Democratizing Data Science through Interactive Curation of ML Pipelines

Zeyuan Shang[1] Emanuel Zgraggen[1] Benedetto Buratti[2] Ferdinand Kossmann[1]
Philipp Eichmann[2] Yeounoh Chung[2] Carsten Binnig[2,3] Eli Upfal[2] Tim Kraska[1]
[1]Massachusetts Institute of Technology, USA [2]Brown University, USA [3]TU Darmstadt, Germany

ABSTRACT

Statistical knowledge and domain expertise are key to extract actionable insights out of data, yet such skills rarely coexist together. In Machine Learning, high-quality results are only attainable via mindful data preprocessing, hyperparameter tuning and model selection. Domain experts are often overwhelmed by such complexity, de-facto inhibiting a wider adoption of ML techniques in other fields. Existing libraries that claim to solve this problem, still require well-trained practitioners. Those frameworks involve heavy data preparation steps and are often too slow for interactive feedback from the user, severely limiting the scope of such systems.

In this paper we present *Alpine Meadow*, a first *Interactive Automated Machine Learning* tool. What makes our system unique is not only the focus on interactivity, but also the combined systemic and algorithmic design approach; on one hand we leverage ideas from query optimization, on the other we devise novel selection and pruning strategies combining cost-based Multi-Armed Bandits and Bayesian Optimization.

We evaluate our system on over 300 datasets and compare against other AutoML tools, including the current NIPS winner, as well as expert solutions. Not only is *Alpine Meadow* able to significantly outperform the other AutoML systems while — in contrast to the other systems — providing interactive latencies, but also outperforms in 80% of the cases expert solutions over data sets we have never seen before.

ACM Reference Format:
Zeyuan Shang[1] Emanuel Zgraggen[1] Benedetto Buratti[2] Ferdinand Kossmann[1] Philipp Eichmann[2] Yeounoh Chung[2] Carsten Binnig[2,3] Eli Upfal[2] Tim Kraska[1] . 2019. Democratizing Data Science through Interactive Curation of ML Pipelines. In *2019 International Conference on Management of Data (SIGMOD '19), June 30-July 5, 2019, Amsterdam, Netherlands.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3299869.3319863

1 INTRODUCTION

Truly democratizing Data Science requires a fundamental shift in the tools we use to analyze data and build models [18]. On one hand it requires to move away from Python-like scripting languages, SQL and batch processing to visual and interactive environments [10, 15, 21, 26, 33, 43]. On the other hand, it requires to significantly reduce the required expertise to build a machine learning pipeline. Ideally, a user should be able to specify a high-level task (e.g., predict label X based on my data), and the system automatically composes a machine learning pipeline to achieve that task, including all necessary data cleaning, feature engineering, and hyper-parameter tuning steps.

The latter challenge is largely referred to as *AutoML* or *Learning to Learn* and comes in various flavors. For example, there already exists a huge amount of work on a subset of the problem: automatic hyper-parameter tuning and model family selection. Most noticeable, TuPAQ [35, 37], Hyperband [22] and the various Bayesian Optimization approaches [11, 17, 40] all have the goal to automatically determine the best model family (e.g., SVM vs Linear regression) or parameters for a given algorithm (e.g., step-size, kernel, etc.). However, hyper-parameter and model selection is only one aspect of automatically finding the best ML pipeline for a given task. Rather an end-to-end solution also has to consider data cleaning operation, feature engineering, and potentially even data augmentation and transfer learning. For example, in some cases min-max scaling and feature crosses might help, whereas in others standard scaling and feature selection to avoid over-fitting is the better choice. In some cases filtering out outliers and imputing missing values can have significant benefits, whereas in others it harms the accuracy.

The closest existing solutions, which allow such end-to-end training are probably the recent Learning to Learn approaches to find neural net (NN) architectures [3, 45]. The

SIGMOD '19, June 30-July 5, 2019, Amsterdam, Netherlands
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-5643-5/19/06…$15.00
https://doi.org/10.1145/3299869.3319863

view of some “purist” is that the input of a NN should be the raw data and that the model – if correctly tuned, for example, by an automatic NN architecture search – should do all the rest. However, deep learning based approaches only work with huge amount of training data and output a black box solution (i.e., a neural net), which is extremely hard to interpret. While this approach might be amenable for some scenarios, many real-world problems are rather small in terms of data size. For example, in the current DARPA D3M AutoML competition, only 5% out of the 300 datasets are actually larger than 10MB. We made similar observations when working with our partners in industry and hospitals.

More importantly though, we are not aware of a single AutoML solution, which can provide interactive response times to enable users to steer the computation and contribute to the optimization with their domain knowledge. For example, Google’s Architecture search can run for weeks [3], whereas even SciKit-Learn’s Hyperparameter Tuner often take hours before producing a first high-quality result. At the same time, interactive response times are key: users should see and understand how the system tries to find the best possible AutoML pipeline and potentially contribute their knowledge. For example, a doctor might decide to remove questionable features from the training set after seeing that the model starts to rely too much on it. Furthermore, as shown in interactive data exploration [44], interactive response times can improve the rate at which insights are uncovered: a team might try to build a model quickly during a meeting rather than having a week-long back and fourth between meetings, coding and running experiments, etc.

In this paper, we present *Alpine Meadow*, a first interactive AutoML tool, which is intended to be integrated into a visual environment similar to Tableau or Vizdom [10]. However, for this paper our focus is entirely on the ML optimizer rather than the visual integration and user feedback. Furthermore, we have a particular focus on small data and traditional statistical supervised machine learning pipelines, rather than architecture search for neural nets, unsupervised learning, or automatic data acquisition and cleaning. While the here described optimization framework can be easily extended with these operations, and in fact, our implementation already does support many of them (e.g., transfer learning for neural nets, unsupervised learning) describing and evaluating these operation in detail is beyond the scope of this paper.

Interestingly, the problem of finding the best possible ML pipeline for a given task (e.g., classify X) has many commonalities with query optimization as already pointed out in the MLBase vision paper [19]. It requires to explore a potentially enormous search space and select the best possible plan (i.e., pipeline). We therefore borrow many ideas from query optimization including rule-based search-space creation. Yet, what differentiates our approach the most from other AutoML tools is the joint algorithmic and system-based approach to ML auto-tuning, the focus on interpretable ML pipelines, and our goal to produce a high quality results in less than a few seconds.

In summary, our end-to-end interactive and automated machine learning system makes the following contributions:

- We present a novel architecture of an AutoML system with interactive responses.
- We show how rule-based opimization, can be combined with multi-armed bandits, Bayesian optimization and meta-learning to find more efficiently the best ML pipeline for a given problem. Here, the novelty lies in the fact how we combine the various techniques into a single system.
- We devise an adaptive pipeline selection algorithm to prune unpromising pipelines early by comparing train and validation errors on increasingly larger sample sizes of training instances.
- We show in our evaluation that *Alpine Meadow* significantly outperforms other AutoML systems while — in contrast to the other systems — provides interactive latencies on over 300 real world datasets. Furthermore, *Alpine Meadow* outperforms expert solutions in 80% of the cases for datasets we have never seen before. Finally, as of April 2019 *Alpine Meadow* was ranked first in DARPA performed D3M Automatic Machine Learning competition.

The remainder of this paper proceeds as follows. In Section 2 we provide a system overview, whereas Section 3 to 7 discuss the different auto-tuning steps. We evaluate our system and compare with baselines and other systems in Section 8, summarize related works in Section 9, and finally conclude in Section 10.

2 OVERVIEW

In this section we give an overview of *Alpine Meadow* and introduce the main terminology.

2.1 System Architecture

Alpine Meadow is part of *Northstar*[18], a system for Interactive Data Science where domain experts interact with data through an interactive visual environment called *Vizdom*[10]. In this environment, a prediction problem can be specified through drag and drop gestures and can be as simple as binary classification (i.e. spam detection) or as complex as graph community detection.

Based on such a problem specification, *Alpine Meadow* will automatically begin to search and progressively return machine learning pipelines to the end-user. The system gradually optimizes over the search space, and periodically returns best-so-far pipelines to the end-user. Unlike other AutoML systems, we envision our system to be used in an interactive setting, which allows users to constrain and refine a problem, early stop a search and embed their domain knowledge.

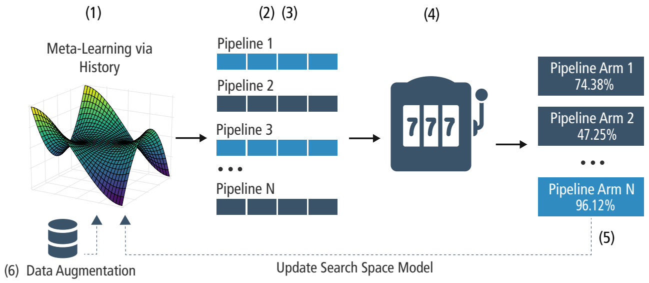


Figure 1: Optimization loop: (1) search space model, (2) logical-plan selection, (3) physical-plan selection, (4) pipelines evaluation and pruning, (5) search space model update, (6) data augmentation

2.2 The Optimization Process

The core design idea is to solve ML problems by emulating the decision-making process of an experienced data scientist. How does an experienced data scientist approach a problem: First, she would inspect the data and, based on her experience, make high-level decisions about feature scaling, embeddings, data cleaning, etc. The key is to start out simple. Furthermore, the data scientist would probably use a reliable and often successful model family, such as random forests, and check for the most common mistakes (e.g., imbalance of labels or duplicate label columns). Finally, the data scientist would setup a simple optimization strategy for the primitives' hyper-paremeters and if the data is large, probably first try to build a model over a sample of the data. Then, after initial results, the data scientist will start to modify the pipeline by adding more complex processing steps, changing the model family, adding/removing features, increasing the sample size and so on. It is an iterative and incremental process. It is further a process with memory as the data scientist remembers, what worked well over what data in the past.

This process is exactly what we aimed to mirror and automate in our system. We therefore broke our architecture up into steps that data scientists perform, which has the advantage to make the problem more tractable than treating it as optimization problem on a gigantic and heterogeneous space. Figure 1 shows the individual steps in *Alpine Meadow*:

(1) Search Space: The system first creates a search space of *logical pipelines*. We define a *logical pipeline* plan as:

Definition 1. ***Logical Pipeline Plan:*** *a Directed Acyclic Graph (DAG) of primitives, with their hyper-parameters' domain specification (not fixed).*

We create the logical plans through applying rules, similar to how SQL transformation rules can create a space of equivalent logical query plans. For example, a rule might say that all categorical features should be one-hot encoded, or that numerical features can be scaled. Also similar to logical query plans, *logical pipelines* do not yet contain any details about how the pipeline should be executed (e.g., no hyper-parameters are set).

