# Machine Teaching

## A New Paradigm for Building Machine Learning Systems

Patrice Y. Simard, Saleema Amershi, David M. Chickering, Alicia Edelman Pelton, Soroush Ghorashi, Christopher Meek, Gonzalo Ramos, Jina Suh, Johan Verwey, Mo Wang, and John Wernsing, MICROSOFT RESEARCH

The current processes for building machine learning systems require practitioners with deep knowledge of machine learning. This significantly limits the number of machine learning systems that can be created and has led to a mismatch between the demand for machine learning systems and the ability for organizations to build them. We believe that in order to meet this growing demand for machine learning systems we must significantly increase the number of individuals that can teach machines. We postulate that we can achieve this goal by making the process of teaching machines easy, fast and above all, universally accessible.

While machine learning focuses on creating new algorithms and improving the accuracy of "learners", the machine teaching discipline focuses on the efficacy of the "teachers". Machine teaching as a discipline is a paradigm shift that follows and extends principles of software engineering and programming languages. We put a strong emphasis on the teacher and the teacher's interaction with data, as well as crucial components such as techniques and design principles of interaction and visualization.

In this paper, we present our position regarding the discipline of machine teaching and articulate fundamental machine teaching principles. We also describe how, by decoupling knowledge about machine learning algorithms from the process of teaching, we can accelerate innovation and empower millions of new uses for machine learning models.

Keywords: machine teaching, machine learning, programming, software engineering, concept decomposition, teaching principles, teaching process, democratization of machine learning, featuring

## 1 INTRODUCTION

The demand for machine learning (ML) models far exceeds the supply of "machine teachers" that can build those models. Categories of common-sense understanding tasks that we would like to automate with computers include interpreting commands, customer support, or agents that perform tasks on our behalf. The combination of categories, domains, and tasks leads to millions of opportunities for building specialized, high-accuracy machine learning models. For example, we might be interested in building a model to understand voice commands for controlling a television or building an agent for making restaurant reservations. The key to opening up the large space of solutions to is to increase the number of machine teachers by making the process of teaching machines easy, fast and universally accessible.

A large fraction of the machine learning community is focused on creating new algorithms to improve the accuracy of the "learners" (machine learning algorithms) on given labeled data sets. The machine teaching (MT) discipline is focused on the efficacy of the teachers *given the learners*. The metrics of machine teaching measure performance relative to human costs, such as productivity, interpretability, robustness, and scaling with the complexity of the problem or the number of contributors.

Many problems that affect model building productivity are not addressed by traditional machine learning. One such problem is concept evolution, a process in which the teacher's underlying notion of the target class is defined and refined over time [Kulesza et al. 2014]. Label noise or inconsistencies can be detrimental to traditional machine learning because it assumes that the target concept is fixed and is defined by the labels. In practice, concept definitions, schemas, and labels can change as new sets of rare positives are discovered or when teachers simply change their minds.

Consider a binary classification task for *gardening web pages* where the machine learner and feature set is fixed. The teacher may initially label botanical garden web pages as positive examples for the gardening concept, but then later decide that these are negative examples. Relabeling the examples when the target concept evolves is a huge burden on the teacher. From a teacher's perspective, concepts should be decomposable into sub-concepts and the manipulation of the relationship between sub-concepts should be easy, interpretable, and reversible. At the onset, the teacher could decompose gardening into sub-concepts (that include botanical gardens) and label the web page according to this concept schema.

In this scenario, labeling for sub-concepts has no benefits to the machine learning algorithm, but it benefits the teacher by enabling concept manipulation. Manipulation of sub-concepts can be done in constant time (i.e., not dependent on the number of labels), and the teacher's semantic decisions can be documented for communication and collaboration. Addressing concept evolution is but one example of where the emphasis on the teacher's perspective can make a large difference in model building productivity.

Machine teaching is a paradigm shift away from machine learning, akin to how other fields in programming language have shifted from optimizing performance to optimizing productivity with the notions of functional programming, programming interfaces, version control, etc. The discipline of machine teaching follows and extends principles of software engineering and languages that are fundamental to software productivity. Machine teaching places a strong emphasis on the teacher and the teacher's interaction with data, and techniques and design principles of interaction and visualization are crucial components. Machine teaching is also directly connected to machine learning fundamentals as it defines abstractions and interfaces between the underlying algorithm and the teaching language. Therefore, machine teaching lives at the interaction of the human-computer interaction, machine learning, visualization, systems and engineering fields. The goal of this paper is to explore machine learning model building from the teacher's perspective.

## 2 THE NEED FOR A NEW DISCIPLINE

In 2016, at one of Microsoft's internal conferences (TechFest) during a panel titled "How Do We Build and Maintain Machine Learning Systems?", the host started the discussion by asking the audience "What is your worst nightmare?" in the context of building machine learning models for production. A woman raised her hand and gave the first answer:

> "[...] Manage versions. Manage data versions. Being able to reproduce the models. What if, you know, the data disappears, the person disappears, the model disappears... And we cannot reproduce this. I have seen this hundreds of times in Bing. I have seen it every day. Like... Oh yeah, we had a good model. Ok, I need to tweak it. I need to understand it. And then... Now we cannot reproduce it. That is my biggest nightmare!"

To put context to this testimony, we review what building a machine learning model may look like in a product group:

(1) A problem owner collects data, writes labeling guidelines, and optionally contributes some labels.
(2) The problem owner outsources the task of labeling a large portion of the data (e.g., 50,000 examples).
(3) The problem owner examines the labels and may discover that the guidelines are incorrect or that the sampled examples are inappropriate or inadequate for the problem. When that happens, GOTO step 1.
(4) An ML expert is consulted to select the algorithm (e.g., deep neural network), the architecture (e.g., number of layers, units per layer, etc.), the objective function, the regularizers, the cross-validation sets, etc.

