AutoML Pipeline Selection: Efficiently Navigating the Combinatorial Space

Chengrun Yang, Jicong Fan, Ziyang Wu, Madeleine Udell
Cornell University
{cy438,jf577,zw287,udell}@cornell.edu

ABSTRACT

Data scientists seeking a good supervised learning model on a dataset have many choices to make: they must preprocess the data, select features, possibly reduce the dimension, select an estimation algorithm, and choose hyperparameters for each of these pipeline components. With new pipeline components comes a combinatorial explosion in the number of choices! In this work, we design a new AutoML system TensorOboe to address this challenge: an automated system to design a supervised learning pipeline. TensorOboe uses low rank tensor decomposition as a surrogate model for efficient pipeline search. We also develop a new greedy experiment design protocol to gather information about a new dataset efficiently. Experiments on large corpora of real-world classification problems demonstrate the effectiveness of our approach.
As of 12/17/2020, this version corrects the errors in the version in the ACM Digital Library.

CCS CONCEPTS

• **Computing methodologies** → **Active learning settings**; **Learning latent representations**; **Principal component analysis**; **Discrete space search**; **Continuous space search**.

KEYWORDS

AutoML; meta-learning; pipeline search; tensor decomposition; submodular optimization; experiment design; greedy algorithms

ACM Reference Format:
Chengrun Yang, Jicong Fan, Ziyang Wu, Madeleine Udell. 2020. AutoML Pipeline Selection: Efficiently Navigating the Combinatorial Space. In *26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '20), August 23–27, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3394486.3403197

1 INTRODUCTION

A machine learning pipeline is a directed graph of learning components including imputation, encoding, standardization, dimensionality reduction, and estimation, that together define a function mapping input data to output predictions. Each component may also include hyperparameters, such as the output dimension of PCA, or the number of trees in a random forest. Simple pipelines may consist of sequences of these components; more complex pipelines may combine inputs to form pipelines with more complex topologies. An example pipeline is shown as Figure 1.

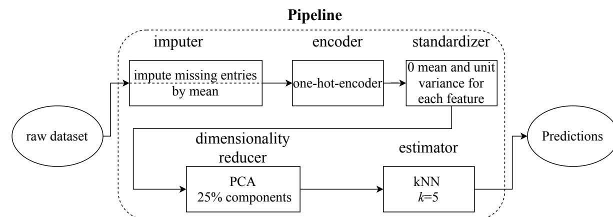


Figure 1: An example pipeline.

The job of a data scientist facing a new supervised learning problem is to choose the pipeline that yields a low out-of-sample error from among all possible pipelines. This task is challenging. First, no component dominates all others: there is "no free lunch" [46]. Rather, each performs well on certain data distributions. For example, the PCA dimensionality reducer works well on data points in $\mathbb{R}^d$ that roughly lie in a low rank subspace $\mathbb{R}^k$ with $k < d$; the feature selector that keeps features with large variances works well on datasets if such features are more informative; the Gaussian naive Bayes classifier works well on features with normally distributed values in each class. However, it is difficult to check these distributional assumptions without running the component on the data: an expensive proposition! The second is the dependence of these choices: for example, standardizing the data may help some estimators, and harm others. Moreover, as the number of possible machine learning components grows, the number of possibilities grows exponentially, defying enumeration. Automating the selection of a pipeline is thus an important problem, which has received attention both from academia and industry [8, 11, 27, 30].

Human experts tackle this difficulty by choosing the right combination according to their domain knowledge. However, finding the right combination takes substantial expertise, and still requires several model fits to find the right combination of components and hyperparameters. An automated pipeline construction system, like a human expert, first forms a *surrogate model* to predict which pipelines are likely to work well. Surrogate models are meta-models that map dataset and machine learning model properties to quantities that characterize performance or informativeness.

A good surrogate model enables efficient search through the pipeline space. "All models are wrong, but some are useful [2]": a good surrogate model makes predictions that guide the search for pipelines without the need for many model fits. Auto-sklearn [11] and Alpine Meadow [36] use meta-learning [1, 25, 39, 41] to choose promising pipelines from those that perform the best on neighboring datasets, and use Bayesian optimization to fine-tune

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
KDD '20, August 23–27, 2020, Virtual Event, USA
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7998-4/20/08…$15.00
https://doi.org/10.1145/3394486.3403197

hyperparameters. TPOT [30] uses genetic programming to search over pipeline topologies. Alpine Meadow [36] uses multi-armed bandit to balance the exploration and exploitation of pipeline structures. In this paper, we use a low multilinear rank tensor as our surrogate model. This model makes explicit use of the combinatorial structure of the problem: as a result, the number of pipeline evaluations required to fit the surrogate model on a new dataset is modest, and independent of the number of pipeline components.

Our system learns the surrogate model for a new dataset by fitting a few pipelines on it. The problem of which pipelines to evaluate first, in order to predict the effectiveness of others, is called the *cold-start problem* in the literature on recommender systems. This problem is also of great interest to the AutoML community. Proximity in meta-features, "simple, statistical or landmarking metrics to characterize datasets [47]", are used by many AutoML systems [11, 12, 14, 33, 36] to select models that work well on neighboring datasets, with the belief that models perform similarly on datasets with similar characteristics. Probabilistic matrix factorization has been used to extract dataset latent representations from pipeline performance [14]. Other dataset and pipeline embeddings have also been proposed that use pipeline performance or even textual dataset or algorithm descriptions to build surrogate models [9, 44, 47].

In this work, we build pipeline embeddings by fitting a tensor decomposition to the (incompletely observed) tensor of pipeline performance on a set of training datasets. The tensor model is easy to extend to a new dataset by fitting a constant number of pipelines on it. We describe a simple rule to select which pipelines to observe by solving a constrained version of the classical experiment design [3, 22, 34, 43] problem using a greedy heuristic [28].

We consider the following concrete challenge in this paper: select several pipelines that perform the best within a given time limit for a new dataset, in the case that we already know or have time to collect pipeline performance on some existing datasets. We focus on small data and traditional supervised machine learning pipelines in our experiments, although the methodology can be generalized to a wider range of disciplines. Our main technical contributions are: a new tensor model to exploit the combinatorial pipeline performance structure, and a new pipeline search mechanism that builds on ideas from greedy experiment design. Together, these ideas yield a new state-of-the-art system for AutoML pipeline selection. Since Oboe [47] is an AutoML system that selects machine learning models by matrix factorization, we name our system in this paper TensorOboe: the AutoML system that uses tensor decomposition to select pipelines.

This paper is organized as follows. Section 2 introduces notation and terminology. Section 3 describes the main ideas used efficiently search the pipeline space. Section 3.1 gives details on TensorOboe. Section 4 shows experimental results.

2 NOTATION AND TERMINOLOGY

Meta-learning. Meta-learning, also called "learning to learn", uses results from past tasks to make predictions or decisions on a new task. In our setting, we learn from a corpus of datasets called *meta-training* datasets by fitting pipelines to these datasets in an offline stage; the new dataset, which requires a fast recommendation for a pipeline, is called the *meta-test* dataset.

Model. A *model* $\mathcal{A}$ is a specific combination of algorithm and hyperparameter settings, e.g. k-nearest neighbors with $k = 3$.

Pipeline component. A pipeline component is a model or model type. Examples include missing entry imputers, dimensionality reducers, supervised learners, and data visualizers. We consider the following components in this paper:

- *Data imputer*: A preprocessor that fills in missing entries.
- *Encoder*: A transformer that converts categorical features to numerical codes. Here, we consider encoding categoricals as integers or with a one-hot encoder.
- *Standardizer*: A standardizer centers and rescales data.
- *Dimensionality reducer*: A transformer that reduces the dimensionality of the dataset by either creating new features (like PCA) or subsampling features.
- *Estimator*: The supervised learner. For the classification tasks in this paper, estimators are classifiers.