This step is best compared to asking the data scientist: "What can I do to predict X based on my data" and she lists a whole bunch of options, e.g., different ways of encoding categorical features, scaling numerical features and feature selection, and different models for prediction.

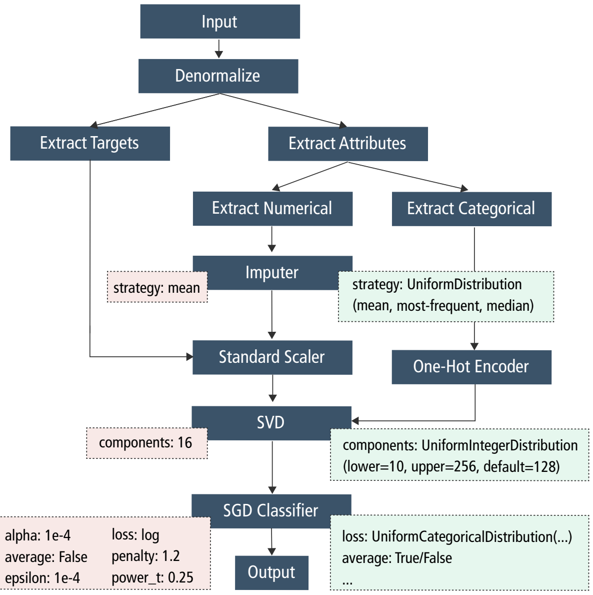


Figure 2: An example pipeline. The boxes in red show fixed hyper-parameters and they compose a *physical pipeline* plan with this DAG. While the boxes in green give distribution of hyper-parameters and they compose a *logical pipeline* with this DAG.

(2) Logical Pipeline Selection: Similar to query optimization the space of all possible *logical pipelines* can be huge. We therefore select the most promising *logical pipelines* based on a cost/quality model learned from past experiments to favor fast pipelines to provide better interactivity.

This step is best compared to asking the data scientist "What should I try first". A data scientist will provide you with a few good general options after taking a quick look at your data. For example, she might say "Try to normalize all features and use a boosted decision tree as a start" or she might say, "Given the data size, don't even try neural nets".

(3) Physical Pipeline Selection: After selecting the *logical pipelines*, they are instantiated into *k physical pipeline* plans, which are defined as:

Algorithm 1: PipelineSelection

```
  Input: Problem $\mathcal{P}$, Dataset $\mathcal{D}$, $Q$
1 while $Q.has\_space$ do
2     lp ← NextLogicalPipeline($\mathcal{P}$, $\mathcal{D}$)
3     pps ← NextPhysicalPipelines(lp, $k$)
4     $Q$.putAll(pps)
```

Algorithm 2: PipelineExecution

```
  Input: Problem $\mathcal{P}$, Dataset $\mathcal{D}$, $Q$, $score_{best}$
  Output: Pipeline Found
1 while $!Q.empty$ do
2     p ← $Q$.take()
3     for $score$ ← $AdaptivePipelineSelection(p, \mathcal{D})$ do
4         if $score > score_{best}$ then
5             $score_{best}$ ← score
6             yield $pipeline_{best}$
7     Update models using running history of pipeline;
```

Definition 2. ***Physical Pipeline Plan:*** *an end-to-end solution to a user-defined problem, represented as Directed Acyclic Graph (DAG) of primitives with fixed hyper-parameters.*

An example of physical pipeline plan is shown in Figure 2. *Physical pipelines* are generated from a *logical pipeline* via Bayesian optimization. Each *logical pipeline* hyper-parameters space has an associated performance-model used to find promising configuration. If a *logical pipeline* has never been used, there is not any model associated with it, hence we start out using default or random configurations. As soon as the first results are collected, our system starts to select the next hyper-parameters based on Bayesian-Optimization. The logical and physical plans are a vague analogy to the query optimization, however, *physical pipeline* plans don't include any implementation details as the physical plans in query optimization do.

This step is best compared to turning the general pipelines into actual Python code.

(4) Incremental Execution: For large datasets, it is often beneficial to run a *physical pipeline* on a smaller sample first, and then if the results look promising try it on a larger portion of the dataset. We therefore, treat every physical plan as a bandit arm, from which every pull increases the sample size. The bandit mechanism together with the sampling guarantees that we focus our attention on promising pipelines early on and get good results quickly, which we can stream back to the user with short response time.

This is similar to a data scientist first building a model over a sample of the data before using all available data especially when the data is big.

(5) Iterative Refinement: By evaluating different *physical pipelines*, we gathered some experience over the current dataset that we can use to update our cost- and quality-model to select *logical pipeline* and the Bayesian-Optimization model for selecting *physical pipeline*.

This step can be best compared to the iterative refinements that a data scientist performs after that she observes the results from a tested model.

(6) Data Augmentation: A more recent step that we started adding to the process is automatic data augmentation. That is, as part of step (1) we now also consider, if we can use already trained models as starting solutions or to create new features. For example, if the goal is to train a classifier based on only 100 training images, the most promising solution is to actually transfer an existing model or use existing models to create more powerful features for the given images. Currently, we only use this approach for image tasks but with very remarkable results. While not discussed in detail in this paper, we briefly outline that our system can easily be extended to support this.

As we showed, *Alpine Meadow* tries on a high-level to emulate the steps a data scientist takes. Furthermore, as the distinction between a *logical pipeline* and *physical pipeline* already shows, our optimizer has many similarities with traditional query optimization and a lot of optimization potential exists as we discuss in the remaining sections. It should also be noted, that this is not the only way to build an interactive end-to-end AutoML tool and in Section 7 we discuss alternative designs. However, like the original Selinger paper [32] on query optimization, it is a start, and will hopefully result in various follow up work.

2.3 Algorithmic Walkthrough

Algorithm 1 and 2 provide a simplified outline of the entire optimization process following the previous described steps (minus the augmentation). First, we create a master *Pipeline Selection* thread running Algorithm 1, and several *Pipeline Execution* worker threads. The two are connected through a fixed size execution queue Q. Every time the queue has free space, the *Pipeline Selection* thread tries to find a promising *logical pipeline*, and based on it creates k *physical pipelines* to execute, which it then adds to the execution queue.

Whereas the worker threads take up a *physical pipeline* from the queue and execute it using our sample-based execution strategy (line 3 in Algorithm 1). Note, that for a single *physical pipeline* we receive more than one scores, as we incrementally train and test the sampled pipelines. If the score for a pipeline is higher than the so far best seen score, we report it to the user (line 4-7) and update our history of pipeline runs to make better decisions in the future (line 7).

3 RULE-BASED SEARCH SPACE

Data scientists rely on their expertise and past experience to solve challenging problems. We imitated this process by adapting the idea of rule-based search space definition commonly used in database optimizers to our AutoML system. Rules in our system encapsulate best practices similar to those data scientists might use. Given the definitions in the previous section we propose three kinds of rules: *primitive*, *parameter* and *enforcement rules*.

Primitive Rules add new primitives to the search space dependent on the task (e.g., using different algorithms for classification, regression, recommendation, or graph-related problems) or the dataset schema (e.g., applying one-hot encoding for categorical features). Until now, we have integrated close to hundred primitive rules derived from winning Kaggle competitions, expert solutions to problems provided by DARPA, and interviews with data scientists. These rules, for example, include things like encoding categorical features, scaling numerical values, imputation of empty values, selection of features, choosing models for different problem types, extracting features from raw text and images, building the graphs for graph datasets etc. Primitive rules are used to build and rewrite *logical pipelines*. Applying a rule can either start a new *logical pipeline* or extend existing ones by adding primitives that operate on all or a subset of columns. What makes our approach unique is that we create two types of *logical pipelines*:

- **General *logical pipeline***: General pipelines always use primitives over all features if they share the same semantic type, and only use one primitive type per category. For example, a general pipeline would encode that we run a one hot encoder on all *categorical* columns, a min-max scaler on all *numerical* columns, then do an SVD on the concatenation of these two results, and feed them into a SVM. A general pipeline would thus not use two different encodings for the same numeric feature, or first apply min max scaling followed by standard scaling. This approach allows us to severely restrict the number of general *logical pipeline* and also make the transfer learning of pipelines between different datasets possible.
- **Data-specific *logical pipeline***: These are *logical pipelines* with no restrictions on the primitive compositions and can be dataset dependent. For example, for a problem of predicting whether a player can be selected into the hall of fame, we can run a standard scaling on the number of seasons of the player, and a min max scaling on the average scores of the player. Obviously, for any given problem, there can be a large amount of *data-specific logical pipelines*.

Parameter Rules generate reasonable distributions for hyper-parameters of primitives. For example, a rule might be that the set of possible values for the kernel of a SVM are *linear*, *poly*, *sigmoid* or *rbf*, or that the value for the regularization factor λ should be sampled from a log uniform distribution.

Enforcement Rules check the feasibility of a *logical pipeline*. Not every generated *logical pipeline* is feasible. For example, most algorithms will fail if not all the categorical features are encoded into numerical values or raw data (e.g., text) are not featurized. *Alpine Meadow* uses enforcement rules to validate *logical pipeline* and aborts the generation of unfeasible ones.

For execution of primitive rules, we have the probability of γ to create general *logical pipelines* or data-specific *logical pipelines*. In our implementation, γ is set to 0.5. We only return a *logical pipeline* when it passes all the enforcement rules, and users have the opportunity to affect our selection of *logical pipeline* here, for example, we can add a enforcement rule to only allow for *logical pipeline* with SVMs or *logical pipeline* with no more than 10 steps. After that, we execute parameter rules to assign reasonable distributions of hyper-parameters for primitives of a *logical pipeline*. Before applying any rule, we always check the predicate of the rule to make sure it works for the given problem and dataset.

By applying rules to build the search space, we make the generation of *logical pipeline* plans flexible. It allows to add new rules to extend the system to support new problems, datasets and incorporate best practises from machine learning experts. Moreover, rules also create easy-to-explain solutions for better interpretability by users; especially general *logical pipeline* are often easy to understand. Furthermore, it allows to inspect which set of rules led to the creating of a specific *logical pipeline*.

Finally, rules can be learned and automatically added. In the simplest form, we add a new expansion rule for every newly-added primitive. For example, if one adds a new feature scaler for numeric value, we add a rule that the optimizer can use this new feature scaler for numeric values. However, it is possible to use the rules to apriori restrict the search space (e.g., only use this feature scaler if the classifier is an SVM) and these rules could be learned from Kaggle and OpenML. In our current implementation, we do not make such restrictions and leave it up to the meta-learning algorithm to make the right choices early on.

