Machine learning 2.0
Engineering data driven AI products

Max Kanter
Feature Labs, Inc.
Boston, MA 02116
max.kanter@featurelabs.com

Benjamin Schreck
Feature Labs, Inc.
Boston, MA 02116
ben.schreck@featurelabs.com

Kalyan Veeramachaneni
MIT LIDS,
Cambridge, MA 02139
kalyanv@mit.edu

v0.1.0
Date: 2018-03-06

Abstract

ML 2.0: In this paper, we propose a paradigm shift from the current practice of creating machine learning models that requires months-long discovery, exploration and "feasibility report" generation, followed by re-engineering for deployment, in favor of a rapid 8 week long process of development, understanding, validation and deployment that can executed by developers or *subject matter expert*s (non-ML experts) using reusable `APIs`. It accomplishes what we call a "minimum viable data-driven model," delivering a ready-to-use machine learning model for problems that haven't been solved before using machine learning. We provide provisions for the refinement and adaptation of the "model," with strict enforcement and adherence to both the scaffolding/abstractions and the process. We imagine that this will bring forth a second phase in machine learning, in which *discovery* is subsumed by more targeted goals of delivery and impact.

1 Introduction

Attempts to embed machine learning-based predictive models to make products smarter, faster, cheaper, and more personalized will dominate activity in the technology industry for the foreseeable future. Existing applications range from financial services systems employing simple fraud detection models, to patient care management systems in intensive care units employing more complex models that predict events. Future ones will be fueled by the enormous number of *discoveries* made by applying machine learning to numerous data stores – medical (see [1], [9]), financial and others – which are themselves evident from the sheer number of research articles, news articles, blogs, and data science competitions spearheaded by the folks making these *discoveries*. But it is arguable how many of these predictive models have actually been deployed, or how many have been effective and are serving their intended purpose of saving costs, increasing revenue or enabling better experiences. [1]

To explicate this, we highlight a few important observations about how machine learning and AI systems are currently built, delivered and deployed [2]. In most cases, development of these models: relies on first making the *discovery* from the data; uses a historically defined and deeply entrenched workflow; and finally

[1] The authors of [8] offer one perspective on the challenges of deploying and maintaining a machine learning model, and others in the tech industry have highlighted these challenges as well. The core problem, however, goes deeper than deployment challenges.

[2] These are our gathered from our experiences. We have created, evaluated, validated, published and delivered machine learning models for data from BBVA, ACCENTURE, KOHLS, NIELSEN, MONSANTO, JAGUAR AND LAND ROVER, EDX, MIMIC, GE, DELL among others. Additionally, we have participated using our automation tools in numerous publicly held data science competitions on KAGGLE and others. We are also a part of the MIT team in the DARPA D3M initiative. Development of the automation system, Featuretools is funded under DARPA. The findings, opinions expressed here are of ours alone and do not represent any entity we are involved or work with.

struggles to find the functional interplay between robust software engineering practice and the complex landscape of mathematical concepts required to build these models We call this state of machine learning **ML 1.0**, where the focus is on *discovery*. [3]

The "discovery first" paradigm: Most of the products in which we are now trying to embed intelligence have already been collecting data to enable their normal, day-to-day functioning, and machine learning or predictive modeling is usually an afterthought. With machine learning models we attempt to predict a future outcome or predict "what a human would say?" upon seeing the evidence in the form of the data, so there is uncertainty as to whether that is even is possible, given the data at hand.

This is unlike software engineering projects that develop or add a new feature to the product: in these cases, the end outcome is deterministic and visible, and allows designers, architects, and managers to make a plan, establish a workflow, release the feature, and manage it.

In contrast, a new machine learning project usually starts with a *discovery* phase, in which attempts are made to answer the quintessential question: *"Is this predictable using our data?*. If yes, with how much accuracy, and what variables mattered?; Which one of the numerous modeling possibilities worked better?; If the data happens to be temporal, which one of the numerous time series models (with latent states) can model it best?; What explainable patterns did we notice in the data and what models surfaced those patterns better?; and so on, in an endless list.

It is not uncommon to have dozens of research papers be written about the same prediction problem - each with a slightly different variation in how any of the questions above are answered. A recently established prediction problem - "predicting dropout in Massive Open Online Courses, to enable interventions" resulted in at least 100 research papers- written in a span of 4 years[4], and a competition at a Tier 1 data mining conference KDD - KDDCup 2015 [10, 7, 4, 3].

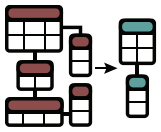
The expectation is that once this question has been answered, a continuously working predictive model can be developed, integrated with the product and put to use rather quickly. Perhaps this expectation would not be that unrealistic were it not for the deeply entrenched workflow generally used for *discovery*, as we will next describe.

A "deeply entrenched workflow": Developing and evaluating machine learning models for industrial-scale problems predates the emergence of internet-scale data. Our observations of machine learning in practice over the past decade, as well as our own experiences developing predictive models, have enabled us to delineate a number of codified steps involved in these projects. These steps are shown in the Table 1. Surprisingly, not much has changed as to how we approach a data store - much like a feasibility study or a research paper. To wit, compare the structure of the two papers, one written in 1994 - [2] and two latest papers written in 2017 - [1, 9]. *So what is the problem?*: The generalized codified steps shown in the Table 1 represent a good working template, but the workflow under which they are executed carries with it problems that are now entrenched. In Figure 3, we depict how these steps are executed as in three disjoint parts. Tools, collaborative systems and other recent systems stay within one of these parts thus providing ability to accelerate discovery - when data is ready to be used in that part. We present our detailed commentary on the current state of applied machine learning in Section 6.

ML 2.0: Delivery and impact: In this paper, we propose a paradigm shift from the current practice of creating machine learning models that requires months-long discovery, exploration and "feasibility report" generation, followed by re-engineering for deployment, in favor of a rapid 8 week long process of development, understanding, validation and deployment that can executed by developers or *subject matter expert*s (non-ML experts) using reusable `APIs`. It accomplishes what we call a "minimum viable data-driven model," delivering a ready-to-use machine learning model for problems that haven't been solved before using machine learning.

[3] Throughout this paper, our core focus is on data that is temporal, multi entity, multi table, relational (and/or transactional data). In most cases we are attempting to predict using a machine learning model, and in some cases ahead of time.

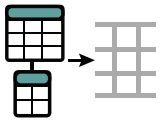
[4] These papers are published in premier AI venues - NIPS, IJCAI, KDD, and AAAI. One of our first work in predictive modeling for weblog data in 2013-14 focused on this very problem



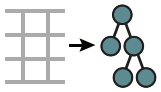
1. Extract relevant data subset
Data warehouses (now largely centrally organized) contain several data elements that may not be related to the predictive problem one is trying to solve. In this step, a subset of tables, fields, are first determined. Second, data may span a large time period. Depending upon the problem to be solved, a specified time period is selected. Based on these two a number of filters are applied and the resulting data is passed on to Step 2.

