A Closer Look at 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

Heinrich Hußmann
LMU Munich
Munich, Germany
hussmann@ifi.lmu.de

ABSTRACT

Software using Machine Learning algorithms is becoming ever more ubiquitous making it equally important to have good development processes and practices. Whether we can apply insights from software development research remains open though, since it is not yet clear, whether data-driven development has the same requirements as its traditional counterpart. We used eye tracking to investigate whether the code reading behaviour of developers differs between code that uses Machine Learning and code that does not. Our data shows that there are differences in what parts of the code people consider of interest and how they read it. This is a consequence of differences in both syntax and semantics of the code. This reading behaviour already shows that we cannot take existing solutions as universally applicable. In the future, methods that support Machine Learning must iterate on existing knowledge to meet the challenges of data-driven development.

CCS CONCEPTS

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

KEYWORDS

machine learning, code reading, eye tracking

ACM Reference Format:
Thomas Weber, Christina Winiker, and Heinrich Hußmann. 2021. A Closer Look at Machine Learning Code. In *CHI Conference on Human Factors in Computing Systems Extended Abstracts (CHI '21 Extended Abstracts), May 8–13, 2021, Yokohama, Japan.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3411763.3451679

1 INTRODUCTION

With Machine Learning (ML) becoming a common part of everyday software, the interest in its development and the management of the necessary data volumes is growing steadily. Particularly for applications in critical domains – like healthcare [5, 16, 22, 34, 39], the automotive industry [15, 19, 25, 29] or IT security [17, 40] – it becomes important to have clear understanding and a rigorous development process for data-driven software.

CHI '21 Extended Abstracts, May 8–13, 2021, Yokohama, Japan

ACM ISBN 978-1-4503-8095-9/21/05…$15.00
https://doi.org/10.1145/3411763.3451679

While there is some research in making the user-facing parts of data-driven software more accessible, we still have a limited understanding of how data-driven differs from traditional software on the developer side, e.g. what are the best practices and processes to deal with data-driven software during development to ensure its quality. Do they differ between these development paradigms? Or are they the same?

Ideally, of course, we would want to build on the decades of research in general "traditional" software development, but it is still open whether this is a legitimate path: ML, other rather all data-driven methods, are a different development paradigm; one that requires new skills, methods and processes. A first step, therefore, must be to investigate how this paradigm compares to the more traditional way of creating software, which focuses on algorithms rather than data. Only then do we know whether the knowledge transfer from prior work makes sense.

In this paper we contribute to this process, focusing on how developers interact with their code, specifically how they read it and whether they read ML code differently than more traditional code. To this end, we conducted a preliminary user study, which explores and compares and reading behaviour of programmers.

Sect. 3 outlines our study setup for recording the programmers gaze data while reading the code and additional qualitative feedback. It also gives an overview of how we evaluated this data. We visualize the data and present specific results in Sect. 4 and highlight implications and potential applications for this information in Sect. 5.

2 RELATED WORK

While ML is around for half a century, only the advances in storage and processing power of the last decades have made it useful for a large amount of real world applications. Naturally, the complexity of the data, models, and software has increased enormously.

Many efforts are being undertaken to counter this and make ML easier to understand [2, 11, 13, 14, 18, 20, 21, 39]. While often not the main target group, the developer has been recognized as an essential stakeholder that needs support mechanisms for working with ML systems [6, 8, 28, 30].

In an effort to support ML development, academia and industry have jumped ahead and created a myriad of tools. Libraries like TensorFlow [1] or Scikit-learn [27] make it easier to set up complex ML models but often rely on low-level data management. The way they are programmed also does seemingly not differ much from traditional programming in appearance, tooling, etc., indicating how the libraries came to be. Yet, it this iteration on existing methods the correct choice or does the paradigmatic shift require new methods and tools?

Graphical development tools like RapidMiner [31] or Orange [10], by comparison, hide away a lot of the low-level programming

and offer a high-level interface for the data pipeline. They hearken back to graphical tools as used for example in Model-Driven Development [9, 35, 36]. For traditional software development, graphical tools have not found wide acclaim but remain a niche tool. It remains to be seen, whether graphical tools find greater adoption for data-driven development where the high-level view certainly has benefits.

