FLASH: Fast Bayesian Optimization for Data Analytic Pipelines

Yuyu Zhang   Mohammad Taha Bahadori   Hang Su   Jimeng Sun

Georgia Institute of Technology

{yuyu, bahadori, hangsu}@gatech.edu, jsun@cc.gatech.edu

ABSTRACT

Modern data science relies on data analytic pipelines to organize interdependent computational steps. Such analytic pipelines often involve different algorithms across multiple steps, each with its own hyperparameters. To achieve the best performance, it is often critical to select optimal algorithms and to set appropriate hyperparameters, which requires large computational efforts. Bayesian optimization provides a principled way for searching optimal hyperparameters for a single algorithm. However, many challenges remain in solving pipeline optimization problems with high-dimensional and highly conditional search space. In this work, we propose Fast LineAr SearcH (FLASH), an efficient method for tuning analytic pipelines. FLASH is a two-layer Bayesian optimization framework, which firstly uses a parametric model to select promising algorithms, then computes a nonparametric model to fine-tune hyperparameters of the promising algorithms. FLASH also includes an effective caching algorithm which can further accelerate the search process. Extensive experiments on a number of benchmark datasets have demonstrated that FLASH significantly outperforms previous state-of-the-art methods in both search speed and accuracy. Using 50% of the time budget, FLASH achieves up to 20% improvement on test error rate compared to the baselines. FLASH also yields state-of-the-art performance on a real-world application for healthcare predictive modeling.

Keywords

Bayesian optimization; Automated hyperparameter tuning; Data analytic pipeline; Health analytics

1. INTRODUCTION

Modern data science often requires many computational steps such as data preprocessing, feature extraction, model building, and model evaluation, all connected in a *data analytic pipeline*. Pipelines provide a natural way to represent, organize and standardize data analytic tasks,

which are considered to be an essential element in the data science field [11] due to their key role in large-scale data science projects. Many machine learning toolboxes such as scikit-learn [36], RapidMiner [31], SPSS [10], Apache Spark [30] provide mechanisms for configuring analytic pipelines.

An analytic pipeline skeleton is shown in Figure 1. Each step, such as feature preprocessing and classification, includes many algorithms to choose from. These algorithms usually require users to set hyperparameters, ranging from optimization hyperparameters such as learning rate and regularization coefficients, to model design hyperparameters such as the number of trees in random forest and the number of hidden layers in neural networks. There are an exponential number of choices for the combination of algorithms and hyperparameters in a given analytic pipeline skeleton. Because of the interdependency between all the algorithms and their hyperparameters, the choices can have huge impact on the performance of the best model.

Tuning hyperparameters of a single algorithm can be viewed as an optimization problem of a black-box objective function, which is noisy and often expensive to evaluate. Here the input of black-box are the hyperparameters, and the objective function is the output performance such as accuracy, precision and recall. To tackle this problem, simple methods have been applied such as grid or random search [5, 3]. While on difficult problems where these simple approaches are not efficient, a more promising model-based approach is Bayesian optimization [32, 27, 7, 39]. The high-level idea of Bayesian optimization is to define a relatively cheap surrogate function and use that to search the hyperparameter space. Indeed, there exist other global optimization methods, such as evolutionary algorithms [2] and optimistic optimization [33]. We choose Bayesian optimization framework due to its great performance in practice. Recently, Bayesian optimization methods have been shown to outperform other methods on various tasks, and in some cases even beat human domain experts to achieve better performance via tuning hyperparameters [42, 4].

Despite its success, applying Bayesian optimization for tuning analytic pipelines faces several significant challenges: Existing Bayesian optimization methods are usually based on nonparametric models, such as Gaussian process and random forest. A major drawback of these methods is that they require a large number of observations to find reasonable solutions in high-dimensional space. When tuning a single algorithm with several hyperparameters, Bayesian optimization works well with
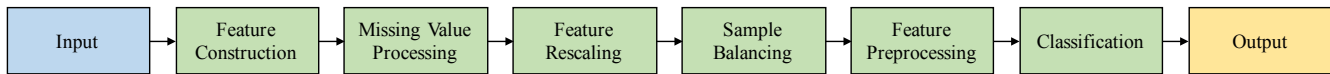
Figure 1: A typical data analytic pipeline.

just a few observations. However, when it comes to pipeline tuning, thousands of possible combinations of algorithms plus their hyperparameters jointly create a large hierarchical high-dimensional space to search over, whereas existing methods tend to become inefficient. Wang et al. [46] tackled the high-dimensional problem by making a low effective dimensional assumption. However, it is still a flat Bayesian optimization method and not able to handle the exploding dimensionality problem caused by hierarchically structured hyperparameters in analytic pipeline tuning.

Motivating example: We build an analytic pipeline for classification task (details in Section 4). If we give 10 trials for each hyperparameter over 1,456 unique pipeline paths and 102 hyperparameters, we have more than 2 million configurations, which can take years to complete with a brute-force search. Even with the state-of-the-art Bayesian optimization algorithm such as Sequential Model-based Algorithm Configuration (SMAC) [22], the process can still be slow as shown in Figure 2. If we know the optimal algorithms ahead time (Oracle) with just hyperparameter tuning of the optimal algorithms, we can obtain significant time saving, which is however not possible. Finally, our proposed method FLASH can converge towards the oracle performance much more quickly than SMAC.

In this paper, we propose a two-layer Bayesian optimization algorithm called Fast LineAr SearcH (FLASH): the first layer for selecting algorithms, and the second layer for tuning the hyperparameters of selected algorithms. FLASH is able to outperform the state-of-the-art Bayesian optimization algorithms by a large margin, as shown in Figure 2. By designing FLASH, we make three main contributions:

- We propose a linear model for propagation of error (or other quantitative metrics) in analytic pipelines. We also propose a Bayesian optimization algorithm for minimizing the aggregated error using our linear error model. Our proposed mechanism can be considered as a hybrid model: a parametric linear model for fast exploration and pruning of the algorithm space, followed by a nonparametric hyperparameter fine-tuning algorithm.

- We propose to initialize the hyperparameter tuning algorithm using the optimal design strategy [1, 15, 38] which is more robust than the random initialization. We also propose a fast greedy algorithm to efficiently solve the optimal design problem for any given analytic pipeline.

- Finally, we introduce a caching algorithm that can significantly accelerate the tuning process. In particular, we model the (time) cost of each algorithm, and incorporate that in the optimization process. This ensures the efficiency of fast search.

We demonstrate the effectiveness of FLASH with extensive experiments on a number of difficult problems. On the benchmark datasets for pipeline configurations tuning, FLASH substantially improves the previous state of the art
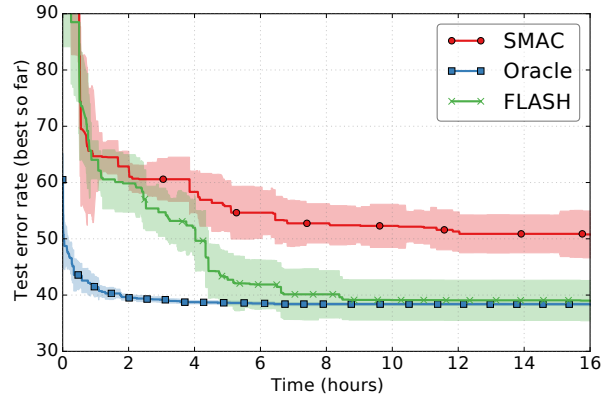

Figure 2: Performance comparison of a data analytic pipeline on MRBI dataset, including the previous state of the art SMAC, proposed method FLASH, and Oracle (i.e., pretending the optimal algorithm configuration is given, and only performing hyperparameter tuning with SMAC on those algorithms). We show the median percent error rate on the test set along with standard error bars (generated by 10 independent runs) over time. FLASH outperforms SMAC by a big margin and converges toward Oracle performance quickly.

by 7% to 25% in test error rate within the same time budget. We also experiment with large-scale real-world datasets on healthcare data analytic tasks where FLASH also exhibits superior results.

