An Investigation of How Software Developers Read Machine Learning Code

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

Christina Winiker
LMU Munich
Munich, Germany
christina.winiker@stud.ifi.lmu.de

Sven Mayer
LMU Munich
Munich, Germany
info@sven-mayer.com

Abstract

Background Machine Learning plays an ever-growing role in everyday software. This means a paradigmatic shift in how software operators from algorithm-centered software where the developers defines the functionality to data-driven development where behavior is inferred from data.

Aims The goal of our research is to determine how this paradigmatic shift materializes in the written code and whether developers are aware of these changes and how they affect their behavior.

Method To this end, we perform static analysis of 3,515 software repositories to determine structural differences in the code. Following this, we conducted a user study using eye tracking (N=18) to determine how the code reading of developers differs when reading Machine Learning source code versus traditional code.

Results The results show that there are structural differences in the code of this paradigmatically different software. Developers appear to adapt their mental models with growing experience resulting in distinctly different reading patterns.

Conclusions These difference highlight that we cannot treat all code the same but require paradigm-specific, empirically validated support mechanisms to help developers write high-quality code .

CCS Concepts

• **Software and its engineering** → Software creation and management; • **Computing methodologies** → **Machine learning**; • **Human-centered computing** → **Empirical studies in HCI**.

Keywords

software developers, eye tracking, code reading, human computer interaction

ACM Reference Format:
Thomas Weber, Christina Winiker, and Sven Mayer. 2024. An Investigation of How Software Developers Read Machine Learning Code. In *Proceedings of the 18th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '24), October 24–25, 2024, Barcelona, Spain.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3674805.3686678

ESEM '24, October 24–25, 2024, Barcelona, Spain

ACM ISBN 979-8-4007-1047-6/24/10
https://doi.org/10.1145/3674805.3686678

1 Introduction

Nowadays, researchers and practitioners develop Machine Learning for many different domains. HereIt enables us to develop more effective systems or even some functionality that was not possible before, e.g., in medicine [41], disaster management [31], and the automotive industry [29]. This growing interest naturally means that these domains increasingly depend on the quality of these data-driven software systems. Therefore, developers are increasingly responsible for delivering high-quality, reliable software regardless of complexity. However, by using Machine Learning, programming is shifting from explicitly defining machine instructions to providing the appropriate data, context, and infrastructure for the machine to learn. A program's desired behavior is then not primarily defined by instructions in code but implicitly encoded in the data or, for example, in the weights and biases of a Neural Network. This can be extremely challenging to understand for humans. Furthermore, the code that developers *do* write themselves becomes increasingly abstract. Karpathy [23] also refers to software based on Machine Learning and Neural Networks as "Software 2.0" to denote this fundamental shift in development practice. While the technological differences between the traditional "Software 1.0" and "Software 2.0" are well understood [4], it is not fully clear how this paradigmatic shift changes the work of developers. To support developers of data-driven applications, we must first understand the different aspects of the developers' behavior and how development has changed.

For traditional development, decades of research exist in software Engineering [54], which has yielded processes, best practices, and tools [19, 55] that have proven themselves in practice, giving us a firm understanding of the developers' behavior. "Software 2.0" and *data-driven development* still need to transition from *data science* to *data-driven engineering* [3, 18], and a human-centered understanding of the difference can help with this. Furthermore, given that current tools and programming languages for data-driven development have either adapted or emerged from their traditional counterpart, the obvious question is how much the underlying behavior differs and whether we can build on existing knowledge from Software Engineering research. Therefore, we must first determine how far these development paradigms deviate from each other and how these differences can impact the developers behavior.

In this paper specifically, we focus on how software developers read code in this context. Prior work has, so far, investigated code reading behavior for traditional code, e.g., [10, 39, 44]. In this work, we go beyond this related work and compare the reading behavior for traditional code with that for code that implements Machine Learning models. To this end, we first perform static analysis on publicly available code repositories that contain traditional and machine learning code. Motivated by the structural differences this

showed, we investigate whether these differences have an impact on the reading behavior of developers. We utilize eye tracking to record the developers' gaze to investigate the following questions:

RQ1 What structures in the code are of interest to developers in Machine Learning code compared to code that does not use Machine Learning?

We expand on the existing literature by including data science experts in the experiments' sample, which should be more representative and yield more valuable insights. Additionally, testing both novices and experts allows us to compare these target groups to determine the effect of prior knowledge and expertise:

RQ2 Does the expertise of experienced Machine Learning developers lead to them focusing on different aspects of the Machine Learning code compared to novice developers?

To address these questions, we conducted a user study (N=18) in which we asked participants to engage with different Python code fragments (e.g., see Figure 2). Our results showed differences in the reading behavior, which show that some syntactic structures are read differently in code that implements Machine Learning models. This also implies that developers are aware of the differences and adapt their behavior accordingly. Thus, for supporting developers, machine learning code should not be treated the same as traditional code. We also uncovered differences in expertise, which further underpin the need for approaches specific to the target group and context rather than one-size-fits-all solutions.

Given the changing nature of data-driven development as a paradigm for software creation, our insights inform decisions on how to better support developers. Future development can achieve this in numerous ways, e.g., via the design of dedicated programming languages, libraries and frameworks, and support mechanisms and tools that take into account which parts of the code are relevant or not. Thus, we see this work as a foundational step in assessing developer behavior in the evolving field of data-driven development.

2 Related Work

While the concept and field of Machine Learning are many decades old, more recent advances like deep learning, i.e., Machine Learning with highly complex models, and an increase in computational power have propelled it into the mainstream for software development. As previously discussed, this brings with it changes to software development. The following section will briefly outline how literature has addressed these changes. Beyond that, we will also provide some background about the methodology we used to advance the state of research in this field, eye tracking.

2.1 Developing Machine Learning Systems

Developers now use Machine Learning (ML) in many areas; thus, it is essential for software engineers to be familiar with the general concepts. However, ML adds additional complexity to the development since developers must create and improve not just the code but also the underlying data and ML models [20]. Thus, making these complex ML systems easier to understand is important. While there are successful efforts to make traditional code easier to understand, e.g. using ML [7], the different level of complexity in code of data-driven applications remains an open challenge.

Prior work has recognized developers as a valuable target group for improving the ML experience [3, 11, 35, 36, 40, 42]. Consequently, researchers and industry have produced several methods, tools, and support mechanisms to assist software developers in creating data-driven applications. Here, they have created tools to support ML programming in a more user-friendly manner for decades. One of the earlier, well-known ones is *RapidMiner* (originally *YALE*) [43], which leverages graphical programming to present complex data processing pipelines in a simple, visual format. This graphical approach has been extended and adopted in many tools since, like *Orange* [13], *Darwin* [15], *KNIME* [8], and many more.

Nonetheless, textual programming remains the predominant way of creating ML systems. At this level, the most substantial improvements have come more from the now ubiquitous libraries and frameworks (e.g., *scikit-learn* [38] or *TensorFlow* [1]), which take over the low-level details and aim to provide easy-to-use abstractions. Their development often happens in an ad-hoc fashion and still needs more validation, e.g., whether the provided abstractions are appropriate. From the language design perspective, there is a growing effort to optimize various aspects of programming languages for ML, e.g., with the Compilers for Machine Learning workshop series. However, so far, they have focused more on technical implementation. Meanwhile, improving the literal syntax or creating ML-specific languages specific remains an interesting avenue to make ML programming effective or efficient.

For now, Python is arguably one of the most widely adopted programming languages in the area of data-driven applications, particularly in combination with computational notebooks [50] like Jupyter notebooks and Google Colab. This type of editor is a traditional code editor at its core but offers increased interactivity and allows developers to mix documentation, code, and outputs in one document. Naturally then, there are efforts to improve these notebooks as well. The mage [26, 27] extensions for Jupyter notebooks, for example, adds interactive widgets which let data workers modify the output of their code. These changes are automatically reflected, similar to *Programming-by-Example* systems [32]. The feedback from a subsequent user study was positive, highlighting some of the challenges of ML code which can be complex, particularly for novices. Therefore, they suggested hiding parts of the code that would confuse them. More experienced developers, on the other hand, viewed the presentation in code more favorably.

