Deep Mining: Scaling Bayesian Auto-tuning of Data Science Pipelines

by

Alec W. Anderson

S.B., Massachusetts Institute of Technology (2017)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2017

© Massachusetts Institute of Technology 2017. All rights reserved.

Author . . . .
Department of Electrical Engineering and Computer Science
August 14, 2017

Certified by . . . .
Kalyan Veeramachaneni
Principal Research Scientist
Thesis Supervisor

Accepted by . . . .
Christopher J. Terman
Chairman, Masters of Engineering Thesis Committee

Deep Mining: Scaling Bayesian Auto-tuning of Data Science Pipelines

by

Alec W. Anderson

Submitted to the Department of Electrical Engineering and Computer Science
on August 14, 2017, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Within the automated machine learning movement, hyperparameter optimization has emerged as a particular focus. Researchers have introduced various search algorithms and open-source systems in order to automatically explore the hyperparameter space of machine learning methods. While these approaches have been effective, they also display significant shortcomings that limit their applicability to realistic data science pipelines and datasets.

In this thesis, we propose an alternative theoretical and implementational approach by incorporating sampling techniques and building an end-to-end automation system, *Deep Mining*. We explore the application of the Bag of Little Bootstraps to the scoring statistics of pipelines, describe substantial asymptotic complexity improvements from its use, and empirically demonstrate its suitability for machine learning applications. The Deep Mining system combines a standardized approach to pipeline composition, a parallelized system for pipeline computation, and clear abstractions for incorporating realistic datasets and methods to provide hyperparameter optimization at scale.

Thesis Supervisor: Kalyan Veeramachaneni
Title: Principal Research Scientist

Acknowledgments

I would like to thank my adviser, Kalyan Veeramachaneni. His feedback and ideas made this work possible, and his vision and passion guided me throughout this work. I'd also like to thank my DAI labmates and the Feature Labs team for their helpful feedback and company.

I'd like to thank my friends for all of their support over the last four years and for making my MIT experience very special. The ideas and support you gave me continued to push me, and I couldn't have done it without you.

Finally, I'd like to thank my mom, my dad, and my brother. Words can't express what your support means to me, and I'm forever grateful to you.

Contents

1 Introduction **15**

1.1 Automating what a data scientist does . . . 16

1.2 Deep mining . . . 19

1.3 Contributions . . . 20

1.4 Previous Work . . . 21

1.5 Outline . . . 21

2 Background and Related Work **23**

2.1 Search Algorithms and Meta-modeling . . . 24

2.1.1 Meta-modeling Approach . . . 25

2.2 Choices of $g(\mathbf{h})$ . . . 27

2.3 Current Hyperparameter Optimization Systems . . . 27

3 Deep Mining: Overview **31**

3.1 Sampling-based Approach for Hyperparameter Search . . . 34

3.2 The Deep Mining System . . . 35

4 Gaussian Copula Processes **39**

4.1 Gaussian Processes . . . 39

4.2 Gaussian Copula Process (GCP) . . . 41

5 Composing arbitrary pipelines **45**

5.1 Abstractions for data transformation steps . . . 46

5.2 Abstractions for the pipeline . . . 48

5.2.1 Abstractions in scikit-learn 49
5.3 Advantages of pipeline abstraction 52
5.4 Pipeline Construction in Deep Mining 53
5.4.1 Custom Pipeline Example 54
5.5 Conditional Pipeline Evaluation 57
5.6 Enabling Raw Data 57
5.7 Contribution Framework 58

6 Deep Mining: Sampling-based estimation 61
6.1 Sampling Applications 62
6.2 Bag of Little Bootstraps 63
6.3 Application to Tuning Data Science Pipelines 65
6.3.1 Sampling for Raw Data Types 66
6.3.2 Cross-validation in BLB 67
6.3.3 Application of BLB to Varied Pipeline Methods 68
6.4 Implementation 69
6.5 Extensions 70
6.5.1 Reducing Complexity of Data Science pipelines 70

7 Deep Mining: Parallel computation 73
7.1 Parallelization Methods 73
7.2 Parallelization Frameworks 74
7.3 Current Implementation 75
7.4 Future Work 76

8 Interacting with Deep Mining 77
8.1 End-to-end System 77
8.1.1 Data Loading 78
8.1.2 Defining A Pipeline 80
8.1.3 Run Hyperparameter Optimization 80
8.1.4 Built-in Pipeline Example 83

8.1.5 Custom Pipeline Example 83

9 Experimental Results **87**

9.1 Datasets 87

9.1.1 Handwritten Digits 87

9.1.2 Sentiment Analysis 88

9.2 Pipelines 88

9.3 Methodology 90

9.4 Evaluation 90

9.4.1 MNIST Dataset 91

9.4.2 Text Dataset 91

9.5 Results 91

9.6 Discussion 94

10 Conclusion **97**

10.1 Future Work 97

A Comments on Apache Spark **99**

B Functions and abstractions in *DeepMine* **101**

List of Figures

1-1 End-to-end pipeline to build a predictive model. It includes the preprocessing and feature extraction steps along with the modeling steps. 17

2-1 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. 29

2-2 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. 29

3-1 Illustration of the portion of the facial recognition pipeline to optimize. 32

3-2 Illustration of the Deep Mining system's functionality. The user specifies a pipeline using the provided framework, providing hyperparameter ranges and the necessary pipeline steps to the pipeline constructor and executor. The user also provides the data in either raw or matrix format and cross-validation parameters to the data loader as well as optimization parameters to the Bayesian optimizer. Outside of the user's view, Deep Mining constructs a scoring function using the provided pipeline and parameters, incorporating distribution and sampling to accelerate the evaluation process. Deep Mining also loads the data into a cross validation split for the pipeline executor, which chooses between different parallel and non-parallel evaluation methods that return a score to the pipeline executor. The Bayesian optimizer interfaces with the pipeline executor, suggesting hyperparameter sets to evaluate given scores for previous sets. Within the Bayesian optimizer, the Smart Search algorithm provides previous hyperparameters H and associated scores $g(h)$ to the GP or GCP model, receiving an estimate $\hat{g}(h)$ of the scoring function in return. Finally, the pipeline executor returns the tuned pipeline and (optionally) tested hyperparameters and associated performances. In this figure, the labels are as follows: (1) corresponds to the hyperparameter ranges; (2) to the (custom) pipeline steps; (3) to the tested hyperparameters and associated performances; (4) to the pipeline with the best hyperparameter set; and "P + D + Params" to the scikit-learn Pipeline object, data, and BLB hyperparameters. 36

9-1 Experimental averages for the HOG image pipeline. 92

9-2 Experimental averages for the CNN Image pipeline. 93

9-3 Experimental averages for the traditional text pipeline. 93

List of Tables

2.1 (Approximate) Classification of current state-of-art Hyperparameter Systems. **BB** refers to the type of **api** or the black box function method they use. `BB`= 1 implies that the api is as shown in Figure 2-1 and `BB`= 2 implies that the system has api as shown in Figure 2-2. . 28

8.1 Arguments for `d3m_load_data` function. 79

8.2 Arguments for `DeepMine` function. R= Required?, D = Default . . . 81

9.1 Types of pipelines. 89

9.2 BLB and non-BLB processes statistic. 94

B.1 Deep Mining Functions 102

B.2 Classification of Pipeline Steps 103

Chapter 1

Introduction

Data science as an endeavor is based around the goal of generating insights and predictive models from data. When given data along with an analytical or predictive problem, data scientists develop an end-to-end solution, processing the data over numerous steps, including preprocessing, feature extraction, feature transformation, and modeling, until a solution is achieved. At each step, they choose which functions to apply, along with associated hyperparameters. They make these choices through a trial-and-error process using quantitative metrics and intuition from prior experience, with the goal of producing either more useful analytical results or more accurate predictions.

Data scientists' solutions add value to an enterprise, and they are rewarded for their skill and experience in making these decisions. As the industrial and academic need for data-driven solutions grows along with the computational power and data resources available, the number of problems to be solved far outweighs the number of data scientists available to solve them. At the same time, for any given data problem, the number of choices available for each step and the overall complexity of pipeline is increasing exponentially [1], producing a space of possible solutions that is typically too large for a data scientist to understand and explore.

[1] For example, deep learning now provides solutions for image problems competitive with (and in many cases, better than) traditional HOG or SIFT based feature engineering. Many newer versions of deep learning models are also emerging, but developing these models may require tuning a number of hyperparameters.

Given these two problems: (1) the increasing supply of data science problems and (2) the growing suite of processing functions and rising complexity of pipelines, data scientists face these two challenges:

- **Making modeling choices**: How to pick the best functions for each of the processing steps? How to specify the hyperparameters for each step? Decisions corresponding to each step interact in ways that are often not apparent to the data scientist, and the resulting choices often fail to take advantage of potential performance improvements from their coupling.
- **Incorporating the latest tools**: In every area of data science, new methods and software packages are being introduced on a daily basis. These tools are not standardized, and incorporating a new method may entail significant amounts of exploration and software engineering effort before it can be integrated into a workflow.

The promise of automation: Given these challenges, we ask: how can we enable or augment existing data scientists? Can some of their work be automated? Given data and a problem, could we create a better system that chooses appropriate preprocessing functions, constructs a pipeline, and tunes the pipeline, trying different hyperparameters as a data scientist would?[2] This automated solution could even provide a baseline for a human data scientist to build upon. In the next section, we examine the challenges we encountered when thinking about these questions, and how we address them through a unique system we call *Deep Mining*.

1.1 Automating what a data scientist does

To automate the data science process, we must consider the entire sequence of steps involved in generating a machine learning model. This process includes loading data, a variable number of data transformation and feature extraction steps, model construction, and prediction. Figure 1-1 illustrates a data science pipeline, where the

[2] We present a detailed description of a realistic data science problem in Chapter 3.

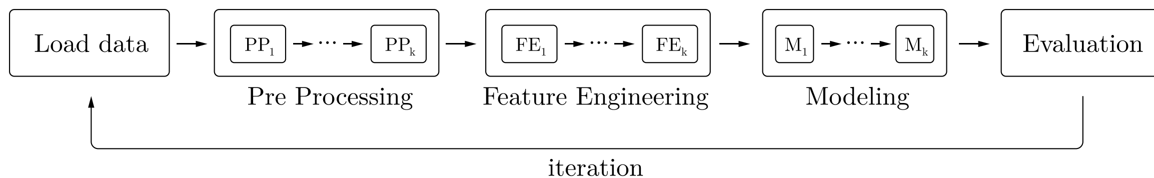

Figure 1-1: End-to-end pipeline to build a predictive model. It includes the preprocessing and feature extraction steps along with the modeling steps.

sequence of steps is such that the input of step i is the output of step $i - 1$. We define the term *data science pipeline* as this entire process, from inputting raw data to constructing predictive models.

The AutoML community: One group attempting to address this problem is the automated machine learning (AutoML) community, which is working to automate traditionally manual machine learning decisions in order to both reduce data scientists' efforts and improve ultimate solutions. While a detailed overview of these methods is included in Chapter 2, we highlight major points here. Work in this field can be divided into two main (and often overlapping) themes: 1) design of the search algorithm, and 2) open source software development and release.

Modeling step as primary focus: This community also focuses chiefly on the final stages of the pipeline: model selection and hyperparameter tuning. A *hyperparameter* of a machine learning method is distinct from the parameters learned during the training process. While training parameters include learned parameters such as the coefficients of a linear regression, a hyperparameter is a value set before the model fitting begins. A hyperparameter can be a categorical (e.g., a kernel used in a Support Vector Machine), integer (e.g., a degree of polynomial basis function), or continuous value (e.g., step size in stochastic gradient descent), and some hyperparameters are available only when other hyperparameters have been given particular settings. For example, the degree of a polynomial kernel for an SVM is not available with a radial basis function kernel. Many methods of hyperparameter tuning have been proposed, and automated model selection remains an area of active research.

Search algorithm design: In pursuing the first set of goals, researchers focus primarily on reducing the number of hyperparameter sets to evaluate, because each test

of these multidimensional functions can be extremely expensive. To evaluate alternate search algorithm designs, this community constructed multidimensional black box functions on which to test search algorithms, and benchmark datasets on which to report system performance. Most benchmark datasets are arguably far removed from real world, industrial scale problems.

Data input representation: The AutoML community has typically focused on only a stylized subset of data science methods, either optimizing within the space of classifiers or regressors, or tuning only pipelines with featurized datasets in matrix format.

When building their systems, AutoML researchers typically either 1) treat the function to be optimized (model performance) as a black box, or 2) treat the model construction process itself as a black box, providing and choosing between data science pipelines outside of the user's view. Their system APIs focus either on improving the algorithm itself or providing out-of-the-box solutions to problems.

Real world data science problems, however, have characteristics that challenge these approaches:

- Realistically, most data scientists spend more of their time at the earlier stages of the pipeline, such as pre-processing and feature extraction, than at the modeling stage. Because decisions made at these stages can significantly impact accuracy and are often made in an ad hoc fashion, any automated solution should include the full end-to-end pipeline and test its efficacy on realistic problems.

- Because of the size of datasets in practice and the computationally intensive nature of earlier stages of pipelines, evaluating each pipeline can be computationally expensive, and multiple pipelines must be assessed for tuning.

- Including the feature extraction and preprocessing stages in automated solutions requires the incorporation of non-standard methods whose implementations do not have well-defined APIs. These methods can be hard to find and are often scattered across programming libraries.

- Realistic datasets include a combination of multiple data types, such as images, text, and relational data, and may have temporal components. This data cannot always be presented as a clean, labelled matrix, as is expected by most software systems.

1.2 Deep mining

In building *Deep Mining*, we address these challenges by taking a contrarian approach to both the theoretical and implementational aspects of AutoML.

Provide efficient computation in order to scale to realistic datasets: While Deep Mining uses sophisticated methods to reduce the number of hyperparameter sets evaluated, we also incorporate subsampling methods and distributed computation to *reduce the time needed for the evaluation of each hyperparameter set.* Rather than improving the algorithms supplying hyperparameter sets to be evaluated, our theoretical work in this thesis focused on reducing the time needed for the evaluation of each hyperparameter set.

Provide immense flexibility to design an arbitrary custom pipeline: From a systems perspective, we neither treat the data science pipeline performance as a black box nor deny users access to their choice of pipelines. Rather, we provide a simple API that specifies custom data science pipelines for a variety of data types, enabling domain experts to both use existing data-specific feature transformers and contribute their own. In both cases, we "open the black boxes" to take advantage of significant performance improvements overlooked in traditional literature and systems.

The goal of the Deep Mining system is not to pursue superior out-of-the-box performance on existing small-data benchmarks, but to allow users the flexibility to implement custom pipelines that are appropriate for specific applications and to provide the distributed computation structure with sampling methods that allow them to apply these pipelines to substantial datasets. Potential applications include the Human Vision and Biometrics communities, which employ pipelines with complex, custom feature construction methods.

