Swearingen et al., "ATM: A Distributed, Collaborative, Scalable, System for Automated Machine Learning," Proc. of IEEE International Conference on Big Data, (Boston, USA), December 2017.

ATM: A distributed, collaborative, scalable system for automated machine learning

Thomas Swearingen*, Will Drevo†, Bennett Cyphers†, Alfredo Cuesta-Infante‡, Arun Ross* and Kalyan Veeramachaneni†

*Department of Computer Science and Engineering
Michigan State University, East Lansing, Michigan 48824
Email: swearin3,rossarun@cse.msu.edu

†Laboratory for Information and Decision Systems
MIT
Cambridge, MA - 02139
Email: drevo, bcyphers, kalyanv@mit.edu

‡Universidad Rey Juan Carlos, Madrid, Spain
Email: alfredo.cuesta@urjc.es

***Abstract*—In this paper, we present Auto-Tuned Models, or ATM, a distributed, collaborative, scalable system for automated machine learning. Users of ATM can simply upload a dataset, choose a subset of modeling methods, and choose to use ATM's hybrid Bayesian and multi-armed bandit optimization system. The distributed system works in a load-balanced fashion to quickly deliver results in the form of ready-to-predict models, confusion matrices, cross-validation results, and training timings. By automating hyperparameter tuning and model selection, ATM returns the emphasis of the machine learning workflow to its most irreducible part: *feature engineering*. We demonstrate the usefulness of ATM on 420 datasets from OpenML and train over 3 million classifiers. Our initial results show ATM can beat human-generated solutions for 30% of the datasets, and can do so in 1/100th of the time.**

I. Introduction

One of the most common data science problems is that of deriving a predictive model from a labeled set of raw training data. From the beginning, a data scientist working on such a problem is flooded with choices and contingencies. The data set may comprise images, text, relational data, or a mix of these types. Pre-processing it often requires a number of steps, including data cleaning, feature engineering, and feature selection, before model training can even begin. These pre-processing steps are often domain-specific, and the space of possible options for them is vast. Formulating the correct set of steps, also known as a *pipeline*, requires iteration, collaboration and even verification.

In most cases, the last step in the pipeline is building and learning a model, given a feature matrix and labels. This last step presents its own choices and challenges, as there are now an overwhelming number of ways to learn a model. Over the years, many methods for classification have been developed, including *support vector machines* (`svm`), *neural networks* (`nn`), *bayesian networks* (`bn`), *deep neural networks* (`dnn`), and *deep belief networks* (`dbn`). Each method requires numerous hyperparameters to be set before learning a model. A model's performance can also be judged by many possible metrics, including *accuracy*, *precision*, *recall*, and *F1-score*.

It is important to note that this last step is abstracted away from domain-specific intricacies by relying on a domain-agnostic input data format, the feature matrix. This makes it a good target for automation. The past few years have seen the development of an overwhelming number of automatic method selection and hyperparameter tuning algorithms and systems[1] [1], [2], [3], each purporting to be better then the other in its ability to search the space. Data scientists hoping to take advantage of these advances are once again lost in an enormous option space: which AutoML method should one use?

At the same time, several pragmatic data science needs remain chronically unaddressed. These are: (a) support for parallel exploration, in which multiple data scientists may simultaneously search through the model space for the same dataset but different pipelines (or even different datasets), (b) an ability to extend the model search space, and algorithms that allow for simultaneously searching through methods and their hyperparameters, and (c) abstractions for bringing together these numerous search approaches, so data scientists can explore multiple search methods as needed.

In this paper, we present Auto-Tuned Models, or ATM, a *multi-user* machine learning service that can be hosted on any distributed computing infrastructure, whether a cloud or a cluster. ATM is a multi-method, multi-parameter, self-optimizing machine learning system for the automation of classification modeling and hyperparameter tuning. By enabling the simultaneous execution of multiple methods

[1]which now define a sub-area within the machine learning community, called AutoML

and providing a suite of best possible models, ATM returns the emphasis to the part of the machine learning workflow that has proven most irreducible: *feature engineering*.

While commercial solutions can attempt to solve some of the aforementioned problems, we believe this technology should be open source, extensible, community-driven, and widely available. This has motivated us to develop abstractions, systems, and contributory frameworks in addition to the actual search algorithms. We imagine our system will have dual effects. First, it will demystify AUTOML by providing standardized abstractions in a common library where continually improving approaches are integrated – similar to the way that `scikit-learn` has demystified and democratized machine learning. Second, it will provide a trusted open-source solution for enterprises to save time and resources and bring back focus to the impactful, domain-specific aspects of the data science endeavor: formulating the precise machine learning task and transforming their data into information-rich features.

Summary of results: We provided ATM with 420 datasets downloaded from the OpenML platform and trained models for several days.

- **ATM generated the largest repository of trained models**: We learned a total of 3 million models across these 420 datasets, and generated a total of 4 terabytes of files consisting of models and their performance metrics. We will release this data in a structured way and provide `APIs` to access it. To the best of our knowledge, this will be the largest repository of trained models available to date.
- **ATM is able to beat human-generated solutions for 30% of the datasets**: Once the dataset is available on the OpenML platform, data scientists can download the data, try different classifiers on their local machines, and upload their results. At the time of writing this paper, we were able to extract the first 500 submissions made to 47 datasets using their `APIs`. Figure 1 shows the percentage of datasets ATM was able to beat the best amongst these first 500 by using simple grid search or by an intelligent search technique called GP/Bandit.
- **ATM is able to perform in 1/100th of the time**: For every dataset for which ATM beats the human-submitted solution, we calculated the time difference between the first and the best submission made by humans to the OpenML platform (best within the first 500). On average, this difference was 243 days [2]. A similar calculation reveals that ATM's grid search can generate the solution that beats the best human solution within a few minutes.

[2]We note that there was no particular incentive for humans to submit or enhance solutions. We also acknowledge that we do not know the skill level of the humans who submitted these solutions.

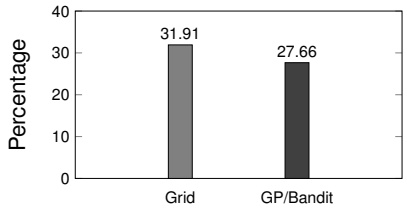

Figure 1: The percentage of datasets out of 47 for which ATM models outperform the first 500 human-in-the-loop attempts.

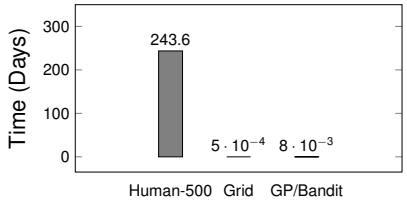

Figure 2: The average time required for ATM to find a classifier that outperforms the first 500 human-in-the-loop attempts. The Human-500 bar shows the average time needed for the best-performing classifier to be found by an OpenML user in the first 500 attempts.