Linear algebra. Our paper follows the notation of [47] and [24]. We define $[n] = \{1, \dots, n\}$ for $n \in \mathbb{Z}$, and denote *vector*, *matrix*, and *tensor* variables respectively by lowercase letters (x), capital letters (X) and Euler script letters ($\mathcal{X}$). The order of a tensor is the number of dimensions; matrices are order-two tensors. Each dimension is called a mode. Throughout this paper, all vectors are column vectors. To denote a part of matrix or tensor, we use a colon to denote the dimension that is not fixed: given a matrix $A \in \mathbb{R}^{m \times n}$, $A_{i,:}$ and $A_{:,j}$ (or a_j) denote the ith row and jth column of A, respectively. A fiber is a one-dimensional section of a tensor $\mathcal{X}$, defined by fixing every index but one; for example, one fiber of the order-3 tensor $\mathcal{X}$ is $X_{:jk}$. Fibers of a tensor are analogous to rows and columns of a matrix. A slice is an $(N-1)$-dimensional section of an order-N tensor $\mathcal{X}$. The mode-n matricization of $\mathcal{X}$, denoted as $\mathcal{X}^{(n)}$, is a matrix whose columns are the mode-n fibers of $\mathcal{X}$. $\mathcal{X}$ has *multilinear rank* $(r_1, r_2, \dots)$ if r_n is the rank of $\mathcal{X}^{(n)}$. For example, given an order-3 tensor $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$, we have $X^{(1)} \in \mathbb{R}^{I \times (J \times K)}$, and $\mathcal{X}$ has multilinear rank (r_1, r_2, r_3) if $\mathcal{X}^{(n)}$ has rank r_n for $n \in [3]$. We denote the *n-mode product* of a tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots I_N}$ with a matrix $U \in \mathbb{R}^{J \times I_n}$ by $\mathcal{X} \times_n U \in \mathbb{R}^{I_1 \times \cdots I_{n-1} \times J \times I_{n+1} \times \cdots I_N}$; the $(i_1, i_2, \dots, i_{n-1}, j, i_{n+1}, \dots, i_N)$-th entry is $\Sigma_{i_n=1}^{I_n} x_{i_1 i_2 \cdots i_{n-1} i_n i_{n+1} \cdots i_N} u_{j i_n}$. Given two tensors with the same shape, we use $\odot$ to denote their entrywise product. Given an ordered set $\mathcal{S} = \{s_1, \dots, s_k\}$ where $s_1 < \dots < s_k \in [n]$, we write $A_{:\mathcal{S}} = [A_{:,s_1}, A_{:,s_2}, \dots, A_{:,s_k}]$; given an ordinary set S, we use $A_{:S}$ to denote $A_{:\mathcal{S}}$, in which $\mathcal{S}$ is the ordered version of set S.

Pipeline performance. The performance of a machine learning pipeline is usually characterized by cross-validation error. Given a dataset $\mathcal{D}$ and a pipeline $\mathcal{P}$, we denote the error of $\mathcal{P}$ on $\mathcal{D}$ as $\mathcal{P}(\mathcal{D})$. It is common practice to evaluate this error by cross-validating $\mathcal{P}$ on $\mathcal{D}$ with a certain number of folds (often 3, 5 or 10) and a fixed dataset partition. We use $\mathcal{P}(\mathcal{D})$ to denote the cross-validation error we observe with a certain number of folds and a certain partition.

Error tensor and error matrix. Pipeline errors on training datasets form an *error tensor*, which we denote as $\mathcal{E}$. In our experiments, $\mathcal{E}$ is an order-6 tensor, with 6 modes corresponding to datasets, imputers, encoders, standardizers, dimensionality reducers and estimators, respectively. The $(i_1, i_2, \dots, i_6)$-th entry of $\mathcal{E}$ is the error of the pipeline formed by composing the i_2-th imputer, i_3-th encoder, i_4-th standardizer, i_5-th dimensionality reducer, and i_6-th estimator and evaluating this pipeline on the i_1-th dataset. If a

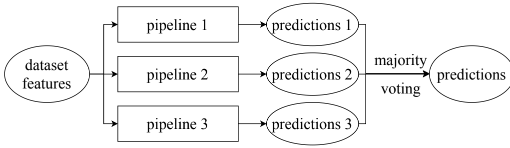


Figure 2: A pipeline ensemble with 3 base learners.

pipeline-dataset combination has been evaluated, we say the corresponding entry in the error tensor $\mathcal{E}$ is observed. The first unfolding of the error tensor, $\mathcal{E}^{(1)}$, is called the *error matrix E*, whose (i, j)-th entry $E_{ij} = \mathcal{P}_j(\mathcal{D}_i)$ is the error of pipeline j on dataset i.

Ensemble. An ensemble [4, 7, 35, 45] combines a finite set of individual machine learning models into a single prediction model. For simplicity, the combination method we use is majority voting for classification. We define the *candidate learner* to be individual machine learning pipelines that we select from to create the ensemble, and *base learner* to be pipelines that are included in the ensemble. An ensemble of pipelines is itself a pipeline, but not a simple linear pipeline. By creating ensembles of linear pipelines, TensorOboe can perform better than any linear pipeline.

3 METHODOLOGY

3.1 Overview

TensorOboe has two phases. In the offline phase, we compute the performance of pipelines on meta-training datasets to build a tensor surrogate model. In the online phase, we run a small number of pipelines on the new meta-test dataset to specialize the surrogate model and identify promising pipelines.

Offline Stage. We collect a partially observed error tensor using the approach described in Section 3.2 to limit the total runtime of the offline phase. We complete and decompose the error tensor $\mathcal{E}$ using the EM-Tucker algorithm, shown as Algorithm 1, with dataset and estimator ranks empirically chosen to be the ones that give low reconstruction error, described in Section 4.2.

Online Stage. Online, given a new dataset $\mathcal{D}$ with $n^{\mathcal{D}}$ data points and $p^{\mathcal{D}}$ features, we first predict the running time of each pipeline by a simple model: order-3 polynomial regression on $n^{\mathcal{D}}$ and $p^{\mathcal{D}}$ and their logarithms. This simple model works well because the time to fit the estimator dominates the time to fit the pipeline, and the theoretical complexities of estimators we use have no higher order terms [21, 47].

The initial dataset and estimator ranks are set to the number of principal components that capture 97% of the energy in the respective tensor matricizations. We double the runtime budget at each iteration and increment the estimator rank if the performance improves. In each iteration, we build ensembles whose base learners are the 5 pipelines with the best cross-validation error. An ensemble can improve on the performance of the best base pipeline. An example is shown as Figure 2.

3.2 Tensor Collection for Meta-Training

In the meta-training phase of meta-learning, meta-training data is generally assumed to be already available or cheap to collect. Given the large number of possible pipeline combinations, though, collecting meta-training data can be prohibitively expensive. As an example, even if it takes one minute on average to evaluate each pipeline on each dataset, evaluating 20,000 pipelines on 200 meta-training datasets would take more than 7 years of CPU time. This motivates us to use tensor completion to limit the time spent on the collection of meta-training data, while preserving accuracy of our surrogate model.

We collect pipeline performance in a biased way: using 3-fold cross-validation, we only evaluate pipelines that complete within 120 seconds. This rule gives a missing ratio of 3.3%. Notice that the entries are not missing uniformly at random: for example, some datasets are large and expensive to evaluate; our training data systematically lacks data from these large datasets. Nevertheless, we will show how to infer these entries using tensor completion in Section 3.3, and demonstrate in Section 4.2 that the method performs well despite bias.

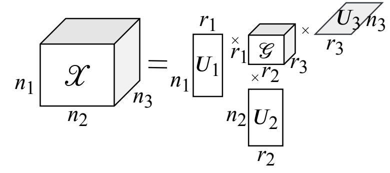


Figure 3: Tucker decomposition on an order-3 tensor.

3.3 Tensor Decomposition and Rank

The meta-training phase constructs the error tensor $\mathcal{E}$. In the meta-test phase, we see a new dataset, corresponding to a new slice of $\mathcal{E}$. To learn about the slice efficiently, we use a low rank tensor decomposition to predict all the entries in this slice from a subset of its informative entries.

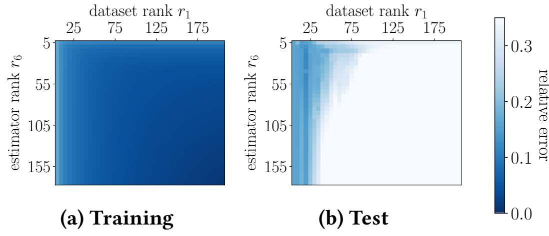
Unlike matrices, there are many incompatible notions of tensor ranks and low rank tensor decompositions, including CANDECOMP/PARAFAC (CP) [5, 19], Tucker [40], and tensor-train [31]. Each emphasizes a different aspect of the tensor low rank property. In this paper, we use Tucker decomposition; an illustration on an order-3 tensor is shown as Figure 3. As a form of higher-order PCA, Tucker decomposes a tensor into the product of a *core tensor* and several *factor matrices*, one for each mode [24]. A tensor with low multilinear rank has a low rank Tucker decomposition. In our setting of order-6 tensors, Tucker decomposition of $\mathcal{E}$ is

$$\mathcal{E} \approx \hat{\mathcal{E}} = \mathcal{G} \times_1 U_1 \times \cdots \times_6 U_6, \quad (1)$$

with core tensor $\mathcal{G} \in \mathbb{R}^{r_1 \times r_2 \times \cdots \times r_6}$ and column-orthonormal factor matrices $U_i \in \mathbb{R}^{n_i \times r_i}$, $i \in \{1, 2, \ldots, 6\}$. $\hat{\mathcal{E}}$ is linear in the factor matrices. Each factor matrix can thus be viewed as embedding the corresponding dataset or pipeline component, with pipeline embeddings as columns of $Y = (\mathcal{G} \times_2 U_2 \times \cdots \times_6 U_6)^{(1)} \in \mathbb{R}^{r_1 \times (\Pi_{i=2}^6 n_i)}$, the mode-1 matricization of the product. We can use this observation to approximately factor the error matrix E, using Equation 1, as

$$X^\top Y \approx E \in \mathbb{R}^{n_1 \times (\Pi_{i=2}^6 n_i)}, \quad (2)$$

in which $X \in \mathbb{R}^{r_1 \times n_1}$ and $Y \in \mathbb{R}^{r_1 \times (\Pi_{i=2}^6 n_i)}$ are dataset and pipeline embeddings, respectively.



