Hands for AI

Programming in Machine Learning as labor, work, and action

Johanna Fischer

> "These algorithmic systems are not standalone little boxes, but massive, networked ones with hundreds of hands reaching into them, tweaking and tuning, swapping parts and experimenting with new arrangements." (Seaver 2019, 419)

Introduction

This chapter sheds light on software programming in Machine Learning (ML). ML, evolved as a subfield of artificial intelligence (AI) and concentrates on the development of self-learning algorithms. These algorithms extract insights from data in order to make statistical predictions (Raschka and Mirjalili 2017, 25). Algorithms are a key technology in digitalization, a process of constant translation, combination, and use of societies' representations in digital technologies (Häußling 2020, 1,358). The translation process from data into algorithms is rooted in data scientists' everyday routines. I delineate the practices involved in producing ML algorithms and describe their cultural and material aspects through ethnography in the field of data science.

My perspective to study programming integrates data science, sociology, and design, and stems from years of experience in applied research projects in which I worked across disciplinary boundaries. Positioned at the interfaces of these fields, my ethnographic approach captures the moments at which translations occur: between domains, between technical and social phenomena, and between different stages of development. This vantage point has allowed me to trace how knowledge, tools, and responsibilities shift across domains, thereby making the otherwise implicit negotiations that shape programming practices visible. This contribution is the result of my ethnographic observations in the field of ML.

The chapter is built on the concepts of *labor*, *work*, and *action* as Hannah Arendt has described them in *The Human Condition* (1958). The use of labor, work, and action are fruitful because they conceptualize the practices of data scientists in distinct categories, thereby organizing observed phenomena in clear concepts. While Arendt's framework, dating back to 1958, does not address digitality, it nev-

ertheless provides a valuable perspective on technologies through which to examine the relationship between human agencies and automated systems. Others have discussed Arendt's concept for new work forms, such as contingent work (Fayard 2021). Arendt's thought helps to articulate the stakes of human involvement beyond mere functionality or efficiency in programming environments in which the boundary between tool use and world-making is often blurred. Her framework, thus, speaks to our present by making visible how digital infrastructures participate in reshaping the conditions of human action, responsibility, and judgment in the public realm. This chapter focuses on highly skilled workers whose objectives lie in building automated systems.

Furthermore, I draw from current perspectives in Science and Technology Studies (STS). STS scholars aim to document science, technology, and societies as co-constituted, given that it is an intersectional studies program with a strong influence from humanities, social sciences, and information sciences. The parenthesis connecting these distinct fields is the conviction that knowledge does not preexist, and thus cannot be determined by, logical empiricism, only by collective arrangements (Jasanoff 2004, 261; Hilgartner, Miller, and Hagendijk 2015; Law 2017, 33). There is an entire research array dedicated to algorithms in STS from which I will draw in order to highlight the sociotechnical aspects of programming in ML. In the context of programming for machine learning, I engage with studies that ethnographically trace how algorithms are developed, stabilized, and interpreted. (e.g., Amoore and Piotukh 2016; Neyland 2019; Seaver 2019; Vertesi et al. 2019; Jaton 2020). Recent scholarship on algorithms emphasizes their cultural and processual dimensions, challenging perceptions of them as autonomous or disruptive forces. Nick Seaver (2022) highlights how algorithmic systems reflect the cultural practices and interpretations of their creators, framing them as human-made cultural phenomena, rather than external intruders. Similarly, Florian Jaton (2021) demonstrates the contingent, labor-intensive nature of algorithmic constitution, revealing how the illusion of seamless automation arises from concealed human effort. These insights align with Arendt's distinctions between labor, work, and action and offer a framework to further theorize algorithms as both technical artifacts and as expressions of human culture. This chapter extends the understanding of algorithms as embedded in human practice by integrating Arendt's concepts.

Programming is defined broadly as "[…] the situated activity of inscribing numbered lists of instructions that can be executed by computer processors to organize the movement of bits and to modify given data in desired ways" (Jaton 2020, 93). Regarding ML, the act of inscribing these numbered lists is partly automated by learning algorithms. In addition, inscribing code is only a small part of data scientists' practices. Surprisingly, it is only this small productive part that receives attention as *the* practice of programming in the general sense. I argue that there is more to programming than the inscription of numbered lists of instructions. Following Alex

Preda's discussion on coding and expertise (2024), this chapter takes a wide perception on what belongs to the making of ML algorithms. Thus, the argumentation is built upon descriptions of the situatedness of labor, work, and action, the relationship of these concepts, and their role in the production of ML algorithms.

The text is structured into four parts: The first part is a short excursion into computing infrastructures, programming cultures, and ML algorithms.

