2017 International Conference on Data Science and Advanced Analytics

# Sample, Estimate, Tune: Scaling Bayesian Auto-tuning of Data Science Pipelines

Alec Anderson
MIT, LIDS
Cambridge, MA
alecand@mit.edu

Sebastien Dubois
Stanford University
Palo Alto, CA
sdubois@alumni.stanford.edu

Alfredo Cuesta-Infante
Universidad Ray Juan Carlos
Madrid, Spain
alfredo.cuesta@urjc.es

Kalyan Veeramachaneni
MIT, LIDS
Cambridge, MA
kalyan@csail.mit.edu

***Abstract*—In this paper, we describe a system for sequential hyperparameter optimization that scales to work with complex pipelines and large datasets. Currently, the state-of-the-art in hyperparameter optimization improves on randomized and grid search by using sequential Bayesian optimization to explore the space of hyperparameters in a more informed way. These methods, however, are not scalable, as the entire data science pipeline still must be evaluated on all the data. By designing a sub sampling based approach to estimate pipeline performance, along with a distributed evaluation system, we provide a scalable solution, which we illustrate using complex image and text data pipelines. For three pipelines, we show that we are able to gain similar performance improvements, but by computing on substantially less data.**

## I. Introduction

When given a problem and data, a data scientist puts together an end-to-end pipeline that performs a series of transformations on the data and then learns a model. Over the course of this process, the data scientist makes a number of decisions regarding the hyperparameters associated with each of the transformation blocks. These decisions are made at the pre-processing, feature extraction, feature transformation and modeling stages.

According to our observations, data scientists use three strategies to make decisions about hyperparameters: (1) they use past experiences and domain knowledge, (2) they choose in an ad-hoc manner (usually setting them to default values), assuming that they can regain any lost accuracy by fine-tuning the later stages of the pipeline (essentially the machine learning model), or (3) they choose quickly in pursuit of an end-to-end solution that will enable them to assess whether the data will solve the problem at hand, intending to revisit this initial choice later. When multiple members are involved in the process, a number of intermediate representations of data are stored and shared in order to enable team members to simultaneously work on different parts of the pipeline. This makes it harder to revisit the choices made at earlier stages.

These strategies ultimately result in sub-optimal solutions, as decisions made at the earlier stages of the pipeline can have a huge impact on performance. Because dependencies between the hyperparameters chosen and the pipeline's overall performance can be hard to understand, data scientists often do not build the most effective possible pipeline. Then, when the resulting end-to-end solution does not perform well, it is unclear which part of the pipeline needs to be tuned or refined.

In this paper, our goal is to develop an auto-tuning system that enables data scientists to efficiently explore numerous possibilities for hyperparameters, and then confidently make a choice, backed with tests and metrics based on the execution of the pipeline. With regards to efficiently searching the space, we are encouraged by the progress made by the Bayesian hyperparameter optimization community [1]–[3] [4]. These methods provide a way of modeling the search space from a few samples, and sequentially attempting to find better points given a metric of choice, as computed by a "scoring function." Their goal is to reduce the number of "scoring function" evaluations.

However, when optimizing an entire data science pipeline, the scoring function depends on the execution of a complex set of data transformations that make up the earlier stages of the pipeline. Hence, even a single execution can require substantial computing power. Our aim with this paper is to address the challenge of computing the scoring function for an arbitrary data science pipeline. We utilize Bayesian hyperparameter optimization, but address the data and computing requirements by estimating the score using a subsampling method based on Bag of Little Bootstraps (BLB). This method can be executed in an extremely parallel fashion, as the pipeline can be simultaneously executed on different subsamples. We develop an Apache Spark-based implementation for using Bag of Little Bootstraps. We demonstrate that even after sampling down the data, a user can estimate the scoring function of the pipeline with reasonable accuracy, and use it to direct the search.

The rest of the paper is organized as follows. Section II presents the overview of our system. We describe how users can construct arbitrary pipelines in our system in Section III. In Section IV, we present the Bag of Little Bootstraps technique. Section V presents our copula-based Bayesian hyperparameter optimization process. We present multiple pipelines that we designed, as well as results, in Section VI.

## II. Overview

Consider a function $g(\mathbf{h})$, where the function's output is a measure of how well a data science pipeline is performing, as measured by a user-specified scoring function. The data

978-1-5090-5004-8/17 $31.00 © 2017 IEEE
DOI 10.1109/DSAA.2017.82

science pipeline has certain hyperparameters, **h**. [1] The goal of hyperparameter optimization is to find the settings of the vector **h** that maximize (or minimize) the scoring function $g(\mathbf{h})$. Subject to a specified range $R$ for **h**:

$$\begin{aligned} &\underset{\mathbf{h}}{\operatorname{argmin}}\ g(\mathbf{h}) \\ &subject\ to\ \ \mathbf{h} \in R \end{aligned} \tag{1}$$

In our system, a data scientist who wishes to tune a pipeline can do so by executing the following steps:

- define an arbitrary pipeline in our abstraction, exposing all hyperparameters, and selecting what parts of the pipeline s/he wants to tune (described in Section III),
- write an evaluation function for the pipeline,
- specify the settings for Bag of Little Bootstraps, the tuning algorithm and the cluster size. These settings are detailed in section IV, and section V.
- launch a cluster using our API, and
- run the `DeepMine` function specifying the `pipeline`, the `cluster` location, and other required `configs`.

The system then:

- uses our novel open-source implementation of Bayesian hyperparameter optimization with Gaussian Copula Processes to propose hyperparameters for the pipeline,
- evaluates the pipeline using subsampling, and
- responds iteratively, each time proposing a new set of hyperparameters for the `pipeline` and ultimately giving the best possible `pipeline` as output.

## III. Composing arbitrary pipelines

Our goal in this paper is to enable tuning of the entire pipeline, which includes a series of data transformations that can be sorted into three types: preprocessing, feature extraction, and feature transformation. This process presents several challenges:

- **Too many possibilities**: When considering data science pipelines, numerous possibilities exist for early-stage data transformations. Transformations can be specific to a domain or a problem, and/or specifically developed to mitigate issues in data collection. This variety makes it impossible to develop a system around a fixed set of transformations *a priori*. These steps also accept a variety of inputs, such as data in non-matrix formats, that many existing tools cannot incorporate.
- **Unstructured processes**: Unlike software libraries for machine learning algorithms, software for these transformations is written by domain experts, who do not generally write it in a structured and uniform way. This lack of structure and uniformity inhibits development of a general purpose system like ours.

In order to address these two issues, we resort to familiar abstractions developed for machine learning methods, and evaluate whether they might be extended for pipelines in general.

**Abstractions in scikit-learn** Scikit-learn, a widely adopted machine learning software library, offers a powerful *fit-transform* abstraction. It has two types of objects: *transformers* and *estimators*. As their name suggests, transformers transform the data, and they must have *fit* and *transform* methods (the *fit* method allows them to learn parameters to be used for *transform*). Principal Component Analysis (PCA) is an example of a transformer. The *fit* method in PCA identifies the principal vectors in the data, which are then used to transform the feature matrix. *Estimators fit* a model and use a fitted model to predict, and hence must have *fit* and *predict* methods. Any machine learning classifier is an example of an *estimator*. This has provided a uniform way to specify disparate methods in machine learning.

`Scikit-learn` also extends these methods to pipelines composed of multiple steps chained together. Within scikit-learn, a user can do the following:

- **Create arbitrary pipelines**: A user can chain together multiple transformers, provided they always end with an estimator method. Consider an example in which a user has a pipeline consisting of steps `A`, `B`, and `C`, where `A` and `B` are `scikit-learn` transformers, and `C` is a `scikit-learn` estimator. First, the pipeline object is constructed by specifying a list of the variables for the pipeline steps. These variables denote objects on which the fit and transform methods can be called.