The interactivity of computational notebooks greatly facilitates developers' exploratory behavior. This also brings challenges; since ML systems are a tight interplay between models and data, maintaining an overview of various configurations can be difficult [12, 24, 25]. The evaluations conducted for some of these systems show their potential benefits, but these are difficult to separate from the concrete implementations. Creating fully working software tools like that requires much development upfront. Rigorous requirement elicitation is desirable to motivate this kind of effort. Yet, while some of these projects start with an initial phase of gathering user requirements [14, 24, 26] the reporting typically emphasizes summative evaluations. Additionally, they often build on anecdotal evidence and the experience of their creators. This limits how well other researchers can rely on the insights that informed the original design. We argue that this common top-down approach, i.e., evaluating a full, working system with a focus on summative results, can be supported by

a bottom-up perspective, i.e., early formative assessment, where one investigates various low-level aspects of the development first to get a foundational understanding of the requirements of data-driven development (cf. [28, 33]). These insights inform further design decisions, especially with existing knowledge from traditional software development. Instead, we can first determine where these development paradigms match and differ. To get a comprehensive overview, so far, researchers investigated multiple different aspects that play a role during software development, from development processes to collaboration or how developers write code. Hesenius et al. [18], for example, investigate these differences on the process level and highlight where current development processes do not accommodate data-driven development. In consequence, they propose an extension to established development processes, the Engineering Data Driven Applications (EDDA) process, which contains additional steps specific to data-driven development.

Given the number of publicly available software projects on online collaboration platforms, they have also become a prominent source for further insights into how developers work. For example, Simmons et al. [51] compared roughly 2,000 public repositories, half with ML and half using traditional code, concerning code quality. Their analysis concludes that there are some similarities but notable differences between traditional and ML code. While the complexity of the code base, as measured by McCabe's Cyclomatic Complexity [30], is similar, it appears that repositories of data-driven applications adhere less to existing coding and naming conventions. While using the code repositories alone, they could not determine the reason for these differences. They did highlight differences in the developers' behavior, which warrant further investigation. However, the use of code quality metrics leaves it unclear whether the repositories are, in fact, also structurally different or whether the same structures are just present in a different quality.

In this paper, we will continue this line of research to determine the differences between ML and traditional code. However, we will primarily focus on human behavior using eye tracking, a method frequently used in empirical analyses of software developers.

2.2 Research using Eye Tracking

Sharafi et al. [46] provide an overview of several publications which emphasize the need for a uniform nomenclature to carry out eye-tracking studies, particularly for behavioral observation of Software Engineering. Their paper provides an overview of terms such as "fixation rate," "saccades," and their use in existing eye-tracking studies in Software Engineering. We will follow the terminology by Sharafi et al. [46]. In addition to that, Sharafi et al. [47] expand on this with a systematic review of eye tracking in software engineering, showcasing the benefits of the method for investigating several questions, including but not limited to debugging [5, 17], code comprehension [6, 44], and collaboration [48], all of which could contribute to a better understanding of software development [21].

Eye tracking as code reading behavior measurement has been discussed in a series of publications. For example, Busjahn et al. [10] compared the linearity of the reading order in software code structures of inexperienced and experienced software developers. They analyzed this by first having inexperienced programmers read natural language text, then programming code, and then having experienced programmers read code. They also show that inexperienced programmers read code less linearly than natural text but still more linearly than experts [10]. Peachock et al. [37] show, in a replication study, that developers read source code less linearly than natural text but that there are no differences in linearity regarding the level of experience or programming skills.

Peitek et al. [39] investigated the linearity and reading order of code. They conclude that "the linearity of source code is a major driving factor that determines programmers' reading order, while experience and comprehension strategy seems to play more minor roles" [39]. Since the functionality of Machine Learning systems, particularly those using Neural Networks, depend less on the linearity of instructions in the code and more on implicit factors like the data, weights, etc., this effect may not translate to this different development paradigm and thus warrants further analysis.

Weber et al. [53] investigated these differences and leveraged the structure of code to determine which elements in ML code are areas of interest for novice programmers. They found reading patterns that suggest that there are benefits to linearity in this context. Since the study by Peitek et al. [39], which suggests that prior knowledge is only of minor importance for code reading behavior, was on traditional code only, and the study by Weber et al. [53] was on novices only, it remains still unclear how ML professionals and experts are affected. Particularly for a paradigm like Machine Learning, which may not yet be part of the early education for all programmers, expertise may play a more significant role.

Thus, we can conclude that the reading behavior of developers can but does not have to vary based on multiple factors like code structure and prior knowledge. In the context of Machine Learning code, these effects have only been explored for novices [53]. So, we expand on this in this work by collecting new data from a study that includes experts. This should increase generalizability and provide insights into an important target group.

3 Static Analysis of Code Repositories

In preparation for our study of developer behavior, we analysed publicly available code repositories to determine whether there were quantifiable differences between the two types of repositories – repositories with machine learning functionality and without. For this, we selected 3,515 code repositories from the literature [9, 51] more recent additions, based on GitHub's popularity metrics, and counted the different grammatical constructs in the code. Of these repositories 1,534 contained applications that utilize machine learning, while the remaining 1,981 were traditional applications. We screened all repositories to manually decide to which of these two categories each repository belonged, since, for example, metadata indicating *machine learning* content was not enough to decide whether a repository used ML functionality or implemented it using traditional programming methods. These repositories all used the Python programming language, since it is both general purpose language with wide adoption but is also very popular for machine learning [52], making it an ideal candidate for comparative studies.

After downloading the current state of these repositories, we used the Python *ast* package to construct the Abstract Syntax Trees for the Python code in the repositories. We accumulated the number of occurrences for each grammar construct from Python's formal

grammar for all valid ASTs (1,431 traditional repositories, 885 with ML) and then normalized that by the total repository size. Thus, we received a relative frequency for each construct for every repository.

We conducted a statistical analysis on the log-transformed data. Based on visual analysis and the Shapiro-Wilk test, we used both Student's t-test and Mann-Whitney-U test, where we compared the relative frequencies between each pair of data for the machine learning and the traditional repositories. Of the 86 different pairs of code constructs as part of the formal grammar, we found significant differences ($p < 0.05$) for 79 different code constructs (see supplementary for the complete listing).

From this, we concluded that there are structural differences between the two development paradigms. Certain code elements seem to appear more frequently, while others seem to be less common in ML code. For example, ML repositories appear to have a relatively higher ratio of imports but fewer instances of exception handling compared to repositories with traditional code. Thus, we continued to investigate whether developers are implicitly or explicitly aware of these differences and how they incorporate this in their behavior. Thus, we conducted the following eye tracking study.

4 Methodology

To determine the reading behavior of developers for Machine Learning code, we adopt and adapt the methodology proposed by Weber et al. [53], which relies on the structured nature of software code. We expand upon their methodology, though, with a systematic selection of the code fragments to enhance comparability. The following section describes this general methodology and any changes to the process which we took to address our research questions.

4.1 Study Design

We performed an in-person, within-subject user study using eye-tracking data from developers while they read a selection of code fragments. We used a graphical user interface nearly identical to that in the study of Rodeghero et al. [44], which uniformly displayed the code. Like in the original interface, we used no syntax highlighting or other visual support mechanisms, as this could impact the gaze, e.g., via preattentive perception. At this point, we do not yet know which code structures we could or should highlight, as this is part of the potential implications of our study, so we opted for the highlight-less approach. Nonetheless, the UI displayed the code in a monospaced font, as is typical for code editors. This also helped with defining clearly distinct areas of interest later on. We also excluded any comments from the code, as they could draw a disproportionate amount of attention and thus skew the results. See the work of Rodeghero et al. [44] for a further discussion of the benefits and drawbacks of this presentation. Since we used a *within-subject* design, each participant used the UI to read each code fragment described below.

4.2 Procedure

Besides displaying the code, the interface served a secondary function: to create engagement of the study participants with the code. The participants had the task of summarizing each code fragment's key functionality and purpose in their own words. In this way, we required participants to read the code carefully. We asked them to type their summary directly in the interface, which offered an input field next to the code on the right half of the screen. To better understand their behavior, we also used the think-aloud protocol [49], i.e. we asked participants to verbalize their behavior and thought process, which we manually transcribed. After fragment summarization, they moved on to the next code fragment. We limited the total duration of the study to 40 minutes, i.e., five minutes per fragment, to prevent effects due to exhaustion. However, this only served as a guideline for the participants; they were free to move on from a code fragment if they believed they had sufficiently summarized it. Beyond this, we also collected additional feedback about the code fragments, demographic data, and prior knowledge with a questionnaire at the end.