II. ATM OVERVIEW

We describe our system from the point of view of an end user. We imagine ATM drawing various categories of users, as detailed below:

Data scientists using ATM can simply upload a dataset, select their desired methods, and choose hyperparameter ranges to search over. [3] They can decide to use either of ATM's two optimization/model search/selection methods: (1) a hybrid Bayesian and multi-armed bandit optimization system, or (2) a model recommender system that works by exploiting the modeling techniques' previous performances on a variety of datasets. Multiple users can submit multiple disparate datasets to be run simultaneously. They can also submit a different version of the same dataset – for example, one that has undergone a different set of preprocessing and transformation steps. After submitting the dataset, the user can begin getting results (models) via streaming. The system works in a load-balanced fashion to quickly deliver results in the form of ready-to-predict models, confusion matrices, cross-validation results, and training timing. We describe this in detail in Section V.

AutoML experts can contribute newer search methods by following the abstractions we define for tuning hyperparameters and selecting methods. In our current library, we propose and use two different methods, and provide abstractions so that anyone can modify and propose new ones. We describe

[3]Scaling and dimensionality reduction techniques are available to aid in the discovery process.

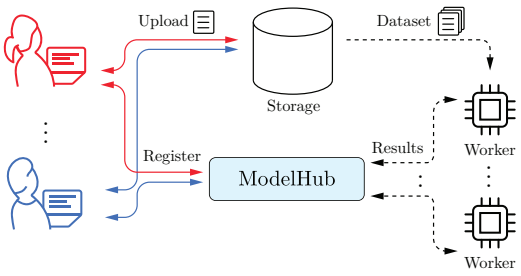

Figure 3: Overview of the ATM system. A user uploads their dataset to storage and registers it to the ModelHub database. The ATM workers query ModelHub for classifiers to run and report the results back to the ModelHub.

this in detail in Section IV

Machine learning enthusiasts who know more about each of the individual methods used in our library can enhance them by (a) proposing appropriate ranges for their hyperparameters and (b) proposing newer methods or better implementations for them.

The contributions of our work include:

- An asynchronous, distributed system that simultaneously generates multiple classifiers (for multiple datasets), records classification results (including performance, models, and meta-information) and logs any errors as they occur.
- An innovative organization of the complex search space of classifiers. This allows users to extend this space to include newer classification approaches in a structured way.
- A hierarchical algorithm that allows for search at the method level and tunes hyperparameters.
- Abstractions to enable the contribution and integration of disparate AUTOML techniques while simultaneously testing them.
- A database repository of several hundred thousand classifiers trained on hundreds of datasets.
- An open source software release at: https://github.com/HDI-Project/ATM

III. RELATED WORK

There are a few popular existing AutoML packages, such as Auto-WEKA [1], auto-sklearn [2], and TPOT [3]. Each package has its own proprietary model search/tuning method and set of hyperparameters. In general, these current AutoML routines present a few problems. First, they are developed as single-user systems run on local machines, as opposed to a multi-user systems run in a cloud-based environment. Second, they do not track the current state of runs, including previously run experiments, results of those experiments, parameters for those experiments, *etc*. Third, they do not organize the solution space in an intelligent, search-supporting way. Fourth, they focus tuning solely on one metric, even though the user may be more concerned with a different metric.

The preponderance of existing work focuses on the classifier part of the pipeline, which accepts as its input a set of features with labels. Thornton, *et al.* released the first major AutoML system, Auto-WEKA [1]. They develop the Combined Algorithm Selection and Hyperparameter (CASH) objective function, and use two tree-based Bayesian optimization methods (SMAC and TPE) to solve the problem. With the goal of creating an AutoML system to remove the parameter tuning, they build their system around WEKA [4]. WEKA itself is a suite of machine learning algorithms intended for a large audience, with the caveat that parameters must be tuned using a trial and error process, which may be difficult for people with limited knowledge of machine learning to grasp. With the addition of Auto-WEKA, the parameter estimation stage of the ML pipeline is automated, and machine learning is more accessible to non-experts.

Feurer, *et al.* later proposed the auto-sklearn system, which makes two substantive improvements to Auto-WEKA [2]. First, they add a meta-learning step to initialize the parameters, which they find improves the final classification performance. Second, they do not discard models once trained and tested on the data, but instead store them and use them all to build an ensemble classifier at the end. They replace WEKA with sklearn, which has become the dominant machine learning software package. They also follow the CASH model developed in Auto-WEKA, but only use the SMAC Bayesian optimization method, as it performed better than TPE [1].

There have been other approaches to and extensions of AutoML. Salvador, *et al.* propose one such extension, for time-varying data [5]. They combine Bayesian optimization with four different adaptive strategies, which leads to improved predictive performance in the majority of test datasets. Olson, *et al.* eschew the CASH objective function and instead evolve the machine learning pipeline using genetic algorithms [3]. They demonstrate the "evolved" machine learning pipeline's potential, and show that for some datasets, it can even outperform primitive machine learning. This is especially interesting because there is no strict, human-derived pipeline in the problem formulation – instead, the pipeline is learned by the genetic algorithm.

One drawback of AutoML can be the cost of performing multiple runs of the training stage. Some works circumvent this issue by using a meta-learning approach to recommend a classifier using a process called algorithm selection [6]. There is an offline setup stage where each base-AutoML procedure is run on a set of preselected datasets. A classifier is then recommended for an unseen query dataset. They find some correlation between the query dataset and the preselected set of databases with performances based on a few randomly selected classifier/parameter pairs on the

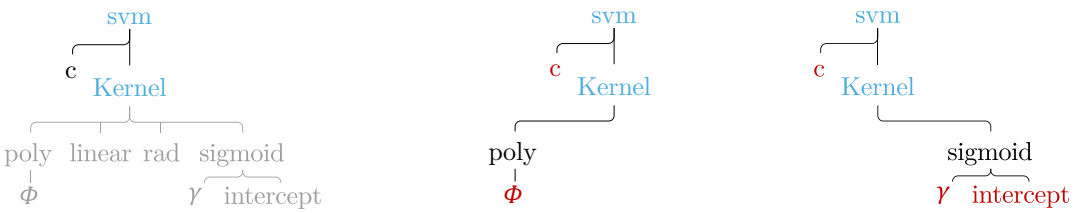

Figure 4: Hyperparameters and hyperpartitions within support vector machines. The leftmost diagram shows the full CPT, while the subfigures in the center and on the right show hyperpartitions 1 and 2. In hyperpartion 1, degree (ϕ) is a tunable parameter that is conditioned on the `kernel` being `polynomial`. In hyperpartion 2, γ and `intercept` are tunable parameters that are conditioned on the kernel type being `sigmoid`. Thus, kernel and soft margin are *method* hyperparameters, while `degree`, γ, and `intercept` are *conditional* hyperparameters.

query dataset. Abdulrahman, *et al.* evaluate this type of work when there is missing meta information [7].

