Is Machine Learning Software Just Software: A Maintainability View

Tommi Mikkonen[1], Jukka K. Nurminen[1], Mikko Raatikainen[1], Ilenia Fronza[2], Niko Mäkitalo[1], and Tomi Männistö[1]

[1] University of Helsinki
Helsinki, Finland
email: first.[initial.]last@helsinki.fi
[2] Free University of Bozen-Bolzano
Bolzano, Italy
email: Ilenia.Fronza@unibz.it

Abstract. Artificial intelligence (AI) and machine learning (ML) is becoming commonplace in numerous fields. As they are often embedded in the context of larger software systems, issues that are faced with software systems in general are also applicable to AI/ML. In this paper, we address ML systems and their characteristics in the light of software maintenance and its attributes, modularity, testability, reusability, analysability, and modifiability. To achieve this, we pinpoint similarities and differences between ML software and software as we traditionally understand it, and draw parallels as well as provide a programmer's view to ML at a general level, using the established software design principles as the starting point.
Keywords: Software engineering, software maintenance, artificial intelligence, machine learning, modularity, reusability, analysability, modifiability, testability

1 Introduction

Artificial intelligence (AI) and machine learning (ML) is becoming commonplace in numerous fields. Such techniques help us to build interactive digital assistants, plan our route in traffic, or perform stock transactions.

While there are several ways to implement AI features, for the purposes of this paper, we focus on ML, a flavor of AI where algorithms improve automatically through experience [12] particularly by approaches based on neural networks. Very large neural networks, commonly called deep learning, can have billions of parameters whose values are optimized during the training phase. The resulting trained networks are able, for example, to detect objects in pictures, understand natural language, or play games at a superhuman level.

With these ML features, a new challenge has emerged: how to integrate ML components into a large system? So far, we have found ways to build individual (sub)systems but there is little established engineering support for creating and maintaining large systems that should be always on, produce reliable and valid

results, have reasonable response time and resource consumption, survive an extended lifetime, and, in general, always place humans first in the process. Since ML systems are built using software, we believe that these issues can only be mitigated by considering both software engineering and data science building ML software.

There are several overview papers about software engineering of ML applications. Reporting workshop results [7] discuss the impact of inaccuracy and imperfections of AI, system testing, issues in ML libraries, models, and frameworks. Furthermore, Arpteg et al. [1] identified challenges for software design by analyzing multiple deep learning projects. Some surveys have focused on particular aspects of software development, such as testing [15]. However, in comparison to most of the prior overviews, we take a more holistic system view, and structure the challenges brought by ML to qualities of software design, in particular in the context of software maintenance.

More specifically, we study ML systems and their features in the light of maintenance as defined in software product quality model standard ISO/IEC-25010 [6], and its five subcharacteristices (or attributes) – modularity, reusability, analysability, modifiability, testability. The work is motivated by our experiences in developing applications that include ML features as well as observations of others, in particular Google's machine learning crash course[1], that points out that even if ML is at the core of an ML system, only 5% or less of the overall code of that total ML production system in terms of code lines. Our focus was on the characteristics of ML themselves; patterns, such as wrappers and harnesses, that can be used to embed them into bigger systems in a more robust fashion will be left for future work. Furthermore, in this work we explicitly focused on software maintainability, and other characteristics addressed by the standard are left for future work.

The rest of this paper is structured as follows. Section 2 discusses the background and motivation of this work. Section 3 then continues the discussion by presenting a programmer's view to AI. Section 4 studies maintainability of AI systems, and Section 5 provides an extended discussion and summary of our findings. Finally, Section 6 draws some final conclusions.

2 Background: ML Explained for Programmers

Machine learning is commonly divided into three separate classes. The most common is *supervised learning* where we have access to the data and to the "right answer" often called a label, e.g., a photo and the objects in the photo. In *unsupervised learning*, we just have the data and the ML systems try to find some common structure in the data, e.g., classify photos of cats and dogs to different categories. Finally, in *reinforcement learning* the system learns a sequence of steps that leads it to a given goal, e.g., to a winning position in a game of chess. In this paper, we primarily deal with supervised learning, which is the most

[1] https://developers.google.com/machine-learning/crash-course/production-ml-systems, accessed Aug. 18, 2020.