Provide flexibility of inputs: In addition to providing a distributed, open-box system, we also enable hyperparameter tuning in feature engineering methods by allowing data to be inputted not only in the traditional matrix structure, but also in raw data formats such as text, image, audio, and relational datasets. Existing hyperparameter tuning systems typically accept data only in matrix format, providing either X and Y matrices or matrices in a cross validation split. By enabling more costly data science pipeline steps to be tuned through a distributed framework, we facilitate the inclusion of feature processing methods that do not operate on data in this matrix format, but rather on raw data files. For example, the feature extraction tools provided by *Deep Feature Synthesis* [23] operate on data from *relational databases*. In Deep Mining, we use a structure based on the under-development D3M data format and build a parser to allow for the inclusion of datasets in non-matrix format.

The resulting system includes novel hyperparameter tuning techniques that use Copula processes, a distributed computation framework that incorporates Apache Spark and other tools, subsampling methods that dramatically improve the asymptotic complexity of the tuning process, an API that allows for the tuning of custom data science pipelines using raw datasets, and an open-source framework that includes the contributions of data science domain experts, statisticians, and systems programmers.

1.3 Contributions

The contributions of this thesis are as follows:

1. Explored the use of subsampling techniques and developed an algorithm using Bag of Little Bootstraps (BLB) for pipeline evaluation

2. Evaluated the effectiveness of BLB evaluation for data science pipelines, using multiple custom pipelines to empirically demonstrate that performance improvements are comparable to pipeline evaluations on the entire dataset

3. Designed and built an open source system for constructing and tuning arbitrary pipelines on large, non-matrix datasets by
    (a) enabling a structured approach to pipeline construction;
    (b) providing a distributed system capable of running multiple parallelization techniques for hyperparameter optimization of entire pipelines on local or cluster computing resources;
    (c) establishing clear abstractions for incorporating a variety of data types and for contributing across application areas.

1.4 Previous Work

Previous work done by Sébastien Dubois on Deep Mining involved implementing the Gaussian Copula Process model described in section 4.2, as well as much of the search framework for black-box function tuning. In this system, users simply specified a scoring function along with function parameters and ranges. This framework treated the pipeline scoring as a black box, proposing new parameter sets to the scoring function, receiving function values in return, updating the Bayesian model, and proposing new hyperparameter sets according to the chosen acquisition function. This work established the viability of the GCP model for modeling hyperparameter spaces for tuning.

My initial work on the project included documenting and editing this existing codebase, which was functional and included examples of tuning functions such as the Branin and Hartmann 6D, as well a Random Forest Classifier using the black box function format.

1.5 Outline

In this thesis, I will

1. Examine the existing hyperparameter optimization community, detailing the

search algorithms and meta-modeling approaches used as well as any currently available tuning systems;

2. Describe Gaussian Processes and their extension, Gaussian Copula Processes, for modeling the space of hyperparameters;
3. Give an overview of the goals of the Deep Mining project and the sampling approach used to speed pipeline evaluation;
4. Describe work done on the Deep Mining system prior to this thesis;
5. Give a structure for composing arbitrary pipelines using utilities from scikit-learn;
6. Describe the Bag of Little Bootstraps algorithm for sampling-based estimation;
7. Detail parallelization methods and frameworks used in Deep Mining;
8. Illustrate the use of the Deep Mining system through examples using the current API;
9. Present results from experiments with the use of sampling-based pipeline evaluation in hyperparameter optimization;
10. Provide a discussion of future work and possible implications.

Chapter 2

Background and Related Work

Today's machine learning algorithms provide insights in a number of diverse fields, from computer vision to recommender systems. However, these algorithms have hyperparameters that need to be set before training a model, the values of which can drastically affect performance. This distinction has spawned an entire discipline within data science that seeks to find the most effective values for these hyperparameters.

Consider a function $g(\mathbf{h})$, where the function's output is a measure of how well a machine learning pipeline is performing on the data. This output is usually measured using a scoring function that measures the accuracy of the model on the data. The machine learning method/pipeline has certain hyperparameters, $\mathbf{h}$ [1]. The goal of hyperparameter optimization is to find the settings of the vector $\mathbf{h}$ that maximize (minimize) the scoring (loss) function $g(\mathbf{h})$. Subject to a specified range R for $\mathbf{h}$:

$$\begin{aligned} &\underset{\mathbf{x}}{\operatorname{argmin}}\; g(\mathbf{h}) \\ &\textit{subject to} \;\; \mathbf{h} \in R \end{aligned} \tag{2.1}$$

The hyperparameter optimization community has been extremely active in recent years, providing novel algorithms and systems to accelerate the automated model

[1] These are called *hyperparameters*, and they are distinct from the parameters learned during the model training process.

selection and tuning process. Generally, the setup is as follows:

A search algorithm is given a multi-dimensional, black box function, $g(\mathbf{h})$ [2] and ranges for each input dimension h. A *hyperparameter* set consists of values for each of the input dimensions, which correspond to the different hyperparameters. Given this set of inputs, the black box function is called to give a resulting score. The goal is to find $\underset{\mathbf{h}}{\operatorname{argmin}}\ g(\mathbf{h})$ in the shortest possible time, which researchers frequently equate with the lowest number of hyperparameter set evaluations. Often, researchers then build systems on top of these algorithms, either querying the evaluation of the black box function in a style similar to active learning, or choosing the function(s) themselves by implementing different machine learning methods.

In this chapter, we will describe the following areas:

- **Search algorithms**

  Current mathematical approaches to hyperparameter tuning, including conventional search algorithms and popular meta-modeling techniques.

- **Types of black box functions tackled**

  How many different types of machine learning problems/pipelines have been addressed so far

- **Existing optimization systems**

  Current hyperparameter optimization systems, including their choices of search algorithms, meta-modeling techniques, $g(\mathbf{h})$, and apis.

2.1 Search Algorithms and Meta-modeling

Data scientists may choose hyperparameters manually, using a combination of guesswork and prior experience and allocating considerable time to the optimization of individual methods. Alternatively, they may adopt a more exhaustive, systematic

[2] The entire machine learning pipeline is encapsulated in this black box function, returning the score of the pipeline as output.

approach, choosing to search through every possibility in the hyperparameter space. This space can be visualized as a "grid" of sorts, with each hyperparameter comprising one of the grid's axes. Note that in this and many search frameworks, hyperparameters with infinite possibilities must be constrained to specific ranges. *Grid search*, then, explores every possibility in this constrained space by evaluating the pipeline over all of the hyperparameter sets in the grid. For machine learning applications, where the data and hyperparameter space can be quite large, this method quickly becomes intractable, as a particular hyperparameter set can take hours or even days to evaluate. Even with sparse grids and few hyperparameters, the space can include thousands of possibilities, each of which can take a non-trivial amount of time to test. This complexity translates to an extremely high cost, providing the motivation for reducing the number of points in the grid that must be evaluated in order to achieve best possible score in a given time.

One alternative to this search method is a *random search* [3] in which data scientists explore the hyperparameter space by choosing hyperparameter combinations at random until sufficient coverage of the grid is reached or a time budget is expended. This random search has been shown to significantly improve on a grid search, dramatically decreasing the number of evaluations of hyperparameter sets needed by simply sampling uniformly from the hyperparameter space.

What if there were a way to more intelligently select regions of the hyperparameter space to explore? Rather than simply choosing points in the grid at random, *adaptive search* processes seek to identify and explore regions of the hyperparameter space that promise to improve the objective at hand by taking previous hyperparameter evaluations into account.

2.1.1 Meta-modeling Approach

To identify promising regions of the hyperparameter space, various *meta-modeling* approaches have been introduced. In these approaches, a mathematical model of the space is constructed and used to estimate the score on candidate hyperparameter sets, without actually evaluating the actual pipeline on the data. These heuristics make

a trade-off between exploration and exploitation in choosing hyperparameter sets to evaluate next.

One such approach is *Bayesian optimization*, which treats the black box function, a scoring function of the hyperparameters, as an unknown. Bayesian optimization places a prior over the black box function that captures beliefs about its behavior, updates it with observations, and uses the resulting model and a chosen acquisition function to choose the next set of promising points in the space. The next set is tried by setting the hyperparameters for the pipeline, executing the pipeline, and generating the score. This evaluation results in new data points that can be incorporated to update the prior to form a posterior distribution, which is then used in the construction of an acquisition function and determines which part of the hyperparameter space to explore next. We discuss the *Gaussian Process* (GP), and the associated *acquisition functions* used for this type of hyperparameter optimization, in section 4.1 [5, 39].

Hyperparameter optimization is an active research field and has incorporated many other methods of modeling and exploration. This summary is not intended to give a comprehensive list of hyperparameter tuning methods, but rather a coarse overview of current methods. Multi-armed bandit methods have been used to model the space of different hyperparameter sets [26]. The Tree of Parzen Estimators has been used, along with the Expected Improvement acquisition function, modeling the posterior indirectly using $p(x|y)$ and $p(y)$ rather than modeling $p(y|x)$ directly as in Gaussian processes [5]. Reinforcement learning has been used on neural network architectures [2], and gradient descent has been shown to be effective for some continuous hyperparameters [27]. The radial basis function has also been used [12], as well as a spectral approach that improves on the asymptotic complexity of the GP fitting process [19]. Multi-task Gaussian processes have been applied to Bayesian hyperparameter optimization to incorporate information from previous optimizations [41], and transformations have been applied to construct a more flexible prior for the Bayesian optimization [40].

2.2 Choices of $g(\mathbf{h})$

A data science pipeline involves many different steps in addition to the machine learning method used to learn a model. Throughout this thesis, we make a distinction between *machine learning pipelines* and *data science pipelines* that is not often made explicit in existing literature. Consider a pipeline consisting of only a Support Vector Machine (SVM) classifier to be tuned. 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 pipeline, the term is misleading because the method consists of only one step. Extending one step further, the pipeline may also incorporate Principal component analysis (PCA), scaling the feature matrix in addition to SVM. However, we would still call this a machine learning pipeline.

In practice, data science pipelines extend to earlier steps in addition to what machine learning pipelines entail. For example, they typically include preprocessing and feature extraction steps such as Bag-of-words (for natural language) or HOG feature extraction (in case of images).

With that distinction, we can classify different systems as either those focused on machine learning pipelines and those that can be extended to entire data science pipelines. It is worth noting that the kind of pipelines they incorporate has implications on the data domains and representations they can address.

2.3 Current Hyperparameter Optimization Systems

Given the varied choices of $g(\mathbf{h})$ and search algorithms, in this section we will examine the currently available open source frameworks, noting their explicit and implicit choices. Hyperparameter optimization systems typically take one of two "black box" approaches. In the first case, they treat the performance of a machine learning 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 2-1 presents a schematic for this abstrac-

tion. Table 2.3 describes systems that use this framework, including Spearmint, the startup SigOpt, MOE, SMAC, BayesOpt, REMBO, and HPOlib.

Table 2.1: (Approximate) Classification of current state-of-art Hyperparameter Systems. **BB** refers to the type of **api** or the black box function method they use. `BB`= 1 implies that the api is as shown in Figure 2-1 and `BB`= 2 implies that the system has api as shown in Figure 2-2.

System	BB	Implementation	Paper
Spearmint	1	`https://github.com/HIPS/Spearmint`	Various
SigOpt	1	`https://sigopt.com/getstarted`	[11]
MOE	1	`https://github.com/Yelp/MOE`	[46]
SMAC	1	`https://github.com/automl/SMAC3`	[21]
BayesOpt	1	`https://github.com/rmcantin/bayesopt`	[28]
REMBO	1	`https://github.com/ziyuw/rembo`	[44]
HPOlib	1	`https://github.com/automl/hpolib`	[15]
Hyperopt	1	`https://github.com/hyperopt/hyperopt`	[4]
Hyperopt-sklearn	2	`https://github.com/hyperopt-sklearn`	[25]
Auto-WEKA	2	`http://www.cs.ubc.ca/labs/beta/Projects/autoweka/`	[42]
Hyperband	2	`https://github.com/zygmuntz/hyperband`	[26]
TPOT	2	`https://github.com/rhiever/tpot`	[33]
Auto-sklearn	2	`http://automl.github.io/auto-sklearn/stable/`	[17]
Osprey	2	`https://github.com/msmbuilder/osprey`	[29]
Optunity	2	`https://github.com/claesenm/optunity`	[9]
mlr	2	`https://github.com/mlr-org/mlr`	[6]
Scikit-optimize	Other	`https://github.com/scikit-optimize/scikit-optimize`	[16]

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

- 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 is what 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, requiring input data to be in a

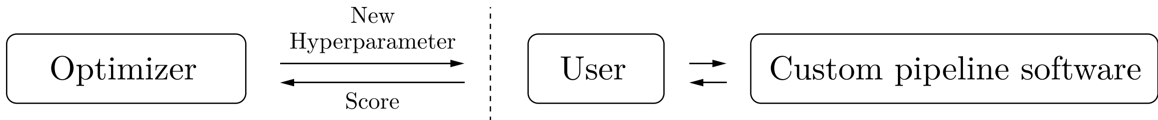

Figure 2-1: Illustration of the first common abstraction approach. The optimization algorithm gives hyperparameter sets to and gets performance metrics from the pipeline, which it treats as a black box.

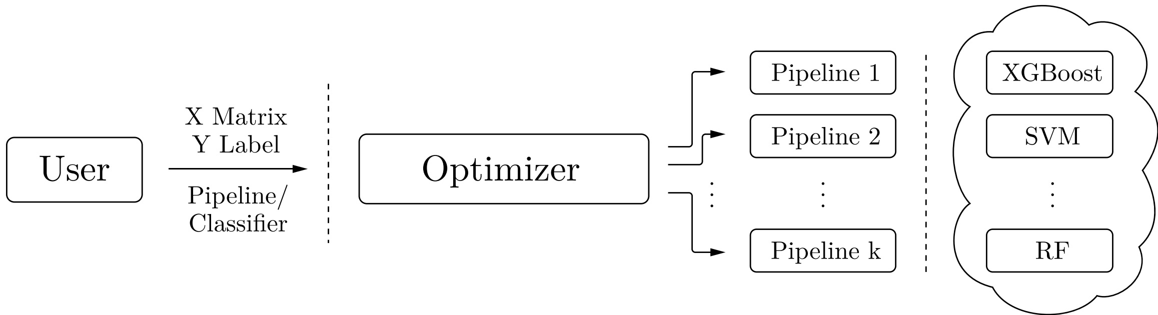

Figure 2-2: Illustration of the second common abstraction approach. The user provides data in a matrix format (either X, Y, or X and Y in a cross validation split). The optimizer chooses a pipeline from within its implemented machine learning algorithms, and returns a classifier or pipeline with the highest score to the user.

matrix format. Users of the system cannot include custom pipelines that they judge to be appropriate for their problem, and these systems constrain them by forcing them to choose among a set suite of classifiers or regressors, as well as the occasional feature preprocessing method. Figure 2-2 illustrates the schema for this abstraction.

