A Framework to Search for Machine Learning Pipelines

by

Akshay Ravikumar

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

Masters of Engineering in Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2018

Author .....
Department of Electrical Engineering and Computer Science
May 25, 2018

Certified by .....
Kalyan Veeramachaneni
Principle Research Scientist, LIDS
Thesis Supervisor

Accepted by .....
Katrina LaCurts
Chair, Masters of Engineering Thesis Committee

A Framework to Search for Machine Learning Pipelines

by

Akshay Ravikumar

Submitted to the Department of Electrical Engineering and Computer Science
on May 25, 2018, in partial fulfillment of the
requirements for the degree of
Masters of Engineering in Computer Science

Abstract

In this thesis, we present DeepMining, a framework to search for machine learning pipelines. The high-level goal of DeepMining is to solve *the pipeline search problem*: given a problem and a dataset, find the pipeline best-suited to solve that problem. The DeepMining platform serves as a testbed for developers to experiment with different methods of computing and evaluating machine learning pipelines. Specifically, developers have autonomy over how to evaluate different configurations in parallel, score a pipeline given a dataset and hyperparameter configuration, and efficiently search over the pipeline space. DeepMining was designed with modularity and extensibility in mind: developers can easily implement new search algorithms, scoring functions, and computation frameworks. At the same time, users can switch between these modules with minimal effort.

Thesis Supervisor: Kalyan Veeramachaneni
Title: Principle Research Scientist, LIDS

Acknowledgments

This project was done under the supervision of Kalyan Veeramachaneni in the Data to AI group. First and foremost, I'd like to thank Kalyan for his constant feedback and support. His previous experience with DeepMining, as well as his high-level vision for the project, were instrumental to its success.

Second, I'd like to thank William Xue and Laura Gustafson, my close collaborators throughout this project, for making the refactoring process as smooth as possible. I'd also like to thank FeatureLabs, as well as the rest of the Data to AI Group, for their feedback, advice, and company. In particular, I'd like to thank Arash for his help with the graphics in this paper.

In addition, I'd like to acknowledge Xylem and the National Science Foundation for their generous funding support for this project.

Finally, I'd like to thank my mom Vani Ravikumar, my dad Ravikumar Ganapathi, and my brother Adarsh Ravikumar, without whose constant love, guidance, and support I wouldn't be where I am today.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

When confronted with a problem and dataset, data scientists aim to find the best pipeline to solve that problem. After finding the best pipeline, the data scientist also gets to make decisions regarding the hyperparameters of this pipeline, which could significantly impact its accuracy when it is fit to the data.

However, it isn't easy to find the best pipeline for a given dataset and problem, nor is it easy to manually guess the optimal hyperparameters for a given pipeline: in an ideal world, every part of this process can be automated.

Therefore, we would like to create a library to tackle what we will refer to as *the pipeline search problem*. However, because developments in this field occur rapidly, settling for any one specific algorithm or methodology would be hasty. To address this, we aim to create a platform that is easy to use, but lets contributors easily implement new ideas and frameworks. Therefore, the library serves as a testbed for data scientists, and should evolve along with the state-of-the-art.

With this in mind, we enhance DeepMining, a library to solve *the pipeline search problem*: specifically, DeepMining accepts a dataset (images, text, audio, tabular, relational, timeseries), a problem (classification, regression, collaborative filtering, clustering), and a list of candidate pipelines, and finds the optimal pipeline to solve that problem [3]. In this chapter, we specify *the pipeline search problem*, discuss the initial state of DeepMining, and summarize our contributions to the project.

1.1 Notation

To objectively specify *the pipeline search problem*, we introduce the following notation.

1.1.1 Pipeline Notation

Define a *pipeline* as a sequence of steps whose hyperparameter have not been fixed yet. For example, a pipeline could be a random forest classifier, or a four-layer neural network.

Each pipeline p has a set of hyperparameters $H(p) = \{h_1, h_2, \dots h_{|H(p)|}\}$. Let the range of some hyperparameter $h_i \in H(p)$ be $r(h_i)$.

Let $R(p) = r(h_1) \times r(h_2) \times ... \times r(h_{|H(p)|})$ denote the hyperparameter space of that pipeline. In other words, this represents every possible configuration of hyperparameters for that pipeline.

When a pipeline has fixed hyperparameters, we refer to it as a *fully specified pipeline*. For a fully specified pipeline t and hyperparameter configuration $r \in R(t)$, let $P(t, r)$ be the pipeline represented by template t and hyperparameters r.

For some pipeline p and input X, let $p(X)$ be the predictions when X is inputted to p: these might be class labels, float values, etc.

1.1.2 Problem Notation

Assume we are given a dataset $X_1, X_2, \dots X_n$ of m-dimensional vectors, along with a vector $y_1, y_2, \dots y_n$ of labels. If not already specified, we can divide this data into training and validation sets.

In addition, we define the problem type b as the type of predictions we need to perform: this might include binary classification, multiclass classification, regression, etc.

Finally, we define a scoring function $g(p)$ that assigns a score to a pipeline: for example $g(p)$ might compute predictions $p(X_{\text{test}})$ on a test set, and compare them to the *true* labels y_{test}. Depending on the problem, we might want to maximize or

minimize this score, although this is not an important distinction: without loss of generality, we will assume we wish to maximize g.

When we refer to *the pipeline search problem*, we generally mean the following: given (X, y, b, g), find the pipeline that optimizes g with respect to the dataset (X, y) and problem type b.

Specifically, given a given problem and dataset, let $P(X, y, b)$ be the space of all pipelines that might potentially solve the problem: in other words, they accept data in the same format as X and y, and the pipeline is well-suited to solve problem b. Finding $P(X, y, b)$ is difficult, because it requires a knowledge of large space of possible pipelines and detailed metadata about each pipeline. This problem has not been fully solved in DeepMining.

Once we have the space of potential pipelines (a smaller subset), we can perform hyperparameter optimization on each pipeline. Therefore, considering every possible pipeline and hyperparameter configuration, we wish to compute the following:

$$\text{argmax}_{p \in P(X,y,b), r \in R(p)} \; g(P(p, r))$$

Of course, we cannot always search through the entire hyperparameter space, nor do we have access to every possible pipeline that can solve the problem at hand. Therefore, we must approximate in both dimensions.

1.2 Related Work

The high-level goal of DeepMining is akin to that of AutoML efforts today, which aim to automate every aspect of the machine learning process: these include feature engineering, architecture search, and hyperparameter search. We summarize efforts in each of these fields in the following sections, and discuss how DeepMining's goals align with them.

Feature engineering: At a high level, feature engineering is the process of parsing the input data, and generating salient features to perform predictions on. Feature

engineering can be done in conjunction with model selection: if the features are well-engineered, a model might perform better, or it might be easier to find a sufficiently accurate model.

Usually, feature engineering is a very time-consuming process, where data scientists manually generate and evaluate different feature representations. Modern-day research focuses on automating the feature engineering process: for example, the Data to AI group has created Deep Feature Synthesis, which automatically infers relationships in the data, and programmatically performs transformations to extract valuable features [4].

Architecture search: Another difficult aspect of machine learning is model selection: oftentimes, this involves tampering with meta-hyperparameters involving the layers in the model, which is a difficult process to conduct manually.

Current research focuses on two levels of this problem: automatically generating model architectures, and allocating resources given a fixed set of candidate architectures. For example, Zoph. et al. take a reinforcement learning approach to generating candidate neural network architectures, by training over the space of model descriptions [11]. Contrarily, the TuPAQ system accepts the search space of architectures as input, and performs a bandit-based hyperparameter tuning procedure, where resources are allocated to the most promising candidates [7].