4.3 Selection of the Code Fragments

In total, we selected eight Python code fragments for our study. We only used code written in the Python programming language, which we previously also selected for the static analysis due to its popularity in general but also for machine learning [52]. To facilitate the comparison in *RQ1*, half of these was code that uses Machine Learning in some capacity, while the other half was traditional code without Machine Learning.

Since code quality can differ widely in software projects, we selected our code fragments from Python libraries with a large user base. The idea behind this is that these libraries typically have fairly high standards concerning code style and quality and act as a role model for other developers. Therefore, we picked our fragments evenly from example code from the repositories of the TensorFlow [1] and scikit-learn [38] libraries for ML code, i.e. using code that uses these libraries, and from the matplotlib, flask, and request libraries for the non-ML code. For the ML code examples, we made sure to pick code that uses multiple different ML models using both unsupervised and supervised learning. We looked for similar pairwise ML and non-ML fragments to enable gaze pattern comparison. The first criterion for this was the length in lines of code where we paired fragments of equal length. We also had to consider our screen size since we wanted to minimize interfering variables, such as scrolling through long code examples. Therefore the code fragments did not exceed 60 lines of code.

We followed the approach by Shaffer et al. [45] and Weber et al. [53], who used the constructs from the programming languages' formal grammar to determine areas of interest in the gaze data. While we had previously shown that different constructs occur at different frequencies, picking code that would accurately reflect these differences would skew the gaze dwell times. Thus, we aimed for a similar distribution of the constructs for each pair of ML and non-ML code fragments, while keeping distinctly ML and non ML functionality in the code. Table 1 lists the frequency of construct in each pair of code fragments. Since not all 84 grammar constructs were present in the code, the following evaluation will deal exclusively with those present. For each participant, we displayed ML and traditional code fragments in an alternating sequence.

Finally, we selected two sets of code fragments to account for the different expertise levels of novices and experts. This allows us to probe **RQ2** further. As Kahney [22] discussed, choosing a single,

simple task can easily lead to task performance becoming indistinguishable between the groups. Gugerty and Olson [16] compared novices and experts using the same LOGO code fragments with only minor modifications for the experts, but also discuss that this approach has the issue that with such similar code, one cannot fully explore the experts' skills. Using two structurally similar sets of different complexity, we hoped not to overwhelm the novices and, at the same time, ensure that the experts read source code appropriate for their prior knowledge, thus avoiding the effects of mental over- and underload. To gauge how effective the selection was, we collected feedback on the code fragments using the questionnaire, qualitatively using free text answers, and quantitatively using a five-point Likert scale, where participants provided their perspectives on how complex and challenging they considered the two types of code fragments. Furthermore, we collected the code summarizations each participant wrote to gauge whether they understood the key concepts and actions in the code.

4.4 Apparatus

We recorded the gaze of our participants using a *Tobii 4C* on-screen eye tracker, which provides data at a sample rate of 90 Hz. The tracker was attached to a 20" monitor, set up in a well-lit room without outside distractions. We calibrated the tracker for each participant individually before reading the code.

4.5 Participants

We invited 18 participants from local companies and academic institutions via personal and professional contacts. For their participation of up to one hour, they received compensation equivalent to 10 USD. Of the twelve experienced and six novice developers, half were female, and half were male, with a mean age of 26 years (SD: 2.78). With a background in computer science, they were either in the final year of their Master's program or had a completed Master's degree and were now either working in industry or pursuing a Ph.D. In addition, two-thirds of the experts reported currently working on one or more projects involving a substantial amount of ML. The remaining third was not actively working on such a project but had prior experience with the technology.

Our distinction between novices and experts is based on participants' self-assessment during recruitment and is corroborated by additional background information which the participants provided in the questionnaire. For example, all novices but one listed their programming experience to be between three and five years, while most experts have at least five years of general programming experience. Focusing on the Python programming language, all participants were familiar with it, although all experts indicated a high or very high level of expertise (5-point Likert scale), while the novices rated their knowledge as low to average. The same applies to prior knowledge about Machine Learning, where the two groups are discretely split along the middle of the Likert scale.

4.6 Data Processing

The structured nature of software code and its formal division into defined structural elements allows for very defined areas of interest. In addition, the presentation using a monospaced font gives clear bounding boxes for gaze targets. After collecting the gaze data in the user study, we applied an analytical approach as follows, in line with Weber et al. [53]. For this process, we recorded how many gaze points fell into each bounding box and could then determine what grammatical construct in the code and, thus, to which programming concept each gaze point belongs. Figure 2 shows an example of the subdivision. We then counted the number of recorded gaze points per area, thus giving us a measure of the dwell time for each area and, when accumulated, for each type of code element.

Not all participants spent the same time reading each code fragment, so we consider only the relative percentage of gaze points per participant. Furthermore, different structural code elements take up more screen space, so naturally, they attract more gaze points than smaller code elements. Thus, we also divided the gaze point percentage by each code fragment's relative area. Since we use a monospace font, this automatically also normalizes the data against character count. Thus, the resulting data is normalized against temporal and spatial factors and allows for a comparison between groups and code elements to answer the research questions.

To determine differences in the participant's code-reading behavior, we first used heatmaps and a replay of the participants' gaze paths to gain a high-level overview of the data. Second, we used a quantitative method to determine the dwell time for each area of interest, i.e., each syntactical code structure, which allowed us to find general reading patterns and specific parts of the code which caught the developers' attention. After normalizing the gaze to accommodate user-specific offsets, we visualized the heatmaps using the kernel density estimation (KDE) of the seaborn library with the Epanechnikov Kernel. We used Silverman Algorithm [2] for bandwidth optimization, leading to heatmaps, as shown in Figure 1. We overlaid each heatmap with its respective code fragment to better determine which part of the code participants looked at.

While these visualizations already suggest some interesting patterns, purely accumulated dwell times may not accurately reflect, for example, frequent jumps between the same points, thus, we also employed a tool for replaying the gaze tracks. With this tool, we could visualize the path of gaze points across time to review the eye movement across code fragments. This showed the reading progression of our participants, clearly indicating, e.g., what elements participants looked at first. It also provided a first glance at which elements were most relevant, as indicated by a long dwell time.

5 Results

In the following section, we will summarize the results of the user study and the analysis of the recorded gaze data using the methodology described in the previous section and the literature.

5.1 Research Question 1

To address **RQ1**, we first investigate the reading behavior for different code fragments, specifically whether code with ML elements is read differently to code without. When we compared the reading behavior of different developers, we could already see different search strategies being used by participants. Replaying the participants' gaze, we saw that several participants read code fragments sequentially, top to bottom, showcasing patterns akin to Nielsen's F-patterns [34]. The F-pattern means that the reader first performs horizontal scans of the document and then continues with vertical

Table 1: Absolute number of each grammatical code construct per code fragment for both novices and experts. We paired ML and non-ML code fragments based on a similar distribution frequency of those constructs.