2. Formulate problem, assemble training examples
Developing a model that can predict future requires us to find examples in the past to learn from. Thus, a typical predictive model development involves first defining the outcome one wants to predict, and then finding the past occurrences of that outcome, that could be used to learn a model from.

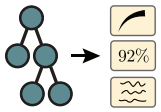
3. Prepare data
To train a model, we use the data retrospectively, emulate the prediction scenario, that is, we learn a model using data prior to the occurrence of the outcome, and evaluate its ability to predict the outcome. As a result this requires careful annotation of what data elements can be used for modeling. In this step, largely these annotations are added and the data is filtered to create dataset ready for machine learning.



4. Engineer features
For each training example, given the usable data, one computes features (aka variables) and creates a machine learning ready matrix. Each column in the matrix is a feature, the last column is the label, and each row is a different training example.



5. Learn a model
Given the feature matrix, in this step a model; either classifier (if the outcome is categorical) or regressor (if the outcome is continuous) is learnt. Numerous models are evaluated and selected based on a user specified evaluation metric.



6. Evaluate the model and report
Once a modeling approach is selected and trained, it is evaluated using the training data and a number of metrics like precision, recall are reported. Additionally, one evaluates the impact of variables on the predictive accuracy.

Table 1: Steps to making a *discovery* using machine learning from a data warehouse. This process does not account for deployment or define a robust software engineering practice, testing of models. The end result of the process is to generate a report, or a research paper. The approach is further broken down into silos in a typical workflow underwhich it is executed, presented in Figure 3.

We posit that any system claiming to be **ML 2.0** should deliver on the key goals and requirements listed in Figure 1. Each of these steps is important for different reasons: (1) ensures that we are bringing new intelligence services to life, (2) requires that the system considers the entire process and not just one step, and (3), (4) and (5) guarantee model's deployment and use.

(1) **subsume discovery**

deliver a machine learning model M for a problem that has not been solved before, and for which no prior solution exists or no prior steps or partial steps have been taken. The system subsumes discovery by enabling generation of a report of accuracy metrics, feature importances - all things originally done as part of *discovery*;

(2) **be end-to-end**

start with data D from the warehouse with several tables containing data for several entities and several years up to a current time t, and enable an developer to execute **all** steps 1 through 5, build and validate a model M;

(3) **enable confidence**

enable testing and validation of the end-to-end system under different simulated scenarios and conditions, and establish developer confidence;

(4) **provide deployable code**

enable deployment of the model at t for making predictions at $T > t$ by providing code that carries new data from its raw shape to predictions; and

(5) **provide provisions for updates**

provide provisions for updating the model, re- assessing/validating and re-deploying it.

Figure 1: Goals and requirements for **ML 2.0**

All of this should:

– happen with minimal amount of coding (possibly with simple `API` calls);

– use the same abstractions/software for (1), (2) and (3) - (5);

– take minimal amount of time to deliver the first version of the model;

– require minimal amount of manpower with or without "machine learning expertise."

These ensure reduction in design complexity, speed in bringing new AI applications to life and democratization of AI. We imagine that this will bring forth a second phase in artificial intelligence, in which the elevation of *discovery* is subsumed by more targeted goals of delivery and impact.

2 The path that lead to ML 2.0

For the last decade, the demand for predictive models has been growing at an increasing rate. As long-time data scientists we have found that our biggest challenge typically isn't creating accurate solutions for incoming prediction problems, but rather the time it takes as to build an end-to-end model and being able to match the supply of expertise to the incoming demand. Even more, we have been frequently disappointed when we see that our models aren't deployed. ML 2.0 is the result of a series of work to automate different human intensive parts of creating and deploying machine learning solutions.

In 2013, we initially focused our efforts on the problem of selecting a machine learning method and then tuning its hyperparameters. This eventually resulted in Auto-Tuned Models (ATM)[11][5], which takes advantage of cloud-based computing to perform a high-throughput search over modeling options, and find the best possible modeling technique for a particular problem[6]. While ATM helped us quickly build more accurate models, we couldn't take advantage of it until we had our data in the form a feature matrix. Most real-world use cases we worked on didn't not start in such a form, so we had to go earlier in the data science process.

This lead us to explore automating feature engineering, or the process of using domain specific knowledge to extract predictive patterns from raw dataset. Unlike most data scientists who work in a single domain, we worked with a wide range of industries. This gave us the unique opportunity to develop innovative solutions to use with the diverse problems we faced. Across industries like finance, education, and health care, we started to see commonalities in how features were constructed. These observations led to the creation of the Deep Feature Synthesis (DFS) algorithm for automated feature engineering [5].

We put DFS to the test against human data scientists to see if it accomplished the goal of saving us significant time while preparing raw data. We used DFS to compete in 3 different world data science competitions and found that we could build models in a fraction of the time of the human competitors, while achieving similar predictive accuracies.

With this success we took our automation algorithms to industry. Now that we could quickly build accurate models for a raw dataset after we were provided specific problem, a natural new automation question arose – "how do we figure out what problem to solve?". This question was particularly pertinent to companies and teams new to machine learning who frequently had a grasp of the high level business problem, but struggled translating it to a specific prediction problem.

We discovered that there was no existing process for systematically defining prediction problem. In fact, by the time labels reached data scientists they would typically be reduced to a list of true / false values, with no annotation about how they came about. Even if a data scientist had access to the code that extracted the labels, the code was implemented such that it could not be tweaked based on domain expert feedback or reused for different problems. To remedy this, we started to explore "prediction engineering" or the process of defining prediction problems in a structured way. In a 2016 paper [6], we laid out the Label Segment Featurize (LSF) abstraction and associated `traversal` algorithm to search for labels.

As we explored prediction engineering and defined the LSF abstraction, the concept of `time` started to play a huge role. Predictive tasks are intricately tied to time because we had simulate past predictive scenarios and carefully extract training examples preventing label leakage while training machine learning models. This brought in perhaps the most important set of requirements - to provide a way to annotate every data point with the timestamp when it occurred. Subsequently, we revisited each data processing algorithm we wrote for feature engineering, prediction engineering, or preprocessing to accept time periods where data was valid to be used. To address this, we introduced a simple yet powerful notion of `cutoff_time`. Simply put, when specified for a data processing block, any data that "occurred" beyond `cutoff_time` cannot be used by it.

[5] https://github.com/HDI-Project/ATM

[6] ATM was published and open source in 2017 to aid in the release of ML 2.0, but was originally developed in 2014

At the same time it became clear that we needed meta data about the data to do automatic processing and to maintain consistency through out a modeling endeavor. We defined a relational representation for pandas data frames called Entityset and started to define operations over it (described in Section 3.1). To enable easy access to the meta data we defined a `metadata.json`. We also started to define a more concrete way to organize the meta information about the processing done on the data -from a predictive modeling standpoint - which led to the development of `model_provenance.json`.