  `pipeline` = Pipeline(`[A, B,C]`)

  Next, the pipeline hyperparameters are mapped to individual steps by following a strict convention:

  `pipeline.set_params` = *{A__a1: $v_{a1}$, B__b1: $v_{b1}$, C__c1: $v_{c1}$}*

  where $a1$ is a hyperparameter for step A and $v_{a1}$ is the corresponding name specified for the same step in the global dictionary of hyperparameters.
- **Integrated training and validation**: One of the advantages of setting up the pipeline object and mapping the hyperparameters as shown above is that users can now call the same "fit" and "predict" functions that they would have used on individual steps to execute the entire pipeline.
  - Calling

    `pipeline.fit(X_tr, y_tr)`

    will transform the data sequentially in the order the steps are specified in the `pipeline` leading up to the `estimator`. It will apply `fit_transform` for each `transformer`, and finally "fit" the estimator on the resulting transformed data.
  - The fitted `pipeline` can then be evaluated on the validation data, and used to make predictions as follows:

    `score` = `pipeline.score(X_v, y_v)`
    `predictions` = `pipeline.predict(X_v)`

[1] *Hyperparameters* are distinct from the parameters learned during the model training process.

In both cases above, $X_v$ is transformed with the pipeline's intermediate steps, this time using only the `transform` method, and scored using the fitted estimator.

**Integrating custom functions**: One potential problem with the pipeline abstraction is its strict structure, as users will have to alter their program to implement *fit* and *transform*. However, scikit-learn provides the `FunctionTransformer`, `TransformerMixin`, and `BaseEstimator` classes to streamline this conversion process.

`FunctionTransformer` objects take an arbitrary transforming function as an input and output a valid scikit-learn transformer, which implements the fit and transform methods necessary for the pipeline object.

For transformers that also include a fitting process, the `TransformerMixin` class can be inherited, and *fit* and *transform* methods can be implemented as desired.

The `BaseEstimator` class allows for the construction of arbitrary estimators, requiring users to implement the fit and score methods.

**Advantages of this abstraction**: This abstraction greatly simplifies the data science pipeline construction process, requiring only the implementation of the *set_pipeline* function. This construction has many benefits for data scientists, including

- standardizing the pipeline construction process, giving an intuitive template and adding discipline to the often unstructured process of pipeline construction,
- making pipelines *modular*, allowing steps to be interchanged arbitrarily as long as each step provides a valid output for the following step,
- accelerating training and testing by encapsulating the process into only two function calls,
- providing a framework flexible enough to handle arbitrary transformers and estimators, and
- enabling code-sharing through the use of a simple, modular template.

At the beginning of this project, we considered developing our own set of abstractions, but recognized that existing ones provided most of the functionality the system needed. There are, however, some intricacies involved in mapping the hyperparameter dictionary to each step (for example, underscores may be used in different places), in changing the mapping process when a non-scikit-learn function is involved, and in how an external transformer can be integrated when it needs both fit and transform functions. In our framework, we have started to develop a higher-level `API` that enables easier integration of domain experts' code into the framework.

**Conditional Pipeline Evaluation** A potential method for achieving large-scale speed improvements is to conditionally evaluate pipeline steps according to which steps' hyperparameters have changed. For example, consider a three-step pipeline with transform steps $A$ and $B$ followed by estimator $C$, in which each step has one hyperparameter. If only $C$'s hyperparameter changes between two hyperparameter set evaluations, then we do not need to recompute steps $A$ and $B$. A possible system would then store the outputs of the intermediate steps for the hyperparameter set from previous iteration and use those precomputed outputs whenever another hyperparameter set overlaps with previously computed steps.

Additionally, users can specify steps of the pipeline whose hyperparameters they do not want to tune. In the $A - B - C$ example above, if they do not want to tune the hyperparameters of step $A$, then we simply precompute the output of the transform $A$. Using the pipeline abstraction from above, we define an auxiliary function $precompute\_transforms$, which constructs a pipeline of only transforms, calculated before hyperparameter optimization.

**Algorithm 1:** General pipeline construction

```
1 set_pipeline
    ($X\_tr, y\_tr, X\_v, y\_v$, hyperparam_dict);
  Input  : Data provided in a train-validation split
           ($X\_tr, y\_tr, X\_v, y\_v$) and a dictionary of
           hyperparameters to evaluate
  Output: pipeline object
2 step$_1$ = Transformer$_1$()
3 ...
4 step$_n$ = Estimator()
5 pipeline = Pipeline([step$_1$,step$_2$,....step$_n$])
6 pipeline.hyperparams =
    {step$_1$_hyperparam$_1$: hyperparam_dict[pname$_1$], ... ,
    step$_1$_hyperparam$_k$: hyperparam_dict[pname$_k$], ... ,
    step$_n$_hyperparam$_n$: hyperparam_dict[pname$_n$]}
7 return pipeline
```

## IV. Sampling based estimation

One of the main roadblocks getting in the way of tuning of an entire data processing pipeline is the computational time such tuning incurs, and the amount of data that would need to be maintained in memory during a single iteration of the tuning run. This part of the pipeline scales with the raw data. For image problems, it is now common to deal with several thousands of images, representing hundreds of gigabytes of memory, each of which must be processed to extract features like SIFT descriptors [5].

In our system, we propose using the *Bag of Little Bootstraps* (BLB), a subsampling method that allows for the evaluation of an arbitrary statistic of the data over multiple subsamples. Thus BLB can potentially allow for the evaluation of the scoring function on the pipeline in significantly reduced time. By executing the computation in this algorithm in parallel using Apache Spark, the theoretical speed gains become even greater, significantly lessening the computation time involved in hyperparameter optimization and making it possible to automatically tune complex pipelines on large datasets.

### A. Bag of Little Bootstraps

The Bag of Little Bootstraps, proposed in [6], enables the calculation of statistics for data with the same statistical guarantees as the traditional bootstrap, allowing for scalable assessments of the quality of estimators. Bootstrap-based quantities are typically computed by repeatedly applying a given estimator to resamples of the original dataset. Because the sizes of these resamples are of the same order as the original data (typically around 63% of the data points), the bootstrap cannot be applied to large datasets in practice. The BLB method alleviates this problem by introducing an additional level of subsampling, constructing "bags" within which bootstraps are created.

Consider a dataset with $n$ elements. In the BLB algorithm, the data is first subsampled into a "bag" of size $b \in [n^{1/2}, n]$ by sampling the original data without replacement. Within each of these bags, a bootstrap sample of size $n$ is created using sampling with replacement.

Then, the estimator in question is computed on this bootstrap sample of the data within each of the bags. For each bag, the value of the estimator is the average of the values on each of the bootstrap samples. Then, the overall estimator value given by the algorithm is the average of the estimator values generated for each bag.

**Algorithm 2:** Bag of Little Bootstraps

```
Input  : Data: $n$ (m-dimensional) data points
         $X_{1...n} = \{x^1_{1...m} \cdots x^n_{1...m}\}$;
         $b$: Bag size; $s$ Number of bags;
         $r$: Number of Multinomial samples in each bag
         $E$: Estimator in question; $y$: statistic of interest.
Output: Estimate of $y$
1: for $j \to 1$ to $s$ do
2:     Randomly sample $b$ data points without replacement
       forming a bag
       $X^j_{1...b} \sim \texttt{random}(X_{1...n})$
3:     for $k \to 1$ to $r$ do
4:         Sample the counts $(n_1,...,n_b) \sim$
           $\texttt{Multinomial}(n,\mathbf{1}_b/b)$
           Estimate $y$ on the bootstrap sample using $X^j_{1...b}$
           and the counts as weights
           $\hat{y}_k = E(X^j_{1...b}, n_1,...,n_b)$
5:     end for
       $\hat{y}_j = r^{-1} \sum_{k=1}^{r} estimate_k$
6: end for
return $\hat{y} = s^{-1} \sum_{j=1}^{s} \hat{y}_j$
```

The computational benefit of this algorithm is only realized because each bootstrap sample consists of only $b$ distinct points. Since $b << n$, instead of creating repeated versions of data points, a multinomial sample of counts is generated $n_1 \ldots n_b$, s.t. $\sum_{i=1}^{b} n_i = n$. These counts represent how many times a particular data point in the "bag" would be repeated, if we were to sample with replacement from the data points in the bag.