In the following I expand on the concepts of labor, work, and action for practices in ML programming. In the second part, I focus on Labor (Arbeit). Despite automation being a focus of data scientists, human labor is also very important. Labor can be seen in initializing, maintaining, and repairing the material infrastructures for ML algorithms to run; it involves the repetitious and tedious practice of programmer's routines. The third part focuses on code and curated datasets as the result of work (Herstellen). The code will eventually – executed by a computer – produce an ML algorithm in combination with data. Work can be observed in writing, reading, testing, structuring, combining, and repairing software. By doing work, the programmer orchestrates and instructs automated labor-doing machines – she[1] is setting the structure by which a machine is *learning* rules and patterns from data. The fourth part describes the practices necessary to establishing epistemic infrastructures through action (Handeln). The programmer in ML is engaging in creative, informal, and communicative ways to produce, access, and distribute knowledge both before and after labor and work. These ways can be distinguished into two directions: Firstly, in the interaction with data as playful explorations[2] to gain insights and form new knowledge about the data's characteristics. Secondly, in interaction with and in exchanges with other programmers.

Programmers collaborate in vast networks to exchange ideas and to learn new methods; to draw concepts and models for the work ahead with colleagues; and to both communicate and narrate results to the public, thereby embedding them in the epistemic infrastructure. In each part, I draw from my ethnographic experiences in the field of Data Science and will bring my observations into dialogue with Arendt's framework, culminating in a reflection of the results in the final part.

1 To counteract the common notion of developers as male, but defining myself as female, I will refer to programmers with the pronouns "she/her." For the Interviewees and my observations, I will use the preferred genders of my colleagues and interview partners.

2 The professional term for this is Exploratory Data Analysis (EDA). This practice lies between action and work, given that EDA is often accompanied by cleaning and structuring the data. I decided to focus on this practice's exploratory aspect here and to focus on its role in knowledge creation.

Excurse: infrastructures, machines and programmers' self-perception

Material and epistemic infrastructures

Susan Leigh Star suggests that we think of computers not as "information highways," but more modestly as "symbolic sewers" (Star 1999, 379). This notion helps to overcome the perception of computer infrastructures as impeccable systems. Instead, it opens observations up to a messy structure that is established by entangling partly compatible material from computers, cables, and plugs. This structure is characterized by ill-fitted interfaces and routinely small disruptions. Repair becomes a constant task to keep the system running (Graham and Thrift 2007, 19). "The organization of this work[3] of maintenance and reproduction is central to developments in ML as well as in its application" (Pardo-Guerra 2024, 380).

Two forms of infrastructures are relevant for ML: material and epistemic. Infrastructures are not static, but are instead processual structures (Schabacher 2022, 323). Infrastructures get highlighted through the labor practices of maintenance, reproduction, and repair as well as the action of both speaking and interacting. Both infrastructural forms have the objective of distribution: resources for the material or knowledge for the epistemic infrastructure.

Technological functionality depends on material infrastructures. Sara Hooker (2021) described it as a hardware lottery. This means successful technologies just happen to be "compatible with available software and hardware" (Hooker 2020, 1).

ML and its material contents are organized and embedded in hardware, such as server space and computing powers. Prerequisites for ML's operation are – simply put – electronic computer circuits for massive calculations, organized in cultural and material structures.

The epistemic infrastructure enables collaboration that is a prerequisite for programming. Collaboration is something that is deeply embedded in software's multiple aspects. Comments are inscribed residues for collective creation. However, exchange about the product goes far beyond comment lines in the code. As explicit documentation of a globally connected software manufacturing world, there are libraries to exchange tools and either chat rooms or small-scale manuals for almost every question about production.

"[L]ike other textual practices of knowledge creation, coding and creation of new ML applications are forms of bricolage that use existing resources in order to produce novel combinations." (Bergström and Blackwell 2016 in Pardo-Guerra 2024).

3 While Pardo-Guerra calls it work, I would suggest considering the practice of maintenance and reproduction as labor, in Arendt's sense, because it helps to take care of a system in which producing work can enfold.

To conclude, infrastructures – material or knowledge – are evolving constructions that need constant maintenance.

Cultures of anthropomorphism

I showed my colleague from an ML project a revised dataset that had been meticulously prepared to be filtered by topics in a collection of 70,000 speeches. Each speech had been labeled with one of 24 possible themes. We aimed to train a model to predict the most prominent topics for a given composition of speakers in the German Federal Parliament. My colleague, sounding surprised, exclaims, "Oh my God! You didn't do that yourself, did you?" His tone was one of shock. "Did you build a tool for this?" (fieldnote from a data science workshop in 2025, J.F.).

His reaction clearly highlighted the view that, for him, repetitive and tedious tasks are something that the computer should take over. In addition, I observed that tutorials often emphasized how cleanly the machine works, never tiring and executing the code script identically over hundreds of iterations. However, the limitations of a code script and the lack of interpretative ability of a machine solution, as well as the hidden labor that either fades into the background or is outsourced, are frequently overlooked.