	Code for novice developers								Code for expert developers							
	Pair 1		Pair 2		Pair 3		Pair 4		Pair 5		Pair 6		Pair 7		Pair 8	
	ML	non-ML	ML	non-ML	ML	non-ML	ML	non-ML	ML	non-ML	ML	non-ML	ML	non-ML	ML	non-ML
Import statement	2	-	2	2	-	-	-	-	3	5	3	1	8	3	7	2
Variable declaration	4	2	8	2	9	8	7	13	7	22	9	9	19	11	12	14
Function definition	-	1	-	-	2	1	1	3	-	-	-	-	2	-	-	1
Function body	-	1	-	-	2	1	1	3	-	-	-	-	2	-	-	1
Function parameter	2	4	6	4	14	6	28	10	17	30	14	9	18	12	20	26
Function call	3	2	7	5	11	9	39	3	21	28	21	6	18	14	18	27
Literal	3	3	2	3	11	12	70	15	22	41	14	4	56	18	24	36
Index based access	1	-	4	-	3	5	12	-	4	7	2	1	1	-	-	-
if block condition	-	3	-	-	-	4	5	8	-	-	-	-	-	-	-	1
if block body	-	2	-	-	-	1	5	7	-	-	-	-	-	-	-	1
Loop condition	-	1	1	1	2	-	-	-	1	-	-	-	2	1	-	1
Loop body	-	1	1	1	1	-	-	-	1	-	-	-	-	1	-	1
Return statement	-	1	-	-	-	1	-	1	-	-	-	-	2	-	-	1
Total lines of code	9	7	16	9	20	23	38	41	51	59	30	30	60	39	59	60

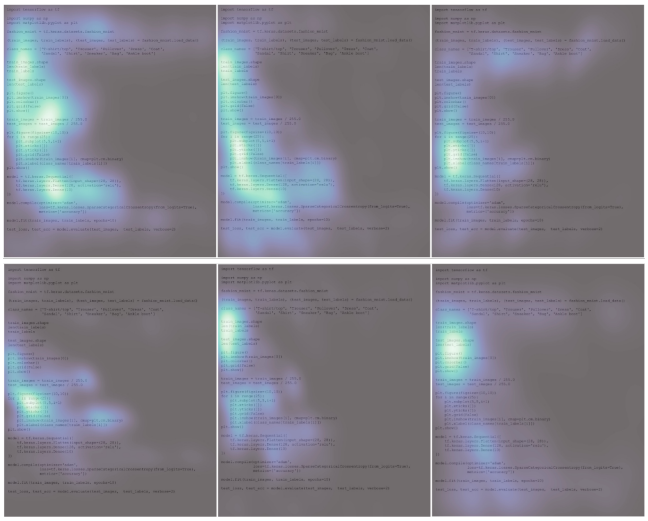

Figure 1: Heatmap of different search and reading strategies. A replay of the gaze tracks showed that the novices (top) showed mostly sequential reading, while the experts (bottom) jumped to selective points of interest.

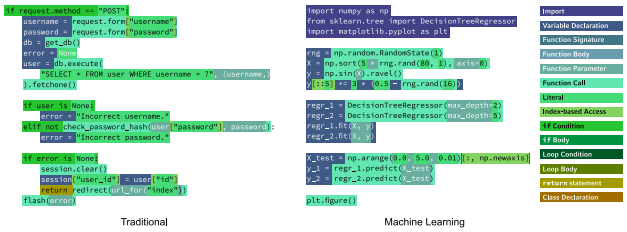

Figure 2: Examples from each type of code and how they were divided into discrete areas of interest based on Pythons' formal grammar.

scanning. Those who do not show the F-pattern behavior ignore or skim the majority of the code and jump to areas of seeming interest, where they remain for a considerable amount of time.

Reading behavior without F-patterns is much more prominent in code that implements ML models, while the traditional code is read sequentially for the majority of the time. Of the eight code fragments per participant, we observed sequential reading in the traditional code 46 times and 28 times in ML code. Selectively reading ML code fragments also includes backtracking, i.e., jumping back to parts of the code that participants had previously read. This is also much less pronounced for the traditional code fragments.

To further investigate which parts of the code are seemingly more important, we used code's grammatical structure to analyze at which elements the participants look at the longest and, thus, may be particularly interesting. Therefore, we subdivided the total text area into small rectangular areas of interest for each type of code element and determined the dwell time for each, see Figure 2. We then normalized the dwell time to account for differences in area and how long different participants spent for the whole code fragment. Figure 3 shows the normalized dwell time for the two groups in our study and the different code elements for source code that does and does not use ML. This already visualizes that certain fragments in the code receive relatively more attention than others.

Considering that the two groups, experts and novices, had different code fragments according to their skill levels, we analyzed the two groups separately. In addition, a first inspection also showed that participants entirely ignored some of the code structures in some test conditions. Thus, we first created a binary distinction at which code fragments novices and experts had looked at. This analysis showed that participants ignored some code elements, namely conditional *if*-statements, class definitions, and function definitions. However, this is an infrequent behavior. Return statements, while similarly infrequent, received more attention. Here, one of the participants even mentioned that they consider return statements an important indicator for the function of a code fragment. Based on these initial insights, we will not interpret behavior for code that participants did not read in any of the study conditions in the further analysis since we cannot determine meaningful statistical

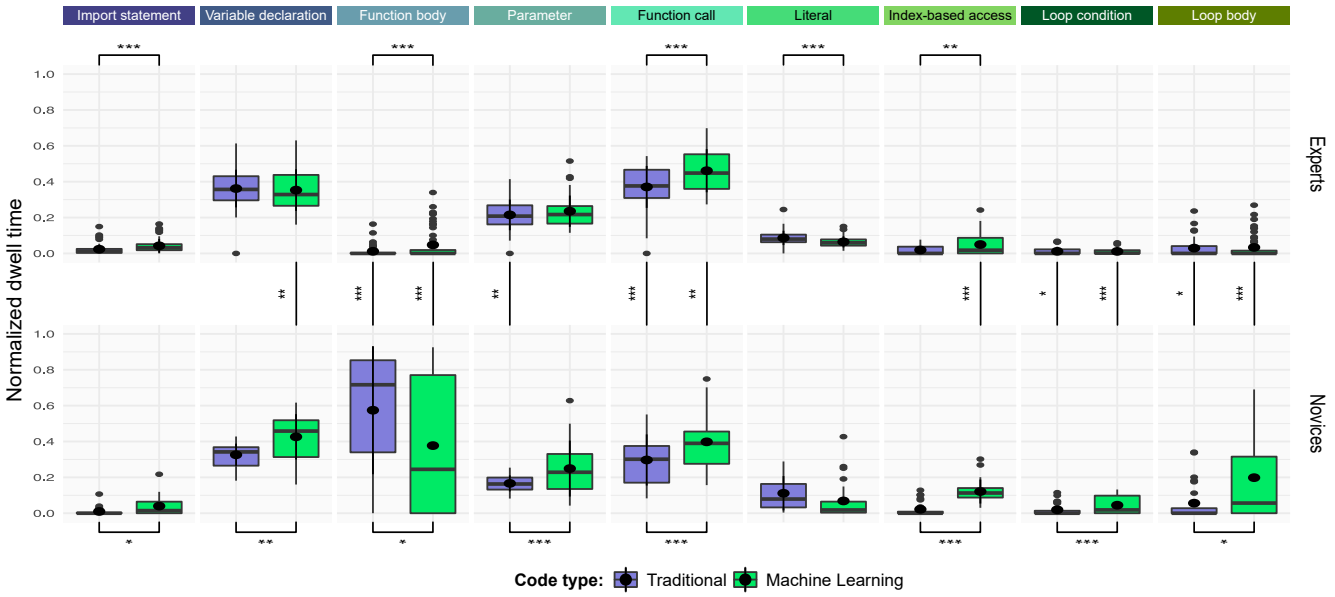

Figure 3: Distribution of the relative number of gaze points per area of interest for both participant groups and both types of code fragment. (*: $p < 0.05$, **: $p < 0.01$, ***: $p < 0.001$, cf. Table 2, Table 3)

results beyond the binary distinction, see Figure 3. This resulted in nine types of code elements for further analysis.

To systematically investigate the question of whether there are differences between the two types of code, we compare the mean normalized dwell time for each code element between the code fragments with and without ML, see Table 2. Based on the results of a Shapiro-Wilk test, we then performed a paired t-test or Wilcoxon test for all code elements to determine which statistically significant differences between *Traditional* and *ML* exist for the two groups, *Novices* and *Experts*, independently. There are several significant differences between the types of code for both groups, see Table 2. We note that these differences are not the same for both groups.

We found similar differences for function, import statements, and index-based data access, with both groups spending significantly more time. The experts also spend significantly less time on literals in ML code. On the other hand, the novices looked more at function parameters and variable declarations in the ML fragments.

5.2 Research Question 2

To answer the second research question (**RQ2**), we compare the six novice developers and their gaze behavior with that of the twelve that report a higher degree of experience with Machine Learning. While these two groups read two different types of code fragments, we ensured to select syntactically similar code fragments of roughly equal length and structure (see Table 1). Based on their feedback, both groups also considered the code fragments semantically to be of similar, medium difficulty, as indicated by the median of the Likert scales being the neutral option and no significant differences (Mann-Whitney-U-test, $p = 0.294$) between the study conditions.