In DeepMining, we focus on the latter: using a multi-armed bandit to intelligently decide which architectures to focus resources on. We discuss this further in Chapter 5.

Hyperparameter tuning: Hyperparameter tuning is one of the most important parts of the machine learning process: without an efficient means of searching the hyperparmaeter space, data scientists might resort to default parameters, or produce suboptimal architectures. The machine learning community has done extensive work on making this process more efficient by training a model over the hyperparameter space. This both automates the process, and increases efficiency by reducing the search space.

There are a variety of approaches to the hyperparameter tuning problem: these include Bayesian optimization and bandit-based approaches [8, 6].

Hyperparameter search has been a key goal of DeepMining, and continues to be a major focus in the next iteration. We discuss this further in Chapter 5.

1.3 Initial State of DeepMining

Over the last few years, students in the Data to AI group created an initial implementation of DeepMining, discussed in [3]. In short, the original implementation allowed data scientists to define an arbitrary pipeline using the `Pipeline` object from `scikit-learn` (allowing for custom-implemented functions, not necessarily imported from `scikit-learn`). Then, developers can expose the type, name, and range of each of its hyperparameters, and pass this information to a `DeepMining` function along with a dataset and a scoring function. Then, the system performs the following:

- Trains a novel implementation of Bayesian hyperparameter optimization with a Gaussian Copula Process.

- If the user chooses to, evaluates each hyperparameter configuration using the Bag of Little Bootstraps (BLB) method, proposed in [5], which evaluates bootstraps created from subsamples of the dataset. The user has the option to parallelize these computations, either using Spark or the Pathos Multiprocessing module. The specifics of BLB will be discussed in Chapter 4.

- Feeds this information back into the hyperparameter model, which then proposes new candidates to explore.

The authors make the interesting observation that Bag of Little Bootstraps, when combined with a Gaussian Copula Process, provides significant speedup without compromising accuracy. While doing so, they provide an abstraction that is convenient for developers to perform hyperparameter optimization on any pipeline.

1.4 Goals for the New Iteration of DeepMining

While the initial iteration of DeepMining provided a valid proof of concept, we believe there could be improvements, both in terms of functionality and design. These are discussed in the following sections.

Feature goals: The existing iteration of DeepMining only performs hyperparameter optimization on one given pipeline. However, we wish to go one step further, and perform optimization over any number of feasible pipelines. This enables us to allocate resources efficiently on two levels: choosing pipelines to explore, and hyperparameters to explore for each pipeline.

Design Goals: In addition, we believe the platform could be made more extensible, to make it easier for developers to contribute new modules. With this in mind, we have the following design goals:

- **Isolation:** First, we want to separate distinct functionalities into different modules: this way, they can be tested and developed independently. As an example, scoring and searching can be isolated from each other. In addition, abstracting out these modules might benefit other open-source projects, including those developed by the Data to AI Group, because they can be reused.

- **Modularity:** Machine learning is an ever-expanding field, and new techniques are developed regularly: for this reason, we want to make the codebase as modular as possible. If a decision to use a particular algorithm or framework is arbitrary, then we strive to make it an extensible module. This way, developers can easily implement new algorithms and frameworks, and users can easily switch between these implementations.

  On a similar vein, machine learning pipelines consist of several steps, and these steps might be reusable between different pipelines. Therefore, we wish to provide the capability to represent these different steps in a simple, composable way.

- **Flexibility:** The API for DeepMining should be simple enough that it can

support a variety of frameworks. For example, it should support a variety of machine learning frameworks like Keras, TensorFlow, etc. instead of just supporting `scikit-learn`.

- **Simplicity:** Finally, DeepMining needs to provide enough functionality that developers can easily add new modules to the framework. Developers should only have to focus on the core logic of the module when contributing to DeepMining.

Achieving these design goals involves several tradeoffs. For example, the more rigid the abstraction, the easier it is for DeepMining to provide facilities specific to the implementation. However, if the abstraction is too constrained, developers lose the flexibility to implement novel algorithms.

1.5 Contributions

We kept these goals in mind when designing the next iteration of DeepMining. In this redesign, we separated two functionalities into independent repositories: an abstraction for defining machine learning pipelines and fitting them to data, and a library for Bayesian hyperparameter tuning and multi-armed bandits. We still have a core DeepMining library, which aims to solve *the pipeline search problem*. The DeepMining library, the primary focus of this thesis, directly depends on the other two repositories. In this thesis, we present the current state of the redesigned codebase, along with the major enhancements to the DeepMining library.

1.6 Outline

In Chapter 2, we summarize the refactored DeepMining codebase. In Chapter 3, we discuss different datatypes and input formats, and how they should be represented in DeepMining. Chapter 4, Chapter 5, and Chapter 6 describe the `ScorePipeline`, `DeepMineSearch`, and `Compute` modules, which comprise my major contributions to the project. Next, Chapter 7 explores the the performance benefits of switching

between different score, search, and compute modules. Chapter 8 offers concluding remarks, and discusses potential future work. Finally, Appendix A contains API tables for the `ScorePipeline`, `DeepMineSearch`, and `Compute` modules.

Chapter 2

Overview of the refactored code

We had four design goals in mind when refactoring DeepMining: isolation, modularity, flexibility, and simplicity. With these design goals in mind, we refactored the DeepMining codebase into three major repositories, described below.

- **MLBLocks**, which lets users create machine learning pipelines.
- **BTB**, or Bayesian Tuning and Bandits, which offers functions to solve the multi-armed bandit problem and perform hyperparameter optimization.
- **DeepMining**, which lets users specify a machine learning problem, and finds the pipeline best-suited to solve that problem. This thesis focuses on DeepMining.

In this chapter, we discuss the design and function of each of these libraries, and how they depend on each other.

2.1 MLBlocks

The `MLBlocks` library lets users specify data transformation functions (blocks), and compose these primitives to form pipelines. This library consists of three major parts: `MLHyperparam`, `MLBlock`, and `MLHyperparam`. These will be discussed in the subsequent sections.

MLHyperparam: The `MLHyperparam` object lets users specify tunable hyperparameters of the following types: integer, float, string, and categorical. The object simply contains a name, type, and range: for example, a float might have the range $[0, 1]$, and a categorical parameter might have the range $[2, 6, 10]$. Each hyperparameter also contains a value, which is randomly initialized.

MLBlock: The `MLHyperparam` object represents a machine learning primitive. Most importantly, a `MLBlock` contains the following attributes:

- `model`: The model or function that the `MLBlock` executes.
- `fit(X, y)`: A function that fits `model` to labelled data.
- `produce(X)`: A function that returns predictions from `model`, given new unlabeled data points.
- `tunable_hyperparams`: A list of `MLHyperparam` objects, indicating the parameters in `model` that can be tuned.
- `fixed_hyperparams`: A list of hyperparameters that do not need to be tuned, but whose values can be modified by the user. For example, input dimension might be a fixed hyperparameter.

A user can create a `MLBlock` by creating a Python file containing the model, as well as the `fit` and `produce` methods, then creating a JSON file pointing to these functions. In addition, the JSON contains a list of tunable hyperparameters, along with their types and ranges. See Figure 2-1 for an example JSON, depicting the histogram of oriented gradients (HOG) model for image processing.

Depending on the library used, MLBlocks might use a different JSON parser. For example, MLBlocks has a specific parser for `scikit-learn` and Keras. The parser for `scikit-learn` is simple; however, because Keras primitives cannot be composed in the same way, a Keras JSON needs to include every layer the model consists of.