The development of tools, however, is a fairly advanced steps, although one that frequently precedes detailed empirical analysis. Still, to best support developers it is valuable to first understand how they work and this particular case, how they work differently for data-driven software. There is some research on this comparison so far, e.g. Thung et al. [38] who applied empirical bug analysis to ML code, which before had only been used for traditional software. Sun et al. [37] on the other hand analyzed how bugs are then fixed in ML projects on the code sharing platform GitHub. Bangash et al. [7] looked at questions from the popular Q&A platform StackOverflow that deal with ML and determined that the way these questions are asked and answered is still lacking.

As some of the authors above point out though, their work is merely an iteration of similar methods being applied to traditional software development. Understanding how people work in the traditional setting has been subject of a much greater body of research. Particularly the sub-field of Code Comprehension (cf. [41] for an overview) deals with the question what contributes to the understanding of code and how to support it.

One example of this effort is the work by Rodeghero et al. [32] who used eye tracking with the goal of summarizing Java code. In their experiments, they had their participants, regular software developers, read and manually summarize different Java methods. They then used the gaze data to determine what the programmers found important during reading. Using the Vector Space Model (cf. [33]), they extracted important keywords from the Java methods in order to then automatically summarize source code to aid in code comprehension.

Ishida and Uwano [23] also highlight the importance of program understanding for efficient software development. They performed a combined eye tracking and EEG study during which programmers had to solve tasks in code, while they recorded both their eye movements and brain waves. Analyzing 80 data sets, Ishida and Uwano found a correlation of the fixation ratio and EEG activity with the comprehension of materials.

Using eye tracking in an effort to help with understanding unfamiliar code, Ahrens et al. [4] recorded gaze data, which they visualized via heatmaps to guide the readers attention to relevant code sections. While helpful for some, their results also show that too much visual information can be detrimental and distracting.

All of the examples, however, focus on "traditional" software code where the developer encodes information about the program directly in instructions. For now there appears to be a gap in the literature how all these effects play out when the program is not directly encoded but is automatically extracted from data, while the written code deals mostly with just preparing the necessary data sets. The remainder of this paper deals with our efforts to determine whether this has an effect that is detectable in gaze data.

3 METHODS

The following section describes the design of our eye tracking study, from its preparation to data collection and evaluation. Our study is as a within-subject design where we asked participants to look at ML code snippets as well as traditional Python code and to then summarize it while we tracked their eye movement.

3.1 Research Questions

Interested in how people read ML code, we formulated the following research questions:

RQ1 What parts of the ML code do participants look at most (dwell time)?

RQ2 What aspects/points of interest of the code do participants consider most relevant?

RQ3 Do programmers look at different aspects of ML code versus traditional code to understand it?

The first question is a primarily quantitative evaluation of the data and should give an indication of what parts of the code are important. RQ2 expands on this data using the total dwell time and the subjective opinions of participants from a questionnaire. The primary research interest, however, of course lies in RQ3: to determine whether ML as a new development paradigm and ML code specifically are equivalent to traditional software of whether it is treated fundamentally differently. If it were similar, we could rely on existing research while, if ML proves to be very different, the transfer of knowledge may not be as straight forward.

3.2 Study Procedure

To answer these questions we conducted a user study during which participants had to read code, ML and traditional, while we recorded their eye movement. We invited the participants to our study environment consisting of a neutral, well lit desk. We provided a computer with the study user interface (see Fig. 1), as well as a keyboard, mouse and the eye tracker mounted on the monitor.

As eye tracker we used a *tobii-92* on-screen tracker, with a sampling rate of 90 Hz, which was attached to the bottom of the screen. It was re-calibrated before each use to minimize measurement errors.

In each run, one subject spent about 45 minutes working on specific tasks, starting with the pre-questionnaire, followed by a series of the code snippet which they had to summarize, and wrapping up with the post-questionnaire. Since participants could decide when they considered a code snipped completed, the total time varied slightly.