(a) Training **(b) Test**

Figure 4: Relative error heatmaps when varying ranks in dataset and estimator dimensions. Here, training entries are the ones with runtime less than 90 seconds; the test entries are the ones with runtime between 90 and 120 seconds.

Figure 4 shows the low rank Tucker decomposition fits the error tensor well.

3.4 Tensor Completion

To infer missing entries in the error tensor we collected, namely the entries that take more than the time threshold to evaluate, we use the expectation-maximization (EM) [6, 38] approach together with Tucker decomposition in each step, which we call EM-Tucker and present as Algorithm 1.

Algorithm 1 EM-Tucker algorithm for tensor completion

Input: order-n error tensor $\mathcal{E}$ with missing entries, target multilinear rank $[r_1, \ldots, r_n]$
Output: imputed error tensor $\mathcal{E}$
1 $\mathcal{E}_{\text{obs}} \leftarrow \mathcal{E}$
2 $\Omega \leftarrow$ observed entries in $\mathcal{E}_{\text{obs}}$
3 **do**
4 $\quad \mathcal{G}, \{U_i\}_{i=1}^n \leftarrow$ Tucker($\mathcal{E}$, ranks=$[r_1, \ldots, r_n]$)
5 $\quad \mathcal{E}_{\text{pred}} \leftarrow \mathcal{G} \times_1 U_1 \times \cdots \times_n U_n$
6 $\quad \mathcal{E} \leftarrow \Omega \odot \mathcal{E}_{\text{obs}} + (1 - \Omega) \odot \mathcal{E}_{\text{pred}}$
7 **while** not converged

In Algorithm 1, Ω is a binary tensor that denotes whether each entry of the error tensor $\mathcal{E}$ is observed or not. Ω has the same shape as the original error tensor, with the corresponding entry $\Omega_{i_1, i_2, \ldots, i_n} = 1$ if the $(i_1, i_2, \ldots, i_n)$-th entry of the error tensor is observed, and 0 otherwise. The algorithm is regarded to have converged when the decrease of relative error is less than 0.1%.

Why bother with tensor completion? To recover the missing entries of a tensor, we can also perform matrix completion after matricization or perform matrix completion on every slice separately. Tensors are more constrained and so provide better fits to sparse and noisy data. Consider a tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots I_N}$ with multilinear rank $[r_1, r_2, \ldots, r_n]$, where $I_1 = I_2 = \ldots = I_n = I$ and $r_1 = r_2 = \ldots = r_n = r$. The number of degrees of freedom of $\mathcal{X}$, which is the minimum number of entries required to recover $\mathcal{X}$, is $r^n + n(rI - r^2) =: m_0$. If we unfold $\mathcal{X}$ to $X \in \mathbb{R}^{I \times I^{n-1}}$, the number of degrees of freedom of X is $(I + I^{n-1} - r)r =: m_1$. If we treat every slice of $\mathcal{X}$ separately, the number of degrees of freedom is $I^{n-2}(2rI - r^2) =: m_2$. Therefore, when $r < I$, we have $m_0 < m_1 < m_2$, which means we need fewer parameters to determine $\mathcal{X}$, compared to the matricization and union of slices. Thus, tensor completion may outperform matrix completion on $\mathcal{X}$ with the same number of observed entries.

3.5 Fast and Accurate Resource-Constrained Active Learning

Given a new dataset, we first select a subset of pipelines to fit, so that we may estimate the performance of other pipelines. We use ideas from linear experiment design, which picks a subset of low-cost statistical trials to minimize the variance of the resulting estimator, to make this selection.

Concretely, we estimate the embedding x of the new dataset by linear regression. Given the linear model as Equation 2, with known performance e_S of a subset $S \subseteq [n]$ of pipelines on the new dataset, we have

$$e_S = (Y_{:S})^\top x + \epsilon, \tag{3}$$

in which Y collects the latent embeddings of pipeline performance, and ϵ is the error in this linear model. An example of the source of error is the misspecification of target multilinear rank for the Tucker decomposition. We estimate x by linear regression and denote the result as $\hat{x}$. Then we estimate the performance of pipelines in $[n] \backslash S$ by the corresponding entries in $\hat{e} = Y^\top \hat{x}$.

Now we consider which S to choose to accurately estimate x. We will motivate the use of the experiment design model and its greedy approach by first showing how to constrain the *number* of pipelines sampled in Section 3.5.1, and then develop a *time*-constrained version that we use in practice in Section 3.5.2.

3.5.1 Greedy method for size-constrained experiment design. Suppose the error $\epsilon \sim \mathcal{N}(0, \sigma^2 I)$. Using the linear regression model, Equation 3, we want to minimize the expected ℓ_2 error $E_\epsilon \|\hat{x} - x\|^2 = E_\epsilon \|\hat{x} - E_\epsilon \hat{x}\|^2 + \|E_\epsilon \hat{x} - x\|^2$. Here, the second term is 0 since linear regression is unbiased, and the first term is the covariance $\sigma^2 (YY^\top)^{-1}$ of the estimated embedding $\hat{x}$, which is straightforward to compute.

Imagine we have enough time to run at most m pipelines (and all pipelines run equally slowly). Given pipeline embeddings $\{y_j\}_{j=1}^n$ (which we call *design vectors* or *designs*), in which each $y_j \in \mathbb{R}^k$, we minimize a scalarization of the covariance to obtain the (number-constrained) D-optimal experiment design problem

$$\begin{array}{ll} \text{maximize} & \log\det\left(\sum_{j\in S} y_j y_j^\top\right) \\ \text{subject to} & |S| \le m \\ & S \subseteq [n]. \end{array} \tag{4}$$

Here, $\sum_{j\in S} y_j y_j^\top$, the inverse of (scaled) covariance matrix, is called the Fisher information matrix.

Obtaining an exact solution for a mixed-integer nonlinear combinatorial optimization problem like Problem 4 is prohibitively expensive. Convexification is commonly used to solve such a problem [3, 34, 47]. However, we have more than 20,000 pipelines to select from, making convex relaxations also too slow. Moreover, we can find better solutions with the greedy heuristic we present next.

Greedy methods form another popular approach to combinatorial optimization problems like Problem 4. Importantly, the objective function of Problem 4, $f(S) = \log\det\left(\sum_{j\in S} y_j y_j^\top\right)$, is submodular. (Recall a set function $g : 2^V \to \mathbb{R}$ defined on a subset of V is submodular if for every $A \subseteq B \subseteq V$ and every element $s \in V \backslash B$,

we have $g(A \cup \{s\}) - g(A) \geq g(B \cup \{s\}) - g(B)$. This characterizes a "diminishing return" property.) Given a size constraint, the submodular function maximization problem

$$\begin{array}{ll} \text{maximize} & g(S) \\ \text{subject to} & S \subseteq V \\ & |S| \leq m \end{array} \tag{5}$$

can be solved with a $1 - \frac{1}{e}$ approximation ratio [29] by the greedy approach: in every step, add the single design vector that maximizes the increase in function value. In D-optimal experiment design, we can compute this increase efficiently using Lemma 3.1.

