transactions on Intelligent Systems and Technology* 2, 2, Article 10 (Feb. 2011), 21 pages.

[74] Zhirong Wu, Shuran Song, Aditya Khosla, Fisher Yu, Linguang Zhang, Xiaoou Tang, and Jianxiong Xiao. 2015. 3D ShapeNets: A Deep Representation for

Volumetric Shapes. In *IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, 1912–1920.

[75] Shouxing Xiang, Xi Ye, Jiazhi Xia, Jing Wu, Yang Chen, and Shixia Liu. 2019. Interactive Correction of Mislabeled Training Data. In *Proceedings of the IEEE Conference on Visual Analytics Science and Technology*. IEEE, 57–68.

[76] Zuobing Xu, Ram Akella, and Yi Zhang. 2007. Incorporating Diversity and Density in Active Learning for Relevance Feedback. In *Proceedings of the European Conference on IR Research*. Springer-Verlag, 246–257.

[77] Yang Yang, Bin B. Zhu, Rui Guo, Linjun Yang, Shipeng Li, and Nenghai Yu. 2008. A Comprehensive Human Computation Framework: With Application to Image Labeling. In *Proceedings of the ACM International Conference on Multimedia*. ACM, 479–488.

[78] Hao Ye, Weiyuan Shao, Hong Wang, Jianqi Ma, Li Wang, Yingbin Zheng, and Xiangyang Xue. 2016. Face Recognition via Active Annotation and Learning. In *Proceedings of the ACM International Conference on Multimedia*. ACM, 1058–1062.

[79] Bowen Yu and Claudio T. Silva. 2017. VisFlow - Web-based Visualization Framework for Tabular Data with a Subset Flow Model. *IEEE Transactions on Visualization and Computer Graphics* 23, 1 (Jan. 2017), 251–260.

[80] Jan Zahálka and Marcel Worring. 2014. Towards Interactive, Intelligent, and Integrated Multimedia Analytics. In *Proceedings of the IEEE Conference on Visual Analytics Science and Technology*. IEEE, 3–12.

[81] Jan Zahálka, Marcel Worring, and Jarke J. van Wijk. 2020. II-20: Intelligent and pragmatic analytic categorization of image collections. *IEEE Transactions on Visualization and Computer Graphics* 27, 2 (2020), 422–431.

[82] Lishi Zhang, Chenghan Fu, and Jia Li. 2018. Collaborative Annotation of Semantic Objects in Images with Multi-Granularity Supervisions. In *Proceedings of the ACM International Conference on Multimedia*. ACM, 474–482.

[83] Tianye Zhang, Haozhe Feng, Wei Chen, Zexian Chen, Wenting Zheng, Xiao-Nan Luo, Wenqi Huang, and Anthony K. H. Tung. 2021. ChartNavigator: An Interactive Pattern Identification and Annotation Framework for Charts. *IEEE Transactions on Knowledge and Data Engineering* (2021), 1–1.

[84] Yexun Zhang, Wenbin Cai, Wenquan Wang, and Ya Zhang. 2017. Stopping Criterion for Active Learning with Model Stability. *ACM Transactions on Intelligent Systems and Technology* 9, 2, Article 19 (Oct. 2017), 26 pages.

[85] Yu Zhang, Bob Coecke, and Min Chen. 2021. MI3: Machine-Initiated Intelligent Interaction for Interactive Classification and Data Reconstruction. *ACM Transactions on Interactive Intelligent Systems* 11, 3–4, Article 18 (Aug. 2021), 34 pages.

[86] Yu Zhang, Martijn Tennekes, Tim de Jong, Lyana Curier, Bob Coecke, and Min Chen. 2021. Using Simulation to Aid the Design and Optimization of Intelligent User Interfaces for Quality Assurance Processes in Machine Learning. arXiv:2104.01129 [cs.HC]

[87] Jingbo Zhu, Huizhen Wang, Eduard Hovy, and Matthew Ma. 2010. Confidence-Based Stopping Criteria for Active Learning for Data Annotation. *ACM Transactions on Speech and Language Processing* 6, 3, Article 3 (April 2010), 24 pages.

[88] Xingquan Zhu, Xindong Wu, and Qijun Chen. 2003. Eliminating Class Noise in Large Datasets. In *Proceedings of the International Conference on Machine Learning*. AAAI Press, 920–927.

