Usability and Adoption of Graphical Tools for Data-Driven Development

Thomas Weber
LMU Munich
Munich, Germany
thomas.weber@ifi.lmu.de

Sven Mayer
LMU Munich
Munich, Germany
sven.mayer@ifi.lmu.de

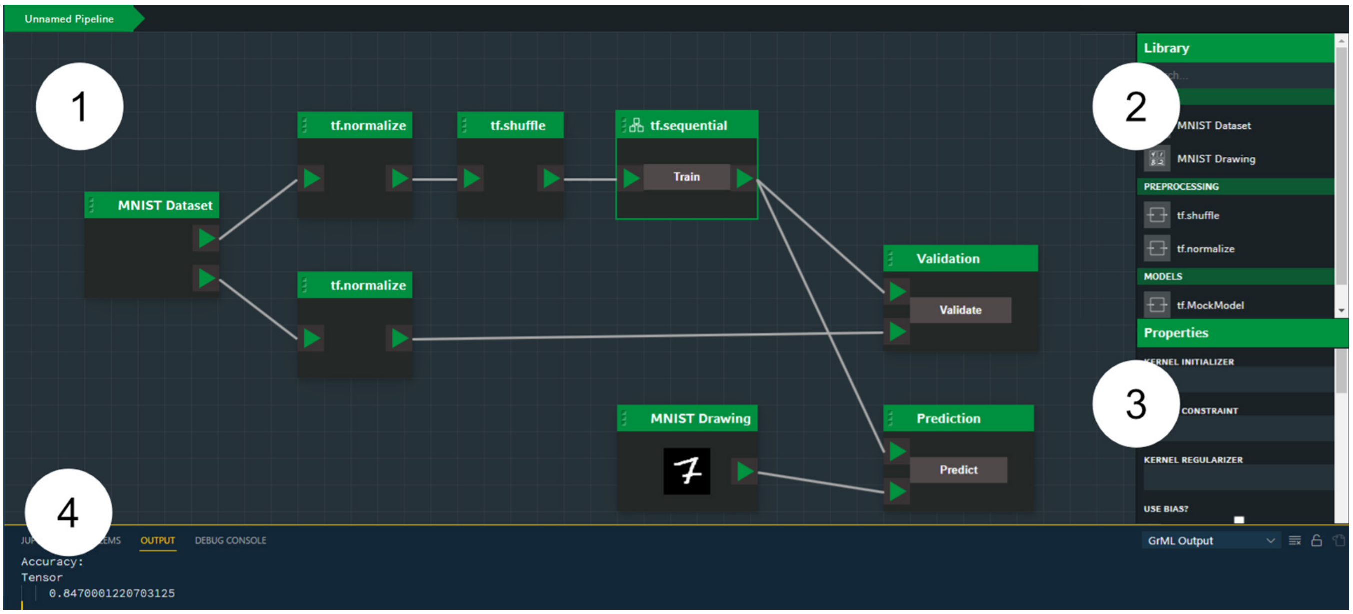

Figure 1: The graphical programming interface we created for this study to represent rich interaction with direct manipulation. It consists of a primary canvas (1) on which blocks can be placed. These blocks are dragged from the block library (2) and can have additional properties which can be edited (3). Any output is displayed in a dedicated area at the bottom of the screen (4).

ABSTRACT

Software development of modern, data-driven applications still relies on tools that use interaction paradigms that have remained mostly unchanged for decades. While rich forms of interactions exist as an alternative to textual command input, they find little adoption in professional software creation. In this work, we compare graphical programming using direct manipulation to the traditional, textual way of creating data-driven applications to determine the benefits and drawbacks of each. In a between-subjects user study (N=18), we compared developing a machine learning architecture with a graphical editor to traditional code-based development. While qualitative and quantitative measures show general benefits of graphical direct manipulation, the user's subjective perception does not always match this. Participants were aware of the possible benefits of such tools but were still biased in their perception. Our findings highlight that alternative software creation tools cannot just rely on good usability but must emphasize the demands of their specific target group, e.g., user control and flexibility, if they want long-term benefits and adoption.

CCS CONCEPTS

• **Software and its engineering → Software development techniques**; • **Human-centered computing → Empirical studies in interaction design**.

KEYWORDS

data-driven software, data-driven development, graphical programming, interaction paradigms, neural network, machine learning

ACM Reference Format:
Thomas Weber and Sven Mayer. 2024. Usability and Adoption of Graphical Tools for Data-Driven Development. In *Proceedings of Mensch und Computer 2024 (MuC '24), September 01–04, 2024, Karlsruhe, Germany*. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3670653.3670658

MuC '24, September 01–04, 2024, Karlsruhe, Germany

ACM ISBN 979-8-4007-0998-2/24/09
https://doi.org/10.1145/3670653.3670658

1 INTRODUCTION

Machine Learning, particularly using Neural Networks, has enabled many improvements in fields that are critical to everyday life in our society, like healthcare [13], communication [57], and security [6, 45]. Therefore, neural networks are an essential tool for software developers going forward. Yet, while these systems become increasingly complex and hard to understand, an issue exacerbated by their opacity [21, 24], how developers create them is not much different from how other traditional software has been created for decades: the probably most popular tool, Jupyter notebooks[1], and many more, are fundamentally still textual command input. Meanwhile, there are many alternative forms of interaction for programming, which could be applied here. Graphical programming, in particular, is an interesting candidate, given the nature of Machine Learning systems where data visualization often plays a large role, but many other systems also rely on some form of visual presentation for different goals like interpretability [55] and explainability [3, 41], optimization [11, 36], debugging [1, 4] and many more. Graphical Programming systems like RapidMiner [32], KNIME [8], or Orange [15] fully embrace this and omit textual programming for the most part. And yet, most developers still fall back to their text editors or Jupyter notebooks, which raises the question of whether these alternative systems are inferior or, more importantly, what they are missing.

In this paper, we investigated the behavior of programmers in these contexts to determine how and which aspects of graphical programming are beneficial and when developers prefer textual programming. To this end, we performed a between-groups user study (N=18), where participants had to create a Machine Learning system for image classification either traditionally with code or using a simple graphical programming tool, which we designed and developed specifically for this investigation. This tool allows for the graphical composition via direct manipulation of a data pipeline from its individual steps and of Neural Networks from their layers. We set this tool in contrast to the Jupyter Notebooks, an increasingly popular and close to de facto standard tool for data-driven development [37, 51], in which participants also implemented the same application. This allowed us to understand the differences from a qualitative and quantitative perspective, thus addressing the question of what alternative systems are missing today and how we can improve them.

Our results show that graphical programming – for our task – is, in fact, already faster and produces less workload. While it is, therefore, viewed positively, participants still expect that they work better with the traditional, text-based editor. This feedback provides insights into the potential benefits of graphical programming for data-driven development but also highlights a disparity in users' subjective perceptions. Giving developers a high degree of control and flexibility over their tools appears to be necessary to ensure the success of graphical development tools. Based on our results, we argue that a hybrid between text editors and graphical programming may be a way to consolidate the different benefits and expectations in the future.

[1]https://jupyter.org

2 RELATED WORK

Richer interactions than text input are not new for software development in general or data-driven development in particular. Direct manipulation as one of such rich interactions was, in fact, originally proposed by Shneiderman [49] as an alternative to programming languages and command-based control of computers. Since then, research and industry have developed many different forms of interaction. Yet, many of the ideas specific to professional software engineering and, more recently, data-driven development have only found limited adoption in practice. Anecdotal evidence suggests that particularly more radical changes that fundamentally change how developers interact with the software they create, fare poorly. A drastically different interaction paradigm is graphical programming, which replaces text-based commands with visual representations.

While not embraced by the broad majority, graphical programming still succeeded in some niche application domains for professional software engineering. Often, these are areas where the software is and needs to fulfill very strict requirements in timing, safety, etc. [5, 20, 42, 46]. Thus, the benefits of graphical programming, particularly for complex systems, are apparent. However, given the adoption rate, from the inside perspective, the point of view of developers, these benefits seemingly are not yet convincing.