First, this implies that we are essentially operating on $b$ data points. Second, many estimators can work directly with a weighted data representation. That is they can estimate the statistic by taking the $b$ unique data points and the counts $n_1 \ldots n_b$ associated with each data point. This is presented in algorithm 2.

The computational complexity of the estimator, then, scales with $b$ rather than $n$ for both time and disk space used. The benefits from this reduction in data are substantial, as for an original dataset with $n = 500,000$ data points and a conventional bag size of $b = n^{0.6}$, estimators can be computed by operating on only 2,627 data points each. While we will not go into the derivation of the theoretical guarantees here, the statistical properties of asymptotic consistency and higher-order correctness are identical to those of the bootstrap, allowing for the computation of a variety of functions. The BLB method is also quite amenable to parallelization, as each bag can be computed in parallel with no modification to the original algorithm.

## *B. Application to data science pipelines*

Our system uses an algorithm based on the Bag of Little Bootstraps to construct samples for training and validating data science pipelines. We incorporate cross-validation in BLB using a train-validation split approach based on that proposed in [7]. In this approach, training and validation bags are both constructed: For the parameter $s$ in the original BLB, $2 \cdot s$ bags are created. Then, bootstraps are taken as before within the training bag, and the pipeline in question is trained on each of these weighted, size $b$ bootstraps. The weights drawn from the Multinomial distribution bias the training of the pipeline. These trained models are then scored on the validation bag, and the scores from the validation bag are averaged to compute a final estimate for the training bag. This is repeated for every training bag and scores are averaged across all training bags. The detailed process can be found in Algorithm 3.

As mentioned before, the computational benefits are achieved because transformers and estimators in the pipelines can work with weighted data representations. For custom transformers, we require users to develop a version that can accept weighted data points.

**Complexity analysis**: Using this BLB train/validation split provides considerable asymptotic computational benefits, as machine learning algorithms now scale with $b$ rather than $n$. For example, Support Vector Machine methods now have asymptotic complexity $O(b^2)$ rather than $O(n^2)$, a considerable improvement when $b \in [n^{0.5}, n]$. This complexity is achieved by using machine learning algorithms that can use the data given as weighted samples, and implementations with this capability can be easily found in open-source libraries like scikit-learn.

BLB settings is a major consideration in this algorithm, as changing the number of bags $s$ and the number of bootstraps in each bag $r$ can dramatically change the running time. In Algorithm 3, we train the estimator $s \cdot r$ times, and test it $s \cdot r$ times. If the original algorithm has time complexity $O(n)$ for training or testing on $n$ data points, the rough complexity of this process is $2 \cdot s \cdot r \cdot b$ for bags of size $b$. We make

**Algorithm 3:** Bag of Little Bootstraps for Data Science Pipelines

```
Input : Data: $n$ (m-dimensional) data points
        $X_{1...n} = \{x^1_{1...m} \cdots x^n_{1...m}\}$;
        $b$: Bag size; $s$ Number of bags;
        $r$: Number of Multinomial samples in each bag
        $pipeline$: pipeline in question; $y$: statistic of
interest.
Output: Estimate of $y$, $\hat{y}$
1: for $j \rightarrow 1$ to $s$ do
2:     Randomly sample $b$ data points without replacement
       to form a training bag
       $X^{j(t)}_{1...b} \sim$ random($X_{1...n}$)
3:     Randomly sample $b$ data points without replacement
       to form a validation bag
       $X^{j(v)}_{1...b} \sim$ random($X_{1...n}$)
4:     for $k \rightarrow 1$ to $r$ do
5:         Sample $(n_1,...,n_b) \sim$ Multinomial($n$,$\mathbf{1}_b/b$)
6:         pipeline = pipeline.fit($X^{j(t)}_{1...b}$, $n_1,...,n_b$)
7:         $\hat{y}_k$ = pipeline.score($X^{j(v)}_{1...b}$)
8:     end for
       $\hat{y}_j$ = $r^{-1} \sum_{k=1}^{r} \hat{y}_k$
9: end for
return $\hat{y} = s^{-1} \sum_{j=1}^{s} \hat{y}_j$
```

these hyperparameter considerations empirically, as shown in section VI.

### C. Implementation

The BLB algorithm for machine learning is quite parallelizable, as bags can be computed upon in parallel. Deep Mining uses Apache Spark on EC2 clusters, as well as the pathos multiprocessing method, to parallelize this computation. We will not go into the the details of our use of Apache Spark, but in essence, we use a map-reduce structure in which we compute estimator scores for a given bootstrap and bag, then aggregate these scores by averaging across all bootstraps and bags.

To run the BLB algorithm in Deep Mining, the user need specify only the value of the BLB settings: the size of the bags $b$, the number of sampled bags $s$, and the number of bootstraps $r$. In section VI, we have found $b = n^{0.6}$, $s = 8$, and $r = 20$ to be sufficient for our data science pipelines.

### D. Extensions

The map-reduce implementation of the BLB algorithm in our system also allows for different reduction functions, giving users the flexibility to calculate other statistical quantities relating to the estimator scores in question. For example, confidence intervals and estimator variances can be constructed by using the empirical multinomial sample scores [8].

In any distributed computation system, application-specific tuning can significantly increase computational performance. Fine-tuning of Apache Spark applications by altering configuration values, incorporating file systems such as HDFS, and altering the level of parallelism can significantly speed hyperparameter optimization in the future. Just-in-time compilers such as SEJITS [9], [10] can also speed computation by porting high-level Python to lower-level languages such as C, and these systems could be applicable to Deep Mining.

## V. Hyperparameter Tuning

In machine learning, the Gaussian Process is a particular prior to perform Bayesian learning. Specifically, if the goal is to predict the values of a function $g : \mathcal{H} \rightarrow \mathcal{Y} \subset \mathbb{R}$, this means that we model $g$ such that for any finite set of $N$ points $\{h_i\}_{i=1}^{N}$ in $\mathcal{H}$, $\{g(h_i)\}_{i=1}^{N}$ has a multivariate Gaussian distribution on $\mathbb{R}^N$, determined by the mean function $m : \mathcal{H} \rightarrow \mathcal{Y}$ and the covariance function $k : \mathcal{H} \times \mathcal{H} \rightarrow \mathbb{R}$ :

$$m(h) = \mathbb{E}[g(h)] \tag{2}$$

$$k(h, h') = \mathbb{E}[(g(h) - m(h))(g(h') - m(h'))]. \tag{3}$$

Note that these functions depend only on $g(h)$.

Usually, the mean function is fixed as the average of the known $g$ values on the training data, so we will take it to be zero (assuming the data are centered). The covariance function is parametrized, and its parameters are learned using data via Maximum Likelihood Estimation [11].

In the Bayesian framework, predictions are simply made by looking at the posterior distribution, which is the prior conditioned on the training data $(\bar{h}_t, \bar{g}_t)$. Thus the predicted value $g_*$ of the function $g$ at point $h_*$ would be:

$$g_* = \mathbb{E}(\ g(h_*) \mid \bar{h}_t,\ \bar{g}_t\ ), \tag{4}$$

where :

$$g(h_*) \sim \mathcal{N}(\ k(h_*, \bar{h}_t)K_t^{-1}\bar{g}_t, \\ k(h_*, h_*) - k(h_*, \bar{h}_t)K_t^{-1}k(\bar{h}_t, h_*)). \tag{5}$$

and $K_t = k(\bar{h}_t, \bar{h}_t)$ is the square covariance matrix of the training data.