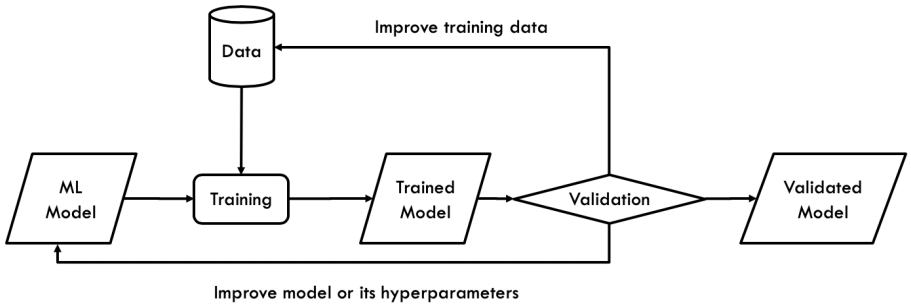

Fig. 1. Illustration of the ML training process and the search for a good model.

common approach. Many of our examples deal with neural networks although the ideas apply to other forms of supervised machine learning as well.

Developing an ML model requires multiple steps, which in industrial development are more complicated than in academic exploration [4]. Figure 1 gives an overview of the process. As the starting point, data must be available for training. There are various somewhat established ways of dividing the data to training, testing, and cross-validation sets. Then, an ML model has to be selected, together with the hyperparameters of the model. The hyperparameters define, e.g., how many and what kind of layers a neural network has and how many neurons there are in each layer.

Next, the model is trained with the training data. During the training phase, the weights of the neurons in the network are iteratively adjusted so that the output of the neural network has a good match with the "right answers" in the training material.

The trained model can then be validated with different data – although in software engineering this would more correspond to verification rather than validation that takes place with end users. If this validation is successful (with any criteria we decide to use) the model is ready for use. Then, it has to be embedded with the rest of the software system. Often the model, which can be the core of the application, is just a small part of the whole software system, so the interplay between the model and the rest of the software and context is essential [13].

To summarize, on the surface, the promise of ML systems is tempting for solving various problems. Based on the data only, the system learns to produce results without any human involvement or consideration. Internally, however, ML is just software and one way to think of deep learning is that it is just a new, yet very powerful, algorithm to the toolbox of the software developers. Its characteristics, however, are different from engineering tradition – instead of predefined executions, ML defines its own algorithms based on training data that is used to tailor the exact behavior, following the patterns that the system was able to absorb from the data.

So far, some work on the differences between AI software and "vanilla" software exist. In particular, Zhang et al. [15] refer to a number of challenges for AI testing: statistical nature of machine learning; data-driven programming paradigm with high sensitivity to data; evolving behavior; oracle problem; and emergent properties when considering the system as a whole. Moreover, the black-box nature of many ML approaches, especially neural networks, and the huge input space making the testing of rare cases difficult are common problems [10].

Next, we will elaborate the above challenges in the context of a real-life software project, and then we relate them to maintainability in terms of modularity, testability, reusability, analysability, and modifiability, as proposed by the ISO/IEC-25010 standard [6].

3 Challenges with an ML component and experiences from a sample Project

In this section, we present a set of practical issues with an ML component that is concretized by a practical example. In this example research project[2] discussed in this paper, we worked with the Jira[3] issues tracking software. The Jira installation for managing the development and maintenance of a large industrial system contained over 120,000 Jira issues. Frequently, multiple (Jira) issues are about the same thing and should be *linked*: they can be duplicates, or otherwise related, such as an issue requiring the solution of another issue. As links are encoded manually in Jira, many may be missing. Because of the large number of issues, finding potentially missing links is hard. To help users to manage the links, we studied different natural language processing services based on existing algorithms and implementations to analyze the textual issue descriptions and to propose the users potentially missing links between them.

To summarize the experiences from the example, services that were promising in small scale or during development produced less good outcome in industrial scale use. There were also several challenges related to deployment, such as continuous integration and security, which are not covered here. In the end, rather than having one ML service that would produce the results, we used a design that combines the best results from different services and adds application-specific filters and contextualization. For a traditional software application, such design could have implied design flaws, but for AI software such design seemed a somewhat normal, i.e., things that one must be prepared to deal with.

While the example can be regarded as simple, it allows us to highlight detailed experiences, listed below in different subsections.

Stochastic results An ML model usually reaches certain accuracy, e.g., 98%. But how do we deal with the cases which are not correct? In classical software

[2] https://openreq.eu, accessed Aug. 18, 2020.
[3] https://www.atlassian.com/fi/software/jira, accessed Aug. 18, 2020.

we usually do not experience this at all. The problem gets even more complex when considering the different kinds of errors, for instance false positives against false negatives. The severity of errors is also system-specific.