In our first pass through creating each of these automation approaches, we focused on defining the right abstractions, process organization and algorithms because of their foundational role in data science automation. We knew from the beginning the right abstractions would enable us to write code that could be reused across development, deployment and domains. This not only increases the rate at which we build models, but increases the likelihood of deployment by enabling the involvement of those subject matter expertise. As we have applied our new tools to industrial-scale problems over the last three years, these abstractions and algorithms have transformed into production-ready implementations. With this paper, we are formally releasing multiple tools and data, model organization schemas. They are `Featuretools`, `Entityset`, `metadata.json`, `model_provenance.json`, `ATM`.

In the sections below, we show how abstractions, algorithms, and implementations all come together to enable ML 2.0.

3 An ML 2.0 system

In **ML 2.0**, users of the system starts with raw untapped data residing in the warehouse and a rough idea of the problem they wants to solve. The user formulates or specifies a prediction problem, follows steps 1-5, and delivers a stable, tested and validated, deployable model in a few weeks - in addition to a report that documents the *discovery*.

Though this workflow seems challenging and unprecedented, we are motivated by recent developments in proper abstractions, algorithms, and data science automation tools that lend themselves to the restructuring of the entire predictive modeling process. In Figure 2, we provide an end-to-end recipe for **ML 2.0**. We next describe each step in detail, including its input and output, the parts that are automated, and the hyperparameters that allow developers to control the process. We highlight both what the human contributes and what the automation enables, illustrating how they form a perfect symbiotic relationship. In an accompanying paper, we present the first industrial use case that follows this paradigm, as well as the successfully deployed model.

3.1 Step 1 $\rightarrow$ Form an Entityset (Data organization)

An *Entityset* is an unified `API` for organizing and querying relational datasets to facilitate building predictive models. By providing a single `API`, an Entityset can be reused throughout the life cycle of a modeling project and reduce the possibility of human error while manipulating data. An Entityset is integrated with the typical industrial-scale data warehouse or data lake that contains multiple tables of information, each with multiple fields or columns.

The first, and perhaps the most human-driven, step of a modeling project is to identify and extract the subset of tables and columns that are relevant to the prediction problem at hand from the data source. Each table added to the Entityset is called an entity and has at least one column, called the `index`, that uniquely identifies each row. Other columns in an entity can be used to relate instances to another entity similar to a foreign-key relationship in a traditional relational database. An Entityset further tracks the semantic variable types of the data (e.g the underlying data maybe be stored as an float, but in context it is important to know that data represents a latitude or longitude).

In predictive modeling, it is common to have at least one entity that changes to over time as new information become available. This data is appended as a row to the entity it is an instance of and must

contain a column which stores the value for the time that particular row became available. This column is known as the `timeindex` for the entity and is used by automation tools to understand the order in which information became available for modeling. Without this functionality, it would not be possible to correctly simulate historical states of data while training, testing, and validating the predictive model.

The `index`, `timeindex`, relationships, and semantic variable types for each entity in an Entityset are documented in **ML 2.0** using a `metadata.json`[7]. After extracting and annotating the data, an Entityset provides the interface to this data for each of the modeling steps ahead.

The open source library `Featuretools` offers an in-memory implementation of the Entityset API in python to be used by humans and automated tools [8]. `Featuretools` represents each entity as a Pandas DataFrame, provides standardized access to metadata about the entities, and offers an existing list of semantic types for variables in the dataset. It offers additional functionality that is pertinent to predictive modeling including:

– querying by time - select a subset of data across all entities up until a point in time (uses time indices)

– incorporating new data - append to existing entities as new data becomes available or is collected

– checking data consistency - verify new data matches old data to avoid unexpected behavior

– transforming the relational structure - create new entities and relationships through normalization (i.e create a new entity with repeated information from an existing entity and add a relationship)

3.2 Step 2 → Assemble training examples (Prediction engineering)

To learn an ML model that can predict a future outcome, past occurrences of that outcome are required in order to train it. In most cases, defining which outcome one is interested in predicting is itself a process riddled with choices. For example:

> *Consider a developer maintaining an online learning platform. He wants to deploy a predictive model predicting when a student is likely to drop out. But dropout could be defined in several ways: as "student never returns to the website," or "student stops to watch videos," or "student stops submitting assignments."*

Labeling function: Our first goal is to provide a way to describe the outcome of interest easily. This is achieved by allowing the developer to write a *labeling* function given by

$$\texttt{label}, \texttt{cutoff\_time} = \texttt{f}(\texttt{E}, \texttt{e}_\texttt{i}, \texttt{timestamp}, \texttt{hparams}) \tag{1}$$

where, $\texttt{e}_\texttt{i}$ is the id of the i^{th} instance of the entity (i^{th} student, in the example above), `E` is Entityset, and `timestamp` is the time point – at which we are evaluating whether the outcome happened – *in future*. It is worth emphasizing that this outcome can be a complex function that uses a number of columns in the data. The output of the function is the `label` - which the predictive model will learn to predict using data up until `cutoff_time`.

Search: Given the *labeling* function, the next task it to search the data to find past occurrences. One can simply iterate over multiple instances of the entity and at different time points and call 1 in each iteration to generate a label.

A number of conditions are applied to this search. Having executed this search for several different problems across many different domains, we were able to precisely and programatically describe this process

[7] Documentation is available at https://github.com/HDI-Project/MetaData.json

[8] Documentation is available at https://docs.featuretools.com/loading_data/using_entitysets.html

parameter	description
`prediction_window`	the period of time to look for the outcome of interest
`lead`	the amount of time before the prediction window to make a prediction
`gap`	the minimum amount of time between successive training examples for an single instance
`examples_per_instance`	maximum number of training examples per entity instance
`min_training_data`	the minimum amount of training required before making a prediction

Table 2: Hyperparameters for the prediction engineering

and expose all of the hyperparameters that a developer may want to control during this search. Table **??** presents a subset of these hyperparameters, and we refer an interested reader to [6] for a detailed description of these and other search settings.

Given the *labeling* function, the settings for search, and the Entityset, a search algorithm searches for past occurrences across all instances of an entity and returns a list of three tuples called label-times:

$$\texttt{e}_{1\ldots\texttt{n}}, \texttt{label}_{1\ldots\texttt{n}}, \texttt{cutoff\_time}_{1\ldots\texttt{n}} = \texttt{search\_training\_examples}(\texttt{E}, \texttt{f}(.), \texttt{hparams}) \quad (2)$$

where e_i is the i^{th} instance of the entity, $\texttt{label}_\texttt{i}$ is a binary 1 or 0 (or multi category depending upon the *labeling*) representing whether or not the outcome of interest happened for the i^{th} instance of the entity, and $\texttt{cutoff\_time}_i$ represents the time point after which $\texttt{label}_\texttt{i}$ is known. A list of these tuples provide the possible training examples. While this process is applied in any predictive modeling endeavor, be it in healthcare, educational data mining, or retail, there have been no prior attempts made to abstract this process and provide widely applicable interfaces.

Subject matter expert	$\rightarrow$	implements the *labeling* function and sets the hyperparameters.
Automation	$\rightarrow$	searches for and compiles a list of training examples that satisfy a number of constraints.