### A. Gaussian Copula Process (GCP)

The Gaussian Copula Process, as introduced in [12], is a prior based on a GP that can more precisely model the multivariate distribution of $\{g(h_i)\}_{i=1}^{N}$. This is done through a mapping $\Psi : \mathcal{Y} \rightarrow \mathcal{Z}$ that transforms the output of $g$ into a new variable $z$. We define a new function $w$, $w : \mathcal{H} \rightarrow \mathcal{Z}$, a combination of $g$ and $\Psi$ given by $w(h) = \Psi(g(h))$; and we model $w$ with a GP.

By doing this, we actually change the assumed Gaussian marginal distribution of each $g(h)$ into a more complex one. More specifically, the Gaussian prior on $w(h)$ yields the prior for $g(h)$ given by the following cumulative distribution function:

$$F(y) = \Phi(\Psi(y)), \tag{6}$$

where $F(y) = \mathbb{P}(g(h) \leq y)$ and $\Phi$ is the standard univariate Gaussian cumulative distribution function. From this point on, we will use $y$ for $g(h)$ and $z$ for $w(h)$ and $z = \Psi(y)$, for notational convenience.

So far in the literature, *e.g.* [12], [13], a parametric mapping is learned so that $z$ is best modeled by a Gaussian Process. In [12], the authors propose to parametrize $\Psi^{-1}$ by $\{a_j, b_j, c_j\}$ such that:

$$y = \Psi^{-1}(z; \{a_j, b_j, c_j\}_{j=1}^{K}) = \sum_{j=1}^{K} a_j log\big(e^{b_j(z+c_j)} + 1\big), \tag{7}$$

with $a_j, b_j > 0$. They are interested in predicting the values of a positive function $g$. So in the general case, we can add another variable $m$:

$$y = \Psi^{-1}(z; \{a_j, b_j, c_j\}_{j=1}^{K}, m) = \sum_{j=1}^{K} a_j log\big(e^{b_j(z+c_j)} + 1\big) - m,$$

where $m, a_j, b_j > 0$. Thus for $K = 1$ we have :

$$z = \Psi(y) = \frac{log(e^{\frac{y+m}{a}} - 1)}{b} - c; \quad m, a, b > 0. \tag{8}$$

However, we have found that such a mapping was unstable: for many trials on the same dataset, different mappings were learned. Moreover, the induced univariate distribution for $y$ was almost Gaussian most of the time, and the parametric mapping could not offer great flexibility. We indeed see in e.q. (7) that for $K = 1$, if $bz \geq bc, 1$, then $\Psi^{-1}(z; a, b, c) \sim abz$, *ie.* the mapping is linear and the GCP is actually a GP.

Thus we introduce a novel approach where a marginal distribution is learned from the observed data through kernel density estimation [14] of $F$. Then the mapping $\Psi$ is numerically computed from equation (6), so that the observations of the training data $w(h_{t,i})$ have a Gaussian distribution:

$$z = \Psi(y) = \Phi^{-1}(F_{est}(y)). \tag{9}$$

As the mapping function is learned in a non-parametric manner, we call this novel approach *Non-Parametric Gaussian Copula Process* (nGCP).

**Non-Parametric Latent GCP (nLGCP)**: As stated above, the prior mean of a Gaussian process is usually fixed as the empirical mean of the observations $y_{t,i}$. In the context of hyperparameter optimization, however, we can easily feel that there should be some region where the hyperparameters would be *rather good*, and others where they would be *rather bad*. Thus it appears that it may be convenient to set a different mean for the prior depending on the region in which the hyperparameter is found. Regarding the GP, it could be argued that this would not have much impact, as the covariance function is meant to induce such smoothness. However with GCP we fix not only the mean function, but the mapping function as well. We recall that the mapping function reflects the distribution of the data. Thus with nGCP, a latent model aims to learn several distributions of $y$ over the input space. In particular, we think this could have a positive influence when trying to locate good regions to explore in a Bayesian optimization process.

Thus, we introduce a non-parametric *Latent Gaussian Copula Process* prior (nLGCP), where the mapping function also depends on the input $h$. Intuitively, the goal is to include in the prior not only the distribution $\mathcal{D}$ of $y$ on the entire space $\mathcal{H}$ but the distributions $\mathcal{D}_1$, ..., $\mathcal{D}_k$ of $y$ on $k$ regions of $\mathcal{H}$. This way, we design a prior that *really* depends on $h$ (whereas in the previous equations, $h$ was only an index to denote the random variable $y$).

To design the nLGCP prior, we look for a mapping function that depends on the input $h$ *and* output $y$.

To this end, the training data $\{(h_i, y_i)\}$ are clustered in $\mathcal{H} \times \mathcal{Y} \subset \mathbb{R}^{m+1}$ using *K-means*. For each cluster $\mathcal{C}_k$, a mapping $\Psi_k$ is learned as described in the previous section. Then for each $h$ in $\mathcal{H}$, the final mapping $\Psi$ is computed as:

$$\Psi(h, y) = \sum \alpha_k(h).\Psi_k(y), \tag{10}$$

where: $\alpha_k(h) = exp(-s \sum (\frac{d_k}{\sigma_k})^2)$ and $d_k = dist_{\mathcal{H}}(h, \mathcal{C}_k)$ and $\sigma_k = std_{\mathcal{H}}(\mathcal{C}_k)$. $s$ is a smoothing coefficient, and $c_k$ is the projected center of the cluster $\mathcal{C}_k$ on $\mathcal{H}$.

**Predictions with nLGCP**: Predictions with GP are straightforward given the equations (4,5) but this is no longer the case with nLGCP. A faster but more approximate way to compute the predicted value of $y_*$ for a given $h_*$ is to calculate $z_*$, which is the standard GP prediction of the warped output $\Psi(y_*)$ given by the posterior: $z_* \sim \mathcal{N}(\mu_*, \sigma_*)$, where $\mu_*, \sigma_*$ correspond to the detailed expressions of equation (5). Noting from the equation (6) that the predicted cumulative distribution function of $y_*$ is $F_* = \Phi(\Psi; \mu_*, \sigma_*)$, we can evaluate the prediction as:

$$y_* = \int_{u=-\infty}^{\infty} u \cdot \Psi'(u) \cdot \phi(\Psi(u); \mu_*, \sigma_*) \cdot du \tag{11}$$

where $\phi_{\mu_*,\sigma_*}$ denotes the probability density function of a univariate Gaussian $\mathcal{N}(\mu_*, \sigma_*)$.

The particular expression of $\Psi$ in e.q. (9) for the nGCP prior finally enables us to express its derivative directly:

$$\Psi'_{nGCP}(y) = \frac{d_{est}(y)}{\phi(\Psi(y))}, \tag{12}$$

where $\phi$ and $d_{est}$ are the probability density function of the standard univariate Gaussian and the one corresponding to $F_{est}$ defined in section V-A, respectively. Similarly, we can calculate the derivatives for $\Psi_{nLGCP}$.

**Tuning the algorithm**: The tuning algorithm executes as follows:

- **Step 1**: It receives the hyperparameter set and their estimated scores from the BLB.
- **Step 2**: It fits an nLGCP model to this data.
- **Step 3**: To choose the next hyperparameter set, a grid of candidates is built from the space of hyperparameters. The model is used to predict mean and variance for the score for every point in the hyperparameter space.
- **Step 4**: An acquisition function based on Expected Improvement or an Upper Confidence Bound [1] is used to select the next hyperparameter set.

- **Step 5**: Then, the BLB estimator is evaluated on that hyperparameter set, and the process continues for the specified number of iterations.

## VI. Experiments

In our Deep Mining experimentation, we aimed to answer the following question: Does Bayesian hyperparameter optimization using BLB effectively search the hyperparameter space? If this optimization is effective, then the asymptotic complexity of hyperparameter optimization will be dramatically improved, giving a theoretical result that can accelerate the tuning of data science pipelines. If optimization with estimator scores using BLB on the pipeline is effective, then we will see improvement in the pipeline performance roughly comparable to the non-BLB approach, where we calculate the pipeline scores using all of the data.