Petre [38] investigated this disparity and compared the benefit of graphical programming for novices and experts, where she describes some of the benefits of that arise from the rich interaction like accessibility, memorability, etc. However, the article also emphasizes the challenges that arise from the different needs of novices and experts. While graphical programming is often advocated for beginners, Petre [38] instead describes how they would rather benefit from more constrained interactions. The ability to grasp and utilize graphical representation is something that also requires some expertise, and poorly designed visualizations can easily be confusing or overwhelming. Erwig and Meyer [18] also provide an early discussion of the benefits of visual languages and their integration with the more common textual counterpart in heterogeneous visual languages for general programming activities. Mohagheghi and Dehlen [33] later reviewed the state in the early 2000s in the context of Model-driven development tools and practices, which are often graphical in nature. At that point, they found adoption to be limited and only at a small scale, and evaluations were lacking. Since then, many projects in which new graphical programming tools have been created now also conduct user studies to evaluate them, especially the application for children and in education [26, 44, 50, 54] and end-user programming [2, 7] but also beyond. After all these years, graphical programming remains an active research interest, as seen in the literature review by Smith et al. [47] and recent examples like the work of Homer [25], who investigates the use of graphical programming for data-flow programming.

2.1 Graphical Programming for Data-Driven Applications

Data-driven applications have a high degree of inherent complexity. Graphical programming could be a viable route to mitigate this. Consequently, academia and industry have developed some systems that try to leverage this interaction to simplify development. Tools like RapidMiner have existed for two decades now [43] and have been applied in several domains and case studies (e.g., [22, 29, 31, 34]). RapidMiner and other tools (e.g., Orange [15] or KNIME [8]) typically visualize the data flow in data-driven applications as a sequence of blocks, which represent different steps in the processing. With this representation, they can make a relatively abstract process easier to grasp, while hiding unnecessary details. Of course, many of these details are necessary for experienced users, so there are different interaction patterns for accessing them, from popups to the common "bento box" [14] layout to hierarchical structures. They also often provide additional visualizations as support, e.g., for data exploration or to display output metrics. While not specifically targeted at novices or experts, they are still often considered more suitable for novices [9] or non-programmers [48]. Weber et al. [53] provide a further overview of tools for data-driven software development from the last decade. Based on their literature review, they also draw the conclusion that graphical programming is one area that could improve data-driven development, but has not yet managed to gain enough traction and practical use. Instead, for developers, Jupyter Notebooks so far seem the tool of choice [39]. From an interaction design point of view, these notebooks are still different from the setup common in traditional software development with either a text editor and tools for compilation and execution, or the Integrated Development Environment (IDE), which combines them. Jupyter notebooks rather focus on small code fragments and short execution and feedback cycles, very similar to a REPL (Read-Evaluate-Print-Loop) or interactive shell environment. In this sense, Jupyter notebooks can be seen as a regression towards the interaction one would have with a terminal with superior backtracing and presentation. Still, it is the development tool of choice, particularly in the domain of data-driven applications, so any novel and improved tools will have to measure up to it.

Furthermore, the Jupyter environment provides infrastructure for extending it, which not only contributed to its success across many programming languages, but has also enabled researchers to extend the fairly conservative forms of interaction. Kery et al. [28] and Zhao et al. [56] both attempted to enhance the Jupyter programming environment with visualizations that add a more graphical representation to the traditional, code-based one. The user studies found that professional users enjoy the flexibility of switching between different representations, but they also received critical feedback indicating that experts struggle with some interactions. In the ODEN tool by Zhao et al. [56], the participants were sometimes confused by the mapping between graphical and textual representation, prompting them to implement a "calm mode." Designing tools suitable for developers of data-driven applications is not a trivial task. To ensure that they can provide the benefit they promise, we must first better understand what interactions are suitable and what users prefer in a given context.

3 A GRAPHICAL PROGRAMMING EDITOR FOR NEURAL NETWORKS

Most existing graphical programming tools suffer from a common, well-known issue in that they have gathered many features. Comparing an extension of an existing tool against "plain" Jupyter notebooks would likely heavily bias the results and not give us adequate insights into the underlying interaction paradigms and how they affect developers. Thus, we considered it more adequate to build a reduced version of a graphical programming tool that offers a feature set similar to an unmodified Jupyter environment but with a different presentation. Therefore, we limit the volume of features and interactions to a minimal, comparable subset and completely control what kind of interactions were possible during our study. A custom implementation also allowed us to instrument the tool, giving us much more detailed information about the interactions. Nonetheless, the design of our tool is heavily inspired by and based on the established visual language of tools like RapidMiner [32] or KNIME [8] and their block-based interaction: instead of instruction as lines of code, the functions for processing the data are displayed as blocks. Blocks can be arranged in space via drag and drop. Blocks can then be combined into a sequence of operations. This is done via direct manipulation by dragging a connection from one block to another. Combining blocks like this lends itself to the domain of data-driven applications, since a mostly sequential pipeline is a typical structure. By basing our design on these established tools and a common style of presentation, the findings of our evaluation will be more applicable to a broader set of existing tools.

Figure 1 presenters the main interface. In addition, similar to how Jupyter Notebooks display their output below each cell, the UI had a panel for displaying any program output in a consistent location. The output was kept unchanged from what would also be displayed on a terminal, in an IDE console or in a Jupyter notebook. Besides this, the tool offers a library of blocks, which we grouped into categories like *Datasets*, and *Models*. Furthermore, each block can have parameters, just like a function call has parameters. These are displayed in a third panel as input fields depending on the parameter's type, e.g., checkboxes for boolean parameters.

Since we base the task on a tutorial that uses the Keras layers API, we implemented a subset of its functionality as blocks in our application. This was convenient as the TensorFlowJS library, which we used internally for the UI, uses mostly the same API, which allowed us to use the same instructions and functions in both experimental conditions. The blocks we implemented included all the necessary models, layers, and data processing steps for the tutorial and a few additional distractors. Each of these blocks used the same parameters as listed in the Keras/TensorFlowJS documentation. Furthermore, we added the MNIST dataset as a block, similar to how it is available in the *tensorflow_datasets* Python package used in our conventional tutorial, and two blocks to enable interactive prediction of new inputs. In the Keras API, a typical model is composed of multiple layers, i.e., a function composed of sub-functions. For this reason, our tool offers the possibility of composite blocks, i.e., blocks that consist of a sequence of subordinate blocks. Double-clicking such a composite block opens up a new editor view for arranging its components. Breadcrumb navigation at the top of the main canvas allows for navigation along this hierarchy. In our scenario, we used this type of block to create a *sequential* model from different types of layers, like *dense* or *flatten* layers. The same mechanisms can also be used to address a common issue of graphical programming tools, namely that of scaling up to larger projects. Grouping blocks hierarchically can reduce visual clutter while keeping information accessible on demand [14].

We implemented the tool as an extension for Microsoft's Visual Studio Code editor in TypeScript. The source code for the extensions is part of the supplementary material and is ready to be part of a public code repository.

4 USER STUDY

We conducted a between-subject study to understand how different types of tools and interaction paradigms affect programmers. Participants had to create a simple machine learning system using a neural network for classifying the MNIST handwritten digits [30] based on a set of instructions and provide feedback on their experience and preferences using a questionnaire. Additionally, we conducted semi-structured interviews with the participants to get additional in-depth feedback. In the study, one group of participants created their neural network using the established interaction of textual command input, using *Jupyter notebooks*, the de-facto standard tool for data science programming. The second group created the same system using an equivalent set of instructions but with the graphical programming tool described above. As a first step, our study focuses on general impressions, developer experience and task execution. However, the setup can be used in future work, for example for investigating how visual presentation helps with understanding complex, large-scale ML pipelines.

4.1 Task

The instructions were designed to follow the general outline of a typical machine-learning tutorial. For this reason, we took an existing example from the TensorFlow documentation[2] and adapted it, where necessary, for our study. The most notable change is that we split the training and validation, which in the original are a single function call, into two separate steps. The tutorial version for the graphical programming group received some additional changes: we replaced all code examples with screenshots of our tool in the respective state, and we had to rephrase some sentences such that they reflect the difference in interaction, which, for example, meant replacing occurrences of *to type* or *to write* with more suitable verbs. We opted to keep the visual presentation of the tutorial as close to the original as possible, but removed any clutter and unnecessary links from the site to ensure that participants were focused on the core instructions. Any external links were also removed to keep participants on the tutorial page.