There have also been many studies which target specific parts of the AutoML pipeline. Dewancker, *et al.* improve the AutoML pipeline with more sophisticated acquisition functions [8]. Kim, *et al.* propose candidate parameter sets with a Random Space Partitioning Optimizer [9]. Kim, *et al.* improve the scalability of a structure discovery algorithm [10]. Li, *et al.* discretize the *entire* hyperparameter space and then apply a multi-armed bandit scheme to select the hyperparameters [11].

IV. Defining the Search Space

In order to explore the space of possible model pipelines for a given problem, we must first have a way to programmatically enumerate the sets of hyperparameters which describe the models. This is difficult, because hyperparameters are hierarchical, and values selected at higher levels affect which choices must be made at lower levels. At the highest level, the choice of which modeling method to use affects which hyperparameters must be specified for that model. For example, a support vector machine (SVM) is described by a different set of hyperparameters than a decision tree. Furthermore, certain hyperparameters for a given model affect which other hyperparameters must be chosen. Consider a data scientist who is training an SVM. First she must choose which kernel to use. If she chooses a `polynomial` kernel, she must then specify the value for a discrete-ordinal hyperparameter `degree`. If she chooses a `sigmoid` kernel instead, she must specify values for two additional continuous hyperparameters, γ and `intercept`. Therefore, choosing a set of hyperparameters is not as simple as sampling from a fixed vector space, even for a given type of model. We define three levels of hyperparameter:

methods: At the highest level, a modeling method must be selected first. ATM includes support for several common modeling methods, including support vector machines, deep belief networks, random forests, and naive Bayes classifiers. Table I lists the supported methods.

method hyperparameters: For a given modeling method, several *method hyperparameters* must be specified no matter what. Hyperparameters may be categorical choices, such as which `kernel` or which `transfer function` to use, discrete-ordinal choices like `number of epochs` and `number of hidden layers`, and continuous valued choices like that of `learning rate` and `soft-margin`.

conditional hyperparameters: Some hyperparameters must only be specified if certain choices for method-level hyperparameters are made. These are *conditional hyperparameters*. For example, when a Gaussian kernel is chosen for `svm`, a value for σ – the standard deviation of the kernel – needs to be specified.

We define two concepts to help explain the hyperparameter search space.

Definition 1: A *conditional parameter tree* (CPT) expresses the combined hyperparameter space for a given modeling method as a search tree. The *root* node of the CPT identifies the model-building method, and each node below the root represents a hyperparameter. Certain hyperparameter nodes also act as *branches*, with child nodes beneath them. *Branch nodes* are always categorical variables; the value chosen for a branch node determines which conditional hyperparameters below it must be specified. Nodes which do not have children are *leaves*. Method hyperparameters always descend directly from the root, while conditional hyperparameters always descend from branch nodes.

A CPT can be traversed to enumerate all possible models for a given method, essentially defining its search space. Figure 4 shows a typical CPT for a support vector machine. There are two method hyperparameters – a continuous valued `soft margin`, and a categorical valued `kernel`. The `kernel` is represented as a branch node. The choice of *kernel* determines which conditional hyperparameters must be specified. If *kernel* is `polynomial`, `degree` must be specified; if `sigmoid` is chosen, both `gamma` and `intercept` must be specified instead.

Definition 2: A *hyperpartition* is a subset of a CPT which

Table I: Table showing the list of methods, their hyperparameters ("m-hyp"), and conditional hyperparameters ("c-hyp"). The table also shows the number of hyperpartitions per method ("# hyp"). Categorical hyperparameters have the set of possible values, all other hyperparameters are continuous. Abbreviations used in the table are expanded as follows: SVM ← support vector machine, RF← random forest, ET ← extreme trees, DT← decision tree, SGD ← stochastic gradient descent, PA ← passive aggressive, KNN← k-nearest neighbors, LR ← logistic regression, GNB ← gaussian naive bayes, MNB← multinomial naive bayes, BNB← bernoulli naive bayes, GP← gaussian process, MLP← multi-layer perceptron, DBN← deep belief network.

Method	m-hyp	c-hyp	# hyp
SVM	kernel={linear, rbf, sigmoid, polynomial}, C	gamma, coef0, degree	4
RF	criterion={gini, entropy}, min-samples-leaf, max-features, min-samples-split, max-depth	*None*	2
ET	criterion={gini, entropy}, min-samples-leaf, max-features, min-samples-split, max-depth	*None*	2
DT	criterion={gini, entropy}, min-samples-leaf, max-features, min-samples-split, max-depth	*None*	2
SGD	loss={hinge, modified-huber, log, squared-hinge}, learning-rate={optimal, constant}, alpha, fit-intercept={False, True}, eta0, n-iter, penalty={l1, l2, elasticnet}, l1-ratio	*None*	48
PA	loss={hinge, squared-hinge}, C, n-iters	*None*	2
KNN	weights={uniform, distance}, n-neighbors, algorithm={brute, kd-tree, ball-tree }, metric={euclidean, manhattan, minkowski, chebyshev}	p, leaf-size	24
LR	penalty ={l1, l2}, fit-intercept= {True, False}, C, tol		4
GNB	*None*	*None*	1
MNB	alpha, fit-prior	*None*	1
BNB	binarize, alpha, fit-prior	*None*	1
GP	kernel= {constant, rbf, matern, rational-quadratic, exp-sine-squared}	nu, length-scale, alpha, periodicity	5
MLP	num-hidden-layers= {1, 2, 3}, activation= {relu, logistic, identity, tanh}, solver= {lbfgs, sgd, adam}, alpha	layer-1-hidden-size, beta-1, layer-2-hidden-size, beta-2, layer-3-hidden-size, learning-rate= {constant, invscaling, adaptive}, learning-rate-init	84
DBN	num-hidden-layers= {1, 2, 3}, epochs, learn-rates, output-activation-function= {softmax, sigmoid, linear, tanh}, learn-rate-decays, learn-rate-pretrain	layer-1-hidden-size, layer-2-hidden-size, layer-3-hidden-size,	12

consists of choices for all branch nodes on a path from the root to the leaves. A hyperpartition fully defines a set of tunable conditional hyperparameters.

These abstractions allow us to break the process for model selection into two steps. First is *hyperpartition selection*, in which the CPT is traversed and a value is specified for every branch node encountered. Next is *hyperparameter tuning*. Once a hyperpartition has been chosen, the remaining unspecified parameters can be selected from a fixed vector space. The vector space may have continuous elements as well as discrete elements, and is likely to have some combination of both. The process of selecting hyperparameters from the space defined by a hyperpartition can be optimized with Gaussian Processes and other sampling methods, as described in section VI. An example of a full set of hyperparameter choices is as follows:

```
{
  "method": "svm",
  "method-hyperparameters": {
    "c": 0.87,
    "kernel": "polynomial"
  },
  "conditional-hyperparameters": {"degree": 3}
}
```

Figure 4 shows two distinct hyperpartitions for a support vector machine CPT.