A CODING DETAILS

In the coding process, we first extract phrases from the flowchart figures. Each phrase forms a preliminary code. The preliminary codes are first categorized into states and modules. Then, we categorize preliminary codes into themes. Finally, we filter out irrelevant themes and merge related themes into the final codes. In the following, we introduce details of the coding process, including the themes we excluded, the occurrences of final codes in the 36 flowcharts we have coded, and an example of coding a recent labeling tool.

A.1 Themes

State themes: The 163 preliminary state codes are grouped into 10 themes. 5 of the themes are directly included in the final codes as described in Section 4, including **Data Objects**, **Labels**, **Samples**, **Model**, and **Features**. The other 5 themes we identified are (marked with frequency):

- **Labeled data**: the dataset together with (partial) labels (22/163).
- **Search related**: the states related to searching (7/163).
- **Knowledge**: the states related to annotators' knowledge (4/163).
- **Data source**: the source where data objects are extracted (2/163).
- **Others**: the other states not fitting previous themes (9/163).

The theme "labeled data" maps to two final codes "data objects" and "labels" as "labeled data" covers both aspects. There are 2 cases where we group one preliminary code into two themes, where the preliminary code contains the word "with" (e.g., "selected subsets with default labels" in Zhang et al. [85]'s Figure 3).

Modules themes: The 188 preliminary are grouped into 16 themes. 7 of the themes are directly included in the final codes as described in Section 4, including **data object selection**, **model training**, **feature extraction**, **default labeling**, **quality assurance**, **stoppage analysis**, and **label ideation**. The other 9 themes we identified are (marked with frequency):

- **User labeling**: the process where annotators create/edit data labels (33/188).
- **Labeling interface**: the process to present data (possibly with labels) to annotators for creating/editing labels (17/188).
- **Preprocessing**: the application-specific process to precompute auxiliary data structures used in the data labeling tool (11/188).
- **Model understanding**: the process to understand the prediction criteria or quality of the model (5/188).
- **Data exploration**: the process to browse the dataset for the purpose of familiarizing with the dataset (4/188).
- **Postprocessing**: the process to postprocess data labeling results (i.e., labels) for application-specific needs (4/188).
- **Data collection**: the process to collect data or enlarge the dataset (4/188).
- **User modeling**: the process to model user and interpret user intent (2/188).
- **Others**: the other processes not fitting previous themes (13/188).

The themes "user labeling" and "labeling interface" are merged into the final code "interactive labeling" as the former theme is related to user interaction in interactive labeling, and the latter is related to the interface supporting interactive labeling. For module coding, there are 8 edge cases where we group one preliminary code into multiple themes. It happens typically when the phrase contains the word "and", such as "sampling and annotation" in Liao et al. [43]'s Figure 1. For the preliminary codes for both states and modules, we categorize them into "others" when the preliminary code is hard to categorize, typically when the code is too verbose (e.g., "adding to labeled dataset" in Zhang et al. [85]'s Figure 3) and when the code is too application-specific (e.g., "superpixels" in Zhang et al. [82]'s Figure 2).

A.2 State and Module Occurrences

Table 2: Occurrences of states in the literature. The final codes are abbreviated by initials (e.g., "DO" refers to "data objects"). Slash (/) is used in final codes when the flowchart contains no states or the flowchart's states are excluded in the coding procedure.