The history of computers is marked by the fact that intensive repetitive work was outsourced and hidden. For instance, one of the early computer experts, John von Neumann, did not acknowledge the work of women referred to as *operators*[4] on the ENIAC computing machine in his often-cited *First Draft* from 1945, which laid the foundation for the development of subsequent computer models (von Neumann 1993). The female operators' work was seen as involving merely the mechanical implementation of previously conceived concepts for the functioning of the computing machine. However, recent reviews of historical records show how important these operators' contributions were. They not only translated and corrected the blueprints of complex hardware architecture, but they also assembled and monitored the machines in collaboration with colleagues, created the punch card systems, and fed the machine with new inputs (Chun 2013, 29–34; Haigh, Priestley, and Rope 2016, 132–140). In other words, the operators became an invisible part of the system in this process.

The code script, as the ingenious product from the programmer, has received the great deal of attention since von Neumann's *First Draft*. Von Neumann's perception of the computer persists: "Whereas computing systems were, in practice, sociotechnical *processes* that could ultimately—perhaps—produce meaningful results, the formalism of the *First Draft* surreptitiously presented them as brain-like *objects*

4 Von Neumann did not credit many people involved in ENIAC's construction in his swiftly drafted paper. This led to several patent problems and lawsuits.

that could automatically transform inputs into outputs [emphasis in original]" (Jaton 2020, 102). The assumption that the computer is a tool, comparable to a human mind, persists. A wide spread concept enfolds from this angle: with the right software, electronic minds might perform any task a human brain does and would, thus, be able to automate any human labor sooner or later.

This perception still shapes programmers' self-perception and the culture of how they think about automation; however, the code as the product of their work is just the technical artefact that transfers rules for labor from humans to machines. There are more practices that belong to programming with regard to knowledge and hardware. Defining, structuring, and translating rules needs much more than the transcriptions of a single individual. This can also be found in coding expertise. Alex Preda describes "coding expertise as bundles of skilled, both discursive and nondiscursive activities" (Preda 2024, 718).

We can take from the historical example the legacy of the cultural background that is part of programmers' practices in labor: The programmer's self-perception is influenced by the misconception of the computer as an electronical brain that is run by the programmers' law of code. The comparison of human cognition and computerized models is leading to the conclusion that almost all human cognitive activities can be automated. This results in ongoing attempts to automate increasing numbers of tasks through computed instruments. The analogy of the computer as an electronical brain is seductive, but it is also dangerous; it forms part of a larger ideological project that anthropomorphizes computation, blurring the distinction between automation and cognition in the process. This conflation matters: Wendy Chun even sees in this approach a sign of neo-liberal governmentality by building a stronger trust in technically computed outcomes than those achieved by human cognition and debate (Chun 2013, 10).

A machine learning model does not infer, unlike a human brain. It optimizes a loss function over data representations that are encoded in ways it does not *understand*. It also produces a statistically likely mapping. To illustrate this: an algorithm for movie recommendations will simply classify by meta data. It does not even need to understand the concept of a movie. It outputs recommendations based on the data that it has learned from and follows the goal to continue human-machine interactions. This exemplifies that machines' results are not directed toward the liberation of living organisms from labor, but instead to the monotonous processing of data according to pre-learned rules. In a certain sense, the machine performs labor (the movie recommendation), but the machine is no longer oriented towards humans' reproduction, but rather to the reproduction of the human-machine interaction. Their objectives are derived from the production process and according to the logic of the companies: Cost and time efficiency, creating awareness for advertising or reproducing and expanding the ML environment.

Interestingly, programmers aim to transfer such tasks to others without recognizing the newly emerging precariat among so-called *click-workers* like Amazon's *mechanical turks* not only through software, but also through outsourcing (Jaton 2020, 66; Fayard 2021, 207). This is a sign that for programmers the emancipation from labor means handing over tedious and repetitive tasks and making the arduous labor behind the tasks disappear from their own perception: This happens both through outsourcing and automation. Fayard describes and explains this as the unequal distribution of power in an economic system: "Similarly, one can view on-demand mobile workers as standing in for the human infrastructure that supports the creative or leisure activities of a segment of the population. Work (as defined by Arendt) is not for all; it can exist only because of the labor of others" (Fayard 2021, 215).

The computer, as a machine that is capable of taking over mathematical and rule-based tasks, leads to a strong separation into a) labor that can be automated, and b) labor performed by humans. This creates a field of tension for programming in ML: tension between the urge to free oneself of labor, so as to enable (creative) work, while managing the labor necessary to reproducing their own field. Part of the work is, thus, directed at itself: By using automated technologies, programmers free themselves from the burden of repetitive tasks, or at least that is a prominent narrative in the programmer's domain (Ramirez 2020, 99; Munn 2022, 17).

In summary, the reproductive parts (initializing, maintaining, and repairing the material infrastructures) of programming are often overlooked, made invisible, outsourced, and automated whenever possible. Programmers' perception of their own work lies in the architecture and rules written in code that are handed over to machines. Thus, cultural and material components are constantly overlooked and underestimated. These aspects have only regained entry into the theoretical consideration of programming practices with the practical turn in Human Computer Interaction Studies (HCI) and an increasing number of studies on programming in STS (Bergström and Blackwell 2016; Seaver 2019; Jaton 2020). This shift has not only reframed algorithms as sociotechnical processes, but have also fostered greater self-awareness among programmers, who begin by recognizing their own role in shaping algorithmic systems as cultural artifacts.