V. ATM: Auto-Tuned Models

We present ATM, a distributed, collaborative, scalable system for hyperparameter tuning and model generation. ATM consists of a cluster of `workers` which train and test predictive models with different sets of hyperparameters and sync up with a master database, the `ModelHub`. ATM can process multiple prediction problems and multiple datasets at once. ATM's worker cluster is persistent: a data scientist can decide to upload specifications for a variation of an unfinished problem or a different problem altogether while ATM is running, and workers will adopt the task on the fly.

A. Components

ATM comprises the following components:

Worker cluster: A `worker` is a persistent process which runs on a compute instance (in our tests, an EC2 instance). Workers execute model training and communicate with the rest of the system by appending to a central, immutable log in the ModelHub, as described below. When activated, a worker queries the ModelHub for a task to work on. It chooses an incomplete `datarun` from the database based on the dataruns' *priority* scores, then queries for the results of

all classifiers that have already been trained on the datarun. It runs *hyperpartition selection* to choose a set of partition hyperparameters, then runs *hyperparameter tuning* to select values for the remaining unspecified hyperparameters. Once it has chosen a full set of hyperparameters, it trains and tests a classifier to produce a model and various performance metrics. It saves the model and the metrics to storage (in our test, Amazon S3) and logs the results to the ModelHub. We implemented all workers in Python 2.7, and we used the `scikit-learn` library for specific classifier method implementations. Algorithm 1 presents the algorithmic logic within a worker node.

ModelHub: The `ModelHub` is a central database that stores information about data sets, task configurations, hyperpartitions, and previously-trained classification models, including links to the models themselves and information about their performance metrics. Our implementation of ModelHub uses a MySQL database in conjunction with Amazon S3 to store this information. The ModelHub does not perform any actual computation on the classifiers; its role is to provide a single, consistent log which all of the workers can query and with which they can sync their results. The data structures it holds are described in more detail below.

The ModelHub database contains four tables, each representing a logical data structure. Together they describe all the classifiers that have been tested for as long as the ModelHub has been running. Figure 5 shows the database schema, and the tables are summarized below.

- **datasets**: A `dataset` represents a set of train/test data. The table contains its name, paths to train and test data files (if applicable), and the name of the column containing the label. All datasets must be in tabular format. Currently, our implementation only accepts data in csv format.
- **dataruns**: A `datarun` is the logical abstraction for a single run of a machine-learning problem. The table stores configuration and settings specified by the user, including the methods for hyperpartition selection (e.g. "UCB1") and hyperparameter tuning (e.g. "Gaussian Process"), and their associated parameters. Each `datarun` has an associated budget, which tells the worker how much work to perform. If `budget_type == ''time''`, the column `budget_amount` bounds the amount of wall clock time workers may spend on the datarun; if `budget_type == ''classifiers''`, it bounds the total number of classifiers which may be computed. The `priority` column tells the workers which runs to work on first.
- **hyperpartitions**: A `hyperpartition` is a set of hyperparameters which define a path through a CPT, as described in section IV. The table contains the fixed values for the hyperparameters corresponding to the branch nodes in the CPT, as well as data about the unspecified conditional hyperparameters. Each `hyperpartition` is linked to a `datarun`.
- **classifiers**: A `classifier` represents a model trained on a dataset with a single, fully-qualified set of hyperparameters. It is also the smallest logical unit of work that can be assigned to a worker. Once a classifier is finished, `model_location` and `metrics_location` point to URLs in the cloud where the model and metrics objects for the classifer are stored. The `status` column can be set to "started", "completed", or "errored"; if it is set to the last, an `error_message` is also stored.

To summarize, when a data scientist wants to solve a prediction problem with ATM, he or she will upload a set of data and create a corresponding entry in the `datasets` table. They will then create a new `datarun` which includes configurations for budget, priority, and hyperparameter selection and links to the `dataset`. The script for registering a datarun automatically registers each possible `hyperpartition` based on the datarun's configuration. Once the datarun has been registered, workers will automatically begin training, testing, and saving `classifiers` for the datarun, and will continue until the datarun's budget has been exhausted.

Algorithm 1 ATMWorker

```
procedure ATMWorker
    while True do
        D = ModelHub.GetDataruns()
        for d ∈ D do
            if IsBudgetExceeded(d) then
                D = D \ d
            end if
        end for
        if D = {∅} then
            Sleep()                          ▷ Wait for dataruns to be added
            Continue
        end if
        d = SelectPriorityDatarun(D)
        θ̄_p^j ← Search(d.dataset_id, ModelHub.classifiers)
        M_p ← LearnClassifier(θ̄_p^j)
        y_p^j ← Evaluate(M_p d.dataset_id)
        ModelHub.StoreResult(M_p, y_p^j, θ̄_p^j)
    end while
end procedure
```

B. Outputs

ATM generates three different types of files during its operation: data, models, and metrics. We used Amazon's S3 service for storage. Workers and end users can access the S3 bucket with credentials established during the initial setup. Labeled data files must be provided by the user, but ATM performs one-time preprocessing and cleaning on the data to convert it into a consistent csv format. For each data and model pair `dataset_id`, `classifier_id` the following additional elements are stored:

– Metrics: To enable post-hoc analysis of the models, we store a set of several metrics generated by each round of a k-fold cross validation on the available

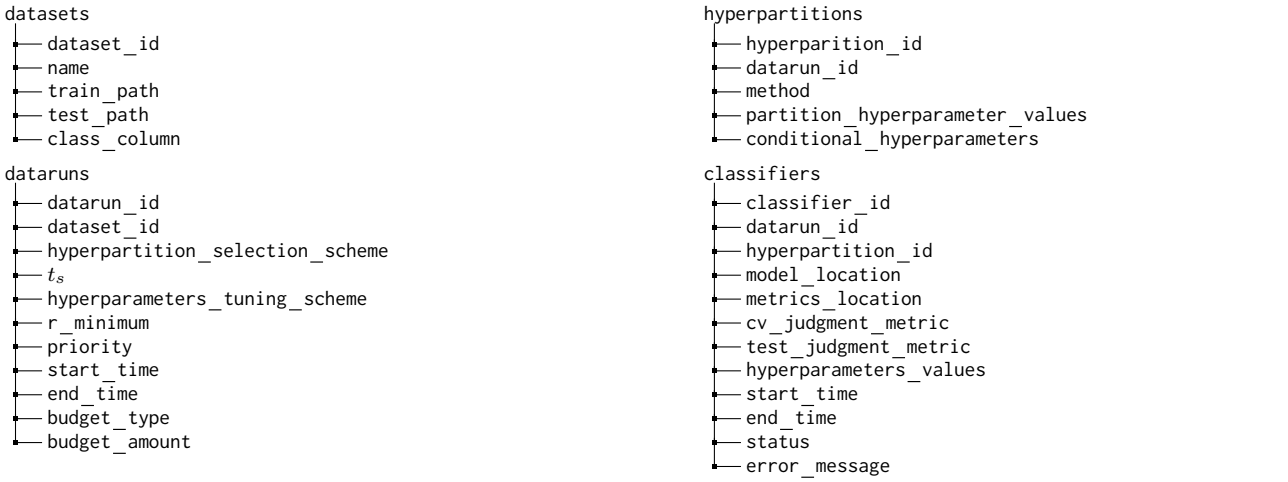

Figure 5: Diagram showing the relational ModelHub database schema.