2. BACKGROUND AND RELATED WORK

2.1 Data Analytic Pipelines

The data analytic pipeline refers to a framework consisting of a sequence of computational transformations on the data to produce the final predictions (or outputs) [26]. Pipelines help users better understand and organize the analysis task, as well as increase the reusability of algorithm implementations in each step. Several existing widely adopted machine learning toolboxes provide the functionality to run analytic pipelines. Scikit-learn [36] and Spark ML [30] provide programmatic ways to instantiate a pipeline. SPSS [10] and RapidMiner [31] provide a visual way to assemble an analytic pipeline instance together and run. Microsoft Azure Machine Learning[1] provides a similar capability in a cloud setting. There are also specialized pipelines, such as PARAMO [35] in healthcare data analysis.

However, a major difficulty in using these systems is that none of the above described tools is able to efficiently help users decide which algorithms to use in each step. Some of the tools such as scikit-learn, Spark ML, and PARAMO

---

[1]https://studio.azureml.net

allow searching all possible pipeline paths and tuning the hyperparameters of each step using an expensive grid search approach. While the search process can be sped up by running in parallel, the search space is still too large for the exhaustive search algorithms.

2.2 Bayesian Optimization

Bayesian optimization is a well-established technique for global and black-box optimization problems. In a nutshell, it comprises two main components: a probabilistic model and an acquisition function. For the probabilistic model, there are several popular choices: Gaussian process [41, 42], random forest such as Sequential Model-based Algorithm Configuration (SMAC) [22], and density estimation models such as Tree-structured Parzen Estimator (TPE) [5]. Given any of these models, the posterior mean and variance of a new input can be computed, and used for computation of the acquisition function. The acquisition function defines the criterion to determine future input candidates for evaluation. Compared to the objective function, the acquisition function is chosen to be relatively cheap to evaluate, so that the most promising next input for querying can be found quickly. Various forms of acquisition functions have been proposed [43, 20, 45, 21]. One of the most prominent acquisition function is the *Expected Improvement* (EI) function [32], which has been widely used in Bayesian optimization. In this work, we use EI as our acquisition function, which is formally described in Section 3.

Bayesian optimization is known to be successful in tuning hyperparameters for various learning algorithms on different types of tasks [42, 14, 4, 41, 46]. Recently, for the problem of pipeline configurations tuning, several Bayesian optimization based systems have been proposed: Auto-WEKA [44] which applies SMAC [22] to WEKA [17], auto-sklearn [13] which applies SMAC to scikit-learn [36], and hyperopt-sklearn [24] which applies TPE [5] to scikit-learn. The basic idea of applying Bayesian optimization to pipeline tuning is to expand the hyperparameters of all algorithms and create large search space to perform optimization as we will show in the experiments. However, for practical pipelines the space becomes too large which hinders convergence of the optimization process. Auto-sklearn [13] uses a meta-learning algorithm that leverages performance history of algorithms on existing datasets to reduce the search space. However, in real-world applications, we often have unique datasets and tasks such that finding similar datasets and problems for the meta-learning algorithm will be difficult.

3. METHODOLOGY

A data analytic pipeline $G = (V, E)$ can be represented as a multi-step Directed Acyclic Graph (DAG), where V is the set of algorithms, and E is the set of directed edges indicating dependency between algorithms. Algorithms are distributed among multiple steps. Let $V_i^{(k)}$ denote the ith algorithm in the kth step. Each directed edge $(V_i^{(k)}, V_j^{(k+1)}) \in E$ represents the connection from algorithm $V_i^{(k)}$ to $V_j^{(k+1)}$. Note that there is no edge between algorithms in the same step. We also have an input data vertex V_{in} which points to all algorithms in the first step, and an output vertex V_{out} which is pointed by all algorithms in the last step.
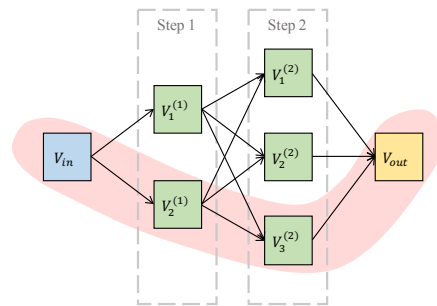

Figure 3: A toy example of data analytic pipeline. One possible pipeline path, flowing from the input V_{in} to the output V_{out}, is highlighted in shaded area.

A *pipeline path* is any path from the input V_{in} to the output V_{out} in pipeline graph G. To denote a pipeline path of K steps, we use K one-hot vectors $\boldsymbol{p}^{(k)}$ ($1 \leq k \leq K$), each denoting the algorithm selected in the k-th step. Thus, the concatenation of one-hot vectors $\boldsymbol{p} = [\boldsymbol{p}^{(1)}, \ldots, \boldsymbol{p}^{(K)}] \in \{0, 1\}^N$ denotes a pipeline path, where N is the total number of algorithms in the pipeline G. Figure 3 shows a small data analytic pipeline with two steps. The first step contains two algorithms, and the second step contains three. One possible pipeline path is highlighted in the shaded area. On this pipeline path, $V_2^{(1)}$ and $V_3^{(2)}$ are selected in the first and second step, so that we have $\boldsymbol{p}^{(1)} = [0, 1]$ and $\boldsymbol{p}^{(2)} = [0, 0, 1]$. Thereby, the highlighted pipeline path is given by $\boldsymbol{p} = [\boldsymbol{p}^{(1)}, \boldsymbol{p}^{(2)}] = [0, 1, 0, 0, 1]$. For any pipeline path $\boldsymbol{p}$, we concatenate all of its hyperparameters in a vector $\boldsymbol{\lambda_p}$. The pair of path and hyperparameters, i.e. $(\boldsymbol{p}, \boldsymbol{\lambda_p})$, forms a pipeline configuration to be run. For ease of reference, we list the notations in Table 1.

The problem of tuning data analytic pipelines can be formalized as an optimization problem:

Problem 1. *Given a data analytic pipeline G with input data $\mathcal{D}$, resource budget T, evaluation metric function $m(G, \mathcal{D}; \boldsymbol{p}, \boldsymbol{\lambda_p})$, resource cost of running pipeline $\tau(G, \mathcal{D}; \boldsymbol{p}, \boldsymbol{\lambda_p})$, how to find the pipeline path $\boldsymbol{p}$ and its hyperparameters $\boldsymbol{\lambda_p}$ with best performance $m^\star$?*

The performance of the best pipeline path is denoted by $m^\star = \min_{\boldsymbol{p}, \boldsymbol{\lambda_p}} m(G, \mathcal{D}; \boldsymbol{p}, \boldsymbol{\lambda_p})$ subject to budget T. The objective is to approach the optimal performance within the budget T via optimizing over $\boldsymbol{p}, \boldsymbol{\lambda_p}$; i.e., we would like our solution $\widehat{\boldsymbol{p}}, \widehat{\boldsymbol{\lambda}}_{\widehat{\boldsymbol{p}}}$ to satisfy $m(G, \mathcal{D}; \widehat{\boldsymbol{p}}, \widehat{\boldsymbol{\lambda}}_{\widehat{\boldsymbol{p}}}) \leq m^\star + \epsilon$ for small values of ϵ.

To efficiently tackle this problem, we propose a two-layer Bayesian optimization approach named Fast LineAr SearcH (FLASH). We generally introduce the idea of linear model and describe the algorithm in Section 3.1. An immediate advantage of using linear model is that we can use more principled initialization instead of random initialization, as discussed in Section 3.2. We use cost-sensitive modeling to prune the pipeline, as described in Section 3.3. Finally, we accelerate the entire optimization procedure via pipeline caching, which we describe in Section 3.4.