3.3 Step 3 $\rightarrow$ Generate features (Feature engineering)

The next step in machine learning model development involves generating features for each training example. Consider:

> *At this stage, the developer has a list of those students who dropped out of a course and those who did not, based on his previously specified definition of dropout. He posits that the amount of time a student spends on the platform could predict whether he is likely to drop out. Using data prior to the* `cutoff_time`*, he writes software and computes the value for this feature.*

Features, a.k.a variables, are quantities derived by applying mathematical operations to the data associated with an entity-instance – *but only prior to the* `cutoff_time` *as specified in the label_times*. This step is usually done manually and takes up a lot of developer time. Our recent work enables us to automate this process by allowing developer to specify a set of hyperparameters for an algorithm called Deep Feature Synthesis [5]. The algorithm exploits the relational structure in the data and applies different statistical aggregations at different levels [9] [10].

[9] An open source implementation of this is available via www.featuretools.com.

[10] A good resource is at: https://www.kdnuggets.com/2018/02/deep-feature-synthesis-automated-feature-engineering.html

hyperparameter	description
`target_entity`	entity in Entityset to create features for
`training_window`	amount of historical data before `cutoff_time` to use to calculate features
`aggregation_primitives`	reusable functions that create new features using data at the intersection of entities
`transform_primitives`	reusable functions that create new features from existing features within an entity
`ignore_variables`	variables in an Entityset that shouldn't be used for feature engineering

Table 3: Hyperparameters for feature engineering

$$\texttt{feature\_list} = \texttt{create\_features}(\texttt{E}, \texttt{hparams}) \tag{3}$$

$$\texttt{feature\_matrix} = \texttt{calculate\_feature\_matrix}(\texttt{E}, \texttt{e}_{\texttt{i...n}}, \texttt{cutoff\_time}_{\texttt{i...n}}, \texttt{label}_{\texttt{i...n}}, \texttt{feature\_list}, \texttt{hparams}) \tag{4}$$

Given the Entityset and hyperparameter settings, the automatic feature generation algorithm outputs a `feature_list` containing a list of feature descriptions defined for the `target_entity` using `transform_primitives`, `aggregate_primitives`, and `ignore_variables`. These definitions are passed to 4 which generates a matrix where each column is a feature and each row pertains to an entity-instance e_i at the corresponding $cutoff\_time_i$ and `training_window`. The format of the `feature_matrix` is also shown in Figure 2. The `feature_list` is stored as a serialized file for use in deployment. Users can also apply a feature selection method at this stage to reduce the number of features.

Subject matter expert	→	guides the process by suggesting which primitives to use or variables to ignore, as well as how much historical data to use for calculating the features.
Automation	→	suggests the features based on the relational structure and precisely calculating the features for each training example - only within the allowed window `cutoff_time`- `training_window` - `cutoff_time`.

3.4 Step 4 → Generate a model `M` (Modeling and operationalization)

Given the `feature_matrix` and its associated labels, the next task is to learn a model. The goal of training is to use examples learn a model that generalize well to the future, which will be evaluated using a validation set. Typically, a model is not able to predict accurately for all the training examples and must make trade offs. While the training algorithm optimizes a *loss* function, metrics like - fscore, aucroc, recall, precision - are used for model selection and evaluation on a validation set. These metrics are derived solely from counting how many correct and incorrect `predictions` are made within each class. In practice, we find these measures inadequate to capture the domain specific needs. Consider for example:

> *In fraud prediction, consider a developer comparing two models,* $\texttt{M}_1$ *and* $\texttt{M}_2$. $\texttt{M}_1$ *while having the same false positive rate, had a 3% higher true positive rate. That is, it was able to detect more frauds. When fraud goes undetected the bank has to reimburse the customer, so the developer decides to measure the financial impact by adding the amount for the fradulent transactions that were missed by* $\texttt{M}_1$ *and* $\texttt{M}_2$ *respectively. He finds that* $\texttt{M}_1$ *actually performs worse. Ultimately, the*

parameter	description
methods	list of machine learning methods to search
budget	the maximum amount of time or number of models to search
automl_method	path to file describing the automl technique to use for optimization

Table 4: Hyperparameters for model search process

count of how many frauds did $\texttt{M}_2$ *miss does not matter, because missing a fraudulent transaction that is worth $10 is far less costly than missing a $10,000 transaction.*

This can be handled by using a domain specific cost function, $g(.)$, which the developer must implement.
Cost function $g(.)$: A cost function, given `predictions` and true `labels` and the Entityset, calculates the domain-specific evaluation of the model's performance. Its abstraction is specified as:

$$\texttt{cost} = \texttt{g}(\texttt{E}, \texttt{predictions}, \texttt{labels}) \tag{5}$$

Search for a model: Given the cost function, we can now search for the best possible model. An overwhelming number of possibilities exist for M. The search space includes all possible methods. In addition, each method contains hyperparameters, and choosing certain hyperparameters may lead to another set of hyperparameters that must then be chosen in turn. For example:

A developer is ready to finally learn a model $\texttt{label} \leftarrow \texttt{M}(\texttt{X})$. *He has to choose between numerous modeling techniques including* `svm`*,* `decisiontrees`*, and* `neuralnetworks`*. If* `svm` *is chosen, s/he must then choose the* `kernal`*, which could be* `polynomial`*,* `rbf`*, or* `Gaussian`*. If* `polynomial` *is chosen, then he has to choose another hyperparameter– the value of the degree.*

To fully exploit the search space, we designed an approach in which humans fully and formally specify the search space by creating a `json` specification for each modeling method (see Figure 4 for an example) and the automated algorithm searches the space for the model that optimizes the `cost` assessed using predictions. [11][11]

$$\texttt{M} = \texttt{search\_model}(\texttt{g}(.), \texttt{feature\_matrix}, \texttt{labels}, \texttt{cutoff\_times}, \texttt{hparams}) \tag{6}$$

To search for a model, based on the cost function $g(.)$, we split `feature_matrix` into three sets, a set for training the model, a set to tune the decision function, and a set to test/validate a model. These splits can be made based on time and the could be made much earlier in the process allowing for different label search strategies to extract training examples from different splits. A typical model search works as follows:

– Train a model M_i on the training split

– Use trained model to make predictions on the threshold tuning set.

– Pick threshold for making decision such that it minimizes the cost function $g(.)$

– Use the threshold and the model M to make predictions on the test split and evaluate the cost function $g(.)$

– Repeat steps 1 - 4 k times for model M_i and produce average statistics for it.

[11]Once the modeling method has been chosen, numerous search algorithms for the following steps have been published in academia and/or are available as open source tools.

– Repeat steps 1-5 for different models and pick the one that does best on average - on the test split.

At the end, the output of this step is a full model specification, the threshold, the model M. M is stored as a serialized file, and the remaining settings are stored in a $json$ called `model_provenance.json`. An example $json$ is shown in Figure 5.