We choose the JSON abstraction for two reasons. First, contributors can easily add new `MLBLock` objects to the library, and users can easily compose these JSON

```
{
    "name": "HOG",
    "class": "mlblocks.components.functions.image.HOG.HOG",
    "fit": "fit",
    "produce": "make_hog_features",
    "hyperparameters": {
        "num_orientations": {
            "type": "int",
            "range": [9, 9]
        },
        "num_cell_pixels": {
            "type": "int",
            "range": [8, 8]
        },
        "num_cells_block": {
            "type": "int",
            "range": [3, 3]
        }
    }
}
```

Figure 2-1: A simplified example of an `MLBlock` JSON.

files to create a `MLPipeline` object, which will be discussed in the next section. In addition, developers have autonomy over how they implement the model, fit, and produce methods, as long as they provide a JSON pointing to those methods. Finally, developers can easily modify which hyperparameters to tune, along with the ranges for these hyperparameters.

MLPipeline: The `MLPipeline` class lets users compose primitive `MLBlock` objects to create machine learning pipelines. See Figures 2-2 and 2-3 for a simple visualization.

At the simplest level, users can simply initialize an `MLPipeline` with a list of JSON file paths, each referring to a different `MLBlock`. See Figure 2-4 for a simple example. In general, `MLPipeline` assumes that all blocks are executed sequentially, although the abstraction can be easily modified to support more complex data flows.

Over the course of this year, we have implemented a library of `MLBlock` and `MLPipeline` objects. These pipelines that support a variety of problems, such as classification and regression. In addition, these pipelines cater to a variety of data types like image, audio, text, and graph. These will be discussed further in Chapter 3.

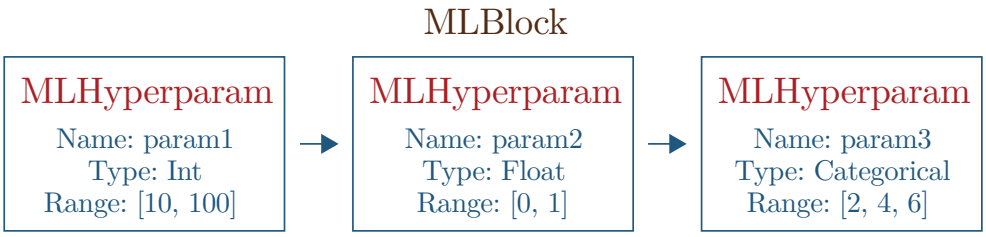

Figure 2-2: An `MLBlock` object.

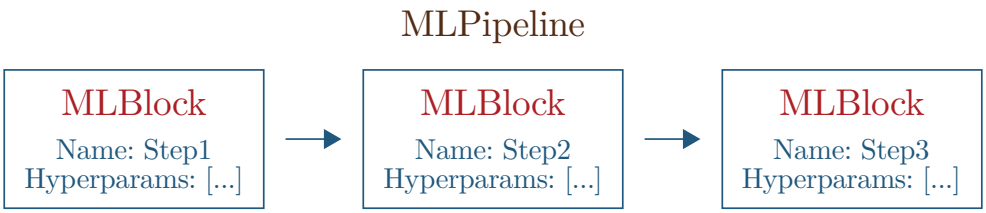

Figure 2-3: An `MLPipeline` object.

```
class TraditionalImagePipeline(MLPipeline):
    """
    Traditional image pipeline using HOG features.
    """
    def __new__(cls, *args, **kwargs):
        return MLPipeline.from_ml_json([
            'HOG',
            'random_forest_classifier'
        ])
```

Figure 2-4: A simple example of an `MLPipeline` object.

```
from btb.tuning import GP
from btb import HyperParameter, ParamTypes

# We are tuning a random forest classifier.
model = RandomForestClassifier()

# Initialize the hyperparameters along with their ranges.
tunables = [
    ('n_estimators', HyperParameter(ParamTypes.INT, [10, 500])),
    ('max_depth', HyperParameter(ParamTypes.INT, [3, 20]))
]

# Initialize a Gaussian Process (GP) tuner.
tuner = GP(tunables)


# Get a proposal for the next hyperparameters to consider.
parameters = tuner.propose()

# Set these hyperparameters in the model.
# This method is hypothetical, for demonstration purposes.
model.set_hyperparameters(parameters)

# Generate a score for this hyperparameter configuration.
score = model.score(X_test, y_test)

# Pass this information along to the tuner, which will
# update its model and propose new hyperparameters.
tuner.add(parameters, score)
```

Figure 2-5: An example usage of a BTB tuner.

2.2 BTB

The Bayesian Tuning and Bandits (BTB) library, also developed by the Data to AI group, offers tuners and selectors to aid in the model tuning process. These two modules are described below.

Tuners: BTB tuners create models over the hyperparameter space, and propose configurations to evaluate next. Each tuner accepts a list of hyperparameters, each with a particular type (integer, float, string categorical, etc.) and range. The tuner accepts a list of hyperparameter configurations and their scores, and learns a model over the space, which it uses to propose new candidate hyperparameters. BTB offers tuners like uniform, Gaussian Process, and Gaussian Copula Process. Refer to Figure 2-5 for a hypothetical usage of a BTB tuner.

Selectors: BTB selectors help solve the multi-armed bandit problem. Specifically, a selector takes in a discrete list of choices, and scores associated with those choices,

```
from sklearn.svm import SVC
from btb.selection import UCB1

# Assume we are selecting between a random forest classifer and a
# suppoer vector machine.
models = {
    'RF': RandomForestClassifier,
    'SVC': SVC
}

# We use a Upper Confidence Bound bandit, provided by BTB.
selector = UCB1(['RF', 'SVM'])

# Assume we run the RandomForestClassifier on several different
# hyperparameter configurations, and return a list of scores.
rf_scores = [....]

# Similarly, we have several scores for the SVC model.
svc_scores = [....]

# This uses the scores and proposes the model to consider next,
# either RF or SVC.
next_model = selector.select({'RF': rf_scores, 'SVC': svc_scores
    })
```

Figure 2-6: An example usage of a BTB selector.

and proposes the next choice to explore. To accomplish this, the selector makes the tradeoff between exploration and exploitation, deciding whether to try new choices or focus on the most promising choices. BTB offers several selectors, including uniform, hierarchical, and Upper Confidence Bound (UCB). Refer to Figure 2-6 for a hypothetical usage of a BTB selector. In the case of DeepMining, selectors can be used to choose between different pipelines given their scores.

Relevance to DeepMining: When applying BTB to DeepMining, we might use selectors to choose from a set of potential pipeline templates, and tuners to choose from the hyperparameter space for each pipeline. Figure 2-7 depicts an example of how the selectors and tuners might be used, assuming the search space consists of two pipelines. We implement this concept as the `SearchBTB` module in DeepMining, as discussed in Chapter 5.

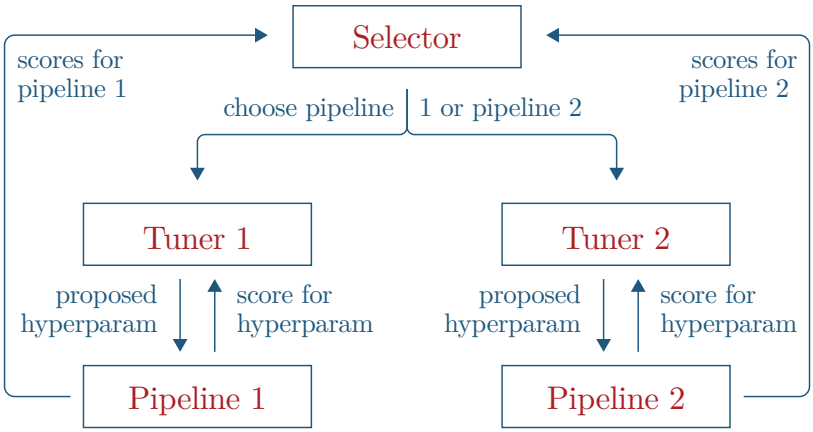

Figure 2-7: How BTB might be used for hyperparameter optimization over a search space of several pipelines.