3.1 Two-layer Bayesian Optimization

Inspired by the performance of linear regression under

Table 1: Mathematical notations used in this paper.

Symbol	Description
G	data analytic pipeline
V	set of algorithms in G
E	set of dependency between algorithms
K	total number of steps in G
N	total number of algorithms in G
$\mathcal{D}$	input data of pipeline
V_{in}	input vertex of G
V_{out}	output vertex of G
$V_i^{(k)}$	ith algorithm in kth step
$\boldsymbol{p}^{(k)}$	one-hot vector for kth step
$\boldsymbol{\lambda}_{\boldsymbol{p}}^{(k)}$	hyperparameters for kth step
$\boldsymbol{p}$	pipeline path
$\boldsymbol{\lambda}_{\boldsymbol{p}}$	all hyperparameters of $\boldsymbol{p}$
$m(\cdot)$	evaluation metric function
$\tau(\cdot)$	time cost of running pipeline
T_{init}	budget for Phase 1
T_{prune}	budget for Phase 2
T_{total}	budget for Phase 3

Algorithm 1: Fast Linear Search (FLASH)

input : Data analytic pipeline G; input data $\mathcal{D}$; total budget for entire optimization T_{total}; budget for initialization T_{init}; budget for pipeline pruning T_{prune}; number of top pipeline paths r

output: Optimized pipeline configuration $\widehat{\boldsymbol{p}}$ and $\widehat{\boldsymbol{\lambda}}_{\widehat{\boldsymbol{p}}}$

/* Phase 1: Initialization (Section 3.2)   */
1 **while** budget T_{init} not exhausted **do**
2    $\boldsymbol{p} \leftarrow$ new pipeline path from Algorithm 2
3    $\boldsymbol{\lambda}_{\boldsymbol{p}} \leftarrow$ random hyperparameters for $\boldsymbol{p}$
4    $m, \tau \leftarrow$ RunPipeline$(G, \mathcal{D}; \boldsymbol{p}, \boldsymbol{\lambda}_{\boldsymbol{p}})$ with Algorithm 3
5    $\boldsymbol{P} \leftarrow [\boldsymbol{P}; \boldsymbol{p}^\top], \boldsymbol{m} \leftarrow [\boldsymbol{m}, m], \boldsymbol{\tau} \leftarrow [\boldsymbol{\tau}, \tau]$
6    $\boldsymbol{\beta} \leftarrow \widehat{\boldsymbol{\beta}}(\boldsymbol{P}, \boldsymbol{m}), \boldsymbol{\beta}_\tau \leftarrow \widehat{\boldsymbol{\beta}}_\tau(\boldsymbol{P}, \boldsymbol{\tau})$ using Eq. (1)

/* Phase 2: Pipeline pruning (Section 3.3)   */
7 **while** budget T_{prune} not exhausted **do**
8    $\boldsymbol{p} \leftarrow \arg\max_{\boldsymbol{p}} EIPS(\boldsymbol{p}, \boldsymbol{P}, \boldsymbol{\beta}, \boldsymbol{\beta}_\tau)$ using Eq. (4)
9    $\boldsymbol{\lambda}_{\boldsymbol{p}} \leftarrow$ random hyperparameters for $\boldsymbol{p}$
10    $m, \tau \leftarrow$ RunPipeline$(G, \mathcal{D}; \boldsymbol{p}, \boldsymbol{\lambda}_{\boldsymbol{p}})$ with Algorithm 3
11    $\boldsymbol{P} \leftarrow [\boldsymbol{P}; \boldsymbol{p}^\top], \boldsymbol{m} \leftarrow [\boldsymbol{m}, m], \boldsymbol{\tau} \leftarrow [\boldsymbol{\tau}, \tau]$
12    $\boldsymbol{\beta} \leftarrow \widehat{\boldsymbol{\beta}}(\boldsymbol{P}, \boldsymbol{m}), \boldsymbol{\beta}_\tau \leftarrow \widehat{\boldsymbol{\beta}}_\tau(\boldsymbol{P}, \boldsymbol{\tau})$ using Eq. (1)
13 $G' \leftarrow$ construct subgraph of G with top r pipeline paths with largest $EIPS(\boldsymbol{p}, \boldsymbol{P}, \boldsymbol{\beta}, \boldsymbol{\beta}_\tau)$ using Eq. (4)

/* Phase 3: Pipeline tuning   */
14 $S \leftarrow$ history observations within G'
15 Initialize model $\mathcal{M}$ given S
16 **while** budget T_{total} not exhausted **do**
17    $\boldsymbol{p}, \boldsymbol{\lambda}_{\boldsymbol{p}} \leftarrow$ next candidate from $\mathcal{M}$
18    $m \leftarrow$ RunPipeline$(G', \mathcal{D}; \boldsymbol{p}, \boldsymbol{\lambda}_{\boldsymbol{p}})$ with Algorithm 3
19    $S \leftarrow S \cup \{(\boldsymbol{p}, \boldsymbol{\lambda}_{\boldsymbol{p}}, m)\}$
20   Update $\mathcal{M}$ given S
21 $\widehat{\boldsymbol{p}}, \widehat{\boldsymbol{\lambda}}_{\widehat{\boldsymbol{p}}} \leftarrow$ Best configuration so far found for G'

---

model misspecification [48, 16, 28] and superior sample complexity compared to more flexible nonparametric techniques [47], we seek parametric models for propagation of error (or other quantitative metrics) in analytic pipelines. The high level idea of FLASH is as follows: we propose a linear model for estimating the propagation of error (or any other metric) in a given analytic pipeline. The linear model assumes that the performance of algorithms in different steps are independent, and the final performance is additive from all algorithms. That is, we can imagine that each algorithm is associated with a performance metric, and the total performance of a pipeline path is the sum of the metrics for all algorithms in the path. This linear model will replace the Gaussian process or random forest in the initial stages of the pipeline tuning process. In the rest of this section, we provide the details of Bayesian optimization with our linear model.

We apply the linear model only to the pipeline selection vector $\boldsymbol{p}$ and assume that the variations due to hyperparameters of the algorithms are captured in the noise term. That is, we assume that the error of any pipeline path $\boldsymbol{p}$ can be written as

$$m = \boldsymbol{\beta}^\top \boldsymbol{p} + \varepsilon$$

where $\boldsymbol{\beta} \in \mathbb{R}^N$ denotes the parameters of the linear model. Given a set of observations of the algorithm selection and the corresponding evaluation metric for the selected pipeline path in the form of $(\boldsymbol{p}_i, m_i)$, $i = 1, \ldots, n$, we can fit this model and infer its mean $\mu(\boldsymbol{p})$ and variance $\sigma^2(\boldsymbol{p})$ of the performance estimation for any new pipeline path represented by $\boldsymbol{p}$. In particular, let the design matrix $\boldsymbol{P} \in \mathbb{R}^{n \times N}$ denote the stacked version of the pipeline paths, i.e., $\boldsymbol{P} = [\boldsymbol{p}_1, \ldots, \boldsymbol{p}_n]^\top$, and $\boldsymbol{m} \in \mathbb{R}^n$ be the corresponding response values of the evaluation metrics, $\boldsymbol{m} = [m_1, \ldots, m_n]$. We use the following L_2 regularized linear regression to obtain the robust estimate for $\boldsymbol{\beta}$ from history observations:

$$\widehat{\boldsymbol{\beta}}(\boldsymbol{P}, \boldsymbol{m}) = \underset{\boldsymbol{\beta}}{\arg\min} \left\{ \frac{1}{n} \|\boldsymbol{P}\boldsymbol{\beta} - \boldsymbol{m}\|_2^2 + \lambda \|\boldsymbol{\beta}\|_2^2 \right\} \quad (1)$$

where for any vector $\boldsymbol{x} \in \mathbb{R}^n$ the L_2 norm is defined as $\|\boldsymbol{x}\|_2 = \sqrt{\left(\sum_{i=1}^n x_i^2\right)}$. The predictive distribution for the linear model is Gaussian with mean $\widehat{\mu}_{\boldsymbol{p}} = \widehat{\boldsymbol{\beta}}^\top \boldsymbol{p}$ and variance $\widehat{\sigma}_{\boldsymbol{p}} = \sigma_\varepsilon^2 (1 + \boldsymbol{p}^\top (\boldsymbol{P}^\top \boldsymbol{P} + \lambda \boldsymbol{I})^{-1} \boldsymbol{p})$ where σ_ε^2 is the variance of noise in the model. We estimate σ_ε as follows: the residual in the ith observation is computed as $\widehat{\epsilon}_i = \widehat{\mu}_i - m_i$ where $\widehat{\mu}_i = \widehat{\boldsymbol{\beta}}^\top \boldsymbol{p}_i$ is the estimate of m_i by our model. Thus, the variance of the residual can be found as $\widehat{\sigma}_\epsilon^2 = var(\widehat{\mu}_i - m_i)$ where $var(\cdot)$ denotes the variance operator.