4.2 Procedure

We gave them a small introduction and asked for their consent to record the data. Next, we introduce the tool. For *Jupyter Notebooks*, this includes information on how to write and execute code and how to extend the notebook with new cells. For *Graphical Programming*, we introduce the panels described above and the direct manipulation via drag and drop. After this, participants received access to the instructions, and we asked them to work through these at their own pace. The whole task was scheduled to take about 15 minutes and participants were made aware of this estimate. However, they were instructed to work through the task at the pace they personally deemed necessary for successfully completing it. Thus, we did not rush them to finish the programming task or cut them off after the allocated time. At the end of the programming task, we asked them to complete a survey about the task and tool. Next, we showed the participants a screen recording of their task being performed in the alternative condition. Having watched this, we asked participants to compare what they had seen to what they had used, using the same survey as before. Five participants also participated in a semi-structured interview after they entered the collected additional qualitative feedback.

4.3 Survey and Interview Guidelines

First, we asked participants about their demographics, and prior knowledge. We used Edison and Geissler's scale [17] to record their attitude towards technology. Then, we queried participants about their experience with the technologies specific to our study and different topics about machine learning. The survey after completing the task and after seeing the video, we used 100-point slider ranging from strongly disagree to strongly agree. [12] for a series of general statements about the task and the tool. We decided to use visual analog scales (VAS), as they have been shown to lead to more precise responses and higher data quality [19]. Finally, as VAS collects continuous data, they allow for more statistical tests [40]. In line with recommendations for scale development, we phrased the statements strongly as mildly phrased statements have shown to result in too much agreement [16].

Second, we asked participants to rate the tool based on several categories based on Nielsen's usability heuristics [35] where applicable and some additional categories (see Figure 3). Finally, we used the Systems Usability Scale (SUS) [10] and the raw NASA Task Load Index [23]. Participants could also provide additional qualitative feedback using free text fields. We discussed participants' experiences in a semi-structured interview. The interview questions first focused on the general experience during the study and what aspects of the task were particularly challenging. We then asked about the tool, its benefits and drawbacks, and under what circumstance or for which target group it might be most suitable. Finally, we asked for feedback on how to make the machine learning programming easier to use in the everyday work context of the participants.

4.4 Apparatus

Participants completed both the survey and the programming task on a computer provided by us using a mouse and keyboard. This way, we ensured that the training phase of the machine learning system was using the same hardware, making it more consistent. The page with the instructions was displayed on a second screen to minimize the need for switching windows. Aside from the survey, additional information like the timing of interactions and the task result was automatically recorded in the background.

[2] https://www.tensorflow.org/datasets/keras_example

4.5 Participants

We recruited participants with prior experience with programming, particularly in data-driven applications. Knowledge of specific Machine Learning libraries or methods was not a prerequisite. We recruited them via a combination of personal and professional contacts and multiple mailing lists over the course of four weeks. We verified that all participants were from fields where data-driven programming is relevant. In total, 20 participants completed our study (4 female, 16 male). With a mean age of 29 ($SD = 5.2$) and six years of programming on average ($SD = 3.8$). Half of the participants had a background in computer science, while the other half came from STEM areas where data processing and machine learning methods are common. The participants were employed full-time, typically with a completed Master's degree (14), or were currently still pursuing a degree in these domains but worked part-time on projects involving data-driven applications (6). Based on their feedback, the participants generally had a positive attitude towards technology (average score: 76 of 100, $SD = 11$). They were at least somewhat familiar with all the specific technologies used in our study, like machine learning or Jupyter notebooks. While they knew of the MNIST data set, none reported completing the specific tutorial upon which our task was based. Four participants answered that they use graphical programming tools in their work, although not for creating data-driven applications. We scheduled for a session of 45 minutes per participant. We compensated participants with an equivalent of 10 US$.

5 RESULTS

Of the 20 people who participated in our study, we excluded two from the data set, one from each group, after the first data screening, as their responses strongly suggested that they misunderstood the instructions. The remaining data yielded the following insights. As Figure 2 shows, the task was perceived as easy to complete in both conditions. Given the overall very similar feedback about the task, we conclude that the task design was suitable, as it was not notably easier or better supported by any of the two tools. The scales asking about the tools show some differences: while the graphical programming tool received slightly better feedback on enjoyment, how it supports understanding, and its overall design, these differences are not significant.

5.1 Quantitative Results

In terms of usability, both tools scored high on the SUS [10], 81 of 100 ($SD = 11.7$) for the graphical tool and 74 ($SD = 10.4$) for Jupyter Notebooks, see Figure 4. This difference is not significant. The workload as measured by the raw NASA TLX [23], meanwhile, is significantly lower for the graphical tool (Shapiro-Wilk-test: $W = 0.934$, $p = 0.287$, t-test: $t(17) = -4.027$, $p < 0.001$). Moreover, each of the six subscales of the NASA TLX is lower for the graphical tool individually, highlighting the reduction in workload even further (see Figure 4). Furthermore, the graphical tool was also rated slightly superior in appearance, ease of understanding, and learning in the questions (see Figure 3). The participants also rated the graphical programming UI to be significantly superior in terms of *error prevention* ($p = 0.023$) and how *easy to learn* they considered it ($p = 0.025$, see Table 1 for a complete list and test details).

While these responses showed only moderate differences, we observed a stronger change when participants were asked to compare the interface that they used to the one they saw in the screen recording. After seeing the alternative tool, participants rated the graphical tool significantly more favorable concerning appearance, how it offered assistance, and how easy it might be to learn. The Jupyter notebooks, on the other hand, were rated positively for their perceived increased flexibility compared to the graphical tool. See Figure 4 and Table 1 for additional details. Figure 4 also shows the difference in opinions for participants for these categories, which further highlights how the comparison skews the participants' perceptions.

We then compared the recorded task completion time for all participants. As Figure 4 shows, completing the task using textual input on average takes longer by a significant amount (Mann-Whitney-U test, $p = 0.040$). While this is in part due to two outliers, the questionnaire responses from these two do not suggest that they considered the task more challenging or struggled more than the other participants.

5.2 Qualitative Results

After an initial screening of the responses to the survey and the interviews, we performed open coding of the free-text answers. Those were then further clustered through three iterations, first into positive or negative comments, then into thematically similar groups and finally into four higher-level categories: responses specific to the task, the tool's appearance, understanding of the system, and feedback on the interaction paradigms.

5.2.1 Task. All nine participants who commented on the task described it as "simple and easy (as it was) split into meaningful parts" (P3). This is the case for both groups.

5.2.2 Appearance. The feedback on the appearance of the graphical tool was positive, with multiple participants enjoying the visual presentation of the data flow via connections. The focus on the blocks led one participant to note that the output can be "easy to overlook" (P15), though. The presentation of output close to the code as done in Jupyter, on the other hand, was considered to be convenient but constituted the only comment on the appearance of Jupyter notebooks.