3.3 Tasks

In order for the study participants to get familiar with the code and thus to look at it longer and more intensely, we considered tasks that they should solve while reading the code. In our study we asked participants to "describe what happens (in each) code example (and to write down what they) think the developer of this code intend it to do?" For the longest snippet, we also asked participants to write an additional doc-comment for more in-depth insights. These tasks are intended to foster engagement of the participants with the code; it also gives us an idea whether the subjects actually understood

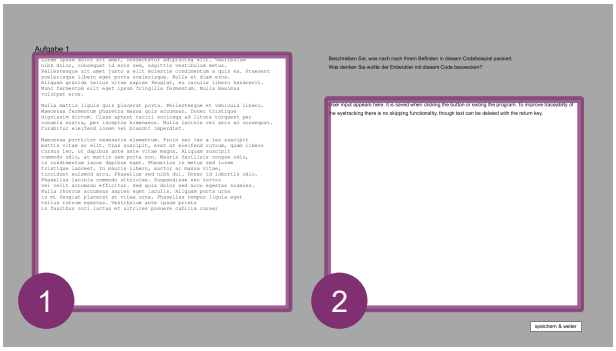

Figure 1: Screenshot of the user interface used in the study with the code on the left (1) and instructions and input field on the right (2).

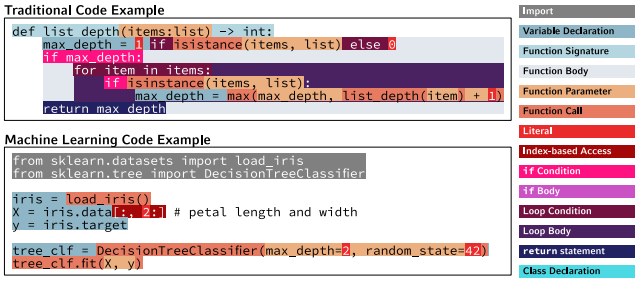

Figure 2: Two examples of the grammar-based subdivision of the code in areas of interest. In areas where the classification overlaps, the most specific type is shown.

the code snippet and what they subjectively focused on. It also acts as a simple sanity check, to verify that participants did, in fact, read the code.

3.4 Code Examples

We asked the study participants to look at eight code snippets in total, four of which are code that uses ML. The other four snippets were written as "traditional" code that does not use ML and were of similar length. Every participant looked at the same code snippets in the same order.

Both types of code we selected from GitHub[1], an online platform for sharing code, based on expert opinions. All code is written in Python, which was the natural choice, since it is a popular language for ML applications. At the same time, there are also many "traditional" applications available in Python.

Since we could not assume our participants to be ML experts, we decided to use code that is often used in ML tutorials, using the popular "Iris" [12] and MNIST [24] data set. We made sure to choose code that was short enough to be shown without scrolling to keep the eye tracking data consistent.

3.5 User Interface

Similar to Rodeghero et al. [32] we built a user interface (see Fig. 1) for the study, which had the following functionality:

- On the left side the user can see a text field, which displays the code. For the code we used a non-proportional font, as is common for code editors, but without syntax highlighting to eliminate this influence on the results.
- On the right side we placed a small text editor, allowing user to log their answers to the task directly in the interface. The instructions are displayed above.
- The interface allows participants to move through the experiment at their own pace, committing their answers and moving on to the next code snippet via a button in the bottom right.
- Overall, the interface is kept muted in tones and design so that it does not distract from the code.

[1]https://github.com

As soon as the interface is opened the eye tracker records the study participants eye movements. The task interface is able to measure the time a participant takes engage with the code. This allows us to create static information visualizations for data analysis as well as dynamic time lapses of the collected gaze data. These are particularly helpful to review how a subject read the code in real time.

The interface also had an indicator for the study conductor in case the participant is not looking at the eye tracker or the tracker failed to record allowing us to ensure the eye tracking process is working as intended.

3.6 Questionnaire

As mentioned, aside from the interface for reading the code, participants filled in a pre- and post-questionnaire.