Lemma 3.1 (Matrix Determinant Lemma [20, 28]). *For any invertible matrix $A \in \mathbb{R}^{k\times k}$ and $a, b \in \mathbb{R}^k$,*

$$\det(A + ab^\top) = \det(A)(1 + b^\top A^{-1} a).$$

At the t-th step in our setting, with an already constructed Fisher information matrix $X_t = \sum_{j\in S} y_j y_j^\top$, we have

$$\text{argmax}_{j\in[n]\backslash S} \det(X_t + y_j y_j^\top) = \text{argmax}_{j\in[n]\backslash S} y_j^\top X_t^{-1} y_j.$$

Here, $y_j^\top X_t^{-1} y_j$ can be seen as the payoff for adding pipeline j. From the t-th to the $(t + 1)$-th step, with the selected design vector at the t-th step as y_t, we update X_t to $X_{t+1} = X_t + y_t y_t^\top$ by Lemma 3.2:

Lemma 3.2 (Sherman-Morrison formula [18, 37]). *For any invertible matrix $A \in \mathbb{R}^{k\times k}$ and $a, b \in \mathbb{R}^k$,*

$$(A + ab^\top)^{-1} = A^{-1} - \frac{A^{-1}ab^\top A^{-1}}{1 + b^\top A^{-1}a}.$$

Pseudocode for the greedy algorithm to solve Problem 4 is shown as Algorithm 2, with per-iteration time complexity $O(k^3 + nk^2)$: it takes $O(k^3)$ (for a naive matrix multiplication algorithm) to update X_t^{-1} and $O(nk^2)$ to choose the best pipeline to add.

Algorithm 2 Greedy algorithm for size-constrained D-design

Input: design vectors $\{y_j\}_{j=1}^n$, in which $y_j \in \mathbb{R}^k$; maximum number of selected pipelines m; initial set of designs $S_0 \subseteq [n]$, s.t. $X_0 = \sum_{j\in S_0} y_j y_j^\top$ is non-singular
Output: The selected set of designs $S \subseteq [n]$

1 **function** Greedy_ED_Number
2 $\quad S \leftarrow S_0$
3 $\quad$ **do**
4 $\quad\quad i \leftarrow \text{argmax}_{j\in[n]\backslash S} y_j^\top X_t^{-1} y_j$
5 $\quad\quad S \leftarrow S \cup \{i\}$
6 $\quad\quad X_{t+1} \leftarrow X_t + y_i y_i^\top$
7 $\quad$ **while** $|S| \leq m$
8 $\quad$ **return** S

There remains the problem of how to select an initial set of designs S to start from, such that $X_0 = \sum_{j\in S} y_j y_j^\top = Y_{:S} Y_{:S}^\top$ is non-singular. This is equivalent to the problem of finding a subset of vectors in $\{y_j\}_{j=1}^n$ that can span $\mathbb{R}^k$. We select this sized-k subset S_0 to be the first k pivot columns from QR factorization with column pivoting [16, 17] on Y, with time complexity $O((n + k)k^2)$.

3.5.2 *Greedy method for time-constrained experiment design.*

We here move on to the realistic case in AutoML pipeline selection: which pipelines should we select to gain an accurate estimate of the entire pipeline space? In this setting, each pipeline is associated with a different cost. We characterize the cost as running time, and form the time-constrained version of experiment design as

$$\begin{array}{ll} \text{maximize} & \log\det\left(\sum_{j\in S} y_j y_j^\top\right) \\ \text{subject to} & \sum_{j\in S} \hat{t}_j \leq \tau \\ & S \subseteq [n], \end{array} \tag{6}$$

in which $\{\hat{t}_i\}_{i=1}^n$ are the estimated pipeline running times. The payoff of adding design i in the t-th step can thus be formulated as $\frac{y_i^\top X_t^{-1} y_i}{\hat{t}_i}$, giving Algorithm 3 the greedy method to solve Problem 6.

Algorithm 3 Greedy algorithm for time-constrained D-design

Input: design vectors $\{y_j\}_{j=1}^n$, in which $y_j \in \mathbb{R}^k$; estimated running time of pipelines $\{\hat{t}_i\}_{i=1}^n$; maximum running time τ; initial set of designs $S_0 \subseteq [n]$, s.t. $X_0 = \sum_{j\in S_0} y_j y_j^\top$ is non-singular
Output: The selected set of designs $S \subseteq [n]$

1 **function** Greedy_ED_Time
2 $\quad S \leftarrow S_0$
3 $\quad$ **do**
4 $\quad\quad i \leftarrow \text{argmax}_{j\in[n]\backslash S} \frac{y_j^\top X_t^{-1} y_j}{\hat{t}_j}$
5 $\quad\quad S \leftarrow S \cup \{i\}$
6 $\quad\quad X_{t+1} \leftarrow X_t + y_i y_i^\top$
7 $\quad$ **while** $\sum_{i\in S} \hat{t}_i \leq \tau$
8 $\quad$ **return** S

The initialization problem is solved similarly by the QR method. Given runtime limit τ, we select among columns with corresponding pipelines predicted to finish within $\frac{\tau}{2k}$. Pseudocode for this initialization algorithm is shown as Algorithm 4.

Algorithm 4 Initialization of the greedy algorithm for time-constrained D-design, by QR factorization with column pivoting

Input: design vectors $\{y_j\}_{j=1}^n$, in which $y_j \in \mathbb{R}^k$; (predicted) running time of all pipelines $\{\hat{t}_i\}_{i=1}^n$; maximum running time τ
Output: A subset of designs $S_0 \subseteq [n]$ for Algorithm 3 initialization

1 **function** QR_Initialization
2 $\quad S_{\text{valid}} \leftarrow \{i \in [n] : \hat{t}_i \leq \frac{\tau}{2k}\}$
3 $\quad S_0 \leftarrow \emptyset, \hat{t}_{\text{sum}} \leftarrow 0$
4 $\quad$ **if** $|S_{\text{valid}}| < k$ **then** ▷ Case 1
5 $\quad\quad$ **do**
6 $\quad\quad\quad i \leftarrow \text{argmin}_{j\in[n]\backslash S} \hat{t}_j$
7 $\quad\quad\quad S_0 \leftarrow S_0 \cup \{i\}$
8 $\quad\quad\quad \hat{t}_{\text{sum}} \leftarrow \hat{t}_{\text{sum}} + \hat{t}_i$
9 $\quad\quad$ **while** $\hat{t}_{\text{sum}} \leq \tau$
10 $\quad$ **else** ▷ Case 2
11 $\quad\quad S_0 \leftarrow \text{QR_with_column_pivoting}(Y_{:S_{\text{valid}}})[: k]$
12 $\quad$ **return** S_0

A corner case of Algorithm 4, shown as Case 1, is that there are not enough pipelines predicted to be able to finish within time limit.

This corresponds to the case that the runtime limit is relatively small compared to the time of fitting pipelines on current dataset. In this case we greedily select the fast pipelines and do not run Algorithm 3 afterwards.

As a side note, the assumption that performance of different pipelines are predicted with equal variance is not quite realistic, especially when some components have much more pipelines than others. If the variance is known (but unequal), we obtain a weighted least squares problem. In the error matrix E, we can estimate the variance of prediction error of each pipeline $j \in [n]$ by the sample variance of $e_j - X^\top y_j$ and select the promising pipelines with the goal of minimizing the rescaled covariance. Practically, however, this rescaled method does not systematically improve on the standard least squares approach in our experiments (shown in Appendix B), so we retrench to the simpler approach.

4 EXPERIMENTAL EVALUATIONS

Code for all experiments is in the GitHub repository at https://github.com/udellgroup/oboe. We use a Linux machine with 128 Intel® Xeon® E7-4850 v4 2.10GHz CPU cores and 1056GB memory. Offline, we collect cross-validated pipeline performance on meta-training datasets: 215 OpenML [13, 42] classification datasets with number of data points between 150 and 10,000, listed in Appendix A.1. The 215 datasets are chosen alphabetically. Pipelines are combinations of the machine learning components shown in Appendix A.3, Table 2, which lists 4 data imputers, 2 encoders, 2 standardizers, 8 dimensionality reducers and 183 estimators, resulting in 23,424 linear pipeline candidates in total.

4.1 Comparison with Time-Constrained AutoML Pipeline Build Systems

In this section, we demonstrate the performance of TensorOboe as an AutoML system for pipeline selection.

A naive approach for pipeline selection is to choose the one that on average performs the best among all meta-training datasets, which we call the baseline pipeline. Given the pipeline selection problem, it is common for human practitioners to try out the best pipeline at the very beginning. On our meta-training datasets, the baseline pipeline is: impute missing entries with the mode, encode categorical features as integers, standardize each feature, remove features with 0 variance, and classify by gradient boosting with learning rate 0.25 and maximum depth 3. The baseline pipeline has an average ranking of 1568 among all 23,424 pipelines across all 215 meta-training datasets.

Human practitioners may also reduce the number of trials by choosing certain pipeline components to be the type that performs the best on average. Figure 5, however, shows that although some estimator types (gradient boosting and multilayer perceptron) are commonly seen among the best pipelines, no estimator type uniformly dominates the rest.