Labor: maintaining ML environments

Arendt defines labor (Arbeit) as follows: "Labor is the activity which corresponds to the biological process of the human body, whose spontaneous growth, metabolism, and eventually decay are bound to the vital necessities produced and fed into the life process by labor. The human condition of labor is life itself" (Arendt 1958, 7). Labor is the basis of life and ensures the individual's survival and, thus, the preservation of the species (ibid., 15). Labor points towards the fragility of the material world (Scha-

bacher 2022, 286), where "decay and the vitality of matter" are the most prevalent subject for humans and things (Denis and Pontille 2015, 355). The repair and maintenance of things as labor practices are open and have no assigned ending. They are undertaken by constant "acts of perceptual and affective attention" towards the objects being maintained (Jackson 2016, 183).

Arendt had already recognized that technological progress is being used to advance the automation of labor and to fulfill humanity's dream of an effortless life. According to her, the individual is striving to be freed from the perceived burden of necessary labor. However, there is a danger in this for modern working society, where the activity of the working individual has been glorified, as Max Weber's notion of a mechanical base in form of a professional duty (Berufspflicht) illustrates in the Protestant Ethic (Weber 2010, 179). Capitalistic societies know no other form of occupation, and only a few super-rich or artistically engaged people would understand themselves as being outside of labor (Arendt 1994, 11).

Today's societies are highly differentiated through the increase of work, due to the division of labor, its distribution, and also the transfer of these activities, namely as jobs, in the public sphere (ibid., 48). As a result, tasks can no longer be directly related to the survival of the individual carrying them out. Monotonous routines and one-dimensional activities arise because there is a division of labor. The tasks get distributed in the public, as the practices being carried out are fragmented. This isolates practitioners from their context of effects. The fragmentation of practices, thus, is the background for societies with a high automation of labor. It establishes the environment for automation that was to be expanded.

Labor's characteristic is that it does not produce a durable product; what it achieves is immediately consumed and utilized, thereby freeing up capacities for other activities (ibid., 81). "[…] *[L]aboring* always moves in the same circle, which is prescribed by the biological process of the living organism and the end of its *toil* and *trouble* comes only with the death of this organism. [emphasis in original]" (Arendt 1958, 90).

In highly technological societies, labor produces consumable goods that are abstracted from the biological needs of the individual. Labor in ML is the maintenance for the technological environment in which ML algorithms can be produced.

Work: producing code, structuring datasets

The objects that result from work (Herstellen) are not consumed or used up; they have durability and, as such, they are used in a world of things. Usage wears material things out in such a world. For digital objects, though, there is no sign of use because they perfectly reproduce themselves. A song will not lose quality when copied and a text does not wrinkle when read multiple times. Here, manufactured

objects, namely data and code are, in a sense, absolute; only their interweaving with other objects, such as hardware and the constant progress and change in development, formats, storage, and processing power conditions their continual decay. Therefore, regardless of its reproducibility, software has a very short half-life as a tool because its functionality is highly dependent upon the environment in which it operates (Hooker 2020, 3).

Arendt's view on tools for manufacturing was that "[t]he process of making a thing is limited and the function of the instruments comes to an predictable, controllable end with the finished product" (Arendt 1958, 122). With continual change on the sides of the instruments and the constant decay of software, though, the manufacturing work is bound to constant labor, shaping the environment in which this object performs.

In Arendt's consideration, production is preceded by a mental model. The concept of the product and the manufacturing concepts are prerequisites for their production (Arendt 1994, 129). In STS, it is agreed that these are not just mental ideas generated by the programmers, but rather include shared knowledge between the manufacturers and explicit ideas that circulate through inscriptions. With ML models, they are sometimes so abstract that an explicable idea of the final product is missing. For example, the modeling of image recognition: One needed several steps to understand how to let the computer deconstruct the picture into machine readable lists of numbers when it first arrived on the scene.[5] Programmers first needed to abstract the recognition of images from their own perception and think in pixel-shaped 2D signals on a grid (Jaton 2020, 37–38). What becomes more important than the imagined model is a clear conception of the outcome. The results, in the form of desired outcomes, enable the programmer to evaluate the machines performance and can be presented publicly.