These systems are effective in many small, matrix-formatted datasets, achieving impressive results on benchmark datasets such as MNIST. However, for applications on large datasets in non-matrix format (e.g., relational databases), these systems are less well-suited, and the lack of clear APIs to specify custom pipelines severely limits their effectiveness. Examples of systems in this style are noted in Table 2.3, and include hyperopt-sklearn, Auto-WEKA, Hyperband, TPOT, Auto-sklearn, Osprey, Optunity, and mlr. Scikit-optimize provides an API for potentially arbitrary pipelines using scikit-learn's Pipeline interface, but the system's lack of subsampling, parallelization frameworks like Apache Spark, and incorporation of raw data limits its effectiveness.

Chapter 3

Deep Mining: Overview

To explain the considerations motivating the Deep Mining system, we consider a problem faced by a data scientist in practice: face recognition. In this situation, the data scientist is provided images of a variety of faces and has the goal of predicting to which person a given facial image belongs. Assuming all of the hyperparameter optimization systems from section 2.3 are available, this process consists of the following steps:

1. **Compose a pipeline**

   First, a data science pipeline is chosen. This process includes choosing preprocessing and feature extraction methods as well as a classifier [1]. In the first stage of this pipeline, the images are put through *photometric normalization* using one of four methods: contrast limited adaptive histogram equalization (CLAHE), the multi-scale retinex algorithm, the discrete cosine transform (DCT) algorithm, or the single scale self quotient (SQI) image algorithm. In the second stage of this pipeline, features are extracted using local binary patterns (LBP), local phase quantization (LPQ), histograms of oriented gradients (HOG), and binarized statistical images (BSIF). Finally, these features are consolidated using Principal Component Analysis (PCA) or Latent Dirichlet Allocation (LDA),

[1] The following pipeline example was provided by Thomas Swearingen and Dr. Arun Ross of Michigan State University's i-PRoBe Lab and has been incorporated into Deep Mining. We want to thank Thomas and Dr. Ross for their help in implementation and testing.

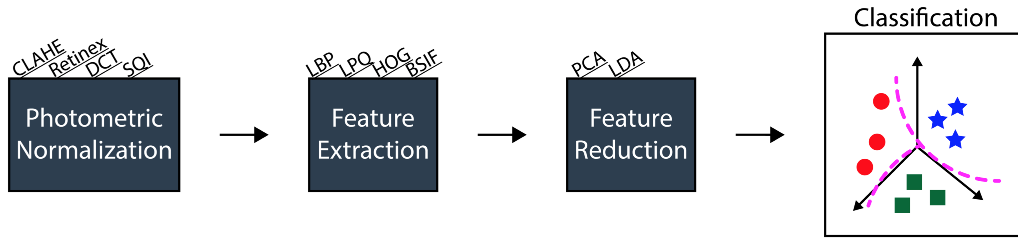

Figure 3-1: Illustration of the portion of the facial recognition pipeline to optimize.

and either a Random Forest Classifier or Support Vector Machine (SVM) is applied to the data.

2. **Implement the chosen pipeline**

   Next, the data scientist must implement this pipeline by constructing its custom steps, writing wrapper functions for feeding the intermediate transformations to subsequent pipeline steps, fitting classifiers, and outputting the eventual score. This process involves either writing the code from scratch or copying code for the desired pipeline steps from other libraries. Both choices may involve significant debugging in both constructing the steps and placing them inside a pipeline.

3. **Import dataset**

   After this implementation, the data scientist must import this dataset in a format suitable for the constructed pipeline.

4. **Specify hyperparameters and ranges**

   Once the pipeline has been chosen, the data scientist must ensure that the hyperparameters for each of the step are exposed. The data scientist also chooses possible ranges for each of the hyperparameters, confining the search within those bounds.

5. **Tune hyperparameters**

   The data scientist can now tune the hyperparameters of the pipeline, using either a manual search or an existing hyperparameter tuning system. In this

example, because the pipeline has custom steps and includes more than conventional feature transformations (e.g., PCA, LDA) and classifiers, the data scientist can use a system in the "black box function" paradigm, in which he provides a scoring function interface and tests inputs suggested by the hyperparameter optimizer.

6. **Speed up pipeline evaluation for hyperparameter tuning**

   The hyperparameter optimizer requires scores for dozens of hyperparameter sets in order to build a meta model and tune the whole pipeline. Because the pipeline is computationally intensive and the dataset is large, evaluation of each of these sets can take hours. In order to find a solution in a reasonable time, the data scientist often implements a parallelization framework for simultaneous hyperparameter set evaluation by launching multiple worker machines (say, on an Amazon cluster) and writing supporting software.

   If the pipeline evaluation remains too slow, the data scientist implements sampling techniques to further speed up this process, getting approximate scores for hyperparameter sets by using only parts of the dataset. He chooses these parts arbitrarily, taking random subsamples of the original data.

7. **Output predictions for new data**

   Finally, the data scientist receives the highest-scoring hyperparameter set and trains a model with those hyperparameters, altering his existing pipeline code if necessary. He then outputs his predictions on new data, either noting the pipeline performance on this new data or using these newly formed predictions in production.

This process illustrates a number of the difficulties that data scientists will face even with the widespread availability of hyperparameter tuning systems. Lack of automation and support in each of these steps makes for a labor-intensive and error-prone process. Constructing a pipeline is difficult without examples from domain experts for similar problems, and importing raw data into the matrix format required

by many machine learning methods takes significant time. Many existing hyperparameter optimization systems are entirely unsuited to this particular domain-specific problem, and the black box function API provides no support for pipeline specification. Real-world problems require efficiently evaluating hyperparameter sets, and the data scientist may spend significant time implementing a parallelization and sampling framework to speed up the model evaluation process. Even after he has found the best hyperparameters in the space, using that information to train a model and make predictions requires additional software construction and debugging.

To solve this problem, we propose *Deep Mining*, a system for sequential hyperparameter optimization that scales to complex pipelines and large datasets and provides the necessary frameworks for solving the particular problems data scientists face. In this thesis, we also describe an alternative approach to accelerating hyperparameter optimization using a subsampling algorithm called the *Bag of Little Bootstraps.*

3.1 Sampling-based Approach for Hyperparameter Search

Much of hyperparameter optimization theory focuses on improving the models and algorithms exploring the hyperparameter space. In this framework, the focus is on reducing the number of hyperparameter sets evaluated. However, the ultimate goal of these automated model selection methods is to reduce the human and computational *time* it takes to optimize, and the number of model evaluations is only a proxy for that metric. The contrarian approach we outline in this work is to reduce the time needed for the evaluation of a single hyperparameter set. We reduce this time using two methods: 1) intelligent sampling, and 2) distributed computation. By focusing on reducing the time necessary for each model evaluation along with intelligently choosing hyperparameter sets, Deep Mining significantly improves the time needed for hyperparameter optimization.

Existing work related to this approach includes the Apache Spark MLLib package

[31], which executes a grid search of hyperparameters in a distributed fashion. Work has also been done in deep neural networks to extrapolate the learning curve in early epochs so as to stop model evaluations that are not promising [13]. Multi-task Bayesian optimization is also promising, as hyperparameter evaluations on smaller datasets are used to predict performance on larger datasets [41]. The effects of random sampling on hyperparameter optimization have also been explored [20].

3.2 The Deep Mining System

Deep Mining is an automated system that begins with raw or matrix-formatted data and ends with a tuned predictive model. Figure 3-2 illustrates the Deep Mining system, detailing its major components and their interactions, which aim to provide the functionalities neglected by existing hyperparameter tuning systems.

The end-to-end system in Deep Mining provides these essential functionalities: solving the difficulties that come with non-standard pipeline specifications and implementations, allowing importing of a variety of datasets, exposing hyperparameters, speeding up hyperparameter tuning, and aiding the use of pipelines in production. In this project, we explore these functionalities along with higher-level goals.

This project spans various research areas: meta- machine learning (by pursuing novel meta modeling techniques using copulas), systems engineering (by incorporating data and distribution frameworks to enable parallel computation), human-data interaction (by designing an API that enables domain experts to share, compose and tune novel/arbitrary pipelines), statistics (by incorporating sampling bootstrapped estimation), and ultimately artificial intelligence (by pursuing automation of steps otherwise performed by data scientists).

State-of-the-art model of the hyperparameter space: To provide a competitive hyperparameter tuning system and to efficiently select points in the hyperparameter space for evaluation, the model of the hyperparameter space and the resulting acquisition algorithm must improve upon existing methods. The Gaussian Copula Process provides this improvement by supplying a transformation to the Gaussian process

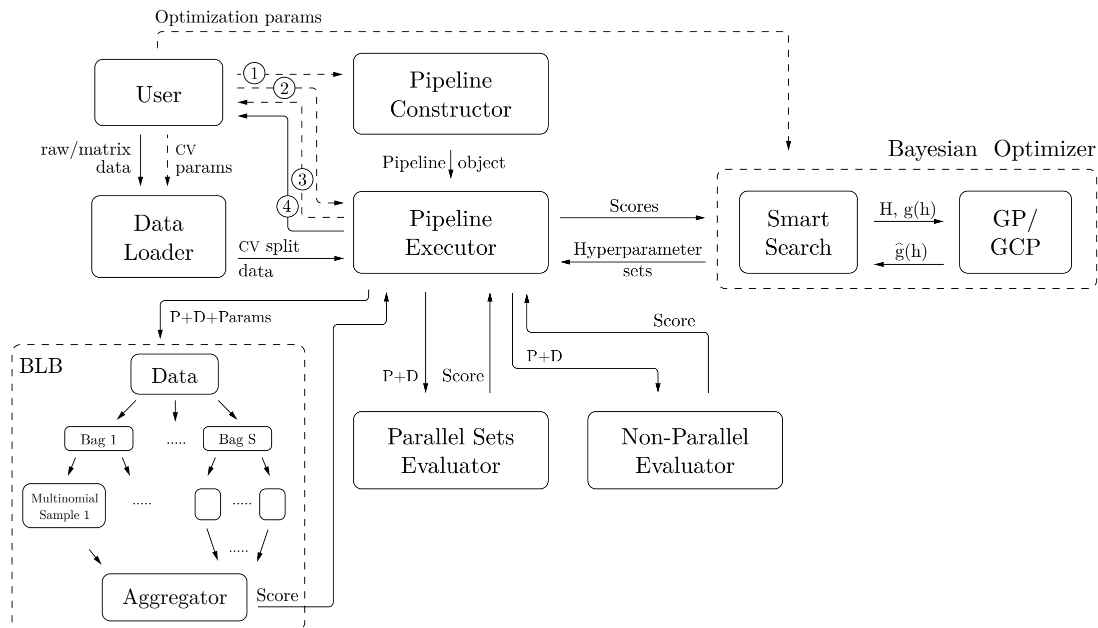

Figure 3-2: Illustration of the Deep Mining system's functionality. The user specifies a pipeline using the provided framework, providing hyperparameter ranges and the necessary pipeline steps to the pipeline constructor and executor. The user also provides the data in either raw or matrix format and cross-validation parameters to the data loader as well as optimization parameters to the Bayesian optimizer. Outside of the user's view, Deep Mining constructs a scoring function using the provided pipeline and parameters, incorporating distribution and sampling to accelerate the evaluation process. Deep Mining also loads the data into a cross validation split for the pipeline executor, which chooses between different parallel and non-parallel evaluation methods that return a score to the pipeline executor. The Bayesian optimizer interfaces with the pipeline executor, suggesting hyperparameter sets to evaluate given scores for previous sets. Within the Bayesian optimizer, the Smart Search algorithm provides previous hyperparameters H and associated scores $g(h)$ to the GP or GCP model, receiving an estimate $\hat{g}(h)$ of the scoring function in return. Finally, the pipeline executor returns the tuned pipeline and (optionally) tested hyperparameters and associated performances. In this figure, the labels are as follows: (1) corresponds to the hyperparameter ranges; (2) to the (custom) pipeline steps; (3) to the tested hyperparameters and associated performances; (4) to the pipeline with the best hyperparameter set; and "P + D + Params" to the scikit-learn Pipeline object, data, and BLB hyperparameters.

system that allows for the flexible marginal distributions appropriate for models of the hyperparameter space. We describe this method in detail in Chapter 4.2.

Sampling algorithm to accelerate model evaluation: We use a model selection algorithm derived from *Bag of Little Bootstraps*, a sub sampling based approach that evaluates functions of data (over sub samples) with the same statistical guarantees as the traditional bootstrap. The Bag of Little Bootstraps algorithm provides a significant asymptotic improvement to the evaluation of hyperparameter sets for a variety of algorithms by taking advantage of weighted representations of data. We experiment with using this system to show that even with these asymptotic improvements, the improvement in model performance from hyperparameter tuning remains comparable. We describe this method in detail in Chapter 6.

Distributed computation for parallel execution: Executing pipelines in a parallel, distributed fashion significantly speeds up the evaluation process. By using frameworks including Apache Spark, we can compute either the pipeline score on multiple bootstrap in parallel or compute scores for multiple different pipelines in parallel. Deep Mining provides hyperparameter optimization at the scale demanded by modern applications. We describe our parallelization framework in detail in Chapter 7.

Abstractions for arbitrary datasets and pipelines: Deep Mining offers an alternative to the black box approaches used by existing hyperparameter systems by allowing composition of arbitrary pipelines and allowing ingestion of different data types. Using the Pipeline interface from scikit-learn [35] and a custom API, users specify arbitrary pipelines operating either on matrices (as in other systems) or on the raw data itself, seamlessly enabling the tuning of feature engineering methods that are unavailable in other auto-tuning systems. We describe how to compose arbitrary pipelines using our api in Chapter 5.

As a consequence of this approach, we do not seek out performances on benchmark datasets to show the superiority of our tuning algorithm or machine learning method suite. Instead, we enable pragmatic applications such as face recognition. By using a standardized format for the storage of raw data, Deep Mining allows for the tuning of

feature extraction methods that do not operate on data in matrix format and provides utilities for using the tuned model in production.

Open-source software for collaborative data science: By providing abstractions for constructing and combining custom pipeline steps, Deep Mining incorporates the input of various domain experts. Domain experts can contribute to the library by providing custom pipelines and software for data transformations. Systems programmers can contribute in tuning the distributed frameworks used, and statisticians can improve existing copula processes and subsampling methods.

Chapter 4

Gaussian Copula Processes

In this chapter, we will describe the *Gaussian process* (GP), which is often used to model the hyperparameter space, and introduce *Gaussian Copula Processes* (GCP) as an improvement on traditional GP, detailing algorithms used by Deep Mining in hyperparameter search. The author acknowledges Kalyan Veeramachaneni and Sébastien Dubois for their development of and help in describing the GCP method [14].