In the pre-questionnaire we asked for demographic data, self-reporting of the general programming skills and in their previous ML experience using five-point Likert scales.

After conducting the eye tracking study, the post-questionnaire asked open questions about the code-reading during the study, particularly any highlights in the examples in terms of complexity, code structures that contribute to understanding, and differences between the ML and non-ML code.

3.7 Evaluation

To evaluate the gaze data, we used both qualitative analysis using visualizations and quantitative measures based on the formal grammar underlying the Python programming language.

The gaze data has two aspects we were interested in: spatial and temporal. The spatial distribution of gaze points, i.e. parts of the screen a person look at, we visualized using heat maps, smoothed with kernel density estimation (KDE). Fig. 3 shows an example of such a heatmap as overlay over the code. The KDE uses the *seaborn* library [42] with the *Epanechnikov* kernel and dynamically selected bandwidth via *Silverman* algorithm (cf. [3]). We considered other parameters but those above yielded a reasonable trade-off between noise and expressiveness. To visualize the temporal aspect of the data, we implemented a dynamic visualization, effectively a re-playable time-lapse of the study session.

To analyze the data more precisely, we quantified the gaze data based on which aspects of the code the participants looked at. For this purpose we divided the area of the code based on its syntax in potential areas of interest as shown in Fig. 2. To this end we used the labels from the official Python grammar specification[2] which occurred in our code snippets. Fig. 2 lists all grammar constructs we used for labeling the code snippets.

Based on this subdivision, we could gauge how long participants looked at individual parts of the code by adding up the dwell time of each rectangular area of each code construct.

4 RESULTS

At the current point in time we had six ML novices participate in our study (three male, three female; mean age: 24.5, std: 1.8). In the following we report the results this sample has yielded so far.

RQ1: Based on the replay of the gaze data, we could determine that, unsurprisingly, early parts in the code, particularly import statements and variable declarations at the beginning, frequently catch the readers eye first. Our participant's gaze behaviour also often shows patterns similar to Nielsen's F-patterns [26] for initial scanning. These patterns describe the reading behavior of an individual when looking at a given text, which resembles the letter "F". It consists of three typical parts: Participants scan vertically the left margin of the code structure. They tend to read horizontally the upper part of the code snippets first, then read shorter horizontal paths of the code in the middle area.

RQ2: Regarding the code structures look at most, there is no entirely clear picture but rather distinct groups of reading behaviour: some participants stuck to the F-patterns, therefore focusing mostly on the beginning of line. Other participants, particularly those with some self-reported ML experience, skipped ahead and focused on later parts, especially index-based list access and specific literals and function names that are common in ML. Fig. 3 shows an examples of this with two of the participants showing reading patterns resembling the F-structure while the other two focused the majority of their attention on specific literals. This generally matches with the subjective opinions which aspects of the code people consider important, as reported in the questionnaire. Here participants stated that they usually read the code top to bottom, thus focusing more on early parts. They also reported, though, that they sometimes skip over entire sections if they consider it less relevant, particularly the imports at the top. Instead the participants report to focus function parameters, and familiar function names. Function and variable names but also indentation were also named as important factors for code comprehension.

For a quantitative view of which parts of the code might have been important, Fig. 4 shows an overview of all code elements and their proportional viewing frequency. Since the data is normalized to eliminate the fact that different code constructs occupy a different amount of space, a value of one would indicate that the percentage of the participants gaze on a piece of code is proportional to the are it occupies. A value greater or less than one therefore indicate that the code was looked at disproportionally long of short respectively.

[2]https://docs.python.org/3/reference/grammar.html

The gaze data therefore suggests that the index-based access, often used for slicing data sets, is considered fairly important, as are variable declarations and loops.

RQ3: The proportional viewing frequency also allows us to draw a comparison between traditional Python code and code that uses ML. As the Fig. 4 shows, there are some similarities and differences between how our participants read the two types of code: The relative dwell time for a number of code constructs, like literals and variable declarations, is very similar for both types of code. For others, the difference, however, is very stark: loops, signatures and calls receive a fair amount of attention for ML but much more so for traditional code. With the high variance across our participants, it is unclear though, whether this is just an effect of personal preference or a real difference.