We compare TensorOboe with auto-sklearn [11], TPOT [30], and the baseline pipeline in Figure 6. To ensure fair comparisons, we use a single CPU core for each AutoML system. We allow each to choose from the same primitives. We can see that:

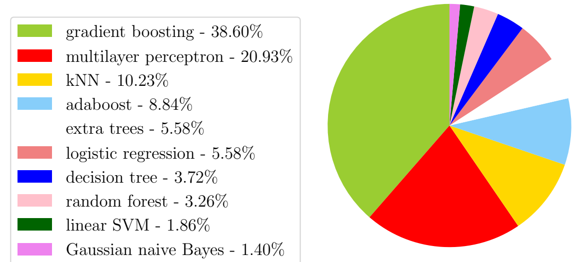
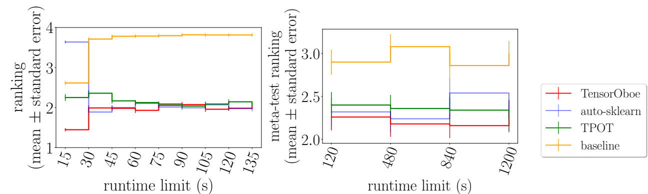


Figure 5: Which estimators work best? Distribution of estimator types in best pipelines on meta-training datasets.



(a) OpenML (meta-LOOCV) **(b) UCI (meta-test)**

Figure 6: Rankings of AutoML systems for pipeline search in a time-constrained setting, vs the baseline pipeline. We meta-train on OpenML classification datasets and meta-test on UCI classification datasets [10]. Until the first time the systems can produce a pipeline, we classify every data point with the most common class label. Lower ranks are better.

1 All AutoML frameworks are able to construct pipelines that outperform the baseline on average once the method returns a pipeline (for auto-sklearn, this takes 30 seconds).
2 TensorOboe on average outperforms the competing methods and produces meaningful pipeline configurations fastest.
3 With the longer running time in Figure 6b, TensorOboe still outperforms in most cases.

These results show that TensorOboe is able to accurately approximate the hyperparameter landscape. We discuss these results in greater detail in Section 4.5.

4.2 Tensor Completion vs Matrix Completion for Error Tensor Completion

Given meta-training data $\{\mathcal{D}, \mathcal{P}, \mathcal{P}(\mathcal{D})\}$ on a subset of dataset-pipeline combinations, a good surrogate model should accurately predict the performance of new dataset-pipeline combinations.

Figure 7 shows that most pipelines run quickly on most datasets: for example, over 90% finish in less than 20 seconds and over 95% finish in less than 80 seconds.

Figure 8 compares relative errors of predictions by tensor and matrix surrogate models. For each runtime threshold, we treat pipeline-dataset combinations with running time less than the threshold as training data, and those that take longer than threshold and less than 120 seconds as test. We compute relative errors on test data, hence the name "runtime generalization". To ensure a fair comparison, we set the dataset and estimator ranks to be equal in

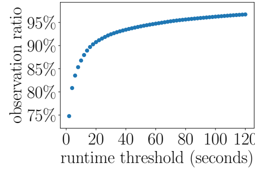
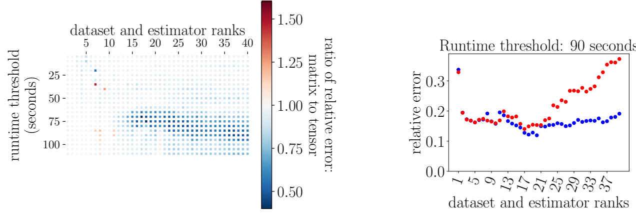
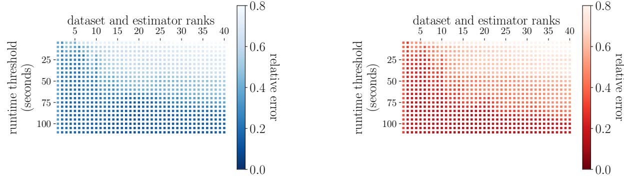


Figure 7: CDF of pipeline runtime on meta-training datasets.



(a) Runtime generalization for tensor vs matrix models. Blue means the tensor model achieves lower error; red means the matrix model does. The tensor model outperforms for longer running times.

(b) Tensor and matrix completion errors when runtime threshold = 90 seconds (3.3% of entries in error tensor missing).



(c) Runtime generalization error by tensor model. Darker colors mean smaller errors.

(d) Runtime generalization error by matrix model. Darker colors mean smaller errors.

Figure 8: Tensor completion vs matrix completion for inferring pipeline performance.

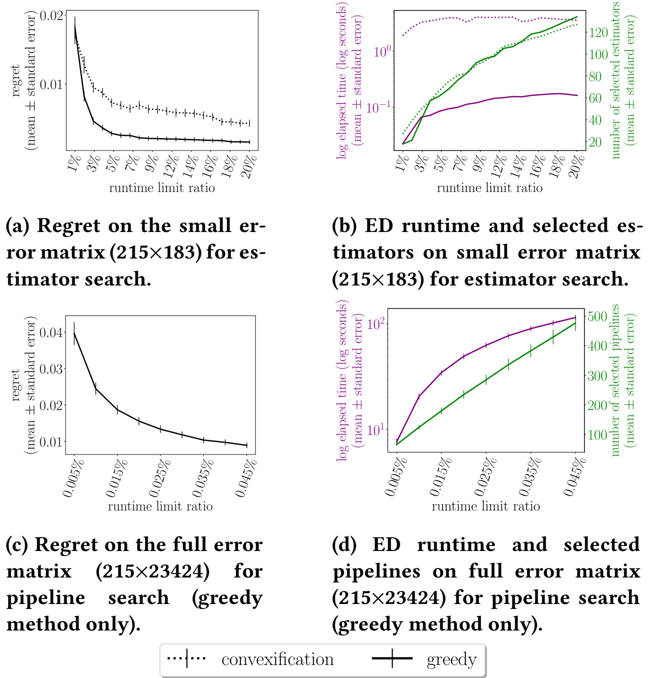
the tensor model, which is required for the matrix model, since column rank equals row rank for a matrix. We can see that:

1 The tensor model outperforms the matrix model in nearly all cases, demonstrating that the additional combinatorial structure provided by the tensor model helps recover the combinatorial relationships among different pipeline components.

2 Figure 8b shows the U-shaped error curve as we increase the dataset and estimator ranks for both matrix and tensor models, moving from underfitting (decreasing error) to overfitting (increasing error). Informed by these results, we select both ranks to be 20, the rank in the middle, in the tensor surrogate model.

4.3 Cold-Start Performance by Greedy Experiment Design

We compare the performance of different approaches to solve the experiment design problem, so as to choose which pipelines we should sample. Recall that there are two approaches:



(a) Regret on the small error matrix (215×183) for estimator search.

(b) ED runtime and selected estimators on small error matrix (215×183) for estimator search.

(c) Regret on the full error matrix (215×23424) for pipeline search (greedy method only).

(d) ED runtime and selected pipelines on full error matrix (215×23424) for pipeline search (greedy method only).

Figure 9: Comparison of time-constrained experiment design methods across meta-training datasets. The y-axes in 9a and 9c are regrets: the difference between minimum pipeline error found by each method and the true minimum. The x-axes are runtime limit ratios: ratios of the runtime limit to the total runtime of all pipelines on each dataset.

- **Convexification**: Solve the relaxed problem (Equation 6 with $v_i \in [0, 1], \forall i \in [n]$) with an SLSQP solver, sort the entries in the optimal solution v^*, and greedily add the pipeline with large v_i^* until the runtime limit is reached.
- **Greedy**: Solve the original integer programming problem (Equation 6) by the greedy algorithm (Algorithm 3), initialized by time-constrained QR (Algorithm 4).

For our problem, the greedy approach is superior, since the convexification method is prohibitive on our large 215×23424 error matrix. Hence we compare these methods on a subset of pipelines that only differ by estimators, 183 in total. This setting matches an experiment in [47]. Shown in Figure 9, we can see that:

1 The greedy method performs better for cold-start than convexification (Figure 9a): it selects informative designs that better predict the high-performing pipelines (Figure 9b).

2 The greedy method is more than 30× faster than convexification, which allows TensorOboe to devote its runtime budget to fitting pipelines instead of searching for the informative pipelines.

3 Shown in Figure 9d, the greedy algorithm still takes a fair amount of time if the number of designs we select is large; however, the dataset ranks we choose are less than 50, so it generally takes less than 10 seconds to choose informative pipelines. This time can be further reduced using Lemma 3.2.