2.3 DeepMining

The DeepMining module, which is the focus of this thesis, aims to solve any machine learning problem. The library accepts a dataset and a list of candidate pipelines, and performs hyperparameter optimization over those pipelines to find the optimal pipeline. To aid in this process, DeepMining has three main modules: `ScorePipeline`, `DeepMineSearch`, and `Compute`. Each of these modules is easily extensible by developers: we summarize each of these modules below, and focus on specific details in subsequent chapters.

ScorePipeline: The `ScorePipeline` class takes in a fitted pipeline, a dataset, and a scoring function, and returns a score. Depending on the implementation, this score may be computed in several ways: using cross validation, bootstrapping, subsampling, etc.

DeepMineSearch: The `DeepMineSearch` class takes in a dataset, a problem type, a scoring function, and a list of candidate pipelines, and tries to find the optimal pipeline and hyperparameters. The search algorithm might be a simple grid search, or it might allocate resources more intelligently.

Compute: The `Compute` class lets users perform simple MapReduce computations on any computation framework: these may include Dask, Spark, Pathos MultiProcessing, etc. This module helps speed up embarrassingly parallel computations in the

`ScorePipeline` and `DeepMineSearch` classes.

2.4 Overall workflow

In this section, we summarize the overall workflow of DeepMining, thereby demonstrating how all these modules interact. See Figure 2-8 for the dependencies between these modules, explained below. Each item in this list corresponds to its respective label in the figure.

1. First, the `DeepMineSearch` accepts a list of candidate `MLPipeline` objects to consider.

2. A BTB selector might help search over the pipeline space, allocating resources to pipelines that seem to perform better. Similarly, for a given pipeline, a BTB tuner might help search over the hyperparameter space, proposing interesting candidates to evaluate next. Note that while using BTB isn't required, it might help developers when writing search algorithms.

3. The `ScorePipeline` assigns a score to a potential pipeline and hyperparameter configuration. If using BTB, a developer can use this score to train the tuners and selectors.

4. Evaluating multiple hyperparameter configurations can easily be done in parallel, using the `Compute` class.

5. A scoring algorithm might require running the pipeline on independent bootstraps or subsamples. Because these are embarrassingly parallel, this can easily be parallelized using the `Compute` class.

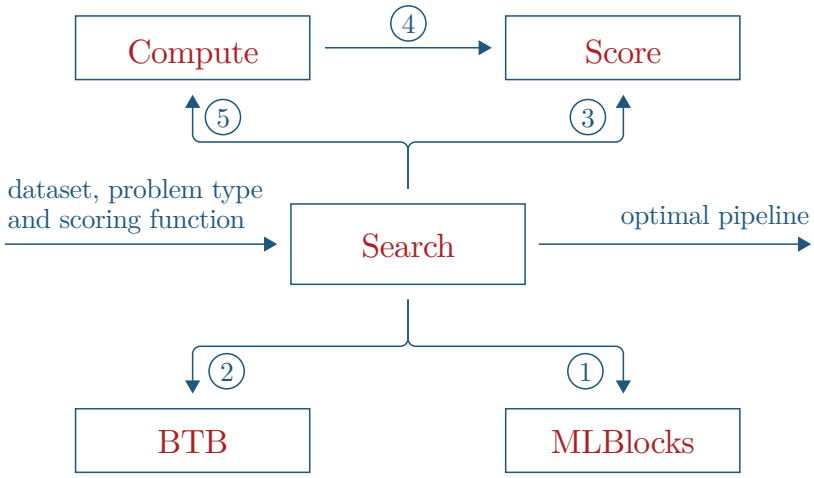

Figure 2-8: The `DeepMining` workflow. Each of the labeled relationships corresponds to its respective list item in Section 2.4.

Chapter 3

Data Input

The DeepMining system works on a variety of data formats: these include image, audio, text, single-table, multi-table, graphs, and multi-graphs. This classification of datatypes is motivated by the Data-Driven Discovery of Models (D3M) program, a program that aims to further automated machine learning efforts [1]. The Data to AI Group, along with FeatureLabs [2], participates in a competition hosted by D3M to essentially solve *the pipeline search problem*. In this competition, teams are given a dataset, along with a problem type, and are tasked with outputting a model that accurately solves that problem.

The D3M competition provides datasets in a very specific format, and each dataset falls under one of these datatypes. As many of the pipelines in MLBlocks were written for the D3M competition, we found this a natural way to classify the different datatypes one might encounter in DeepMining. We aim to support a variety of datatypes to support any pipeline search problem a data scientist might wish to solve.

As each of these datatypes can be represented in a variety of ways, agreeing on a standardized representation is crucial. Specifically, the `fit` method in the `MLPipeline` class accepts three arguments: the samples (`X`), the labels (`y`), and optional auxiliary data (`fit_params`). In this chapter, we summarize each of these datatypes, and how they should be represented when inputted to DeepMining.

3.1 Image

For image pipelines, we expect the input format to be that of a matrix, with one row for each image. These can be simple RGB values loaded from image files, or a flattened binary representation in the case of grayscale images. See Figure 3-1.

As an example, we have implemented pipelines that work on the MNIST dataset.

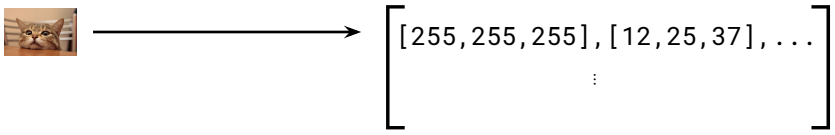

Figure 3-1: Format for the image datatype. In this example, each image is a 3-dimensional matrix of dimension height $\times$ width $\times$ 3, where the final dimension is RGB values. We concatenate the rows of each image, resulting in a (height $\times$ width) $\times$ 3 matrix for each image.

3.2 Audio

The audio pipelines implemented thus far ingest audio data from WAV files, which can be decomposed into a list of segments. Therefore, we expect each training example to be a list of lists, each of which represents a segment. See Figure 3-2.

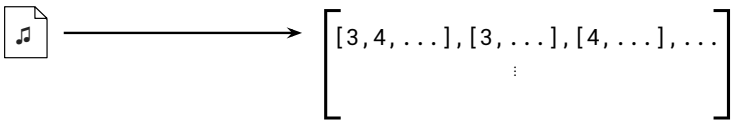

Figure 3-2: Format for the audio datatype. Assuming each sample is loaded from a WAV file, each training example is a list of lists, each of which represents a segment.

3.3 Text

For text pipelines, we expect each training example to be one ASCII string representing the text. If the text is comprised of multiple lines or words, they should be concatenated. See Figure 3-3.

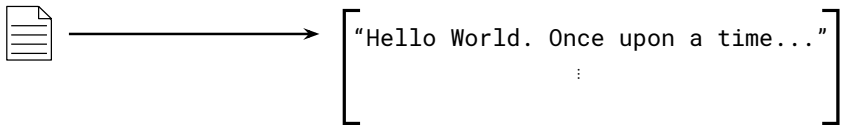

Figure 3-3: Format for the text datatype. Every example should be one concatenated string: if the text is divided into multiple lines or words, these should be concatenated.

3.4 Single-Table

The single-table datatype represents any matrix data that doesn't fall into the previous categories: this data is already provided as matrix, so no special transformations are necessary. See Figure 3-5.

As an example, we have a dataset of housing prices in Boston, offered by `scikit-learn`.