4 PIPELINE LOGICAL PLAN SELECTION

Ideally, we want to select pipelines from the search space, which worked well in the past over similar datasets. However, occasionally we want also try out new approaches (e.g., an estimator that we never tried before). Furthermore, we should probably favor solutions in the beginning, which are more general, fast and reliable, but later specialize and use more complex models. Finally, we can not enumerate all

Algorithm 3: NextLogicalPlan (NLP)

```
Input: Problem $\mathcal{P}$, Dataset $\mathcal{D}$
Output: Next logical pipeline
1 if rand() < $\beta$ then          // Selection (Exploitation)
2     Compute $\mu_k$, $\delta_k$ and $c_k$ for each logical pipeline k
        using the history
3     LogicalPlan ← select a logical pipeline k with a
        probability proportional to $\mu_k + \frac{\theta}{c_k} \cdot \delta_k$
4 else                               // Random (Exploration)
5     if random() < $\gamma$ then         // General pipeline
6         LogicalPlan ← general logical pipeline
7     else                           // Data-specific pipeline
8         LogicalPlan ← data-specific logical pipeline
9 return LogicalPlan
```

potential pipelines; so any strategy has to use some kind of heuristic to traverse the search space.

Obviously, there is no single "right" way to balance all these objectives. In the following, we first describe on a high level how our selection process works, before we discuss the individual components in more depth.

4.1 Overview

The most important difference between building an AutoML optimizer and query optimizer is that for ML pipelines we can actually try and evaluate hundreds if not thousands of pipelines, while in query optimization once a plan is executed there is nothing left to try out. The goal of our optimizer is to select and try out various logical plans in a way that maximizes the probability that one of them contains the best possible *physical pipeline*: often *logical pipelines* diversity can help. Furthermore, it is a iterative process: we can stop the evaluation of a pipeline at any point in time and start a new one as it deems fit; something which rarely pays off in traditional query optimization, but which is common practice for ML. Our goal is therefore to build a function called *NLP*, short for *next* logical pipeline, which we invoke to obtain promising *logical pipelines*. More importantly, we found that using past history is the best predictor for future performance and thus balancing exploitation (leveraging what worked well in the past) and exploration (trying out new things) are key to finding good solutions. The high-level pseudo-code for selecting the next *logical pipeline* is shown in Algorithm 3.

Exploitation To balance the two objectives, exploitation and exploration, we use a simple random process: with likelihood β, we select a general *logical pipeline*, which worked well in the past (lines 1-4 in Algorithm 3). We evaluated β over various datasets (see Appendix A.1) and found that $\beta = 0.5$ provides a good balance. We steer exploitation based on a score measuring how promising each *logical pipeline* is, while the score is calculated based on past experiences. We restrict transferring past experience to general *logical pipelines* as we found that it is less reliable for data-specific pipelines because of the sheer amount of options and the sensitivity to the dataset. Therefore, *Alpine Meadow* stores information about every *physical pipeline* ever run including its corresponding *logical pipeline*, final accuracy, execution time, task information, and dataset characteristics. This allows us, for example, to calculate the average and variance of the accuracy and execution time of a model for a given task and set of data characteristics. Based on this historic information and given a new task, *Alpine Meadow* then creates a score of every previously run general pipeline. This ranking is based on the execution time. That is, in the beginning we rank *logical pipelines* higher which return quickly, whereas later execution time might be less of a concern. Finally, its selects randomly one of the pipelines depending on the score: the higher the score, the higher the chance that the general pipeline gets selected. Furthermore, in the moment we receive results on how well a selected *logical pipeline* performs, this information is also stored, which in turn might change the scores for the next selection.

Exploration In contrast to ensure that *Alpine Meadow* also tries new things, with the likelihood $1-\beta$ we select a *logical pipeline* which we have never run before. Here we again randomly select with likelihood γ either a general *logical pipeline*, or with likelihood $1-\gamma$ a data-specific pipeline (lines 4-9 in Algorithm 3). We evaluated γ over various datasets (see Appendix A.1) and found that $\gamma = 0.5$ provides a good balance. Note, that by adjusting γ over time, we can favor general pipelines in the beginning and maybe later in the execution split it evenly between general and data-dependent pipelines, which are more specialized. For example, with a large γ, we prefer general *logical pipelines*, then we are more likely to generate general ones like the pipeline in Figure 2. With a smaller γ, data-specific pipelines are more likely chosen, while they are usually more complicated, e.g., run min-max scaling on one column and standard scaling on another column, followed by a PCA. Many ways exists on how to select the potential *logical pipeline* for which we have no experience yet. However, what we found is that randomly selecting a solution often performs as good as a more advanced techniques. The reason is, that the number of general pipelines is relatively small, so that we will anyway try them all in a short amount of time, if γ is not set too low. In contrast, the number of data-specific pipelines is very sensitive to the data properties (much more than the general pipelines) and the search space is so big, that we can often not create enough samples that any advanced optimization technique would actually pay off.

Finally it should be noted, that this selection process does not yet tune any of the hyper-parameters and that for every logical plan we usually try several hyper-parameters, as explained in Section 5.

4.2 Selecting Based on History

In this section we focus our attention on how we select a general pipeline from the past (lines 1-4 in Algorithm 3). We modeled this selection process as a **Multi-Armed Bandit** (MAB) problem. We adapt the definition of MAB presented in [34] to the concept of *logical pipeline* selection as follows:

Definition 3. ***Multi-Armed Bandits (MAB) Problem:*** *given a set of actions $a \in \mathcal{A}$ and a time-budget T, in each round $t \in [T]$:*

(1) An algorithm picks an arm $a_t \in \mathcal{A}$
(2) Algorithm observes a reward from the chosen arm a_t

Given that the arms reward distributions $\mathcal{D}_\mathcal{A}$ are unknown and independent, find the algorithm that approximate the best solution with the smallest reward loss (regret)

We base the selection of past pipelines on MAB as many algorithms exists approximating the optimal solution; among the most known are *Upper Confidence Bound* (UCB) [8] and *ϵ-greedy* [39]. This provides us a powerful and proven solution.

Selecting With Bandits The core idea is as follows: **(0 - Init)** We have one arm for every related (based on the task and dataset) general *logical pipeline* we ran in the past and we preset a score for each arm based on our past experience. **(1 - Selection)** We select an arm (i.e., logical plan) to play (i.e., run) randomly but proportional to the score. When the execution is done, we **(2 - Store History)** store the result in our history log and **(3 - Adjust Scores)** adjust the scores accordingly, and then the process repeats from (1).

They are four core problems we have to address (1) how we select arms (i.e., pipelines) based on similarity of the task and data, (2) how we define the score, (3) how we transfer the past observed performance to the current dataset and task, and (4) how we adjust the score based on the feedback we get of actually running pipelines for the given tasks. We will address those challenges in this order.

4.3 Remembering the Past

To find related history for a given task and datasets can be regarded as a meta-learning problem. *Meta-learning* [7] tries to infer learning algorithms performances from the performance of learning algorithms across different datasets. We use the same idea from meta-learning to quantify the similarity between datasets. [12] proposes many meta-features to capture the high-level characteristics of a dataset. Those include: number of features, the imbalance ratio of classes, the number of instances, PCA statistics and information-theoretic features etc.

Further, [12] proposes a distance function based on the performances over a fixed set of n representative pipelines on two datasets. Formally, assume that there are n pipelines $(\theta_1, \ldots, \theta_n)$, we use the negative Spearman's correlation coefficient between the ranked results on both datasets (denoted as d_c):

$$d_c(D_i, D_j) = 1 - Corr([f^{D_i}(\theta_1), \ldots, f^{D_i}(\theta_n)], \\ [f^{D_j}(\theta_1), \ldots, f^{D_j}(\theta_n)])$$

where $f^{D_i}(\theta_1)$ denotes the computed score after evaluating pipeline θ_1 on D_i.

For a new dataset D_{new}, since we have not yet evaluated these n reference pipelines, we can not directly compute d_c. However, assume there are N pre-provided datasets, [12] addressed this by computing $d_c(D_i, D_j)$ for all $1 \leq i, j \leq N$ and using regression methods to learn a function $R : \mathbb{R}^F \times \mathbb{R}^F \rightarrow \mathbb{R}$, mapping from pairs of meta-features $< m^i, m^j >$ to $d_c(D_i, D_j)$. Then with this learned model, the distance function can be approximated as

$$d_c(D_{new}, D_i) \sim R(m^{new}, m^i)$$

In our implementations, we built R using a random forest because of its robustness.

With distance function d_c, we can get the list of applicable history (i.e., *logical pipelines* and their performances) from similar datasets for a given dataset D_{new}, such that the dataset D associated with the instance of history has $d_c(D_{new}, D) = R(m^{new}, m) \leq \tau$. In our implementation, we use $\tau = 0.3$ and return all pipelines below the threshold together with their mean performance μ, performance variance δ, and averaged execution time c. Each of these pipelines become a bandit arm, which can then be executed.

4.4 The Scoring Model

We want to balance the expected quality vs time. We therefore defined the score for each *logical pipeline* plan as:

$$s_k = \mu_k + \frac{\theta}{c_k} \cdot \delta_k \tag{1}$$

where μ_k and δ_k are the mean and standard deviation of the rewards (i.e., quality of the *logical pipeline* plan) and c_k is the cost, or execution time, for *logical pipeline* a_k based on the past history. Note, that we divide only the variance by the execution time and multiply it by θ. Here θ is a factor on how much risk we want to take to try a pipeline, which might have a high upside (i.e., variance). We normalize the variance by the execution time as proposed in [23]; so the higher the potential payback, the longer we are willing to wait for it. However, we do not adjust the mean reward by the execution time. A pipeline which always performs good should be selected from the beginning. However, this is only reasonable as (1) we assume a high parallelism (explained later), (2) assume that some short running pipelines will always be selected, and (3) our physical execution quickly prunes out long running under performing *physical pipelines*.

The last step to achieve a complete solution to the logical plan selection problem is the initialization of the scores based on the history, which involves two main challenges: (1) we have to identify the similar tasks and according dataset from the past (i.e., learning from the past), and (2) normalize the scores to the new problem so that they can actually be used (i.e., transferring the experience).

4.5 Transferring the Experience

While it seems that we can immediately use the score formula above and fill it using the values from the history, we actually can not. Even for similar datasets, the same *physical pipeline* may have different scales of scores (since we only consider relative rankings to quantify the similarity between datasets). To this end, for all pipelines of a specific dataset, we can standardize their scores to fit into the same scale. For a pipeline p and its score s, we normalize it as

$$s_{new} = \frac{s - \mu_d}{\delta_d}$$

where μ_d and δ_d are the mean and standard deviation of scores of all pipelines on this dataset. We cluster these past iterations and their normalized scores to their corresponding *logical pipelines*, which has the same structure (DAG). Then we compute the mean $\mu_k^{'}$ and standard deviation $\delta_k^{'}$ for each *logical pipeline* k using these normalized scores.