Example: The detection of duplicates provides a score that can be used to filter and order expected true positives. However, users did not consider false positives, i.e., wrong proposals, a major problem when the results are ordered by the score. In this case the system only assists users leaving them the final decision and any help was considered beneficial. False negatives, i.e., not detecting duplicates, are more problematic.

High sensitivity to data ML results are extremely sensitive to training data. A very minor change – even such as changing the order of two training examples – can change the system behavior. Similarly, any new training data will most likely change the outcome. Furthermore, measures such as accuracy and precision rely often on incomplete data, resulting in under- or overfitting due to, e.g., imperfect training data. Furthermore, operating in a dynamic environment of constantly changing data is challenging, because batch processing of data causes discontinuity and excessive resource consumption.

Example: Very careful fitting turned out to be unnecessary for the users and impossible because training data is always incomplete. The missing links, which we tried to detect, were also naturally largely missing from any available training data sample. We decided that better solution would be to monitor users' acceptance and rejection rates for the proposals.

Oracle problem In many cases where AI is involved, we do not know what is the right answer. This oracle problem is also common in the context of some algorithms, such as in optimization, where we do not know what is the best answer either. An additional aspect of this, related to AI ethics, is that we may have difficulty to agree what is the right answer [2]. A commonly used example of such a case is how a self-driving car should behave in a fatal emergency situation – to protect the persons inside the car or those around it, or, given a choice between a baby and an elderly person, which should be sacrificed.

Example: The users differ: one prefers to add many links while another uses links more scarcely. Likewise, if the issues are already linked indirectly via another issue, it is a subjective and context-dependent decision whether to add a redundant link. Thus, there are no single answers whether to add a link. Another challenge is that two issues can be first marked as duplicates but then they are changed and do not duplicate each other anymore.

Evolving behavior Many ML systems are build around the idea that the system can learn on-the-fly as it is being used. For instance, so-called reinforcement learning allows ML to chart the unknown solution space on the fly, aiming to maximize some utility function. The learning can also take place in operational use, meaning that the system can adapt to different situations by itself.

Example: All users decisions are anonymously recorded and the decisions can be used to change the behavior. However, when to change behavior? Changing behavior after each user might result in unbalanced behavior because decisions are subjective. Constant behavior change was also considered computationally expensive compared to its benefits.

Black box Neural networks are to a large degree regarded as black boxes. Therefore, understanding why they concluded something is difficult. Although there are new ways to somehow study the node activations in the network, it is still hard to get an understandable explanation. On one hand, this influences the trust users have to the system, and on the other hand, also makes debugging neural networks and their behavior very hard.

Example: Even in our simple case, the proposals do not provide any rational why a link is proposed so the users are not informed either. Without explanation and too many false positives – and perhaps even false negatives – users' trust and interest in the system in the long term remains a challenge.

Holistic influences With ML, it is not possible to pinpoint the error to a certain location in the software, but a more holistic view must be adopted. The reason why the classical approach of examining the logic of the computer execution does not work is that both the quality of entire training data set as well as the selected model have an influence. Consequently, there is no single location where to pinpoint the error in the design. As a result, a lot of the development work is trial-and-error to try to find a way how the system provides good results.

Example: It was not always clear should we improve data, model or software around it. Testing was largely manual requiring inspection and domain knowledge. We also tested another duplicate detection service but any larger data (more than 30 issues) crashed the service without explanation of the cause. As the results were similar with the first service, we quickly disregarded this service.

Unclear bug/feature division The division between a bug and a feature is not always clear. While it is possible there is a bug somewhere, a bad outcome can be a feature of the ML component caused by problems in the data (too little data, bad quality data, etc.). Some of the data related problems can be very hard to fix. For instance, if a problem is caused by shortage of training data it can take months to collect new data items. Moreover, even if the volume of data is large it can cover wrong cases. For example, in a system monitoring case, we typically have a lot of examples of the normal operation of a system but very few examples of those cases where something accidental happens.

Example: The Jira issues use very technical language and terminology, and can be very short and laconic. This caused sometimes incorrect outcome that is immediately evident for a user, e.g., by considering the issue, its creation time, and the part of software the issue concerns.