Concerning the search strategies, it was noticeable that of the six novice developers, four showed F-patterns across both types of source code, while the other two sometimes followed this strategy and sometimes jumped with their gaze. We could, however, not determine how the code influenced the pattern concerning the type of code when they would choose one or the other strategy. In comparison, the experienced developers also applied the sequential, F-pattern-based reading strategy for the traditional Python code but for the Machine Learning code only at most half, typically three to four, were scanning through the code fragments while the other half was selective in their reading. It is worth mentioning that all experts applied both strategies at some point. This indicates that expertise may come with knowledge about which parts of the code are worth looking at and which are not.

When asked what aspects of the code they deemed most helpful to foster understanding, almost all novices mentioned clear variable names. They also listed comments, although, as previously mentioned, we removed them from the code fragments. For the experts, seven mentioned the names of certain models in the code and especially in the import statements to be helpful. This also showed in the reading behavior where the less experienced participants, who selectively read the ML fragments, often focused not on ML-specific keywords, unlike the expert, but instead paid attention to more commonly understood terminology like the use of plots to visualize results. This effect was visible in relatively longer dwell times and was corroborated by the feedback in both the code summarization and the questionnaire. In comparison, these parts were almost always skipped by the expert developers who were, as they mentioned, more interested in more advanced terminology, like the names of the ML models used in the code. Based on the feedback in the questionnaire, the novices did, however, notice some of these ML-specific keywords but, as they were sometimes unfamiliar with them, looked elsewhere for cues to understand the code.

Table 2: Statistical analysis of differences between the dwell time in various code elements in ML code fragments vs. non-ML code fragments for both participant groups. All Wilcoxon-test are indicated with a $^{\triangle}$. (*: $p < 0.05$, **: $p < 0.01$, ***: $p < 0.001$)

	Novices						Experts					
	Normality test		t-test / Wilcoxon-test				Normality test		t-test / Wilcoxon-test			
	W	p	t/W	df	p		W	p	t/W	df	p	
Import statement	0.866	.059	3.593	5	0.017	*	0.882	0.009	$78^{\triangle}$	11	< 0.001	***
Variable declaration	0.968	.893	4.9127	5	0.004	**	0.970	0.675	−0.507	11	0.620	
Function body	0.823	.017	$0^{\triangle}$	5	0.031	*	0.925	0.076	8.148	11	< 0.001	***
Parameter	0.957	.739	8.3604	5	< 0.001	***	0.986	0.972	1.945	11	0.079	
Function call	0.924	.319	10.931	5	< 0.001	***	0.951	0.281	5.569	11	< 0.001	***
Literal	0.951	.646	−2.3253	5	0.068		0.925	0.077	−5.785	11	< 0.001	***
Index-based access	0.875	.076	7.3625	5	< 0.001	***	0.898	0.020	$78^{\triangle}$	11	0.003	**
Loop condition	0.900	.161	20.759	5	< 0.001	***	0.955	0.353	−0.210	11	0.838	
Loop body	0.824	.017	$21^{\triangle}$	5	0.031	*	0.959	0.427	0.461	11	0.653	

Beyond this qualitative feedback, we also performed the quantitative analysis of the gaze data for differences between expertise, as listed in Table 3 and visualized in Figure 3. One difference that immediately becomes apparent is the fact that the distribution of gaze data of the novices is more spread across the different code elements. Experts, on the other hand, focus the majority of their attention on function calls and their parameters, as well as variable declarations, while the other elements play a relatively minor role. Still, function parameters and variables declarations are similarly important for the novices, but the function calls receive significantly more attention from the experts, both in traditional and ML code fragments. For function parameters and variable declarations, there appears to be a certain dependence on the type of code: experts value function parameters significantly more than novices in traditional code, while novices spend more time with variables in ML code. Beyond this, control flow via loops offers consistent differences, where novices look at loop conditions and the loop body significantly longer. The same also applies to the encapsulation of code in functions, with novices spending a significantly larger portion of their gaze on the body of such functions. Lastly, the parts to do with index-based data access in lists or similar data structures are notable. Here, novices spend little time reading them in the traditional code examples, just like the experts, but spend significantly more time on them in the ML code. The answers in the questionnaire provide some indication as to why this might be: Python has a syntax that goes beyond simple numerical access of list elements and allows, for example, to select ranges or "slices" of data. While this syntax is often used in ML code, it has less frequent applications in traditional code. For this reason, two novice participants mentioned that they found this unfamiliar syntax challenging.

6 Discussion

Our results show differences between the development paradigm and the respective source code and between the developers of different levels of expertise. This suggests that developers are aware of the structural differences in machine learning code and have developed behavioral strategies for different scenarios and code types. Specifically, we saw different search strategies when reading code, depending on expertise, where sequential scanning in F-patterns was more prominent for novice developers and depending on the development paradigm. Here, code without ML was more likely to be read in order from top to bottom. These two observations may very well be connected since the expert could be aware that ML code warrants a distinct reading strategy, while the novices fall back to the same way of reading they are used to from traditional code and other areas. If this is the case, it implies that certain aspects of the ML code are either trivial or entirely irrelevant to people with prior ML knowledge. In turn, this has implications for tool and framework design, which should strive to eliminate irrelevant steps by abstraction so that developers do not spend time on them. However, as this distinction was not completely binary, it may also be the case that some information is only unnecessary in some use cases and for some people, so flexibility and control over what is hidden should be considered. An example from which we can take inspiration is how modern IDEs handle import statements: since they are often automatically generated and considered to be of little relevance, they are typically "folded," i.e., displayed as a group in a single line. The developer can expand this presentation, offering the necessary flexibility should a developer be interested in specific import statements. Similarly, an IDE for data-driven development could automatically detect a sequence of preparatory or visualization steps and then collapse them into a single line labeled "data preparation" or "data visualization," respectively. Automatic refactoring is an alternative that likewise detects these common blocks and suggests moving them into a separate function, increasing encapsulation, which adds to code readability and maintainability.

Sequential reading behavior may lead to issues with attention in our study but also in general: in many of the code examples we surveyed for our study, the first part of the code is preparatory while the latter parts deal with the Machine Learning parts. This is a widespread pattern, but the nature of F-pattern-shaped reading could mean that the novices spend considerable attention on early, less interesting parts and have little attention left for the learning models at the end. At the same time, this could explain why experts often skip early parts and directly jump further toward the end. As mentioned above, one option is to minimize the space these parts take up could alleviate this. Additionally, by extracting common steps (e.g., data preparation, model configuration, training), an

Table 3: Statistical analysis of differences in the normalized dwell times between the two participant groups, novices, and experts, for both traditional code fragments without Machine Learning and those with Machine Learning. All Mann–Whitney U tests are indicated with a $^{\triangle}$. (*: $p < 0.05$, **: $p < 0.01$, ***: $p < 0.001$)

	Traditional Code Fragments						Machine Learning Code Fragments					
	Normality		t-test / MWU-test				Normality		t-test / MWU-test			
	W	p	t/W	df	p		W	p	t/W	df	p	
Import statement	0.752	< 0.001	$52^{\triangle}$	17	0.151		0.892	0.042	$38^{\triangle}$	17	0.892	
Variable declaration	0.974	0.873	1.825	15.781	0.087		0.976	0.893	−3.168	10.637	0.009	**
Function body	0.650	< 0.001	$0^{\triangle}$	17	< 0.001	***	0.699	< 0.001	$0^{\triangle}$	17	< 0.001	***
Parameter	0.922	0.143	3.706	11.721	0.003	**	0.969	0.782	−0.919	14.272	0.373	
Function call	0.907	0.077	5.316	15.696	< 0.001	***	0.933	0.223	2.957	15.797	0.009	**
Literal	0.952	0.464	−2.184	11.044	0.051		0.904	0.066	−0.330	7.265	0.751	
Index-based access	0.972	0.828	−0.684	8.269	0.513		0.941	0.302	−5.645	7.296	< 0.001	***
Loop condition	0.958	0.558	−2.464	9.703	0.034	*	0.845	0.007	$0^{\triangle}$	17	< 0.001	***
Loop body	0.935	0.240	$12^{\triangle}$	17	0.024	*	0.751	< 0.001	$0^{\triangle}$	17	< 0.001	***

editor could provide a table of contents, allowing one to quickly jump to the relevant parts without the need to scan the complete source code. In notebook-style editors, we could take this further, since documentation and code are often mixed, by automatically generating fitting headings for the steps in the process.