In this chapter, we use x as the input vector, y as the outcome that GP or GCP is trying to model, and f as the unknown function that relates $f(x) \leftarrow y$. In the context of hyperparameter tuning they are h, y, and g respectively.

4.1 Gaussian Processes

The *Gaussian Process* is a non-parametric method that allows for inference over the continuous space of functions. The chief insight it provides for tuning purposes is that it allows for the construction of a Bayesian model of the hyperparameter space that updates with additional observations and is not confined to a specific functional form. Much of the background in this section is based on work in [36], which can provide additional detail.

Definition 4.1.1. Gaussian Process (GP) A collection of random variables, any finite combination of which has a multivariate Gaussian distribution.

A GP can be completely defined by its mean and covariance functions, which are functions of (multidimensional) locations in the input space. The mean function at a location $\mathbf{x}$ in the space is:

$$m(\mathbf{x}) = \mathbf{E}[f(\mathbf{x})] \tag{4.1}$$

and the covariance function is:

$$k(\mathbf{x}, \mathbf{x}') = \mathbf{E}[(f(\mathbf{x}) - m(\mathbf{x}))(f(\mathbf{x}') - m(\mathbf{x}'))] \tag{4.2}$$

The mean function is the average of the known function f values and is typically taken to be zero, assuming centered data. A typical covariance function used is the *Squared Exponential (SE)* covariance function:

$$cov(f(\mathbf{x}), f(\mathbf{x}')) = k(\mathbf{x}, \mathbf{x}') = exp(-\frac{1}{2}|\mathbf{x} - \mathbf{x}'|^2) \tag{4.3}$$

The function in the hyperparameter search case is the model performance (e.g., accuracy) or loss, and the Gaussian process is used as a prior for that function over the hyperparameter space. Predictions in this space are made using a posterior distribution that has been conditioned on hyperparameter sets $\mathbf{x}_1, ..., \mathbf{x}_N$ and their associated performances $f(\mathbf{x}), ..., f(\mathbf{x}_N)$. The posterior distribution is used to evaluate *acquisition functions* for a number of candidates in the space, and the value of these acquisition functions is used to determine which hyperparameter set should be evaluated next, based on the data and the machine learning method to be tuned. I describe those acquisition functions used in Deep Mining in Section 4.2.

The computational complexity of using a GP for hyperparameter optimization becomes clear upon examination of the expressions for the predictive mean and variance. The predictive mean is then

$$K(X_*, X)[K(X, X) + \sigma_n^2 I]^{-1} y \tag{4.4}$$

where $K(X, X)$ is the *kernel matrix* for all previous observations, $K(X*, X)$ is the kernel matrix between the points to be predicted $X*$ and the observations X, σ_n^2 is

the observation noise (assuming $y = f(\mathbf{x}) + \epsilon$), I is the N x N identity matrix, and y is the vector of previous observations.

The predictive variance is

$$K(X_*, X_*) - K(X_*, X)[K(X, X) + \sigma_n^2 I]^{-1} K(X, X_*) \tag{4.5}$$

The asymptotic complexity of this Gaussian process, then, is dominated by the inversion of the N x N kernel matrix. This results in an overall complexity of $O(N^3)$, where N is the number of (hyperparameter set, performance) observations. More detail on the Gaussian Copula Process and other aspects of this thesis can be found in the related published paper [1].

4.2 Gaussian Copula Process (GCP)

The Gaussian Copula Process, introduced in [45], is a prior based on a GP that can more precisely model the multivariate distribution of $f(\mathbf{x})$. A mapping $\Psi : \mathcal{Y} \rightarrow \mathcal{Z}$ transforms the output of f into a new variable z. We define a new function g, $g : \mathcal{X} \rightarrow \mathcal{Z}$, and a combination of f and Ψ given by $g(x) = \Psi(f(\mathbf{x}))$; and we model g with a GP.

By doing this, we actually change the assumed Gaussian marginal distribution of each $f(x)$ into a more complex one. This is because the Gaussian prior on $g(x)$ yields the prior for $f(\mathbf{x})$ given by the following cumulative distribution function:

$$F(y) = \Phi(\Psi(y)), \tag{4.6}$$

where $F(y) = \mathbb{P}(f(\mathbf{x}) \leq y)$ and Φ is the standard univariate Gaussian cumulative distribution function.

So far in the literature [45, 38], a parametric mapping is learned so that $g(x)$ is best modeled by a Gaussian Process. In [45], the authors propose to parameterize

Ψ^{-1} by $\{a_j, b_j, c_j\}$ such that:

$$\Psi^{-1}(z; \{a_j, b_j, c_j\}_{j=1}^{K}) = \sum_{j=1}^{K} a_j log\big(e^{b_j(z+c_j)} + 1\big), \tag{4.7}$$

with $a_j, b_j > 0$. The authors are interested in predicting the values of a positive function f. In the general case, we can add another variable m:

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

where $m, a_j, b_j > 0$. For $K = 1$ we then have :

$$\Psi(y) = \frac{log(e^{\frac{y+m}{a}} - 1)}{b} - c, m, a, b > 0. \tag{4.8}$$

However, this mapping is unstable in practice: we found that over many trials on the same dataset, different mappings were learned. Moreover, the induced univariate distribution for $f(\mathbf{x})$ was almost Gaussian most of the time, and the parametric mapping did not offer great flexibility. We see in eq. (4.7) that for $K = 1$, if $bz >> bc, 1$, then $\Psi^{-1}(z; a, b, c) \sim abz$, *ie.* the mapping is linear, and the GCP is actually a GP.

Given this observation, we introduce a novel approach where a marginal distribution is learned from the observed data through kernel density estimation [37] of F. After this, the mapping Ψ is numerically computed from equation (4.6), so that the observations of the training data $g(x_{t,i})$ have a Gaussian distribution:

$$\Psi(y) = \Phi^{-1}(F_{est}(y)). \tag{4.9}$$

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)

The prior mean of a Gaussian process is usually fixed as the empirical mean of the observations $f(x_{t,i})$. In the context of hyperparameter optimization, however, one can imagine that there should be some region where the hyperparameters would be *rather good* and others where they would be *rather bad.* For this reason, it may be convenient to set a different mean for the prior, depending on the region in which the hyperparameter is. When it comes to GP, researchers assert that this alteration would have little impact, as the covariance function is already meant to induce this smoothness. With GCP, however, we not only fix the mean function, but the mapping function as well. Because the mapping function reflects the distribution of the data, with nGCP a latent model aims at learning several distributions of $f(\mathbf{x})$ over the input space. In particular, this change may facilitate the location of promising regions to explore in a Bayesian optimization process.

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

To design the nLGCP prior, we look for a mapping function that depends on the input x *and* output $f(\mathbf{x})$. To construct this function, the training data $\{(x_i, f(x_i))\}$ are clustered in $\mathcal{X} \times \mathcal{Y} \subset \mathbb{R}^{m+1}$ using *K-means*. For each cluster k, a mapping Ψ_k is learned, as described in the previous section. Then, for each x in $\mathcal{X}$, the final mapping Ψ is computed as

$$\Psi(x, y) = \sum \alpha_k(x).\Psi_k(y), \tag{4.10}$$

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

Predictions with nLGCP

Predictions with GP are straightforward given a posterior, but this calculation is no longer simple with nLGCP. A faster but approximate way to compute the predicted value of $f(x_*)$ for a given x_* is to calculate g_*, the standard GP prediction of the warped output $\Psi(f(x_*))$ given by the posterior: $g(x_*) \sim \mathcal{N}(\mu_*, \sigma_*)$, Noting from the equation (4.6) that the predicted cumulative distribution function of $f(x_*)$ is $F_* = \Phi(\Psi; \mu_*, \sigma_*)$, we can evaluate the prediction as:

$$f_* = \int_{u=-\infty}^{\infty} u.\Psi'(u).\phi(\Psi(u); \mu_*, \sigma_*).du \tag{4.11}$$

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

The particular expression of Ψ in eq. (4.9) for the nGCP prior finally enables us to express directly its derivative:

$$\Psi'_{nGCP}(y) = \frac{d_{est}(y)}{\phi(\Psi(y))}, \tag{4.12}$$

where ϕ and d_{est} are respectively the probability density function of the standard univariate Gaussian and the one corresponding to F_{est} defined in Section 4.2. We can calculate the derivatives for Ψ_{nLGCP} similarly.

Chapter 5

Composing arbitrary pipelines

An overarching goal of this work is to enable the tuning of an entire pipeline, which includes preprocessing, data transformations and feature extraction. 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 domain or problem as well as specifically developed to mitigate issues in data collection. Because of this variety, developing a fixed set of transformations a priori is impossible. These steps also accept a variety of inputs, such as data in non-matrix formats, that many existing tools cannot incorporate.

- **Unstructured process**: Unlike software for machine learning algorithms, software for these transformations are written by domain experts, who do not in general construct the code for this part of the process in a structured and uniform way. This lack of structure and uniformity inhibits code sharing and slows the development process.

A good pipeline specification must also be easy to use, in order to encourage adoption and pipeline experimentation, and modular, allowing data scientists to combine different components quickly.

5.1 Abstractions for data transformation steps

Pipelines consist of series of data transformation steps ending with a modeling step in which a machine learning model is trained and evaluated. We call the steps that transform the data Transformers and the step that trains and evaluates the model an Estimator [1]. To be able to develop generalized abstractions for programmatically defining transformers, we categorize them based on their input, output, and type of computation involved.

Type of computation: Transformers can be classified into two groups:

- **Direct methods**: These methods allow transformations on data to be computed using only the function and its hyperparameters without requiring any learning of parameters from the data itself. An example of this type of method is patch extraction from a collection of images. For this category of methods, the applied transformation can be defined simply:

$$f(x_{old}) = x_{new} \tag{5.1}$$

  where x represents a single data point (a single image, a single text, or a feature vector), and the function f is applied to all data points in the entire dataset X.

- **Fitted methods**: Other transformers require parameters to be learned before transformations can be applied to the data. An example of this type of transformation is PCA, which must learn parameters from the structure of the data before applying its transformations. Two sub-types of learning exist within this methods:

  - **Fitting using individual data points**: In this category, transformer methods learn parameters from individual data points:

$$params = L(x_{old}) \tag{5.2}$$

[1] This is similar to the terminology used by a popular machine learning package, scikit-learn

then apply a transformation function using those parameters:

$$f(x_{old}, params) = x_{new} \tag{5.3}$$

– **Fitting using all data points**: Transformer methods can also learn parameters using the entire dataset and can be expressed as:

$$params = L(X_{old}) \tag{5.4}$$

then applying a function to the entire dataset using those global parameters

$$f(X_{old}, params) = X_{new} \tag{5.5}$$

PCA and Bag-of-words are examples of transformers that learn the parameters for the transformation function using all the data points.

Data input and output: Transformers vary in terms of the input data format they accept. Data domains broadly fall into four categories, which each require a uniform structure for designing transformers.

- **Image data**: For image datasets, regardless of the original data file type (e.g., jpeg, png), the raw image can be represented as a multidimensional array of numeric values corresponding to pixels. A flattened representation of the images represents each image as a row in a matrix. The intermediate transformations for images therefore typically map between matrices of floating point or integer values.

- **Text data**: For text datasets, the initial data can again be represented as a matrix, where each row corresponds to a different text document (data point). The data types within the matrix eventually change from strings to numeric values. Transformations for text data, then, fall entirely within the categories defined earlier in this section.

- **Relational Data**: Relational data undergoes the most stark transformation before eventual classification or regression. This type of data begins as a set of tables, which are eventually consolidated into a feature matrix over the course of the pipeline before presenting it to the classifier or regressor. Transformers for relational data have more complicated mappings than the typical matrix-to-matrix scheme, often going from "a set of tables" to a transformed "set of tables" or consolidating " a set of tables" into one. These transformations also often require meta data about data types of the variables in the tables and the relational structure between the tables. Combinations of these data types can also have this characteristic, as the consolidation of varied data sources into an eventual matrix format requires transformers with more complicated mappings. Once data is in matrix format, all transformer methods once again fall into the categories for image and text data.

- **Time series data**: In time series data, each data point is a collection of observation arrays along with an additional array corresponding to a time stamp, sequence number, or order number for the values in the other arrays. Transformers can output another time series given a time series or consolidate them into a vector.

5.2 Abstractions for the pipeline

Once these Transformer and Estimator pipeline steps have been specified as Python functions, a unifying framework is necessary to ensure that (1) intermediate outputs are correctly piped between steps (2) the pipeline can be used for training and prediction. Data scientists often construct a wrapper function that will take as input all hyperparameters for the pipeline and data and contains the end-to-end training pipeline code. This function outputs all of the "fitted" parameters across all steps. A separate wrapper function that takes as input the learned set of parameters and data is written to execute the pipeline given the learned parameters and produce predictions. This customized code is not transferable to a new problems, and constructing

these functions for each problem it is an error-prone and time consuming process.

A generalizable framework to solve this problem must allow users to:

- Easily compose the pipeline using only functions for each pipeline step without constructing extensive wrapper functions
- Effectively expose hyperparameters for all pipeline steps,
- Provide a simple command to train or "fit" the pipeline steps, saving fitted parameters and any meta data associated with the fitting process,
- Output a performance metric on a training dataset, and
- Provide a simple command to "predict" using the fitted pipeline.

5.2.1 Abstractions in scikit-learn

Scikit-learn, a popular machine learning software library, offers a powerful *fit-transform* abstraction that greatly simplifies the process we describe above. It has the previously mentioned two types of object: *transformers* and *estimators*.

Transformer objects transform the data, and must have *fit* and *transform* methods implemented, as the *fit* method allows them to learn parameters to be used for *transform*. Given this object-oriented approach, a data scientist can

- Call `transformer.fit_transform(X)`, which first "fits" using the data X and then outputs the transformed version of the data, X_{new}.
- Call `transformer.transform(X)`, which uses the fitted parameters from transforming X to output X_{new}.

Principal Component Analysis (PCA) is an example of a transformer that must fit itself before transforming the data. The *fit* method in PCA identifies the principal vectors in the data, which are then used to transform the feature matrix. All transformer methods in `scikit-learn` are written this way, enforcing uniformity across all functions.

Estimators, which make predictions using a fitted model, must implement *fit* and *predict* methods. Any machine learning classifier is an example of an *estimator*.

This fit-transform abstraction is also seen in various forms in other machine learning libraries, including Apache Spark's MLlib and Keras [8].