To perform Bayesian optimization with linear model, we use the popular *Expected Improvement* (EI) criteria, which recommends to select the next sample $\boldsymbol{p}_{t+1}$ such that the following acquisition function is maximized. The acquisition function represents the expected improvement over the best observed result m^+ at a new pipeline path $\boldsymbol{p}$ [44]:

$$EI(\boldsymbol{p}) = \mathbb{E}[I_{m^+}(\boldsymbol{p})] = \sigma_{\boldsymbol{p}}[u\Phi(u) + \phi(u)] \quad (2)$$

where $u = \frac{m^+ - \xi - \mu_{\boldsymbol{p}}}{\sigma_{\boldsymbol{p}}}$ and ξ is a parameter to balance the trade-off between exploitation and exploration. EI function is maximized for paths with small values of $m_{\boldsymbol{p}}$ and large values of $\sigma_{\boldsymbol{p}}$, reflecting the exploitation and exploration trade-offs, respectively. To be more specific, larger ξ encourages more exploration in selecting the next sample. The functions $\Phi(\cdot)$ and $\phi(\cdot)$ represent CDF and PDF of standard normal distribution, respectively. The

idea of Bayesian optimization with EI is that at each step, we compute the EI with the predictive distribution of the existing linear model and find the pipeline path that maximizes EI. We choose that path and run the pipeline with it to obtain a new $(\boldsymbol{p}_i, m_i)$ pair. We use this pair to refit and update our linear model and repeat the process. Later on we also present an enhanced version of EI via normalizing it by cost called *Expected Improvement Per Second* (EIPS).

We provide the full details of FLASH in Algorithm 1. While the main idea of FLASH is performing Bayesian optimization using linear model and EI, it has several additional ideas to make it practical. Specifically, FLASH has three phases:

- Phase 1, we initialize the algorithm using ideas from optimal design, see Section 3.2. The budget is bounded by T_{init}.

- Phase 2, we leverage Bayesian optimization to find the top r best pipeline paths and prune the pipeline G to obtain simpler one G'. The budget is bounded by T_{prune}[2].

- Phase 3, we use general model-based Bayesian optimization methods to fine-tune the pruned pipeline together with their hyperparameters.

In Phase 3, we use state-of-the-art Bayesian optimization algorithm, either SMAC or TPE. These algorithms are iterative: they use a model $\mathcal{M}$ such as Gaussian process or random forest and use EI to pick up a promising pipeline path with hyperparameters for running, and then update the model with the new observation just obtained, and again pick up the next one for running. The budget is bounded by T_{total}. Note that our algorithm is currently described for a sequential setting but can be easily extended to support parallel runs of multiple pipeline paths as well.

3.2 Optimal Design for Initialization

Most Bayesian optimization algorithms rely on random initialization which can be inefficient; for example, it may select duplicate pipeline paths for initialization. Intuitively, the pipeline paths used for initialization should cover the pipeline graph well, such that all algorithms are included enough times in the initialization phase. The ultimate goal is to select a set of pipeline paths for initialization such that the error in estimation of $\boldsymbol{\beta}$ is minimized. Given our proposed linear model, we can find the optimal strategy for initialization to make sure the pipeline graph is well covered and the tuning process is robust. In this section, we describe different optimality criteria studied in statistical experiment design [1, 15] and active learning [38], and design an algorithm for initialization step of FLASH.

Given a set of pipeline paths with size n, there are several different optimality criteria in terms of the eigenvalues of the Gram matrix $\boldsymbol{H} = \sum_{i=1}^{n} \boldsymbol{p}_i \boldsymbol{p}_i^\top$ as follows [38]:

A-optimality: maximize $\sum_{\ell=1}^{n} \lambda_\ell(\boldsymbol{H})$.

D-optimality: maximize $\prod_{\ell=1}^{n} \lambda_\ell(\boldsymbol{H})$.

E-optimality: maximize $\lambda_n(\boldsymbol{H})$, the nth largest eigenvalue.

---

[2]In practice, it is better to use the time normalized EI (that is EIPS) during Phase 2; this idea is described in Section 3.3.

Algorithm 2: Initialization with Optimal Design

/* *Batch version* */
input : B initial candidates $\{\boldsymbol{p}_i\}_{i=1}^{B}$; number of desired pipeline paths n_{init}
output: Optimal set of pipeline paths Q
1 $\boldsymbol{p}_1 \leftarrow$ random pipeline path for initialization
2 $\boldsymbol{H} \leftarrow \boldsymbol{p}_1 \boldsymbol{p}_1^\top$
3 $Q \leftarrow \{\boldsymbol{p}_1\}$
4 **for** $\ell = 2, \ldots, n_{init}$ **do**
5 $\quad$ $j^\star \leftarrow \text{argmax}_j D_\ell(\boldsymbol{H} + \boldsymbol{p}_j \boldsymbol{p}_j^\top)$ for $j = 1, \ldots, B$.
6 $\quad$ $\boldsymbol{H} \leftarrow \boldsymbol{H} + \boldsymbol{p}_{j^\star} \boldsymbol{p}_{j^\star}^\top$
7 $\quad$ $Q \leftarrow Q \cup \{\boldsymbol{p}_{j^\star}\}$

/* *Online version* */
input : B candidates $\{\boldsymbol{p}_i\}_{i=1}^{B}$; current Gram matrix $\boldsymbol{H}$
output: Next pipeline path $\boldsymbol{p}_{j^\star}$, $j^\star \in \{1, \ldots, B\}$
8 $j^\star \leftarrow \text{argmax}_j D_\ell(\boldsymbol{H} + \boldsymbol{p}_j \boldsymbol{p}_j^\top)$ for $j = 1, \ldots, B$

It is easy to see that any arbitrary set of pipeline path designs satisfies the A-optimality criterion.