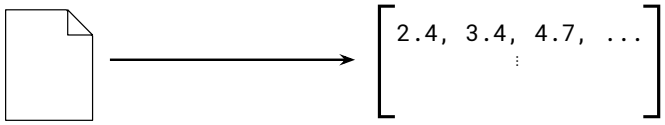

Figure 3-4: Format for the single-table datatype. This data is already inputted as a matrix, so no specific transformations are necessary.

3.5 Multi-Table

The multi-table datatype refers to matrix data that is augmented with extra matrix data, which shares a key with the primary matrix. For example, the primary key of an auxiliary matrix might be a foreign key in the primary matrix. In this case, `X` refers to primary matrix, we let `fit_params` contain the auxiliary matrix data, along with metadata about the foreign keys and primary keys. See Figure **??**.

For example, we have a dataset of retail store data, where the primary table stores transactions, and an auxiliary table stores information about the products involved in each transaction.

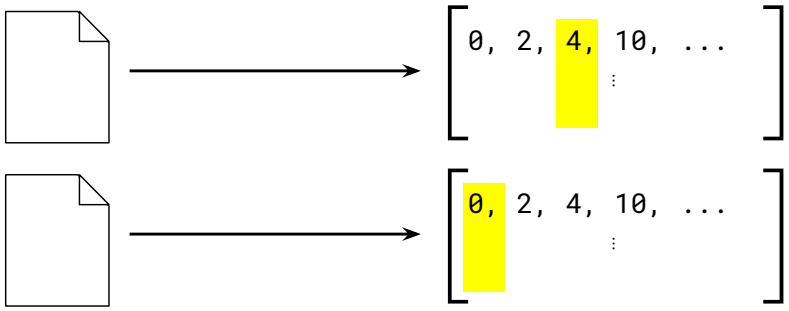

Figure 3-5: Format for the multi-table datatype. The primary matrix is provided as `X`, while any auxiliary matrix data is provided in `fit_params`. In addition, we must provide metadata about the foreign keys in the primary table, and which auxiliary tables they refer to.

3.6 Graph

The graph datatype refers to matrix data that is augmented with a graph, depicting the relationship between samples in the matrix. The graph data structure is stored as a nested data structure `D` (likely a Python dictionary), where `D[a][b]` stores auxiliary information about the relatonship between rows `a` and `b`. For this datatype, we let `X` refer to the primary matrix, while `fit_params` contains the graph represented as a Python dictionary. See Figure 3-7.

As an example, we have a dataset of Amazon products, along with an auxiliary data structure depicting which products are frequently co-purchased.

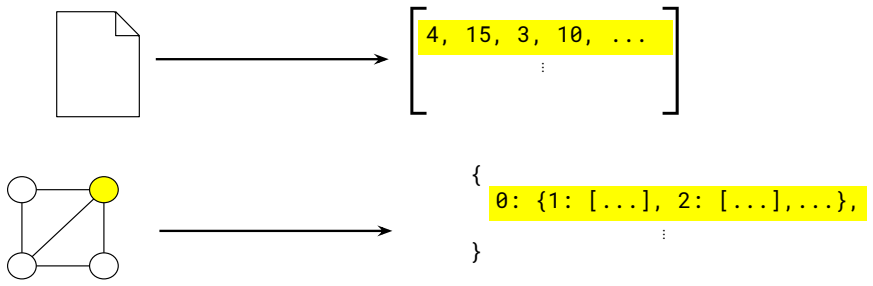

Figure 3-6: Format for the graph datatype. The primary matrix is provided as `X`, while the auxiliary graph is represented as a nested data strcutre in `fit_params`.

3.7 Multi-Graph

Similarly, the multi-graph datatype refers to matrix data that is augmented with multiple graphs, depicting the relationship between rows in the matrix. For this datatype, we let `X` refer to the primary matrix, while `fit_params` contains a list of graphs, each represented as a Python dictionary. See Figure **??**.

In our experience, multi-graph datasets are often used to accompany graph-matching problems. As an example, consider a Facebook user dataset, where the rows depict users, two auxiliary graph structures depict relationships between users, and the problem posed is graph matching.

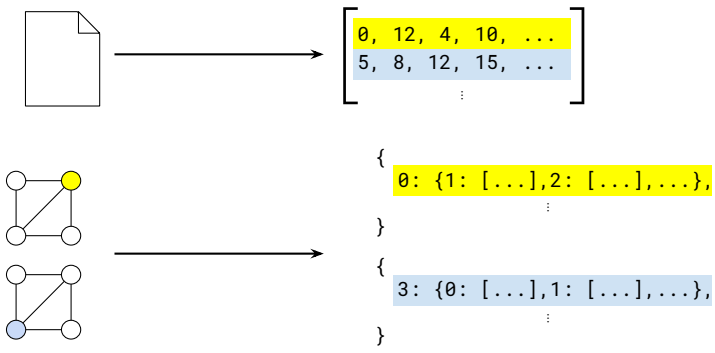

Figure 3-7: Format for the multi-graph datatype. Format for the graph datatype. The primary matrix is provided as `X`, while the auxiliary graphs are represented as a list of Python dictionaries in `fit_params`.

Chapter 4

Score

When evaluating a dataset for a given pipeline and hyperparameter configuration, we choose the optimal configuration based on a particular scoring function. This might be an accuracy score, f1-score, least-squares loss, etc. When solving *the pipeline search problem* with DeepMining, a user needs to provide the scoring function used to measure accuracy.

However, with large datasets, it might be unfeasible to score every candidate pipeline using the entire dataset. When refactoring DeepMining, we realized that while the search function requires the score for a given pipeline and hyperparameter configurationt, it is agnostic to *how* this score is computed: by this, we refer to how the data is partitioned or subsampled, and how these intermediate scores are coalesced. As a result, we abstract this functionality into the `ScorePipeline` class. This way, developers can easily create new scoring methods, and users can switch between these methods with minimal effort.

In this chapter, we discuss the design of the `ScorePipeline` class, demonstrate potential use cases, and justify that this abstraction suffices for most searching and scoring algorithms

4.1 The `ScorePipeline` class

To implement a scoring method in DeepMining, a developer need only subclass the `ScorePipeline` class, which requires implementing the following methods. For a summary, refer to Table A.1.

- The `__init__` function, which initializes a particular scoring method. This method accepts arguments specific to the computation framework: these might include the number of subsamples, size of each subsample, etc.

- The `score` function, which evaluates a pipeline given training data, validation data, and a scoring function. Specifically, this method accepts the following arguments:

  - `pipeline`: The `MLPipeline` object to be evaluated: developers should set any relevant hyperparameters before passing a pipeline into this function.

  - `X_train`, `y_train`, `X_val`, `y_val`: The training data, training labels, validation data, and validation labels, represented as `numpy` array-like objects. For more information about how to represent different datatypes in a standardized format, refer to Chapter 3.

  - `score_func`: The scoring function to use, which takes in a test set (`X`) and predictions (`y`), and returns a score. If `score_func` is `None`, this method uses a default scoring function. For example, this might include functions like F1 score, precision score, etc., many of which are available in `sklearn.metrics`.

This abstraction should be sufficient to implement any scoring method: as long as developers have access to the pipeline, dataset, and scoring function, they have complete autonomy over how to perform the computation.

4.2 Implemented Scoring Methods

To demonstrate the feasibility of the `ScorePipeline` class, we implement three scoring methods: `ScoreDefault`, `ScoreCV`, and `ScoreBLB`.

ScoreDefault: In the `ScoreDefault` class, we implement a basic scoring method that fits the pipeline on the training data, and runs the scoring function the validation data.

ScoreCV: The `ScoreCV` class runs k-fold cross validation (defaulting to $k = 3$) using the train and validation data, and returns the average score of all k trials.