The software script: code

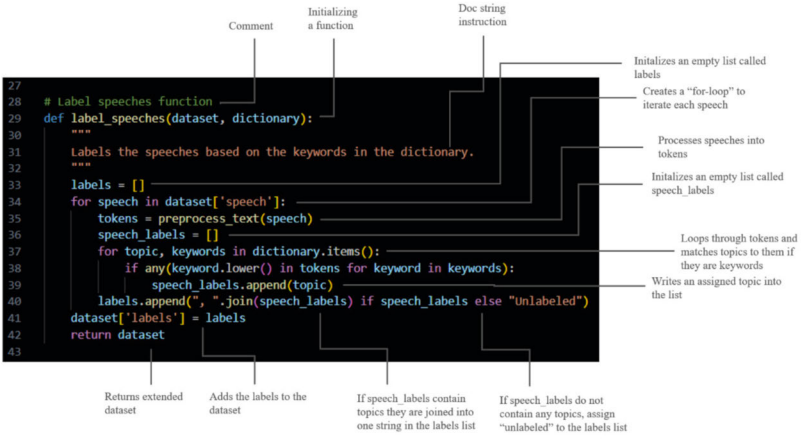
The software script, known as code, is a technical artifact that is also referred to as a rhetorical device (Preda 2024, 718) that delegates the tasks defined by a programmer to the machine. A perfect guide, instruction, and executive framework in one – a law that enforces itself. "It is better than law; it is what lawyers have always dreamed the law to be: an inhumanly perfect *performative* uttered by no one. Unlike any other law or performative utterance, code almost always does what it says because it needs no human acknowledgment. [emphasis in original]" (Chun 2013, 1) However, these are not simply or exclusively messages for the machine in code. By taking a closer look,

5 See this website for a useful demonstration of how a model "looks at" handwritten numbers: https://adamharley.com/nn_vis/cnn/2d.html (Accessed June 1, 2025).

there are patterns with no value for the machine, but that are made for human readers: comments, common structures, naming conventions and, lastly, the software languages that make code readable to humans.

There are two ways to write rules into a software script, without ML and with ML: Without ML, the machine needs its tasks in a strictly structured form – this is achieved through the programming language. The tasks must be specified in clear logic. Machine-readable syntax helps with this: dictionaries, lists, classes for processing data, functions, if-statements, for-loops, etc. Let me illustrate this using the example of 70,000 speeches in a dataset. This is an example of software code that was created without machine learning, thereby being structured through human work.

Figure 1: Screenshot of a code script explained line by line. Fieldnotes J.F.



Credits: Johanna Fischer.

In this example, a dictionary was created in which topics were stored with keywords from German political parties' programs. This input must be stored in a special format and is integrated into the repository's structure (i.e., the code directory). The code shown here gives the computer a series of numbered commands: 1. load dataset with speeches, 2. call dictionary, 3. convert speeches into machine-readable sequences of numbers (tokenization), it follows a function that iterates a task until all parliamentary speeches are labeled, 4. go through each speech and look for the keywords in the dictionary; if the keyword is contained, then enter the appropriate topic label until the dataset has been run through once. This illustrates that writing

software-scripts is the process by which to define a step-by-step instruction for the computer.

It is more abstract to achieve such a task with machine learning. Here, a pre-trained ML algorithm, which acts like a teacher, is provided with a dataset to learn about topics. The algorithm creates new models by 'showing' them the dataset. The models learn patterns and structures from the dataset and write these back into their code. The ML algorithm trains until one of the new models meets the predefined requirements. The models' quality will be measured against various test values on a test dataset. Most of the models will be discarded. The models' code scripts are no longer readable for humans. Programmers can influence this process by adjusting and fine-tuning hyperparameters, assembling the training algorithms, and through careful preparation of the dataset. Once trained, the new model can be shared and provided with new data to process.

Programmers freed themselves of defining rules for models into code by handing tasks over to an ML algorithm. As a result, their work shifted from defining rules to shaping infrastructures for rule-deriving machines. In contrast to writing software-scripts, the training of machines is a process of data preparation, building data *pipelines*, tuning hyperparameters, evaluating output results, and managing hardware to run the training.

Action: exploring and distributing knowledge

Action (Handeln), according to Arendt, is a political practice and an endless form of human process. In interaction with other people, action unfolds as an exchange of objects, knowledge, and stories that form isolated individuals into groups with common goals within a public space (Arendt 1994, 191). Social action also has high significance in sociology; for example, Max Weber described action in contrast to mere reactions to impulses. He defines it as the act, tolerance, or omission, in front of a socially subjective intention (Weber 1976, 13). In this sense, the communicative exchange, the collective planning of machines, and the action according to one's own inclinations and interests between the production processes are social actions.

Arendt's 'action' underscores that programming is not merely technical labor, but is instead a public, world-shaping practice. It is a point acutely relevant today as algorithmic systems mediate politics, culture, and sociality. Programmers often unconsciously become actors in contested social arenas whose actions shape visibility, agency and power by encoding cultural assumptions into infrastructures.

Actions can be understood in the production of MLs in two ways: Firstly, in the interaction with data to gain insights. The exploration of data and the playful engagement with data often comprises bricolage or tinkering (Turkle and Papert 1990, 128). This is considered to be a dialogue between the programmer and the data. Sec-

ond, Human interaction plays a significant role in interaction and exchange with other programmers. It transfers knowledge and exchanges tools to overcome problems. In summary, action encompasses the crucial political, cultural, and subjective aspects of programming.