Proposition 1. *Any arbitrary set of pipeline paths with size n is a size-n A-optimal design.*

PROOF. For any arbitrary set of pipeline paths with size n, we have:

$$\sum_{\ell=1}^{n} \lambda_\ell(\boldsymbol{H}) = \text{tr}(\boldsymbol{H}) = \text{tr}\left(\sum_{i=1}^{n} \boldsymbol{p}_i \boldsymbol{p}_i^\top\right) = \sum_{i=1}^{n} \text{tr}\left(\boldsymbol{p}_i \boldsymbol{p}_i^\top\right) = nK.$$

The last step is due to particular pattern of $\boldsymbol{p}$ in our problem. Thus, we show that $\sum_{\ell=1}^{n} \lambda_\ell(\boldsymbol{H})$ is constant, independent of the design of pipeline paths. $\square$

Proposition 1 rules out use of A-optimality in pipeline initialization. Given the computational complexity of E-optimality and the fact that it intends for optimality in the extreme cases, we choose D-optimality criterion. The D-optimality criterion for design of optimal linear regression can be stated as follows: suppose we are allowed to evaluate n_{init} samples $\boldsymbol{p}_i$, $i = 1, \ldots, n_{\text{init}}$, these samples should be designed such that the determinant of the Gram matrix $\boldsymbol{H}$ is maximized. While we can formulate an optimization problem that directly finds $\boldsymbol{p}_i$ values, we found that an alternative approach can be computationally more efficient. In this approach, we first generate B candidate pipeline paths for an integer B larger than the number of algorithms in the pipeline N. This set may include all possible pipeline paths if the total number of paths is small. Then, our goal becomes selecting a subset of size n_{init} from them. We can formulate the optimal design as follows

$$\boldsymbol{a}^\star = \underset{\boldsymbol{a}}{\text{argmax}} \ \det\left\{\sum_{i=1}^{B} a_i \boldsymbol{p}_i \boldsymbol{p}_i^\top\right\} \qquad (3)$$

$$s.t. \qquad \boldsymbol{a} \in \{0,1\}^K, \qquad \mathbf{1}^\top \boldsymbol{a} = n_{\text{init}}.$$

The last constraint $\mathbf{1}^\top \boldsymbol{a} = n_{\text{init}}$ indicates that only n_{init} pipeline paths should be selected. The objective function is concave in terms of continuous valued $\boldsymbol{a}$ [6, Chapter 3.1.5]. Thus, a traditional approach is to solve it by convex programming after relaxation of the integrality constraint on $\boldsymbol{a}$. The matrix in the argument of the determinant is only

N-dimensional which means calculation of the determinant should be fast. Nestrov's accelerated gradient descent [34] or Frank-Wolfe's [23] algorithms can be used for efficiently solving such problems.

An even faster solution can be found by using greedy forward selection ideas which are fast and popular for optimal experiment design, for example see [37, 18, 25] and the references therein. To apply greedy technique to our problem, we initialize the solution by picking one of the pipeline path $\boldsymbol{H} = \boldsymbol{p}_i\boldsymbol{p}_i^\top$. Then, at ℓth step, we add the path that maximizes $j^\star = \mathrm{argmax}_j\, D_\ell(\boldsymbol{H} + \boldsymbol{p}_j\boldsymbol{p}_j^\top)$ where $D_\ell(\boldsymbol{H}) = \prod_{i=1}^{\min(\ell,p)} \lambda_i(\boldsymbol{H})$ denotes the product of top $\min(\ell,p)$ eigenvalues of its argument. The algorithm is described in Algorithm 2. The optimization problem in Eq. (3) appears in other fields such as optimal facility location and sensor planning where greedy algorithm is known to have a $1 - \frac{1}{e}$ approximation guarantee [8, 40].

One further desirable property of the greedy algorithm is that it is easy to run it under a time budget constraint. We call this version the online version in Algorithm 2, where instead of a fixed number of iteration n_{init}, we run it until the exhaustion of our time budget. See Line 2 in Algorithm 1 and the online version of Algorithm 2.

3.3 Cost-sensitive Modeling

The *Expected Improvement* aims at approaching the true optimal (doing well) within a small number of function evaluations (doing fast). However, the time cost of each function evaluation may vary a lot due to different settings of hyperparameters. This problem is particularly highlighted in pipeline configurations tuning, since the choice of algorithms can make a huge difference in running time. Therefore, fewer pipeline runs are not always "faster" in terms of wall-clock time. Also, in practice, what we care about is the performance we can get within limited resource budget, rather than within certain evaluation times. That is why we need cost-sensitive modeling for the pipeline tuning problem.

Expected Improvement Per Second (EIPS) [41] proposes another acquisition function for tuning of a single learning algorithm by dividing the EI of each hyperparameter by its runtime. To apply EIPS in pipeline tuning problem, we use a separate linear model to model the total runtime of pipeline paths. Similar to the linear model for error propagation, the linear model for time assumes that on a pipeline path each algorithm partly contributes to the total time cost and the runtimes are additive. To apply the linear model, we replace the performance metric $\boldsymbol{m}$ with the cost metric $\boldsymbol{\tau}$. The linear cost model parametrized by $\boldsymbol{\beta_\tau}$ can be efficiently updated using Eq. (1). As described in Algorithm 1, $\boldsymbol{\beta_\tau}$ will be updated together with $\boldsymbol{\beta}$ at the end of Phase 2. We note that, in practice, the budget T and the cost $\tau(\cdot)$ can be any quantitative costs of budgeted resources (e.g., money, CPU time), which is a natural generalization of our idea.

With the cost model above, we get the cost-sensitive acquisition function over the best observed result m^+ at a new pipeline path $\boldsymbol{p}$:

$$EIPS(\boldsymbol{p}, \boldsymbol{P}, \boldsymbol{\beta}, \boldsymbol{\beta_\tau}) = \frac{\mathbb{E}[I_{m^+}(\boldsymbol{p})]}{\mathbb{E}[\log \tau(\boldsymbol{p})]} = \frac{\sigma_{\boldsymbol{p}}[u\Phi(u) + \phi(u)]}{\mathbb{E}[\log \tau(\boldsymbol{p})]} \quad (4)$$

where $\quad u = \dfrac{m^+ - \xi - \mu_{\boldsymbol{p}}}{\sigma_{\boldsymbol{p}}}$.

Algorithm 3: Pipeline Caching

input : Data analytic pipeline G; input data $\mathcal{D}$; pipeline configuration $\boldsymbol{p}$ and $\boldsymbol{\lambda_p}$ to be run; available caching budget T_{cache}; current cache pool C

1   $\mathcal{D}^{(1)} \leftarrow \mathcal{D}$
    /* Run pipeline with cache pool   */
2   **for** $k \leftarrow 1, \ldots, K$ **do**
3      $h \leftarrow \mathrm{Hash}(\boldsymbol{p}^{(1)} \ldots \boldsymbol{p}^{(k)}, \boldsymbol{\lambda_p}^{(1)} \ldots \boldsymbol{\lambda_p}^{(k)})$
4   **if** $h \in C$ **then**
5        $\mathcal{D}^{(k+1)} \leftarrow$ cached result from C
6   **else**
7        $\mathcal{D}^{(k+1)} \leftarrow \mathrm{RunAlgorithm}(G, \mathcal{D}^{(k)}, \boldsymbol{p}^{(k)}, \boldsymbol{\lambda_p}^{(k)})$
8        $C \leftarrow C \cup \{\langle h, \mathcal{D}^{(k+1)}\rangle\}$
    /* Clean up cache pool when necessary   */
9   **if** T_{cache} *exhausted* **then**
10   Discard least recently used (LRU) items in C

Here the dependency in $\boldsymbol{\beta}$ and $\boldsymbol{\beta_\tau}$ is captured during computation of $\mu_{\boldsymbol{p}}$, $\sigma_{\boldsymbol{p}}$, and $\tau(\boldsymbol{p})$. We take logarithm of cost $\tau(\cdot)$ to compensate the large variations in the runtime of different algorithms. This acquisition function balances "doing well" and "doing fast" in selecting the next candidate path to run. During the optimization, it will help avoid those costly paths with poor expected improvement. More importantly, at the end of Phase 2 in Algorithm 1, EIPS is responsible to determine the most promising paths, which perform better but cost less, to construct a subgraph for the last phase fine-tuning. For this purpose, we set the exploration parameter ξ to 0 to only select (Line 13 in Algorithm 1).

3.4 Pipeline Caching

During the experiments, we note that many pipeline runs have overlapped algorithms in their paths. Sometimes these algorithms have exactly the same pipeline path and the same hyperparameter settings along the path. This means that we are wasting time on generating the same intermediate output again and again. For example, consider the min-max normalization algorithm in the first pipeline step: this algorithm will be executed many times, especially when it performs well so that Bayesian optimization methods prefer to choose it.