ID	Paper	Venue	Year	Figure Index	Final Codes
1	Fails2003Interactive [27]	IUI	2003	2	/
2	Hoi2005Semi [32]	CVPR	2005	1	DO, L, S
3	Tian2007Face [66]	CVPR	2007	1	DO, L, S, F
4	Cui2007EasyAlbum [19]	CHI	2007	13	DO, L, F
5	Hua2008Online [33]	MM	2008	2	DO
6	Rooij2010MediaTable [21]	CG&A	2010	3	DO, L, S
7	Wang2011Active [73]	TIST	2011	3	DO, L, M
8	Wang2011Active [73]	TIST	2011	5	DO, L, M
9	Wang2011Active [73]	TIST	2011	7	DO
10	Hoeferlin2012Inter [31]	VAST	2012	1(b)	DO, L, M
11	Tang2013Towards [65]	TOMM	2013	2	DO, L, S
12	Zahalka2014Towards [80]	VAST	2014	9	DO, L, M, F
13	Bryan2014ISSE [13]	CHI	2014	5	DO
14	Paiva2015Approach [53]	TVCG	2015	1	DO
15	Russakovsky2015Best [58]	CVPR	2015	2	L, S
16	Liao2016Visualization [43]	TMM	2016	1	DO, L, F
17	Ye2016Face [78]	MM	2016	1	DO, L, S
18	Kucher2017Active [36]	TIIS	2017	1	DO, L, M
19	Ratner2017Snorkel [55]	VLDB	2017	2	DO, L, M
20	Bernard2018VIAL [6]	TVC	2018	1	DO, L, S
21	Bernard2018VIAL [6]	TVC	2018	2	DO, L, S
22	Felix2018Exploratory [28]	UIST	2018	1	L
23	Zhang2018Collaborative [82]	MM	2018	2	DO, L
24	Shang2019Annotating [63]	ICMR	2019	2	/
25	Xiang2019Interactive [75]	VAST	2019	2	DO, L, S
26	Liu2019Interactive [46]	TVCG	2019	3	DO, L
27	Choi2019AILA [18]	CHI	2019	6	/
28	Wang2019DeepIGeoS [72]	TPAMI	2019	1	DO, L
29	Halter2019VIAN [30]	CGF	2019	2	DO, L, S, F
30	Evensen2020Ruler [25]	EMNLP	2020	2	DO, S, M
31	Baeuerle2020Classifier [4]	CGF	2020	1	DO, L
32	Lekschas2020Peax [40]	CGF	2020	3	DO, L
33	Oelen2021Crowdsourcing [52]	IUI	2021	2	DO, L, M
34	Rietz2021Cody [57]	CHI	2021	4	DO, L, M
35	Zhang2021ChartNavigator [83]	TKDE	2021	1	DO, F
36	Zhang2021MI3 [85]	TIIS	2021	3	DO, L, S, M

Table 2 summarizes the occurrences of final codes for states in the 36 flowchart figures. In the table, for each flowchart figure, if the flowchart contains at least one phrase that corresponds to a specific final code, we add the final code to the "Final Codes" column. When a flowchart figure contains no phrases that refer to inputs/outputs, or that the phrases that refer to inputs/outputs are excluded in the processing of generating final codes from the themes, we mark the figure's "Final Codes" empty, represented by a slash (/). Similarly, Table 3 summarizes the occurrences of final codes for modules in the 36 flowchart figures.

A.3 A Coding Example

AILA [18] supports labeling document classification. Figure 6 in its paper is a flowchart. The figure contains nested blocks, and we use a hyphen to denote the nesting relation in the preliminary code. For example, "preprocessing - stemming" refers to a block named "stemming" placed in a block named "preprocessing". We decompose the nested blocks to the lowest level instead of treating the root block, such as "preprocessing", as a single phrase because different nested

Table 3: Occurrences of modules in the literature. The final codes are abbreviated by initials (e.g., "IL" refers to "interactive labeling").

ID	Paper	Venue	Year	Figure Index	Final Codes
1	Fails2003Interactive [27]	IUI	2003	2	IL, MT, DL
2	Hoi2005Semi [32]	CVPR	2005	1	IL, DOS, MT
3	Tian2007Face [66]	CVPR	2007	1	IL, DOS, FE
4	Cui2007EasyAlbum [19]	CHI	2007	13	IL, DOS, FE
5	Hua2008Online [33]	MM	2008	2	DOS
6	Rooij2010MediaTable [21]	CG&A	2010	3	IL, DOS, LI
7	Wang2011Active [73]	TIST	2011	3	IL, DOS, MT
8	Wang2011Active [73]	TIST	2011	5	IL, DOS, MT
9	Wang2011Active [73]	TIST	2011	7	IL, DOS, MT, QA
10	Hoeferlin2012Inter [31]	VAST	2012	1(b)	IL, DOS, MT
11	Tang2013Towards [65]	TOMM	2013	2	IL, DOS, SA
12	Zahalka2014Towards [80]	VAST	2014	9	IL, MT, FE, LI
13	Bryan2014ISSE [13]	CHI	2014	5	IL, MT
14	Paiva2015Approach [53]	TVCG	2015	1	IL, DOS, MT, FE, DL
15	Russakovsky2015Best [58]	CVPR	2015	2	IL, DOS, DL
16	Liao2016Visualization [43]	TMM	2016	1	IL, DOS, MT
17	Ye2016Face [78]	MM	2016	1	MT
18	Kucher2017Active [36]	TIIS	2017	1	IL
19	Ratner2017Snorkel [55]	VLDB	2017	2	MT
20	Bernard2018VIAL [6]	TVC	2018	1	IL, DOS, FE, DL, SA
21	Bernard2018VIAL [6]	TVC	2018	2	IL, DOS, MT, FE, DL
22	Felix2018Exploratory [28]	UIST	2018	1	IL, QA, LI
23	Zhang2018Collaborative [82]	MM	2018	2	IL, DL
24	Shang2019Annotating [63]	ICMR	2019	2	IL
25	Xiang2019Interactive [75]	VAST	2019	2	IL, DOS, DL
26	Liu2019Interactive [46]	TVCG	2019	3	IL, DOS, FE, DL, QA
27	Choi2019AILA [18]	CHI	2019	6	IL, DOS, FE
28	Wang2019DeepIGeoS [72]	TPAMI	2019	1	IL, DL, SA
29	Halter2019VIAN [30]	CGF	2019	2	IL, FE, DL
30	Evensen2020Ruler [25]	EMNLP	2020	2	IL, DOS, MT
31	Baeuerle2020Classifier [4]	CGF	2020	1	MT, DL, QA
32	Lekschas2020Peax [40]	CGF	2020	3	IL, DOS
33	Oelen2021Crowdsourcing [52]	IUI	2021	2	IL, DOS, DL
34	Rietz2021Cody [57]	CHI	2021	4	IL, MT, DL
35	Zhang2021ChartNavigator [83]	TKDE	2021	1	DOS, FE
36	Zhang2021MI3 [85]	TIIS	2021	3	IL, DOS, MT, FE, DL, QA, SA