Table II: Metrics stored on S3 for each of the classifier trained. PR ← Precision-recall, ROC ← Receiver operating characteristic, AUC ← Area under the curve

	Across folds	**For each fold**	
Problem	**Judgement metric**	**Curves**	**Metric**
Binary	F_1	PR ROC	Accuracy F_1-score AUC-PR AUC-ROC
Multi 3, 4, or 5	$\mu(F_1^c) - \sigma(F_1^c)$	PR/class ROC/pair	F_1/class AUC/class AUC/pair
Big-Multi > 5	$\mu(F_1^c) - \sigma(F_1^c)$	-	Rank-N F_1/class

training data. This allows data scientists to assess models based on the consistency of their performance as well as on their overall performance, and by a variety of different metrics. Data sets vary in number of classes, class balance, and number of training examples, which makes it impossible to decide on one universal "best" metric. While designing ATM, we consulted several data science experts about what metrics would be best. There was no agreement about an ideal subset of metrics, so we decided to include a wide range. The metrics computed for each classifier are shown in Table II. The `judgement metric` is a statistic computed across all folds and the `average` of this number is stored in the `ModelHub.classifiers` database. For Binary class problem we use the F_1 score achieved for each fold and calculate its average. For multi class problems, for each fold, (i) we calculate the F_1 scores for every pairing of classes, (ii) calculate the average and standard deviation of these F_1 scores and (iii) and finally subtract the standard deviation from the mean. This judgement metric is calculated for every fold and an `average` of this statistic across all folds in stored in the databse.

– Final trained model: After performing cross-validation testing on the dataset, the worker trains a final model using all the training data available. This model is serialized as a Python pickle (`.pkl`) file. The file is saved to the cloud and is uniquely identified using a hash of the method name, the hyperparameter settings, and the dataset path.

VI. Automatic Search

Combining the search spaces of multiple modeling methodologies and hyperpartitions creates a number of challenges with regard to finding the best model, either in the isolation of one methodology or from an hyperpartition. In particular, one encounters:

Discontinuity and non-differentiability: Categorical hyperparameters make the search space non-differentiable, and do not yield to simple search techniques (e.g. hill climbing) or to methods that rely on learning about the search space (e.g. Bayesian optimization approaches).

Varying dimensions of the search space: By definition, conditional hyperparameters imply that the hyperpartitions within a methodology have different dimensions. Because choosing one categorical variable over another can imply a different set of hyperparameters, the dimensionality of a hyperpartition also varies.

Non-transferability of methodology performance: Unfortunately, when conducting searches among modeling methodologies, robust heuristics are limited. Training on the dataset with an `svm` model gives us no information about how well a `dbn` might perform.

Table III: Notation table.

Symbol	Description
$t = 1 \dots T$	index into classifiers trained so far, ordered by time
i	index into dataset
$j = 1 \dots J$	index into hyperpartitions
$Y_j = y_1^j \dots y_T^j$	`judgement_metric` for classifiers tried so far for jth hyperpartition
$\bar{\alpha}^j$	a hyperparameter set for a classifier for jth hyperpartition
$\bar{\alpha}_1^j \dots \bar{\alpha}_T^j$	hyperparameter sets for classifiers tried so far for jth hyperpartition
z_j	reward for hyperpartition j
f_j	a meta model learned so far for hyperpartition j
$\bar{\alpha}_p^j$	proposed hyperparameters for hyperpartition j

Algorithm 2 ATM - Search algorithm

procedure SEARCH(Dataset $\leftarrow D_i$, `ModelHub.classifiers`)
  GET `judgement_metric` $y_1^j \dots y_T^j \ \forall j | D_i$
  for $j \in J$ **do**
    EVALUATE-REWARD: $z_j \leftarrow \mathbb{Q}(y_1^j \dots y_T^j)$
  end for
  SELECT HYPERPARTITION:

$$s_j = z_j + \sqrt{\frac{2 \ln n}{n_j}} \tag{1}$$

where j is hyperpartition index, z_j is the reward gained from pulling the j-th arm n_j times, and $n = \sum_{j=1}^{J} n_j$ over all J hyperpartitions.

$$\text{Select } j = \underset{j}{\operatorname{argmax}}\, s_j \tag{2}$$

  For selected j
  GET hyperparameters $\bar{\alpha}_{1 \dots T}^j$ corresponding to $y_{1 \dots T}^j$
  if $T \geq r_{minimum}$ **then**
    $f_j \leftarrow$ FIT$((\bar{\alpha}_1^j, y_1^j) \dots (\bar{\alpha}_T^j, y_T^j)$
    $\bar{\alpha}_p^j \leftarrow$ PROPOSE(f_j)
  else
    $\bar{\alpha}_p^j \leftarrow$ RANDOM(j)
  end if
  return $\bar{\alpha}_p^j$
end procedure

With a conditional parameter space framework in place, we now design automatic meta-learning techniques that iteratively select among hyperpartitions and tune hyperparameters. In our system, we use bandit based method to select amongst hyperpartitions and meta modeling technique to tune hyperparameters for a given hyperpartition. Algorithm 2 presents the combined bandit-based selection and metamodel-based tuning algorithm.

A. Hyperpartition selection using bandit learning

We employ bandit learning strategies which model each hyperpartition as an arm in a multi-armed bandit (MAB) framework.

Concept of reward: Given that each arm, when pulled, provides a randomized reward drawn from a hidden underlying distribution, the goal of a MAB problem is to decide which arm to pull to maximize long-term reward, re-evaluating the decision after each pull's reward is observed. In the context of ATM, UCB1 treats each hyperpartition as an arm with its own distribution of rewards. As time goes on, the bandit learns more about the distribution, and balances exploration and exploitation by choosing the most promising hyperpartitions to form classifiers.

Concept of Memory: Memory (moving window) strategies exist in order to change the bandit formulation when reward distributions are non-stationary. The UCB1 algorithm assumes that the underlying distribution of rewards at each arm choice is stationary. In the context of ATM, during model optimization, as the meta model (GP) continues to learn about the hyperpartitions and is able to discover parameterizations that continue to score better and better, it essentially shifts the bandit's perceived reward distribution. The distributions of classifier performance from the set of all parameterizations within that hyperpartition have not changed, but the bias with which the GP samples has changed. This could effectively lead a hyperpartition to be selected based on stale rewards. Memory strategies have a parameter t_s that determines how many previously trained classifiers to use to calculate the rewards.