To reduce this overhead, we propose a pipeline caching algorithm, as described in Algorithm 3. When running a pipeline, we check the cache before we run each algorithm. If it turns out to be a cache hit, the result will be immediately returned from cache. Otherwise, we run the algorithm and cache the result. There is a caching pool (e.g., disk space, memory usage) for this algorithm. We use the Least Recently Used (LRU) strategy to clean up the caching pool when budget becomes exhausted.

Caching can significantly reduce the cost of pipeline runs, and accelerates all three phases of FLASH. Algorithms closer to the pipeline input vertex, usually the data preprocessing steps, have higher chance to hit the cache. In fact, when we deal with large datasets on real-world problems, the preprocessing step can be quite time-consuming such that caching can be very efficient.

4. BENCHMARK EXPERIMENTS

In this section, we perform extensive experiments on a number of benchmark datasets to evaluate our algorithm compared to the existing approaches. Then we study the impact of different algorithm choices in each component of FLASH.

Benchmark Datasets.

We conduct experiments on a group of public benchmark datasets on classification task, including MADELON, MNIST, MRBI and CONVEX[3]. These prominent datasets have been widely used to evaluate the effectiveness of Bayesian optimization methods [44, 13, 5]. We follow the original train/test split of all the datasets. Test data will never be used during the optimization: the once and only usage of test data is for offline evaluations to determine the performance of optimized pipelines on unseen test set. In all benchmark experiments, we use percent error rate as the evaluation metric.

Baseline Methods.

As discussed in Section 2, SMAC [22] and TPE [5] are the state-of-the-art algorithms for Bayesian optimization [44, 13, 24], which are used as baselines. Note that Spearmint [41], a Bayesian optimization algorithm based on Gaussian process is not applicable since it does not provide a mechanism to handle the hierarchical space [12]. Besides SMAC and TPE, we also choose random search as a simple baseline for sanity check. Thus, we compare both versions of our method FLASH (with SMAC in Phase 3) and FLASH* (with TPE in Phase 3) against three baselines in the experiments.
Implementation: To avoid possible mistakes in implementing other methods, we choose a general platform for hyperparameter optimization called HPOlib [12], which provides the original implementations of SMAC, TPE, and random search. In order to fairly compare our method with others, we also implement our algorithm on top of HPOlib, and evaluate all the compared methods on this platform. We make the source code of FLASH publicly available at https://github.com/yuyuz/FLASH.

Experimental Settings.

We build a general data analytic pipeline based on scikit-learn [36], a popular used machine learning toolbox in Python. We follow the pipeline design of auto-sklearn [13]. There are four computational steps in our pipeline: 1) feature rescaling, 2) sample balancing, 3) feature preprocessing, and 4) classification model. Each step has various algorithms, and each algorithm has its own hyperparameters. Adjacent steps are fully connected. In total, our data analytic pipeline contains 33 algorithms distributed in four steps, creating 1,456 possible pipeline paths with 102 hyperparameters (30 categorical and 72 continuous), which creates complex high-dimensional and highly conditional search space. More details and statistics of this pipeline are available on our Github project page.

In all experiments, we set a wall-clock time limit of 10 hours for the entire optimization, 15 minutes time limit and 10GB RAM limit for each pipeline run. We perform 10

---

[3]The benchmark datasets are publicly available at http://www.cs.ubc.ca/labs/beta/Projects/autoweka/datasets.

independent optimization runs with each baseline on each benchmark dataset. All experiments were run on Linux machines with Intel Xeon E5-2630 v3 eight-core processors at 2.40GHz with 256GB RAM. Since we ran experiments in parallel, to prevent potential competence in CPU resource, we use the `numactl` utility to bound each independent run in single CPU core.

For our algorithm FLASH, we set both T_{init} and T_{prune} as 30 iterations (equal to the number of algorithms in the pipeline), which can be naturally generalized to other budgeted resources such as wall-clock time or money. We set ξ to 100 in the EIPS function. Note that the performance are not sensitive to the choices of those parameters. Finally, we set the number of pipeline paths r to 10 , which works well in generating a reasonable-size pruned pipeline G'. In benchmark experiments, we compare the performance of FLASH without caching to other methods because the pipelines do not have complex data preprocessing data like many real-world datasets have. We will use caching for real-world experiments later in Section 5.

4.1 Results and Discussions

Table 2 reports the experimental results on benchmark datasets. For each dataset, we report the performance achieved within three different time budgets. As shown in the table, our methods FLASH and FLASH* perform significantly better than other baselines consistently in all settings, in terms of both lower error rate and faster convergence. For example, on the MADELON dataset, our methods reach around 12% test error in only 3 hours, while other baselines are still far from that even after 10 hours.

Performing statistical significance test via bootstrapping, we find that often FLASH and FLASH* tie with each other on these benchmark datasets. For all the methods, the test error is quite consistent with the validation error, showing that the potential overfitting problem is well prevented by using cross validation.

Figure 4 plots the convergence curves of median test error rate along with time for all baseline methods. As shown in the figure, after running about 4 hours, FLASH and FLASH* start to lead others with steep drop of error rate, and then quickly converge on a superior performance.

4.2 Detailed Study of FLASH Components

FLASH has three main components: optimal design for initialization, cost-sensitive model for pipeline pruning, and pipeline caching. To study their individual contributions to the performance gain, we drop out each of the component and compare the performance with original FLASH. Since caching will be used for real-world experiments on large dataset, we describe the analysis of caching component in Section 5. Here we use MRBI dataset for these experiments.

Figure 5(a) shows the difference between using random initialization and optimal design by plotting the performance on initial 30 pipeline runs. The desirable property of optimal design ensures to run reasonable pipeline paths, giving FLASH a head start at the beginning of optimization. While random initialization is not robust enough, especially when the number of pipeline runs is very limited and some algorithms will have no chance to run due to the randomness. Figure 5(b) shows the impact of pipeline pruning in the second phase of FLASH. Dropping out the pruning phase with EIPS, and using SMAC immediately after Phase 1, we see

Table 2: Performance on both 3-fold cross-validation and test data of benchmark datasets. For each method, we perform 10 independent runs of 10 hours each. Results are reported as the median percent error across the 10 runs within different time budgets. Test data is never seen by any optimization method, which is only used for offline evaluations to compute test error rates. Boldface indicates the best result within a block of comparable methods. We underline those results not statistically significantly different from the best according to a 10,000 times bootstrap test with $p = 0.05$.