4.6 Learning from the Current Experience

Finally, we want to adjust the scores based on the actual feedback by running pipelines over the actual data. Assuming μ_k is the just observed new mean quality and $\mu_k^{'}$ is the old mean quality. We then calculate the new value for $\hat{\mu}_k$ by first normalizing the score and by means of exponential smoothing as:

$$\hat{\mu}_k = \frac{\mu_k - \mu}{\delta} + \alpha \cdot \mu_k^{'} \tag{2}$$

where α achieves the trade-off between current results and history results. In our implementation, we set $\alpha = 0.2$. In the future, we plan to make α degrade over time to prioritize information from current session. The adjustment for the variance δ_k and execution time c_k is very similar.

Furthermore, all pipelines (general and data-specific) which get selected for execution, will become a new arm as soon as they return a first result. This has the advantage, that as soon as we try something new, it becomes part of the memory of our system and the bandit algorithm might select it based on its score in the future.

5 PIPELINE PHYSICAL PLAN SELECTION

For a specific *logical pipeline*, there exists possibly an infinite number of *physical pipelines* with different hyper-parameter configurations. In order to to find the best possible hyper-parameters configuration, we uses Bayesian Optimization:

Definition 4. ***Bayesian Optimization (BO)****: optimization strategy that finds a global optimum point $x^* \in \mathbb{X}$ of a function $f : \mathbb{X} \to \mathbb{R}$ (which analytical form is unknown), building a surrogate model $\mathcal{M}_f$ of f to guide the optimization.*

In our setting, f is the unknown score function that maps the *physical pipelines* with their respective performances. Since the evaluation of f is expensive, we use Bayesian Optimization, specifically *Sequential Model-Based Optimization* (SMBO), to build a model of f to keep track of which are the most promising regions in the search space.

For every selected *logical pipeline*, the optimizer probes $\mathcal{M}_f$ using a *sampling policy* to find the next most promising hyper-parameter configuration to be evaluated. In our work we use the widely adopted *expected improvement* (EI) [31] as *sampling policy*, due its ability to balance exploration (search in unexplored regions) and exploitation (search in promising regions) [16, 31, 38]. We adopt the implementation of SMBO from [14] which uses a random forest to build the surrogate model. This random forest is trained using past configurations performances and estimates for a new hyperparameter configuration θ, its *predictive mean* μ_θ and its *variance* σ_θ^2, to compute for each *physical pipeline* its expected improvement with respect to the current best.

Based on SMBO we create k *physical pipelines* for each *logical pipeline* (we set k as 10 in our implementation), and pushes them into a shared queue that is consumed by the evaluation module as described next. Here, again we not only pick the most promising hyper-parameters, but also introduce some random candidates in its *physical pipeline* candidate-list to avoid to get stuck in local optimum point. Once those *physical pipelines* evaluation is complete, their scores are returned and the $\mathcal{M}_f$ updated accordingly.

6 PIPELINE EVALUATION AND PRUNING

We implement the execution engine of *Alpine Meadow* using master-slave paradigm to allow scalable training and testing of *physical pipelines* and coordinate the work using a single producer, multiple consumer paradigm as shown in Algorithm 1 and 2. That is, we have a queue of physical plans to run of size m, which are picked up by execution workers. Every time k slots become available, the producer runs Algorithm 1, first select a logical plan (Section 4), based on it select k *physical pipelines* (Section 5) and insert them to the queue. Those *physical pipelines* are then picked up by the workers running Algorithm 2 and executed and the

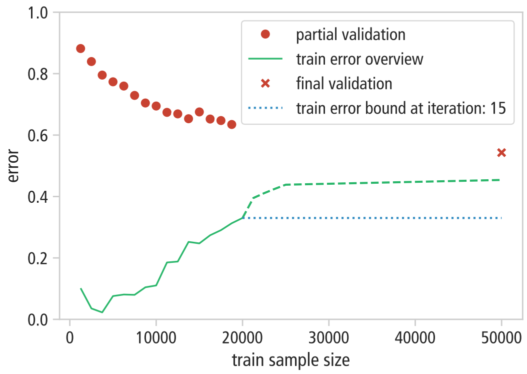


Figure 3: the more iid train samples we provide to the *physical pipeline*, the closer the train and validation error become.

```
Algorithm 4: Adaptive Pipeline Selection (APS)
   Input: Pipeline pipeline, $\mathcal{D}$
   Output: Score (negation of error)
1  Split $\mathcal{D}$ into $\mathcal{D}_{train}$ and $\mathcal{D}_{validation}$.
2  Split $\mathcal{D}_{train}$ into equal-sized $\mathcal{D}^1_{train}, \ldots, \mathcal{D}^N_{train}$;
3  foreach $i \in 1 \ldots N$ do
4      Train pipeline on $D^{1 \ldots i}_{train}$;
5      $err_{validation} \leftarrow$ Test pipeline on $D_{validation}$;
6      if $err_{validation} < err_{best}$ then
7          $err_{best} \leftarrow err_{validation}$;
8      yield $-err_{validation}$;
9      $err_{train} \leftarrow$ Test pipeline on $D^{1 \ldots i}_{train}$;
10     if $err_{train} > err_{best}$ then
11         return -inf
12 return $-err_{validation}$
```

process repeats. Note, that this approach only works well as we assume that the number of workers w we have is much larger than k, $w >> k$, and m is set to be larger than w.

However, two main problems remain: (1) how do we return results early based on samples to ensure interactivity, and (2) can we potentially stop the execution as soon as we detect that a pipeline is not promising. It turns out, those problems are actually related as explained next. Finally, our execution strategy has huge potential for result reuse, which is explained in Section A.5.

6.1 Incremental Execution and Pruning

To achieve incremental computation and return results early, as well as reduce the computational resources spent on bad pipelines we devised **Adaptive Pipeline Selection** (APS), a bandits-based pruning strategy able to detect bad performing pipelines, without using the whole training set.

As shown in Algorithm 4, APS gets as input a dataset $\mathcal{D}$, and it splits into a training dataset D_{train} and a validation dataset $D_{validation}$. Than splits the train-set into N smaller samples of the same size $\mathcal{D}^1_{train}, \ldots, \mathcal{D}^N_{train}$. For each sub-epoch, APS generates a partial training sample as follows:

Definition 5. ***Sub-epoch:*** *a partial training phase in which the sampled* physical pipeline *trains on a partial training sample. At sub-epoch i-th the partial training sample is equal to the union of the first $\mathcal{D}^1_{train} \cup \mathcal{D}^2_{train} \cup \ldots, \mathcal{D}^i_{train}$ data splits.*

As shown from line 4 to 6, after the partial training phase, APS computes $err_{validation}$ and updates $err_{validation}$ if necessary. For fast response, we will also return the score (negation of the error) to the main loop in Algorithm 2 to enable interactivity, therefore the master can make the decision of whether using such a pipeline trained on samples by comparing its score with the current best. After this it computes the *physical pipeline* partial training error and uses it as a lower bound of the final test error. Thus at the end each sub-epoch i, APS applies the following halting criterion:

Halting Criterion 1. *At sub-epoch i, if the* physical pipeline *partial training error is above the best validation error (seen so far), terminate it.*

The halting criterion is based on the facts that $err_{train} < err_{test}$ and they will eventually converge if enough data is provided (under the iid assumption). This idea is well displayed in figure 3 where the more data we provide to the *physical pipeline* the smaller the gap between train and test error becomes, making the bound on the final test error tighter and tighter. In the appendix we provide additional experimental evidence regarding the high correlation between train-test error, showing in the striking majority of the tested pipelines, the train error successfully bounds the final validation error.

The actual execution of APS is asynchronous. At each sub-epoch the individual threads compare their *physical pipeline* partial training and validation results against the current best performer, and if there is an improvement, they return it to the end user. APS saves computational resources by spending less on bad performing *physical pipelines* and more on promising ones and returning them faster to the end-user. *This has a direct impact on the system's interactiveness.*

7 DISCUSSION

Pipeline Selection: By combining multi-armed bandit and Bayesian Optimization (BO), our algorithm essentially adopts a two-step strategy: finding the best primitives (or *logical pipeline*), and then finding the best hyper-parameters (or *physical pipeline*). Another approach would be to take the structure of primitives as hyper-parameters, build a giant search space of all *logical pipelines*, and use BO over this space to find the best pipeline. However, such a search space is highly-heterogeneous and conditional, making it difficult to train an accurate regression model. For example, some hyper-parameters are specific to certain models but for BO to work all possible parameters need to be represented in a single feature vector. Updating the search space is also challenging. For example, if a new primitive (e.g., a new classification algorithm) is added, we cannot reuse previous history anymore as the feature space has changed. Moreover, it is very hard to find good optima with existing optimization methods when the search space is giant and highly complex.

Finally, it is also difficult to consider performance and cost at the same time in the traditional BO methods, and BO is also hardly explainable since it is essentially an optimization method for black-box functions. In contrast, multi-armed bandits provide a better intuitions what is happening. Therefore, by combing these two methods, we can inject context information (e.g., cost of executing *physical pipeline*) when solving the multi-armed bandit problem. By splitting the whole search space into several smaller ones (i.e., *logical pipeline*), we can avoid more quickly useless sub search spaces and build a more accurate BO model with less data since the complexity of search space is greatly decreased.

Interactivity: As *Alpine Meadow* is used in the interactive setting. In order to support interactivity, we employ time-based cost models that favor fast pipelines early on, train pipelines over small samples first, prune unpromising pipelines early, and even make extensive use of caching for our own developed operators as discussed in Section A.5.

Novelty over `auto-sklearn`: Our system is similar to `auto-sklearn` [11] as both use meta-learning and Bayesian Optimization. However, there are several key points where *Alpine Meadow* differs significantly from `auto-sklearn`: 1) *Alpine Meadow* uses a rule-based search space, which is more extensible and supports more problem types than `auto-sklearn` (see Figure 4). 2) *Alpine Meadow* combines multi-armed bandits with Bayesian Optimization (BO) to better explore the search space and improve interactivity. That is, `auto-sklearn` only returns one pipeline after a fixed time-budget while *Alpine Meadow* reports a stream of results with updates whenever a better pipeline is found. 3) The "warm-starting" techniques (i.e., the use of meta-learning) are different: `auto-sklearn` uses a few good pipelines as starting points, whereas *Alpine Meadow* uses the history of the quality and cost of all so far run pipelines. This allows us to tradeoff between performance and speed, leading to better performance in early stages. 4) *Alpine Meadow* adopts the Adaptive Pipeline Selection to prune unpromising pipelines at an early stage while `auto-sklearn` evaluates the pipeline on the full data, which makes it unable to produce results quickly, as justified in Figure 6(a).