Table 4: Coding the flowchart in AILA [18]. The final codes are abbreviated by initials. A slash (/) in the final code refers to the case that the theme is excluded from the final code.

Preliminary Code	Theme	Final Code
preprocessing - stemming	feature extraction	FE
preprocessing - bag of words	feature extraction	FE
preprocessing - term-document matrix	feature extraction	FE
preprocessing - word vector	feature extraction	FE
preprocessing - sentence vector	feature extraction	FE
document analysis - re-ordering - selecting	data object selection	DOS
document analysis - re-ordering - sorting	data object selection	DOS
document classifier - interactive attentive module - attention weight	preprocessing	/
document classifier - interactive attentive module - prediction score	preprocessing	/
labeling interface - document embedding	data object selection	DOS
labeling interface - document visualization	labeling interface	IL

blocks may correspond to different themes. The flowchart contains 11 phases, all categorized as preliminary codes for modules because they describe actions. The 11 phrases are grouped into themes and final codes as shown in Table 4. We categorize "attention weight" and "prediction score" as preprocessing because they prepare the data visualized in the labeling interface. "Labeling interface - document embedding" is categorized as "data object selection" because it refers to an interactive projection where the annotator can select data objects to label.

B GRAPH-THEORETIC CONSTRAINTS ON DATA LABELING WORKFLOWS

B.1 Notations

- $G = (V, E)$: a directed graph modeling a data labeling tool.
- V: a set of nodes, each corresponding to a software module, or initialization, decision, and exit node.
- E: a set of edges specifying the execution order.
- $type(v)$: the type of a node v, which takes one of the following values: initialization, process, decision, exit.
- $fun(v)$: the function of a node v. When $type(v)$ takes value in { initialization, decision, exit }, $fun(v) = type(v)$. When $type(v) = process$, $fun(v)$ takes one of the following values: interactiveLabeling, dataObjectSelection, modelTraining, featureExtraction, defaultLabeling, qualityAssurance, stoppageAnalysis, labelIdeation.
- $input(v)$: the set of input states to a node v.
- $output(v)$: the output state of a node v.
- $Exec(G)$: the set of all the directed walks on the graph from the initialization node to the exit node.

B.2 Constraints on the Workflow Graph

Valid Flowchart:

- The graph contains no parallel edges (i.e., E is not a multi-set).
- The graph contains one initialization node (denote as v_s).

$$\exists! v(v \in V \wedge type(v) = initialization)$$

- The graph contains one exit node (denote as v_e).

$$\exists! v(v \in V \wedge type(v) = exit)$$

- A process node has outdegree 1.

$$\forall v(v \in V \wedge type(v) = process \rightarrow deg^+(v) = 1)$$

- A decision node has outdegree 2.

$$\forall v(v \in V \wedge type(v) = decision \rightarrow deg^+(v) = 2)$$

- An initialization node has indegree 0 and outdegree 1.

$$\forall v(v \in V \wedge type(v) = initialization \rightarrow deg^+(v) = 1 \wedge deg^-(v) = 0)$$

- An exit node has outdegree 0.