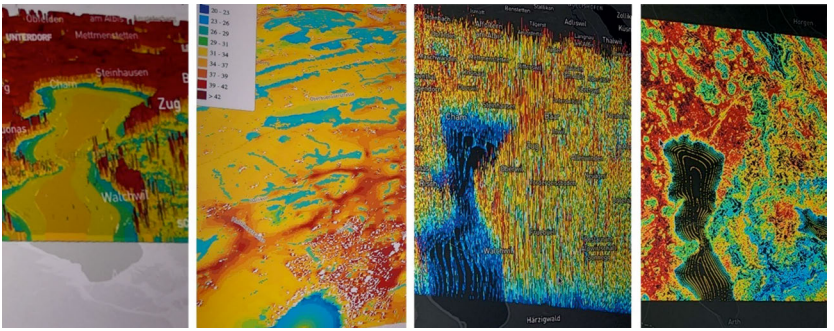
Programming emerges through three modes of action: *experimentation* (playful, iterative engagement with data), *inscription* (codifying and transferring knowledge through texts and tools), and *public dialogue* (negotiating meaning in collaborative spaces). These practices sustain programming as a cultural and epistemic endeavor. By centering action, we expose programming's irreducible sociality: its reliance on human interpretation, its entanglement with public norms, and its capacity to shape the world that it mediates.

Exploratory data analysis – a playful start

Rather than beginning with a fully clear mental model and executing it through code, developers often arrive at understanding through iterative, hands-on experimentation. This can even be experienced as fun and is described as a form of play. One of my colleagues once put it this way:

> "Playing with data meant for me: I was experimenting how to visualize it. […] So, I use these polygons and depict the temperature inside the polygon. The playing was in the visualization itself" (Interview with a data scientist, transcript J.F.).

Figure 2: Visual experiments with sensor data from fieldnotes, J.F.



Credits: Johanna Fischer.

This kind of play is a crucial part of knowledge production. My colleague is describing the trying out of different visualizations. Her goal was to decide on the best form of data representation for insights. Practitioners explore possibilities, test in-

tuitions, and surface insights that may not emerge through formalized, procedural approaches through visual experimentation and through the practice of tinkering.

She reflects further: "I do think that 'tinkering' is an important aspect, but I can get lost in it, enter a flow state, and spend endless amounts of time on it. There's no clear endpoint. You can always go deeper […]. That's why it's important to focus on deliverables, on products that you can hand over and show to clients. Because that's when ideas come back. […] if you present a prototype, you enter an iterative loop." (Interview with a data scientist, transcript J.F.).

As this programmer highlights, tinkering serves as a pivotal mode of engagement, what Bergström and Blackwell (2016, 4) have described as a casual, yet intentional, form of exploration that cuts across distinctions between amateur and expert. It is through such playful and experimental interactions with materials and representations that ideas for production begin to take shape. However, and interestingly, prototypes emerge only in interaction with others in a public sphere. Only through communicating the idea, do programmers find an end to play. The collaborative and iterative shaping of ideas through discourse starts after the exploratory dialogue with the data.

Collaboration and knowledge transfer through inscriptions and public dialogue

Speech is a social act of expressing one's thoughts to another person. Thoughts about programming can be expressed through boundary objects like inscriptions in comments, graphs and notes (Star and Griesemer 1989) in a scientific discourse, as documents, in chatrooms, blogposts, papers, and recordings (Callon 1997, 250), through dialogue on conferences, weekly meetings, or even in coffee breaks (Jaton 2020, 42–45). The impact of these collaboration forms is difficult to measure and is, therefore, overlooked as a part of programming.

Collaboration is institutionalized through information infrastructures that are embedded in code itself: code scripts, documentation, online notebooks, and other sharing devices. They are not meant to be readable by machines, but should instead facilitate communication and knowledge distribution between programmers. Comments describe the function of the following code lines, or 'docstrings,' which are used to insert longer instructions for the use of functions. This standardized form of exchange that is embedded in the code is supported by a broader culture of collaborative software development, which relies on specific tools and organizational practices.

Version control systems provide storage and sharing infrastructure to track changes, thereby facilitating shared authorship and enabling the systematic factoring of code. Shared repositories make codebases accessible to distributed teams, allowing for collective maintenance and the transparent documentation of development histories. Task management systems like ticket systems structure

coordination by segmenting complex projects into tasks, which are often linked to issues of code quality, errors, or refactoring needs. Together, they transform isolated writing into a durable, socially embedded process.

The practices of sharing and distributing textual elements are embedded in organizational frameworks, such as Kanban or Agile; these not only regulate workflows, but also cultivate norms of continuous improvement and code stewardship. Together, the sociotechnical arrangements of programming infrastructures make collaborative coding, refactoring, and long-term maintenance possible across teams and time.