5.2.3 Understanding of the System. Issues with understanding what was going on were related to general machine learning concepts. One participant expressed the opinion that "*(Jupyter) notebooks require prior knowledge*" (P6), whereas the graphical UI was described as especially suitable for beginners. To enable growth and learning, one participant suggested that "*a person learning could switch between*" (P6) graphical and text-based UI to use the presentation that suits them best and at any point in time. This flexibility was also a point of multiple participants asking for more information to be displayed on demand in the graphical UI, particularly for the parameters, as it was not always clear "*what parameters (...) were doing*" (P9). In addition to documentation, participants also favored more continuous feedback. Similar to how in Jupyter notebooks, each cell has its output adjacent, our participants wanted to see information about the success and output of blocks close to them in the graphical UI. In the Jupyter notebooks, the only indicator for a complete

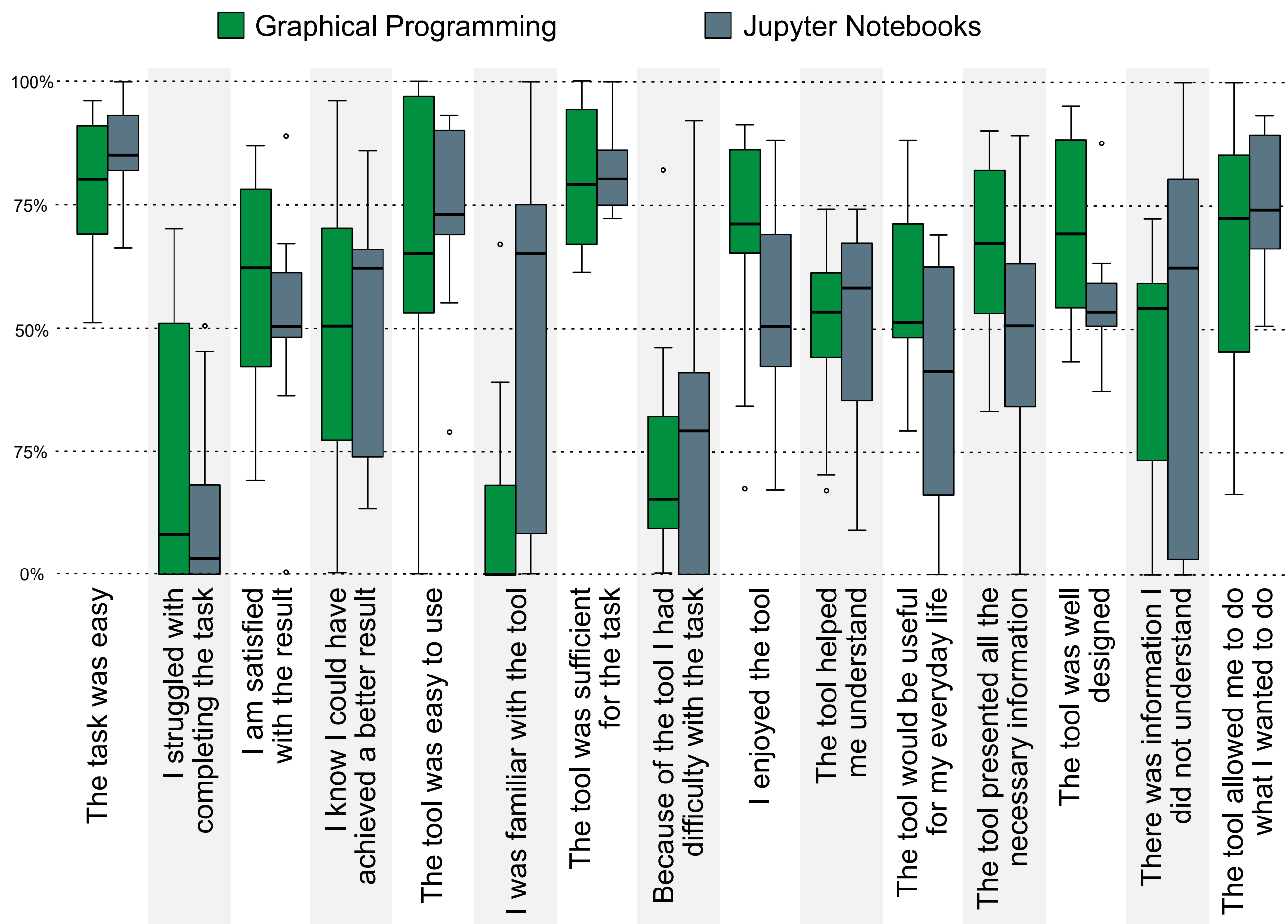

Figure 2: While there are no significant differences between the responses to various statements about the task and the tools, they show that the task was perceived very similarly.

computation without output is a change in cell numbering – an indicator that, according to participants, is easily overlooked. Still, one participant praised the blocks, into which code is split in notebooks, as they help to structure longer code. This can lead to other issues, though, as "*boxes can be compiled in any order, meaning that code can crash if compiled in the wrong order,*" so there was a wish for information about data flow and notifications about changes with dependencies. If something does not work, Jupyter shows the corresponding error messages. Four participants criticized these messages to be "*cryptic*" (P1) and not helpful. However, this is not per se an issue of Jupyter, but of the Python programming language. The error messages of the graphical UI, which use natural language, was praised by one participant to be good for "*understanding the way things worked*" (P15).

5.2.4 Feedback on the Interaction Paradigms. The key difference between the two systems lies in the form of interaction, though. Thus, participants provided a number of comments on the interaction after both using the system and seeing the alternative system. The interaction with drag and drop and direct manipulation to create the connections and the blocks were commented upon positively by seven participants. Grabbing the blocks from a library of predefined blocks was also mentioned as a desirable feature, as it reduces the need to recall API functionality from memory. While this does apply to the properties of selected blocks, two participants voiced confusion, as they were overwhelmed by the number of available parameters, even though they only needed to change one or two for the task. Four participants mentioned the issue of typing errors, which can be hard to notice and typically lead to immediate errors in the code. Writing code, however, forces it to adhere to a strict sequence of instructions, which matches how it is executed. On the other hand, the graphical presentation is not constrained like that, so more emphasis is put on the connections to indicate the execution sequence. However, two participants highlighted liking the presentation with lines as connections. This also enables the tool to display data flow across steps that are not directly adjacent. An example of this in our task is the validation data set. It is used fairly late in the process, which in code means that there are multiple code cells between its declaration and usage, whereas the blocks have a direct connection.

Table 1: Results of the statistical analysis of how participants judged the systems concerning usability heuristics (Single Judgment) and comparing the system they used to the alternative as seen in a screen recording (Comparative Judgment). Shapiro-Wilk test was used for normality and t-test for the subsequent analysis of normally distributed data and Mann-Whitney-U test for not-normally distributed data.

	Rating after usage					Comparative Rating				
	Normality test		MWU/t-test			Normality test		t-test		
	W	p	W/t	df	p	W	p	t	df	p
Visibility of System Status	0.973	0.852	0.8	15.991	0.435	0.945	0.442	−1.903	9.724	0.087
Familiarity	0.857	0.011	45.5		0.691	0.956	0.627	0.157	12.763	0.878
Exploration	0.910	0.085	1.283	13.907	0.221	0.943	0.428	−0.858	11.273	0.409
Consistency	0.952	0.460	−0.018	12.85	0.986	0.950	0.634	0.157	9.644	0.879
Error Prevention	0.953	0.480	2.480	15.638	0.025	0.915	0.141	−1.636	12.345	0.127
Easy to Understand	0.907	0.077	1.331	14.827	0.203	0.955	0.536	−2.089	14.414	0.019
Flexibility	0.973	0.848	−0.474	13.387	0.643	0.920	0.168	3.212	13.915	0.006
Efficiency of Use	0.900	0.050	1.133	15.852	0.274	0.964	0.740	−0.556	10.509	0.590
Aesthetic Design	0.823	0.003	57.5		0.145	0.935	0.287	−3.305	11.829	0.006
Assistance	0.946	0.369	1.034	15.984	0.317	0.946	0.462	−3.318	11.604	0.006
Easy to Learn	0.951	0.439	2.517	15.279	0.023	0.927	0.191	−2.384	14.7	0.031
Useful	0.948	0.397	0.564	13.443	0.582	0.956	0.557	−0.665	12.401	0.518