(5) Engineers adjust existing features or create new features to improve performance. Models are trained and deployed on a fraction of traffic for testing.
(6) If the system does not perform well on test traffic, GOTO step 1
(7) The model is deployed on full traffic. Performance of the model is monitored, and if that performance goes below a critical level, the model is modified by returning to step 1.

An iteration through steps 1 to 6 typically takes weeks. The system can be stable at step 7 for months. When it eventually breaks, it can be for a variety of reasons: the data distribution has changed, the competition has improved and the requirements have increased, new features are available and some old features are no longer available, the definition of the problem has changed, or a security update or other change has broken the code. At various steps, the problem owner, the machine learning expert, or the key engineer may have moved on to another group or another company. The features or the labels were not versioned or documented. No one understands how the data was collected because it was done in an ad hoc and organic fashion. Because multiple players with different expertise are involved, it takes a significant amount of effort and coordination to understand why the model does not perform as well as expected after being retrained. In the worst case, the model is operating but no one can tell if it is performing as expected, and no one wants the responsibility of turning it off. Machine learning "litter" starts accumulating everywhere. These problems are not new to machine learning in practice [Sculley et al. 2014].

The example above illustrates the fact that building a machine learning model involves more than just collecting data and applying learning algorithms, and that the management process of building machine learning solutions can be fraught with inefficiencies. There are other forms of inefficiencies that are deeply embedded in the current machine learning paradigm. For instance, machine learning projects typically consist of a single monolithic model trained on a large labeled data set. If the model's summary performance metrics (e.g., accuracy, F1 score) were the only requirements and the performance remained unchanged, adding examples would not be a problem even if the new model errs on the examples that were previously predicted correctly. However, for many problems for which predictability and quality control are important, any negative progress on the model quality leads to laborious testing of the entire model and incurs high maintenance cost. A single monolithic model lacks the modularity required for most people to isolate and address the root cause of a regression problem.

### 2.1 Definitions of machine learning and machine teaching

It is difficult to argue that the challenges discussed above are given a high priority in the world's best machine learning conferences. These problems and inefficiencies do not stem from the machine learning algorithm, which is the central topic of the machine learning field; they come from the processes that use machine learning, from the interaction between people and machine learning algorithms, and from people's own limitations.

To give more weight to this assertion, we will define the machine learning research field narrowly as:

*Definition 2.1 (Machine learning research).* Machine Learning research aims at making the *learner* better by improving ML algorithms.

This field covers, for instance, any new variations or breakthroughs in deep learning, unsupervised learning, recurrent networks, convex optimization, and so on.

Conversely, we see version control, concept decomposition, semantic data exploration, expressiveness of the teaching language, interpretability of the model, and productivity as having more in common with programming and human-computer interaction than with machine learning. These "machine teaching" concepts, however, are extraordinarily important to any practitioners of machine learning. Hence, we define a discipline aimed at improving these concept as:

*Definition 2.2 (Machine teaching research).* Machine teaching research aims at making the *teacher* more productive at building machine learning models.

We have chosen these definitions to minimize the intersection between the two fields and thus provide clarity and scoping. The two disciplines are complementary and can evolve independently. Of course, like any generalization, there are limitations. Curriculum learning [Bengio et al. 2009], for instance, could be seen as belonging squarely in the intersection because it involves both a learning algorithm and teacher behavior. Nevertheless, we have found these definitions useful to decide what to work on and what not to work on.

### 2.2 Decoupling machine teaching from machine learning

Machine teaching solutions require one or more machine learning algorithms to produce models throughout the teaching process (and for the final output). This requirement can make things complex for teachers. Different deployment environments may support different runtime functions, depending on what resources are available (e.g., DSPs, GPUs, FPGAs, tight memory or CPU constraints) or what has been implemented and green-lighted for deployment. Machine learning algorithms can be understood as "compilers" that convert the teaching information to an instance of the set of functions available at runtime. For example, each such instance might be characterized by the weights in a neural network, the "means" in K-means, the support vectors in SVMs, or the decisions in decision trees. For each set of runtime functions, different machine learning compilers may be available (e.g., LBFGS, stochastic gradient descent), each with its own set of parameters (e.g., history size, regularizers, k-folds, learning rates schedule, batch size, etc.)

Machine teaching aims at shielding the teacher from both the variability of the runtime and the complexity of the optimization. This has a performance cost: optimizing for a target runtime with expert control of the optimization parameters will always outperform generic parameter-less optimization. It is akin to in-lining assembly code. But like high-level programming languages, our goal with machine teaching is to reduce the human cost in terms of both maintenance time and required expertise. The teaching language should be "write once, compile anywhere", following the ISO C++ philosophy.

Using well-defined interfaces describing the inputs (feature values) and outputs (label value predictions) of machine learning algorithms, the teaching solution can leverage any machine learning algorithms that support these interfaces. We impose three additional system requirements:

(1) The featuring language available to the teacher should be expressive enough to enable examples to be distinguished in meaningful ways (a hash of a text document has distinguishing power, but it is not considered meaningful). This enables the teacher to remove feature blindness without necessarily increasing concept complexity.
(2) The complexity (VC dimension) of the set of functions that the system can return increases with the dimension of the feature space. This enables the teacher to decrease the approximation error by adding features.
(3) The available ML algorithms must satisfy the classical definition of learning consistency [Vapnik 2013]. This enables the teacher to decrease the estimation error by adding labeled examples.

The aim of these requirements is to enable teachers to create and debug any concept function to an arbitrary level of accuracy without being required to understand the runtime function space, learning algorithms, or optimization.