Another question regarding the sequence of the code and the relevance of those steps remains whether this observation depends on the Python programming language and the summarization task. In different situations, developers may have different goals when reading code, e.g., finding errors, finding new solutions, or communicating with collaborators. All of these might come with different reading strategies. Thus, the differences we observed in our study may be specific to the code summarization task and other activities will show different reading patterns. Regarding the impact of the Python language, we think its popularity makes Python a good choice for a first analysis. Furthermore, its simple syntax makes it fairly approachable and no participant severely struggled with the task. Given that it was designed as a general-purpose language and not specifically for data-driven development, some of the steps, e.g., the re-restructuring of data, can sometimes be fairly verbose without offering new insights. We argue that this might be why experts can use their experience to identify these areas as irrelevant quickly. Meanwhile, novices needed help understanding the specific syntax and spent more time with these parts without getting more valuable information. A syntax that is easier to understand or explore would be the obvious solution but needs to be embedded in the existing programming environment. Of course, *tool designers must strike a balance to ensure the interaction is sufficiently versatile, but the generated code remains understandable and ideally helps novices learn what is going on.* Furthermore, *any change in syntax should be flexible enough to facilitate learning so that novices can, over time, gain the expertise to use more compact, efficient means of data access.* Thus, UI designers should refrain from forcing novices into using certain presentations without alternatives. Beyond language and tool design, there may be similar effects due to sub-optimal framework design, where steps that could be abstracted need to be explicit in the code. A next step could be a comparison of popular libraries and how they are received and used. Particularly those that offer the same functionality across different programming languages, like TensorFlow, could yield interesting insights into the effect of the underlying language.

Another factor influenced by high prior knowledge is which code structures is of interest, where experts appear to know the important terms that allow them to quickly grasp the meaning of a code fragment, for example, via the imported libraries and the ML terminology. Meanwhile, novices need significantly more time, relying on control structures, e.g., loops, to follow what the code does. Because often Machine Learning code uses fewer of these control structures — after all the decision logic, and thus conditions are learned from the data — this new development paradigm may offer new challenges for understanding unknown code. Developers who cannot rely on the high-level structure must either understand the data, which often requires domain knowledge, or focus on specific details like known terminology, as we saw from our expert developers. However, the appropriate presentation could help reinstate high-level structures to data-driven development. Since many ML systems follow a general structure of a data pipeline, with data sources as input on the one side and a model as output, we could leverage the sequential nature for presentation: graphical programming already commonly uses blocks, which represent individual steps in the data processing, and connections, which denote the data-flow. However, the existing graphical tools (e.g., [8, 13, 43] for data-driven application development are more niche. In principle, this should also be easily possible in computational notebooks, but the experience shows that maintaining an overview of data flow in notebooks, where no execution order is enforced, can be challenging [12, 24, 25]. *Explicitly visualizing data flow will assist developers and mitigate the reliance on certain code structures for novices and support debugging for both novices and experts.*

When data defines how the software works, it is not surprising that knowledge of the data itself and the code that manipulates said data becomes more relevant. This may be why participants looked more at index-based accesses in the code. When asked what parts of the code were useful in the post-study questionnaire, participants mentioned the names of the data set. Naturally, in our study, the experts would have an advantage here, as they are typically intimately

familiar with common data sets like IRIS or MNIST, but many of the novices also voiced familiarity with those data sets, and none of them listed the data set as detrimental to their understanding. For real-world applications, however, data sets are usually more complex and require domain knowledge, so *support for data exploration remains highly relevant for understanding ML systems.* This also increases the reliance on proper documentation, which may explain the increasing popularity of more literate programming.

Another important question is how we can bridge the divide of expertise. Since the experts learned their more selective and focused reading over the years, we cannot expect novices to pick up these skills immediately. Thus, *developers will benefit from more visual guidance, e.g., via visual highlighting.* This may support novices in acquiring the selection skills of experts more quickly. Our data can be a basis for selecting which code elements need to be visually more salient, e.g., data access, and which can be more subdued or moved into configuration files, e.g., specific parameters and literals. The ability to read code with an eye for the relevant bits should also be part of programming education. Ultimately, the goal should be that truly irrelevant code is either hidden away or takes up little mental load so that novices are guided to the relevant code, and experts benefit from less clutter.

6.1 Limitations

The differences we observed in the code structure level depend on the code fragments' selection. While we chose our code examples from popular libraries and tutorials, these may not reflect the reality of software projects in the field of data-driven development. The same applies, as previously mentioned, artificially picking structurally similar code to facilitate comparability, even though the larger code landscape has known differences. A further in-depth study of code structure beyond just the grammatical construct might lead to novel insights. For example, it could tell us whether the structure we noted, with many preparatory steps at the beginning and the seemingly interesting block in the second half, is common and how we can use abstraction to reduce the necessity of parts that expert developers mostly ignore. Some participants disregarded the syntactic structures, this may be specific to our code fragments. Thus, we argue that the next step should be extending our insights by investigating real-world software projects.

Furthermore, the requirements of an experimental setup, e.g., the removal of syntax highlighting, offer a limitation as it means that the presentation deviates from how it would be done in the wild. As this is common practice for experimental setups, as described above, it allows for a reasonable comparison with prior work. Since it is consistent within the study, it likely also does not affect the comparative results, but an additional validation in a more realistic setting is desirable. Likewise, the precision of eye tracking is an important factor, as noise can easily skew the data. However, the qualitative feedback of the participants as well as the replay of the gaze tracks corroborates the differences in the quantitative data.

In addition, the comparison of expertise in these scenarios, as in our study, remains quite challenging, as identical pieces of code may over- or underwhelm developers leading to different reading patterns. While our efforts to select syntactically and semantically similar code fragments appear to be successful, since the feedback on their complexity was comparable across all participants, it limits the generalizability of our results and any interpretation should consider this context. Larger code-bases will make it unfeasible, though, to perform such an analysis as a laboratory experiment and would require a field study, which comes with its own challenges. Our experimental setup, so far, only observes these reading patterns of the code as-is. A way to validate our findings could be to adapt the presentation, e.g., highlighting or selection, or by excluding certain code constructs, to see how this affects reading behavior, allowing us to infer the importance of individual code constructs. A long-term study could also add further insights into the effect of expertise by observing how reading behavior changes over time with growing experience. Lastly, the sample of participants for our study was limited and unbalanced between novices and experts. While the results already show some statistically significant differences, a larger pool of participants would strengthen these findings.

7 Conclusion

In this paper, we investigated developers' gaze patterns and reading behavior when reading code that implements ML models. Our results show that developers with prior Machine Learning knowledge appear to fixate more on certain parts of the code, such as those sections that contain the creation and training of ML models. Novices also stick to specific code structures, although the qualitative feedback indicates that this is not always because they consider this part relevant but can also be due to its complexity and unfamiliarity. Generally experienced developers more frequently skip through the code, searching for the essential parts, while developers with less prior knowledge tend to read source code linearly, independent of the type of development paradigm. This indicates that certain parts of the code are highly relevant while others offer little value, which opens up opportunities for simplifying and streamlining this type of code. By abstracting away distracting details, data-driven development could become not just more efficient but also more accessible to a wider, less experienced audience.

Concrete solutions that make a developer's life easier can be numerous, e.g., dedicated ML IDEs (integrated development environments) that support developers' focus on the essential bits, e.g., with syntax highlighting adapted to ML-specific structures. However, even if they, theoretically, provide a benefit, there is no guarantee of their adoption if they do not match how actual developers work in the real world. Therefore, these solutions should be informed by empirical investigations such as the one presented in this paper, and their success needs to be validated similarly.

Since ML is on its way into development mainstream, now may be an ideal time to shape this area of software development and introduce development concepts that go beyond how software has been written for the last decades. We hope that this paper contributes to the informed decisions behind these changes based on empirical, human-centered evidence.