Clicking through the contents on Stack Overflow,[6] a platform for programmers to discuss problems, makes it clear how important exchange between people, problem presentation, and discourse with experts for software production is. This is also reflected in numerous general meetings in which little programming is actually done, but in which much is discussed. Here, problems get described and possible solutions are explained, production concepts are developed, and common knowledge bases are created. The relevance of meetings can be underscored by the observation of translation during such meetings, by translating vague ideas into exact expressions that are ready to be coded (Pütz 2021). In other cases, the definition of a ground truth[7] may limit the ambiguity of human expression (Jaton 2020).

This shows that programming is not just an individual activity, but is instead a social process that involves communication through various means and grades of publicity. These exchanges are supported by institutionalized infrastructures, such as version control systems, shared repositories, and task management tools, all of which enable collaboration and long-term maintenance. Organizational frameworks further structure workflows, thereby fostering continuous progress through feedback. Meetings play a crucial role in translating ideas into executable code, clarifying problems, and establishing shared knowledge, thereby highlighting the importance of discourse in software development, despite difficulty relating to its quantification.

Invisible maintenance, technical production, and public discourse

Programming in ML encompasses diverse sociotechnical practices that can be examined in a structured way through the concepts of labor, work, and action by Han-

6 https://stackoverflow.com/questions (Accessed June 1, 2025).

7 Ground truth refers to the accurate, real-world data or measurements used to train, validate, or test a model; it serves as a reliable reference point, like an answer key in a test, to check whether a machine learning system's predictions are correct.

nah Arendt. The concepts provide a comprehensive understanding of how data scientists engage with and shape material and epistemic infrastructures to produce ML models. I have shown that practices of creating ML algorithms can be described and viewed as cultural and material practice directed to material infrastructures, to the production of artefacts, and to epistemic infrastructures.

Labor practices set the stage for productional work. Labor reproduces, maintains, repairs, and transforms the material infrastructure in which models can be trained and run. These activities are time-intensive and are perceived by programmers as boring prerequisites. Programmers create the environment for ML to be done by infrastructuring the hardware into connected layers. Thus, labor in ML is the reproductive and maintaining force that sets the ground for automation.

Work contains practices, such as writing code to structure, guide, and to evaluate the training of MLs. Code can be seen as exemplified signals that contain rules that interface with automated actants and that can be shared, changed, and implemented by other humans or technological actants. The results of training with data are applicable to ML algorithms.

The relevance of Arendt's action to ML lies in its capacity to frame programming not merely as technical labor, but as a public, collaborative practice that shapes epistemic infrastructures. Exploration and iterative model refinement are not neutral technical steps, but are instead acts of disclosure. Programmers test hypotheses, challenge assumptions, and reveal new possibilities for data's meaning. This experimentation is inherently uncertain and generative, thereby opening up spaces for reinterpretation. The exchange of ideas via inscriptions among programmers, through meetings and chats, constitutes a public dialogue that sustains practices in ML as a collective endeavor. This notion grounds programming not as isolated work, but as a cultural practice.

In summary, there is a shift from inscribing computational rules to inscribing structures for a machine that learns rules from data with ML. This ethnography is contrasted with the traditional image of the developer as a brain-heavy genius who can architect complex systems entirely in her head before ever touching a keyboard. Instead, what emerges is the collective labor of maintaining the material environment, the producing work, and the public action of hundreds of hands. Their doing is grounded in the messiness of materials, play, aesthetics, collaboration, and in situated responsiveness. Algorithms get tweaked and tuned in endless loops of updates to keep pace with material boundaries. Working with data becomes a form of craft, one in which insights are shaped through manipulation, feedback, and sensory engagement.

By examining ML programming through the concepts of labor, work, and action, we gain a clearer understanding of the multifaceted nature of this practice. Programmers in ML not only strive to automate tasks and to create durable code, but also to engage in dynamic, playful, and collaborative processes that perpetu-

ate the circular nature of programming in the field of ML. Their labor will shape, maintain, and repair material infrastructures. Their work creates executable code and shapes datasets. Their action will transform epistemic infrastructures by both exploring and distributing knowledge.

References

Amoore, Louise and Piotukh, Volha (2016) *Algorithmic Life: Calculative Devices in the Age of Big Data* London; New York: Routledge.

Arendt, Hannah (1958) *The human condition*. Chicago, IL and London: University of Chicago Press.

Arendt, Hannah (1994) *Vita activa oder Vom tätigen Leben*. München: Piper.

Bergström, Ilias and Blackwell, Alan F. (2016) The practices of programming, *2016 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pp. 190–198.

Callon, Michel (1997) Four Models for the Dynamics of Science. In Tauber, Alfred I. (ed.) *Science and the Quest for Reality*. London: Palgrave Macmillan UK, pp. 249–292.

Chun, Wendy Hui Kyong (2013) *Programmed Visions: Software and Memory*. Cambridge, MA: MIT Press.

Denis, Jérôme and Pontille, David (2015) Material Ordering and the Care of Things, *Science, Technology, & Human Values*, 40(3), pp. 338–367.

Fayard, Anne-Laure (2021) Notes on the Meaning of Work: Labor, Work, and Action in the 21st Century, *Journal of Management Inquiry*, 30(2), pp. 207–220.