## 3 ANALOGY TO PROGRAMMING

In this section, we argue that teaching machines is a form of programming. We first describe what machine teaching and programming have in common. Next, we highlight several tools developed to support software development that we argue are likely to provide valuable guidance and inspiration to the machine teaching discipline. We conclude this section with a discussion of the history of the discipline of programming and how it might be predictive of the trajectory of the discipline of machine teaching.

### 3.1 Commonalities and differences between programming and teaching

Assume that a software engineer needs to create a stateless *target function* (e.g., as in functional programming) that returns value $Y$ given input $X$. While not strictly sequential, we can describe the programming process as a set of steps as follows:

(1) The target function needs to be specified
(2) The target function can be decomposed into sub-functions
(3) Functions (including sub-functions) need to be tested and debugged
(4) Functions can be documented
(5) Functions can be shared
(6) Functions can be deployed
(7) Functions need to be maintained (scheduled and unscheduled debug cycles)

Further assume that a teacher wants to build a target *classification* function that returns class $Y$ given input $X$. The process for machine teaching presented in the previous section is similar to the set of programming steps above. While there are strong similarities, there are also significant differences, especially in the debugging step (Table 1).

Table 1. Comparison of debugging steps in programming and machine teaching

| Debugging in programming | Debugging in machine teaching |
|---|---|
| (3) Repeat: | (3) Repeat: |
| (a) Inspect | (a) Inspect |
| (b) Edit code | (b) Edit/add knowledge (e.g., labels, features,...) |
| (c) Compile | (c) Train |
| (d) Test | (d) Test |

In order to strengthen the analogy between teaching and programming, we need a machine teaching *language* that lets us express these steps in the context of a machine learning model building task. For programming, the examples of languages include C++, Python, JavaScript, etc. which can be compiled into machine language for execution. For teaching, the language is a means of expressing teacher knowledge into a form that a machine learning algorithm can leverage for training. Teacher knowledge does not need to be limited to providing labels but can be a combination of

schema constraints (e.g., mutually exclusive labels for classification, state transition constraints in entity extraction[1]), labeled examples, and features. Just as new programing languages are being developed to address current limitations, we expect that new teaching languages will be developed that allow the teacher to communicate different types of knowledge and to communicate knowledge more effectively.

### 3.2 Programming paving the way forward

As we have illustrated in the previous sections, current machine learning processes require multiple people of different expertise and strong knowledge dependency among them, there are no standards or tooling for versioning of data and models, and there is a strong co-dependency between problem formulation, training and the underlying machine learning algorithms. Fortunately, the emerging discipline of machine teaching can leverage lessons learned from the programming, software engineering and related disciplines. These disciplines have developed over the last half century and addressed many analogous problems that machine teaching aims to solve. This is not surprising given their strong commonalities. In this section, we highlight several lessons and relate them to machine teaching.

*3.2.1 Solving complex problems.* The programming discipline has developed and improved a set of tools, techniques and principles that allow software engineers to solve complex problems in ways that allow for efficient, maintainable and understandable solutions. These principles include problem decomposition, encapsulation, abstraction, and design patterns. Rather than discussing each of these, we contrast the differing expectations between software engineers solving a complex problem and machine teachers solving a complex problem. One of the most powerful concepts that allowed software engineers to write systems that solve complex problems is that of decomposition. The next anecdote illustrates its importance and power.

We asked dozens of software engineers the following:

(1) Can you write a program that correctly implements the game Tetris?
(2) Can you do it in a month?

The answer to the first question is universally "yes". The answer to the second question varies from "I think so" to "why would it take more than 2 days?". The first question is arguably related to the Church-Turing thesis which states that all computable functions are computable by a Turing machine. If a human can compute the function, there exists a program that can perform the same computation on a Turing machine. In other words, given that there is an algorithm to implement the Tetris game, most respectable software engineers believe they can also implement the game on whatever machine they have access to and in whatever programming language they are familiar with. The answer to the second question is more puzzling. The state space in a Tetris game (informally the number of configurations of the pieces on the screen) is very large, in fact, far larger than can be examined by the software engineer. Indeed, one might expect that the complexity of the program should grow exponentially with the size of the representation of a state in the state space. Yet, the software engineers seem confident that they can implement the game in under a month. The most likely explanation is that they consider the complexity of the implementation and the debugging to be polynomial in both the representation of the state space and the input.

Let us examine how machine learning experts react to similar questions asked about teaching a complex problem:

(1) Can you teach a machine to recognize kitchen utensils in an image as well as you do?
(2) Can you do it in a month?

[1]In an address recognizer, we might want to require that the zip code appears after the state.

When these questions were asked to another handful of machine learning experts, the answers were quite varied. While one person answered "yes" to both questions without hesitation, most machine learning experts were less confident about both questions with answers including "probably", "I think so", "I am not sure", and "probably not". Implementing the Tetris game and recognizing simple non-deformable objects seem like fairly basic functions in either fields, thus it is surprising that the answers to both sets of questions are so different.

The goal of both programming and teaching is to create a function. In that respect, the two activities have far more in common than they have differences. In both cases we are writing functions, so there is no reason to think that the Church-Turing thesis is not true for teaching. Despite the similarities, the expectations of success for creating, debugging, and maintaining such function differ widely between software engineers and teachers. While the programming languages and teaching languages are different, the answers to the questions were the same for all software engineers regardless of the programming languages. Yet, most machine learning experts did not give upper bounds on how long it would take to solve a teaching problem, even when they thought the problem was solvable.

Software engineers have the confidence of being able to complete the task in a reasonable time because they have learned to decompose problems into smaller problems. Each smaller problem can be further decomposed until coding and debugging can be done in constant or polynomial time. For instance, to code Tetris, one can create a state module, a state transformation module, an input module, a scoring module, a shape display module, an animation module, and so on. Each of these modules can be further decomposed into smaller modules. The smaller modules can then be composed and debugged in polynomial time. Given that each module can be built efficiently, software engineers have confidence that they can code Tetris in less than a month's time.