Further noteworthy are the function parameters, conditions and index-based access, since they show a discrepancy where in one type of code they are viewed very frequently, while in the other, they are much less in the focus.

The apparent difference for class declarations and return statements is no real effect but a result of the fact that the ML code did not contain either.

5 DISCUSSION

While some of our results outlined above certainly are not surprising – for example, the mixture of F-pattern type and skip-ahead reading in both types of code simply lies in the nature of code – the following section will discuss interesting insights in our data as well as some limitations and future changes that may lead to clearer results.

The first fruitful finding are those aspects of ML and traditional code that are not read with equal importance. Even in our limited code examples, we could observe these differences but while these differences should be further corroborated with more code examples, they share a similarity which points to a potential reason: in ML code index-based operations on data sets are considered more important, while control flow like conditional branching received relative less attention. Since in ML the complexity of the resulting program is encoded or hidden in the data, control flow in the code serves only an auxiliary purpose and does not affect the program as much as it does in traditional code. Therefore understanding how the data is handled might appear more relevant than the order in which the written code is executed. One participant did also note that she uses indentation to get an idea of the code at first glance. With less control flow constructs in ML code and less attention spent on them, different methods for visually structuring the code may be of interest.

Another point of interest is the relative little interest in dependencies and imports. While in traditional code they often serve to make auxiliary functionality accessible. In ML code, where particularly the ML models are rarely re-implemented but rather imported, they offer important insights which models, what data sets, etc. are used. It is very likely though that the similarity in gaze we see in our data is a result of habitation, where developers have gotten into the habit of skipping the import section. This behaviour is even encouraged when tools deliberately hide or fold imports. A future analysis of real world examples of ML code and a comparison of

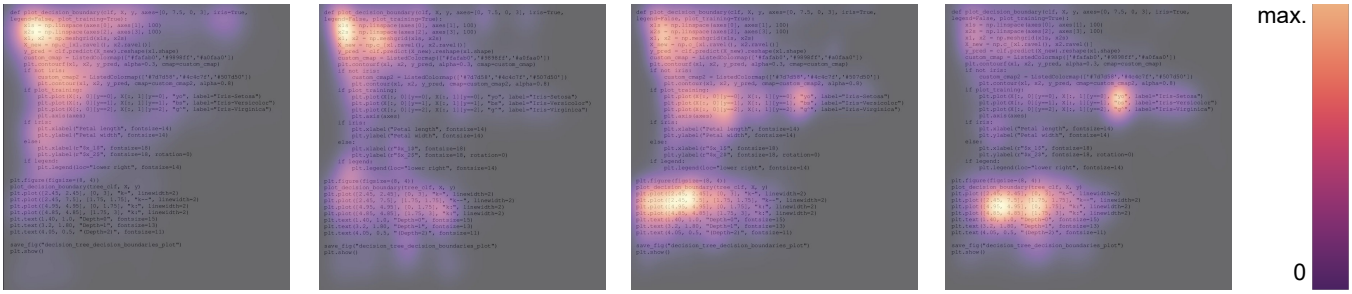

Figure 3: Gaze patterns of four participants for a ML code snippet. The two examples on the left with no prior ML knowledge show loose F-patterns, while the two more knowledgeable participants on the right focus on specific later parts of the code.

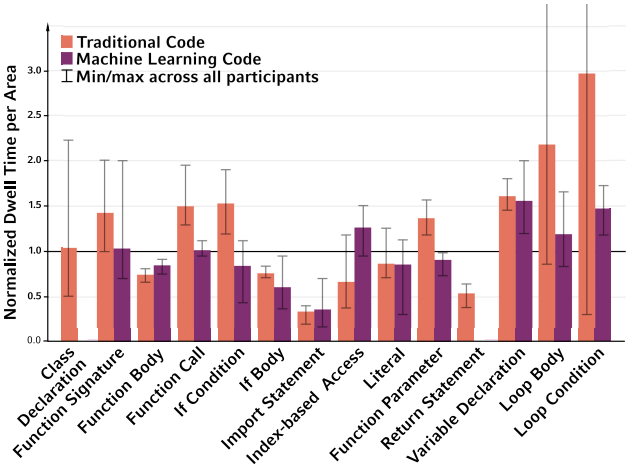

Figure 4: Relative dwell time per structural code element. The data is normalized by the relative area to accommodate for differences between traditional and ML code examples.

how much information developers can extract from code with or without import statements therefore seems worthwhile.