Huge input space Thorough testing of an ML module is not possible in the same sense as it is possible to test classical software modules and to measure coverage. Thorough testing of classical software is as such also difficult but there are certain established ways, especially if the input space to a function/module is constrained. In an ML system, the input data often has so many different value combinations that there is no chance to try out them all or even to find border cases for more careful study. As shown in the adversarial attack examples, a small carefully planned change in input data can completely change the recognition of a picture or spoken command. In all cases we do not have nasty adversarial attackers but those examples show that such situations can happen if the data randomly happens to have certain characteristics.

Example: The Jira issues are very different from each other: Some of them are very short and laconic while others contain a lot of details. Sometimes the title contains the essential information while sometimes the title is very general and information is in the description. Thorough testing for optimal solutions, and even finding archetypal cases is hard.

4 ML in the Light of Maintainability

To study software maintenance in relation to ML software, the characteristics of ML systems were analyzed in the light of the key quality attributes. The analysis was first performed by two first authors, based on their experience on software design and ML, and then validated and refined by the rest of the authors.

Modularity Modularity is the property of computer programs that measures the extent to which programs have been composed out of separate parts called modules. Module is generally defined to be a self-contained part of a system, which has a well-defined interface to the other parts, which do not need to care what takes place inside the module. The internal design of a module may be complex, but this is not relevant; once the module exists, it can easily be connected to or disconnected from the system.

In the sense of modularity, ML modules and classical SW modules coexist. Dependencies between modules may happen via large amounts of data, and output of an ML module can be input to another ML module. However, because the ML module operation is not perfect (e.g., accuracy 97%) modules taking output from ML modules need to live with partly incorrect data. When an upstream ML module is learning to be better, it is unclear what happens to downstream ML modules that have learned to deal with faulty input – when input now becomes more correct, will the downstream ML module actually give worse results? Oftentimes this implies that instead of decomposing complex ML function to simpler ones, the ML system is trained as a self-contained entity.

Another issue is related to interfaces. ML and especially neural networks are a bit too good to hide information. Therefore, understanding what is happening in an ML module is challenging and a lot of work on explainable AI is ongoing.

Sometimes – as is the case in our example – dealing with this leads to using several modules that overlap in features for the best results.

Testability Testing the software that implements machine learning can be tested like any other piece of software. Furthermore, the usual tools can be used to estimate the coverage and to produce other metrics. Hence, in the sense of code itself, testing ML software has little special challenges.

In contrast, testing ML systems with respect to features related to data and learning has several complications. To begin with, the results can be stochastic, statistical or evolving over time, which means that they, in general, are correct but there can also be errors. This is not a good match with classical software testing approaches, such as the V-model [11], where predestined, repeatable execution paths are expected. Moreover, the problems can be such that we do not know the correct answer – like in many games – or, worse still, us humans do not agree on the correct answer [2]. Finally, while many systems are assessed against accuracy, precision, or F-score using a test data set, there is less effort on validating that the test data set is correct and produces results that are not over- or underfitted.

In cases where the ML system mimics the human behavior – such as "find traffic sign in the picture" in object detection – a well-working AI system should produce predictable results. Again, most ML systems do not reach 100% accuracy so we need ways to deal with also inaccurate results. In some cases, like in targeted advertisement, it is adequate that the accuracy level is good enough while in other cases, like in autonomous vehicles, high trust to the results is necessary.

Reusability Reusability is a quality characteristic closely associated with portability; it refers to the ability to use already existing pieces of software in other contexts. As already mentioned, in ML, the amount of code is relatively small and readily reusable, but reusing data or learning results is more difficult.

To begin with, there are reuse opportunities within the realm of ML itself. For instance, models can be reused. In fact, the present practice seems to be to pick a successful model from the literature and then try to use it – instead of inventing the models each time from scratch – even in completely different domains and use cases. Furthermore, the same data set can be used for training in several services, or one service can combine different data for training. For instance, in the example presented above, we used a number of different training data. Moreover, some data can be also quite generic, such as corpus from Wikipedia.

In ML, there is also a form of reuse called transfer learning [9]. In essence, transfer learning is an act of storing knowledge gained while solving one problem, and then applying the knowledge to a different but related problem at the level of trained ML modules. While the initial training often requires massive datasets, and huge amount of computing, retraining the module for particular data often requires far less data and computation. However, it may be hard to decide for sure if the retrained module is behaving well, because starting the training with

a pretrained model can lead to rapid learning results, but this process does not guarantee much about its correct eventual behavior.

Finally, as ML modules can evolve over time, it is possible that they help to adapt the software to a new context. This can help reusing the modules in new applications.

Analysability Since computer programs are frequently read by programmers while constructing, debugging, and modifying them, it is important that their behavior can be easily analyzed. Moreover, the behavior of neural networks can be studied and recorded for further analysis.