It is interesting to observe that that the ability to decompose a problem is a learned skill and is not easy to learn. A smart student could understand and learn all the functions of a programming language (variables, arrays, conditional statements, for loops, etc.) in a week or two. If the same student was asked to code Tetris after two weeks, they would not know where to start. After 6 to 12 months of learning how to program, most software engineers would be able to accommodate the task of programming the Tetris game in under a month.

Akin to how decomposition brings confidence to software engineers[2] and an upper bound to solving complex problems, machine teachers can learn to decompose complex machine learning problems with the right tools and experiences, and the machine teaching discipline can bring the expectations of success for teaching a machine to a level comparable to that of programming.

*3.2.2 Scaling to multiple contributors.* The complexity of the problems that software engineers can solve has increased significantly over the past half century, but there are limits to the scale of problems that one software engineer can solve. To address this, many tools and techniques have been developed to enable multiple engineers to contribute to the solution of a problem. In this section, we focus on three concepts - programming languages, interfaces (APIs), and version control.

One of the key developments that enables scaling with the number of contributors is the creation of standardized programming languages. The use of a standardized programming language along with design patterns and documentation enables other collaborators to read, understand and maintain the software. The analog to programming languages for machine teaching is the expressions of a teacher's domain knowledge which include labels, features and schemas. Currently, there is no standardization of the programming languages for machine teaching.

[2]For similar reasons, the ability to decompose also bring confidence to professional instructors and animal trainers.

Another key development that enables scaling with the number of contributors is the use of componentization and interfaces, which are closely related to the idea of problem decomposition discussed above. Componentization allows for a separation of concerns that reduces development complexity, and clear interfaces allow for independent development and innovation. For instance, a software engineer does not need to consider the details of the hardware upon which the solution will run. For machine teaching, the development of clear interfaces for services required for teaching, such as training, sampling and featuring, would enable independent teaching. In addition, having clear interfaces for models, features, labels, and schemas enables composing these constituent parts to solve more complex problems, and thus, allowing for their use in problem decomposition.

The final development that enables scaling with the number of contributors is the development of version control systems. Modern version control systems support merging contributions by multiple software engineers, speculative development, isolation of bug fixes and independent feature development, and rolling back to previous versions among many other benefits. The primary role of a version control system is to track and manage changes to the source code rather than keeping track of the compiled binaries. Similarly, in machine teaching, a version control system could support managing the changes of the labels, features, schemas, and learners used for building the model and enable reproducibility and branching for experimentation while providing documentation and transparency necessary for collaboration.

*3.2.3 Supporting the development of problem solutions.* In the past few decades, there has been an explosion of tools and processes aimed at increasing programming productivity. These include the development of high-level programming languages, innovations in integrated development environments, and the creation of development processes. Some of these tools and processes have a direct analog in machine teaching, and some are yet to be developed and adapted. Table 2 presents a mapping of many of these tools and concepts to machine teaching.

Table 2. Mapping between programming and machine teaching

| Programming | Machine teaching |
|---|---|
| Compiler | ML Algorithms (Neural Networks, SVMs) |
| Operating System/Services/IDEs | Training, Sampling, Featuring Services, etc. |
| Frameworks | ImageNet, word2vec, etc. |
| Programming Languages (Fortran, Python, C#) | Labels, Features, Schemas, etc. |
| Programming Expertise | Teaching Expertise |
| Version Control | Version Control |
| Development Processes (specifications, unit testing, deployment, monitoring, etc.) | Teaching Processes (data collection, testing, publishing, etc.) |

### 3.3 The trajectory of the machine teaching discipline

We conclude this section with a brief review of the history of programming and how that might inform the trajectory of the machine teaching discipline. The history of programming is inexorably linked to the development of computers. Programming started with scientific and engineering tasks (1950s) with few programs and programming languages like FORTRAN that focused on compute performance. In the 1960s, the range of problems expanded to include management information systems and the range of programming languages expanded to target specific application domains (e.g.,

COBOL). The explosion of the number of software engineers led to the realization that scaling with contributors was difficult [Brooks Jr 1995]. In the 1980s, the scope of problems to which programming was applied exploded with the advent of the personal computer as did the number of software engineers solving the problems (e.g., with Basic). Finally, in the 1990s, another explosive round of growth began with the advent of web programming and programming languages like JavaScript and Java. As of writing this paper, the number of software engineers in the world is approaching 20 million!

Machine teaching is undergoing a similar explosion. Currently, much of the machine teaching effort is undertaken by experts in machine learning and statistics. Like the story of programming, the range of problems to which machine learning has been applied has been expanding. With the deep-learning breakthroughs in perceptual tasks in the 2010s (e.g., speech, vision, self-driving cars), there has been an incredible effort to broaden the range of problems addressed by teaching machines to solve the problems. Similar to the expanding population of software engineers, the advent of services like LUIS.ai[3] and Wit.ai[4] have enabled domain experts to build their own machine learning models with no machine learning knowledge. The discipline of machine teaching is young and in its formative stages. One can only expect that this growth will continue at an even quicker pace. In fact, machine teaching might be the path to bringing machine learning to the masses.

## 4 THE ROLE OF TEACHERS

The role of the teacher is to transfer knowledge to the learning machine so that it can generate a useful model that can approximate a concept. Let's define what we mean by this.

*Definition 4.1 (Concept).* A concept is a mapping from any example to a label value.

For example, the concept of a recipe web page can be represented by a function that returns zero or one, based on whether a web page contains a cooking recipe. In another example, an address concept can be represented by a function that, given a document, returns a list of token ranges, each labeled "address", "street", "zip", "state", etc. Label values for a binary concept could be "Is" and "Is Not". We may also allow a "Undecided" label which allows a teacher to postpone labeling decisions or ignore ambiguous examples. Postponing a decision is important because the concept may be evolving in the teacher's head. An example of this is in [Kulesza et al. 2014].