$$\forall v(v \in V \wedge type(v) = exit \rightarrow deg^+(v) = 0)$$

- All the nodes can be reached from the initialization node. The exit node can be reached from all the nodes.

$$\forall v(v \in V \rightarrow \exists exec(exec \in Exec(G) \wedge exec = (v_s, ..., v, ..., v_e)))$$

- No self loops.

$$\forall v(v \in V \rightarrow (v, v) \notin E)$$

Input Initialized:

$$\forall exec = (v_s, ..., v_e) \in Exec(G) \forall v_i \in exec \forall input \in input(v_i)$$
$$\exists j(j < i \wedge output(v_j) = input)$$

No Redundancy:

- After a module is visited, it should not be revisited until at least one of its inputs' value has been changed.

$$\forall exec = (v_1, ..., v_k) \in Exec(G)$$
$$((\exists 1 \leq i < j \leq k(v_i = v_j)) \rightarrow$$
$$(\exists i < l < j(output(v_l) \in input(v_j))))$$

- After a module is visited, its output(s) should be used by a module.

$$\forall exec = (v_1, ..., v_k) \in Exec(G)$$
$$\forall 1 \leq i \leq k(output \in output(v_i) \rightarrow$$
$$(\exists i < j \leq k(output \in input(v_j) \wedge \nexists i < l < j(output \in output(v_l)))))$$

Involve Labeling:

$$\forall exec = (v_s, ..., v_e) \in Exec(G) \exists i(fun(v_i) = interactiveLabeling)$$

C CUSTOMIZED WORKFLOW TEMPLATE

Below is the code snippet to declare a customized workflow template for webpage classification.

```
/** The declaration of a minimal webpage classification workflow. */
export default {
  nodes: [
    {
      label: 'initialization',
      type: 'Initialization',
      value: {
        dataType: 'Webpage',
        labelTasks: ['Classification'],
      },
      layout: { x: 40, y: 40 },
    },
    {
      label: 'random sampling',
      type: 'DataObjectSelection',
      value: ['Random', { params: { nBatch: { value: 2 } } }],
      layout: { x: 160, y: 40 },
    },
    {
      label: 'grid matrix',
      type: 'InteractiveLabeling',
      value: ['GridMatrix', {
        params: {
          nRows: { value: 1 },
          nColumns: { value: 2 },
        }
      }],
      layout: { x: 280, y: 40 },
    },
    {
      label: 'check all labeled',
      type: 'StoppageAnalysis',
      value: 'AllLabeled',
      layout: { x: 400, y: 40 },
    },
    {
      label: 'stop?',
      type: 'Decision',
      layout: { x: 400, y: 130 },
    },
    {
      label: 'exit',
      type: 'Exit',
      layout: { x: 410, y: 220 },
    },
  ],
  edges: [
    { source: 'initialization', target: 'random sampling' },
    { source: 'random sampling', target: 'grid matrix' },
    { source: 'grid matrix', target: 'check all labeled' },
    { source: 'check all labeled', target: 'stop?' },
    { source: 'stop?', target: 'exit', condition: true },
    { source: 'stop?', target: 'random sampling', condition: false },
  ],
}
```

D CUSTOMIZED ALGORITHM MODULE

Below is the code snippet to declare a customized default labeling module for span annotation by detecting nouns. The module is implemented in the form of an algorithm server.

```
# A customized default labeling module
# for span annotation by detecting nouns.
import json
import nltk
import tornado.httpserver
import tornado.ioloop
import tornado.options
import tornado.web
import uuid
from typing import List

# Core function
def predict(data_object: dict) -> List[dict]:
    """ Create default spans by detecting nouns. """
    sentence = data_object['content']
    pos_tag = nltk.pos_tag(sentence.split())
    start = 0
    end = 0
    spans = []
    for i, (segment, tag) in enumerate(pos_tag):
        start = end + 1 if i != 0 else end
        end = start + len(segment)
        # filter tags that are not nouns
        if tag not in ['NN', 'NNP', 'NNS']:
            continue
        spans.append({
            'text': segment,
            'start': start,
            'end': end,
            'category': 'subject',
            'uuid': str(uuid.uuid4()),
        })
    return spans

class DefaultLabelingHandler(tornado.web.RequestHandler):
    def post(self):
        self.set_header('Access-Control-Allow-Origin', '*')
        data_objects = json.loads(self.request.body)['dataObjects']
        labels = [{ 'spans': predict(d) } for d in data_objects]
        self.write({ 'labels': labels })

def main():
    tornado.options.parse_command_line()
    http_server = tornado.httpserver.HTTPServer(
        tornado.web.Application(
            handlers=[(r'/defaultLabels', DefaultLabelingHandler)],
            debug=True,
        )
    )
    http_server.listen(8007)
    print('Serve at http://localhost:8007/defaultLabels')
    tornado.ioloop.IOLoop.instance().start()

if __name__ == "__main__":
    main()
```

E USER STUDY INSTRUCTIONS

The following are the instructions we gave the participants for the 4 labeling tool development tasks.