To elaborate, k-fold cross validation involves dividing a dataset D into k *folds* $d_1, d_2, \ldots d_k$. For each $i = 1 \ldots k$, we fit the model on all but d_i, and validate using d_i, to generate some score s_i. Finally, we return $1/k \sum_{i=1}^{k} s_k$.

ScoreBLB: In the `ScoreBLB` class, we implement the Bag of Little Bootstraps (BLB) method proposed in [5]. BLB enables the approximation of scores with the same theoretical guarantees as the bootstrap. Bootstrapping involves running the model on resamples of the data; however, because the resamples are around 63% the size of the original dataset, it becomes unfeasible to use the bootstrap in a massive data setting. BLB mitigates this by adding another layer of subsampling, first creating a "bag" from which bootstraps are created. This approach provides similar guarantees to that of the bootstrap, while significantly reducing computational overhead.

BLB works as follows: consider a dataset with n elements, and choose a "bag" of size $n^{0.5} \leq b \leq n$ without replacement. Then, sample from this bag with replacement to create a bootstrap of size n. The score returned by BLB is the averages of the scores of each resample.

As another optimization, instead of actually repeating elements with a resample, we simply weigh the elements of the bag by a multinomial sample $n_1, n_2, \ldots n_b$ such that $\sum_{i=1}^{b} n_i = n$. Luckily, many estimators already support a weighted data representation (specifically represented by the `sample_weight` parameter in `scikit-learn`), so the implementation is not difficult.

Therefore, BLB only operates on b data points instead of n: in particular, the

authors claim that $b = n^{0.6}$ is a good value for b. Note that as n gets very large, the benefits of this approach are clear. When $n = 10^9$, for example, using a bootstrap might be unfeasible: however, Bag of Little Bootstraps only operates on approximately $250,000$ points.

See Algorithm 1 for pseudocode of the algorithm. Because the original implementation of DeepMining relied on BLB, and because of its significant guarantees on large datasets, we decided to implement BLB in the new iteration of DeepMining as well. Users can provide values for γ (such that $b = n^\gamma$), r, and s, although they are set to default values recommended by the authors.

Algorithm 1 Bag of Little Bootstraps (BLB)

Input: $X_{1...n}$: data points, b: bag size, s: number of bags,
r: number of samples in each bag, E: scoring function,
y: desired function/statistic
Output Estimate of y

for $j = 1$ to s **do**
  Randomly sample b data points without replacement, forming a `bag`.
  $X^j_{1...b} \sim \texttt{random}(X_{1...n})$
  for $k = 1$ to r **do**
    ▷ Sample the counts
    $(n_1, n_2, \dots n_b) \sim \texttt{Multinomial}(n, \mathbf{1}_b/b)$
    ▷ Estimate y on the bootstrap sample using X^j and the counts as weights
    $est_k \sim E(X^j_{1...b}, n_1, \dots n_b)$
  end for
  $y_j = r^{-1} \sum_{k=1}^{r} est_k$
end for
return $s^{-1} \sum_{j=1}^{s} y_j$

4.3 Simplicity of Implementation

Implementing a new scoring method using the `ScorePipeline` class is reasonably simple. For example, `ScoreDefault` and `ScoreCV` required less than 50 lines of code, and `ScoreBLB` required significantly less than 100 lines of code (see Table 4.1). This is because Bag of Little Bootstraps aligns well with the functionalities provided by `sklearn`, `MLBlocks`, and `ScorePipeline` classes.

Class	Number of Lines
`ScorePipeline` (base class)	47
`ScoreDefault`	23
`ScoreCV`	40
`ScoreBLB`	96

Table 4.1: The number of lines needed to implement each of the scoring methods.

As another optimization, note that in BLB, each bootstrap is evaluated in an embarrassingly parallel fashion. Therefore, using a computation framework like Spark, or a simple threading library, can significantly improve performance. To address this, we add the `Compute` library to DeepMining, which lets users perform simple MapReduce computations agnostic to the framework used: modifying the implementation to support parallel computations requires less than ten lines of code. This optimization will be discussed in Chapter 6.

Therefore, developers can easily use the `ScorePipeline` abstraction to implement new scoring methods in DeepMining.

4.4 Simplicity of Application

In addition, developers can easily use switch between different scoring methods when running DeepMining. See Figure 4-1 for an example. This is particularly useful when implementing searching algorithms, which will be described in Chapter 5.

```
# Import a pipeline from the MLBlocks library.
pipeline = TraditionalImagePipeline()

# We use the F1 scoring function from scikit-learn.
def score_func(*args):
    return sklearn.metrics.f1_score(*args, average='micro')

# Fit the pipeline to the data, and score using
# cross-validation.
scorer = ScoreCV()
score = scorer.score(
    pipeline,
    X_train, y_train, X_val, y_val,
    score_func=sklearn.metrics.accuracy_score
)

# Fit the pipeline to the data, and score using
# Bag of Little Bootstraps,
scorer = ScoreBLB(gamma=0.7, s = 5, r = 10)
score = scorer.score(
    pipeline,
    X_train, y_train, X_val, y_val,
    score_func=score_func
)
```

Figure 4-1: Switching between different scoring methods.

Chapter 5

Search

The overarching goal of DeepMining is the following: given a dataset, a problem, and a scoring function, find the pipeline that optimizes that score. This is a difficult problem because it involves several layers: finding pipelines that might be relevant to the problem, and performing hyperparameter optimization on each of these pipelines. Thoroughly searching such a large space is unfeasible, as it involves fitting each candidate pipeline to the entire dataset. If the search space is large, even using an algorithm like Bag of Little Bootstraps might not suffice. Therefore, the tradeoff between exploration and exploitation is crucial: a developer needs to know which candidates to consider, and how much resources to allocate to each candidate.

Finding which pipelines to consider for a given dataset and problem is a difficult task: it involves metadata about each pipeline to decide whether it is compatible with a dataset, and whether it might perform well for a particular problem or scoring function. We continue this discussion in Chapter 8, which discusses future work: in this chapter, we focus on tuning hyperparameters given a predefined list of candidate pipelines.

The original implementation of DeepMining relied on a Gaussian Copula Process (GCP) to perform hyperparameter optimization. However, efficient hyperparameter optimization becomes increasingly important as machine learning dominates the tech industry, and new algorithms are proposed on a regular basis. Therefore, while GCP might exceed baseline performance, we decided that the implementation of DeepMining

shouldn't depend on any particular search algorithm; instead, it should adjust to advancements in the field. To address this, we abstract out search functionality using the `DeepMineSearch` class.

In this chapter, we discuss the design of the `DeepMineSearch` class, demonstrate potential use cases, and justify that this abstraction suffices for most searching algorithms.

5.1 The `DeepMineSearch` Class

To implement a search algorithm in DeepMining, a developer need only subclass the `DeepMineSearch` class, which requires implementing two methods. In addition, the developer has access to some helper functions that might help write efficient search algorithms. We discuss these in the following two sections.

5.1.1 User-Facing Methods

To implement a search algorithm, a developer needs to implement the `__init__` and `search` functons, described below. These two methods are user-facing: a user can initialize a `DeepMineSearch` object using this schema, then perform hyperparameter search by calling the `search` function. For a summary of these methods, refer to Table A.3.

- The `__init__` function, which simply initializes the search algorithm with a given dataset, problem, and scoring function. The function takes in the following arguments:
  - `scorer`: The scoring method, which must implement the `ScorePipeline` class. For example, this could be Bag of Little Bootstraps (`ScoreBLB`) or cross-validation (`ScoreCV`).
  - `X_train`, `y_train`, `X_val`, `y_val`: The training data, training labels, validation data, and validation labels, represented as `numpy` array-like objects.

For more information about how to represent different datatypes in a standardized format, refer to Chapter 3.

