



Machine Learning

Panos Louridas and Christof Ebert

Machine learning is the major success factor in the ongoing digital transformation across industries. Startups and behemoths alike announce new products that will learn to perform tasks that previously only humans could do, and perform those tasks better, faster, and more intelligently. But how do they do it? What does it mean for IT developers and software engineers? Here, Panos Louridas and I present a brief overview of machine-learning technologies, with a concrete case study from code analysis. I look forward to hearing from both readers and prospective column authors. —Christof Ebert

MACHINE LEARNING ISN'T new; it has been around at least since the 1970s, when the first related algorithms appeared. What has changed is that the explosion in computing power has allowed us to use machine learning to tackle ever-more-complex problems, while the explosion of data being captured and stored has allowed us to apply machine learning to an ever-expanding range of domains.

Machine learning is used in different domains. Here are a few examples:

- security heuristics that distill attack patterns to protect, for instance, ports or networks;
- image analysis to identify distinct forms and shapes, such as for medical analyses or face and fingerprint recognition;
- deep learning to generate rules for data analytics and big data handling, such as are used in marketing and sales promotions;
- object recognition and predictions from combined video streams and

multisensor fusion for autonomous driving; and

- pattern recognition to analyze code for weaknesses such as criticality and code smells (for a related case study, see the sidebar).

The general idea behind most machine learning is that a computer learns to perform a task by studying a training set of examples. The computer (or system of distributed or embedded computers and controllers) then performs the same task with data it hasn't encountered before.

Learning Strategies

Machine learning employs the following two strategies (see Figure 1).

Supervised Learning

In *supervised learning*, the training set contains data and the correct output of the task with that data. This is like giving a student a set of problems and their solutions and telling that student to figure out





CASE STUDY: MACHINE LEARNING FOR CODE ANALYSIS

Machine learning has many practical applications. As befits this magazine, we'll present an example that shows how machine learning can manage quality risks and improve quality assurance productivity.

A CRITICALITY ASSESSMENT TOOL

Project managers and product owners often wonder when would be the right release point and how to assess the criticality of the code to be delivered. Static-analysis tools exist but give numerous warnings that might be difficult to link to actual weak spots. At Vector Consulting Services, we offer clients a machine-learning-based criticality assessment tool for software release management. Criteria include hard factors from static code analysis, such as cyclomatic complexity, the degree of reuse, and the history of defects in preceding versions and variants. We also use soft factors such as designer competences, experience with similar projects, and architectural decisions that might incur technical debt.

Using these criteria, the machine-learning tool builds a criticality prediction model. On the basis of a ranked list of the criticality of the modules used in a build, developers can apply different mechanisms to improve quality—refactoring, redesign, thorough static analysis, and unit testing with increased coverage schemes.

Instead of predicting the number of defects or changes (algorithmic relationships), the tool considers assignments to classes (for example, “defect prone”). The training and test data come from finished projects that had been under configuration control since coding started. After the machine-learning process, the data used operationally comes from active projects.

To achieve feedback to improve predictions, this approach is integrated throughout development (requirements, design, code, system test, and deployment).

STEP-BY-STEP CRITICALITY PREDICTION

Figure A illustrates criticality classification and validation, which consists of eight steps.

Step 1: For the finished projects, provide a list of all the modules used for learning from the configuration system.

Step 2: Provide a defect list for each learning module. For high-ranking defects, you might add a root-cause analysis that allows for a Pareto-based mitigation list.

Step 3: Provide a change history classification (that is, the number of compiles or deliveries) for each learning module.

Step 4: With static code analysis, assemble for each learning module a complexity classification such as hot spots from code analysis.

Step 5: With the machine-learning system, construct an initial criticality list that takes into account the inputs from steps 2, 3, and 4, mapped to the list from step 1. Evaluate the criticality list's validity—for example, by screening for the identified critical modules, outliers, and potential misleading effects. Such screening aims to find undesired influences from the defect or change histories. The screening and ranking must primarily ensure the fewest possible type-I prediction errors. (In type-I errors, defect-prone components are misclassified as uncritical components.)

Step 6: For the current project, repeat steps 1 to 5 to get a predictive result on each new module's criticality. Then, present the rankings so that the developers can decide on further actions.

Step 7: Manually prepare suggestions based on the new modules ranked the most critical. Critical modules should at least undergo a flash review and subsequent refactoring, redesign, or rewriting—depending on their complexity, age, and reuse in other projects. Refactoring includes reducing size, improving modularity, balancing cohesion and coupling, and so on. For instance, apply thorough unit testing with 100 percent CO coverage (statement coverage) to the modules. Investigate the details of those modules' complexity measurements to determine the redesign approach. Typically, the different complexity measurements will indicate the approach to follow.