8 EXPERIMENTS

We aim to answer three main questions: (1) How does our system compare to other state-of-art ML auto-tune systems? (2) Are we able to return answers more quickly than other systems, ideally with interactive latencies? (3) How much do our individual design decisions influence the system?

8.1 Experimental Setup

Datasets For the majority of our evaluation, we use the datasets provided by the DARPA D3M competition. DARPA's program on Data Driven Discovery of Models (D3M) has the goal to build tools to automatically build models for a given task with and without human feedback. As part of this program DARPA performs competitions every 6 month between all participating teams including teams from UC Berkeley, Stanford, MIT, NYU, etc. Every competition compares all the systems on datasets the teams have never seen before. However, in order to prepare the teams for the competition, DARPA released over 300 datasets; 220 classification datasets, the smallest being 151 records large, the largest being 1025000 records large, and 80 regression datasets, the smallest being 159 and the largest being 89640 records large. Here records refer to either tabular structured data, text-data, images, and even audio-files.

As mentioned before, our system heavily relies on the past experience to find good solutions. We therefore randomly split the datasets evenly into a training and test set; we use the training datasets to build up history, and only report the performance on the other half of the data.

Building Up Experience For the training datasets and tasks, we then extensively try out various pipelines to build up past experience. That is for every classification dataset we try 66 general logical pipelines, and for every regression dataset 44 regression pipelines. In total we spend 30 minutes per training dataset to build up sufficient experience to be used for future tasks. In total, we executed around half a million *physical pipelines*, which would take roughly 3 days on a single machine. However, the training is embarrassingly parallel and with 20 machines only takes 4 hours.

Baselines We compare against four baselines: (1) hand-made solutions from DARPA: while some DARPA solutions are state-of-the-art highly tuned solutions, others only represent reasonable solutions; a solution a relatively experienced data scientist can manually come up with in a few days; (2) `auto-sklearn` (version 0.4): automatically searched solutions from auto-sklearn [11], which is the state-of-art open-source AutoML system; (3) `TPOT` (version 0.9) : an interactive AutoML system using genetic programming [29]; (4) `Azure` (as of March 2019): Microsoft Azure AutoML (based on [27]). The experiments are restricted to AutoML, while feature engineering and other transformation primitives are not evaluated.

	Azure	auto-sklearn	TPOT	Alpine Meadow
Tabular Classification	99%	99%	87%	100%
Tabular Regression	100%	98%	100%	100%
Graph Matching	Not Supported	Not Supported	Not Supported	100%
Community Detection	Not Supported	Not Supported	Not Supported	100%
Image Classification	Not Supported	Not Supported	Not Supported	100%
Audio Classification	Not Supported	Not Supported	Not Supported	100%
Collaborative Filtering	Not Supported	Not Supported	Not Supported	100%

Figure 4: Comparison of *Alpine Meadow* with different systems regarding supported problem/dataset types. The percentages are calculated by the ratio of datasets supported by the system.

Metrics for Comparison We use F1 scores for classification problems and mean squared error for regression tasks. We further adopted the *normalized score* $norm_{AB}$ of system A over system B from DARPA D3M AutoML Competition:

$$norm_{AB} = \frac{s_A - s_B}{|s_B|}$$

Here the score S_A (S_B) is either the F1 score or the negation of the mean squared error, such that the higher scores are always better. Intuitively, $norm_{AB}$ measures how much better system A performs over system B. Note, that this normalized score is biased; the best possible score is 1 but the worst can be go to $-\infty$. We decided to use it as it DARPA's main measure, but *refrain from interpreting absolute values.* In addition to normalized scores, we also use relative ranks to compare different systems. For example, if system A gets a F1 score of 0.8 and system B a score of 0.9, the rank is 1 and 2 for system B and A respectively. *Here absolute values are more meaningful, thus we will use relative ranks, as well as discretized scores (as in Figure 5) to compare between different implementations.* Finally, we report an alternative unbiased metrics and raw scores in Section A.3 & A.4.

Hardware Environment All experiments were conducted on a single machine with a 40-core Intel(R) Xeon(R) CPU E5-2660 v2 @ 2.20GHz and 256 GB RAM, running with Ubuntu 16.04 and Python 3.6.3. We set the number of workers to 80 on this machine to utilize all hyper-threads.

8.2 Comparison with other Systems

Functionality: We first compared *Alpine Meadow* with auto-sklearn, TPOT, and Azure in terms of how many datasets they can handle (shown in Figure 4). We found that none of the other systems can handle image, audio, or collaborative filtering problems, whereas *Alpine Meadow* supports a wide range of problems. More surprisingly though, none of the other systems is even able to handle all structured classification and regression tasks.

Performance: Next, to evaluate the performance of the different systems over the 150 test datasets, we allocated each system a time bound of 1h, and for the comparison of *Alpine Meadow* and Azure a time bound of 10 minutes. One thing to note is that Azure didn't support F1 scores, so we use accuracy as the primary metric for classification problems for the comparison between *Alpine Meadow* and Azure. For all systems, we compute the normalized scores between *Alpine Meadow* and the respective system.

Higher scores mean *Alpine Meadow* outperforms the other system, whereas a normalized score of 0 means the systems perform equally well. We further discretize the normalized scores into "better': *Alpine Meadow* outperforms the other system, "same": scores are equal, and "worse": *Alpine Meadow* performs worse than the other system. Here we also only consider the datasets the system was actually able to run and exclude all datasets for which a system failed or didn't find a solution in the given time bound. The results of this experiment are summarized in Figure 5. Overall, *Alpine Meadow* outperforms or equals Azure in 70% of the datasets, 79% for auto-sklearn and 74% for TPOT.

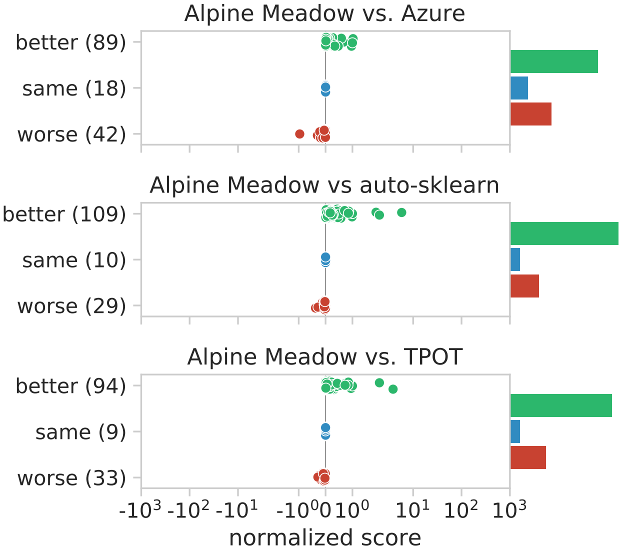
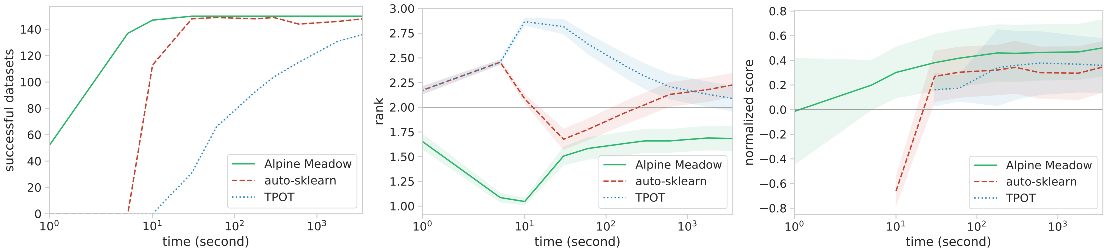


Figure 5: Comparisons of *Alpine Meadow* with different systems across multiple test datasets. Normalized scores are computed as *Alpine Meadow*'s score over the other system's score. Scores are discretized into "better': *Alpine Meadow* outperforms other system, "same": scores are equal, and "worse": *Alpine Meadow* performs worse than other system.

Comparison over Time Being able to provide solutions within interactive latencies is one of the main design goals of *Alpine Meadow*. We therefore measured the quality of the top models auto-sklearn and TPOT return over time. We excluded Azure from this experiment since they didn't support the F1 score and only recently in April 2019 were able to support all datasets. We ran all systems over our 150 test datasets for 1h again excluding failing datasets. Because auto-sklearn only returns the result after a pre-defined time span we run it with various increasing time limits.



(a) Number of Successful Datasets

(b) Ranks

(c) Normalized Score

Figure 6: (a) Time to produce first result per dataset (more early results implies better interactivity) (b) Relative rank of the solutions averaged over all datasets (with 95% confidence bands) over time (lower is better); (c) Normalized scores over the by DARPA provided solutions averaged over all datasets over time (higher is better).

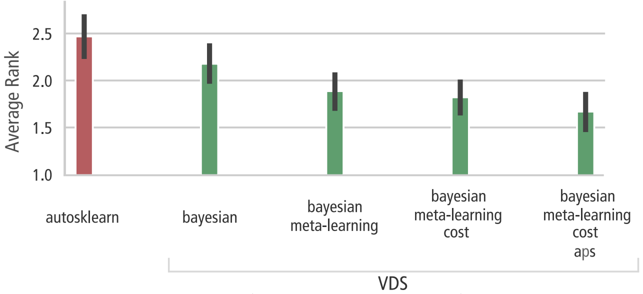


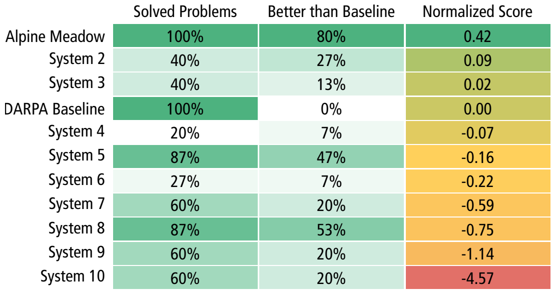
Figure 7: Incremental Comparison with `auto-sklearn`. We compare all these systems together and compute the averaged relative ranks (lower is better).

Figure 6(a) shows when the first result was returned by the individual systems. We note that *Alpine Meadow* is able to return solutions for over a third of the datasets within 1 second and for all datasets after 26 seconds with an average time per dataset of 2.76 seconds. This can be largely contributed to the adaptive execution strategy. The curve of `auto-sklearn` went down because results were collected from runs of different time limits, so it found a pipeline for some datasets in a run of short time while failed to do so in a run of long time.

Second, in Figure 6(b) we show how the relative average rank (over all test datasets) of the three systems evolves over time. Lower rank is better. *Alpine Meadow* consistently holds the best rank throughout the entire time span.