Graham, Stephen and Thrift, Nigel (2007) Out of Order: Understanding Repair and Maintenance, *Theory, Culture & Society*, 24(3), pp. 1–25.

Haigh, Thomas, Priestley, Mark, and Rope, Crispin, (2016) *ENIAC in Action: Making and Remaking the Modern Computer*. Cambridge, MA: MIT Press.

Häußling, Roger (2020) Soziologie des Digitalen. In Frenz, Walter (ed.) *Handbuch Industrie 4.0: Recht, Technik, Gesellschaft*. Berlin and Heidelberg: Springer, pp. 1355–1381.

Hilgartner, Stephen, Miller, Clark, and Hagendijk, Rob (2015) *Science and Democracy: Making Knowledge and Making Power in the Biosciences and Beyond*. New York: Routledge.

Hooker, Sara (2020) *The Hardware Lottery*. arXiv.

Jackson, Steven J. (2016) Speed, Time, Infrastructure: Temporalities of Breakdown, Maintenance, and Repair. In Wajcman, Judy and Dodd, Nigel (ed.) *The Sociology of Speed: Digital, Organizational, and Social Temporalities*. Oxford: Oxford University Press, p. 0.

Jasanoff, Sheila (2004) *States of Knowledge: The Co-Production of Science and the Social Order*. London: Routledge.

Jaton, Florian (2020) *The constitution of algorithms: ground-truthing, programming, formulating*. Cambridge, MA: MIT Press.

Law, John (2017) STS as Method. In Felt, Ulrike, Fouché, Rayvon, Miller, Clark A., and Smith-Doerr, Laurel (eds.) *The Handbook of Science and Technology Studies*. 4th edition. Cambridge, MA: MIT Press, pp.31–57.

Munn, Luke (2022) *Automation Is a Myth*. Stanford, CA: Stanford University Press.

von Neumann, John (1993) First draft of a report on the EDVAC, *IEEE Annals of the History of Computing*, 15(4), pp. 27–75.

Neyland, Daniel (2019) *The Everyday Life of an Algorithm*. Cham: Springer Nature.

Pardo-Guerra, Juan Pablo (2024) Machine Learning, Infrastructures, and Their Sociomaterial Possibilities. In Borch, Christian and Pardo-Guerra, Juan Pablo (eds.) *The Oxford Handbook of the Sociology of Machine Learning*, Oxford University Press, pp. 297–310.

Preda, Alex (2024) Coding and Expertise. In Borch, Christian and Pardo-Guerra, Juan Pablo (eds.) *The Oxford Handbook of the Sociology of Machine Learning*, Oxford University Press, pp. 549–566.

Pütz, Ole (2021) Managing exactness and vagueness in computer science work: Programming and self-repair in meetings, *Social Studies of Science*, 51(6), pp. 938–961.

Ramirez, J. Jesse (2020) *Against Automation Mythologies: Business Science Fiction and the Ruse of the Robots*. New York: Routledge.

Raschka, Sebastian and Mirjalili, Vahid (2017) *Machine Learning mit Python und Scikit-Learn und TensorFlow: Das umfassende Praxis-Handbuch für Data Science, Predictive Analytics und Deep Learning*. Frechen: mitp.

Schabacher, Gabriele (2022) *Infrastruktur-Arbeit: Kulturtechniken und Zeitlichkeit der Erhaltung*. Berlin: Kulturverlag Kadmos.

Seaver, Nick (2019) Knowing Algorithms. In Vertesi, Janet and Ribes, David (eds.) *digitalSTS. A field Guide for Science & Technology Studies*. Princeton, NJ: Princeton University Press.

Star, Susan Leigh (1999) The Ethnography of Infrastructure, *American Behavioral Scientist*, 43(3), pp. 377–391.

Star, Susan Leigh and Griesemer, James R. (1989) Institutional Ecology, "Translations" and Boundary Objects: Amateurs and Professionals in Berkeley's Museum of Vertebrate Zoology, 1907–39, *Social Studies of Science*, 19(3), pp. 387–420.

Turkle, Sherry and Papert, Seymour (1990) Epistemological Pluralism: Styles and Voices within the Computer Culture, *Signs*, 16(1), pp. 128–157.

Vertesi, Janet et al. (2019) *digitalSTS: A Field Guide for Science & Technology Studies*. Princeton, NJ: Princeton University Press.

Weber, Max (1976) *Max Weber Gesamtausgabe. Studienausgabe / Max Weber Studienausgabe: Band I/22,1: Wirtschaft und Gesellschaft. Gemeinschaften*. Tübingen: Mohr Siebeck.

Weber, Max (2010) *Religion und Gesellschaft: gesammelte Aufsätze zur Religionssoziologie [die protestantische Ethik und der Geist des Kapitalismus, die Wirtschaftsethik der Weltreligionen u.a.]*. Frankfurt a.M.: Zweitausendeins.