However, in ML, structural information associated with a neural network or characteristics of individual neurons bear little value in terms of analysability. Instead, the behavior is intimately related to data. Hence, while we can study individual neurons, for instance, the decision making process as a whole cannot be analysed without additional support in the system.

Modifiability An ML module typically requires very little code. Therefore, modifying the logic of the ML module does not require much effort, and it seems that such code is modified somewhat routinely by the developers. There are also options to prune and shrink pre-trained networks so that they can be run with less hardware resources [14].

In contrast, modifying data can have dramatic effects. For instance, during training, a small change in input data – or just the change of the random generator seed – can change the results. Furthermore, the same training set can generate totally different neural network structure if we allow the system to search in an automated fashion – so-called AutoML [5] – for the best hyperparameters and network structure. As the neural network self-organizes itself, chances are that different instances trained with the same datasets organize themselves differently, so that their structures cannot be compared directly, and, worse still, produce partially different results.

The fact that different training data produces different results does not only introduce problems associated with modifiability. There can be cases where the only modification that is needed for using the same software is training with different data, and the software can be used intact.

5 Discussion

Above, we have presented a number of ML/AI related challenges to software maintenance. Table 1 presents the relationships between maintenance related characteristics and different aspects of machine learning. To summarize, software written for implementing ML related features can be treated as any other software from the maintainability perspective. However, when considering the data and the machine learning part, chances are that tools and techniques that are available are not enough. Therefore, in the end, the users should also be involved in the activities to ensure correct behavior.

Table 1. Summary of relationships between quality attributes and ML features.

	Modularity	Testability	Reusability	Analysability	Modifiability
SR	neutral	negative; testing aims to identify bugs, whereas stochastic results escape discrete testing	neutral	negative; stochastic results cannot be easily analyzed	negative; modifying stochastic process can produce results that are hard to predict
HSD	neutral	negative; testing aims to identify bugs, and reliance to data does not lend itself to discrete testing.	mixed; data sets can be reused, whereas reusing trained systems in different contexts can be hard	negative; analysing data centric features often requires additional support from the infrastructure	mixed; code can usually be modified with ease, whereas modifying data set can introduce complications
EB	neutral	negative; testing in general builds on discrete behaviors and faults, and has little room for evolving behavior	mixed; evolving behaviors can adapt to new situations, whereas their validation and verification in a new context can be hard	negative; analysing an evolving behavior is more complex than analyzing static behaviors	mixed; the behavior can evolve to satisfy new needs, but triggering this can be complex
OP	neutral	negative; testing features whose output is not well defined is hard	neutral	negative; analysing an outcome that is based on foreseen results is hard	neutral
BB	positive; modules by default respect modular boundaries	mixed; modules can be tested separately, but calculating metrics is hard	positive; modules can be easily reused	negative; the behavior is invisible and hence escapes analysis	negative; black box behavior cannot be modified directly
HI	negative; separation of concerns does not really happen as ML modules may be intertwined	negative; testing cannot be focused but needs to be holistic	negative; units of reuse are hard to define	negative; holistic behavior is hard to analyze	negative; modifications can have holistic effect
UD	neutral	negative; it is unclear when a test fails for what reason	neutral	negative; it is unclear what to analyze	neutral
HIS	negative; module with arbitrarily large input interface is difficult to manage	negative; testing large input space is complex	neutral	negative; the larger the input space, the more complex analysis might be needed	neutral

Legend:
SR: Stochastic results
HSD: High sensitivity to data
EB: Evolving behavior
OP: Oracle problem
BB: Black box
HI: Holistic influences
UD: Unclear bug/feature division
HIS: Huge input space

Threats to Validity A key threat to the validity of our observations is that the study was performed by the authors based on their subjective experience on software design and maintenance, and ML systems. This can be a source of bias in the results. To mitigate this, all the results were analyzed by two or more authors as they were recorded in Table 1. A further threat to external validity is that there are various approaches to AI/ML, whose characteristics differ considerably. To mitigate this threat, we have narrowed the scope of this work to maintainability as defined by the ISO/IEC-25010 standard [6] and ML, which is only a subset of AI.

Future Work As for future work, there are obvious directions where we can extend this work. To begin with, as already mentioned. we plan to perform a similar analysis of other software quality aspects of ISO/IEC-25010 standard. These include functional suitability, performance efficiency, compatibility, usability, reliability, security, and portability. While some of these are related to maintainablity addressed in this paper, these topics open new viewpoints to AI/ML software.