| Dataset	Budget (hours)	Cross-validation Performance (%)					Test Performance (%)				
		RAND. SEARCH	TPE	SMAC	FLASH	FLASH*	RAND. SEARCH	TPE	SMAC	FLASH	FLASH*
MADELON	3	25.16	18.90	20.25	<u>14.84</u>	**14.04**	19.17	16.15	16.03	<u>12.18</u>	**11.73**
	5	23.60	18.82	19.12	<u>14.31</u>	**14.04**	18.21	15.26	15.38	<u>12.18</u>	**11.60**
	10	20.77	17.28	17.34	<u>13.87</u>	**13.76**	15.58	14.49	13.97	<u>11.49</u>	**11.47**
MNIST	3	7.68	6.78	6.05	**4.93**	<u>5.05</u>	7.75	5.41	6.11	**4.62**	<u>4.84</u>
	5	6.58	5.94	5.83	**4.26**	<u>4.87</u>	7.10	5.41	5.40	**3.94**	<u>4.57</u>
	10	6.58	5.39	5.64	**4.03**	<u>4.46</u>	6.64	5.03	5.23	**3.78**	<u>4.37</u>
MRBI	3	61.80	59.83	62.89	<u>57.43</u>	**57.08**	60.58	59.83	60.58	<u>54.72</u>	**54.28**
	5	58.67	58.61	58.14	**45.11**	54.25	56.42	58.61	55.81	**43.19**	51.65
	10	57.20	53.92	54.60	**41.15**	<u>41.90</u>	54.43	52.01	52.30	**39.13**	<u>39.89</u>
CONVEX	3	28.14	24.70	24.69	**22.63**	<u>23.31</u>	25.04	21.42	21.97	**20.65**	<u>21.04</u>
	5	25.25	23.61	23.30	**21.34**	<u>22.02</u>	23.18	21.37	20.82	**19.56**	<u>19.71</u>
	10	24.51	22.21	23.30	**20.49**	<u>20.62</u>	22.18	20.31	20.82	**18.94**	<u>19.01</u>
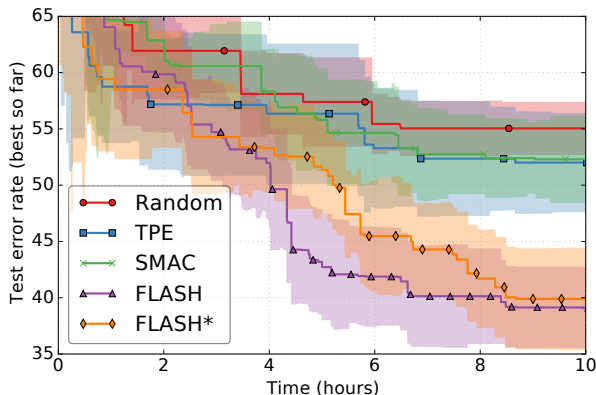
Figure 4: Performance of our methods (**FLASH and FLASH***) and other compared methods on MRBI dataset. We show the median percent error rate on test set along with standard error bars (generated by 10 independent runs) over time.

a major degradation of the performance. The figure clearly shows that in Phase 2 of FLASH, the linear model with EIPS acquisition function is able to efficiently shrink the search space significantly such that SMAC can focus on those algorithms which perform well with little cost. This figure confirms the main idea of this paper that a simple linear model can be more effective in searching high-dimensional and highly conditional hyperparameter space.

5. REAL-WORLD EXPERIMENTS

In this section, to demonstrate a real-world use case, we apply FLASH on a large de-identified medical dataset for classifying drug non-responders. We show how our method can quickly find good classifier for differentiating non-responders vs. responders.

Experimental Settings.

With a collaboration with a pharmaceutical company, we created a balanced cohort of 46,455 patients from a large claim dataset. Patients who have at least 4 times of treatment failure are regarded as drug non-responders (case group). Other patients are responders of the drug (control group). The prediction target is whether a patient belongs to case group or control group. Each patient is associated with a sequence of events, where each event is a tuple of format (`patient-id`, `event-id`, `timestamp`, `value`). Table 3 summarizes the statistics of this clinical dataset, including the count of patient, event, medication, and medication class.

Table 3: Statistics of the medical dataset.

	#Patient	#Event	#Med	#Class
TRAIN CASE	18,581	982,025	434,171	547,854
TRAIN CONTROL	18,582	622,777	286,198	336,579
TEST CASE	4,646	245,776	108,702	137,074
TEST CONTROL	4,646	153,303	70,395	82,908
TOTAL	46,455	2,003,881	899,466	1,104,415

Unlike benchmark experiments, the input to the real-world pipeline is not directly as feature vectors. Given a cohort of patients with their event sequences, like [9] the pipeline for non-responder classification has two more additional steps than the pipeline described in previous benchmark experiments: 1) *Feature construction* to convert patient event sequence data into numerical feature vectors. This step can be quite time-consuming as advanced feature construction techniques like sequential mining [29] and

(a) The impact of optimal design on MRBI dataset

(b) The impact of pipeline pruning on MRBI dataset

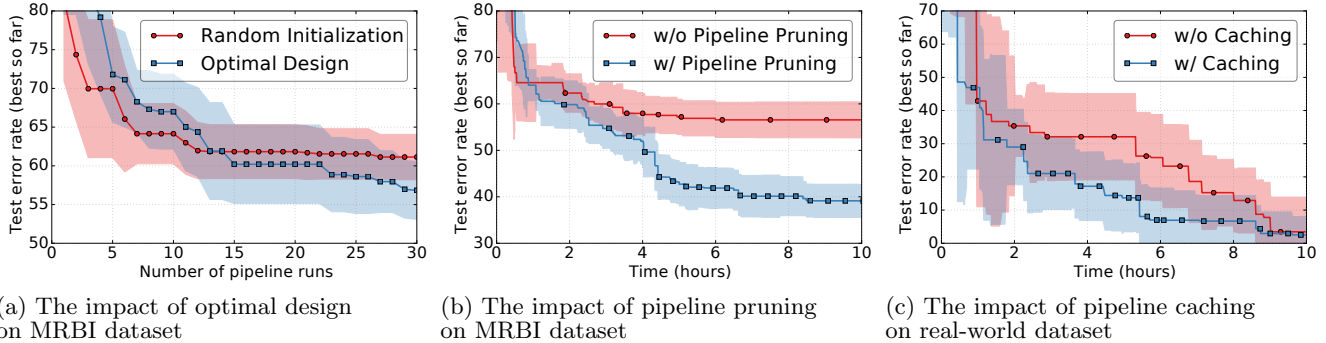(c) The impact of pipeline caching on real-world dataset

Figure 5: Component analysis experiments. (a) Optimal design makes the initialization phase more robust. (b) Pipeline pruning in the second phase of FLASH is the key to its superior performance. (c) Performance of FLASH without caching and the original FLASH with caching on real-world dataset. In all figures, we show the median error rate on test set along with standard error bars (generated by 10 independent runs). Note that (a) and (b) are plotted with different x-axes; (c) is on a different dataset as (a) and (b).

Table 4: Performance of real-world dataset. Results are reported using the same settings as Table 2.

Budget (hours)	Test Performance (%)				
	RAND. SEARCH	TPE	SMAC	FLASH	FLASH*
3	30.32	27.03	35.40	**21.02**	23.28
5	16.66	19.09	33.22	**14.40**	19.86
10	11.75	4.86	21.03	**2.51**	3.44

tensor factorization [19] can be expensive to compute. On this medical dataset, we consider two kinds of parameters in this step: i) frequency threshold to remove rare events (frequency ranging from 2 to 5) ii) various aggregation functions (including binary, count, sum and average) to aggregate multiple occurrence of events into features. The output of this step will be a feature matrix and corresponding classification targets; 2) *Densify* the feature matrix from above feature construction step if necessary. Features of this real-world dataset can be quite sparse. We by default use sparse representation to save space and accelerate computation in some algorithms. Unfortunately, not all algorithm implementations in scikit-learn accept sparse features. A decision has to be made here: either sparse matrix for faster result or dense matrix for broader algorithm choices in later steps.

We run the experiments on same machine, use same parameter setting and same budget as benchmark experiments. We compare our method with the same baselines as benchmark experiments and we continue using error rate as metric. Our algorithm has built-in caching mechanism and we will use that. For this real-world dataset, we first compare with baselines with cache enabled. Then we analyze the contribution of caching.

5.1 Results and Discussions

Table 4 shows the performance of our methods compared to baselines when caching is enabled. Due to lack of space we only report the test performance. All cases FLASH and FLASH* significantly outperform all the baselines.

Figure 5(c) shows the performance of FLASH without caching and original FLASH with caching on the real-world medical dataset. With caching, more pipeline paths can be evaluated within given period of time and our EIPS-based path selection leverages caching to select paths with high performance that run fast. As a result, we can see FLASH with caching converges much faster. For example, with caching we can get low test error within 6 hours.

6. CONCLUSIONS

In this work, we propose a two-layer Bayesian optimization algorithm named FLASH, which enables highly efficient optimization of complex data analytic pipelines. We showed that all components of FLASH complement each other: 1) our optimal design strategy ensures better initialization, giving a head start to the optimization procedure; 2) the cost-sensitive model takes advantage of this head start, and significantly improves the performance by pruning inefficient pipeline paths; 3) the pipeline caching reduces the cost during the entire optimization, which provides a global acceleration of our algorithm. We demonstrate that our method significantly outperforms previous state-of-the-art approaches in both benchmark and real-world experiments.