Table 1: Runtime prediction accuracy on OpenML datasets

Pipeline estimator type	Runtime prediction accuracy	
	within factor of 2	within factor of 4
Adaboost	73.6%	86.9%
Decision tree	62.7%	78.9%
Extra trees	71.0%	83.8%
Gradient boosting	53.4%	77.5%
Gaussian naive Bayes	67.3%	82.3%
kNN	68.7%	84.4%
Logistic regression	53.6%	76.1%
Multilayer perceptron	74.5%	88.9%
Perceptron	64.5%	82.2%
Random Forest	69.5%	84.9%
Linear SVM	56.8%	79.5%

4.4 Pipeline Runtime Prediction Performance

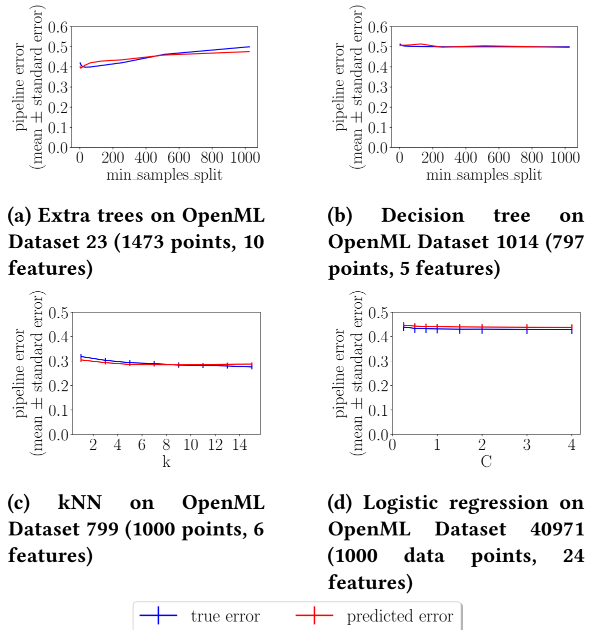
Runtime prediction accuracy is critical for the performance of our time-constrained pipeline selection system. Recall that our predictions use order-3 polynomial regression on $n^{\mathcal{D}}$ and $p^{\mathcal{D}}$, the numbers of data points and features in $\mathcal{D}$, and their logarithms. We shown in Table 1 that this runtime predictor performs well.

4.5 Learning the Hyperparameter Landscapes

Hyperparameter landscapes plot pipeline performance with respect to hyperparameter values. While parameter landscapes have been extensively studied, especially in the deep learning context (for example, [15, 23, 26]), hyperparameter landscapes are less studied. The previous sections focus on how we can choose among different pipeline component types. In this section, we show that our tensor surrogate model is able to learn hyperparameter landscapes of different estimator types that exhibit qualitatively different behaviors.

Figure 10 shows some examples of both real and predicted hyperparameter landscapes after running our system for 135 seconds. We can see that our predictions match the overall tendencies of the curves. Larger plots (Figure 13 in Appendix C) show our predictions also capture most of the small variations in these landscapes.

Note TensorOboe does not use a subroutine for hyperparameter optimization: it chooses the hyperparameter for each estimator from a predefined grid of values instead of optimizing hyperparameters by, for example, Bayesian optimization. The hyperparameter landscapes visualized here give confidence that grid search effectively samples performant hyperparameter settings within the range of hyperparameters: a coarse grid suffices.



(a) Extra trees on OpenML Dataset 23 (1473 points, 10 features)

(b) Decision tree on OpenML Dataset 1014 (797 points, 5 features)

(c) kNN on OpenML Dataset 799 (1000 points, 6 features)

(d) Logistic regression on OpenML Dataset 40971 (1000 data points, 24 features)

Figure 10: Hyperparameter landscape prediction examples.

5 OVERFITTING ANALYSIS

Two types of overfitting are of concern in AutoML systems: traditional overfitting (overfitting of models on training folds) and meta-overfitting (overfitting of AutoML surrogate models).

Traditional overfitting may happen in any machine learning system, and is often mitigated by controlling model complexity, cross validation on training set, etc. In TensorOboe, we always evaluate pipelines by k-fold cross validation, and build an ensemble since the pipeline with lowest cross-validation error may not be the one with lowest test error.

Meta-overfitting happens when meta-training datasets are biased in some sense, and when the surrogate model is so complex that it captures noise in addition to model performance. We mitigate meta-overfitting in the following ways: The OpenML meta-training datasets we collect have diverse topics ranging among multiple science and sociology disciplines. The surrogate model we use is low rank tensor decomposition, a model with low complexity. It denoises cross-validated pipeline error, as discussed in Section 2.

Meta-overfitting still presents many perils. The surrogate model may lack training instances. For example, the perceptron algorithm never performs the best on any meta-training dataset, as shown in Figure 5. Hence TensorOboe is unlikely ever to choose a perceptron pipeline. To mitigate this problem, we must collect pipeline performance in a larger space, or consider if the perceptron algorithm (for example) is truly dominated. Another possible source of meta-overfitting is that our meta-training datasets have no more than 10,000 points and smaller number of features. Order-3 polynomial runtime predictors may not generalize well to larger problems.

6 SUMMARY

This papers develops TensorOboe, a new structured model based on tensor decomposition for AutoML pipeline selection. The low multilinear rank tensor surrogate model allows us to efficiently learn about new datasets. The greedy experiment design method selects informative pipelines to evaluate. Together, TensorOboe tames the combinatorial complexity of the pipeline search space: the time complexity scales linearly in the number of candidates for each pipeline component. Empirically, TensorOboe relies on more offline work than competing methods, but such work pays off to improve on the state of the art in AutoML pipeline selection.

This paper is the first tensor method for pipeline selection. There are many avenues for improvement and extensions. For example, one could enlarge the pipeline search space, explore nonlinear surrogate models, explore different mechanisms to initialize the greedy method, develop an extension for neural architecture search, and design task-oriented pipeline selection systems that have better performance on domain-specific datasets. Further, the combinatorial space may be better handled by a method that dynamically adapts to the results of finished pipeline runs, thus leveraging its conditional structure.

ACKNOWLEDGMENTS

The authors gratefully acknowledge support from NSF Awards IIS-1943131 and CCF-1740822, the ONR Young Investigator Program, DARPA Award FA8750-17-2-0101, the Simons Institute, Canadian Institutes of Health Research, and Capital One.

REFERENCES

[1] Marcin Andrychowicz, Misha Denil, Sergio Gomez, Matthew W Hoffman, David Pfau, Tom Schaul, Brendan Shillingford, and Nando De Freitas. 2016. Learning to learn by gradient descent by gradient descent. In *Advances in neural information processing systems*. 3981–3989.

[2] George EP Box. 1976. Science and statistics. *J. Amer. Statist. Assoc.* 71, 356 (1976), 791–799.

[3] Stephen Boyd and Lieven Vandenberghe. 2004. *Convex optimization*. Cambridge University Press.

[4] Leo Breiman. 1996. Bagging predictors. *Machine learning* 24, 2 (1996), 123–140.

[5] J Douglas Carroll and Jih-Jie Chang. 1970. Analysis of individual differences in multidimensional scaling via an N-way generalization of "Eckart-Young" decomposition. *Psychometrika* 35, 3 (1970), 283–319.

[6] Arthur P Dempster, Nan M Laird, and Donald B Rubin. 1977. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society: Series B (Methodological)* 39, 1 (1977), 1–22.

[7] Thomas G Dietterich. 2000. Ensemble methods in machine learning. In *International workshop on multiple classifier systems*. Springer, 1–15.

[8] Iddo Drori, Yamuna Krishnamurthy, Remi Rampin, Raoni Lourenço, J One, Kyunghyun Cho, Claudio Silva, and Juliana Freire. 2018. AlphaD3M: Machine learning pipeline synthesis. In *AutoML Workshop at ICML*.

[9] Iddo Drori, Lu Liu, Yi Nian, Sharath C Koorathota, Jie S Li, Antonio Khalil Moretti, Juliana Freire, and Madeleine Udell. 2019. AutoML using Metadata Language Embeddings. *arXiv preprint arXiv:1910.03698* (2019).

[10] Dheeru Dua and Casey Graff. 2017. UCI Machine Learning Repository. http://archive.ics.uci.edu/ml

[11] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. 2015. Efficient and robust automated machine learning. In *Advances in neural information processing systems*. 2962–2970.

[12] Matthias Feurer, Jost Tobias Springenberg, and Frank Hutter. 2014. Using meta-learning to initialize Bayesian optimization of hyperparameters. In *International Conference on Meta-learning and Algorithm Selection*. Citeseer, 3–10.

[13] Matthias Feurer, Jan N van Rijn, Arlind Kadra, Pieter Gijsbers, Neeratyoy Mallik, Sahithya Ravi, Andreas Müller, Joaquin Vanschoren, and Frank Hutter. 2019. OpenML-Python: an extensible Python API for OpenML. *arXiv preprint arXiv:1911.02490* (2019).