Finally, Figure 6(c) depicts the average normalized score (over all test datasets) where we normalize the system's score over the scores of hand-made solutions. Higher normalized score is better. We note that on average all three systems can beat hand-made solutions but *Alpine Meadow* is consistently leading especially within short time frames.

Incremental Comparison with `auto-sklearn` Figure 7 shows the incremental comparison (with more techniques employed) between *Alpine Meadow* and `auto-sklearn`. As we can see, if we only employ Bayesian Optimization in *Alpine Meadow*, the performance is relatively close to `auto-sklearn`, however, each individual technique (which is either not in `auto-sklearn` or employed in different ways)improves the performance of *Alpine Meadow*, including meta-learning, cost-based pipeline selection and adaptive pipeline selection.



	Solved Problems	Better than Baseline	Normalized Score
Alpine Meadow	100%	80%	0.42
System 2	40%	27%	0.09
System 3	40%	13%	0.02
DARPA Baseline	100%	0%	0.00
System 4	20%	7%	-0.07
System 5	87%	47%	-0.16
System 6	27%	7%	-0.22
System 7	60%	20%	-0.59
System 8	87%	53%	-0.75
System 9	60%	20%	-1.14
System 10	60%	20%	-4.57

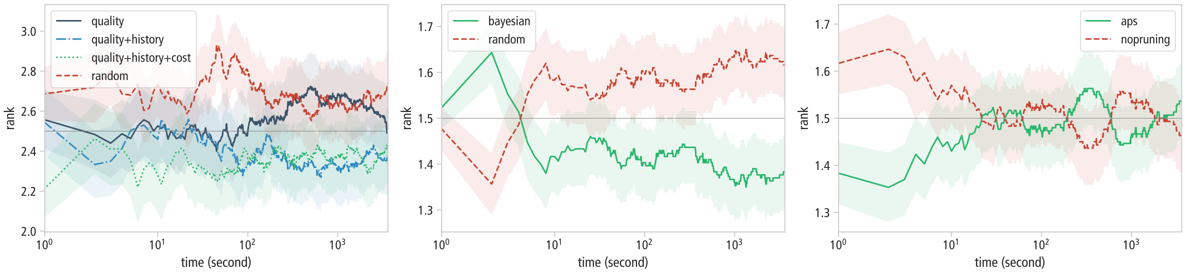
Figure 8: DARPA D3M AutoML competition (latest result in March 2018).

DARPA D3M competition As mentioned earlier, as part of DARPA D3M's program, DARPA evaluates the auto-ml solutions of all teams roughly every 6 month over datasets we have never seen before and also against by DARPA created expert solutions. Figure 8 shows the released results from the last DARPA evaluation which was done March 2018 (DARPA did another evaluation over the summer, but still hasn't released the results yet). In the table we anonymized the other team names, which are from places like Stanford, UC Berkeley, NYU, etc, and report the number of problems the system can solve, if the system is better than the by DARPA created expert solution, and the normalized score to the DARPA expert solution. As it can be seen, currently *Alpine Meadow* leads the competition.

8.3 Evaluation of Design choices

To evaluate our design choices, we ran our system for one hour while enabling or disabling individual components or optimizations. By comparing the results between with and without individual design choices, we can have a better understanding of their benefits. The results are shown in Figure 9.

Logical Pipeline Plan Selection We ran our system with four different configurations to justify the effectiveness of our cost and quality based *logical pipeline* selection techniques: (1) `Random`, which always picks up a *logical pipeline*



(a) Logical Pipeline Plan Selection

(b) Physical Pipeline Plan Selection

(c) Pipeline Pruning

Figure 9: Evaluation of Design Choices. We reported the ranks of different choices along with time (lower better).

randomly; (2) `Quality`, which only considers quality without using history (cold-start) when selecting *logical pipelines*, and also has some probability to randomly select a *logical pipeline*; (3) `Quality+History` extends `Quality` by using history of similar datasets to improve the selection; (4) `Quality+History+Cost` further improves `Quality+History` by considering cost to prioritize fast pipeline.

As we can see in Figure 9(a), at the early stage, `Quality`, `Quality+History` and `Quality+History+Cost` both outperforms `Random`, it is because them all choose pipelines with high potential of quality. By taking history into consideration, `Quality+History` is able to find good results after the first 100 seconds, however, because it doesn't consider cost, it prefers good but probably slow pipelines, it is not as good as `Quality+History+Cost` in the early stage. `Quality+History+Cost` measures quality, history and cost at the same time, so it achieves a good tradeoff between fast response and good performance. Also, by preferring fast pipelines, we can execute more pipelines and fine tune them, and have a better model of the search space. Eventually, these methods all converge to high-quality solutions, while `Random` is still not as good since the search space is infinite and it is difficult to find a good pipeline without any guidance.

Physical Pipeline Plan Generation We ran our system with and without using Bayesian Optimization for the tuning of hyper-parameters to justify the effectiveness of our *physical pipeline* selection design choices: (1) `Random`, which picks random hyper-parameters configurations; (2) `Bayesian`, which uses Bayesian Optimization to find the next promising configuration of hyper-parameters.

As shown in Figure 9(b), after a very short amount of time (10 seconds) `Bayesian` achieves much better performance. During the first several seconds, `Random` and `Bayesian` are pretty comparable since `Bayesian` essentially does random search at first to learn about the hyperparameter space.

Pipeline Pruning We ran our system with and without using Adaptive Pipeline Selection to evaluate the effectiveness of our pipeline early termination method. We compare two modes: (1) `NoPruning`, which just trains a pipeline on the train dataset and tests it on the validation dataset without pruning anything; (2) Adaptive Pipeline Selection `APS`, which prunes unpromising pipelines.

Using `APS` we are able to to test much more pipelines, obtaining better solution in shorter amount of time as depicted in Figure 9(c). However, as time goes on `NoPruning` performances eventually will converge to `APS` ones: this is due to a diminishing returns effect. Testing more and more pipelines leads to decreasing improvements, since the *physical pipelines* search space has been gradually covered.

9 RELATED WORK

AutoML Systems: Most automated ML systems focus on automated learning algorithm selection and hyper-parameter tuning [4, 6, 13, 23, 24, 36, 45] to make machine learning curation fully automated for non-ML experts.

Arguable most related to our approach is Auto-sklearn for which we explained the differences in depth in Section 7.

Spark TuPAQ [36] and Hyperband [22] use variations of multi-armed bandit (MAB) algorithm to better allocate computational resources for hyper-parameter tuning. However, their search space is limited to hyper-parameter sets for a few (often, user specified) learning algorithms. The output ML pipeline is not practical in that the real-world problems require end-to-end pipeline curation with careful feature engineering/selection and data transformation. One major drawback of MAB-based approach is that the number of arms (a unique configuration/pipeline) explodes with the size of the search space, and the total number of arms can easily exceed the memory size for a full search space with models, hyper-parameters and pre-processors.

Auto-WEKA [17, 40] or its sister package Auto-sklearn [11] solves the problem of learning algorithm selection and their associated hyper-parameter optimization in a combined search space. They also consider various feature selection and data transformation methods to generate end-to-end ML pipelines. Auto-WEKA uses Sequential Model-based Algorithm Configuration (SMAC) to explore the large search space, which is partly discrete and conditional as each selected algorithm has a different set of associated hyper-parameters. The idea is that, instead keeping track of all the possible configurations, the search moves towards a more promising region based on the previous search and evaluation results. Unfortunately, standalone SMAC optimization for the large search space can still run for hours if not days.

In addition, Auto-WEKA and its search space construction is limited to classification and regression problems only.

TPOT [29] is a tree-based pipeline optimization tool using genetic programming while requiring little to no input nor prior knowledge from the user.

Microsoft has recently introduced an AutoML tool via Azure, based on the work of [27]. They build predictive ML pipelines combining collaborative filtering and Bayesian Optimization (BO). In particular they model the *search space* as probability distribution defined by a Probabilistic Matrix Factorization [20] and than use expected improvement as acquisition function to choose the most promising pipelines.

In *Alpine Meadow*, we combine BO with MAB to construct more compact (and dense) search space for Bayesian Optimization, which results in more accurate and efficient search. Additionally, the current implementation can work with existing (WEKA [1] and Scikit-learn [30]) ML libraries as well as custom ML primitives for more complex problems. As a result, *Alpine Meadow* can support more complex problem types (e.g., graph matching, image and audio classification, etc.), and more importantly, *Alpine Meadow* finds a comparable ML pipeline much more efficiently and can progressively improve the quality of the pipeline.

The interactivity aspect differentiates *Alpine Meadow* from other systems: we design time-based cost models preferring fast pipelines early on, incremental training pipelines, and pipeline early termination to provide better interactivity.

Human-In-The-Loop Data Analytics: There are tools and systems that focus on the human-in-the-loop aspect of data science. Hellix aims to accelerate the iterative ML model training with responsive user feedback [42]. Vizdom [10] provides a unique pen-and-touch interface for the user to easily construct ML workflows and interactively refine the analytics/ML pipelines. Most industry cloud ML services, such as TensorFlow [2], Amazon SageMaker and Azure Machine Learning [27], fall into this category, in that they provide fully-managed environment for ML applications. Unlike systems, the focus is not automated end-to-end pipeline curation; the services provide programmable APIs or web-based interface for ML workflow construction and managed computing resources for the deployment. *Alpine Meadow* targets domain experts or users without ML expertise, and instead of requiring the user to construct a working ML workflow with selected algorithms and pre-processors, the system generates one based on the problem description and the data.

Neural Architecture Search: Also related to our work is neural architecture search, in that we consider deep neural networks as one of the learning algorithms. *Alpine Meadow* currently uses transfer learning [41], a general framework for re-using models leaned in one task for other tasks, in order to quickly train a neural architecture model. This limits the search space to a fixed architectures (e.g., the depth and width of hidden layers, skip connections). Neural networks are hard to design from scratch, and there are many proposed solution using similar Bayesian Optimization [5, 25] or Reinforcement Learning techniques [45]. In the future, we will integrate some of the automated neural architecture design techniques for the tasks where deep learning is known to perform best.

10 CONCLUSION AND FUTURE WORK

We have discussed a new approach to Interactive Automated Machine Learning. This low-latency automatic selection, based on Multi-Armed Bandits, Bayesian Optimization and Meta-Learning theory, efficiently explores the pipeline search space and enables domain experts to bring value to the optimization process. We have tested *Alpine Meadow* on datasets with very heterogeneous characteristics, from sample size to feature types. Our experiments show that when compared against state-of-the-art systems or expert-solutions, *Alpine Meadow* generally generates better results in a shorter amount of time. Nevertheless, the current implementation of *Alpine Meadow* leaves some interesting open questions.