3.5 Step 5 → Integration testing in production

The biggest advantage of the automation and abstractions designed for the first 4 steps is that they enable an identical and repeatable process with exactly the same `APIs` in the production environment – a requirement of **ML 2.0**. In a production environment, new data is added to the data warehouse D as time goes on. To test a model trained for a specified purpose, a developer , after loading the model M, the `feature_list` and the Entityset E_t, can simply call the following three `APIs`:

$$\mathtt{E_{t+}} = \mathtt{add\_new\_data}(\mathtt{E_t}, \mathtt{new\_data\_path}, \mathtt{metadata.json}) \tag{7}$$

$$\mathtt{feature\_matrix} = \mathtt{calculate\_feature\_matrix}(\mathtt{E_{t+}}, < \mathtt{e_i}, \mathtt{current\_time} >, \mathtt{feature\_list}) \tag{8}$$

$$\mathtt{predictions} \leftarrow \mathtt{generate\_predictions}(\mathtt{M}, \mathtt{feature\_matrix}) \tag{9}$$

This allows the developer to test the end-to-end software for new data, ensuring that the software will work in a production environment.

3.6 Step 6 → Validate in production

No matter how robustly the model is evaluated during the training process, a robust practice must involve evaluating a model in the production environment. This is paramount for establishing trust, as it identifies issues such as whether the model was trained on a biased data sample, or if the data distributions have shifted since the time data was extracted for steps 1-4. A **ML 2.0** system, after loading the model M, the `feature_list`, and having the updated Entityset E_{t+}, enables this evaluation by a simple tweaking of parameters in the `APIs`:

$$\begin{aligned}\mathtt{feature\_matrix} = \mathtt{calculated\_feature\_matrix}(\mathtt{E_{t+}}, < \mathtt{e_i}, \\ \mathtt{arbitrary\_timestamp} >, \mathtt{feature\_list})\end{aligned} \tag{10}$$

$$\mathtt{label}, \mathtt{arbitrary\_timestamp} = \mathtt{f}(\mathtt{E_{t+}}, < \mathtt{e_i}, \mathtt{arbitrary\_timestamp} >, \mathtt{hparams}) \tag{11}$$

$$\mathtt{predictions} \leftarrow \mathtt{generate\_predictions}(\mathtt{M}, \mathtt{feature\_matrix}) \tag{12}$$

$$\mathtt{cost} = \mathtt{g}(\mathtt{predictions}, \mathtt{labels}, \mathtt{E_{t+}}) \tag{13}$$

3.7 Step 7: Deploy

After testing and validation, a developer can deploy the model using the three commands specified by Equations 7, 8, and 9.

item	description
`metadata.json`	file containing a description of the an Entityset
`label_times`	the list of label training examples and the point in time to predict
`feature_matrix`	table of data with one row per label_times and one column for each feature
`M`	serialized model file returned by model search
`feature_list`	serialized file specifying the `feature_list` returned by feature engineering step
$f(.)$	user defined function used to create label times
$g(.)$	cost function used during model search
`model_provenance.json`	description of pipelines considered, testing results, and final deployable model
`predictions`	the output of the model when passed a `feature_matrix`

Table 5: Different intermediate data and domain specific code generated during the end-to-end process

3.8 Meta information

While going through steps 1 -7 we maintain information about what settings/hyperparameters were set for each stage, the results at the modeling stage, and attempt to maintain the full provenance over the process. Our current proposal of this provenance is in `model_provenance.json` and is documented at https://github.com/HDI-Project/model-provenance-json. This allows us to check for data drift, non availability of certain columns and provides provenance as to what was tried during the training phase and did not work. It is also hierarchical as it points to three other `jsons`, one that stores the metadata, the second set stores the search space for each of the methods over which model search was performed and the third stores the automl method used.

4 Why does this matter?

The proposed **ML 2.0** system overcomes the significant bottlenecks and complexities currently hindering the integration of machine learning into products. We frame this as the first proposal to *deliberately* break out of **ML 1.0**, and anticipate that as we grow, more robust versions of this paradigm, as well as their implementations, will emerge. We have used the system as above, powered by a number of automation tools, to solve two real industrial use cases – one from Accenture, and one from BBVA. Below, we highlight what is important about the goals and system described above, and more deeply examine each innovation.

– **Standardized representations and process**: In the past, the process of converting complex relational data first into a *learning task* and subsequently into *data representations* on which machine learning can be applied had mostly been considered an "art." This process lacked scientific form, precise definitions and abstractions, rigor, and – most importantly – generalization. It also took about 80% of the average data scientist's time. The ability to structurally represent this process, and to define algorithms and intermediate representations, has enabled us to think up and formulate **ML 2.0**. If we aim to broadly deliver the promise of machine learning, this process, much like any other software development practice, must be streamlined.

– **The concept of "time"**: A few years ago, we participated in a KAGGLE competition using our automatic feature generation algorithm [5]. The algorithm applied mathematical operations to data by automatically

going through the data and the relationships within it. The process is typically done manually, and doing it automatically meant a significant boost in data scientists' productivity, as well as generating competitive accuracy. Keeping this algorithm's core ability – to generate features given data – we then applied it to industrial datasets. We recognized that, in this new context, we had to first define the machine learning task. When a task is defined, performing step 2 (assembling training examples) implied that *"valid"* data had been identified, and since task definition required flexibility, which data was valid and which was not changed each time we changed the definition. Competition websites perform this step before they even release the data - a key problem we identified in **ML 1.0**. This brought in the third and most important dimension of our automation – "time". To identify valid data, we needed our algorithms to filter data based on the time of their occurrence, and required that every data point have an annotation of the time when it was "known."[12] A complex interplay between how these annotations are stored in databases and how automation algorithms could make use of this ensued, eventually resulting in **ML 2.0**. Our ability to automate the search process led to step 2. Accounting for time allows unprecedented flexibility in defining machine learning task - and maintains provenance.

– **Automation of processes**: While our work in early 2013 focused on the automation of model search (step 4), we moved to automating step 3 in 2015, and subsequently step 2 in 2016. These automation systems enabled us to go through the entire machine learning process end-to-end. Once fully automated, we were then able to design interfaces and `APIs` that enabled easy human interaction and a seamless transition to deployment.

– **Same `APIs` in training and production**: As we show by using the same `APIs` in the previous section and in the example problems, the `APIs` used in steps 5-7 are exactly the same as the ones used in steps 2 and 4. This implies that the same software stack is used both in training and deployment. The same software stack allows the same team to execute all the steps, requires a minimal learning curve when things need to be changed, and most importantly, provides proof of the model's provenance – how it was validated both in the development environment and in the test/production environment.

– **No redundant copies of data**: Throughout **ML 2.0**, new data representations made up of computations over the data – including features, training examples, metadata, and model descriptions – are created and stored. However, nowhere in the process is a redundant copy of a data element made. [13] Arguably, this mitigates a number of data management and provenance issues elucidated by researchers. Although it is hard to say whether all such issues will be mitigated across the board, **ML 2.0** eliminates multiple hangups that are usually a part of machine learning or predictive modeling practice.