Step 8: After the new project is finished, validate and improve the prediction model on the basis of postmortem studies with all collected defects and the population of a “real” criticality list. Then, compare the actual defect ranking with the predicted ranking. Investigate the reasons for deviations, and tune the implemented automatic classification approaches. Improve the screening rules to ensure that type-II prediction errors will be reduced the next time. (In type-II errors, uncritical components are misclassified as defect-prone components.)

continued on next page

continued from previous page

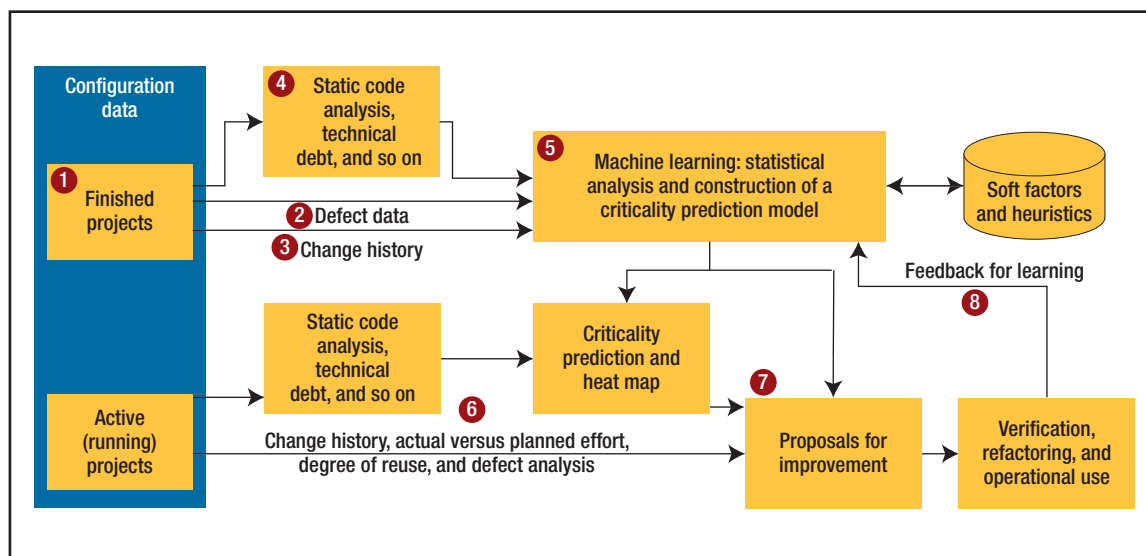


FIGURE A. Machine learning for criticality prediction of source code as implemented in this case study. The numbers refer to the steps described in the sidebar text.

THIS APPROACH'S EFFECTIVENESS

Criticality prediction doesn't aim to detect all defects. Instead, it aims to optimize resource allocation by focusing resources on areas with critical defects that would affect the delivered product's utility. We estimate the trade-off of the costs of applying complexity-based predictive quality models and the eventual code changes versus the improved quality, on the following basis:

- Effort—limited resources are assigned to the high-risk components.
- Effort—gray-box testing strategies are applied to only the high-risk components.
- Benefits—the risk assessment of changes is eased with code analysis on the basis of the affected or changed complexity.
- Benefits—fewer customers reported failures than in previous releases, and security and maintainability improved.

On the basis of the results from many of our client projects (and taking a conservative ratio of only 40 percent defects in critical components), we can calculate the

business case:

- On average, for each project, 20 percent of the modules were selected as the most critical (after coding).
- Those modules contained over 40 percent of all defects (up to release time).

In addition, we've determined that 60 percent of all defects theoretically can be detected until the end of unit testing. Also, defect correction with unit testing and static analysis costs 10 to 50 percent less than defect correction in subsequent testing activities. So, we calculate that developers can detect 24 percent of all defects early by investigating 20 percent of all modules more intensively, with over 10 percent reduced effort than with late defect correction. This yields at least a 20 percent cost reduction for defect correction.

The necessary tools, such as Coverity, Klocwork, Latix, Structure 101, SonarX, and SourceMeter, are off the shelf and account for even less per project. These criticality analyses provide numerous other benefits, such as the removal of specific code-related risks and defects that otherwise are hard to identify (for example, security flaws).

how to solve other problems he or she will have to deal with in the future.

Supervised learning includes *classification* algorithms, which take as input a dataset and the class of each piece of data so that the computer can learn how to classify new data. For example, the input might be a set of past loan applications with an indication of which of them went bad. On the basis of this information, the computer classifies new loan applications. Classification can employ logic regression, classification trees, support vector machines, random forests, artificial neural networks (ANNs), or other algorithms. ANNs are a major topic on their own; we discuss them in more detail later.