Regarding the code structure, while we did account for difference in code structure by normalizing the results, they are still dependant on the code examples we chose. IRIS an MNIST may be typical tutorial examples but are likely very different from real world examples. Ideally this study should be conducted with code that is representative of the two classes of code, but finding such examples is nearly impossible given the breadth and variety of code, different coding styles, etc. Furthermore, we would want both groups, ML and traditional, to have a roughly similar frequency of syntax elements to maintain comparability. Using static analysis of large amounts of code, e.g. GitHub repositories, could help in determining typical features of ML code, which would allow us to create more representative examples while at the same time further investigating whether and how ML code differs from the traditional variety. At the same time, even just ML code can be very diverse with different types of models, libraries, etc. So a selection of code will always only represent a certain subset of the whole development paradigm and it remains to be investigated how well a selection can represent the whole paradigm.

Lastly, the above results are collected from six participants, none of which are professional data analysts or ML developers. In order to yield more reliable results, our study needs to be continues with, ideally, industry professionals and ML experts. Nonetheless, results from novices have value too, since this is a target group the benefits especially from good support mechanisms, while long-time experts would have had a lengthy learning period and would have gotten used to certain practices.

6 CONCLUSION

With this paper we contribute to the effort of determining the differences and similarities between ML and traditional code. In our experiment we collected gaze data and subjective feedback for both ML and traditional code, allowing us to compare those two via visualization and quantitative analysis.

In this data we found that the reading patterns for both types of code are similar with both F-pattern scanning and deliberate skips. Good naming of variables and functions is also important for both types of code. Dependencies beyond the current code, usually in the form of imports, are skipped deliberately in both cases even though they may bear important information especially for ML.

The gaze data also revealed some differences though: for traditional code structural information appears to be important, e.g. loop- and conditional blocks and functions. For ML code these code constructs are less important, so participants infer a lot of semantics from variable and function names like "train" and instead look more at how the data sets are transformed.

This different focus may be taken into account in future research for supporting ML developers via different highlighting in existing tools, by new tools that abstract away irrelevant parts of the code etc. Whether data-driven development will eventually just merge into general software development or remain distinct is still open. Our results so far, however, show that we should not simply assume that we can just apply our knowledge of the one to the other but need to take into account the specific requirements and challenges of the data-driven paradigm.

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] Amina Adadi and Mohammed Berrada. 2018. Peeking Inside the Black-Box: A Survey on Explainable Artificial Intelligence (XAI). *IEEE Access* 6 (2018), 52138–52160. https://doi.org/10.1109/ACCESS.2018.2870052

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

[4] Maike Ahrens, Kurt Schneider, and Melanie Busch. 2019. Attention in software maintenance: an eye tracking study. In *Proceedings of the 6th International Workshop on Eye Movements in Programming, EMIP@ICSE 2019, Montreal, Quebec, Canada, May 27, 2019*, Andrew Begel and Janet Siegmund (Eds.). IEEE / ACM, 2–9. https://doi.org/10.1109/EMIP.2019.00009

[5] Russ B. Altman. 1999. AI in Medicine: The Spectrum of Challenges from Managed Care to Molecular Medicine. *AI Magazine* 20, 3 (Sep. 1999), 67. https://doi.org/10.1609/aimag.v20i3.1467

[6] 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, ICSE (SEIP) 2019, Montreal, QC, Canada, May 25-31, 2019*, Helen Sharp and Mike Whalen (Eds.). IEEE / ACM, 291–300. https://doi.org/10.1109/ICSE-SEIP.2019.00042