  - `score_func`: The scoring function to use, which takes in a test set (`X`) and predictions (`y`), and returns a score. If `score_func` is `None`, this method uses a default scoring function. For example, this might include functions like F1 score, precision score, etc., many of which are available in `sklearn.metrics`.
  - `search_space`: The list of functions to tune, represented as a list of `MLPipeline` objects. To reiterate, an `MLPipeline` might be constructed in two ways: first, it might be a predefined pipeline, such as the `TraditionalImagePipeline`. In addition, an `MLPipeline` might be constructed by directly composting `MLBlock` primitives, by referring to the JSON files specified for each one. For example, `MLPipeline.from_ml_json(['HOG', 'random_forest_classifier'])` represents a pipeline consisting of the histogram of oriented gradients (HOG) primitive, followed by the random forest classifier.
  - `output_dir`: Where to save logs during the search procedure. Defaults to `saved_models/`.

- The `search` function, which performs hyperparameter optimization over a given list of candidate pipelines. Here, a developer might request algorithm-specific arguments, such as the number of iterations, or the number of hyperparameter candidates to evaluate in parallel.

5.1.2 Developer-Facing Methods

When designing the `DeepMineSearch` class, we first considered a *pull* architecture, where the developer yields one hyperparameter configuration at a time (perhaps using a Python generator), and the `DeepMineSearch` class automatically scores these configurations and logs information about them. However, we instead decided to implement a *push* architecture, where the developer has autonomy over when to

score or save hyperparameter configurations. This gives the developer freedom when deciding how many configurations to run, and when to save information about those configurations. In addition, not confining to a generator lets developers implement arbitrarily complex algorithms.

Therefore, as developers are implementing a search algorithm, they have the option to score a hyperparameter configuration at whatever point they desire. In addition, they can also save information to the *history*, which stores information about each configuration. This information is also periodically persisted in the `summary.json` file: see Table 5.1 for the log schema, and Figure 5-1 for an example log entry.

Key	Value
`pipeline_filepath`	The filepath of the pickled `MLPipeline` object
`params`	A JSON containing the hyperparameters for the `MLPipeline` object in the form `{key:value}`.
`pipeline_steps`	A list of the `MLBlock` primitives comprising the `MLPipeline` object. Each of these primitives has an associated JSON file in the MLBlocks repository.
`partition_size`	If the pipeline was run on a subsample of the data, indicates the size of the subsample. If it was run on the entire dataset, `partition_size` is `null`.
`score`	The score received by this pipeline on the (subsample of) the dataset, as returned by the `scorer`.

Table 5.1: The schema for the JSON log.

In addition to the user-facing methods, the `DeepMineSearch` class also offers the following two methods, which developers might find useful when implementing search algorithm. For a more detailed description of these two functions, refer to Table A.2.

- The `score` method, which scores a pipeline and hyperparameter configuration on the dataset. This method accepts the following parameters:
  - `pipeline`: The `MLPipeline` object to score.
  - `hyperparams`: The hyperparameters to pass into the pipeline, represented a dictionary from name to value. For example, `{'criterion': 'mse',`

```
{
    "pipeline_filepath": "saved_models/2018-05-14-16:32:09/QTV8PH9YEN.pickle",
    "pipeline_steps": ["HOG", "rf_classifier"],
    "params": {
        "num_orientations": 9,
        "num_cell_pixels": 8,
        "num_cells_block": 3,
        "criterion": "entropy",
        "max_features": 0.4659799248256272,
        "max_depth": 4,
        "min_samples_split": 4,
        "min_samples_leaf": 3,
        "n_estimators": 100,
        "n_jobs": -1
    },
    "partition_size": null,
    "score": 0.92
}
```

Figure 5-1: An example of a JSON log entry.

`'min_samples_split': 2, 'max_features': 0.39323, ...}` might be an example of such a dictionary, for the traditional image pipeline.

- `partition_size`: If provided, only evaluates the pipeline on a subsample of the data with size `partition_size`.

- The `save` method, which saves information about a scored hyperparameter configuration to the log. This method has the same arguments as the `score` function, except users also provide the score for that trial.

5.2 Implemented Search Algorithms

The `DeepMineSearch` class allows developers to create algorithms that search over two levels: the template space and the hyperparameter space. The developer can simply tune each template equally, or if she chooses, she can intelligently allocate resources between these templates.

To demonstrate the potential for the `DeepMineSearch` class, we implemented a few search algorithms: grid search, a generalized bandit search,and Hyperband [6]. Each of these algorithms works well for one template, and if desired, can be extended to algorithmically decide which templates to focus on. These algorithms are described

in the following subsections.

5.2.1 Grid Search

First, we implement a simple grid search, without using any of the facilities in `BTB`. This primary purpose of this is to demonstrate that a developer can create arbitrary hyperparameter search algorithms solely using the facilities provided in the `ScorePipeline`, `MLBlocks`, and `Compute` classes.

A user provides a `gridding` parameter, which determines how many values to check for every hyperparameter. Then, most of the logic involves reading the `MLPipeline` object, dividing each hyperparameter into intervals, and using `itertools` to create an iterator over every hyperparameter configuration. Then, the algorithm simply loops through these configuration, scores them, and keeps track of the best configuration.

5.2.2 Bandit Search

In the `SearchBTB` class, we implement a generalized bandit search that searches over both the pipeline space and the hyperparameter space. This class relies on the Bayesian Tuning and Bandits (BTB) library, also developed by the Data to AI group.

This search algorithm operates on two levels. First, over the pipeline space, we use a *selector* imported from BTB, which chooses which pipelines to consider by treating the task as a multi-armed bandit problem.

On the second level, we use a *tuner* imported from BTB, which forms a model over the hyperparameter space, and proposes candidates to explore next. In addition, BTB also allows the user to "grid" the hyperparameter space into discrete intervals, so `SearchBTB` does support grid search: however, we still implement `SearchGrid` to demonstrate that the algorithm can be easily implemented without BTB.

Simplified code for `SearchBTB` is available in Figure 5-2. Note that the `self.initialize_tuner` method, provided by the `DeepMineSearch` class, simply integrates the `MLPipeline` object with `BTB` to create a tuner compatible with that pipeline.

```
class SearchBTB(DeepMineSearch):
    def search(self, max_candidates=100,
                     tuner_class=tuning.Uniform,
                     selector_class=select.Uniform):
        # Create a selector over search_space,
        # which contains all candidate pipelines.
        selector = selector_class(self.search_space)

        # This stores the scores recorded for every pipeline,
        # in sequential order.
        scores = {pipeline : [] for pipeline in self.search_space
    }

        # This stores a tuner for every pipeline.
        tuners = {
            pipeline: self.initialize_tuner(pipeline, tuner_class
    )
                for pipeline in self.search_space
        }

        # The total number of candidates should not exceed
    max_candidates.
        total_candidates = 0

        # The number of candidates to run every time a pipeline
        # is proposed by he selector. This value is configurable.
        candidates_per = 10

        while total_candidates < max_candidates:
            pipeline = selector.select(scores)
            num_proposals = min(
                candidates_per,
                max_candidates - total_candidates
            )
            for _ in range(num_proposals):
                hp = tuners[pipeline].propose()
                score = self.score(pipeline, hp)
                self.save(pipeline, hp, score, save=True)
                tuners[pipeline].add(hp, score)
            total_candidates += num_proposals
```

Figure 5-2: A simplified implementation of the `SearchBTB` class.

5.2.3 Hyperband

The tradeoff between exploration and exploitation is crucial when performing hyperparameter optimization, especially when there are resource constraints: for example, we might have a limited amount of time to tune the model. While the general bandit-based algorithm in `SearchBTB` determines which pipelines to focus resources on, it does not specify how those resources should be allocated. The Hyperband algorithm, proposed in [6], treats hyperparameter optimization as a multi-armed bandit problem where a predefined resource like iterations, data samples, or features is allocated to randomly sampled configurations.