Regression algorithms predict a value of an entity's attribute ("regression" here has a wider sense than merely statistical regression). Regression algorithms include linear regression, decision trees, Bayesian networks, fuzzy classification, and ANNs.

Unsupervised Learning

In *unsupervised learning*, the training set contains data but no solutions; the computer must find the solutions on its own. This is like giving a student a set of patterns and asking him or her to figure out the underlying motifs that generated the patterns.

Unsupervised learning includes *clustering* algorithms, which take as input a dataset covering various dimensions and partition it into clusters satisfying certain criteria. A popular algorithm is *k*-means clustering, which aims to partition the dataset so that each observation lies closest to the mean of its cluster. Other clustering approaches include hierarchical clustering, Gaussian mixture models, genetic algorithms (in which the computer learns the

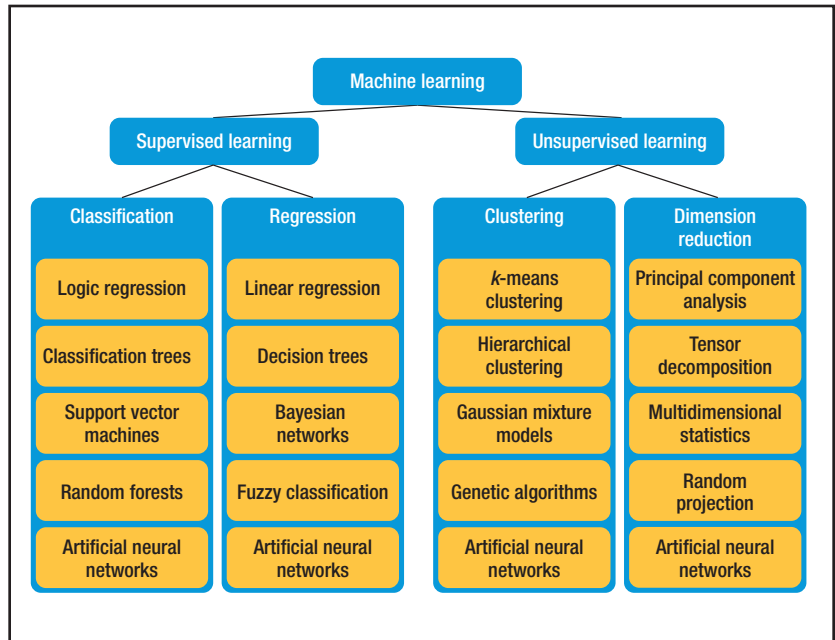


FIGURE 1. Machine-learning approaches. In machine learning, a computer first learns to perform a task by studying a training set of examples. The computer then performs the same task with data it hasn't encountered before.

best way for a task through artificial selection), and ANNs.

Dimensionality reduction algorithms take the initial dataset covering various dimensions and project the data to fewer dimensions. These fewer dimensions try to better capture the data's fundamental aspects. Dimensionality reduction algorithms include principal component analysis, tensor reduction, multidimensional statistics, random projection, and ANNs.

Essential Tools

Machine learning's popularity has brought along a wealth of tools. Most of them are open source, so users can easily experiment with them and learn how to use them. Table 1 compares some popular machine-learning tools.

The numerical and statistical communities are divided into two

camps: one that prefers R and one that prefers Python. Of course, any absolute division makes no sense. For a field as wide as machine learning, no single tool will do. The best a software engineer can do is to become acquainted with many different tools and learn which one is the most appropriate for a given situation.

That said, R is more popular with people with a somewhat stronger statistical background. It has a superb collection of machine-learning and statistical-inference libraries. Chances are, if you find a fancy algorithm somewhere and want to try it on your data, an implementation in R exists for it. R boasts the ggplot2 visualization library, which can produce excellent graphs.

Python is more popular with people with a computer science background. Although not made specifically for machine learning or

TABLE 1

Some popular machine-learning tools.

	Tool				
	Python	R	Spark	Matlab	TensorFlow
License	Open source	Open source	Open source	Proprietary	Open source
Distributed	No	No	Yes	No	No
Visualization	Yes	Yes	No	Yes	No
Neural nets	Yes	Yes	Multilayer perceptron classifier	Yes	Yes
Supported languages	Python	R	Scala, Java, Python, and R	Matlab	Python and C++
Variety of machine-learning models	High	High	Medium	High	Low
Suitability as a general-purpose tool	High	Medium	Medium	High	Low
Maturity	High	Very high	Medium	Very high	Low

statistics, Python has extensive libraries for numerical computing (NumPy), scientific computing (SciPy), statistics (StatsModels), and machine learning (scikit-learn). These are largely wrappers of C code, so you get Python's convenience with C's speed.