First of all, we have found that for many datasets the lack of sufficient information/signal in the data is a major reason for unsatisfactory performances. This issue can take the form of a small training set, inadequate or missing features, or simply an excess of noise. We plan to address those problems by adding external (relevant) information to the dataset, performing what in jargon is called *Data Augmentation.* For example *Alpine Meadow* already boosted the performances of an hand-wrist-size image regression problem using a pre-trained ResNet neural network to extract high-level features from the small train set (just 100 images). Given such encouraging results, we plan to apply *Data Augmentation* to a broader class of tasks under the form of feature extraction, feature addition and sample enlargement . Second, we want to explore new types of strategies for our *logical pipeline* and *physical pipeline* optimizer. We plan to investigate a new scoring model for dataset similarity in order to find relevant datasets with better precision. We also plan to examine more sophisticated early termination techniques by leveraging shared statistics among the pruning threads. Finally, we aim to support Neural Network architecture exploration and compare our system against existing frameworks.

11 ACKNOWLEDGEMENT

This research is funded by the DARPA Award 16-43-D3M-FP-040, NSF Award RI-1813444, NSF Award IIS-1562657 and NSF Award IIS-1514491 and supported by Google, Intel, and Microsoft as part of the MIT Data Systems and AI Lab (DSAIL).

REFERENCES

[1] [n. d.]. Weka. https://www.cs.waikato.ac.nz/ml/weka/. ([n. d.]).

[2] Martín Abadi et al. 2016. Tensorflow: a system for large-scale machine learning.. In *OSDI*, Vol. 16. 265–283.

[3] Marcin Andrychowicz et al. 2016. Learning to learn by gradient descent by gradient descent. In *Advances in Neural Information Processing Systems*. 3981–3989.

[4] James Bergstra and Yoshua Bengio. 2012. Random search for hyper-parameter optimization. *Journal of Machine Learning Research* 13, Feb (2012), 281–305.

[5] James Bergstra, Daniel Yamins, and David Daniel Cox. 2013. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. (2013).

[6] Carsten Binnig, Benedetto Buratti, Yeounoh Chung, Cyrus Cousins, Tim Kraska, Zeyuan Shang, Eli Upfal, Robert Zeleznik, and Emanuel Zgraggen. 2018. Towards interactive curation & automatic tuning of ml pipelines. In *Proceedings of the Second Workshop on Data Management for End-To-End Machine Learning*. ACM, 1.

[7] Pavel Brazdil et al. 2008. *Metalearning: Applications to data mining*. Springer Science & Business Media.

[8] Sébastien Bubeck, Nicolo Cesa-Bianchi, et al. 2012. Regret analysis of stochastic and nonstochastic multi-armed bandit problems. *Foundations and Trends® in Machine Learning* 5, 1 (2012), 1–122.

[9] Moses Charikar et al. 2000. Towards estimation error guarantees for distinct values. In *Proceedings of the nineteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM, 268–279.

[10] Andrew Crotty, Alex Galakatos, Emanuel Zgraggen, Carsten Binnig, and Tim Kraska. 2015. Vizdom: Interactive analytics through pen and touch. *Proceedings of the VLDB Endowment* 8, 12 (2015), 2024–2027.

[11] Matthias Feurer et al. 2015. Efficient and robust automated machine learning. In *Advances in Neural Information Processing Systems*. 2962–2970.

[12] Matthias Feurer et al. 2015. Initializing Bayesian Hyperparameter Optimization via Meta-Learning.. In *AAAI*. 1128–1135.

[13] Daniel Golovin et al. 2017. Google vizier: A service for black-box optimization. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 1487–1495.

[14] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. 2011. Sequential model-based optimization for general algorithm configuration. In *International Conference on Learning and Intelligent Optimization*. Springer, 507–523.

[15] Lilong Jiang, Michael Mandel, and Arnab Nandi. 2013. Gesturequery: A multitouch database query interface. *Proceedings of the VLDB Endowment* 6, 12 (2013), 1342–1345.

[16] Donald R Jones, Matthias Schonlau, and William J Welch. 1998. Efficient global optimization of expensive black-box functions. *Journal of Global optimization* 13, 4 (1998), 455–492.

[17] Lars Kotthoff et al. 2017. Auto-WEKA 2.0: Automatic model selection and hyperparameter optimization in WEKA. *The Journal of Machine Learning Research* 18, 1 (2017), 826–830.

[18] Tim Kraska. 2018. Northstar: an interactive data science system. *Proceedings of the VLDB Endowment* 11, 12 (2018), 2150–2164.

[19] Tim Kraska et al. 2013. MLbase: A Distributed Machine-learning System.. In *Cidr*, Vol. 1. 2–1.

[20] Neil Lawrence and Raquel Urtasun. 2009. Non-linear matrix factorization with Gaussian processes. *Proceedings of the International Conference on Machine Learning* (2009).

[21] Doris Jung-Lin Lee et al. 2017. Accelerating Scientific Data Exploration via Visual Query Systems. *arXiv preprint arXiv:1710.00763* (2017).

[22] Lisha Li et al. 2016. Hyperband: A novel bandit-based approach to hyperparameter optimization. *arXiv preprint arXiv:1603.06560* (2016).

[23] Tian Li et al. 2018. Ease. ml: towards multi-tenant resource sharing for machine learning workloads. *Proceedings of the VLDB Endowment* 11, 5 (2018), 607–620.

[24] Gang Luo. 2016. A review of automatic selection methods for machine learning algorithms and hyper-parameter values. *Network Modeling Analysis in Health Informatics and Bioinformatics* 5, 1 (2016), 18.

[25] Hector Mendoza et al. 2016. Towards automatically-tuned neural networks. In *Workshop on Automatic Machine Learning*. 58–65.

[26] Arnab Nandi. 2013. Querying Without Keyboards.. In *CIDR*.

[27] Melih Huseyn Elibol Nicolo Fusi, Rishit Sheth. 2018. Probabilistic Matrix Factorization for Automated Machine Learning. *arXiv preprint arXiv:1804.05892* (2018).

[28] Milos Nikolic, Mohammed ElSeidy, and Christoph Koch. 2014. LINVIEW: incremental view maintenance for complex analytical queries. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 253–264.

[29] Randal S. Olson et al. 2016. *EvoApplications 2016*. Chapter Automating Biomedical Data Science Through Tree-Based Pipeline Optimization, 123–137. http://dx.doi.org/10.1007/978-3-319-31204-0_9

[30] F. Pedregosa et al. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[31] Matthias Schonlau, William J Welch, and Donald R Jones. 1998. Global versus local search in constrained optimization of computer models. *Lecture Notes-Monograph Series* (1998), 11–25.

[32] P Griffiths Selinger et al. 1979. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*. ACM, 23–34.

[33] Tarique Siddiqui et al. 2016. Effortless data exploration with zenvisage: an expressive and interactive visual analytics system. *Proceedings of the VLDB Endowment* 10, 4 (2016), 457–468.

[34] Alex Slivkins. 2018. Introduction to Multi-armed Bandits. In *Introduction to Multi-armed Bandits*.

[35] Evan R. Sparks et al. 2015. Automating model search for large scale machine learning. In *Proceedings of the Sixth ACM Symposium on Cloud Computing, SoCC 2015, Kohala Coast, Hawaii, USA, August 27-29, 2015*. 368–380. https://doi.org/10.1145/2806777.2806945

[36] Evan R Sparks et al. 2015. Automating model search for large scale machine learning. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*. ACM, 368–380.

[37] Evan R. Sparks et al. 2015. TuPAQ: An Efficient Planner for Large-scale Predictive Analytic Queries. *CoRR* abs/1502.00068 (2015). arXiv:1502.00068 http://arxiv.org/abs/1502.00068

[38] Niranjan Srinivas et al. 2009. Gaussian process optimization in the bandit setting: No regret and experimental design. *arXiv preprint arXiv:0912.3995* (2009).

[39] Richard S Sutton, Andrew G Barto, Francis Bach, et al. 1998. *Reinforcement learning: An introduction*. MIT press.

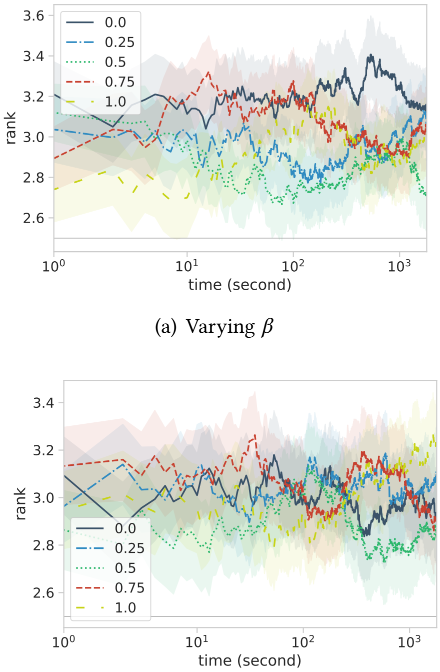
[40] Chris Thornton, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. 2013. Auto-WEKA: Combined selection and hyperparameter optimization of classification algorithms. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 847–855.

[41] Sebastian Thrun and Lorien Pratt. 2012. *Learning to learn*. Springer Science & Business Media.

[42] Doris Xin et al. 2018. Accelerating Human-in-the-loop Machine Learning: Challenges and Opportunities. *arXiv preprint arXiv:1804.05892* (2018).

[43] Emanuel Zgraggen et al. 2015. (s| qu) eries: Visual regular expressions for querying and exploring event sequences. (2015).

[44] Emanuel Zgraggen et al. 2017. How progressive visualizations affect exploratory analysis. *IEEE Transactions on Visualization & Computer Graphics* 8 (2017), 1977–1987.



(a) Varying β

(b) Varying γ

Figure 10: Parameter sensitivity: (a) Rank for different β values (with 90% confidence bands). The higher the β the more exploitation, the lower the more exploration. (b) Rank for different γ values (with 90% confidence bands). The higher γ the more general pipelines are tried, the lower the more data-specific pipelines.

[45] Barret Zoph and Quoc V Le. 2016. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578* (2016).

A APPENDIX

A.1 Parameter Sensitivity

Throughout the paper we used $\beta = 0.5$ and $\gamma = 0.5$ to balance exploration vs exploitation and general pipelines vs data-specific pipelines. In the following, we take a closer look on how these two parameters influence the system performance.

In Figure 10(a) we analyze the impact of β while keeping γ constant at 0.5. As it can be seen, $\beta = 1.0$ achieved the best performance in the beginning (lower means better) as it exploits previous good solutions, while $\beta = 0.0$ was the worst as caused to try pipelines randomly. However, after 10 seconds, $\beta = 0.5$ performs better than $\beta = 1$ as it achieves a good trade-off between exploitation and exploration.