– **Perfect interplay between humans and automation**: After our deliberate attempts to take our tools and systems to industry, and debates over *who it is we are enabling* when the phrase "democratize machine learning" is used, we have come to the conclusion that a successful **ML 2.0** system will require perfect interplay between humans and automated systems. Such an interplay ensures that: (1) Humans control aspects that are closely related to the domain – such as domain-specific cost functions, the outcome they want to predict, how far in advance they want to predict, etc. – which are exposed as hyperparameters they can set in steps 2-5. (2) The process abstracts away the nitty-gritty details of processing data based on the various forms and shapes in which it is organized, generating data representations, searching for models, provenance, testing, validation and evaluation. (3) It provides a structure and common language for end-to-end process of machine learning to aid in sharing

[12] This is not necessarily equivalent to the time it was recorded and stored in the database, although sometimes it is.

[13] Unless the computation is as simple as identity – for example, if the age column is available in the data, and we would use this column as a feature as-is, it will appear in the original data as well as in the feature matrix.

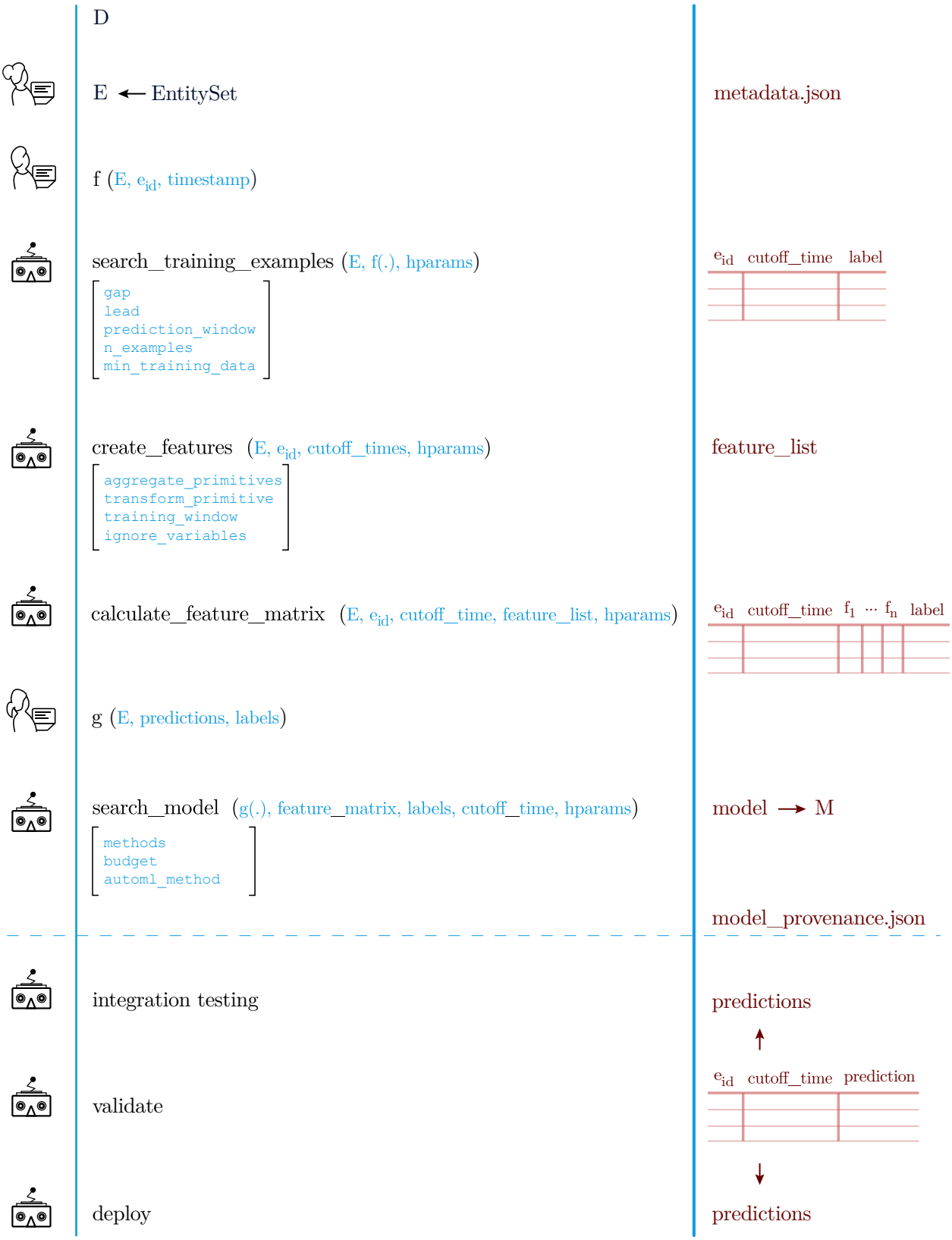


Figure 2: ML 2.0 workflow. The end result is a stable, validated, tested deployable model. An ML 2.0 system must support each of these steps. In the middle, we show each of the steps with well defined high level functional `APIs` with arguments (hyperparameters) that allow for exploration and discovery, but makes it a repeatable process and the data transformation `APIs` do not change in the deployment. The rightmost column presents the result of each of the steps.

5 The first AI product - delivered using ML 2.0

In an accompanying paper called "The AI Project Manager", we demonstrate how we used the seven steps above to deliver a new AI product for the global enterprise Accenture. Not only did this project undertake a ML problem that didn't have an existing solution, it presented us an unique opportunity to work with *subject matter expert*s to develop a system that would be put to use. In this case, the goal was create an AI Product Manager that could augment human product managers while they manage complex software projects. Using over 3 years of historical data that contained 40 million reports and 3 million comments, we trained a machine learning-based model to predict, weeks in advance, the performance of software projects against a host of delivery metrics. The *subject matter expert*s provided these delivery metrics for prediction engineering, as well as valuable insights into the subsets of data that should considered for automated feature engineering. The end-to-end project to develop and deliver spanned 8 weeks due to the practice of ML 2.0. In live testing, the AI Project Manager correctly predicts potential issues 80% of the time, helping to improve key performance indicators related to project delivery. As a result, the AI Project Manager is integrated in Accenture's myWizard Automation Platform and serves predictions on a weekly basis.

References

[1] Anand Avati, Kenneth Jung, Stephanie Harman, Lance Downing, Andrew Ng, and Nigam H Shah. Improving palliative care with deep learning. *arXiv preprint arXiv:1711.06402*, 2017.

[2] Sushmito Ghosh and Douglas L Reilly. Credit card fraud detection with a neural-network. In *System Sciences, 1994. Proceedings of the Twenty-Seventh Hawaii International Conference on*, volume 3, pages 621–630. IEEE, 1994.

[3] Sherif Halawa, Daniel Greene, and John Mitchell. Dropout prediction in moocs using learner activity features. *Experiences and best practices in and around MOOCs*, 7:3–12, 2014.