Furthermore, there are additional considerations, such as ethics [3], which have emerged in the context of AI. Such topics can also be approached from the wider software engineering viewpoint, not only from the perspective of novel techniques.

Finally, running constructive case studies on the impact of the software design principles in AI/ML software is one of the future paths of research. To disseminate the results, we plan to participate in the work of SO/IEC JTC 1/SC 42, which just accepted to start working on a working draft on "Software engineering: Systems and software Quality Requirements and Evaluation (SQuaRE) – Quality Model for AI-based systems".

6 Conclusions

In this paper, we have studied ML in the context of software maintainability. To summarize the results, while ML affects all characteristics of software maintenance, one source of complications is testing and testability of ML in general. Testing builds on the fact that software systems are deterministic, and it has long been realized that systems where different executions may differ – due to parallel executions for instance – often escape the traditional testing approaches. Same concerns arise when modules can have evolving behaviors or which can not be debugged with the tools we have. Hence, building new verification and validation tools that take into account the characteristics of ML are an important direction for future work.

To a degree, concerns that are associated with testability apply to analysability, including in particular black box behavior and reliance on large data sets. Hence, understanding how to measure test coverage or analyze the behavior of an AI module forms an obvious direction for future work. Moreover, since data

is a key element in many ML systems, its characteristics will require special attention in the analysis.

Finally, as already mentioned as well as pointed out, e.g., by Kuwajima et al. [8], pattern-like solutions, such as wrappers, harnesses and workflows, for example, that can be used to embed ML related functions into bigger systems in a more robust fashion form a direction for future software engineering research.

References

1. Arpteg, A., Brinne, B., Crnkovic-Friis, L., Bosch, J.: Software engineering challenges of deep learning. In: Proceedings - 44th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2018. pp. 50–59. IEEE (8 2018). https://doi.org/10.1109/SEAA.2018.00018
2. Awad, E., Dsouza, S., Kim, R., Schulz, J., Henrich, J., Shariff, A., Bonnefon, J.F., Rahwan, I.: The moral machine experiment. Nature **563**(7729), 59 (2018)
3. Bostrom, N., Yudkowsky, E.: The ethics of artificial intelligence. The Cambridge handbook of artificial intelligence **1**, 316–334 (2014)
4. Breck, E., Polyzotis, N., Roy, S., Whang, S.E., Zinkevich, M.: Data infrastructure for machine learning. In: SysML conference (2018)
5. Feurer, M., Klein, A., Eggensperger, K., Springenberg, J., Blum, M., Hutter, F.: Efficient and robust automated machine learning. In: Advances in neural information processing systems. pp. 2962–2970 (2015)
6. ISO: IEC25010: 2011 systems and software engineering–systems and software quality requirements and evaluation (SQuaRE)–system and software quality models (2011)
7. Khomh, F., Adams, B., Cheng, J., Fokaefs, M., Antoniol, G.: Software engineering for machine-learning applications: The road ahead. IEEE Software **35**(5), 81–84 (2018)
8. Kuwajima, H., Yasuoka, H., Nakae, T.: Engineering problems in machine learning systems. Machine Learning pp. 1–24 (2020)
9. Pan, S.J., Yang, Q.: A survey on transfer learning. IEEE Transactions on knowledge and data engineering **22**(10), 1345–1359 (2009)
10. Pei, K., Cao, Y., Yang, J., Jana, S.: Deepxplore: Automated whitebox testing of deep learning systems. In: 26th Symposium on Operating Systems Principles. pp. 1–18 (2017)
11. Rook, P.: Controlling software projects. Software engineering journal **1**(1), 7–16 (1986)
12. Schapire, R.E., Freund, Y.: Foundations of machine learning (2012)
13. Sculley, D., Holt, G., Golovin, D., Davydov, E., Phillips, T., Ebner, D., Chaudhary, V., Young, M., Crespo, J.F., Dennison, D.: Hidden technical debt in machine learning systems. In: Advances in neural information processing systems. pp. 2503–2511 (2015)
14. Wang, H., Zhang, Q., Wang, Y., Hu, H.: Structured probabilistic pruning for convolutional neural network acceleration. arXiv:1709.06994 [cs.LG] (2017)
15. Zhang, J.M., Harman, M., Ma, L., Liu, Y.: Machine learning testing: Survey, landscapes and horizons. IEEE Transactions on Software Engineering (2020)