Given the `Modelhub` data base, a worker first queries and aggregates the cross-validation scores $y_1^j \dots y_T^j$ for all the classifiers trained for a hyperpartition j thus far, and the corresponding hyperparameters set for each of the classifiers given by $\bar{\alpha}_1^j \dots \bar{\alpha}_T^j$. Treating each possible hyperpartition as an arm, we next calculate the reward "accumulated" for each hyperpartition.

Calculation of reward $\mathbb{Q}$: First a subset of classifiers, corresponding to the number t_s, is selected by either picking the t_s most recent trained classifiers, or the best t_s classifiers from the ones trained so far for this hyperpartition. These selected t_s scores are denoted as ys^j.

Reward based on average: This is the traditional way to define rewards. The reward is calculated by simply averaging these scores, given by:

$$z_j^T = \frac{1}{|t_s|} \sum_{i=1}^{|ys|} ys_i^j \tag{3}$$

with $|\cdot|$ being the cardinal of a set.

Reward based on velocity: Velocity strategies seek to rank hyperpartitions by their rate of improvement. This is based on the idea that a hyperpartition whose last few evaluations have made large improvements should be exploited while it continues to improve. To calculate velocity we first sort scores in ys in ascending order. Using velocity, the reward is changed to:

$$\bar{z}_j^T = \frac{1}{|t_s|} \sum_{i=2}^{|ys|} \Delta ys_i^j \tag{4}$$

for $\Delta ys_i^j = ys_i^j - ys_{i-1}^j$.

Velocity strategies are inherently very powerful because they introduce a natural feedback mechanism that controls exploration and exploitation. A velocity optimization strategy will explore each hyperpartition arm until that arm begins improving less quickly than others, going back and forth between hyperpartitions and wasting no time on hyperpartitions which are not promising.

Selection of a hyperpartition: Given the rewards for each of the arm, $z_j \ \forall j$ each arm (hyperpartitions) score is calculated using

$$s_j = \bar{z}_j + \sqrt{\frac{2 \ln n}{n_j}} \tag{5}$$

where n is the total number of classifiers trained so far across all hyperpartitions for this dataset and n_j is the total number of classifiers trained for this hyperpartition. Hyperpartition is selected based on $\text{argmax}_j \ s_j$

B. Hyperparmeter tuning using meta modeling

Within a hyperpartition, we employ a Gaussian Process (GP) -based meta-modeling technique to identify the best hyperparameters given the performance of classifiers already built for that hyperpartition [12]. To accomplish this, we propose a `Fit`, `Propose` abstraction.

`Fit`: In `Fit`, a meta model f_j is learned between the hyperparameter sets tried so far, $\bar{\alpha}_1^j \ldots \bar{\alpha}_T^j$, and their corresponding scores, $y_1^j \ldots y_T^j$.

`Propose`: In `Propose`, a new hyperparameter set $\bar{\alpha}_p$ is proposed. This usually involves creating candidate hyperparameter sets, making predictions using the learnt model f_j and choosing among those candidates by applying an `acquisition` function to their predictions. ATM supports two acquisition functions: *Expected Improvement*, and *Expected Improvement per Time* [12].

C. Abstractions in our software

One of the contributions of our work is to break the automatic machine learning process into two distinct subproblems: *hyperpartition selection* (via multi-armed bandit methods) and *hyperparameter tuning* (via Gaussian process-based methods). While we provide several versions of both bandit-based selection and GP-based tuning mechanisms, there are many ways these approaches could be improved upon for different circumstances.

The problem of hyperpartition selection is essentially one of picking between multiple discrete choices, each of which is associated with a set of scores representing its past performance. Multi-armed bandit algorithms, used for this, usually involve first computing a set of *rewards* for each arm/choice, then using the set of all rewards to determine which arm to pull next. To tackle this problem, we define a *Selector* interface with the following methods:

- **`compute_rewards`**: Accepts a list of scores pertaining to a single choice and returns a list of rewards.
- **`bandit_select`**: Accepts a mapping of choices to lists of rewards, and returns the choice which it believes will maximize the expected reward and minimize expected regret.
- **`select`**: Accepts a mapping of choices to historical performance scores, calls `compute_rewards` and `bandit_select` in sequence, and returns a single recommended choice.

Hyperparameter tuning involves taking a set of hyperparameter vectors, $\bar{\alpha}_{1\ldots T}$, and a set of corresponding scores, $y_{1\ldots T}$, and generating a new hyperparameter vector which is expected to improve on the previously-achieved y-values. For this problem, we define a *Tuner* interface with the following methods:

- **`fit`**: Accepts historical hyperparameter performance data ($\bar{\alpha}_{1\ldots T}$ and $y_{1\ldots T}$) and trains a model which can predict the performance of arbitrary hyperparameter vectors, e.g. a Gaussian process.
- **`predict`**: Accepts a set of proposed hyperparameter vectors and returns predicted performance for each one.
- **acquire**: Accepts the predicted performances of a set of proposed hyperparameter vectors and returns the index of the vector which it believes an ATM worker should try next.
- **`create_candidates`**: Generates a list of random hyperparameter vectors, to be passed to `predict` and then to `acquire`.
- **`propose`**: Calls `create_candidates`, `predict`, and `acquire` in sequence and returns a single proposed set of hyperparameters.

Our system includes several implementations of both Selectors and Tuners by default. However, we also expect that AUTOML experts will want to expand and improve upon our methods. An expert who wishes to contribute can do so by creating a subclass of Selector or Tuner and modifying one or more of the methods above. He or she can then use our testing functionality to automatically evaluate its efficacy.

VII. DATASETS

To demonstrate ATM, we use datasets from OpenML [13]. OpenML is a website[4] that allows users to upload datasets for public viewing and track the results of various machine learning pipelines. OpenML has over 19,000 datasets available for download. The uploader specifies details about the dataset such as which attribute must be predicted or which, if any, attributes should be ignored, the authors, citation information, collection date, or license information. Currently, the datasets must be uploaded in Attribute-Relation File Format[5]

[4] www.openml.org
[5] http://weka.wikispaces.com/ARFF

(ARFF), a common structure for specifying the values in the datasets as well as a common way of specifying feature types (*e.g.,* numeric or categorical), feature names, missing data values, instance weights, or sparse data.

Additional details about a data science exercise are specified in OpenML in what are called "tasks." A task defines whether the end result is classification or regression, what metrics must be returned, and what protocol to use (*e.g.,* a classification task would typically include cross-validation). A potential solution for a task is called a "flow" which is a machine learning pipeline. An example could be: Data $\rightarrow$ PCA $\rightarrow$ SVM $\rightarrow$ Report Predictive Accuracy. A flow specifies default values for the hyperparameters of a pipeline.