E.1 Task 1 (Image Segmentation)

Please create a labeling tool for image segmentation according to the following specification of the labeling tool's functionality. Please do not use OneLabeler's built-in templates for this task.
Specification:

(1) In the image segmentation tool, data objects are iteratively selected for annotators to label. 1 data object is randomly selected each time.
(2) The interface displays the 1 data object in a single object display.
(3) After annotating the 1 data object, the system checks whether all the data objects are labeled.
(4) If all the data objects are labeled, the labeling STOPS.
(5) Otherwise, goto step 1.

Please tell the experimenter when you have finished.

E.2 Task 2 (Image Classification)

Based on the previous image segmentation workflow in Task 1, please modify the workflow to create another labeling tool for image classification according to the following specification of the labeling tool's functionality.
Specification:

(1) In the image classification tool, data objects are iteratively selected for annotators to label. 16 data objects are randomly selected each time.
(2) The interface displays the 16 data objects in a grid matrix with 4 rows and 4 columns.
(3) After annotating the 16 data objects, the system checks whether all the data objects are labeled.
(4) If all the data objects are labeled, the labeling STOPS.
(5) Otherwise, goto step 1.

Please tell the experimenter when you have finished.

E.3 Task 3 (Machine-aided Image Classification)

Based on the previous image classification workflow in Task 2, please modify the workflow to create another labeling tool for image classification according to the following specification of the labeling tool's functionality.
Specification:

(1) At the beginning, SVD features are extracted for the data objects.
(2) Then, data objects are iteratively selected for annotators to label. 16 data objects are randomly selected each time.
(3) The selected data objects are then assigned default labels by a decision tree classifier.
(4) The interface displays the 16 data objects in a grid matrix with 4 rows and 4 columns.
(5) After annotating the 16 data objects, the system checks whether all the data objects are labeled.
(6) If all the data objects are labeled, the labeling STOPS.
(7) Otherwise, the decision tree classifier used for default labeling is updated by retraining.
(8) Then, goto step 2.

Please tell the experimenter when you have finished.

E.4 Task 4 - Option 1 (Customize Data Type)

Based on the given image classification workflow template, please modify the workflow to create another labeling tool for webpage classification according to the following specification of the labeling tool's functionality.
Specification:

(1) In the webpage classification tool, data objects are iteratively selected for annotators to label. 2 data objects are randomly selected each time.
(2) The interface displays the 2 data objects in a grid matrix with 1 row and 2 columns.
(3) After annotating the 2 data objects, the system checks whether all the data objects are labeled.
(4) If all the data objects are labeled, the labeling STOPS.
(5) Otherwise, goto step 1.

The webpage data type is not currently supported in OneLabeler. Before you start to build the labeling tool in the visual programming interface, you will need to first create the new webpage data type with some coding. To create this new data type, please follow the instructions below.

(1) Enter the "./client" folder of OneLabeler's source code.
(2) Execute "npm run customize" in the command-line, and choose suitable options in the command-line interface to create template code files for the webpage data type.
(3) Revise template code files to create the webpage data type.

The webpage data type should satisfy this specification:

When using a grid matrix to display webpage data objects, each grid should display the webpage content. Each webpage data object should fill the grid that contains it.

Additional Tips:

- You may first read the "README.md" file in the created template code files, which gives you an overview of what the template code files are.
- When editing the code, you may refer to the HTML iframe tag API (https://developer.mozilla.org/docs/Web/HTML/Element/iframe) if you are unfamiliar with it.
- In this task, you will not need to edit any code files other than the files created by the "npm run customize" command.
- You may refer to the "Customization" section of the documentation website.
- If you do not have web development experience, you may ask the experimenter to help you with the programming part.

Please tell the experimenter when you have finished.

E.5 Task 4 - Option 2 (Reproduce Your Tool)

Please use OneLabeler to reproduce your labeling tool. The reproduction should have similar major functionalities as your labeling tool but does not need to be the same. For this task, you may ask the experimenter for help.