[14] Nicolo Fusi, Rishit Sheth, and Melih Elibol. 2018. Probabilistic matrix factorization for automated machine learning. In *Advances in Neural Information Processing Systems*. 3348–3357.

[15] Timur Garipov, Pavel Izmailov, Dmitrii Podoprikhin, Dmitry P Vetrov, and Andrew G Wilson. 2018. Loss surfaces, mode connectivity, and fast ensembling of dnns. In *Advances in Neural Information Processing Systems*. 8789–8798.

[16] Gene H Golub and Charles F Van Loan. 2012. *Matrix computations*. Vol. 3. JHU press.

[17] Ming Gu and Stanley C Eisenstat. 1996. Efficient algorithms for computing a strong rank-revealing QR factorization. *SIAM Journal on Scientific Computing* 17, 4 (1996), 848–869.

[18] William W Hager. 1989. Updating the inverse of a matrix. *SIAM review* 31, 2 (1989), 221–239.

[19] Richard A Harshman et al. 1970. Foundations of the PARAFAC procedure: Models and conditions for an" explanatory" multimodal factor analysis. (1970).

[20] David A Harville. 1998. Matrix algebra from a statistician's perspective.

[21] Frank Hutter, Lin Xu, Holger H Hoos, and Kevin Leyton-Brown. 2014. Algorithm runtime prediction: Methods & evaluation. *Artificial Intelligence* 206 (2014), 79–111.

[22] RC St John and Norman R Draper. 1975. D-optimality for regression designs: a review. *Technometrics* 17, 1 (1975), 15–23.

[23] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. 2017. On large-batch training for deep learning: Generalization gap and sharp minima. (2017).

[24] Tamara G Kolda and Brett W Bader. 2009. Tensor decompositions and applications. *SIAM review* 51, 3 (2009), 455–500.

[25] Brenden M Lake, Tomer D Ullman, Joshua B Tenenbaum, and Samuel J Gershman. 2017. Building machines that learn and think like people. *Behavioral and brain sciences* 40 (2017).

[26] Hao Li, Zheng Xu, Gavin Taylor, Christoph Studer, and Tom Goldstein. 2018. Visualizing the loss landscape of neural nets. In *Advances in Neural Information Processing Systems*. 6389–6399.

[27] Sijia Liu, Parikshit Ram, Deepak Vijaykeerthy, Djallel Bouneffouf, Gregory Bramble, Horst Samulowitz, Dakuo Wang, Andrew Conn, and Alexander Gray. 2019. An ADMM Based Framework for AutoML Pipeline Configuration. *arXiv preprint arXiv:1905.00424* (2019).

[28] Vivek Madan, Mohit Singh, Uthaipon Tantipongpipat, and Weijun Xie. 2019. Combinatorial Algorithms for Optimal Design. In *Conference on Learning Theory*. 2210–2258.

[29] George L Nemhauser, Laurence A Wolsey, and Marshall L Fisher. 1978. An analysis of approximations for maximizing submodular set functionsâĂŤI. *Mathematical programming* 14, 1 (1978), 265–294.

[30] Randal S Olson and Jason H Moore. 2019. TPOT: A tree-based pipeline optimization tool for automating machine learning. In *Automated Machine Learning*. Springer, 151–160.

[31] Ivan V Oseledets. 2011. Tensor-train decomposition. *SIAM Journal on Scientific Computing* 33, 5 (2011), 2295–2317.

[32] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[33] Bernhard Pfahringer, Hilan Bensusan, and Christophe G Giraud-Carrier. 2000. Meta-Learning by Landmarking Various Learning Algorithms. In *ICML*. 743–750.

[34] Friedrich Pukelsheim. 1993. *Optimal design of experiments*. Vol. 50. SIAM.

[35] Robert E Schapire. 2003. The boosting approach to machine learning: An overview. In *Nonlinear estimation and classification*. Springer, 149–171.

[36] Zeyuan Shang, Emanuel Zgraggen, Benedetto Buratti, Ferdinand Kossmann, Philipp Eichmann, Yeounoh Chung, Carsten Binnig, Eli Upfal, and Tim Kraska. 2019. Democratizing data science through interactive curation of ml pipelines. In *Proceedings of the 2019 International Conference on Management of Data*. 1171–1188.

[37] Jack Sherman and Winifred J Morrison. 1950. Adjustment of an inverse matrix corresponding to a change in one element of a given matrix. *The Annals of Mathematical Statistics* 21, 1 (1950), 124–127.

[38] Qingquan Song, Hancheng Ge, James Caverlee, and Xia Hu. 2019. Tensor completion algorithms in big data analytics. *ACM Transactions on Knowledge Discovery from Data (TKDD)* 13, 1 (2019), 1–48.

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

[40] Ledyard R Tucker. 1966. Some mathematical notes on three-mode factor analysis. *Psychometrika* 31, 3 (1966), 279–311.

[41] Joaquin Vanschoren. 2018. Meta-learning: A survey. *arXiv preprint arXiv:1810.03548* (2018).

[42] Joaquin Vanschoren, Jan N. van Rijn, Bernd Bischl, and Luis Torgo. 2013. OpenML: Networked Science in Machine Learning. *SIGKDD Explorations* 15, 2 (2013), 49–60. https://doi.org/10.1145/2641190.2641198

[43] Abraham Wald. 1943. On the efficient design of statistical investigations. *The Annals of Mathematical Statistics* 14, 2 (1943), 134–140.

[44] M. Wistuba, N. Schilling, and L. Schmidt-Thieme. 2015. Learning hyperparameter optimization initializations. In *IEEE International Conference on Data Science and Advanced Analytics*. 1–10. https://doi.org/10.1109/DSAA.2015.7344817

[45] David H Wolpert. 1992. Stacked generalization. *Neural networks* 5, 2 (1992), 241–259.

[46] David H Wolpert and William G Macready. 1997. No free lunch theorems for optimization. *IEEE transactions on evolutionary computation* 1, 1 (1997), 67–82.

[47] Chengrun Yang, Yuji Akimoto, Dae Won Kim, and Madeleine Udell. 2019. OBOE: Collaborative filtering for AutoML model selection. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 1173–1183.

For reproducibility, refer to Section A for datasets and the pipeline search space. All the code is in the GitHub repository at https://github.com/udellgroup/oboe.

A REPRODUCIBILITY FOR META-TRAINING

A.1 Meta-training OpenML Datasets

Indices of the OpenML datasets we use for meta-training: 2, 3, 5, 7, 9, 11, 12, 13, 14, 15, 16, 18, 20, 22, 23, 24, 25, 27, 28, 29, 30, 31, 35, 36, 37, 38, 39, 40, 41, 42, 44, 46, 48, 50, 53, 54, 59, 60, 181, 182, 183, 187, 285, 307, 313, 316, 329, 336, 337, 338, 375, 377, 389, 446, 450, 458, 463, 469, 475, 694, 715, 717, 718, 720, 721, 723, 725, 728, 730, 732, 733, 735, 737, 740, 742, 743, 744, 745, 746, 747, 748, 749, 750, 751, 753, 763, 769, 773, 776, 778, 779, 788, 792, 794, 796, 797, 799, 803, 805, 806, 807, 813, 818, 819, 820, 824, 825, 826, 830, 832, 837, 838, 847, 853, 855, 863, 866, 869, 870, 871, 873, 877, 880, 884, 888, 896, 900, 903, 904, 906, 907, 908, 909, 910, 911, 912, 913, 915, 917, 920, 923, 925, 926, 933, 934, 935, 936, 937, 941, 943, 952, 953, 954, 955, 958, 962, 970, 971, 973, 976, 978, 979, 980, 983, 987, 991, 994, 995, 996, 997, 1005, 1011, 1012, 1014, 1016, 1020, 1021, 1022, 1025, 1026, 1038, 1039, 1041, 1042, 1048, 1049, 1050, 1054, 1056, 1063, 1065, 1067, 1068, 1069, 1071, 1073, 1100, 1115, 1116, 1121, 4134, 40966, 40971, 40975, 40978, 40979, 40981, 40982, 40983, 40984, 40994, 40997, 41000, 41004, 41005.