OpenML tracks "runs," which are flows applied to a task – that is, a run is a fully-specified (*i.e.,* with specific hyperparameter values) machine learning pipeline applied to a particular task. The hyperparameters may take on the default value from the flow, or they may be updated by the user to values he or she thinks are appropriate. This pipeline is executed on a user's personal computing hardware to generate results as specified in the task (*e.g.,* predictive accuracy or F-measure). These results are uploaded to OpenML through the API. The ability to see previous performance for a dataset is especially helpful, so that other users do not have to rerun flows to determine what does or does not yield good performance. A user simply looks at the performance of previous flows to generate a potential new flow or different hyperparmeter values that he or she thinks may improve performance.

A user can traverse the various runs through the API. A user can search by dataset, flow, or uploader. If we search by dataset, we can find the best performing run for a particular dataset. Since OpenML flows are designed by people, we can find a *human baseline* performance to compare our AutoML results against.

Selecting datasets for our experiment: We only select datasets which (1) are active, (2) have no missing values, and (3) have 2 to 4 classes. We exclude datasets which have missing values as incomplete-data classification is not the goal of this work. After applying our filtering criteria, we downloaded 420 datasets from the OpenML website. The datasets fall within a wide range of domains, including cancer risk detection, diabetes detection, credit risk detection, car acceptability, chess, tic-tac-toe, and spam detection. Figure 6 shows additional information about the datasets.

VIII. Experimental settings

We attempted to run ATM on the datasets using two selection schemes:

Grid Run: In this configuration, the hyperpartitions are selected randomly. The hyperparameter range is split into 3 values (start of range, end of range, and middle of range). One of the three values is then chosen randomly for each hyperparameter. For each dataset, we set a budget of 1,000 classifiers. Since there are 420 datasets, we attempt to learn 420,000 classifiers. Each classifier was trained using 10-fold cross-validation, thus our goal is to learn a total of 4.2 million models.

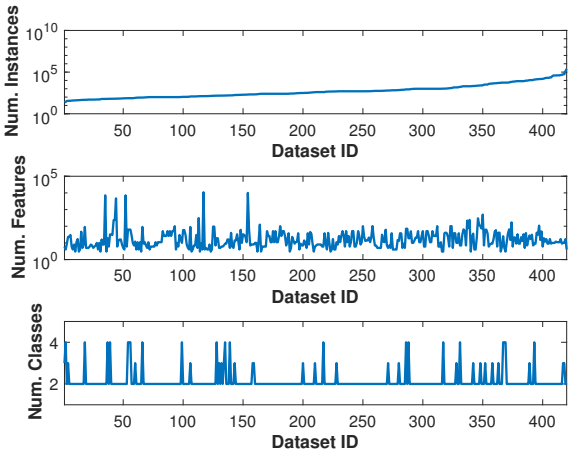

Figure 6: Graph showing the number of instances (top), number of features (middle), and number of classes (bottom) in each dataset. The datasets were sorted by number of instances.

GP+Bandit Run: We run ATM with Best t_s Velocity for hyperpartition selection and Gaussian Process-Expected Improvement for hyperparameter tuning. For each dataset, we set a budget of 100 classifiers. Since there are 420 datasets, we attempt to learn 42,000 classifiers.

IX. Results and discussion

Figure 7 shows the F1-score histograms of the best-performing classifiers for all datasets – both for the Grid run and the GP+Bandit run. In the GP+Bandit run, we attempted to learn 41,647 classifiers across all datasets where 39,342 were successfully learned while 2,305 logged errors. For the Grid run, we attempted to learn 286,577 classifiers across all datasets where 270,331 were successfully learned while 16,246 logged errors. Overall, we are able to achieve good classification performance for many datasets.

Comparison to human baselines: Regardless of whether a Grid or GP+Bandit strategy was used, the process is automatic, with no human involvement. We next compare this with a "human-in-the-loop" baseline. These baseline metrics come from the OpenML site itself. Because datasets hosted on this site are publicly available, data scientists can download the data, try different methods, and upload the best classifier and the performance metrics they achieved. Using their API, we collected the best possible F1-score submitted for a dataset by anyone. Figure 8 shows a histogram of the best ATM performance minus the best human-discovered performance for 47 datasets (as reported on OpenML).

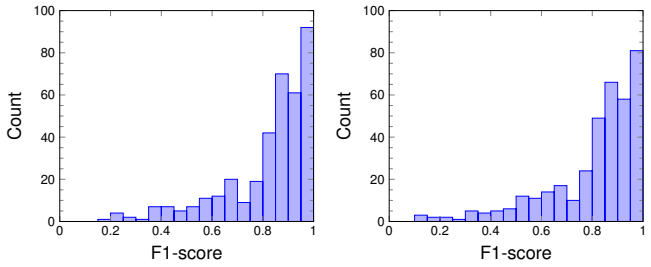

Figure 7: Histogram of the best F1-scores for each dataset using the Grid configuration (left) and GP+Bandit configuration (right).

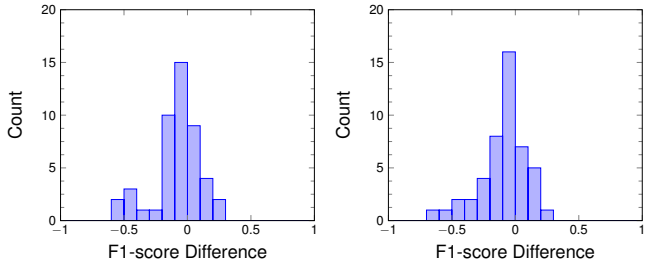

Figure 8: Histogram of the difference between the best F1-score for each dataset and the best "human-in-the-loop" F1-score from OpenML for 47 datasets. The results for the Grid configuration are on the left while the results for the GP+Bandit configuration are on the right.

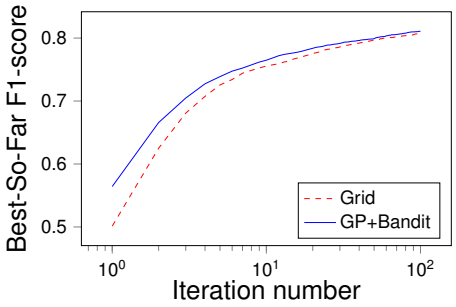

Figure 9: Comparing Grid and GP+Bandit strategies. The y axis is the average of the best-so-far F1-score achieved after x iterations for each problem.

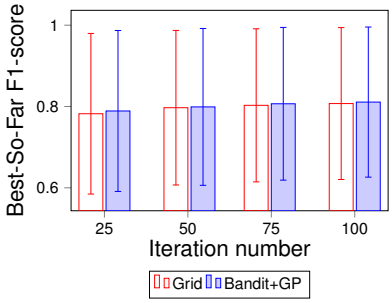

Figure 10: Comparing Grid and GP+Bandit at specific iterations. The y axis is the best-so-far F1-score after x iterations.