### *A. Datasets*

For this question, we experimented with data science pipelines on image and text datasets:

*1) Handwritten digits:* We first considered the famous handwritten digits recognition problem from the MNIST dataset. In this dataset, the training images are handwritten digits from 0 to 9, and the task of the classifier is to predict the digit label that goes with each greyscale image.

*2) Sentiment analysis:* : The second problem we considered is based on the plain text of movie reviews from IMDB, with the goal of predicting whether a review is positive (if it received a rating greater than or equal to 5) or negative (if it received a rating lower than 5). The dataset used is from Kaggle's Sentiment Analysis competition https://www.kaggle.com/c/word2vec-nlp-tutorial/dataBag of Words Meets Bags of Popcorn.

### *B. Pipelines*

We used three pipelines in our experiments. For the image dataset, we used a convolutional neural network pipeline as well as a more traditional pipeline, with the conventional HOG feature extraction methods. For the text dataset, we used a traditional pipeline, with feature extraction and processing methods conventional for the data type.

**Traditional Image Pipeline**: For images, we first considered a pipeline using the scikit-image [15] implementation of Histogram of Oriented Gradients (HOG) and scikit-learn's Random Forest classifier.

**Convolutional Neural Network Image Pipeline**: For images, we also considered a convolutional neural network (CNN), as CNNs have demonstrated state-of-the-art results on image datasets, and tuning of their architecture hyperparameters can significantly affect their performance.

**Traditional Text Pipeline**: For the sentiment analysis problem, we first considered a conventional pipeline using bag of n-grams transformations and a Naive Bayes classifier.

### *C. Methodology*

The following description includes three new terms that we will define here. An `iteration` is an evaluation of a single hyperparameter set – in other words, one single training and validation of the specified data science pipeline. A `trial` involves multiple `iterations` constituting a single run of the SmartSearch algorithm, resulting in the best possible pipeline at the end of the specified iterations. An *experiment*, then, consists of multiple trials on a single pipeline for a single dataset.

For each pipeline, we run one experiment using BLB and another in which we operate on the whole dataset. Each experiment consists of ten trials of SmartSearch with 30 iterations in each trial (non-BLB). Each SmartSearch trial consists of the evaluation of 30 hyperparameter sets, with the first 3 hyperparameter sets chosen at random, the next 24 hyperparameter sets chosen by the nLGCP algorithm with the Upper Confidence Bound acquisition function, and the final 3 sets chosen simply by the highest predicted score. We run three experiments in this case: the HOG image pipeline, the CNN image pipeline, and the traditional text pipeline. The BLB hyperparameters used in these experiments were bag size $b = n^{0.6}$, number of bags $s = 8$, and number of multinomial samples $r = 20$.

**Computation used**:For the BLB experiments, we used an Apache Spark cluster consisting of four m4.4xlarge machines, each of which have 16 vCPUs and 64 GiB of memory. For the non-BLB experiments, we used an m4.16xlarge machine, which has 64 vCPUs and 256 GiB memory.

### *D. Evaluation*

The train/validation split varies for the BLB and non-BLB pipeline processes and by dataset.

*1) MNIST Dataset:* In the MNIST dataset, we have 42,000 data points, which we split into a training dataset of 36,000 data points and associated labels, as well as a validation set of 6,000 data points and associated labels. The non-BLB process is run by training on the 36,000 data points and validating on the 6,000 data points for each hyperparameter set, outputting to SmartSearch the score on the 6,000 data points from the validation set.

During the BLB process, we use the 36,000 data points for training and validation. For final comparison in the below analysis, we calculate the score on the remaining 6,000 data points by training the estimator with the chosen hyperparameters on the 36,000 training points, and outputting the score of that trained estimator on the remaining 6,000 data points.

*2) Text Dataset:* In the dataset from Kaggle's competition, we have 25,000 text reviews and their associated labels, and we choose a training/validation split similar to that used on the MNIST dataset. We split our dataset into a training dataset of 20,000 data points and associated labels as well as a validation set of 5,000 data points and associated labels. The non-BLB and BLB processes are then the same as for the MNIST dataset with these training and validation datasets.

Our comparison framework has the following properties:

TABLE I
TYPES OF PIPELINES.

| Pipeline type | Description | Hyperparameters |
|---|---|---|
| Traditional Image Pipeline | Extract a Histogram of Oriented Gradients (HOG) for each image | • Number of orientation bins: [7,9] • Size (in pixels) of a cell: [3,6] • Number of cells in each block: [3,6] |
| | Classify with a Random Forest Classifier | Number of trees in the random forest: [2,10] |
| Convolutional Neural Network Image Pipeline | Two-dimensional convolutional layer | Width (and height) of the square convolutional kernel: [3,5] |
| | Two-dimensional max-pooling layer | Width (and height) of the square pool: [2,5] |
| | Dropout layer | Dropout percentage: [0.0,0.75] |
| | Densely connected layer with softmax activation | |
| Traditional Text Pipeline | Transform the reviews in Bags of n-grams | • Maximum number of features to keep $n_f \in [1000; 400000]$. • Consider only the terms with a document frequency between $df_{min}$ and $df_{max}$, where $df_{min} \in [0; 0.3], df_{max} \in [0.4; 1]$. • Consider n-grams terms for n between 1 and $n_{ngram,max}$, where $n_{ngram,max} \in [1; 4]$. |
| | Transform the n-grams count vectors in tf-idf vectors | • Norm L1 • Norm L2 |
| | Classify with a multinomial Naive Bayes classifier | Classification: we use a Lidstone smoothing parameter $\alpha \in [0.01; 1]$ |

- Comparison between two hyperparameter sets, one from BLB and one from non-BLB is fair. Even though in BLB version the score is estimated using the sampled version of the datasets, the estimate for comparison is generated by training the pipeline (with the specified hyperparamter set) one last time on all of the training data and testing on the validating data, which is consistent with how non-BLB estimates are produced.
- We gave an unfair advantage to non-BLB by feeding its score on the validation set to its hyperparameter optimizer. This is not the case for the BLB case. Thus the hyperoptimizer in the non-BLB method is optimizing accuracy over the validation set.

### *E. Results*

In our results, our goal is to see if BLB based method in conjunction with hyperparameter tuning can improve the pipeline performance as iterations progress. Are these improvement comparable to non-BLB based method? In our experimental results, we find that *sampling-based hyperparameter optimization using BLB leads to performance improvements comparable with those obtained by evaluating on the entire pipeline.*

In order to see this result, we can first plot the improvement over the course of the SmartSearch process for each pipeline, averaged over the ten experiments. In Figures 1, 2, and 3, the horizontal axis corresponds to the iteration number and the vertical axis corresponds to the best estimator performance before and including a given iteration (on the validation set).

The performances of BLB and non-BLB estimator evaluation are comparable for all of these pipelines, with the BLB iterations demonstrating similar performance improvements. In the HOG image pipeline, the BLB evaluations actually slightly outperform the non-BLB evaluations, likely as a result of the higher average starting accuracy. In the CNN image pipeline, the BLB evaluations show the least improvement of the three pipelines relative to the non-BLB evaluation. This underperformance is likely due to neural networks' reliance on large datasets, as training and validating a CNN on only $\lceil 360000^{0.6} \rceil = 542$ data points is expected to lead to poor performances. The performance increases for the BLB evaluation traditional text pipeline are quite near those of the non-BLB evaluation, and the lower average starting performance can be attributed to the small amount of experimental data.

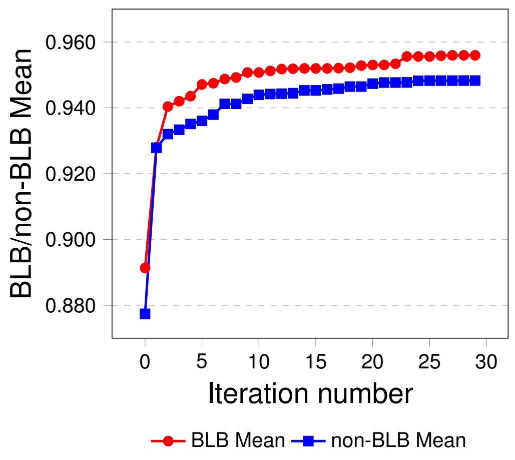

Fig. 1. Experimental averages for the HOG image pipeline.