With these abstractions and provided library of functions, scikit-learn allows the user to do the following:

- **Create arbitrary pipelines**: A user can chain together multiple transformers, provided the last step in each pipeline is 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.

  $$\texttt{pipeline} = \text{Pipeline}(\texttt{[A, B,C]})$$

  Next, the pipeline hyperparameters are mapped to individual steps:

  $$\texttt{pipeline.set_params} = \{A__a1\text{: } v_{a1},\ B__b1\text{: } v_{b1},\ C__c1\text{: } v_{c1}\}$$

  where $a1$ is a hyperparameter for step A and v_{a1} is the corresponding name specified for the same in the overall hyperparameter dictionary. Deep Mining provides a wrapper to simplify this parameter setting, as detailed in section 5.4.1.

- **Integrate training and validation**: One of the advantages of setting up the pipeline object and mapping up the hyperparameters as shown above is that now users can call the same "fit" and "predict" functions provided for individual steps to execute the entire pipeline. This functionality eliminates the need for writing separate software for training and predicting.

– Calling

```
pipeline.fit(X_tr, y_tr)
```

will transform the data sequentially in the order the steps are specified in the `pipeline`. The `fit_transform` method is applied for each `transformer`, and finally estimator is fitted on the resulting transformed data.

– The fitted `pipeline` can then be evaluated on the validation data and used to make predictions:

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

predictions = pipeline.predict(X_v)
```

In both cases, X_v is transformed with the pipeline's intermediate steps – using the `transform` method with parameters fitted from the training process – and scored using the fitted estimator.

This example demonstrates the power of scikit-learn's pipeline abstraction, as once the pipeline has been constructed, only the *fit* and *score* methods of the Pipeline object must be called.

Custom Pipeline Steps in scikit-learn

A potential problem with the pipeline abstraction is its strict structure, as users would have to implement the *fit* and *transform* methods. However, scikit-learn provides the `FunctionTransformer`, `TransformerMixin`, and `BaseEstimator` classes to streamline this conversion process.

The `FunctionTransformer` object takes an arbitrary transforming function as an input and outputs 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.

5.3 Advantages of pipeline abstraction

This pipeline abstraction also effectively exposes the pipeline hyperparameters. For an arbitrary function A that uses the FunctionTransformer objects, the hyperparameters are simply the arguments of A in the user's implementation. For scikit-learn transformers and estimators, the hyperparameters are already exposed in the implementation.

This abstraction greatly simplifies the data science pipeline construction process, requiring only the implementation of the *set_pipeline* function and simplifying that implementation using familiar tools. This construction has many benefits for data scientists, including

- Standardizing the pipeline construction process, giving an intuitive template and enforcing discipline on 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,
- Clarifying the steps and hyperparameters used in a given pipeline,
- Providing a framework flexible enough to handle arbitrary transformers and estimators, and
- Enabling code-sharing through the use of a simple, modular template.

More generally, the pipeline construction is as described in Algorithm 1. Once the user has specified this function, Deep Mining calls the fit and $score$ functions as appropriate, incorporating sampling, cross validation, and information from other function arguments.

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
```

5.4 Pipeline Construction in Deep Mining

The *DeepMine* function executes hyperparameter optimization on the provided pipeline and data and exposes a variety of arguments. After loading the data into a Python object and splitting into training and validation sets, users specify the desired pipeline and optional parameters to execute a Bayesian tuning of their pipeline. This function uses the scikit-learn Pipeline object.

Defining a custom pipeline involves three steps:

1. **Constructing custom functions**

   In Deep Mining, arbitrary *transformer* functions can be included in the pipeline to be tuned. The only constraint on custom transformer functions is that the output of the specified transformer must be a valid input for the subsequent transformer or estimator step. When specifying these functions, users pass the input data as the first argument to the function with exposed hyperparameters as the other arguments, using default arguments as necessary. Using the FunctionTransformer object, X and Y parameters are passed to the *fit_transform* methods of these transformers, creating a transformed X output that is valid for the next step. When the *validate* parameter of the FunctionTransformer initializer is False, any Python object can be passed as X to the transformer

object's methods. While some cross-validation split into X and Y matrices is required, many relational datasets can be incorporated by defining necessary metadata as arguments to the specified transformer function.

Arbitrary estimators can also be included in Deep Mining pipelines by using the instructions for creating custom scikit-learn estimators here.

2. **Defining which hyperparameters to tune and their respective ranges**

   To specify hyperparameters and desired ranges, users construct a dictionary with hyperparameter names as keys. Values are two-element lists: the first element is a string corresponding the variable type (*cat* for categorical, *int* for integer, and *float* for continuous), and the second element is either the inclusive bounds for the hyperparameter (for integer and continuous values) or all of the values the hyperparameter can take (for categorical values).

3. **Specifying the pipeline object**

   To define the pipeline to be tuned, users define a function whose input is a hyperparameter dictionary with the following form: { *hyperparameter_name*: *hyperparameter_value*. The output of this function is a scikit-learn Pipeline object with hyperparameters set using the provided hyperparameter dictionary as input.

5.4.1 Custom Pipeline Example

In this subsection, we detail the construction of a custom pipeline for images that is made up of of Gaussian blur, Principle Component Analysis, and Random Forest Classifier steps. The first step is to define the custom transformer function implementing the Gaussian blur, using a utility from OpenCV:

```
import numpy as np
from cv2 import GaussianBlur


def gaussian_blur_fnc(X,kernel_size,stddev,matrix_type=float):
```

```
    """
    Applies Gaussian blur to the given data

    Args:
        X: data to blur
        kernel_size: Gaussian kernel size
        stddev: Gaussian kernel standard deviation
    """
    output = np.zeros(X.shape)
    X = X.astype(matrix_type)
    for i in range(X.shape[0]):
        output[i,]= np.reshape(GaussianBlur(X[i,],(kernel_size,\
        kernel_size),sigmaX=stddev,sigmaY=stddev),(X.shape[1]))
    return output
```

In the last step, we define the *set_my_pipeline* function.

1. **Specify desired model variables (sklearn Transformers or Estimators)**

   In this pipeline, we use scikit-learn's PCA and RandomForestClassifer classes for steps 2 and 3 of our pipeline. For step 1, we use our custom gaussian_blur_fnc defined above. To set a model variable using that function, we use the Function-Transformer wrapper, which simply takes in a function and returns a wrapper implementing scikit-learn's required fit and transform methods.

2. **Put the model variables in a list of (name, step variable) tuples and feed that to the Pipeline() constructor.**

3. **Set pipeline hyperparameters.**

   To do this, we simply form a list of tuples. The first item in the tuple is the name you assigned to the pipeline step, the second is the name of the parameter to be set (the name taken as an argument to your custom function or scikit-learn estimator), and the third is the name given to the parameter in the hyperparam_ranges argument specified above.

The resulting *set_my_pipeline* function:

```
from sklearn.decomposition import PCA
from sklearn.ensemble import RandomForestClassifier
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import FunctionTransformer

def set_my_pipeline(hyperparam_dict):
    """
    Constructs pipeline variables, sets params according to ↩
       parameter
    dictionary hyperparam_dict, and returns sklearn Pipeline object

    The format for steps_params is [(name of step,name of param in
    hyperparam_ranges, name of param as argument to step)]

    Args:
        hyperparam_dict: dictionary of hyperparameters to set

    Returns
        pipeline: scikit-learn Pipeline object with
                              hyperparameters set
    """
    # Step 1: Define pipeline by specifying model variables
    gaussian_blur = FunctionTransformer(func=gaussian_blur_fnc,\
                    validate=False) # custom pipeline component
    pca = PCA()
    rf = RandomForestClassifier()

    # Step 2: Construct Pipeline object
    pipeline = Pipeline([('gaussianBlur',gaussian_blur),\
                        ('PCA',pca),('RF',rf)])

    # Step 3: Set pipeline hyperparameters
    steps_params_list = [("gaussianBlur","kernel_size","kernel_size↩
       "),\
```

```
                        ("gaussianBlur","stddev","stddev"),("PCA",\
                        "n_components","pca_dim"),("RF","↩
                           n_estimators"\
                        ,"n_estimators")]
    pipeline = set_hyperparams_dm(pipeline,hyperparam_dict,\
               steps_params_list)

    return pipeline
```

5.5 Conditional Pipeline Evaluation

One potential method for achieving appreciable 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, where 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 of the pipeline. A possible system would then store the outputs of the intermediate steps for each hyperparameter set and use those precomputed outputs whenever another hyperparameter set overlaps with previously computed steps. However, the overhead for this storage process may outweigh the potential efficiency gains. This method is not currently implemented in Deep Mining but could be explored in the future. Currently, users can precompute pipeline steps by simply providing data to Deep Mining after applying transformations whose hyperparameters they do not want to tune.

5.6 Enabling Raw Data

A significant barrier to the implementation of data science pipelines is cleaning and formatting the original data. Machine learning methods typically require data in a matrix format, with X data and Y labels provided either as two matrices or in a

cross-validation split. In practice, however, data rarely conforms to this format, and preprocessing and feature engineering methods frequently take raw image, text, or database files as inputs rather than matrices. The Deep Mining system's emphasis on scalability allows for the tuning of these costly methods, and enabling realistic pipelines also requires allowing a natural data format.

The DARPA initiative has proposed a *D3M data format* for specifying datasets that contain both matrix and raw data, including various text, image, and audio formats. The Deep Mining system includes a set of utilities for parsing datasets in this format for use by pipelines in Deep Mining. With this specific structure and parsing capability, Deep Mining enables an alternative data format for users in addition to the typical matrix format.

5.7 Contribution Framework

The Deep Mining project includes elements from systems engineering, statistics, and specific data science domains. Successful future development of this system requires a structured contribution framework allowing for input from experts in each of these groups.

The most pressing group to integrate is domain experts, as their inclusion of custom feature extraction functions and pipelines for a variety of datasets can be helpful for other users. The pipeline specification framework provided by Deep Mining makes it significantly easier to share data science pipeline code, as the consistent interface allows for simple interchanging of both whole pipelines and specific steps. In order to provide new pipelines and custom functions, domain experts simply place their code in the *examples/pipelines* folder of Deep Mining, which is separated into appropriate custom function domains such as images and text to allow for an organized contributing process.

To improve on the novel copula process method of Bayesian optimization, contributors can simply modify the code provided in the *gcp* folder, which contains the entire Gaussian Copula Process model. Detailed instructions for contributors can be

found in the documentation.

By allowing experts to share constructed data science pipelines, Deep Mining also enables the comparison of these pipelines for different datasets. Future work will involve creating a structured format for saving model performances on various datasets. Future work on this contribution framework also includes the incorporation of a code testing framework, clearer separation of systems code for the implementation of distributed processing, and a structured code review process.

Chapter 6

Deep Mining: Sampling-based estimation

Real-world data science applications can involve data points numbering in the millions, introducing issues of computational tractability and memory. In the face recognition example from Chapter 3, the amount of data involved can grow quite large, as each individual data point occupies significant disk space and computation time. Because the methods we tune in our system include operations on raw image files that can occupy entire megabytes of space, the overall data science pipeline must operate on gigabytes of data and takes significant time to execute. Taking these difficulties into account, multiple questions arise:

- **How can a data scientist accelerate the evaluation of a single pipeline?**: Hyperparameter tuning involves iteratively evaluating many candidate hyperparameter sets. that span the entire pipeline, which must be trained and validated in order to produce a performance score. Because performing calculations or transformations that involve earlier stages of pipeline is computationally expensive, computing on the entire data several times during tuning is impractical even in a parallelized system. In this chapter, we investigate whether this computational time could be mitigated by executing the tuning process on a subsample. By *sampling* only part of the data, data scientists can use the same

methods in less time.

  - **How can sampling be done in an intelligent way?**: When made using only part of the data, calculations inevitably become less accurate. How can the data scientist maintain the precision of estimates?
  - **How can uncertainty be quantified?**: To be able to use these estimates in production, data scientists must first quantify their uncertainty. How can bounds be established on the estimate's variation when operating on the entire dataset?

In this chapter, we will describe approaches to answer these questions by introducing sampling and cross validation techniques, discussing the Bag of Little Bootstraps sampling method, and detailing the application of this technique to the tuning of data science pipelines.

6.1 Sampling Applications

When discussing the use of sampling in data science, two major applications arise:

- **Sampling to estimate a statistic**: This approach involves using a sample to estimate a statistic about a given dataset as precisely as possible. Examples of this process include calculating the (multidimensional) mean and variance of elements in a dataset.
- **Sampling to compare models**: When comparing two modeling techniques, models can be trained on multiple (overlapping or disjoint) subsamples to estimate and compare their accuracy. When choosing between models for a particular dataset, the goal is to pick the model with the best rank among possible models rather than calculate the value of the statistic itself (i.e., scoring or loss function).

In this thesis, we will focus on how the second approach can be applied to hyperparameter tuning.

Cross-validation is often used to choose between models and falls into the second category. k-fold cross-validation involves splitting the training set into k smaller sets by sampling without replacement and iteratively training on $k-1$ of the training data folds and validating on the remaining data fold. The resulting k scoring metrics can then be averaged to choose the best model or used to construct confidence intervals. This method serves two purposes: 1. It allows data scientists to compare models using the averaged metric, and 2. It provides an estimate of uncertainty for the model. When large amounts of data are involved, however, cross-validation is extremely expensive, requiring an approximate factor of k more computation than simply training with a single split.

Bootstrap aggregation, or bootstrapping, is a method used to improve performance estimates in machine learning algorithms [32]. In this algorithm, new training sets are generated from the original dataset (by sampling with replacement), and each of these training sets has an expected size of around 63% of the original data. After training the model using each of these training sets, the predictions on each of these sets can be combined to give more stable and accurate predictions.

In our system, we propose using *Bag of Little Bootstraps* (BLB), a subsampling method that allows for the evaluation of an arbitrary statistic of the data in superior asymptotic time with the same statistical guarantees as the bootstrap. BLB was originally developed for the first use case: sampling to estimate a statistic over data. In our case, we hope to use it to tune hyperparameters in significantly reduced time. By executing the computation in this algorithm in parallel using Apache Spark, the theoretical speed gains become even greater, significantly improving the computation time of hyperparameter optimization and making possible the automatic tuning of complex pipelines on large datasets.

6.2 Bag of Little Bootstraps

The Bag of Little Bootstraps, proposed in [24], enables the calculation of statistics on 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 applying a given estimator repeatedly 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, however, alleviates this problem by introducing an additional level of subsampling, constructing "bags" within which bootstraps are created.

More specifically, 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 multinomial sample of size n is drawn to give a set of indices. Essentially, we draw with replacement n data points from our bag of size b. Then, the estimator in question is computed on this sample of the data for each of the multinomial samples within each of the bags. For each bag, the value of the estimator is the average of the estimates on each of the bootstraps. Then, the overall estimator value given by the algorithm is the average of the estimator values for each bag.