*Definition 4.2 (Feature).* A feature is a concept that assigns each example a scalar value.

We usually use feature to denote a concept when emphasizing its use in a machine learning model. For example, the concept corresponding to the presence or absence of the word "recipe" in text examples might be a useful feature when teaching the recipe concept.

*Definition 4.3 (Teacher).* A teacher is the person who transfers concept knowledge to a learning machine.

To clarify this definition of a teacher, the methods of knowledge transfer need to be defined. At this point, they include a) example selection (biased), b) labeling, c) schema definition (relationship between labels), d) featuring, and e) concept decomposition (where features are recursively defined as sub-models). The teachers are expected to make mistakes in all the forms of knowledge transfer. These teaching "bugs" are common occurrences.

[3] https://www.luis.ai/
[4] https://wit.ai/

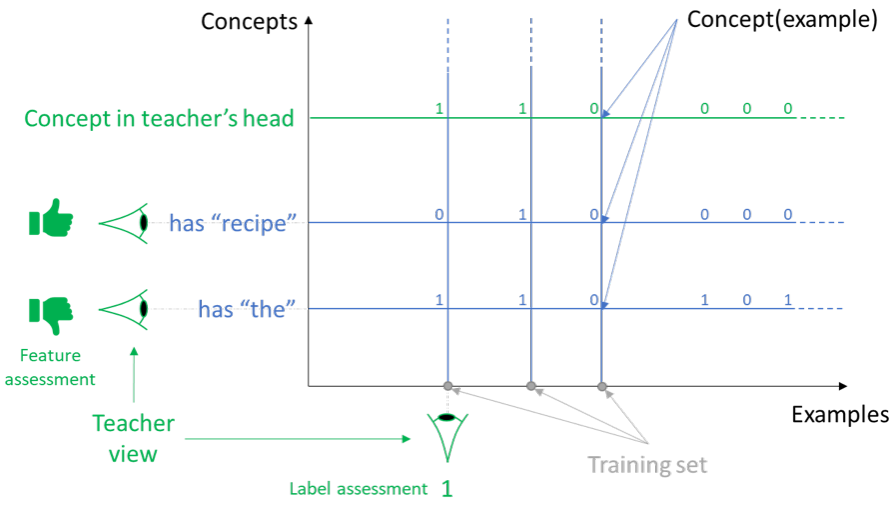


Fig. 1. Representation of examples and concepts. Each column represents an example and contains all concept values for that example. A teacher looks in that direction to "divine" a label. The teacher has access to feature concepts not available to the training set (it is part of the teaching power). However, the teacher does not know his/her own program. Each row represents a concept and contains the value of that concept for all examples. A teacher looks in that direction to "divine" the usefulness of a feature concept. A teacher can guess the values over the distributions of the test set (it is part of the teaching power). Features selected by the teacher looking horizontally are immune to over-training.

Figure 1 illustrates how concepts, labels, features, and teachers are related. We assume that every concept is a computable function of a representation of examples. The representation is assumed to include all available information about each example. The horizontal axis represents the (infinite) space of examples. The vertical axis represents the (infinite) space of programs or concepts. In computer science theory, programs and examples can be represented as (long) integers. Using that convention, each integer number on the vertical axis could be interpreted as a program, and each integer number on the horizontal axis could be interpreted as an example. We ignore the programs that do not compile and the examples that are nonsensical. We now use Figure 1 to refer to the different ways a teacher can pass information to a learning system.

*Definition 4.4 (Selection).* Selection is the process by which teachers gain access to an example that exemplifies useful aspects of a concept.

Teachers can select specific examples by filtering the set of unlabeled examples. By choosing these filters deliberately, they can systematically explore the space and discover information relevant to concepts. For example, a teacher may discover insect recipes while building a recipe classifier by issuing a query on "source of proteins". We note that uniform sampling and uncertainty sampling, which have no explicit input from a teacher, are likely of little use for discovering rare clusters of positive examples. Combinations of semantic filters involving trained models are even more powerful (e.g., "nutrition proteins" and low score with current classifier). This ability to find examples containing useful aspects of a concept enables the teacher to find useful features and provide the labels to train them. Furthermore, the selection choices themselves can be valuable documentation of the teaching process.

*Definition 4.5 (Label).* A label is a (example, concept value) pair created by a teacher in relation to a concept.

Teachers can provide labels by "looking at a column" in Figure 1. It is important to realize that the teachers do not know which programs are running in their heads when they evaluate the target concept values. If they knew the programs, they would transfer their knowledge in programmatic form to the machine and would not need machine learning. Teachers instead look at the available data of an example and "divine" its label. They do this by unconsciously evaluating sub-features and combining them to make labeling decisions. The feature spaces and the combination functions available to the teachers are beyond what is available through the training sets. This power is what makes the teachers valuable for the purpose of creating labels.

*Definition 4.6 (Schema).* A schema is a relationship graph between concepts.

When multiple concepts are involved, a teacher can express relationship between them. For instance, the teacher could express that the concepts "Tennis" and "Soccer" are mutually exclusive, or that concept "Tennis" implies the concept "Sport". These concept constraints are relationships between lines on the diagram (true across all examples). Separating knowledge captured by the schema from the knowledge captured by the labels allows information to be conveyed and edited at a high level. The implied labels can be changed simply by changing the concept relationship. For instance, "Golf" could be moved from being a sub-concept of "Sport" to being mutually exclusive or vice versa. Teachers can understand and change the semantics of a concept by reviewing its schema. Semantic decisions can be reversed without editing individual labels.