[7] 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, MSR 2019, 26-27 May 2019, Montreal, Canada*, Margaret-Anne D. Storey, Bram Adams, and Sonia Haiduc (Eds.). IEEE / ACM, 260–264. https://doi.org/10.1109/MSR.2019.00052

[8] 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, VL/HCC 2019, Memphis, Tennessee, USA, October 14-18, 2019*, Justin Smith, Christopher Bogart, Judith Good, and Scott D. Fleming (Eds.). IEEE Computer Society, 25–34. https://doi.org/10.1109/VLHCC.2019.8818751

[9] Simone Di Cola, Cuong M. Tran, and Kung-Kiu Lau. 2015. A Graphical Tool for Model-Driven Development Using Components and Services. In *41st Euromicro Conference on Software Engineering and Advanced Applications, EUROMICRO-SEAA 2015, Madeira, Portugal, August 26-28, 2015*. IEEE Computer Society, 181–182. https://doi.org/10.1109/SEAA.2015.13

[10] 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

[11] Mengnan Du, Ninghao Liu, and Xia Hu. 2019. Techniques for Interpretable Machine Learning. *Commun. ACM* 63, 1 (Dec. 2019), 68–77. https://doi.org/10.1145/3359786

[12] Ronald A Fisher and Michael Marshall. 1936. Iris data set. *RA Fisher, UC Irvine Machine Learning Repository* 440 (1936), 87.

[13] Randy Goebel, Ajay Chander, Katharina Holzinger, Freddy Lecue, Zeynep Akata, Simone Stumpf, Peter Kieseberg, and Andreas Holzinger. 2018. Explainable AI: The New 42?. In *Machine Learning and Knowledge Extraction*, Andreas Holzinger, Peter Kieseberg, A Min Tjoa, and Edgar Weippl (Eds.). Springer International Publishing, Cham, 295–303.

[14] David Gunning, Mark Stefik, Jaesik Choi, Timothy Miller, Simone Stumpf, and Guang-Zhong Yang. 2019. XAI—Explainable artificial intelligence. *Science Robotics* 4, 37 (2019). https://doi.org/10.1126/scirobotics.aay7120

[15] Oleg Yu. Gusikhin, Nestor Rychtyckyj, and Dimitar P. Filev. 2007. Intelligent systems in the automotive industry: applications and trends. *Knowl. Inf. Syst.* 12, 2 (2007), 147–168. https://doi.org/10.1007/s10115-006-0063-1

[16] Pavel Hamet and Johanne Tremblay. 2017. Artificial intelligence in medicine. *Metabolism* 69 (2017), S36 – S40. https://doi.org/10.1016/j.metabol.2017.01.011 Insights Into the Future of Medicine: Technologies, Concepts, and Integration.

[17] Nutan Farah Haq, Abdur Rahman Onik, Md. Avishek Khan Hridoy, Musharrat Rafni, Faisal Muhammad Shah, and Dewan Md. Farid. 2015. Application of Machine Learning Approaches in Intrusion Detection System: A Survey. *International Journal of Advanced Research in Artificial Intelligence* 4, 3 (2015). https://doi.org/10.14569/IJARAI.2015.040302

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

[19] Martin Hofmann, Florian Neukart, and Thomas Bäck. 2017. Artificial Intelligence and Data Science in the Automotive Industry. *CoRR* abs/1709.01989 (2017). http://arxiv.org/abs/1709.01989

[20] A. Holzinger. 2018. From Machine Learning to Explainable AI. In *2018 World Symposium on Digital Intelligence for Systems and Machines (DISA)*. 55–66. https://doi.org/10.1109/DISA.2018.8490530

[21] Andreas Holzinger, Chris Biemann, Constantinos S. Pattichis, and Douglas B. Kell. 2017. What do we need to build explainable AI systems for the medical domain? *CoRR* abs/1712.09923 (2017). http://arxiv.org/abs/1712.09923

[22] Werner Horn. 2001. AI in medicine on its way from knowledge-intensive to data-intensive systems. *Artificial Intelligence in Medicine* 23, 1 (2001), 5 – 12. https://doi.org/10.1016/S0933-3657(01)00072-0

[23] 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

[24] Yann LeCun, Corinna Cortes, and Christopher J.C. Burges. 1998. The MNIST Database of Handwritten Digits.