[4] Jiazhen He, James Bailey, Benjamin IP Rubinstein, and Rui Zhang. Identifying at-risk students in massive open online courses. In *AAAI*, pages 1749–1755, 2015.

[5] James Max Kanter and Kalyan Veeramachaneni. Deep feature synthesis: Towards automating data science endeavors. In *Data Science and Advanced Analytics (DSAA), 2015. 36678 2015. IEEE International Conference on*, pages 1–10. IEEE, 2015.

[6] James Max Kanter, Owen Gillespie, and Kalyan Veeramachaneni. Label, segment, featurize: a cross domain framework for prediction engineering. In *Data Science and Advanced Analytics (DSAA), 2016 IEEE International Conference on*, pages 430–439. IEEE, 2016.

[7] Arti Ramesh, Dan Goldwasser, Bert Huang, Hal Daumé III, and Lise Getoor. Modeling learner engagement in moocs using probabilistic soft logic. In *NIPS Workshop on Data Driven Education*, volume 21, page 62, 2013.

[8] D Sculley, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, and Michael Young. Machine learning: The high-interest credit card of technical debt. 2014.

[9] Haoran Shi, Pengtao Xie, Zhiting Hu, Ming Zhang, and Eric P Xing. Towards automated icd coding using deep learning. *arXiv preprint arXiv:1711.04075*, 2017.

[10] Tanmay Sinha, Patrick Jermann, Nan Li, and Pierre Dillenbourg. Your click decides your fate: Inferring information processing and attrition behavior from mooc video clickstream interactions. *arXiv preprint arXiv:1407.7131*, 2014.

[11] Thomas Swearingen, Will Drevo, Bennett Cyphers, Alfredo Cuesta-Infante, Arun Ross, and Kalyan Veeramachaneni. Atm: A distributed, collaborative, scalable system for automated machine learning.

6 Commentary on ML 1.0

Perhaps the fundamental problem with the workflow is that it was never intended to create a usable model. Instead, it was designed to make *discoveries*. In order to achieve this end goal of an analytic report or a research paper introducing the *discovery*, numerous tradeoffs are made. Below, we highlight some of these tradeoffs, and how they are at odds with creating a deployable, usable model.

– **Ad hoc non-repeatable software engineering practice for steps 1- 4**: The software that carries out steps 1 - 4 is often ad-hoc and lacks a robust, well-defined, software stack. This is predominantly due to lack of recognized structure, abstractions and methodologies. Most academic research and pursuits in machine learning left this part out for the past 2 decades and only focused on defining a robust practice when a data is ready to be used at step 5, and/or data is a clean time stamped time series X, t (for example, time series of a stock price) [14]. This implies that a subset of transformations done to the data during *discovery* that need to repeated in production - will have to be re-implemented -probably by a different team requiring extensive communication, back and forth.

– **Ad hoc data practices**: In almost all cases, as folks go through steps 1, 2, 3, they insert their domain knowledge, and data output at each stage is stored on the disk and passed around with some notes (sometimes through emails). To ensure confidence in discoveries made, data fields that are questionable or cannot be explained easily or without caveats are dropped. No provenance is maintained for the entire process.

– **Siloed and non-iterative**: These steps are usually executed in a siloed fashion, and often by different individuals within an enterprise. Step 1 is usually executed by people managing the data warehouse. Step 2 is done by domain experts in collaboration with the software engineers maintaining the data warehouse. Steps 3 and 4 fall under the purview of data science and machine learning folks. At each step, a subset of intermediate data is stored and passed on. After a step is finished, very rarely does anyone wants to revisit it or ask for additional data, as this often entails some kind of discussion or back-and-forth, and is thus generally avoided.

– **Over-optimizing steps 4 and 5, especially step 5**: While the data inputs for steps 1 and 2 are complicated and nuanced, the data is usually simplified and shrunk down to a goal-specific subset by the time it is arrives at step 4. It is simplified further when it arrives at step 5, at which stage it is just a matrix. Steps 1, 2 and 3 are generally considered tedious and complex, and so when work is done to bring the data to step 4, engineers make the most of it. This has led to steps 4 and 5 being overemphasized, engineered and optimized. For example, all data science competitions, including those on KAGGLE, start at Step 4, and almost all machine learning startups, tools and systems start at Step 5. At this point, there are several ways to handle these two steps, including crowdsourcing, competitions, automation, and scaling. Sometimes, these over-engineered solutions do not translate to real-time use environments. Take for example, the 2009

[14] Take for example, a well recognized and immensely useful textbook "Elements of Statistical Learning" describes numerous methods, concepts, once the data is ready in a matrix form.