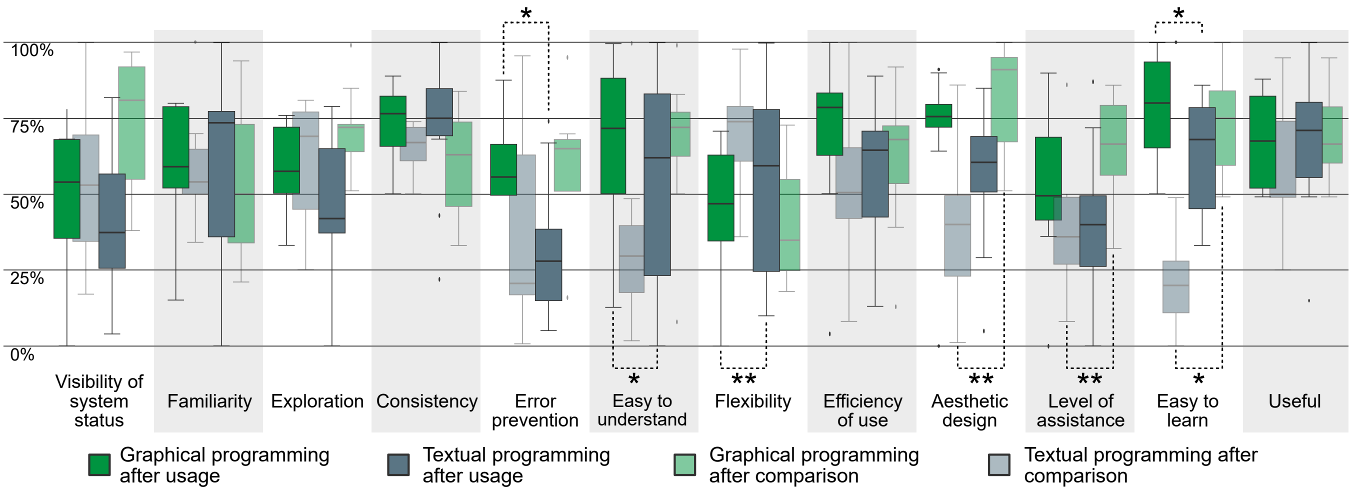

Figure 3: Each participant rated the tool they used and the alternative tool in comparison using the same properties and heuristics about usability (Statistical comparison with Mann-Whitney-U/t-test – *: $p < 0.05$, **: $p < 0.01$)

5.2.5 Additional Feedback. Concerning the code execution, the feedback was that successful computations need not necessarily be displayed spatially close to the source code, like the Jupyter cells below, but a single output terminal was adequate. However, for errors, participants highlighted how a clear visual indicator at the erroneous location is desirable to quickly identify the source of the error, while the output from the error message may not be immediately helpful.

Regarding the adoption of graphical tools, the consensus was that they were willing to try such a tool but did not expect it to replace their familiar work environment. They named this because they expected no dramatic productivity benefit. If participants already used graphical tools, they typically did so for external motivations, e.g., because regulations or a legacy project required it required their use. They were reluctant to use graphical programming tools for their everyday work, both for creating and using data-driven systems and in other scenarios. They justified this with the fact that they had a set of tools with which they were familiar, and adopting a new tool can be a time investment with uncertain revenue. The participants suggested two scenarios, though, where a graphical programming environment may be adopted: first for education or novices, as is a common idea described above. Second, they saw the high-level presentation as beneficial for setting up a data processing pipeline. In their expectation, they would, however, switch

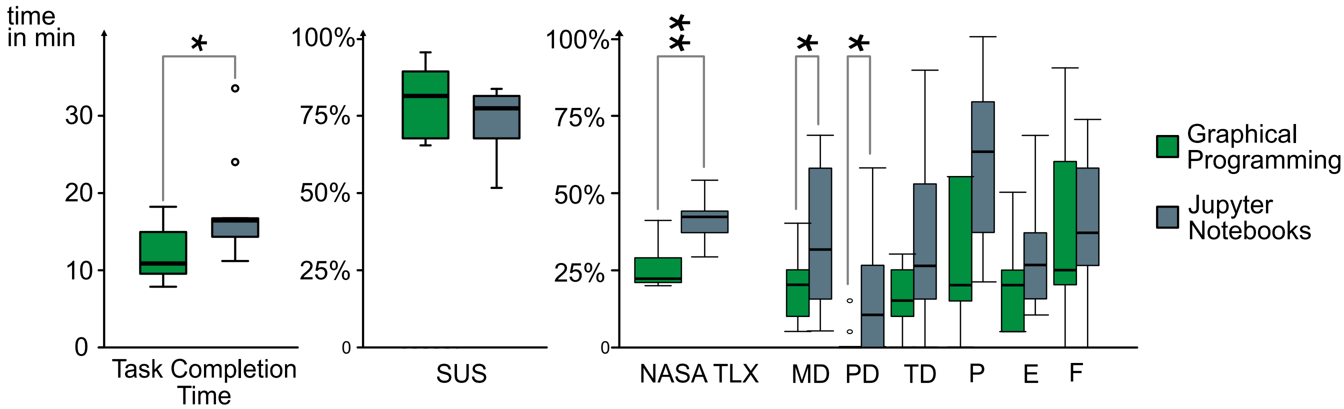

Figure 4: Task completion time and NASA TLX results show significant differences, while the SUS shows positive results for both UIs. (Statistical comparison with Mann-Whitney-U/t-test – *: $p < 0.05$, **: $p < 0.01$)

back to textual code for fine-tuning. Thus, they suggested some form of on-demand translation or switching between the different presentations to use different paradigms in different development project phases.

6 DISCUSSION

Overall, the feedback from participants is mixed. Jupyter notebooks and our prototype of a graphical programming tool appear to be adequate for the given task. The significant differences in task completion time suggest that direct manipulation is superior to creating data-driven applications. However, the additional feedback is not as clear.

6.1 Bias Toward State-of-the-Art

When judging the tools in isolation, the category in which we observed a significant difference was the category of error prevention. While fewer questions corroborate this, it is an almost necessary consequence of the design: text input has high flexibility, while the graphical UI constrains the options. It is much easier to type an incorrect command accidentally. Likewise, when developers have to remember all commands, they are more prone to make mistakes or not remembering the correct parameters, values, etc. However, the perception differed more strongly when we showed the alternative tool to the participants. While in the graphical tool was rated superior in appearance and how easy it is to learn, the text-based coding was rated to offer better flexibility. Even in other categories, like consistency or how it supports exploration, the participants' opinions shifted considerably, suggesting a strong anchoring bias, i.e., the perception of a tool is skewed by what the participants compare it to. This could mean that any new tool will have a harder time measuring up to what developers already use and with which they are very familiar. Therefore, it is unsurprising that adopting alternative tools like graphical programming tools in software development and data-driven development is sluggish and happens only in small incremental changes.

6.2 Novice vs. Expert

The opinions voiced by our participants further emphasize this. Still, also that graphical programming should rather be targeted at novices or for learning. At the same time, the traditional text editor is the tool of choice for experts even though some literature disagrees with this assertion [38]. Even if participants could identify the potential benefits of different forms of interaction and presentation, multiple suggested that any such benefit would not apply to them. In fact, they, for example, expect to be faster with written code, which the data in our study does not support. This perception may hold true for the pure input method, but this fails to consider contextual factors such as potentially difficult debugging, documentation, etc.

6.3 Additional Support Mechanisms

Participants' feedback naturally focused on the study tools, which were limited in features. Additional support mechanisms like good auto-completion can make textual input faster, but the same potential for improvement also applies to other interaction paradigms beyond typing code. While the support mechanisms for text are naturally more apparent, some of them may be adapted to other forms like graphical programming, or they could enable completely new support mechanisms. In fact, many of the available graphical programming tools have an abundance of features. While this may help in some situations, this volume of features can also be overwhelming to users and clutter the interface. At the same time, additional visual features, like color, shape, etc., can be a way to make graphical programming viable for larger projects where many components need to be visually organized in additional to the grouping we already implemented. Since the results our study supports indicate that a simple graphical interface provides benefits, future work may investigate how this translates to these mature tools with greater visual complexity and whether the additional features are in fact beneficial or detrimental. Either way, striking a balance between offering enough support and preventing the UI from becoming cluttered is an important aspect in the interface design of these development tools.

6.4 Adoption

The feedback demonstrates that the lack of adoption is not a consequence of inherently inferior usability. The recorded metrics, in fact, show that a graphical programming environment with an equivalent level of features can be viewed on par and even superior in some aspects. Yet, even though our participants saw the potential benefits for productivity and effectiveness, they remained reluctant to use different types of tools, partially biased by their prior experience and habituation. If we hope to make the benefits of alternative forms of interaction more accessible, maintaining an adequate level of usability appears to be only a prerequisite. Since software developers seem to value some aspects of their user experience more, particularly flexibility and being in control of their software, this may be an aspect that needs to be addressed specifically.