[25] Jun Li, Hong Cheng, Hongliang Guo, and Shaobo Qiu. 2018. Survey on Artificial Intelligence for Vehicles. *Automotive Innovation* 1, 1 (01 Jan 2018), 2–14. https://doi.org/10.1007/s42154-018-0009-9

[26] Jakob Nielsen. 2006. F-Shaped Pattern For Reading Web Content. https://www.nngroup.com/articles/f-shaped-pattern-reading-web-content-discovered/ Retrieved April 25, 2021.

[27] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[28] Alun Preece, Dan Harborne, Dave Braines, Richard Tomsett, and Supriyo Chakraborty. 2018. Stakeholders in Explainable AI. arXiv:1810.00184 [cs.AI]

[29] Danil V. Prokhorov (Ed.). 2008. *Computational Intelligence in Automotive Applications*. Studies in Computational Intelligence, Vol. 132. Springer.

[30] 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), Los Angeles, USA, March 20, 2019 (CEUR Workshop Proceedings, Vol. 2327)*, Christoph Trattner, Denis Parra, and Nathalie Riche (Eds.). CEUR-WS.org. http://ceur-ws.org/Vol-2327/IUI19WS-ExSS2019-12.pdf

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

[32] Paige Rodeghero, Collin McMillan, Paul W. McBurney, Nigel Bosch, and Sidney D'Mello. 2014. Improving Automated Source Code Summarization via an Eye-Tracking Study of Programmers. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 390–401. https://doi.org/10.1145/2568225.2568247

[33] G. Salton, A. Wong, and C. S. Yang. 1975. A Vector Space Model for Automatic Indexing. *Commun. ACM* 18, 11 (1975), 613–620. https://doi.org/10.1145/361219.361220

[34] Edward H. Shortliffe. 1993. The adolescence of AI in Medicine: Will the field come of age in the '90s? *Artificial Intelligence in Medicine* 5, 2 (1993), 93 – 106. https://doi.org/10.1016/0933-3657(93)90011-Q Artificial Intelligence in Medicine: State-of-the-Art and Future Prospects.

[35] Geoffrey Sparks. 2012. Enterprise Architect User Guide.

[36] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. 2009. *EMF: Eclipse Modeling Framework* (2. ed.). Addison-Wesley, Boston, MA. http://proquestcombo.safaribooksonline.com/9780321331885

[37] X. Sun, T. Zhou, G. Li, J. Hu, H. Yang, and B. Li. 2017. An Empirical Study on Real Bugs for Machine Learning Programs. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*. 348–357. https://doi.org/10.1109/APSEC.2017.41

[38] Ferdian Thung, Shaowei Wang, David Lo, and Lingxiao Jiang. 2012. An Empirical Study of Bugs in Machine Learning Systems. In *23rd IEEE International Symposium on Software Reliability Engineering, ISSRE 2012, Dallas, TX, USA, November 27-30, 2012*. IEEE Computer Society, 271–280. https://doi.org/10.1109/ISSRE.2012.22

[39] E. Tjoa and C. Guan. 2020. A Survey on Explainable Artificial Intelligence (XAI): Toward Medical XAI. *IEEE Transactions on Neural Networks and Learning Systems* (2020), 1–21. https://doi.org/10.1109/TNNLS.2020.3027314

[40] Daniele Ucci, Leonardo Aniello, and Roberto Baldoni. 2019. Survey of machine learning techniques for malware analysis. *Computers & Security* 81 (2019), 123 – 147. https://doi.org/10.1016/j.cose.2018.11.001

[41] A. Von Mayrhauser and A. M. Vans. 1995. Program comprehension during software maintenance and evolution. *Computer* 28, 8 (1995), 44–55. https://doi.org/10.1109/2.402076

[42] Michael Waskom. 2020. Visualizing the distribution of a dataset. https://seaborn.pydata.org/tutorial/distributions.html. Retrieved April 25, 2021.