Algorithm 2: Bag of Little Bootstraps

Input : Data: n (m-dimensional) data points $X_{1\ldots n} = \{x^1_{1\ldots m} \ldots x^n_{1\ldots m}\}$;
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` $X^j_{1\ldots b} \sim$ `random`$(X_{1\ldots n})$
3: **for** $k \to 1$ **to** r **do**
4: Sample the counts $(n_1,...,n_b) \sim$ `Multinomial`$(n, \mathbf{1}_b/b)$
Estimate y on the `bootstrap` sample using $X^j_{1\ldots b}$ and the counts as weights $\hat{y}_k = E(X^j_{1\ldots b}, n_1,...,n_b)$
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 multino-

mial sample consists of only b distinct points, meaning we are essentially operating on b data points. The computational complexity of the estimator, then, scales only with b rather than n for both time and disk space used. Because many estimators can work directly with a weighted data representation, in this case this computational benefit is easily realized, as each of the b data points has a count of occurrences such that the counts across all data points sum to n. As an example, consider a (multidimensional) *mean* as a statistic to calculate on data. Implementations of mean in Python can call on the algorithm to take a sample weight as input, allowing duplicates of data to be incorporated with asymptotic complexity that scales with the number of unique data points by simply passing an array specifying the number of times each data point appears.

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 resampled bag can be computed in parallel with no modification to the original algorithm.

6.3 Application to Tuning 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. The high-level idea is to calculate the user-defined scoring function on the multinomial samples from the BLB algorithm, then average these scores within and then across bags. In applying BLB to data science pipelines, multiple questions arise:

- **How to construct samples from raw data?**: The incorporation of raw data types, such as images and relational data, poses questions for sampling algorithms typically applied to matrices, and these processes must be adapted

for practical uses.

- **How to design a cross-validation split within BLB?**: The original BLB algorithm applies to statistics calculated on the whole dataset – how can this algorithm be adapted for machine learning algorithms that operate on training and test data?

- **How to use BLB with varied pipeline methods?**: Several methods in the initial stages of the pipeline may not have equivalent functions that can operate on weighted samples of data, and the adaption of these techniques must be considered when using BLB in data science pipelines.

In this section, we will discuss each of these questions, detailing the methods necessary to apply BLB to practical data science problems.

6.3.1 Sampling for Raw Data Types

In constructing data science pipelines, the goal is to build a supervised model, and sampling implies choosing a subset of training examples.

When organized in a *matrix format*, data is typically arranged so that rows correspond to distinct data points and columns correspond to the attributes of those data points. Taking a random sample in this framework is simple: A random vector can be generated, with each element in the range of the total number of rows (data points), and rows can be selected with or without replacement if their row index appears in the random vector.

Image data can be sampled in much the same way, as the color or greyscale pixel values for each image can simply be flattened so that each row again corresponds to a data point. To enable feature transformation methods that use the rectangular structure of images, three-dimensional arrays can be used so that each "row" now corresponds to a two-dimensional vector. Again, sampling is simple, as rows can be selected by generating indices at random.

Text data can be similarly sampled, as the text(s) in question for each data point

can simply be placed as elements of each row. Sampling can then be performed in the same way as before.

Relational datasets introduce complications in sampling, as multiple tables (matrices) can exist for a single dataset. In this case, sampling can be done in the *labels* table, which has the list of entity-instances and the associated table. While we load and maintain the entire data in memory in this method, computations are done using only a subset of the entity-instances.

Other considerations: Sampling can either be implemented at the file level or after data has already been imported into Python objects. Both approaches have their merits, as sampling at the file level (e.g., by selecting only specific file names from a list of all files in the directory) can reduce the computational burden by loading only raw files used. However, this approach requires a different sampler for different data formats. For example, text datasets can be stored as individual files for each data point, as groups of data points, or in a single file for all data points, and each of these storage options would require a different sampler. For this reason, Deep Mining currently uses a single sampler at the matrix level, with custom samplers implemented for relational datasets.

6.3.2 Cross-validation in BLB

Armed with a sampling method that is valid across datasets, we incorporate cross-validation in BLB using a train-validation split approach based on that proposed in [34]. 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, multinomial samples 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 bags are averaged to compute a final estimate. The detailed process can be found in Algorithm 3.

Algorithm 3: Bag of Little Bootstraps for Data Science Pipelines

Input : Data: n (m-dimensional) data points $X_{1...n} = \{x^1_{1...m} \dots x^n_{1...m}\}$;
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** $j \to 1$ **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** $k \to 1$ **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 $\hat{y} = s^{-1} \sum_{j=1}^{s} \hat{y}_j$

6.3.3 Application of BLB to Varied Pipeline Methods

The asymptotic benefits of BLB, however, can only be realized using pipeline methods that can operate on the weighted forms of data. As discussed earlier, data science pipelines use two major types of pipeline steps: 1) *transformer* methods, examples of which include preprocessing and feature extraction methods, and 2) *estimator* methods, such as classifiers and regressors. Pipelines typically consist of a series of transformer methods followed by a single Estimator method that produces a score or predictions. Transformer methods simply return an altered version of the original X data and Y labels.

When discussing the application of BLB to data science pipelines, two types of transformers emerge:

- **Independent Transformers**

  These methods operate on each data point independently and do not require the whole dataset in order to function. An example of this method is the histogram of oriented gradients (HOG), which operates on each image in isolation.

- **Joint Transformers**

  These methods require the whole dataset to function, as their output depends on input from the whole dataset. Examples include Bag of Words, which if outputting a vector of length number of words in the vocabulary needs knowledge of the vocabulary used by the entire dataset. Another example is Principle Component Analysis (PCA), whose learned outputs vary with the number of occurrences of specific data points.

Consider a pipeline that uses both joint and independent transformer methods, with an eventual estimator capable of using weights from the multinomial sample. In contrast to the previously discussed mean function, multiple methods are involved, and not all of the methods use the sample weights. Independent transformers can be immediately applied to data bags from BLB: Only b unique data points exist, and the weights drawn from the multinomial sample can simply be passed along to subsequent Transformers or Estimators who use them. Joint Transformers sometimes can work with unweighted bag data – in Bag of Words, the algorithm needs only to know the scope of the vocabulary, and repeating words can not affect the output, depending on the output format. However, some joint transformers such as PCA require weighted versions of the data. To accommodate these transformer methods, users must either implement weighted versions of their particular algorithm, or use the wrapper provided by Deep Mining, which gives duplicated data to the transformer method as input. Implementing weighted versions of the algorithm in question is much faster in practice, and examples can be found in packages such as scikit-learn.

6.4 Implementation

The BLB algorithm for machine learning is quite parallelizable, as bags can be computed on in parallel. Deep Mining uses Apache Spark on EC2 clusters as well as the pathos multiprocessing method to parallelize this computation. As explained in detail in chapter 7, Deep Mining uses a map-reduce structure in which it computes

estimator scores for a given multinomial sample and bag, then aggregates these scores by averaging across all multinomial samples and bags.

To run the BLB algorithm in Deep Mining, the user only needs to specify the value of the BLB hyperparameters: the size of the bags b, the number of sampled bags s, and the number of bootstraps r. As discussed in chapter 9, we have found $b = n^{0.6}$, $s = 8$, and $r = 20$ to be sufficient for many data science pipelines.

6.5 Extensions

The map-reduce implementation of the BLB algorithm in our system also allows for different reduction functions, giving the flexibility to calculate other statistical quantities about the estimator scores in question. For example, confidence intervals and estimator variances can be constructed by using the empirical multinomial sample scores [18].

6.5.1 Reducing Complexity of Data Science pipelines

The asymptotic computational benefits that come from using this BLB train-validation split are considerable, 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 through machine learning algorithms that can use data given as weighted samples. Implementations with this capability can be easily found in open-source libraries like scikit-learn.

Setting the BLB hyperparameters is a major consideration in this algorithm, as changing the number of bags s and the number of multinomial samples in each bag r can dramatically change the running time. In Algorithm 3, we train the estimator $s \cdot r$ times and test the estimator s 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. Because of this dependence, setting the BLB hyperparameters can significantly change the runtime of this algorithm. We make these hyperparameter

considerations empirically, as shown in Chapter 9. The theoretical guarantees that come with this method roughly follow from those of traditional BLB, and we show the effectiveness of this algorithm through an empirical analysis.

Chapter 7

Deep Mining: Parallel computation

A major shortcoming of existing hyperparameter tuning systems is their inability to scale to work with large datasets and full pipelines that incorporate costly feature extraction methods. By failing to provide users with software infrastructure for specifying and efficiently evaluating pipelines, these tools require data scientists to fill in the gaps and create the infrastructure necessary to tune real-world pipelines and datasets. Sampling-based methods like BLB, which must be implemented in a distributed system, theoretically improve evaluation time. In this section, we will make a distinction between *parallelization methods*, which are ways to conceptually organize the parallel computation, and *parallelization frameworks*, which are specific software tools used to compute in parallel (e.g., Apache Spark).

7.1 Parallelization Methods

Deep Mining implements two methods of parallelization:

- **Bag of Little Bootstraps algorithm for machine learning**: The variation of the Bag of Little Bootstraps algorithm used in Deep Mining can be found in the Chapter 6. The BLB algorithm provides two possible levels of parallelization.

  The training and validation within each pair of bags can be done completely in

parallel. The first level of parallelization executes computation on each bag on a separate node. Within each training bag, the computation on the bootstraps samples is also independent, so training using these samples can also be done entirely in parallel.

Within Deep Mining, parallelization across each bag pair is currently implemented in the frameworks used, and future work may include parallelization within the training bags to further speed the process. For both levels of parallelization, the algorithm cannot continue until all jobs are completed. The parallelization framework, then, must implement a synchronous process.

- **Hyperparameter sets in parallel**: In addition to the Bag of Little Bootstraps sampling method, Deep Mining provides an alternative method of parallelization in which different hyperparameter sets are evaluated in parallel. In this process, the acquisition function is evaluated for a grid of candidates as before, and, instead of choosing only the single best candidate, the best k candidates are evaluated on the full pipeline.

  When evaluating hyperparameter sets in parallel, the method of parallelization is fairly clear: each set of hyperparameters is evaluated by a distinct worker process. This method is currently implemented in Deep Mining. However, the hyperparameter optimization algorithm is sequential, so the computations could also be performed asynchronously, with worker processes evaluating new hyperparameter sets in immediate succession.

7.2 Parallelization Frameworks

When comparing distributed processing engines, Hadoop MapReduce and Apache Spark [47] commonly emerge as contenders. Hadoop MapReduce, the historical standard, integrates well with the remainder of the Hadoop ecosystem, including Hadoop YARN, HDFS, and Hadoop Common. Apache Spark provides a faster, more general engine for data processing, utilizing in-memory processing that makes it signifi-

cantly more efficient than Hadoop MapReduce. Spark can also run up to ten times faster than MapReduce on disk and incorporates well with existing Hadoop tools like HDFS and Yarn. Users frequently choose Spark over Hadoop MapReduce for its more friendly API (which includes interfaces to Java and Python), general processing framework, and improved speed.

Dask [10] provides an alternative to Apache Spark that encompasses a subset of Spark's more broadly inclusive framework. It is a lightweight alternative to Spark, with fewer features, but includes a simpler API and easier integration with Python libraries such as Numpy and Pandas. Python also provides a multiprocessing module that allows for parallel threads and processes to be run on a single machine and provides a lighter alternative to these more general systems.

7.3 Current Implementation

We began our implementation of parallel processing in Deep Mining by using Apache Spark both locally and on EC2 clusters. Deep Mining now runs on Spark clusters with multiple parallelization methods, provides instructions to set up Apache Spark clusters on EC2 machines, and runs without additional setup on one machine. This implementation came with difficulties, which are outlined in the attached Appendix A.

We explored the Python multiprocessing module for use in parallel computation. We used the pathos package [30] for its support of parallelizing more complicated functions by using *dill*, an extension of Python's *pickle* module for serializing Python objects. Because it has a smaller computational overhead than Spark, pathos enabled faster parallelization on local machines for the pipelines in my experiments.

When the pathos multiprocessing module is used, a new Python process is created for each hyperparameter set. With Apache Spark, new tasks in the given Spark job are created and sent to a Spark executor. With BLB, a user-specified number of tasks are specified: When using the pathos multiprocessing module, this number specifies the number of Python processes spawned. In the Apache Spark framework, an array

corresponding to the indices in the original data for each bag is split into as many partitions as the user specifies.

7.4 Future Work

Because of its broader support for Python, lighter-weight architecture, and tighter integration with numeric libraries, Dask could be an effective additional parallelization framework for Deep Mining. We plan to experiment with its integration in future work.

In addition to providing the necessary software infrastructure, an additional component of Deep Mining could be a hardware system on which users can run their pipelines. This hardware infrastructure could include specialized EC2 machines and clusters pre-configured for use by data scientists.

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 [7, 22] 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.

Chapter 8

Interacting with Deep Mining

An important goal of the Deep Mining system is to provide a simple, intuitive API design that allows for effective hyperparameter tuning. In addition to demonstrating the efficacy of theoretical improvements, we aim to provide a usable system that addresses the shortcomings of existing hyperparameter tuners. In this section, we describe the end-to-end data science pipeline tuning framework of Deep Mining through concrete examples, along with the contribution structure available for a variety of domain experts.

- The source code for Deep Mining can be found here: `https://github.com/HDI-Project/DeepMining`.

- The documentation for the software could be found here: `http://hdi-project.github.io/DeepMining/`

8.1 End-to-end System

Using the Deep Mining system consists of three steps: 1) Load the desired data, 2) Define a pipeline, either by choosing from the existing library or defining custom functions and hyperparameter ranges, and 3) Run the hyperparameter optimization.

8.1.1 Data Loading

A major shortcoming of existing tuning systems is their inflexibility in accepting different data types. Typically, these tools require data in a matrix format, either as a nested Python list or a NumPy array or matrix. This constraint often requires the user to convert their existing data, an error-prone and often lengthy process. To overcome this problem, Deep Mining provides a parser for data in raw file formats such as JPEG or text using the D3M structure [1], enabling users to tune hyperparameters simply by defining auxiliary JSON files describing their data and leaving their data in a native file format. In many cases, this structure reduces user effort and frustration by providing a faster and more reliable method of data importing. The matrix format of data, however, may be preferable in a variety of applications, so Deep Mining also allows for this familiar data input style.

Users load data with the `d3m_load_data` function using arguments as necessary. Full documentation exists for this and other functions, but notable arguments include `data_directory`, a string denoting the location of the data files, X and Y for data in matrix format, and a percentage denoting the (random) percentage of the total data to use as a sample. Table 8.1 lists the arguments for this function.

[1] D3M stands for Data Driven Discovery of Models. Developed by MIT Lincoln Labs, this format is a standardized data structure to represent complex datasets.

Table 8.1: Arguments for `d3m_load_data` function.