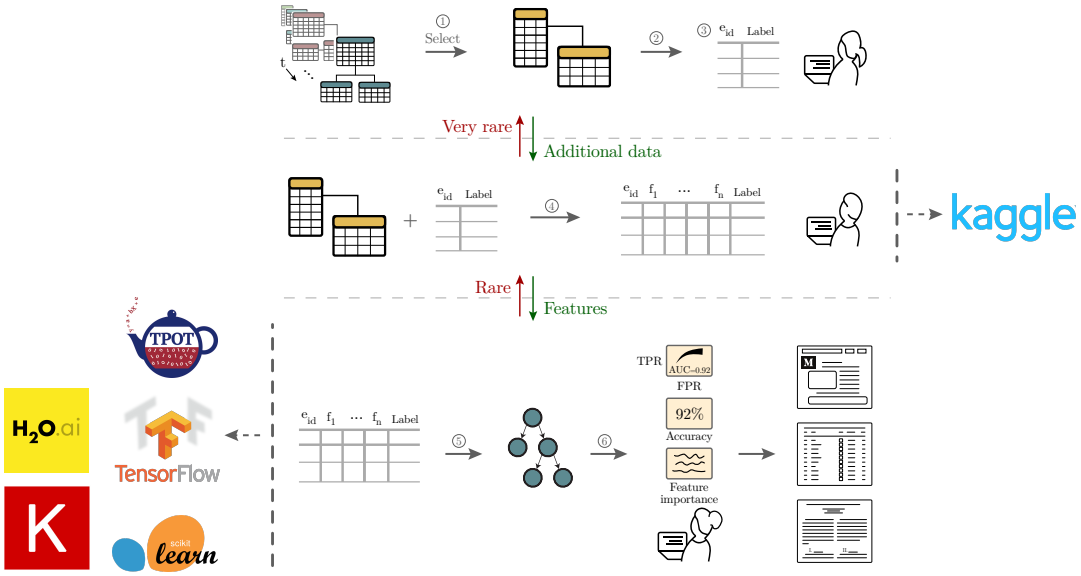


Figure 3: ML 1.0 workflow. This traditional workflow is split into three disjoint parts, each executing a subset of steps, and often executed by different folks. Intermediate data representations generated at the end of each part are stored and passed along to folks next in the chain. At the top, a relevant subset of the data is extracted from the data warehouse, the prediction problem is decided and the past training examples are identified within the data and data is prepared so as to enable model building whilst avoiding issues like "label leakage". This step is usually executed in collaboration with folks that manage/collect the data (and the product where the data emanates from), domain/business experts who help define the prediction problem and the data scientists who would then like to receive the data to build the predictive models. The middle part of the workflow is feature engineering. In this for each training example, a set of features are extracted is executed by the data scientists. Often times if statistics and machine learning experts are involved, this part could be executed by software engineers on the team or the data engineers. This is also the stage when the data could be released as a public competition on KAGGLE. The last part is of building the predictive model, validating it and analyzing it. This part is executed by someone familiar with machine learning. A number of automated tools, open source libraries are available to do this. We show a few of them on the left. The goal of all of this mostly has been *discovery*, ultimately resulting in a *research paper*, or a *competition leader board*, or a *blogpost*. To put the machine learning model to use is an afterthought.This traditional workflow has a couple of problems: first decisions taken at the previous part are rarely revisited, the prediction problem is prematurely decided without the ability to explore how small changes in it can garner significant benefits, and finally an end-to-end software cannot be compiled at one place - so it could be repeated with new data, or a deployable model be created.

Netflix challenge. The winning solution was comprised of a mixture of 126 models, which made it difficult to engineer it for real-time use.

– **Departure from the actual system collecting the data**: Since data is pulled out from the warehouse at time t, and the *discovery* ensues at some arbitrary time in the future, it may be the case that the system is no longer recording certain fields. This could be due to policy shifts, the complexity or cost involved in making those recordings, or simply a change in the underlying product or its instrumentation. While everything written as part of the *discovery* is still valid, the model may not actually be completely implementable, and so might not be put to use.

Ultimately, a *discovery* has to make its way to real use by integrating with real time data. It implies that new incoming data has to be processed/transformed precisely as was to learn the model, a robust software stack making predictions be developed and integrated with the product. Usually, this responsibility falls on the shoulders of the team responsible for the product, who most likely have not been involved in the end-to-end model discovery process. Thus the team starts auditing the discovery, trying to understand the process, simultaneously building the software, learning concepts that may not be in their skill set (for example - cross validation), and trying to gain confidence in the software as well as the predictive capability - before putting it to use. Lack of well documented steps, robust practice for steps 1-4 and ability to replicate model accuracy, can often result in confusion, distrust in the possibility of model delivering the intended result and ultimately result in abandoning of the idea of deploying it altogether.

Appendix

Figure 4: A specification of search parameters for a machine learning method.

```
{
    "name": "dt",
    "class": "sklearn.tree.DecisionTreeClassifier",
    "parameters": {
        "criterion": {
            "type": "string",
            "range": ["entropy", "gini"]
        },
        "max_features": {
            "type": "float",
            "range": [0.1, 1.0]
        },
        "max_depth": {
           "type": "int",
           "range": [2, 10]
        },
        "min_samples_split": {
           "type": "int",
           "range": [2, 4]
        },
        "min_samples_leaf": {
           "type": "int",
           "range": [1, 3]
        }
    },
    "root_parameters": ["criterion", "max_features",
    "max_depth", "min_samples_split", "min_samples_leaf"],
    "conditions": {}
}
```

Figure 5: An example JSON prediction log file detailing the provenance of the accompanying model, including instructions for labeling, data splitting, modeling, tuning, and testing (continued in Figure 6). The most up-to-date specification of this json can be found at https://github.com/HDI-Project/model-provenance-json.

```
{
    "metadata": "/path/to/metadata.json",

    "prediction_engineering": {
        "labeling_function": "/path/to/labeling_function.py",
        "prediction_window": "56 days",
        "min_training_data": "28 days",
        "lead": "28 days"
    },

    "feature_engineering":[
```

```
    {
      "method": "Deep Feature Synthesis",
      "training_window": "2 years",
      "aggregate_primitives": ["TREND", "MEAN", "STD"],
      "transform_primitives": ["WEEKEND", "PERCENTILE"],
      "ignore_variables":{
          "customers": ["age", "zipcode"],
          "products": ["brand"]
      },
      "feature_selection": {
          "method": "Random Forest",
          "n_features": 20
      }
    }
  ],

  "modeling": {
      "methods": [{"method": "RandomForestClassifer",
                   "hyperparameter_options": "/path/to/random_forest.json"
                      },
                  {"method": "MLPClassifer",
                   "hyperparameter_options": "/path/to/
                      multi_layer_perceptron.json"}],
      "budget": "2 hours",
      "automl_method": "/path/to/automl_specs.json",
      "cost_function": "/path/to/cost_function.py"
  },

  "data_splits": [
      {
          "id": "train",
          "start_time": "2014/01/01",
          "end_time": "2014/06/01",
          "label_search_parameters": {
              "strategy": "random",
              "examples_per_instance": 10,
              "offset": "7 days",
              "gap": "14 days"
          }
      },
      {
          "id": "threshold-tuning",
          "start_time": "2014/06/02",
          "end_time": "2015/01/01",
          "label_search_parameters": {"offset": "7 days"}
      },
      {
          "id": "test",
          "start_time": "2015/01/02",
          "end_time": "2015/06/01",
          "label_search_parameters": {"offset": "7 days"}
      }
  ],
```

Figure 6: An example JSON prediction log file detailing the provenance of the accompanying model, including instructions for labeling, data splitting, modeling, tuning, and testing (continued from Figure 5)


```
    "training_setup": {
        "training": {"data_split_id": "train",
                     "validation_method": "/path/to/validation_spec_train.↩
                        json"},
        "tuning": {"data_split_id": "threshold-tuning",
                   "validation_method": "/path/to/validation_spec_tune.json"↩
                      },
        "testing": {"data_split_id": "test",
                    "validation_method": "/path/to/validation_spec_test.json↩
                       "}
    },


    "results": {
      "test": [
          {
              "random_seed": 0,
              "threshold": 0.210,
              "precision": 0.671,
              "recall": 0.918,
              "fpr": 0.102,
              "auc": 0.890
          },
          {
              "random_seed": 1,
              "threshold": 0.214,
              "precision": 0.702,
              "recall": 0.904,
              "fpr": 0.113,
              "auc": 0.892
          }
      ]
    },


    "deployment": {
        "deployment_executable": "/path/to/executable",
        "deployment_parameters": {
            "feature_list_path": "/path/to/serialized_feature_list.p",
            "model_path": "/path/to/serialized_fitted_model.p",
            "threshold": 0.212
        },
        "integration_and_validation": {
            "data_fields_used": {
                    "customers": ["name"],
                    "orders": ["Order Id", "Timestamp"],
                    "products": ["Product ID", "Category"],
                    "orders_products": ["Product Id", "Order Id", "Price", "↩
                       Discount"]
```

```
            },
            "expected_feature_value_ranges":{
                "MEAN(orders_products.Price)": {"min": 9.50, "max": 332.30},
                "PERCENT(WEEKEND(orders.Timestamp))": {"min": 0, "max": 1.0}
            }
        }
}
```