We can also evaluate the improvement more quantitatively, examining the average improvement and standard deviation of that improvement for both the BLB and non-BLB processes in

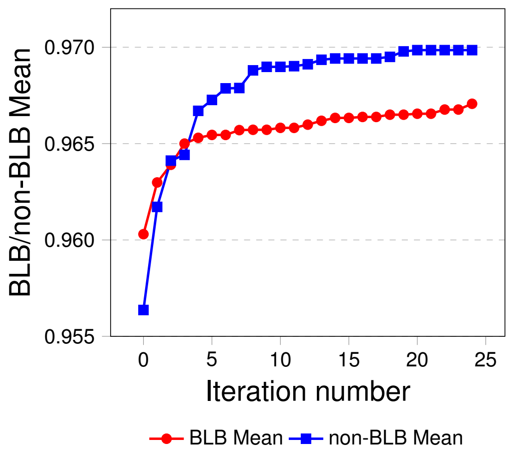

Fig. 2. Experimental averages for the CNN Image pipeline.

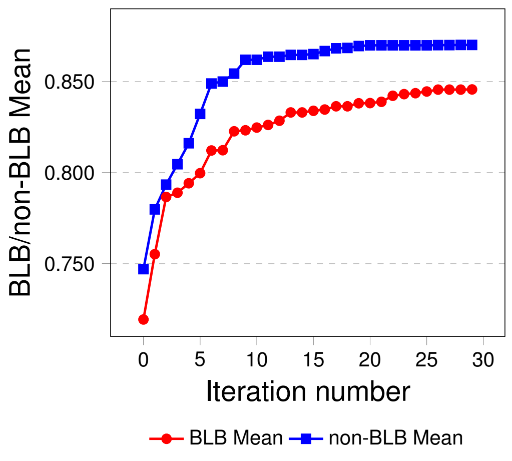

Fig. 3. Experimental averages for the traditional text pipeline.

Table II. As we can see, the mean performance improvements are comparable for the pipelines, with the CNN image pipeline leading to lower improvements for the reasons detailed below.

## *F. Discussion*

When evaluating the hyperparameter sets chosen by the BLB and non-BLB SmartSearch experiments, we can see through manual inspection that their results follow intuition.

TABLE II
BLB AND NON-BLB PROCESSES STATISTIC. STD. DEV. = STANDARD DEVIATION.

| | BLB | | Non-BLB | |
|---|---|---|---|---|
| | Mean | Std. Dev. | Mean | Std. Dev. |
| HOG Image Pipeline | 0.065 | 0.056 | 0.071 | 0.063 |
| CNN Image Pipeline | 0.0068 | 0.0053 | 0.0135 | 0.012 |
| Traditional Text Pipeline | 0.126 | 0.066 | 0.123 | 0.067 |

For example, in the HOG pipeline, we use a random forest estimator, with the number of trees as a hyperparameter. The hyperparameter sets determined by the SmartSearch consistently choose the highest number of trees (ten). This is what we would expect, as our random forest classifier generally improves with additional trees. However, the computation time also scales linearly with the number of trees, so the hyperparameter sets chosen take longer to evaluate. In future work, a scoring metric that incorporates a tradeoff between accuracy and computation time – perhaps by incorporating a linear penalty on the amount of time spent running the pipeline – could be implemented to incorporate a time-accuracy tradeoff based on user preferences.

These initial experiments focused on obtaining accurate estimates that are adequate for use in Bayesian hyperparameter optimization, not empirical speed improvements. While we see that the performance improvements are comparable, we do not see significant computational speed-ups in our initial experiments. This observation is due to three factors: 1) The datasets used are small, so the asymptotic benefits of the BLB algorithm are not realized; 2) We have not tuned the Apache Spark-specific parameters, leading to greater overhead; 3) Our implementation is in Python, incurring substantially more overhead than an implementation in a lower-level language. When we increased the computational power of the cluster machines, we saw that the speed improvements were small. We attribute this to the small dataset size, which leads to the Apache Spark overhead dominating the computation time.

The performance improvements for the CNN image pipeline are smaller than those for the more traditional pipelines, which aligns with the intuitive conclusion that neural networks need large amounts of data to achieve high accuracies. The amount of data given in each bag is $\lceil 360000^{0.6} \rceil = 542$, so the reduced improvement in the hyperparameter optimization is reasonable given the extremely small size of the dataset.

The performance improvements seen are impressive, as even operating on small subsets of the data leads to performance improvements on par with pipeline evaluations on the entire datasets. These empirical results demonstrate the validity of using BLB for pipeline evaluation in hyperparameter optimization, and future work can optimize the existing Deep Mining system to reduce computational overhead and take full advantage of this asymptotic improvement.

## VII. Related Work

The hyperparameter optimization community has been extremely active in recent years. This summary is not intended to give a comprehensive list of current hyperparameter tuning methods, but rather a coarse overview. One approach that has come out of this research is *Bayesian optimization*, which treats the pipeline as a black box function. Bayesian optimization uses *Gaussian Process* (GP) to search the space [1], [16], which is similar to our Gaussian copula process, as presented in Section V. Numerous other methods to model the search space and then sample from it have also been developed, including multi-armed bandit methods in [17]. The Tree of

Parzen Estimators has been used, along with the Expected Improvement acquisition function, which models the posterior indirectly using $p(x|y)$ and $p(y)$ rather than modeling $p(y|x)$ directly as in Gaussian processes [16]. Reinforcement learning has been used on neural network architectures [18], and gradient descent has been shown to be effective for some continuous hyperparameters [19]. The radial basis function has also been used [20], as well as a spectral approach that improves on the asymptotic complexity of the GP fitting process [21]. Multi-task Gaussian processes have been applied to Bayesian hyperparameter optimization to incorporate information from previous optimizations [22], and transformations have been applied to construct a more flexible prior for the Bayesian optimization [23]. In addition, several open source systems have also been released that highlight these methods and enable comparison on benchmarks. Examples of these systems are noted in Table VII.

Considering the existence of these numerous approaches as well as related open source software tools, one would be forgiven for asking: *Why do we need yet another hyperparameter optimization system?*. We believe our system addresses certain needs that have been overlooked by others. First of all, most existing systems focus on the questions *how best can we model the search space?*, and *how best can we query this model?*, aiming to reduce the number of iterations necessary. To compare these approaches, the community has established a number of benchmark datasets and black box functions. In our experience, we find that this emphasis, while important in its own right, does not consider the challenges presented by real-world industrial scale data sets and problems. We highlight some of these challenges below.

**Tuning data science pipeline vs. machine learning pipelines:** We make a distinction between *machine learning pipelines* and *data science pipelines* – one that is not often made explicit in existing literature. In practice, data science pipelines include earlier steps not present in machine learning pipelines, such as preprocessing and feature extraction steps – e.g. bag-of-words (for natural language) or HOG feature extraction (in case of images).

Consider the problem of tuning a pipeline that consists of only a Support Vector Machine (SVM) classifier. Here, data is expected to be delivered in the $X$ and $y$ format, where $X$ is a matrix of features and $y$ is a vector of labels. While this method can be called a "data science pipeline," the term is misleading because the method consists of only one step. Even if it extends one step further, incorporating Principal Component Analysis (PCA) and scaling the feature matrix in addition to SVM, this is still more of a "machine learning pipeline" than a "data science pipeline."

By making this distinction, Deep Learning can differentiate between systems that are focused on machine learning pipelines, and those that should be extended to entire data science pipelines. It is worth noting that the kind of pipeline also has implications as to which data domains and representations that should be addressed.