*Definition 4.7 (Generic feature).* A generic feature is a set of related feature functions.

Generic features are created by engineers in parametrizable form, and teachers instantiate individual features by providing useful and semantic parameters. For instance, a generic feature could be: "Log(1 + number of instances of words in list $X$ in a document)" and an instantiation would be setting $X$ to a list of car brands (useful for an automotive classifier).

Given a set of generic features, teachers have the ability to evaluate different (instantiated) features by looking along the corresponding horizontal lines in Figure 1. Given two features, the teachers can "divine" that one is better than the other on a large unlabeled set. For instance, a teacher may choose a feature that measures the presence of the word "recipe" over a feature that measures the presence of the word "the", even though the latter feature might yield better results on the training set. This ability to estimate the value of a feature over estimated distributions of the test set is essential to feature engineering, and is probably the most useful capability of a teacher. Features selected by the teacher in this manner are immune to over-training because they are created independently of the training set. Note the contrast to "automatic feature selection", which only looks at the training set and concept-independent statistics and is susceptible to over-training.

*Definition 4.8 (Decomposition).* Decomposition is the act of using simpler concepts to express more complex ones.

Whereas teachers do not have direct access to the program implementing their concept, they sometimes can infer how these programs work. Socrates used to teach by asking the right questions. The "right question" is akin to providing a useful sub-concept, whose value makes evaluating the overall concept easier. In other words, Socrates was teaching by decomposition rather than by examples. This ability is not equally available to teachers. It is learned. It is essential to scaling with complexity and with the number of teachers. It is the same ability that helps software engineers decompose functions into sub-functions. Software engineers also acquire this ability with experience. As in programming, teaching decompositions are not unique (in software engineering, switching from one decomposition to another is called refactoring).

The knowledge provided by the teacher through concept decomposition is high level and modular. Each concept implementation might provide its own example selection, labels, schema, and features. These can be viewed as documentation of interfaces and contracts. Each concept implementation may be a black box, but the concept hierarchy is transparent and interpretable. Concept decomposition is the highest form of knowledge provided by the teacher.

Now that we have defined some of the key roles of the (machine) teacher, we turn to the question of how do we meet the demand for them.

### Meeting the demand for teachers

We postulate that the right solution to satisfy the increasing demand for machine learning models is to increase the number of people that can teach machines these models. But how do we do that and who are they?

The current ML-focused work flows put the machine learning or data scientist on the driver's seat. While training more scientists is a way to increase the number of teachers, we believe that that is not the right path to follow. For starters, machine learning and data scientists are a scarce and expensive resource. Secondly, machine learning scientists can serve a better purpose inventing and optimizing learning algorithms. In the same way, data scientists are indispensable applying their expertise to make sense of data and transform it into a usable form.

The machine teaching process that we envision does not require the skills of a ML expert or data scientist. Machine teachers use their domain knowledge to pick the right examples and counterexamples for a concept and explain why they differ. They do this through an interactive information exchange with a learning system. It is within the ranks of the domain experts where we will find the large population of machine teachers that will increase, by orders of magnitude, the number of ML models used to solve problems. We can transform domain experts by making a machine teaching language universally accessible.

A key characteristic of domain experts is that they understand the semantics of a problem. To this point, we argue that if a problem's data does not need to be interpreted by a person to be useful, machine teaching is not needed. For example, problems for which the labeled data is abundant or practically limitless; e.g. Computer Vision, Speech Understanding, Genomics Analysis, Click-Prediction, Financial Forecasting. For these, powerful learning algorithms or hardware may be the better strategy to arrive at an effective solution. In other problems like the above, feature selection using cross validation can be used to arrive at a good solution without the need of a machine teacher.

There is nonetheless, an ever-growing set of problems for which machine teaching is the right approach; problems where unlabeled data is plentiful and domain knowledge to articulate a concept is essential. Examples of these include controlling Internet-of-Things appliances through spoken dialogs and the environment's context, or routing customer feedback for a brand new product of a start-up to the right department, building a one-time assistant to help a paralegal sift through hundreds of thousands of briefs, etc.

We aim at reaching the same number of machine teachers as there are software engineers, a set counted in the tens of millions. Table 3 illustrates the differences in numbers between machine learning scientists, data scientists, and domain experts. By enabling domain experts to teach, we will enable each domain experts to apply their knowledge to directly solve millions of meaningful, personal, shared, "one-off" and recurrent problems at a scale that we have never seen.

## 5 TEACHING PROCESS

A teaching or programming language can be applied in many different ways, some more effective than others. We propose the following principles for the language and process of machine teaching:

Table 3. Where to find machine teachers

| Potential teacher | Quantities | Characteristics |
|---|---|---|
| Machine learning experts | Tens of thousands | Has profound understanding of machine learning. Can modify a machine learning algorithm or architecture to improve performance. |
| Data Scientist / Analyst | Hundreds of thousands | Can analyze big data, detect trend and correlations using machine learning. Can train machine learning models on existing values to extract value for a business. |
| Domain expert | Tens of millions | Understands the semantics of a problem. Can provide examples and counter examples, and explain the difference between them. |

*Universal teaching language.* We do not rely on the power of specific machine learning algorithms. The teaching interface is the same for all algorithms. If a machine learning algorithm is swapped for another one, more teaching may be necessary, but the teaching language and the model building experience is not changed. Machine learning algorithms should be interchangeable. Conversely, the teaching language should be simple and easy to learn given the domain (e.g., text, signal, images). Ideally, we aim at designing an ANSI or ISO standard per domain. Teachers that speak the same language should be interchangeable.