Argument	Data type	Conditional on?	Required?	Purpose
`data_directory`	String	Must be specified if X or Y is None	No	Location of D3M-formatted data
`X`	Numpy array	Must be specified if data_directory is None	No	Data points for train-validation split
`Y`	Numpy array	Must be specified if data_directory is None	No	Data labels for train-validation split
`sample_size_pct`	float	None	No	Percentage of data to use in train and validation sets combined. In range (0.0,1.0]. Lower values can be used either for testing or for running large datasets quickly.
`validation_size_pct`	float	None	No	Percentage of sampled data to use in validation set. In range (0.0,1.0). For example, if sample_size_pct = 0.5 and validation_size_pct = 0.5, then 25% of the data will be sampled at random to be used in the validation set (and 25% for the train set)

8.1.2 Defining A Pipeline

Deep Mining provides pipeline implementations with defined hyperparameter ranges for a variety of datasets. Current pipelines include a Convolutional Neural Network image pipeline, a traditional image pipeline using HOG features, a text pipeline using Bag of Words and term frequency-inverse document frequency features, and a random forest classifier pipeline for general use. Using these pipelines is simple, requiring the user to specify a single argument, `pipeline_filepath`, in the `DeepMine` function. Instructions for defining a pipeline in Deep Mining can be found in Section 5.4.

8.1.3 Run Hyperparameter Optimization

Once a pipeline has been defined, users execute the hyperparameter optimization process by calling the `DeepMine` function with appropriate arguments. Arguments of `DeepMine` include the parallelization system and method, BLB hyperparameters, and total number of iterations. `DeepMine` either returns a pipeline with the highest scoring hyperparameters, or a tuple of best pipeline object and a Pandas DataFrame of hyperparameters and their associated performances, depending on the value of the boolean argument `store_performances`.

After setting the hyperparameters using the parameter dictionary given by a Bayesian hyperparameter optimization process, the pipeline is then fit and validated using the provided cross-validation split, outputting the score that will be used in the optimization. The pipeline executor chooses between parallel and non-parallel execution methods to provide scores to the Bayesian optimizer, which provides new hyperparameter sets to test in a sequential process until the final pipeline is returned to the user.

Table 8.2: Arguments for `DeepMine` function. R= Required?, D = Default

Argument	Data type	Conditional on?	R?	D	Purpose
`data`	dataObject	None	Yes	-	dataObject instance loaded from d3m_load_data. Must contain X_train, y_train, X_val, and y_val attributes.
`hyperparam_ranges`	dict	Must be specified if pipeline_filepath is not	No	None	Dictionary specifying hyperparameter names, ranges, and types.
`set_pipeline`	function	Must be specified if pipeline_filepath is not	No	None	set_pipeline function that returns a scikit-learn Pipeline object set using a hyperparam_dict argument.
`pipeline_filepath`	string	Must be specified if hyperparam_ranges and set_pipeline are not both specified	No	None	Filepath of Python file containing pipeline functions. Filepath starts from the DeepMining directory (e.g., examples.pipelines.traditional_image). Must be separated by “.”, not “‘ or “/”.
`parallelized`	int	None	No	0	0: no parallelization; 1: parallelization using pathos multiprocessing module; 2: parallelization using Apache Spark
`use_BLB`	boolean	If parallelized != 0, then this is active	No	False	If True, use Bag of Little Bootstraps subsampling method for training and testing.
`num_total_iter`	int	None	No	4	Number of SmartSearch iterations to run
`num_parallel_sets`	int	Active only if parallelized != 0 and use_BLB == False	No	3	Number of hyperparameter sets to evaluate in parallel in the non-BLB parallelization method
`SmartSearch _param_dict`	dict	None	No	None	Dictionary of parameters to pass to SmartSearch initializer. Keys are argument names, and values are argument values.
`blb_param_dict`	dict	Active only if parallelized != 0 and use_BLB == True	No	None	Dictionary with keys corresponding to BLB hyperparameters: *gamma* (exponent in size of bag), *s* (number of bags), *r* (number of Monte Carlo iterations), and *partitions* (number of partitions for RDD in Apache Spark). Values in the dictionary correspond to the values of these hyperparameters and are read in blb_utils.py

Table 8.2 continued.

Argument	Data type	Conditional on?	R?	D	Purpose
`my_spark_directory`	string	Active only if parallelized == 2 (and then required)	No	None	Local location of spark installation directory. Required to run Apache Spark method of parallelization.
`spark_cluster_location`	int or string	Active only if parallelized == 2	No	None	Location of spark cluster: either 0 (local mode), 1 (EC2 cluster), or 2 (custom Spark URL as a string). In EC2 mode, the URL is found automatically in the deep_mine function.
`store_performances`	boolean	None	No	False	If True, return Pandas dataframe with all hyperparameter sets and their associated scores.
`score_fnc`	string or sklearn scoring object	None	No	None	String or scikit-learn scoring object denoting the method of scoring used. Examples can be found at http://scikit-learn.org/stable/modules/model_evaluation.html

8.1.4 Built-in Pipeline Example

Running existing pipelines in Deep Mining is simple. First, we load the data, setting the optional `sample_size_pct` so that we load a random 10% portion of the data:

```
1 image_data_directory = 'path/to/data'
2 image_data = d3m_load_data(data_directory = image_data_directory,
3                            sample_size_pct=0.1 # Percentage of data↩
                                 uses as subsample
4                           )
```

Then, we run `DeepMine`, in this case using the traditional image pipeline, located in *examples/pipelines/traditional_image.py*:

```
1 image_pipeline = DeepMine(
2         data = image_data,
3         pipeline_filepath = "examples.pipelines.traditional_image",
4         )
```

When our hyperparameter optimization has finished, we have a scikit-learn Pipeline object in the `image_pipeline` variable, on which we can call `predict` or other methods as desired.

8.1.5 Custom Pipeline Example

Running the custom pipeline from section 5.4.1 requires two steps. First, define the desired hyperparameters and ranges:

```
1 hyperparam_ranges = {
2                 'kernel_size' : ['cat',[3,5]],
3                 'stddev' :      ['int',[0,4]],
4                 'pca_dim' :     ['int',[50,300]],
5                 'n_estimators': ['int',[2,10]]}
```

and the *set_my_pipeline* function from section 5.4.1:

```
from sklearn.decomposition import PCA
from sklearn.ensemble import RandomForestClassifier
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import FunctionTransformer

def set_my_pipeline(hyperparam_dict):
    """
    Constructs pipeline variables, sets params according to ↩
       parameter
    dictionary hyperparam_dict, and returns sklearn Pipeline object

    The format for steps_params is [(name of step,name of param in
    hyperparam_ranges, name of param as argument to step)]

    Args:
        hyperparam_dict: dictionary of hyperparameters to set

    Returns
        pipeline: scikit-learn Pipeline object with
                              hyperparameters set
    """
    # Step 1: Define pipeline by specifying model variables
    gaussian_blur = FunctionTransformer(func=gaussian_blur_fnc,\
                    validate=False) # custom pipeline component
    pca = PCA()
    rf = RandomForestClassifier()

    # Step 2: Construct Pipeline object
    pipeline = Pipeline([('gaussianBlur',gaussian_blur),\
                         ('PCA',pca),('RF',rf)])

    # Step 3: Set pipeline hyperparameters
    steps_params_list = [("gaussianBlur","kernel_size","kernel_size↩
       "),\
```

```
                        ("gaussianBlur","stddev","stddev"),("PCA",\
                        "n_components","pca_dim"),("RF","↩
                           n_estimators"\
                        ,"n_estimators")]
    pipeline = set_hyperparams_dm(pipeline,hyperparam_dict,\
                steps_params_list)

    return pipeline
```

Then, call the `DeepMine` function using the *image_ data* object from before:

```
1 custom_pipeline = DeepMine(
2         data = image_data,
3         set_pipeline = set_my_pipeline,
4         hyperparam_ranges = hyperparam_ranges
5         )
```

Chapter 9

Experimental Results

In our experimentation with Deep Mining, we aimed to answer the question: Is Bayesian hyperparameter optimization using BLB an effective way to 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 the estimator scores that result from using BLB on the pipeline is effective, then we will see improvement in the pipeline performance that is roughly comparable to the non-BLB approach, where we calculate the pipeline scores using all of the data.

9.1 Datasets

For this question, we experimented with image and text pipelines.

9.1.1 Handwritten Digits

We first considered the famous "handwritten digits recognition problem," using the MNIST dataset. In this dataset, training data consists of images depicting handwritten digits from 0 to 9. The task of the classifier is to predict the digit label from each greyscale image.

9.1.2 Sentiment Analysis

The second problem we considered involves using the text of movie reviews to predict whether a particular review is "positive" or "negative". The dataset used is from Kaggle's Sentiment Analysis competition Bag of Words Meets Bags of Popcorn. The reviews are from IMDB and the labels were attributed based on the rating accompanying the IMDB review: "negative" for a rating lower than 5, and "positive" for a rating greater than or equal to 5 (on a one-to-ten scale).

9.2 Pipelines

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

Traditional image pipeline

For images, we used a pipeline consisting of the scikit-image [43] 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, term frequency-inverse document frequency features, and a Naive Bayes classifier.

Table 9.1: Types of pipelines.

Pipeline type	Description	Hyperparameters
Traditional Image	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]
CNN-Image	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	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 $\alpha \in [0.01; 1]$

9.3 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 ran one experiment using BLB and another in which we operated on the whole dataset (non-BLB). Each experiment consisted of ten trials of SmartSearch with 30 iterations in each trial. Each SmartSearch trial then consisted 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 using the highest predicted score. We ran three experiments in this case: the HOG image pipeline, the CNN image pipeline, and the traditional text pipeline. The BLB hyperparameters used for 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.

9.4 Evaluation

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

9.4.1 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, and then outputting the score on the 6,000 data points from the validation set to SmartSearch.

In the BLB process, we use the 36,000 data points for training and validation and, for final comparison in the below analysis, 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. This split was chosen to make a more appropriate comparison between the performances of the BLB and non-BLB hyperparameter sets chosen, as these sets are then trained and validated with the same data.

9.4.2 Text Dataset

In the dataset from Kaggle's competition, we have 25,000 text reviews and their associated labels, and we choose a train-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. With both of these methodologies, BLB has a disadvantage, as fewer data points are available to the training and validation process than in non-BLB evaluation (6,000 more images, and 5,000 more text reviews).

9.5 Results

In our experimental results, we found that *sampling-based hyperparameter optimization using BLB leads to performance improvements comparable with those obtained*

by evaluating the entire pipeline.

To see this result, we can first plot the improvement over the course of the Smart-Search process for each pipeline, averaged over the ten experiments. In Figures 9-1, 9-2, and 9-3, the horizontal axis corresponds to the iteration number and the vertical axis corresponds to the best estimator performance before and including a given iteration.

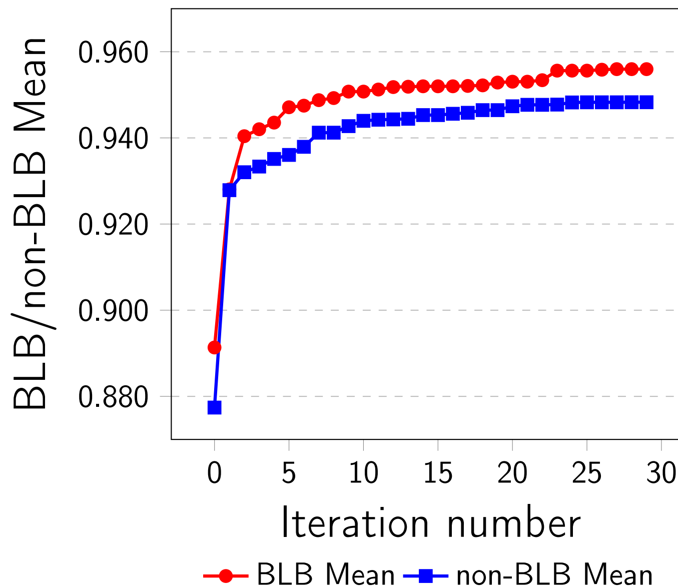

Figure 9-1: Experimental averages for the HOG image pipeline.

The performances from BLB and non-BLB estimator evaluation for all of these pipelines are comparable, 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 $\lceil 36000^{0.6} \rceil = 542$ 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 experiment data.

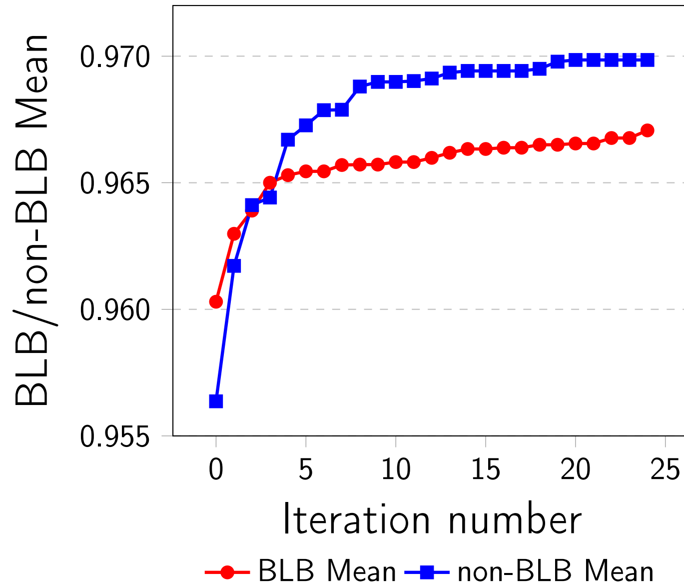

Figure 9-2: Experimental averages for the CNN Image pipeline.

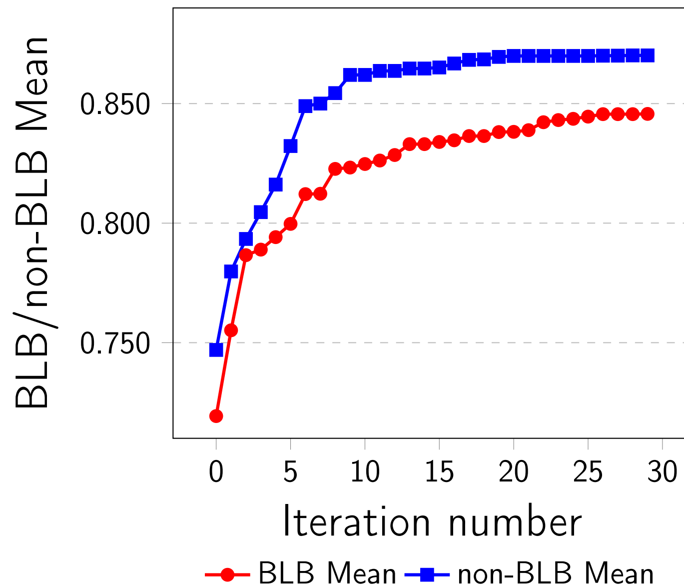

Figure 9-3: Experimental averages for the traditional text pipeline.