**The limitations of the "black box" approach to tuning:** The aforementioned hyperparameter optimization systems typically take one of two "black box" approaches. In the first case, they treat the performance of a pipeline as a function, for which they provide inputs (hyperparameter sets) and receive an output value (e.g. a scoring metric). These systems place the burden of pipeline implementation entirely on the user. Arguably, this abstraction can aid in tuning entire data science pipelines as well. Figure 4 presents a schematic for this abstraction. Table VII describes systems that use this framework, including Spearmint, the startup SigOpt, MOE, SMAC, BayesOpt, REMBO, and HPOlib.

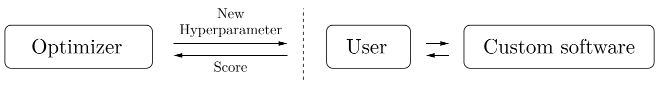

Fig. 4. Illustration of the first common abstraction approach. The optimization algorithm gives hyperparameter sets to and gets performance metrics from the pipeline, which it treats as a black box. User is responsible for writing the software for the pipeline and its execution.

TABLE III
(APPROXIMATE) CLASSIFICATION OF CURRENT STATE-OF-ART HYPERPARAMETER SYSTEMS. **BB** REFERS TO THE TYPE OF **API** OR THE BLACK BOX FUNCTION METHOD THEY USE. 1 IMPLIES THE API DESCRIBED IN FIGURE 4 IS USED. 2 IMPLIES THAT THE API DESCRIBED IN FIGURE 5 IS USED.

| System | BB | Implementation | Paper |
|---|---|---|---|
| Spearmint | 1 | https://github.com/HIPS/Spearmint | Various |
| SigOpt | 1 | https://sigopt.com/getstarted | [24] |
| MOE | 1 | https://github.com/Yelp/MOE | [25] |
| SMAC | 1 | https://github.com/automl/SMAC3 | [3] |
| BayesOpt | 1 | https://github.com/rmcantin/bayesopt | [26] |
| REMBO | 1 | https://github.com/ziyuw/rembo | [27] |
| HPOlib | 1 | https://github.com/automl/hpolib | [4] |
| Hyperopt | 1 | https://github.com/hyperopt/hyperopt | [28] |
| Hyperopt-sklearn | 2 | https://github.com/hyperopt-sklearn | [29] |
| Auto-WEKA | 2 | http://www.cs.ubc.ca/labs/beta/Projects/autoweka/ | [30] |
| Hyperband | 2 | https://github.com/zygmuntz/hyperband | [17] |
| TPOT | 2 | https://github.com/rhiever/tpot | [31] |
| Auto-sklearn | 2 | http://automl.github.io/auto-sklearn/stable/ | [32] |
| Osprey | 2 | https://github.com/msmbuilder/osprey | [33] |
| Optunity | 2 | https://github.com/claesenm/optunity | [34] |
| mlr | 2 | https://github.com/mlr-org/mlr | [35] |
| Scikit-optimize | Other | https://github.com/scikit-optimize/scikit-optimize | [36] |

While this approach allows users the flexibility to implement arbitrary pipelines, it also has multiple weaknesses:

- The lack of an API for specifying pipelines and exposing hyperparameters results in significantly increased user effort in constructing pipelines.
- Because users are responsible for implementing the pipeline and processing the data, their implementation alone determines the efficiency of the hyperparameter optimization process. No framework is provided for them to accelerate the hyperparameter set evaluations.
- The question of importing data is entirely ignored, forcing users to spend time aggregating and formatting data.

In the other typical approach to hyperparameter optimization, system designers choose a predefined set of implemented pipelines, which requires input data to be in a matrix format.

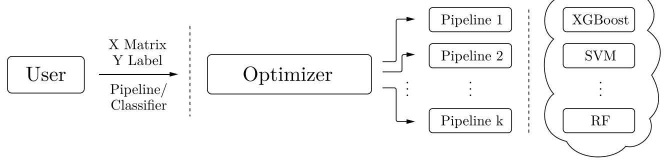

Fig. 5. Illustration of the second common abstraction approach. The user provides data in a matrix format (either $X$, $Y$, or $X$ and $Y$ in a cross validation split). The optimizer chooses a pipeline from within its implemented machine learning algorithms, and returns a classifier or pipeline with the highest score to the user.

Users of the system cannot include custom pipelines that they judge to be appropriate for their problems. These systems also constrain those users by forcing them to choose among a set suite of classifiers or regressors, as well as the occasional feature preprocessing method. Figure 5 illustrates the schema for this abstraction.

These systems are effective for many small, matrix-formatted datasets, achieving impressive results on benchmark datasets. However, they are much less well-suited for applications on large datasets in non-matrix format (e.g., relational databases), and the lack of clear APIs to specify custom pipelines severely limits their effectiveness.

## VIII. Conclusion

We have made multiple contributions through this work. We:

- Formulated a powerful pipeline abstraction using scikit-learn utilities,
- Implemented a system incorporating BLB, using Apache Spark and a structured pipeline interface to evaluate scores of data science pipelines,
- Constructed multiple pipelines using the Deep Mining system,
- Evaluated the effectiveness of BLB evaluation for data science pipelines, showing empirically that performance improvements are comparable to non-BLB pipeline evaluations,
- Formulated new algorithms using copula processes to improve upon the Gaussian process model of the hyper-parameter space, and
- Developed an open-source Bayesian hyperparameter optimization implementation using Gaussian processes with multiple acquisition functions.

The Deep Mining system incorporates distributed computation, subsampling methods, and conditional pipeline evaluation to allow hyperparameter optimization of complex pipelines on large datasets. The empirical results of this paper demonstrate the validity of using BLB for pipeline evaluation in hyperparameter optimization, showing a substantial asymptotic improvement that enables scalable systems capable of tuning the computationally intensive applications of modern data science.

## References

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

[2] J. S. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, "Algorithms for hyper-parameter optimization," in *Advances in Neural Information Processing Systems*, 2011, pp. 2546–2554.

[3] F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Sequential model-based optimization for general algorithm configuration." *LION*, vol. 5, pp. 507–523, 2011.

[4] 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*, vol. 10, 2013.

[5] D. G. Lowe, "Object recognition from local scale-invariant features," in *Computer vision, 1999. The proceedings of the seventh IEEE international conference on*, vol. 2. Ieee, 1999, pp. 1150–1157.

[6] A. Kleiner, A. Talwalkar, P. Sarkar, and M. I. Jordan, "A Scalable Bootstrap for Massive Data," *ArXiv e-prints*, Dec. 2011.

[7] L. Oneto, B. Pilarz, A. Ghio, and D. Anguita, "Model selection for big data: Algorithmic stability and bag of little bootstraps on gpus," in *Proceedings*. Presses universitaires de Louvain, 2015, p. 261.

[8] C. Garnatz, "Trusting the black box: Confidence with bag of little bootstraps," Ph.D. dissertation, Pomona College, 2015.

[9] B. Catanzaro, S. Kamil, Y. Lee, K. Asanovic, J. Demmel, K. Keutzer, J. Shalf, K. Yelick, and A. Fox, "SEJITS: Getting productivity and performance with selective embedded JIT specialization," in *Workshop on Programmable Models for Emerging Architecture (PMEA)*, 2009. [Online]. Available: http://parlab.eecs.berkeley.edu/publication/296

[10] S. A. Kamil, "Productive high performance parallel programming with auto-tuned domain-specific embedded languages," Ph.D. dissertation, EECS Department, University of California, Berkeley, Jan 2013. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-1.html

[11] C. Rasmussen and C. Williams, *Gaussian Processes for Machine Learning*, ser. Adaptive Computation And Machine Learning. MIT Press, 2005. [Online]. Available: http://www.gaussianprocess.org/gpml/chapters/

[12] A. Wilson and Z. Ghahramani, "Copula processes," in *Advances in Neural Information Processing Systems*, 2010, pp. 2460–2468.