A.2 Meta-test UCI Datasets

banknote-authentication, blood-transfusion-service-center, breast-cancer-wisconsin-diagnostic, breast-cancer-wisconsin-original, breast-cancer-wisconsin-prognostic, chess-king-rook-vs-king-pawn, cnae-9, congressional-voting-records, connectionist-bench, connectionist-bench-sonar, contraceptive-method-choice, cylinder-bands, haberman-survival, heart-disease-cleveland, heart-disease-hungarian, heart-disease-va, hepatitis, hill-valley, hill-valley-noise, horse-colic, image-segmentation, indian-liver-patient, iris, libras-movement, mammographic-mass, monks-problems-2, ozone-level-detection-eight, ozone-level-detection-one, parkinsons, pen-based-recognition-handwritten-digits, planning-relax, qsar-biodegradation, seeds, seismic-bumps, statlog-project-german-credit, statlog-project-landsat-satellite, thoracic-surgery, thyroid-disease-allbp, thyroid-disease-allhyper, thyroid-disease-allhypo, thyroid-disease-allrep, thyroid-disease-ann-thyroid, thyroid-disease-dis, thyroid-disease-new-thyroid, thyroid-disease-sick, thyroid-disease-sick-euthyroid, thyroid-disease-thyroid-0387, wall-following-robot-navigation-2, wall-following-robot-navigation-24, wall-following-robot-navigation-4.

A.3 Pipeline Search Space

We build pipelines using scikit-learn [32] primitives. The available components are listed in Table 2. "null" denotes a pass-through.

B EXPERIMENT DESIGN FOR WEIGHTED LEAST SQUARES

When factorizing the error matrix by SVD, we approximate performance of different pipelines to different accuracies. Different accuracies can be characterized by different variances in the linear regression model, thus the weighted least squares (WLS) model that would theoretically give the best linear unbiased estimate to the new dataset embedding may perform better.

In detail, recall that the constrained D-optimal experiment design formulation relies on the assumption that given a low rank matrix multiplication model $X^\top Y = E$, the error term in linear regression $\epsilon \sim \mathcal{N}(0, \sigma^2 I)$, which means each pipeline is predicted to the same accuracy. In the WLS version of our pipeline performance estimation setting, the pipeline performance vector of the new dataset can be written as $e = Y^\top x + \epsilon$, in which $\epsilon \sim \mathcal{N}(0, \Sigma)$. $\Sigma = \text{diag}(\sigma_1^2, \sigma_2^2, \ldots, \sigma_n^2)$ is a covariance matrix; diagonal in the weighted least squares setting. For each pipeline $j \in [n]$, we estimate the variance by the sample variance of $e_j - X^\top y_j$, and show a histogram in Figure 11.

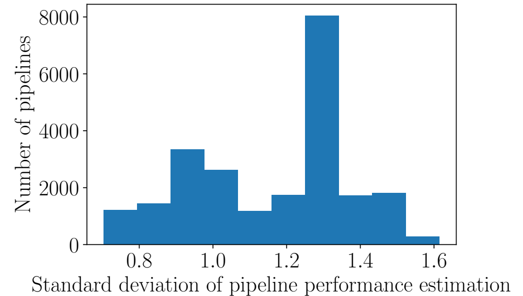


Figure 11: Standard deviation of prediction accuracy of each pipeline, across meta-training datasets.

In this case, the time-constrained D-experiment design problem to solve becomes

$$\begin{array}{ll} \text{minimize} & \log\det\left(\sum_{j=1}^n v_j \frac{y_j y_j^\top}{\sigma_j^2}\right)^{-1} \\ \text{subject to} & \sum_{j=1}^n v_j \hat{t}_j \leq \tau \\ & v_j \in \{0,1\}, \forall j \in [n]. \end{array} \tag{7}$$

The corresponding greedy approach, which we call *weighted-greedy*, is shown as Algorithm 5. It differs from the ordinary greedy approach in that each y_j is scaled by $1/\sigma_j$. Figure 12 shows its performance compared to convexification and greedy. We can see the weighted-greedy approach performs similarly to the ordinary greedy approach in our experiments.

Algorithm 5 Greedy algorithm for time-constrained D-design in WLS setting, with QR initialization

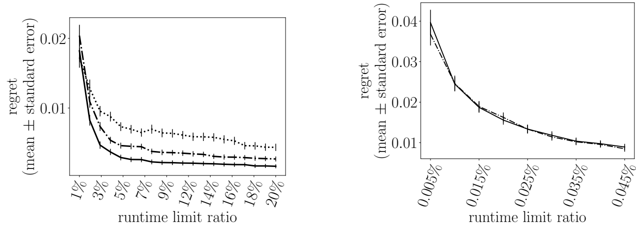
Input: design vectors $\{y_j\}_{j=1}^n$, in which $y_j \in \mathbb{R}^k$; pipeline estimation variances $\{\sigma_j^2\}_{j=1}^n$, (predicted) running time of all pipelines $\{\hat{t}_i\}_{i=1}^n$; maximum running time τ

Output: The selected set of designs $S \subseteq [n]$

1. $y_j \leftarrow y_j/\sigma_j, \forall j \in [n]$
2. $S_0 \leftarrow$ `QR_initialization`$(\{y_j\}_{j=1}^n, \{\hat{t}_i\}_{i=1}^n, \tau)$
3. $S \leftarrow$ `Greedy_without_repetition`$(\{y_j\}_{j=1}^n, \{\hat{t}_i\}_{i=1}^n, \tau, S_0)$

Table 2: Pipeline search space

Component	Algorithm type	Hyperparameter names (values)
Data imputer	Simple imputer	`strategy (mean, median, most_frequent, constant)`
Encoder	null	-
	OneHotEncoder	`handle_unknown (ignore), sparse (0)`
Standardizer	null	-
	StandardScaler	-
Dimensionality reducer	null	-
	PCA	`n_components (25%, 50%, 75%)`
	VarianceThreshold	-
	SelectKBest	`k (25%, 50%, 75%)`
Estimator	Adaboost	`n_estimators (50,100), learning_rate (1.0,1.5,2.0,2.5,3)`
	Decision tree	`min_samples_split (2,4,8,16,32,64,128,256,512,1024,0.01,0.001,1e-4,1e-5)`
	Extra trees	`min_samples_split (2,4,8,16,32,64,128,256,512,1024,0.01,0.001,1e-4,1e-5), criterion (gini,entropy)`
	Gradient boosting	`learning_rate (0.001,0.01,0.025,0.05,0.1,0.25,0.5), max_depth (3, 6), max_features (null,log2)`
	Gaussian naive Bayes	-
	Perceptron	-
	kNN	`n_neighbors (1,3,5,7,9,11,13,15), p (1,2)`
	Logistic regression	`C (0.25,0.5,0.75,1,1.5,2,3,4), solver (liblinear,saga), penalty (l1,l2)`
	Multilayer perceptron	`learning_rate_init (1e-4,0.001,0.01), learning_rate (adaptive), solver (sgd,adam), alpha (1e-4, 0.01)`
	Random forest	`min_samples_split (2,4,8,16,32,64,128,256,512,1024,0.01,0.001,1e-4,1e-5), criterion (gini,entropy)`
	Linear SVM	`C (0.125,0.25,0.5,0.75,1,2,4,8,16)`



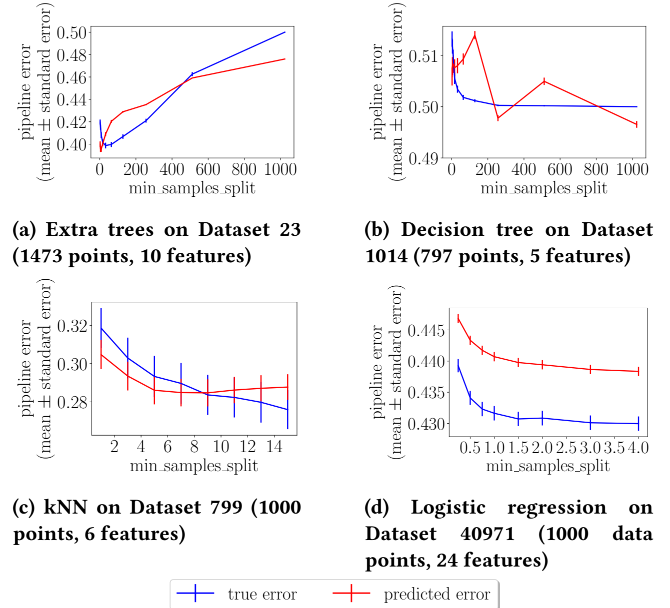
(a) Regret on the subsampled error matrix (215-by-183) for estimator search, including the weighted-greedy method.

(b) Regret on the full error matrix (215-by-23424) for pipeline search, including the weighted-greedy method.

convexification greedy weighted-greedy

Figure 12: Comparison of time-constrained experiment design methods, including the weighted-greedy method.

C ZOOMED-IN HYPERPARAMETER LANDSCAPES



(a) Extra trees on Dataset 23 (1473 points, 10 features)

(b) Decision tree on Dataset 1014 (797 points, 5 features)

(c) kNN on Dataset 799 (1000 points, 6 features)

(d) Logistic regression on Dataset 40971 (1000 data points, 24 features)

Figure 13: Zoomed-in hyperparameter landscapes in Figure 10. The y-axes here do not start from 0.