Although there are fewer machine-learning libraries for Python than there are for R, many programmers find working with Python easier. They might already know the language or find it easier to learn than R. They also find Python convenient for preprocessing data: reading it from various sources, cleaning it, and bringing it to the required formats. For visualization, Python relies on matplotlib. You can do pretty much everything on matplotlib, but you might discover you have to put in some effort. The seaborn library is built on top of it, letting you produce elegant visualizations with little code.

In general, R and Python work when the dataset fits in the com-

puter's main memory. If that's not possible, you must use a distributed platform. The most well-known is Hadoop, but Hadoop isn't the most convenient for machine learning. Making even simple algorithms run on it can be a struggle.

So, many people prefer to work at the higher level of abstraction that Spark offers. Spark leverages Hadoop but looks like a scripting environment. You can interact with it using Scala, Java, Python, or R. Spark has a machine-learning library that implements key algorithms, so for many purposes you don't need to implement anything yourself.

H2O is a relatively newer entrant in the machine-learning scene. It's a platform for descriptive and predictive analytics that uses Hadoop and Spark; you can use it with R and Python. It implements supervised- and unsupervised-learning algorithms and a Web interface through which you can organize your workflow.

A promising development is the

Julia programming language for technical computing, which aims at top performance. Because Julia is new, it doesn't have nearly as many libraries as Python or R. Yet, thanks to its impressive speed, its popularity might grow.

Strong commercial players include Matlab and SAS, which both have a distinguished history. Matlab has long offered solid tools for numerical computation, to which it has added machine-learning algorithms and implementations. For engineers familiar with Matlab, it might be a natural fit. SAS is a software suite for advanced statistical analysis; it also has added machine-learning capabilities and is popular for business intelligence tasks.

ANNs and Deep Learning

Cynics might roll their eyes, arguing that ANNs' resurgence is déjà vu. It's true; ANNs' fundamental components have been around for about half a century. However, it's also true

that you can now architect them in new ways. ANNs can be used across the spectrum of machine learning: classification, regression, clustering, and dimensionality reduction.

Innovations in ANN architectures and the availability of cheap computing resources to run ANNs has brought about the burgeoning of *deep learning*—using big ANNs to perform machine learning. Over the last few years, deep learning has chalked up headline-grabbing successes by beating humans in *Jeopardy!* and Go, learning to play arcade games, showing an uncanny capability to recognize images, performing automatic translation, and so on. Deep learning is particularly good at general tasks requiring the elicitation of higher-level, abstract concepts from the input data, which is what the many layers of an ANN excel at.

Deep learning is usually implemented through matrices, so working with it requires efficient matrix operations and manipulation. Usually the implementations are in C or C++, but designing ANNs at that level is unwieldy. Python programmers can use the Theano library to define ANNs, which are compiled to C code that's then compiled to machine language. Recently, Google released as open source its TensorFlow library for working with ANNs. You can interact with TensorFlow through a Python API. A C++ API is also available; although not as easy to use, it might give some performance benefits.


Before jumping on the deep-learning bandwagon, keep in mind that all machine-learning approaches lie on a spectrum based on the ease of interpreting their results. For example, classification trees produce rules that classify data. By reading

those rules, you can easily understand how a classification tree classifies data. ANNs don't produce anything their users can interpret. An ANN that classifies images doesn't produce any rules; the network itself embodies everything it has learned about image classification.

Many machine-learning books have a practical slant, aiming to introduce machine learning on a particular platform. As technologies quickly evolve, it's better to focus on getting a solid grasp of the fundamentals. After all, using a machine-learning platform isn't difficult; knowing when to use a particular algorithm and how to use it well requires quite a bit of background knowledge. Here are four popular books:

- T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning: Data Mining,*

Inference, and Prediction, 2nd ed., Springer, 2009.

- C.M. Bishop, *Pattern Recognition and Machine Learning*, Springer, 2006.
- K.P. Murphy, *Machine Learning: A Probabilistic Perspective*, MIT Press, 2012.
- E. Alpaydın, *Introduction to Machine Learning*, 3rd ed., MIT Press, 2014. 

PANOS LOURIDAS is an associate professor teaching algorithms and software at the Athens University of Economics and Business. He's also an active developer. Contact him at louridas@aueb.gr.

CHRISTOF EBERT is the managing director of Vector Consulting Services. He is on the *IEEE Software* editorial board and teaches at the University of Stuttgart and the Sorbonne in Paris. Contact him at christof.ebert@vector.com.



Selected CS articles and columns are also available for free at <http://ComputingNow.computer.org>.



ENGINEERING **SCIENCE**

The Perfect Blend

At the intersection of science, engineering, and computer science, *Computing in Science & Engineering (CISE)* magazine is where conversations start and innovations happen.

computing
in SCIENCE & ENGINEERING