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 Table 9.2. As we can see, the mean performance improvements are comparable for the pipelines, with the CNN image pipeline leading to lower im-

Table 9.2: BLB and non-BLB processes statistic.

	BLB		Non-BLB	
	Mean	Std. Deviation	Mean	Std. Deviation
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

provements for the reasons detailed below.

9.6 Discussion

When evaluating the hyperparameter sets chosen by the BLB and non-BLB SmartSearch experiments, we can see that their results follow intuition by manual inspection. For example, in the HOG pipeline we use a random forest estimator with the number of trees as a hyperparameter, and the hyperparameter sets chosen by the SmartSearch consistently choose the highest number of trees (ten). This finding 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.

The focus in these initial experiments was obtaining accurate estimates 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 or a language with more native integration with Spark. When increasing the computational power of the cluster machines, we saw that the speed improvements were small, which we attribute to the small dataset size leading to the Apache Spark overhead dominating the computation time.

The performance improvements for the CNN image pipeline are smaller than for the more traditional pipelines, which aligns with the conventional intuition that neural networks need large amounts of data to achieve high accuracies. The amount of data given in each bag is $\lceil 36000^{0.6} \rceil = 542$, so the reduced improvement in the hyperparameter optimization is reasonable given this extremely small data size.

The performance improvements we have 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. Future work can optimize the existing Deep Mining system to reduce computational overhead to take full advantage of this asymptotic improvement.

Chapter 10

Conclusion

In this work, we have established the validity of using Bag of Little Bootstraps for sampling-based evaluation of data science pipelines. We have presented the Deep Mining system, providing an alternative to the black-box approaches of existing hyperparameter optimization systems. By using a structured approach to pipeline construction, we enable parallelization of pipeline evaluation and provide a distributed system capable of multiple parallelization techniques. By establishing clear abstractions for statisticians, domain experiments, and systems programmers, we enable future contribution across application areas.

10.1 Future Work

This project has significant potential for future development. While various possibilities have been previously discussed in this thesis, various other future directions exist.

Rank-based search. When using the Bag of Little Bootstraps algorithm adapted for machine learning techniques, the scale of pipeline performances changes significantly. However, while we expect that the magnitude of difference between pipeline scores will change, the rank has the possibility to be more consistent. To aid in the Bayesian optimization, a rank-based hyperparameter search could be implemented, using the ranks in the GP or GCP modeling rather than the raw scores themselves.

An adaptive sampling approach: As a single hyperparameter optimization in Deep Mining progresses, the Bayesian model of the hyperparameter space continues to improve. In making the trade-off between exploration and exploitation, *the cost allocated to each pipeline evaluation could increase with more iterations*. In the first hyperparameter sets tested, smaller sample sizes could be used, with more data used for training and validation as the model of the hyperparameter space increases its precision. This strategy could be adopted for BLB and non-BLB hyperparameter evaluations. When using BLB, the BLB hyperparameters could be altered as the optimization progresses: the γ exponent specifying the bag size could gradually be increased, or the number of bags used or Monte Carlo iterations could be set to higher values. In non-BLB pipeline evaluation, the size of the subsample could be gradually increased with higher iterations.

BLB improvements: Bag of Little Bootstraps can be used to produce confidence intervals [18], and this capability could be incorporated into Deep Mining. This addition would give data scientists projections on how the sample-evaluated pipeline would fare on the entire dataset. Also, the bag and Monte Carlo estimates from the BLB sampling hierarchy could all be used as inputs to the Bayesian modeling process, which could improve accuracy by giving more information than a simple average. The hyperparameters for BLB can also be autotuned as described in [24], eliminating the additional hyperparameter specifications required for a user who wants to alter the default hyperparameters in a more informed way.

Improved Bayesian modeling: With increased use of the Deep Mining system, additional pipeline hyperparameter-score datasets will be available. The Bayesian models used can benefit from this information by incorporating information from these previous, related pipeline evaluations into new pipelines in similar domains, as in multi-task Bayesian optimization. A sensitivity analysis in the hyperparameter space could also be outputted by the DeepMine function.

Appendix A

Comments on Apache Spark

While it provides significant speed improvements and enables an extremely general set of workflows, Apache Spark does come with its difficulties. A frequent complaint made about Apache Spark is the use of specialized terminology in its documentation and steep learning curve for users in developing familiarity with the system, often relegating the use of Spark to systems experts. With little prior programming experience with distributed systems, I found these and other difficulties when incorporating Spark into Deep Mining:

- **Tools that are not maintained**

  Spark provides scripts to setup a cluster on EC2 machines, but they are outdated: They require multiple additional steps (e.g., re-installing Spark on the resulting instances) and leave out useful functionality, spawning an array of error messages upon startup. While the principle functionality is accessible after few additional steps, this tool could certainly be improved.

- **PySpark Debugging Difficulties**

  The largest downside I found while using Spark was that debugging code that uses PySpark is quite convoluted: Tracebacks are duplicated and pass through an array of internal Spark files; error messages are cryptic, citing errors in the JVM and giving uninformative messages for common problems; and few suggestions are given to the programmer. The disconnect resulting from processes

running natively on the JVM makes Python programming both less efficient and usable.

- **Inconsistent API between local and cluster machines**

  For example, the addFile() method for SparkContexts supports recursive mode only with local clusters or Hadoop-supported file systems. Requiring changes in user code between local and cluster complicates the implementation process, and the addition of support for features in both local and cluster would unify the API.

- **Lagging Python support**

  Spark development typically focuses on first implementing new features in Scala or Java then updating Python APIs, resulting in a lack of features available in Python and a slower implementation overall.

- **Lack of code examples in documentation**

  Examples for common Spark uses could be expanded, especially for running in cluster mode from within Python files. Tutorials for launching and running code on clusters would significantly speed the learning process for new users.

Appendix B

Functions and abstractions in *DeepMine*

The *DeepMine* function and the data loading utility *d3m_load_data* take as input a variety of arguments to enable their functionality, and the tables in this appendix describe those arguments in detail.

Table B.1: Deep Mining Functions

Function	Description	Is called by	Calls
d3m_load_data	Loads data in either matrix or raw format and splits the data into training and validation sets. Data in matrix format is handled within the function, and dataObject methods are used for D3M-formatted data.	User	dataObject methods
DeepMine	Runs the hyperparameter optimization given a dataObject instance, parsing user arguments and setting up pipeline evaluation framework. All other abstractions are encapsulated within this function except for d3m_load_data.	User	BLB get_pipeline_score, SmartSearch methods (including parallel)
set_hyperparams_dm	Wrapper that sets hyperparameters for sklearn pipelines	User	None
BLB get_pipeline _score	Wrapper for the run_blb function that creates a pipeline given a dictionary of hyperparameters and executes run_blb	DeepMine	run_blb
run_blb	Executes pipeline scoring using BLB given a pipeline object, data, and other parameters.	BLB get_ pipeline_score	score_bag (which executes multinomial sampling within each bag)
dataObject	Python class providing utilities to parse data in D3M format	d3m_load_data	Own methods
(folder) examples/pipelines	Folder containing all pre-made pipelines as well as custom functions for image, text, and other datasets	User	Own methods
(folder) GCP	Implements Gaussian Copula Process model.	SmartSearch	Own methods
(folder) smart_search	Implements the SmartSearch algorithm for hyperparameter tuning using GP or GCP. Also implements the hyperparameter sets in parallel method	GCP	Own methods

Table B.2: Classification of Pipeline Steps

Type	Subtype	Description
Transformer		
	Custom	Transformers implemented using the FunctionTransformer object and custom user functions
	scikit-learn	Transformers already implemented by scikit-learn (e.g., PCA)
	Independent	Transformers that operate on each data point independently (e.g., HOG)
	Joint	Transformers that must operate on the entire dataset (e.g., Bag of Words)
Estimator		
	Custom	Estimators implemented using BaseEstimator, with fit and score methods implemented by user
	scikit-learn	Estimators already implemented by scikit-learn

Bibliography

[1] Alec Anderson, Sebastien Dubois, Alfredo Cuesta-Infante, and Kalyan Veeramachaneni. Sample, estimate, tune: Scaling bayesian auto-tuning of data science pipelines. In *Data Science and Advanced Analytics (DSAA), 2017. IEEE International Conference on*, pages 1–10. IEEE, 2017.

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

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

[4] James Bergstra, Dan Yamins, and David D Cox. Hyperopt: A python library for optimizing the hyperparameters of machine learning algorithms. In *Proceedings of the 12th Python in Science Conference*, pages 13–20, 2013.

[5] James S. Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 24*, pages 2546–2554. Curran Associates, Inc., 2011.

[6] Bernd Bischl, Michel Lang, Lars Kotthoff, Julia Schiffner, Jakob Richter, Erich Studerus, Giuseppe Casalicchio, and Zachary M. Jones. mlr: Machine learning in r. *Journal of Machine Learning Research*, 17(170):1–5, 2016.

[7] Bryan Catanzaro, Shoaib Kamil, Yunsup Lee, Krste Asanovic, James Demmel, Kurt Keutzer, John Shalf, Kathy Yelick, and Armando Fox. SEJITS: Getting productivity and performance with selective embedded JIT specialization. In *Workshop on Programmable Models for Emerging Architecture (PMEA)*, 2009.

[8] François Chollet et al. Keras. `https://github.com/fchollet/keras`, 2015.

[9] Marc Claesen, Jaak Simm, Dusan Popovic, and BD Moor. Hyperparameter tuning in python using optunity. In *Proceedings of the International Workshop on Technical Computing for Machine Learning and Mathematical Engineering*, pages 6–7, 2014.

[10] Dask Development Team. *Dask: Library for dynamic task scheduling*, 2016.

[11] 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.

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

[13] Tobias Domhan, Jost Tobias Springenberg, and Frank Hutter. Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves. In *IJCAI*, pages 3460–3468, 2015.

[14] Sébastien 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.

[15] Katharina Eggensperger, Matthias Feurer, Frank Hutter, James Bergstra, Jasper Snoek, Holger Hoos, and Kevin Leyton-Brown. Towards an empirical foundation for assessing bayesian optimization of hyperparameters. In *NIPS workshop on Bayesian Optimization in Theory and Practice*, volume 10, 2013.

[16] Manoj Kumar et. al. Scikit-optimize. `https://github.com/scikit-optimize/scikit-optimize`, 2017.

[17] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. Efficient and robust automated machine learning. In *Advances in Neural Information Processing Systems*, pages 2962–2970, 2015.

[18] Christopher Garnatz. *Trusting the Black Box: Confidence with Bag of Little Bootstraps*. PhD thesis, Pomona College, 2015.

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

[20] Tomáš Horváth, Rafael G Mantovani, and André CPLF de Carvalho. Effects of random sampling on svm hyper-parameter tuning. In *International Conference on Intelligent Systems Design and Applications*, pages 268–278. Springer, 2016.

[21] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. *LION*, 5:507–523, 2011.

[22] Shoaib Ashraf Kamil. *Productive High Performance Parallel Programming with Auto-tuned Domain-Specific Embedded Languages*. PhD thesis, EECS Department, University of California, Berkeley, Jan 2013.

[23] James Max Kanter and Kalyan Veeramachaneni. Deep feature synthesis: Towards automating data science endeavors. In *Data Science and Advanced Analytics (DSAA), 2015. 36678 2015. IEEE International Conference on*, pages 1–10. IEEE, 2015.

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

[25] Brent Komer, James Bergstra, and Chris Eliasmith. Hyperopt-sklearn: automatic hyperparameter configuration for scikit-learn. In *ICML workshop on AutoML*, 2014.

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

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

[28] Ruben Martinez-Cantin. Bayesopt: A bayesian optimization library for nonlinear optimization, experimental design and bandits. *The Journal of Machine Learning Research*, 15(1):3735–3739, 2014.

[29] Robert T. McGibbon, Carlos X. Hernández, Matthew P. Harrigan, Steven Kearnes, Mohammad M. Sultan, Stanislaw Jastrzebski, Brooke E. Husic, and Vijay S. Pande. Osprey: Hyperparameter optimization for machine learning. *The Journal of Open Source Software*, 1(5), sep 2016.

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

[31] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. Mllib: Machine learning in apache spark. *J. Mach. Learn. Res.*, 17(1):1235–1241, January 2016.

[32] Christopher Z Mooney and Robert D Duval. *Bootstrapping: A nonparametric approach to statistical inference.* Number 94-95. Sage, 1993.

[33] Randal S. Olson, Nathan Bartley, Ryan J. Urbanowicz, and Jason H. Moore. Evaluation of a tree-based pipeline optimization tool for automating data science. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, GECCO '16, pages 485–492, New York, NY, USA, 2016. ACM.

[34] Luca Oneto, Bernardo Pilarz, Alessandro Ghio, and Davide Anguita. Model selection for big data: Algorithmic stability and bag of little bootstraps on gpus. In *Proceedings*, page 261. Presses universitaires de Louvain, 2015.

[35] 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*, 12:2825–2830, 2011.

[36] C.E. Rasmussen and C.K.I. Williams. *Gaussian Processes for Machine Learning*. Adaptive Computation And Machine Learning. MIT Press, 2005.

[37] Bernard W Silverman. *Density estimation for statistics and data analysis*, volume 26. CRC press, 1986.

[38] Edward Snelson, Carl Edward Rasmussen, and Zoubin Ghahramani. Warped gaussian processes. *Advances in neural information processing systems*, 16:337–344, 2004.

[39] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pages 2951–2959, 2012.

[40] Jasper Snoek, Kevin Swersky, Rich Zemel, and Ryan Adams. Input warping for bayesian optimization of non-stationary functions. In *International Conference on Machine Learning*, pages 1674–1682, 2014.

[41] Kevin Swersky, Jasper Snoek, and Ryan P Adams. Multi-task bayesian optimization. In *Advances in neural information processing systems*, pages 2004–2012, 2013.

[42] Chris Thornton, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Auto-weka: Automated selection and hyper-parameter optimization of classification algorithms. *CoRR, abs/1208.3719*, 2012.

[43] Stéfan van der Walt, Johannes L. Schönberger, Juan Nunez-Iglesias, François Boulogne, Joshua D. Warner, Neil Yager, Emmanuelle Gouillart, Tony Yu, and the scikit-image contributors. scikit-image: image processing in Python. *PeerJ*, 2:e453, 6 2014.

[44] Ziyu Wang, Frank Hutter, Masrour Zoghi, David Matheson, and Nando de Feitas. Bayesian optimization in a billion dimensions via random embeddings. *Journal of Artificial Intelligence Research*, 55:361–387, 2016.

[45] Andrew Wilson and Zoubin Ghahramani. Copula processes. In *Advances in Neural Information Processing Systems*, pages 2460–2468, 2010.

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

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