A potentially simple way to facilitate this is to give developers more choice: if a tool supports different forms of interaction to create a data-driven application, e.g., via graphical and text-based view, developers can select the most suitable in a given situation and switch between them. While the simple availability may convince some, it is still very possible that developers simply decide to fall back on what they know and do not utilize their options. The increased functionality adds complexity, so, as previously described by Petre [38], some expertise will be necessary to leverage this effectively.

6.5 Hybrid Interfaces

It remains unclear how such hybrid systems best consolidate the options to prevent this. More interactions and functionality easily run the risk of adding complexity and confusion. There are many alternatives for combining these forms of interaction, be it side-by-side, as a mutually exclusive selection, or any form in between. Which of these or whether any of these have desirable effects should be explored in the future. Jupyter Notebooks could be a viable platform for this. While the core notebooks currently offer only hybrid output, extensions with more interactive widgets already exist (e.g., Kery et al. [27]), and adding to this to allow for fully hybrid interaction certainly seems viable.

6.6 Future Adoption

Given how developers are used to their text editors or IDEs, confining them might be hard. Thus, combining advertising of other forms of interaction with the opinion that graphical programming may be suitable for education, early Computer Science education may be a point where such a change in perception can be fostered. Suppose a new generation of developers is introduced to not just the traditional text editor or IDEs, but also consistently uses other tools in productive use. In that case, they may become normalized and be just one tool in a developer's toolbox. Usage for real-world tasks with long-term observations could then yield insights into how a wider tool choice affects work practices over time and how this can affect tool design. While this is possible in the few domains that already use graphical tools, observing this in professional, large-scale, data-driven development is, unfortunately, not yet possible due to lack of adoption. This area, though, could provide some interesting insights, particularly where deep neural networks are used. They are an especially interesting case for other forms of programming and presentation, since in them, a lot of information and effort is highly condensed in an often opaque fashion, making them hard to grasp. Since people thus already often communicate about them with visualizations, a graphical presentation could make them more transparent and explicit.

6.7 Limitations

Due to the nature of the controlled user study, the complexity and scope of the problem were fairly limited. The MNIST data set is a common introductory example and thus presents an "optimal" workflow that lacks situations where the data is inadequate, the model architecture is subject to trial and error, parameter choices lead to unexpectedly poor performance, and many more issues that may arise during "real-world" development. The plug-and-play nature of graphical components may help when debugging requires trying different configurations. Additionally, the enforced sequencing of the components can also prevent issues due to incorrect execution orders. However, how much of an impact this has in real, large-scale data projects must be subject to future evaluations in the field.

Additionally, our participants had a strong desire for flexibility and control, and a prototypical implementation may be able to offer this or at least give the perception that it does not. Thus, an evaluation of extended usage may eliminate novelty bias and yield more nuanced insights, for example, whether there is a general lack of flexibility or whether there are specific scenarios where graphical programming excels, e.g., setting up an overall structure, avoiding boilerplate. In contrast, textual programming is superior in other areas, e.g., fine-tuning.

The prototypical nature of our development tool also offers an additional limitation. Mature development environments offer a myriad of features, often added by user demand. Thus, these features could further add to user satisfaction. We deliberately limited our tool in scope to have the same level of features as a plain Jupyter notebook to achieve a fair comparison. However, Jupyter notebooks can also be extended with additional features. While it may be easy to assume that the positive feedback from our participants would only be enhanced by more and personalized features, integrating features is not without challenges. Thus it remains to be seen whether our findings on the benefits of graphical programming can be a maintained in a more feature-rich environment. Our ongoing work on modifying our interface as a drop-in replacement into existing development tool infrastructure with interoperability with, e.g., Jupyter kernels is one step towards testing this [52].

Additionally, while block-based programming is one of the most popular variants of graphical programming, they are not the only way to present software visually. Other implementations of visual programming might lead to different conclusions and effects.

Finally, the participant group in our study had a very specific background and level of expertise. As it has been postulated that graphical programming in general may be beneficial, e.g., for novices, it will be interesting to see whether this is actually the case. Given that Machine Learning is becoming a tool for more than just professional developers, e.g., for artists, the area of data-driven software may give graphical programming for novices renewed relevance.

7 CONCLUSION

This paper contributes insights into the design of software development tools for data-driven development. Our experiments compare textual code-based programming to graphical programming as an alternative rich form of interaction. They show the latter's benefits, for example, faster task completion and better error prevention. In addition, the participants in our study can identify further benefits, especially related to learning and the ability to grasp the general structure of a complex data-driven application quickly.

REFERENCES

[1] Shazia Afzal, Arunima Chaudhary, Nitin Gupta, Hima Patel, Carolina Spina, and Dakuo Wang. 2021. Data-Debugging Through Interactive Visual Explanations. In *Trends and Applications in Knowledge Discovery and Data Mining*, Manish Gupta and Ganesh Ramakrishnan (Eds.). Springer International Publishing, Cham, 133–142.

[2] Gopika Ajaykumar and Chien-Ming Huang. 2020. User Needs and Design Opportunities in End-User Robot Programming. In *Companion of the 2020 ACM/IEEE International Conference on Human-Robot Interaction* (Cambridge, United Kingdom) *(HRI '20)*. Association for Computing Machinery, New York, NY, USA, 93–95. https://doi.org/10.1145/3371382.3378300

[3] Gulsum Alicioglu and Bo Sun. 2022. A survey of visual analytics for Explainable Artificial Intelligence methods. *Comput. Graph.* 102 (2022), 502–520. https://doi.org/10.1016/j.cag.2021.09.002

[4] Bilal Alsallakh, Narine Kokhlikyan, Vivek Miglani, Shubham Muttepawar, Edward Wang, Sara Zhang, David Adkins, and Orion Reblitz-Richardson. 2021. Debugging the Internals of Convolutional Networks. In *eXplainable AI approaches for debugging and diagnosis.* https://openreview.net/forum?id=0YRkrxe2blh

[5] Muhammad Waseem Anwar, Muhammad Rashid, Farooque Azam, Muhammad Kashif, and Wasi Haider Butt. 2019. A model-driven framework for design and verification of embedded systems through SystemVerilog. *Des. Autom. Embed. Syst.* 23, 3-4 (2019), 179–223. https://doi.org/10.1007/s10617-019-09229-y

[6] Miloud Bagaa, Tarik Taleb, Jorge Bernal Bernabé, and Antonio F. Skarmeta. 2020. A Machine Learning Security Framework for Iot Systems. *IEEE Access* 8 (2020), 114066–114077. https://doi.org/10.1109/ACCESS.2020.2996214

[7] Barbara Rita Barricelli, Fabio Cassano, Daniela Fogli, and Antonio Piccinno. 2019. End-user development, end-user programming and end-user software engineering: A systematic mapping study. *J. Syst. Softw.* 149 (2019), 101–137. https://doi.org/10.1016/j.jss.2018.11.041

[8] Michael R. Berthold, Nicolas Cebron, Fabian Dill, Thomas R. Gabriel, Tobias Kötter, Thorsten Meinl, Peter Ohl, Kilian Thiel, and Bernd Wiswedel. 2009. KNIME - the Konstanz Information Miner: Version 2.0 and Beyond. *SIGKDD Explor. Newsl.* 11, 1 (Nov. 2009), 26–31. https://doi.org/10.1145/1656274.1656280

[9] Mohamed Bjaoui, Houneida Sakly, Mourad Said, Naoufel Kraiem, and Mohamed Salim Bouhlel. 2020. Depth Insight for Data Scientist with RapidMiner an Innovative Tool for AI and Big Data towards Medical Applications. In *Proceedings of the 2nd International Conference on Digital Tools & Uses Congress* (Virtual Event, Tunisia) *(DTUC '20)*. Association for Computing Machinery, New York, NY, USA, Article 21, 6 pages. https://doi.org/10.1145/3423603.3424059

[10] John Brooke et al. 1996. SUS-A quick and dirty usability scale. *Usability evaluation in industry* 189, 194 (1996), 4–7.