Open Science

We encourage the extension and reproduction of our work, we include the list and analysis of the code repositories as well as the eye-tracking data and code used in the user study as part of the supplementary material to this paper.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/ Software available from tensorflow.org.

[2] Matt Adereth. 2014. Silverman's Mode Estimation Method Explained. http://adereth.github.io/blog/2014/10/12/silvermans-mode-detection-method-explained/

[3] Saleema Amershi, Andrew Begel, Christian Bird, Robert DeLine, Harald C. Gall, Ece Kamar, Nachiappan Nagappan, Besmira Nushi, and Thomas Zimmermann. 2019. Software engineering for machine learning: a case study. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice* (Montreal, QC, Canada) *(ICSE (SEIP) 2019)*. IEEE / ACM, New York, NY, USA, 291–300. https://doi.org/10.1109/ICSE-SEIP.2019.00042

[4] Abdul Ali Bangash, Hareem Sahar, Shaiful Alam Chowdhury, Alexander William Wong, Abram Hindle, and Karim Ali. 2019. What do developers know about machine learning: a study of ML discussions on StackOverflow. In *Proceedings of the 16th International Conference on Mining Software Repositories* (Montreal, QC, Canada) *(MSR 2019)*. IEEE / ACM, 260–264. https://doi.org/10.1109/MSR.2019.00052

[5] Roman Bednarik. 2012. Expertise-dependent visual attention strategies develop over time during debugging with multiple code representations. *Int. J. Hum. Comput. Stud.* 70, 2 (2012), 143–155. https://doi.org/10.1016/j.ijhcs.2011.09.003

[6] Roman Bednarik and Markku Tukiainen. 2006. An eye-tracking methodology for characterizing program comprehension processes. In *Proceedings of the 2006 Symposium on Eye Tracking Research & Applications* (San Diego, California) *(ETRA '06)*. Association for Computing Machinery, New York, NY, USA, 125–132. https://doi.org/10.1145/1117309.1117356

[7] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. 2018. Neural Code Comprehension: A Learnable Representation of Code Semantics. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018* (Montreal, QC, Canada) *(NeurIPS 2018)*. 3589–3601. https://proceedings.neurips.cc/paper/2018/hash/17c3433fecc21b57000debdf7ad5c930-Abstract.html

[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] Sumon Biswas, Md Johirul Islam, Yijia Huang, and Hridesh Rajan. 2019. Boa Meets Python: A Boa Dataset of Data Science Software in Python Language. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. 577–581. https://doi.org/10.1109/MSR.2019.00086

[10] Teresa Busjahn, Roman Bednarik, Andrew Begel, Martha Crosby, James H. Paterson, Carsten Schulte, Bonita Sharif, and Sascha Tamm. 2015. Eye Movements in Code Reading: Relaxing the Linear Order. In *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension (ICPC '15)*. IEEE Computer Society, USA, 255–265. https://doi.org/10.1109/ICPC.2015.36

[11] Carrie J. Cai and Philip J. Guo. 2019. Software Developers Learning Machine Learning: Motivations, Hurdles, and Desires. In *2019 IEEE Symposium on Visual Languages and Human-Centric Computing* (Memphis, Tennessee, USA) *(VL/HCC 2019)*. IEEE Computer Society, New York, NY, USA, 25–34. https://doi.org/10.1109/VLHCC.2019.8818751

[12] Robert DeLine and Danyel Fisher. 2015. Supporting exploratory data analysis with live programming. In *2015 IEEE Symposium on Visual Languages and Human-Centric Computing* (Atlanta, GA, USA) *(VL/HCC)*. IEEE Computer Society, 111–119. https://doi.org/10.1109/VLHCC.2015.7357205

[13] Janez Demsar, Tomaz Curk, Ales Erjavec, Crtomir Gorup, Tomaz Hocevar, Mitar Milutinovic, Martin Mozina, Matija Polajnar, Marko Toplak, Anze Staric, Miha Stajdohar, Lan Umek, Lan Zagar, Jure Zbontar, Marinka Zitnik, and Blaz Zupan. 2013. Orange: data mining toolbox in python. *J. Mach. Learn. Res.* 14, 1 (2013), 2349–2353. http://dl.acm.org/citation.cfm?id=2567736

[14] Jochen Görtler, Fred Hohman, Dominik Moritz, Kanit Wongsuphasawat, Donghao Ren, Rahul Nair, Marc Kirchner, and Kayur Patel. 2022. Neo: Generalizing Confusion Matrix Visualization to Hierarchical and Multi-Output Labels. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems* (New Orleans, LA, USA) *(CHI '22)*. Association for Computing Machinery, New York, NY, USA, Article 408, 13 pages. https://doi.org/10.1145/3491102.3501823

[15] Stephen Gould. 2012. DARWIN: a framework for machine learning and computer vision research and development. *J. Mach. Learn. Res.* 13 (2012), 3533–3537. http://dl.acm.org/citation.cfm?id=2503354

[16] L. Gugerty and G. Olson. 1986. Debugging by Skilled and Novice Programmers. *SIGCHI Bull.* 17, 4 (apr 1986), 171–174. https://doi.org/10.1145/22339.22367

[17] Prateek Hejmady and N. Hari Narayanan. 2012. Visual attention patterns during program debugging with an IDE. In *Proceedings of the Symposium on Eye Tracking Research and Applications* (Santa Barbara, California) *(ETRA '12)*. Association for Computing Machinery, New York, NY, USA, 197–200. https://doi.org/10.1145/2168556.2168592

[18] Marc Hesenius, Nils Schwenzfeier, Ole Meyer, Wilhelm Koop, and Volker Gruhn. 2019. Towards a software engineering process for developing data-driven applications. In *Proceedings of the 7th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering, RAISE@ICSE 2019, Montreal, QC, Canada, May 28, 2019*. IEEE / ACM, 35–41. https://doi.org/10.1109/RAISE.2019.00014

[19] Andreas Höfer and Walter F. Tichy. 2006. Status of Empirical Research in Software Engineering. In *Empirical Software Engineering Issues. Critical Assessment and Future Directions, International Workshop, Dagstuhl Castle, Germany, June 26-30, 2006. Revised Papers (Lecture Notes in Computer Science, Vol. 4336)*. Springer, 10–19. https://doi.org/10.1007/978-3-540-71301-2_3

[20] Fred Hohman, Kanit Wongsuphasawat, Mary Beth Kery, and Kayur Patel. 2020. Understanding and Visualizing Data Iteration in Machine Learning. In *CHI '20: CHI Conference on Human Factors in Computing Systems, Honolulu, HI, USA, April 25-30, 2020*. ACM, 1–13. https://doi.org/10.1145/3313831.3376177

[21] Toyomi Ishida and Hidetake Uwano. 2019. Synchronized Analysis of Eye Movement and EEG during Program Comprehension. In *Proceedings of the 6th International Workshop on Eye Movements in Programming* (Montreal, Quebec, Canada) *(EMIP '19)*. IEEE Press, 26–32. https://doi.org/10.1109/EMIP.2019.00012

[22] Hank Kahney. 1983. What Do Novice Programmers Know about Recursion. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Boston, Massachusetts, USA) *(CHI '83)*. Association for Computing Machinery, New York, NY, USA, 235–239. https://doi.org/10.1145/800045.801618

[23] Andrej Karpathy. 2017. Software 2.0. https://karpathy.medium.com/software-2-0-a64152b37c35

[24] Mary Beth Kery, Amber Horvath, and Brad Myers. 2017. Variolite: Supporting Exploratory Programming by Data Scientists. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems* (Denver, Colorado, USA) *(CHI '17)*. Association for Computing Machinery, New York, NY, USA, 1265–1276. https://doi.org/10.1145/3025453.3025626

[25] Mary Beth Kery, Marissa Radensky, Mahima Arya, Bonnie E. John, and Brad A. Myers. 2018. The Story in the Notebook: Exploratory Data Science using a Literate Programming Tool. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems* (Montreal QC, Canada) *(CHI '18)*. Association for Computing Machinery, New York, NY, USA, 1–11. https://doi.org/10.1145/3173574.3173748

[26] 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 *UIST '20: The 33rd Annual ACM Symposium on User Interface Software and Technology, Virtual Event, USA, October 20-23, 2020*. ACM, 140–151. https://doi.org/10.1145/3379337.3415842