*Feature completeness.* We assume that the feature language available to the teacher is "feature complete". This means that all the target concepts that a teacher may want to implement are "realizable" through a recursive composition of models and features. Feature completeness is the responsibility of the engineer installing the system, not the teacher's. The teacher can make the concept realizable with the following actions:

(1) If a teacher can distinguish two patterns belonging to two different classes in a meaningful way, there must be a feature available that can make an equivalent meaningful distinction. By adding such feature, the teacher can correct feature blindness errors.
(2) If the concept function does not belong to the class of functions currently available for the feature set, the teacher can add features to increase the function space until it contains the concept function. The added features can be trained models (with their own training set and features). This fixes errors resulting from lack of capacity.
(3) Ambiguous patterns and patterns that require features outside the feature language can be marked as "don't care" to avoid wasting features, labels, and the teacher's time on difficult examples. Areas of "don't care" are used as a coping mechanism to keep the realizability assumption despite the Bayes error rate.

*Rich and diverse sampling set.* At any time, the teacher has access a seemingly infinite amount of data that captures the richness and diversity of examples that will be seen in practice. This does not imply that the distribution of the sampling set matches the runtime environment (see below). Unlabeled data should be collected indiscriminately because the cost of storing data is negligible compared to the cost of teaching; we view selectively collecting only the data that

is meant to be labeled as both risky and limiting[5]. A rich and diverse data set allows the teacher to explore it to express knowledge through selection. It also allows the teacher to find examples that can be used to train sub-concepts that are more specific than the original concept. For instance, a teacher could decide to build classifiers for bonsai gardening (sub-concept) and botanical gardening (excluded concept) to be used as features to a gardening classifier. Examples labeled by teachers are kept forever because labels always retain some semantic value.

*Distribution robustness.* The assumption that the training distribution matches the test distribution is unrealistic in practice. The role of the teacher is to create a model that is correct for any example, regardless of the test distribution. Given our assumption of feature completeness and a rich and diverse sampling set, the result of a successful teaching process should be robust to different test distributions. Imagine programming a "Sort" function. We expect "Sort" to work regardless of the distribution of the data it is sorting. We have the same correctness expectation for teaching. Because the training data is discovered and labeled for the training set in an ad hoc way using filtering, distribution robustness is a critical assumption and we therefore favor machine learning algorithms that are robust to covariate shifts. Warning: having a mismatch between the two distributions complicates evaluation.

*Modular development.* Decomposition is a central principle of both programming and machine teaching. The machine teaching process should support the modular development of concept implementation. This includes the decomposition of concepts into sub-concepts, and the use of models as features for other models. We can achieve this by standardizing model and feature interfaces. Similar to a programming integrated development environment (IDE), within our teaching IDE, concept implementation is done through "projects" that are grouped into "solutions". Projects in a solution are trained together because their retraining can affect each other. Dependencies across different solutions are treated as versioned packages, which means that retraining a project in one solution does not affect a project in a different solution (the teacher must update the package reference to incorporate such changes). The modular development principle encourages the sharing of explicit concept implementations.

*Version control.* All teacher actions (e.g., labels, features, label constraints, schema and dependency graph, and even programming code if necessary) are equivalent to a concept "program". They are saved in the same "commit". Like programming code, the teacher's actions relevant to a concept are saved in a version control system, the size of the teacher's contributions is proportional to the number of actions, and different type of actions are kept in different files to facilitate merge operations between contributions from different teachers.

The combination of these principles suggests a teaching process that is different from the standard teaching process. The machine learning agnosticism implies that the machine learning expert can be left out of the loop. The realizability (or featuring completeness) assumption, implies that the engineers can be left out of the teaching loop as well. The engineers can update the data pipeline and the programming language, but neither are concept-dependent so the engineer is not in the teaching loop. These two principles imply that a single person with domain and teaching knowledge can own the whole process. The availability of a rich and diverse sampling set means that the traditional data collection for labeling step is not part of the concept teaching process. The distribution robustness principle allows the teacher to explore and label freely throughout the process without worrying about balancing classes or example types. Concept modularity and version control guarantee that a function created in a project is reproducible if all of its features are deterministic and that training is deterministic. The concept modularity principle enables interpretability

[5]For example, the collected set may not contain important examples that would otherwise be found via the machine teaching process.

**Algorithm 1:** A machine teaching process

```
repeat
    while training set is realizable do
        if quality criteria is met then
            exit
        end
        \\ Actively and semantically explore sampling set using concept based filters.
        Find a test error (i.e., an incorrectly predicted (example, label) pair);
        Add example to training set.;
    end
    \\ Fix training set error
    if training error is caused by labeling error(s) then
        Correct labeling error(s);
    else
        \\ Fix feature blindness. This may entail one or more of the following actions:
        Add or edit basic features;
        Create a new concept/project for a new feature (decomposition);
        Change label constraints or schema (high level knowledge);
    end
until forever;
```

and scaling with complexity. The interpretability comes from being able to explain what each sub-concept does by looking at the labels, features, or schema. Even if each sub-concept is a black box inside, their interface is transparent. The merge functionality in version control enables easy collaboration between multiple teachers.

Based on the above, we propose a skeleton for a teaching process in Algorithm 1. Note that this process is not unique.

Evaluating the quality criteria in a distribution-robust setting is difficult and beyond the scope of this paper. A simple criteria could be to pause when the teacher's cost or time invested reaches a given limit. Finding test error effectively is also difficult and beyond the scope of this paper. The idea is to query over the large sample set by leveraging query-specific teacher-created concepts and sub-concepts. The art is to maximize the semantic expressiveness of querying and the diversity of results. Uncertainty sampling is a trivial and uninteresting case (ambiguous examples are not useful for coming up with new decomposition concepts).