[11] José Eduardo Cejudo Grano de Oro, Petra Julia Koch, Joachim Krois, Anselmo Garcia Cantu Ros, Jay Patel, Hendrik Meyer-Lueckel, and Falk Schwendicke. 2022. Hyperparameter Tuning and Automatic Image Augmentation for Deep Learning-Based Angle Classification on Intraoral Photographs - A Retrospective Study. *Diagnostics* 12, 7 (2022), 12. https://doi.org/10.3390/diagnostics12071526

[12] Carl J Chimi and David L Russell. 2009. The Likert scale: A proposal for improvement using quasi-continuous variables. In *Information Systems Education Conference, Washington, DC*. 1–10.

[13] I. Glenn Cohen and Harry Graver. 2019. A Doctor's Touch: What Big Data in Health Care Can Teach Us About Predictive Policing. *SSRN Electronic Journal* (2019). https://doi.org/10.2139/ssrn.3432095

[14] Robert DeLine and Kael Rowan. 2010. Code Canvas: Zooming towards Better Development Environments. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2* (Cape Town, South Africa) *(ICSE '10)*. Association for Computing Machinery, New York, NY, USA, 207–210. https://doi.org/10.1145/1810295.1810331

[15] Janez Demsar, Blaz Zupan, Gregor Leban, and Tomaz Curk. 2004. Orange: From Experimental Machine Learning to Interactive Data Mining. In *Knowledge Discovery in Databases: PKDD 2004, 8th European Conference on Principles and Practice of Knowledge Discovery in Databases, Pisa, Italy, September 20-24, 2004, Proceedings (Lecture Notes in Computer Science, Vol. 3202)*, Jean-François Boulicaut, Floriana Esposito, Fosca Giannotti, and Dino Pedreschi (Eds.). Springer, 537–539. https://doi.org/10.1007/978-3-540-30116-5_58

[16] Robert F DeVellis and Carolyn T Thorpe. 2021. *Scale development: Theory and applications.* SAGE.

[17] Steve W. Edison and Gary L. Geissler. 2003. Measuring attitudes towards general technology: Antecedents, hypotheses and scale development. *Journal of Targeting, Measurement and Analysis for Marketing* 12, 2 (1 11 2003), 137 – 156.

[18] Martin Erwig and Bernd Meyer. 1995. Heterogeneous Visual Languages-Integrating Visual and Textual Programming. In *Proceedings 11th International IEEE Symposium on Visual Languages, Darmstadt, Germany, September 5-9, 1995.* IEEE Computer Society, 318–325. https://doi.org/10.1109/VL.1995.520825

[19] Frederik Funke and Ulf-Dietrich Reips. 2012. Why Semantic Differentials in Web-Based Research Should Be Made from Visual Analogue Scales and Not from 5-Point Scales. *Field Methods* 24, 3 (2012). https://doi.org/10.1177/1525822X12444061

[20] Abdoulaye Gamatié, Sébastien Le Beux, Éric Piel, Rabie Ben Atitallah, Anne Etien, Philippe Marquet, and Jean-Luc Dekeyser. 2011. A Model-Driven Design Framework for Massively Parallel Embedded Systems. *ACM Trans. Embed. Comput. Syst.* 10, 4 (2011), 39:1–39:36. https://doi.org/10.1145/2043662.2043663

[21] Randy Goebel, Ajay Chander, Katharina Holzinger, Freddy Lécué, Zeynep Akata, Simone Stumpf, Peter Kieseberg, and Andreas Holzinger. 2018. Explainable AI: The New 42?. In *Machine Learning and Knowledge Extraction - Second IFIP TC 5, TC 8/WG 8.4, 8.9, TC 12/WG 12.9 International Cross-Domain Conference, CD-MAKE 2018, Hamburg, Germany, August 27-30, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 11015)*, Andreas Holzinger, Peter Kieseberg, A Min Tjoa, and Edgar R. Weippl (Eds.). Springer, 295–303. https://doi.org/10.1007/978-3-319-99740-7_21

[22] Jianchao Han, Juan C. Rodriguez, and Mohsen Beheshti. 2008. Diabetes Data Analysis and Prediction Model Discovery Using RapidMiner. In *International Conference on Bio-Science and Bio-Technology, BSBT 2008, part of the Second International Conference on Future Generation Communication and Networking, FGCN 2008, Volume 3, Hainan Island, China, December 13-15, 2008.* IEEE Computer Society, 96–99. https://doi.org/10.1109/FGCN.2008.226

[23] Sandra G. Hart. 2006. Nasa-Task Load Index (NASA-TLX); 20 Years Later. *Proceedings of the Human Factors and Ergonomics Society Annual Meeting* 50, 9 (2006), 904–908. https://doi.org/10.1177/154193120605000909

[24] Robert R. Hoffman, Shane T. Mueller, Gary Klein, and Jordan Litman. 2018. Metrics for Explainable AI: Challenges and Prospects. *CoRR* abs/1812.04608 (2018). arXiv:1812.04608 http://arxiv.org/abs/1812.04608

[25] Michael Homer. 2023. Multiple-Representation Visual Compositional Dataflow Programming. (2023).

[26] Michael S. Horn, Erin Treacy Solovey, R. Jordan Crouser, and Robert J. K. Jacob. 2009. Comparing the use of tangible and graphical programming languages for informal science education. In *Proceedings of the 27th International Conference on Human Factors in Computing Systems, CHI 2009, Boston, MA, USA, April 4-9, 2009*, Dan R. Olsen Jr., Richard B. Arthur, Ken Hinckley, Meredith Ringel Morris, Scott E. Hudson, and Saul Greenberg (Eds.). ACM, 975–984. https://doi.org/10.1145/1518701.1518851

[27] Mary Beth Kery, Donghao Ren, Fred Hohman, Dominik Moritz, Kanit Wongsuphasawat, and Kayur Patel. 2020. Mage: Fluid Moves Between Code and Graphical Work in Computational Notebooks. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology* (Virtual Event, USA) *(UIST '20)*. Association for Computing Machinery, New York, NY, USA, 140–151. https://doi.org/10.1145/3379337.3415842

[28] Mary Beth Kery, Donghao Ren, Kanit Wongsuphasawat, Fred Hohman, and Kayur Patel. 2020. The Future of Notebook Programming Is Fluid. In *Extended Abstracts of the 2020 CHI Conference on Human Factors in Computing Systems, CHI 2020, Honolulu, HI, USA, April 25-30, 2020*, Regina Bernhaupt, Florian 'Floyd' Mueller, David Verweij, Josh Andres, Joanna McGrenere, Andy Cockburn, Ignacio Avellino, Alix Goguey, Pernille Bjøn, Shengdong Zhao, Briane Paul Samson, and Rafal Kocielnik (Eds.). ACM, 1–8. https://doi.org/10.1145/3334480.3383085

[29] Jesse Lane and Hak J. Kim. 2015. Big data: web-crawling and analysing financial news using RapidMiner. *Int. J. Bus. Inf. Syst.* 19, 1 (2015), 41–57. https://doi.org/10.1504/IJBIS.2015.069064

[30] Yann LeCun. 1998. The MNIST database of handwritten digits. *http://yann.lecun.com/exdb/mnist/* (1998).

[31] Marcin Michalak, Katarzyna Dusza, Dominik Korda, Krzysztof Kozlowski, Bartlomiej Szwej, Michal Kozielski, Marek Sikora, and Lukasz Wróbel. 2016. Application of RapidMiner and R Environments to Dangerous Seismic Events Prediction. In *Proceedings of the 2016 Federated Conference on Computer Science and Information Systems, FedCSIS 2016, Gdańsk, Poland, September 11-14, 2016 (Annals of Computer Science and Information Systems, Vol. 8)*, Maria Ganzha, Leszek A. Maciaszek, and Marcin Paprzycki (Eds.). IEEE, 249–252. https://doi.org/10.15439/2016F188

[32] Ingo Mierswa, Michael Wurst, Ralf Klinkenberg, Martin Scholz, and Timm Euler. 2006. YALE: rapid prototyping for complex data mining tasks. In *Proceedings of the Twelfth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Philadelphia, PA, USA, August 20-23, 2006*, Tina Eliassi-Rad, Lyle H. Ungar, Mark Craven, and Dimitrios Gunopulos (Eds.). ACM, 935–940. https://doi.org/10.1145/1150402.1150531

[33] Parastoo Mohagheghi and Vegard Dehlen. 2008. Where Is the Proof? - A Review of Experiences from Applying MDE in Industry. In *Model Driven Architecture - Foundations and Applications, 4th European Conference, ECMDA-FA 2008, Berlin, Germany, June 9-13, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 5095)*, Ina Schieferdecker and Alan Hartman (Eds.). Springer, 432–443. https://doi.org/10.1007/978-3-540-69100-6_31

[34] Gabor Nagy, Gergo Barta, and Tamás Henk. 2015. Portfolio optimization using local linear regression ensembles in RapidMiner. *CoRR* abs/1506.08690 (2015). arXiv:1506.08690 http://arxiv.org/abs/1506.08690

[35] Jakob Nielsen. 2005. Ten usability heuristics.