Positive values occur when ATM finds a better-performing classifier than the best "human-in-the-loop" classifier uploaded to OpenML. Negative values occur when ATM does not find a better performing classifier than the best "human-in-the-loop" classifier uploaded to OpenML.

Comparing Grid and GP+Bandit: Our next question targets whether an intelligent selection strategy that uses GP+Bandit achieves better result within a given budget of classifiers. To compare both strategies across all data sets, we first calculated a best-so-far F1-score. Figure 9 compares the two selection methods across all datasets using this metric (the line represents the average across all datasets). Both selection strategies have a budget of 100 classifiers. We see that GP+Bandit has a slight advantage over the random strategy, in that it takes slightly fewer classifiers for it to get to the same result. Figure 10 shows the same information at specific iterations.

X. Conclusion

In this paper, we presented ATM, a distributed AutoML system that can support multiple users, and search through multiple machine learning methods. The system is decentralized in that its many worker nodes work independently and asynchronously, only interacting through a shared database. ATM can learn hundreds of classifiers each for hundreds of datasets at a time. The system can assess performance on a variety of metrics, store metrics and models for future reference or analysis, and present the user with a final, optimized model to use for prediction. Users can configure ATM to search the vast hyperparameter space using several different strategies, and even to try multiple strategies at once in parallel.

Our main contribution was the design, implementation, and testing of the end-to-end system. In addition, we presented a novel method for organizing the hierarchical search space of machine learning methods. We defined the *conditional parameter tree* and *hyperpartition* and demonstrated how these abstractions can be used to better traverse complex hyperparameter spaces.

We designed several different search strategies and gave access to our end user through a simple `API`. To demonstrate our system, we ran the largest-ever training experiment, where we trained an aggregate of several million models across 420 data sets.

XI. Future work: model recommender system

Our current work provides a recommender approach, in which past ATM results are used to recommend models for a new dataset. We briefly describe this below, where we will

attempt to generate models for a dataset using the results from the remaining 419 datasets.

1) Organize results of the Grid Run into a matrix, M of size $420 \times 159,838$ where $159,838$ is the number of every possible classifier definition (method and specific hyperparameter values) and 420 represents the number of datasets on which classifiers have been trained in ATM. M will be a very sparse matrix.
2) M will be split into a row vector $\mathbf{p}$ and matrix G. The probe row ($\mathbf{p}$) is of size $1 \times 159,838$ while the gallery matrix G is of size $419 \times 159,838$.
3) Sample the probe row $\mathbf{p}$ down so it only has 5 values and store the new row vector in $\mathbf{p}_s$.
4) Record the best-so-far performance $\mu = \max(\mathbf{p}_s)$ where $\max(\cdot)$ reports the maximum value among all the defined values and ignores the undefined values.
5) Create a recommender matrix $R = \begin{bmatrix} G \\ \mathbf{p}_s \end{bmatrix}$.
6) Use Soft Imputation to complete the matrix and estimate values for the undefined values ($R' = \text{SoftImpute}(R)$).
7) Find the top 5 values from the last row in R' (probe row).
8) Run the classifiers corresponding to the top 5 values in Step 7.
9) Record the classifier performances in their corresponding column in $\mathbf{p}_s$.
10) Update $\mu = \max(\mathbf{p}_s)$.
11) Repeat steps 5-10 20 times.

Acknowledgments

Prof. Alfredo Cuesta-Infante is supported by the Spanish Government research funding TIN-2015-69542-C2-1-R(MINECO/FEDER). Prof. Arun Ross and Thomas Swearingen acknowledge the support from award no. 2015-R2-CX-0005, from the National Institute of Justice, Office of Justice Programs, U.S. Department of Justice. Bennett Cyphers and Kalyan Veeramachaneni acknowledge the support received from DARPA's Data driven discovery of models (D3M) program. Authors acknowledge comments and feedback received from Dr. Una-May O'Reilly during the early prototype development phase of the project. Authors acknowledge comments received from James Max Kanter. The opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect those of the Department of Justice, DARPA, MIT or any other organization authors are affliated to. Additionally, this work was supported in part by Michigan State University through computational resources provided by the Institute for Cyber-Enabled Research.

References

[1] C. Thornton, F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Auto-WEKA," *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '13*, p. 847, 2013.

[2] M. Feurer, A. Klein, K. Eggensperger, J. Springenberg, M. Blum, and F. Hutter, "Efficient and Robust Automated Machine Learning," *Advances in Neural Information Processing Systems 28*, pp. 2944–2952, 2015.

[3] R. S. Olson and J. H. Moore, "TPOT: A Tree-based Pipeline Optimization Tool for Automating Machine Learning," *Proceedings of the Workshop on Automatic Machine Learning*, vol. 64, pp. 66–74, 2016.

[4] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The WEKA data mining software," *ACM SIGKDD Explorations*, vol. 11, no. 1, pp. 10–18, 2009.

[5] M. Martin Salvador, M. Budka, and B. Gabrys, "Adapting Multicomponent Predictive Systems using Hybrid Adaptation Strategies with Auto-WEKA in Process Industry," *AutoML at ICML 2016*, no. 2011, pp. 1–8, 2016.

[6] P. Brazdil, C. Giraud-Carrier, C. Soares, and R. Vilalta, *Metalearning: Applications to data mining*. Springer Science & Business Media, 2008.

[7] S. M. Abdulrahman and P. Brazdil, "Effect of Incomplete Meta-dataset on Average Ranking Method," *Proceedings of the Workshop on Automatic Machine Learning*, vol. 64, pp. 1–10, 2016.

[8] I. Dewancker, M. McCourt, S. Clark, P. Hayes, A. Johnson, and G. Ke, "A Strategy for Ranking Optimization Methods using Multiple Criteria," *Workshop on Automated Machine Learning (AutoML), ICML*, pp. 1–12, 2016.

[9] J. Kim, "AutoML Challenge : AutoML Framework Using Random Space Partitioning Optimizer," *Workshop on Automated Machine Learning (AutoML), ICML*, pp. 1–4, 2016.

[10] H. Kim and Y. W. Teh, "Scalable Structure Discovery in Regression using Gaussian Processes," *Workshop on Automated Machine Learning (AutoML), ICML*, pp. 1–11, 2016.

[11] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar, "Efficient Hyperparameter Optimization and Infinitely Many Armed Bandits," *arXiv preprint*, 2016.

[12] J. Snoek, H. Larochelle, and R. P. Adams, "Practical bayesian optimization of machine learning algorithms," in *Advances in neural information processing systems*, 2012, pp. 2951–2959.

[13] J. Vanschoren, J. N. van Rijn, B. Bischl, and L. Torgo, "Openml: Networked science in machine learning," *SIGKDD Explorations*, vol. 15, no. 2, pp. 49–60, 2013.