There are a few striking differences between the teaching process above and the standard model building process. The most important aspect is that it can be done by a single actor operating on the true distribution. Knowledge transfer from teacher to learner has multiple modalities (selection, labels, features, constraints, schema). The process is a never-ending loop reminiscent of Tom Mitchell's NELL [Carlson et al. 2010]. Their NELL system is always in a state of zero error on the training set. Capacity is increased on demand, so there is no need for traditional regularization because the teacher controls the capacity of the learning system by adding features only when necessary.

## 6 CONCLUSION

Over the past two decades, the machine learning field has devoted most of its energy to developing and improving learning algorithms. For problems in which data is plentiful and statistical guarantees are sufficient, this approach has paid off handsomely. The field is now evolving toward addressing a much larger set of simpler and more ephemeral problems. While the demand to solve these problems effectively grows, the access to teachers that can build corresponding

solutions is limited by their scarcity and cost. To truly meet this demand, we need to advance the discipline of machine teaching. This shift is identical to the shift in the programming field in the 1980s and 1990s. This parallel yields a wealth of benefits. This paper takes inspiration from three lessons from the history of programming. The first one is problem decomposition and modularity, which has allowed programming to scale with complexity. We argue that a similar approach has the same benefits for machine teaching. The second lesson is the standardization of programming languages: write once, run everywhere. This paper is not proposing a standard machine teaching language, but we enumerated the most important transferable, machine learning agnostic knowledge channels available to the teacher. The final lesson is the process discipline, which includes separation of concerns, and the building of standard tools and libraries. This addresses the same limitations to productivity and scaling with the number of contributors that plagued programming (as described in the "Mythical Man Month" [Brooks Jr 1995]). We have proposed a set of principles that lead to a better teaching process discipline. Some of the tools of programming, such as version control, can be used as is. Some of these principles have been successfully applied in services such as LUIS.ai and by product groups inside Microsoft such as Bing Local. We are in the early stages of building a teaching interactive development environment.

On a more philosophical note, the large monolithic system, as epitomized by deep learning, is a popular trend in artificial intelligence. We see this as a form of machine learning behaviorism. It is the idea that complex concepts can always be learned from just (input, output) pairs. Unsupervised learning can be used to create deep representations (a higher form of input) to make the task easier. If the result of using unsupervised learning is computed prior to the definition of the concept, the concept-specific learning is still done from just (input, output) pairs. This approach is suspicious from a function counting perspective because it is a form of behaviorism. We cannot learn to write Shakespeare quality play from (input, output) pairs if the input is a sequence of words because enumerating enough pairs to generalize them would take longer than the age of the universe. One may argue that there is a representation for which the task is doable using a behaviorist approach, but where does the representation come from? If it is concept independent, it does not pass the counting argument. If it is concept-dependent, does it need to be broken down? If a deep representation is built by optimizing the sampled data (labeled or unlabeled), how can we expect it to provide useful representation for concepts that are underrepresented? Large monolithic systems are very useful, but it is unlikely that such system can solve every problem.

In contrast, it is difficult to foresee a complexity limit to what concept can be learned with a teacher that has the ability and talent to break down that concept into sub-concepts. If the programming analogy holds, the Church-Turing thesis suggest the limit. Arguably, the emergence of intelligence may have been triggered more by teaching than from superior algorithms. Evolution, however, has a handicap when it comes to teaching: it can only optimize from the learner's perspective. The teacher does not benefit directly from teaching. That means that selection must build robustness to poor teachers to maximally benefit from teaching. That mechanism exists: when the concepts that are too simple (not composed enough) or too complex (not decomposed enough) for children to make progress, they have ways to let their teacher know (as any parent can attest). In that sense, they control the quality of teaching. This mechanism (e.g., children acting out), apparently does not give parents enough guidance to teach over long periods such as years. Jean Piaget theorized that children learn in stages (Piaget's theory of cognitive development). This is indisputably part of an encompassing learning algorithm. The learner may be optimized in ways we do not understand. But it is worth mentioning that this arrangement has profound effects on teaching parents. Without learning stages and children guiding teaching, the task of being a good parent teacher would be terrifying. Evolution has created a built-in mechanism to ensure robustness to poor teaching. For obvious reasons, "teacher robustness" is a long term pursuit of machine teaching. It may be an essential ingredient to artificial intelligence.

## ACKNOWLEDGMENTS

We would like to thank Jason Williams for his active support and contributions to Machine Teaching. Jason is the creator of the LUIS (Language Understanding Internet Services) project, a service for building language understanding models, based on the principles mentioned in this paper. We would like to thank Riham Mansour and the Microsoft Cairo team for co-building, maintaining, and improving the www.LUIS.ai service. Finally, we would like to thank Matthew Hurst and his team for building high-performing web page entity extractors leveraging our machine teaching tools. These entity extractors are deployed in Bing Local.

## REFERENCES

Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. 2009. Curriculum Learning. In *Proceedings of the 26th Annual International Conference on Machine Learning (ICML '09)*. ACM, New York, NY, USA, 41–48. https://doi.org/10.1145/1553374.1553380

Frederick P Brooks Jr. 1995. *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition, 2/E*. Pearson Education India.

Andrew Carlson, Justin Betteridge, Bryan Kisiel, Burr Settles, Estevam R. Hruschka, Jr., and Tom M. Mitchell. 2010. Toward an Architecture for Never-ending Language Learning. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence (AAAI'10)*. AAAI Press, 1306–1313. http://dl.acm.org/citation.cfm?id=2898607.2898816

Todd Kulesza, Saleema Amershi, Rich Caruana, Danyel Fisher, and Denis Charles. 2014. Structured labeling for facilitating concept evolution in machine learning. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 3075–3084.

D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, and Michael Young. 2014. Machine Learning: The High Interest Credit Card of Technical Debt. In *SE4ML: Software Engineering for Machine Learning (NIPS 2014 Workshop)*.

Vladimir Vapnik. 2013. *The Nature of Statistical Learning Theory*. Springer science & business media.