[36] Heungseok Park, Yoonsoo Nam, Ji-Hoon Kim, and Jaegul Choo. 2021. HyperTendril: Visual Analytics for User-Driven Hyperparameter Optimization of Deep Neural Networks. *IEEE Trans. Vis. Comput. Graph.* 27, 2 (2021), 1407–1416. https://doi.org/10.1109/TVCG.2020.3030380

[37] Jeffrey M Perkel. 2018. Why Jupyter is data scientists' computational notebook of choice. *Nature* 563, 7732 (2018), 145–147.

[38] Marian Petre. 1995. Why Looking Isn't Always Seeing: Readership Skills and Graphical Programming. *Commun. ACM* 38, 6 (1995), 33–44. https://doi.org/10.1145/203241.203251

[39] Bernadette M. Randles, Irene V. Pasquetto, Milena S. Golshan, and Christine L. Borgman. 2017. Using the Jupyter Notebook as a Tool for Open Science: An Empirical Study. In *Proceedings of the 17th ACM/IEEE Joint Conference on Digital Libraries* (Toronto, Ontario, Canada) *(JCDL '17)*. IEEE Press, 338–339.

[40] Ulf-Dietrich Reips and Frederik Funke. 2008. Interval-level measurement with visual analogue scales in Internet-based research: VAS Generator. *Behavior Research Methods* 40, 3 (2008). https://doi.org/10.3758/BRM.40.3.699

[41] Marco Túlio Ribeiro, Sameer Singh, and Carlos Guestrin. 2016. "Why Should I Trust You?": Explaining the Predictions of Any Classifier. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*, Balaji Krishnapuram, Mohak Shah, Alexander J. Smola, Charu C. Aggarwal, Dou Shen, and Rajeev Rastogi (Eds.). ACM, 1135–1144. https://doi.org/10.1145/2939672.2939778

[42] Elvinia Riccobene, Patrizia Scandurra, Alberto Rosti, and Sara Bocchio. 2006. A model-driven design environment for embedded systems. In *Proceedings of the 43rd Design Automation Conference, DAC 2006, San Francisco, CA, USA, July 24-28, 2006*, Ellen Sentovich (Ed.). ACM, 915–918. https://doi.org/10.1145/1146909.1147141

[43] Oliver Ritthoff, Ralf Klinkenberg, Simon Fischer, Ingo Mierswa, and Sven Felske. 2001. Yale: Yet Another Learning Environment.

[44] Theodosios Sapounidis, Stavros Demetriadis, Pantelis M. Papadopoulos, and Dimitrios Stamovlasis. 2019. Tangible and graphical programming with experienced children: A mixed methods analysis. *International Journal of Child-Computer Interaction* 19 (2019), 67–78. https://doi.org/10.1016/j.ijcci.2018.12.001

[45] Iqbal H. Sarker, Md. Hasan Furhad, and Raza Nowrozy. 2021. AI-Driven Cybersecurity: An Overview, Security Intelligence Modeling and Research Directions. *SN Comput. Sci.* 2, 3 (2021), 173. https://doi.org/10.1007/s42979-021-00557-0

[46] Christian Scheidler, Lorenz Schäfers, and Ottmar Krämer-Fuhrmann. 1993. TRAPPER: A Graphical Programming Environment for Industrial High-Performance Applications. In *PARLE '93, Parallel Architectures and Languages Europe, 5th International PARLE Conference, Munich, Germany, June 14-17, 1993, Proceedings (Lecture Notes in Computer Science, Vol. 694)*, Arndt Bode, Mike Reeve, and Gottfried Wolf (Eds.). Springer, 403–413. https://doi.org/10.1007/3-540-56891-3_32

[47] Ben Selwyn-Smith, Craig Anslow, and Michael Homer. 2022. Blocks, Blocks, and More Blocks-Based Programming. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Programming Abstractions and Interactive Notations, Tools, and Environments* (Auckland, New Zealand) *(PAINT 2022)*. Association for Computing Machinery, New York, NY, USA, 35–47. https://doi.org/10.1145/3563836.3568726

[48] Dvijesh J. Shastri. 2020. Machine Learning for Non-Programmers. In *Extended Abstracts of the 2020 CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) *(CHI EA '20)*. Association for Computing Machinery, New York, NY, USA, 1–3. https://doi.org/10.1145/3334480.3375051

[49] Ben Shneiderman. 1981. Direct manipulation: A step beyond programming languages (abstract only). In *Proceedings of the Joint Conference on Easier and More Productive Use of Computer Systems, CHI 1981, Part II: Human Interface and the User Interface, Ann Arbor, MI, USA, May 20-22, 1981*, Gregory A. Marks and Lorraine Borman (Eds.). ACM, 143. https://doi.org/10.1145/800276.810991

[50] Danli Wang, Yang Zhang, and Shengyong Chen. 2013. E-Block: A Tangible Programming Tool with Graphical Blocks. *Mathematical Problems in Engineering* 2013 (Feb. 2013), 598547.

[51] Jiawei Wang, Li Li, and Andreas Zeller. 2020. Better code, better sharing: on the need of analyzing jupyter notebooks. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results* (Seoul, South Korea) *(ICSE-NIER '20)*. Association for Computing Machinery, New York, NY, USA, 53–56. https://doi.org/10.1145/3377816.3381724

[52] Thomas Weber, Janina Ehe, and Sven Mayer. 2024. Extending Jupyter with Multi-Paradigm Editors. *Proc. ACM Hum.-Comput. Interact.* 8, EICS (2024), 1–22. https://doi.org/10.1145/3660247

[53] Thomas Weber and Heinrich Hußmann. 2022. Tooling for Developing Data-Driven Applications: Overview and Outlook. In *Proceedings of Mensch Und Computer 2022* (Darmstadt, Germany) *(MuC '22)*. Association for Computing Machinery, New York, NY, USA, 66–77. https://doi.org/10.1145/3543758.3543779

[54] Chen Xiajian, Wang Danli, and Wang Hongan. 2011. Design and implementation of a graphical programming tool for children. In *2011 IEEE International Conference on Computer Science and Automation Engineering*, Vol. 4. 572–576. https://doi.org/10.1109/CSAE.2011.5952915

[55] Quanshi Zhang and Song-Chun Zhu. 2018. Visual interpretability for deep learning: a survey. *Frontiers Inf. Technol. Electron. Eng.* 19, 1 (2018), 27–39. https://doi.org/10.1631/FITEE.1700808

[56] Chunqi Zhao, I-Chao Shen, Tsukasa Fukusato, Jun Kato, and Takeo Igarashi. 2022. ODEN: Live Programming for Neural Network Architecture Editing. In *27th International Conference on Intelligent User Interfaces* (Helsinki, Finland) *(IUI '22)*. Association for Computing Machinery, New York, NY, USA, 392–404. https://doi.org/10.1145/3490099.3511120

[57] Guangxu Zhu, Dongzhu Liu, Yuqing Du, Changsheng You, Jun Zhang, and Kaibin Huang. 2020. Toward an Intelligent Edge: Wireless Communication Meets Machine Learning. *IEEE Communications Magazine* 58, 1 (2020), 19–25. https://doi.org/10.1109/MCOM.001.1900103