7. ACKNOWLEDGMENTS

References

[1] A. Atkinson and A. Donev. Optimum experimental designs. 1992.

[2] T. Bäck. Evolutionary algorithms in theory and practice. 1996.

[3] J. Bergstra and Y. Bengio. Random search for hyper-parameter optimization. *JMLR*, 13(1), 2012.

[4] J. Bergstra, D. Yamins, and D. Cox. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In *ICML*, 2013.

[5] J. S. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl. Algorithms for hyper-parameter optimization. In *NIPS*, 2011.

[6] S. Boyd and L. Vandenberghe. *Convex optimization.* Cambridge university press, 2004.

[7] E. Brochu, V. M. Cora, and N. De Freitas. A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *arXiv preprint arXiv:1012.2599*, 2010.

[8] G. Calinescu, C. Chekuri, M. Pál, and J. Vondrák. Maximizing a monotone submodular function subject to a matroid constraint. *SIAM Journal on Computing*, 40(6), 2011.

[9] R. Chen, H. Su, Y. Zhen, M. Khalilia, D. Hirsch, M. Thompson, T. Davis, Y. Peng, S. Lin, J. Tejedor-Sojo, E. Searles, and J. Sun. Cloud-based predictive modeling system and its application to asthma readmission prediction. In *AMIA*. AMIA, 2015.

[10] S. J. Coakes and L. Steed. *SPSS: Analysis without anguish using SPSS version 14.0 for Windows.* John Wiley & Sons, Inc., 2009.

[11] D. Donoho. 50 years of Data Science. Technical report, University of California Berkeley, 2015.

[12] K. Eggensperger, M. Feurer, F. Hutter, J. Bergstra, J. Snoek, H. Hoos, and K. Leyton-Brown. Towards an empirical foundation for assessing bayesian optimization of hyperparameters. In *NIPS workshop on Bayesian Optimization in Theory and Practice*, 2013.

[13] M. Feurer, A. Klein, K. Eggensperger, J. Springenberg, M. Blum, and F. Hutter. Efficient and robust automated machine learning. In *NIPS*, 2015.

[14] M. Feurer, T. Springenberg, and F. Hutter. Initializing bayesian hyperparameter optimization via meta-learning. In *AAAI*, 2015.

[15] P. Flaherty, A. Arkin, and M. I. Jordan. Robust design of biological experiments. In *NIPS*, 2005.

[16] C. J. Flynn, C. M. Hurvich, and J. S. Simonoff. Efficiency for regularization parameter selection in penalized likelihood estimation of misspecified models. *JASA*, 108(503), 2013.

[17] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1), 2009.

[18] X. He. Laplacian regularized d-optimal design for active learning and its application to image retrieval. *Image Processing, IEEE Transactions on*, 19(1), 2010.

[19] J. C. Ho, J. Ghosh, and J. Sun. Marble: high-throughput phenotyping from electronic health records via sparse nonnegative tensor factorization. In *KDD*, 2014.

[20] M. D. Hoffman, E. Brochu, and N. de Freitas. Portfolio allocation for bayesian optimization. In *UAI*. Citeseer, 2011.

[21] M. D. Hoffman, B. Shahriari, and N. de Freitas. On correlation and budget constraints in model-based bandit optimization with application to automatic machine learning. In *AISTATS*, 2014.

[22] F. Hutter, H. H. Hoos, and K. Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *Learning and Intelligent Optimization*. Springer, 2011.

[23] M. Jaggi. Revisiting frank-wolfe: Projection-free sparse convex optimization. In *ICML*, 2013.

[24] B. Komer, J. Bergstra, and C. Eliasmith. Hyperoptsklearn: Automatic hyperparameter configuration for scikitlearn. In *ICML workshop on AutoML*, 2014.

[25] A. Krause and C. Guestrin. Submodularity and its applications in optimized information gathering. *TIST*, 2(4), 2011.

[26] A. Kumar, R. McCann, J. Naughton, and J. M. Patel. Model selection management systems: The next frontier of advanced analytics. *ACM SIGMOD Record*, 2015.

[27] D. J. Lizotte. *Practical bayesian optimization.* University of Alberta, 2008.

[28] J. Lv and J. S. Liu. Model selection principles in misspecified models. *JRSS-B*, 76(1), 2014.

[29] K. Malhotra, T. Hobson, S. Valkova, L. Pullum, and A. Ramanathan. Sequential pattern mining of electronic healthcare reimbursement claims: Experiences and challenges in uncovering how patients are treated by physicians. In *Big Data*, Oct 2015.

[30] X. Meng, J. Bradley, E. Sparks, and S. Venkataraman. Ml pipelines: a new high-level api for mllib, 2015.

[31] I. Mierswa, M. Wurst, R. Klinkenberg, M. Scholz, and T. Euler. Yale: Rapid prototyping for complex data mining tasks. In *KDD*, 2006.

[32] J. Mockus, V. Tiesis, and A. Zilinskas. The application of bayesian methods for seeking the extremum. *Towards Global Optimization*, 2(117-129), 1978.

[33] R. Munos. Optimistic optimization of deterministic functions without the knowledge of its smoothness. In *Advances in neural information processing systems*, 2011.

[34] Y. Nesterov. Gradient methods for minimizing composite objective function, 2007.

[35] K. Ng, A. Ghoting, S. R. Steinhubl, W. F. Stewart, B. Malin, and J. Sun. Paramo: A parallel predictive modeling platform for healthcare analytic research using electronic health records. *Journal of biomedical informatics*, 48, 2014.

[36] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al. Scikit-learn: Machine learning in python. *JMLR*, 12, 2011.

[37] T. Robertazzi and S. Schwartz. An accelerated sequential algorithm for producing d-optimal designs. *SIAM Journal on Scientific and Statistical Computing*, 10(2), 1989.

[38] B. Settles. Active Learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 6(1), jun 2012.

[39] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, and N. de Freitas. Taking the human out of the loop: A review of bayesian optimization. *Proceedings of the IEEE*, 104(1):148–175, 2016.

[40] M. Shamaiah, S. Banerjee, and H. Vikalo. Greedy sensor selection: Leveraging submodularity. In *CDC*, 2010.

[41] J. Snoek, H. Larochelle, and R. P. Adams. Practical bayesian optimization of machine learning algorithms. In *NIPS*, 2012.

[42] J. Snoek, O. Rippel, K. Swersky, R. Kiros, N. Satish, N. Sundaram, M. Patwary, M. Ali, R. P. Adams, et al. Scalable bayesian optimization using deep neural networks. *arXiv preprint arXiv:1502.05700*, 2015.

[43] N. Srinivas, A. Krause, S. M. Kakade, and M. Seeger. Gaussian process optimization in the bandit setting: No regret and experimental design. *arXiv preprint arXiv:0912.3995*, 2009.

[44] C. Thornton, F. Hutter, H. H. Hoos, and K. Leyton-Brown. Auto-weka: Combined selection and hyperparameter optimization of classification algorithms. In *KDD*, 2013.

[45] J. Villemonteix, E. Vazquez, and E. Walter. An informational approach to the global optimization of expensive-to-evaluate functions. *Journal of Global Optimization*, 44(4), 2009.

[46] Z. Wang, M. Zoghi, F. Hutter, D. Matheson, and N. De Freitas. Bayesian optimization in high dimensions via random embeddings. In *IJCAI*. Citeseer, 2013.

[47] L. Wasserman. *All of nonparametric statistics.* Springer Science & Business Media, 2006.

[48] H. White. Maximum likelihood estimation of misspecified models. *Econometrica*, 1982.