Figure 10(b) shows the sensitivity to γ while keeping β at 0.5. Most interesting is that $\gamma = 1.0$ works extremely well until the very end, while 0.5 performs particular well at the end and reasonable before. The reason is simple: $\gamma = 1.0$ implies that no data-specific pipelines are used, whereas $\gamma = 0.5$ gives data-specific pipelines a chance to develop over time.

In the future, we plan to design strategies that adapt the value of β and γ over time.

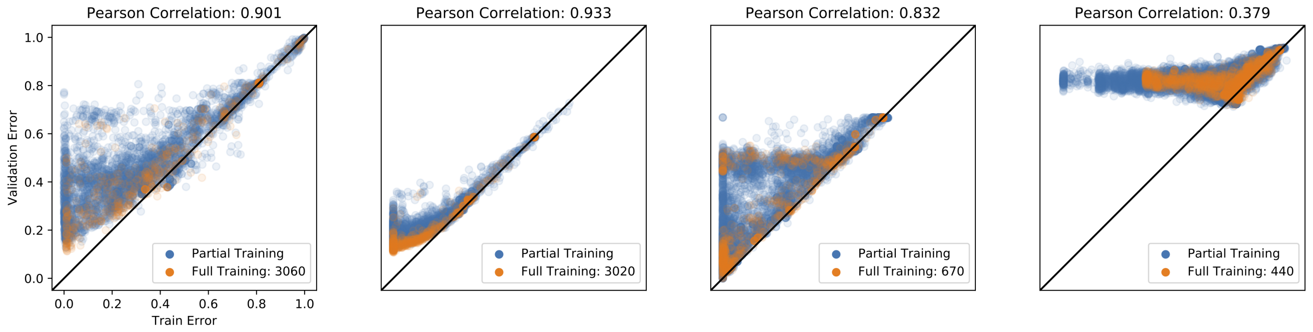
A.2 Halting Criterion

The *Halting Criterion* is motivated by the fact that for a given model the expected validation error is larger than the expected training error, hence it is possible to use the training error as a practical lower bound of the validation error. In Figure 11 we show the correlation plots for four of the datasets that we tested. In the plots each point indicates a pipeline evaluation: blue points are partially trained pipelines; orange points pipelines trained on the full training set. The points above the bisector indicate pipelines for which the lower bound holds (the train error is smaller than the validation error). Figure 11 and table 1 demonstrate that the bound holds for a vast majority of the evaluated pipelines and datasets.

Additionally Figures 11(a), 11(b) and 11(c) show a high correlation between the training and validation error for three tested datasets: this is a general phenomena that we found occurring in the vast majority of the tested datasets, as it is possible to see in correlation column of table 1. A high correlation between those two measures indicates that increasing the training set leads to a more accurate learning of the function $f : \mathcal{X} \rightarrow Y$. While a significantly high correlation between training and validation error occurs in the vast majority of the tested instances, for particularly ill-formed datasets (where the signal in the covariates is not sufficient or well formatted to predict the target variable) this is not always the case. Sub-figure 11(d), displays an instance with low correlation in which the trained models are not able to generate an f that generalizes over the validation set and which incur in overfitting (low train error, high validation error). However it is relevant to note that even in this scenario, where two measure are not correlated, the train error can be still successfully used to lower bound the validation error.

	Datasets	**Correctly Bounded Pipelines (%)**	**Training-Validation Correlation**
Regression	40	75.6	0.867
Classification	110	85.3	0.736
Total	**150**	**83.9**	**0.771**

Table 1: Training vs. Validation error: Each dataset was trained on 400-3000 pipelines (5-95 percentiles). We present the percentage of cases for which validation error was lower bounded by training error and the correlation between training and validation error on those datasets.



(a) Dataset: Page Blocks (b) Dataset: Phenome (c) Dataset: Hill Valley (d) Dataset: Anacalcat Data

Figure 11: Training vs. Validation error on 4 different datasets. Each point represents training and validation error for one pipeline evaluation: blue points are partially trained pipelines; orange points represent pipelines trained on the full training set. Pipelines that are represented above the first quadrant bisector have been correctly bounded by the halting criterion.

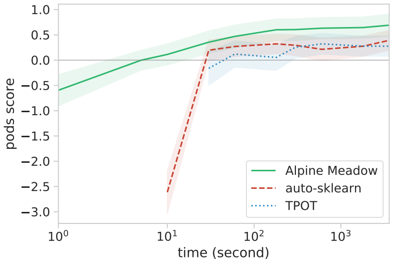


Figure 13: Shows the $pods_{AB}$ scores of each system averaged over all datasets over time, where the $pods_{AB}$ scores are computed as each system over the handmade solutions (higher is better).

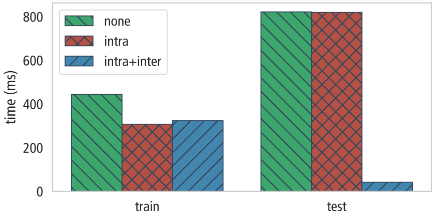


Figure 14: Evaluation of Caching

A.3 Alternative Metric for Comparison

As we outlined previously, the DARPA metric is biased. Therefore, we calculated an alternative metric from [9] for our comparison defined as:

$$pods_{AB} = \begin{cases} \frac{\max(s_A, s_B)}{\min(s_A, s_B)} & s_A \geq s_B \\ -\frac{\max(s_A, s_B)}{\min(s_A, s_B)} & s_A < s_B \end{cases} \tag{3}$$

where s_A and s_B are the F1 score or the negative MSE.

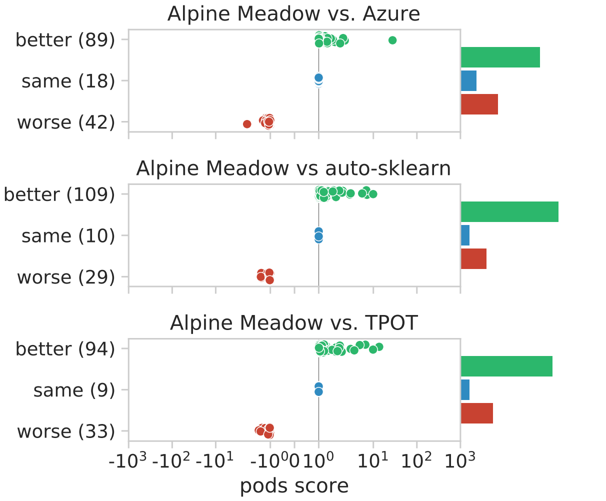


Figure 12: Evaluation of *Alpine Meadow* with different systems. Shows the $pods_{AB}$ scores computed as *Alpine Meadow*'s score over the other system's score, and scores are discretized into "better': *Alpine Meadow* outperforms other system, "same": scores are equal, and "worse": *Alpine Meadow* performs worse than other system.

We replot results from Figure 5 in Figure 12 and results from Figure 6(c) in Figure 13. As we can see, the overall result didn't change much even using the more unbiased metric. *Alpine Meadow* is able to produce results better than baseline scores in just a couple of seconds, and is also significantly better than `auto-sklearn` and `TPOT`.

A.4 Raw Scores on All Datasets

We report the raw scores of all compared systems on all datasets in a GitHub repo https://github.com/vds-automl/alpine_meadow_submission_2019.

A.5 Caching and Incremental Computation

Many pipelines the AutoML tools try share the exact same operation. Therefore, caching can tremendously help to increase the number of pipelines a system can evaluate in a given timeframe. Moreover, *Alpine Meadow* incrementally increases the sample size over time for all promising pipelines. As a result, similar to caching, incremental computation of operations can also significantly increase the number of pipelines the system can test. However, both caching as well as incremental computation is not possible with the standard Scikit-learn operations and without changes to the runtime environment, making it very hard to achieve fair comparison against other AutoML tools. We therefore decided to not evaluate the effect of caching and incremental computation when comparing our system against other baselines.

However, as our system does support both, caching and incremental computation, we want to give at least a high-level overview on how these techniques work in *Alpine Meadow* and how they compare to alternative approaches, such as [28].

Alpine Meadow supports two main types of caching:

Inter-Pipeline Caching: *Alpine Meadow* caches intermediate results of each primitive, such that other pipelines can directly use them without re-computation. For example, if feature-scaling was performed or we use a pre-trained neural net to add additional features, we cache the output such that we don't need to run it again for the same input data. While we only found a moderate impact for operations like feature scaling, it can have a huge impact for neural networks, which we for example use for image classification problems.

Intra-Pipeline Caching Since we train pipelines on increasingly larger samples in Algorithm 4, and smaller samples are always covered by bigger samples, we can utilize this by doing training in an incremental way. Here, we distinguish two cases:

(1) Incremental operator: If we know that an operator is incremental we simple continue the training based on the previous model state if all previous operations are also incremental. Even in cases where the previous operations in the pipeline are not incremental, we might reuse the model state with the assumption that it converges faster from that state. For example, with most gradient descent-based training algorithms this is often the case. However, this requires that the system has access to the internal state of an algorithm. We support this with our own algorithms, but not with the scikit-learn default algorithms.

(2) Non-Incremental operator: For non-incremental operators we observe the state-change over the increasing sample-sizes and based on it decide if we re-use a result. For example, a min-max feature scaler requires to determine the minimum and maximum. If it changes, the entire feature needs to be reprocessed. If not, it can be made incremental. However, after a certain data-size even a change in the minimum or maximum rarely changes the overall of performance of an pipeline. This observation allows us to do "approximate"-caching; re-use results even if the data would slightly change. In our current implementation, we manually tag operations if they are incremental and/or allow for strict or "approximate" caching.

Note, that none of these techniques were used as part of the experiments in Section 8. Also note, that our techniques do re-use on the operator level, which have to be annotated, rather than the more advanced caching/re-use, which is possible on the algebra level [28].

Initial results To evaluate how inter and intra-pipeline caching can improve the performance, we conducted a small experiment with an image classification pipeline. This pipeline uses a pre-trained neural network, i.e., an NN operator, to extract high-level features, which are then used by a traditional classifier. Here, the NN operator allows for intra-pipeline caching as well as inter-pipleine cachine as it is an incremental operator. More precisely, every time the sample is increased, we reuse the feature for data items, which were already included in the previous sample; recall that every increased sample contains all items from the previous smaller sample. Figure 14 shows that for this pipeline intra-caching helps to reduce the training time by up to 33% when increasing the sample size.

Besides expensive primitives or primitives that support incremental training by nature, we also did experiments for encoding operations (e.g., label encoding and one-hot encoding) and non-Incremental operator caching, e.g., scaling functions, and found that the performance improvement with caching is rather small. The reason is, that compared to the actual training of the model, those operations are often relatively fast.

In the future, we plan to further develop our caching and incremental computation techniques. Especially, we believe that there is a lot of potential for non-incremental caching for more complex operations.