[13] E. Snelson, C. E. Rasmussen, and Z. Ghahramani, "Warped gaussian processes," *Advances in neural information processing systems*, vol. 16, pp. 337–344, 2004.

[14] B. W. Silverman, *Density estimation for statistics and data analysis*. CRC press, 1986, vol. 26.

[15] S. van der Walt, J. L. Schönberger, J. Nunez-Iglesias, F. Boulogne, J. D. Warner, N. Yager, E. Gouillart, T. Yu, and the scikit-image contributors, "scikit-image: image processing in Python," *PeerJ*, vol. 2, p. e453, 6 2014. [Online]. Available: http://dx.doi.org/10.7717/peerj.453

[16] J. S. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, "Algorithms for hyper-parameter optimization," in *Advances in Neural Information Processing Systems 24*, J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2011, pp. 2546–2554. [Online]. Available: http://papers.nips.cc/paper/4443-algorithms-for-hyper-parameter-optimization.pdf

[17] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar, "Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization," *ArXiv e-prints*, Mar. 2016.

[18] B. Baker, O. Gupta, N. Naik, and R. Raskar, "Designing Neural Network Architectures using Reinforcement Learning," *ArXiv e-prints*, Nov. 2016.

[19] D. Maclaurin, D. Duvenaud, and R. P. Adams, "Gradient-based Hyperparameter Optimization through Reversible Learning," *ArXiv e-prints*, Feb. 2015.

[20] G. Diaz, A. Fokoue, G. Nannicini, and H. Samulowitz, "An effective algorithm for hyperparameter optimization of neural networks," *ArXiv e-prints*, May 2017.

[21] E. Hazan, A. Klivans, and Y. Yuan, "Hyperparameter Optimization: A Spectral Approach," *ArXiv e-prints*, Jun. 2017.

[22] K. Swersky, J. Snoek, and R. P. Adams, "Multi-task bayesian optimization," in *Advances in neural information processing systems*, 2013, pp. 2004–2012.

[23] J. Snoek, K. Swersky, R. Zemel, and R. Adams, "Input warping for bayesian optimization of non-stationary functions," in *International Conference on Machine Learning*, 2014, pp. 1674–1682.

[24] I. Dewancker, M. McCourt, S. Clark, P. Hayes, A. Johnson, and G. Ke, "Evaluation System for a Bayesian Optimization Service," *ArXiv e-prints*, May 2016.

[25] Yelp, "Metric optimization engine (moe)," https://github.com/Yelp/MOE, 2017.

[26] R. Martinez-Cantin, "Bayesopt: A bayesian optimization library for nonlinear optimization, experimental design and bandits," *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 3735–3739, 2014.

[27] Z. Wang, F. Hutter, M. Zoghi, D. Matheson, and N. de Feitas, "Bayesian optimization in a billion dimensions via random embeddings," *Journal of Artificial Intelligence Research*, vol. 55, pp. 361–387, 2016.

[28] J. Bergstra, D. Yamins, and D. D. Cox, "Hyperopt: A python library for optimizing the hyperparameters of machine learning algorithms," in *Proceedings of the 12th Python in Science Conference*, 2013, pp. 13–20.

[29] B. Komer, J. Bergstra, and C. Eliasmith, "Hyperopt-sklearn: automatic hyperparameter configuration for scikit-learn," in *ICML workshop on AutoML*, 2014.

[30] C. Thornton, F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Auto-weka: Automated selection and hyper-parameter optimization of classification algorithms," *CoRR, abs/1208.3719*, 2012.

[31] R. S. Olson, N. Bartley, R. J. Urbanowicz, and J. H. Moore, "Evaluation of a tree-based pipeline optimization tool for automating data science," in *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, ser. GECCO '16. New York, NY, USA: ACM, 2016, pp. 485–492. [Online]. Available: http://doi.acm.org/10.1145/2908812.2908918

[32] M. Feurer, A. Klein, K. Eggensperger, J. Springenberg, M. Blum, and F. Hutter, "Efficient and robust automated machine learning," in *Advances in Neural Information Processing Systems*, 2015, pp. 2962–2970.

[33] R. T. McGibbon, C. X. Hernández, M. P. Harrigan, S. Kearnes, M. M. Sultan, S. Jastrzebski, B. E. Husic, and V. S. Pande, "Osprey: Hyperparameter optimization for machine learning," *The Journal of Open Source Software*, vol. 1, no. 5, sep 2016. [Online]. Available: https://doi.org/10.21105/joss.00034

[34] M. Claesen, J. Simm, D. Popovic, and B. Moor, "Hyperparameter tuning in python using optunity," in *Proceedings of the International Workshop on Technical Computing for Machine Learning and Mathematical Engineering*, 2014, pp. 6–7.

[35] B. Bischl, M. Lang, L. Kotthoff, J. Schiffner, J. Richter, E. Studerus, G. Casalicchio, and Z. M. Jones, "mlr: Machine learning in r," *Journal of Machine Learning Research*, vol. 17, no. 170, pp. 1–5, 2016. [Online]. Available: http://jmlr.org/papers/v17/15-066.html

[36] M. K. et. al., "Scikit-optimize," https://github.com/scikit-optimize/scikit-optimize, 2017.

[37] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *Journal of Machine Learning Research*, vol. 13, no. Feb, pp. 281–305, 2012.

[38] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, M. J. Franklin, R. Zadeh, M. Zaharia, and A. Talwalkar, "Mllib: Machine learning in apache spark," *J. Mach. Learn. Res.*, vol. 17, no. 1, pp. 1235–1241, Jan. 2016. [Online]. Available: http://dl.acm.org/citation.cfm?id=2946645.2946679

[39] T. Domhan, J. T. Springenberg, and F. Hutter, "Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves." in *IJCAI*, 2015, pp. 3460–3468.

[40] T. Horváth, R. G. Mantovani, and A. C. de Carvalho, "Effects of random sampling on svm hyper-parameter tuning," in *International Conference on Intelligent Systems Design and Applications*. Springer, 2016, pp. 268–278.

[41] 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, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[42] Dask Development Team, *Dask: Library for dynamic task scheduling*, 2016. [Online]. Available: http://dask.pydata.org

[43] M. Zaharia, *An architecture for fast and general data processing on large clusters*. Morgan & Claypool, 2016.

[44] M. M. McKerns, L. Strand, T. Sullivan, A. Fang, and M. A. G. Aivazis, "Building a Framework for Predictive Science," *ArXiv e-prints*, Feb. 2012.

[45] F. Chollet *et al.*, "Keras," https://github.com/fchollet/keras, 2015.

[46] C. Z. Mooney and R. D. Duval, *Bootstrapping: A nonparametric approach to statistical inference*. Sage, 1993, no. 94-95.

[47] S. Dubois, "Deep mining : Copula-based hyper-parameter optimization for machine learning pipelines," Master's thesis, École polytechnique - Massachusetts Institute of Technology, Massachusetts Institute of Technology - CSAIL, 2015.

[48] A. Anderson, S. Dubois, A. Cuesta-Infante, and K. Veeramachaneni, "Sample, estimate, tune: Scaling bayesian auto-tuning of data science pipelines," in *Data Science and Advanced Analytics (DSAA), 2017. IEEE International Conference on*. IEEE, 2017, pp. 1–10.

[49] 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," *CoRR*, vol. abs/1012.2599, 2010. [Online]. Available: http://arxiv.org/abs/1012.2599

[50] H. B. Nielsen, S. N. Lophaven, and J. Sondergaard, "Dace, a matlab kriging toolbox," *Informatics and mathematical modelling. Lyngby–Denmark: Technical University of Denmark, DTU*, 2002.

[51] B. L. Welch, "The significance of the difference between two means when the population variances are unequal," *Biometrika*, vol. 29, no. 3-4, pp. 350–362, 1938.

[52] Student, "The probable error of a mean," *Biometrika*, pp. 1–25, 1908.

[53] G. Bradski, *Dr. Dobb's Journal of Software Tools*.

[54] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.