[27] 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* (Honolulu, HI, USA) *(CHI EA '20)*. Association for Computing Machinery, New York, NY, USA, 1–8. https://doi.org/10.1145/3334480.3383085

[28] Amy J. Ko, Htet Htet Aung, and Brad A. Myers. 2005. Design Requirements for More Flexible Structured Editors from a Study of Programmers' Text Editing. In *CHI '05 Extended Abstracts on Human Factors in Computing Systems* (Portland, OR, USA) *(CHI EA '05)*. Association for Computing Machinery, New York, NY, USA, 1557–1560. https://doi.org/10.1145/1056808.1056965

[29] André Luckow, Matthew Cook, Nathan Ashcraft, Edwin Weill, Emil Djerekarov, and Bennie Vorster. 2016. Deep learning in the automotive industry: Applications and tools. In *2016 IEEE International Conference on Big Data, BigData 2016, Washington DC, USA, December 5-8, 2016*. IEEE Computer Society, 3759–3768. https://doi.org/10.1109/BigData.2016.7841045

[30] Thomas J. McCabe. 1976. A Complexity Measure (Abstract). In *Proceedings of the 2nd International Conference on Software Engineering, San Francisco, California, USA, October 13-15, 1976*. IEEE Computer Society, 407. http://dl.acm.org/citation.cfm?id=807712

[31] Hafiz Suliman Munawar. 2020. *Flood Disaster Management*. John Wiley & Sons, Ltd, Chapter 5, 115–146. https://doi.org/10.1002/9781119682042.ch5

[32] Brad A. Myers. 1986. Visual Programming, Programming by Example, and Program Visualization: A Taxonomy. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Boston, Massachusetts, USA) *(CHI '86)*. Association for Computing Machinery, New York, NY, USA, 59–66. https://doi.org/10.1145/22627.22349

[33] Brad A. Myers, Amy J. Ko, Thomas D. LaToza, and YoungSeok Yoon. 2016. Programmers Are Users Too: Human-Centered Methods for Improving Programming Tools. *Computer* 49, 7 (2016), 44–52. https://doi.org/10.1109/MC.2016.200

[34] Jakob Nielsen. 2006. F-Shaped Pattern For Reading Web Content. https://www.nngroup.com/articles/f-shaped-pattern-reading-web-content-discovered/ Retrieved 2024–11–02.

[35] Kayur Patel. 2012. *Lowering the Barrier to Applying Machine Learning.* Ph. D. Dissertation. University of Washington, USA. https://hdl.handle.net/1773/22015

[36] Kayur Patel, James Fogarty, James A. Landay, and Beverly L. Harrison. 2008. Examining Difficulties Software Developers Encounter in the Adoption of Statistical Machine Learning. In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13-17, 2008.* AAAI Press, 1563–1566. http://www.aaai.org/Library/AAAI/2008/aaai08-263.php

[37] Patrick Peachock, Nicholas Iovino, and Bonita Sharif. 2017. Investigating eye movements in natural language and c++ source code-a replication experiment. In *International Conference on Augmented Cognition.* Springer, 206–218.

[38] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. 2011. Scikit-learn: Machine learning in Python. *the Journal of machine Learning research* 12 (2011), 2825–2830.

[39] Norman Peitek, Janet Siegmund, and Sven Apel. 2020. What Drives the Reading Order of Programmers? An Eye Tracking Study. In *Proceedings of the 28th International Conference on Program Comprehension* (Seoul, Republic of Korea) *(ICPC '20).* Association for Computing Machinery, New York, NY, USA, 342–353. https://doi.org/10.1145/3387904.3389279

[40] Alun Preece, Dan Harborne, Dave Braines, Richard Tomsett, and Supriyo Chakraborty. 2018. Stakeholders in Explainable AI.

[41] Alvin Rajkomar, Jeffrey Dean, and Isaac Kohane. 2019. Machine Learning in Medicine. *New England Journal of Medicine* 380, 14 (2019), 1347–1358. https://doi.org/10.1056/NEJMra1814259 PMID: 30943338.

[42] Mireia Ribera and Àgata Lapedriza. 2019. Can we do better explanations? A proposal of user-centered explainable AI. In *Joint Proceedings of the ACM IUI 2019 Workshops co-located with the 24th ACM Conference on Intelligent User Interfaces (ACM) IUI 2019) (CEUR Workshop Proceedings, Vol. 2327).* CEUR-WS.org. http://ceur-ws.org/Vol-2327/IUI19WS-ExSS2019-12.pdf

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

[44] Paige Rodeghero, Collin McMillan, Paul W. McBurney, Nigel Bosch, and Sidney K. D'Mello. 2014. Improving automated source code summarization via an eye-tracking study of programmers. In *36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE '14).* ACM, 390–401. https://doi.org/10.1145/2568225.2568247

[45] Timothy R. Shaffer, Jenna L. Wise, Braden M. Walters, Sebastian C. Müller, Michael Falcone, and Bonita Sharif. 2015. iTrace: enabling eye tracking on software artifacts within the IDE to support software engineering tasks. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015.* ACM, 954–957. https://doi.org/10.1145/2786805.2803188

[46] Zohreh Sharafi, Timothy Shaffer, Bonita Sharif, and Yann-Gaël Guéhéneuc. 2015. Eye-tracking metrics in software engineering. In *2015 Asia-Pacific Software Engineering Conference (APSEC).* IEEE, New York, NY, USA, 96–103.

[47] Zohreh Sharafi, Zéphyrin Soh, and Yann-Gaël Guéhéneuc. 2015. A systematic literature review on the usage of eye-tracking in software engineering. *Inf. Softw. Technol.* 67 (2015), 79–107. https://doi.org/10.1016/j.infsof.2015.06.008

[48] Kshitij Sharma, Patrick Jermann, Marc-Antoine Nüssli, and Pierre Dillenbourg. 2013. Understanding Collaborative Program Comprehension: Interlacing Gaze and Dialogues. In *10th International Conference on Computer-Supported Collaborative Learning, CSCL 2013, Conference Proceedings, Volume 1: Full Papers & Symposia, June 15-19, 2013, Madison, WI, USA.* International Society of the Learning Sciences, LuLu, Amazon, 430–437. https://repository.isls.org/handle/1/1944

[49] Helen Sharp, Jenny Preece, and Yvonne Rogers. 2019. *Interaction Design: Beyond Human-Computer Interaction.* Wiley. https://books.google.de/books?id=UDeQDwAAQBAJ

[50] Helen Shen. 2014. Interactive notebooks: Sharing the code. *Nature* 515, 7525 (Nov. 2014), 151–152.

[51] Andrew J. Simmons, Scott Barnett, Jessica Rivera-Villicana, Akshat Bajaj, and Rajesh Vasa. 2020. A large-scale comparative analysis of Coding Standard conformance in Open-Source Data Science projects. In *ESEM '20: ACM / IEEE International Symposium on Empirical Software Engineering and Measurement, Bari, Italy, October 5-7, 2020.* ACM, 1:1–1:11. https://doi.org/10.1145/3382494.3410680

[52] G. Varoquaux, L. Buitinck, G. Louppe, O. Grisel, F. Pedregosa, and A. Mueller. 2015. Scikit-Learn: Machine Learning Without Learning the Machinery. *GetMobile: Mobile Comp. and Comm.* 19, 1 (jun 2015), 29–33. https://doi.org/10.1145/2786984.2786995

[53] Thomas Weber, Christina Winiker, and Heinrich Hussmann. 2021. A Closer Look at Machine Learning Code *(CHI EA '21).* Association for Computing Machinery, New York, NY, USA, Article 338, 6 pages. https://doi.org/10.1145/3411763.3451679

[54] Niklaus Wirth. 2008. A Brief History of Software Engineering. *IEEE Annals of the History of Computing* 30, 3 (2008), 32–39. https://doi.org/10.1109/MAHC.2008.33

[55] Li Zhang, Jia-Hao Tian, Jing Jiang, Yi-Jun Liu, Meng-Yuan Pu, and Tao Yue. 2018. Empirical Research in Software Engineering - A Literature Survey. *J. Comput. Sci. Technol.* 33, 5 (2018), 876–899. https://doi.org/10.1007/s11390-018-1864-x