The first step of Hyperband is the SuccessiveHalving procedure, which takes a set of n hyperparameter configurations, a resource budget B, and allocates resources to find the most promising candidates. The core of the algorithm is as follows: uniformly allocate resources to n candidates, evaluate the candidates subject to those resources, discard the worst half, and repeat this procedure until one candidate remains. Note that instead of retaining the best $n/2$ of the candidates, we parametrize this by introducing η, such that we retain the best n/η candidates in every iteration.

However, the choices of B and n are unclear: the developer must decide whether to spend more resources on a fewer set of candidates, or fewer resources on a larger number of candidates. To address this, the complete Hyperband algorithm is as follows: users specify R, the maximum number of resources to allocate to any configuration, and η, the proportion of resources to discard in every iteration of SuccessiveHalving. Then, the algorithm iterates through several (n, B) pairs that are inversely proportional to each other, thereby addressing the exploration-exploitation tradeoff.

For complete pseudocode, refer to Algorithm 2. Here, we explain some of the functions referred to in the pseudocode, and how they might be implemented in DeepMining:

- The `get_hyperparameter_configs`(n) function returns n hyperparameter configurations. The authors of the paper left this open-ended: while developers can simply choose configurations randomly, they welcome developers to explore with

more intelligent methods. The easiest way to implement this in DeepMining is the BTB library, which lets developers switch between different tuners, which are trained using the scores from the search algorithm.

- The `run_then_return_val_loss`(T) function runs the n hyperparameter configurations on the provided dataset. This can be easily implemented using the `scorer`, which subclasses the `ScorePipeline` class.

- The `top_k`(T, L, k) function simply returns the best k hyperparameter configurations from T, keyed by the validation losses in L. While this can be implemented in $O(|T|)$ using QuickSelect, the constant factor is high, and it suffices to perform an $O(|T| \log |T|)$ algorithm using sorting.

Therefore, the Hyperband algorithm should be useful for training on large datasets. We implement the `SearchHyperband` class, which runs Hyperband using subsample size as the constrained resource: however, depending on the context in which it is used, this can be substituted for time, iterations, etc.

Algorithm 2 Hyperband

Input: R: resources per configuration,
η: proportion to discard during SuccessiveHalving
Initialization: $s_{max} = \lceil \log_\eta R \rceil$, $B = (s_{max} + 1)R$

for $s = \{s_{max}, s_{max} - 1, \ldots, 1, 0\}$ **do**
  ▷ Initialize (n, r) parameters for SuccessiveHalving
  $n = \lceil \frac{B}{R} \frac{\eta^s}{s+1} \rceil$, $r = R\eta^{-s}$
  ▷ Get n hyperparameter configurations by any method.
  $T =$ `get_hyperparameter_configs`(n)
  ▷ Begin SuccessiveHalving
  for $i = \{0, 1, \ldots, s-1, s\}$ **do**
    $n_i = \lfloor n\eta^{-i} \rfloor$
    $r_i = r\eta^i$
    ▷ Score each configuration, and discard all but the top $1/\eta$
    $T =$ `run_then_return_val_loss`(T)
    $T =$ `top_k`$(T, L, \lfloor n_i/n \rfloor)$
  end for
end for
return The configuration with the best validation loss.

5.3 Simplicity of Implementation

Therefore, because of the abstractions provided by `ScorePipeline`, `MLPipeline`, and `BTB`, implementing a scoring algorithm in DeepMining requires very little code. As seen in Table 5.2, each of the algorithms implemented requires less than 100 lines of code.

In addition, each of these algorithms can be easily parallelized, because hyperparameter configurations can be evaluated independently. For example, in `SearchBTB` and `SearchGrid` we can simply create batches of a fixed size, and in `SearchHyperband` we naturally batch hyperparameters when running the `run_then_return_val_loss` function. Because these computations are embarrassingly parallel, we abstract this out using the `Compute` class, which lets users run MapReduce style computations using any computation framework. For example, the library supports Spark, Dask, and Pathos Multiprocessing libraries. Modifying these implementations to support parallelization requires less than ten lines of code: this optimization will be discussed in Chapter 6.

Class	Number of Lines
`DeepMineSearch` (base class)	152
`SearchBTB`	59
`SearchGrid`	73
`SearchHyperband`	66

Table 5.2: The number of lines needed to implement each of the search algorithms.

5.4 Simplicity of Application

In addition, users can easily switch between different search algorithms, while only changing the initialization of the `DeepMineSearch` object. For example, Figure 5-3 demonstrates how a user might train the MNIST problem using different search algorithms. This implementation uses MNIST data already provided by `scikit-learn`.

```
# We import MNIST data from scikit-learn, and split it
# into train and validation sets.
mnist = fetch_mldata('MNIST original')

X_train, X_val, y_train, y_val = train_test_split(
    mnist.data,
    mnist.target,
    train_size=1000,
    test_size=300
)

# We use the F1 score, imported from scikit-learn.
def score_func(*args):
    return f1_score(*args, average='micro')

# We only tuner over a traditional image pipeline.
image_pipeline = TraditionalImagePipeline()

# This can be changed to ScoreDefault, for example.
scorer = ScoreBLB()

# This can easily be changed to SearchHyperband, SearchGrid, etc.
dm = SearchBTB(
    ProblemType.MULTICLASS,
    scorer,
    data,
    score_func=score_func,
    search_space=[image_pipeline],
    output_dir="saved_models/"
)

# This will search the hyperparameter space, log the history,
# and save it in output_dir.
dm.search()
print(dm.best_pipeline)
```

Figure 5-3: An example usage of the `DeepMineSearch` class.

Chapter 6

Compute

Tuning machine learning pipelines on different hyperparameter configurations and datasets involves several embarrassingly parallel computations. These computations occur at many levels of the tuning process: for example, if we are considering several candidate pipelines for a given dataset, these can be tuned in parallel. At a deeper level, we can evaluate a given pipeline on several hyperparameter configurations in parallel. Finally, when evaluating a pipeline on a particular hyperparameter configuration, we can evaluate dataset subsamples in parallel.

Each of these computations follows the straightforward MapReduce paradigm: for example, when running Bag of Little Bootstraps, we simply compute a score for each of the r subsamples, then take the average of these scores. We claim that in general, a MapReduce framework is sufficient to implement scoring and searching algorithms in DeepMining. With that in mind, a developer might use several computation frameworks to perform these computations: for example, one might prefer Spark for very large workloads, a simple threading library for small or medium workloads, or a trivial sequential implementation for debugging purposes.

We argue that DeepMining should be agnostic of the computation framework used; therefore, we abstract out this functionality into the `Compute` class, which lets any user implement or run computations on any computation framework that supports MapReduce.

In this chapter, we discuss the design of the `Compute` class, demonstrate potential

use cases, and justify that this abstraction suffices to implement most searching and scoring algorithms.

6.1 The `Compute` Class

To implement a computation framework in DeepMining, a developer need only subclass the `Compute` class, which requires implementing the following two methods. See Table A.4 for a summary of these two functions.

- The `__init__` method, which initializes the `Compute` instance. This method accepts the arguments that are specific to the computation framework: these might include the number of threads, location of the cluster, etc.

- The `run` method, which runs a MapReduce computation on a given map function, reduce function, and list of arguments. Specifically, this function accepts the following arguments:

  - `arguments`: A list of arguments in the form `['args': *args, 'kwargs': **kwargs]`. We decided on this representation to be standardized between different computation frameworks.

  - `map_function`: Any function that takes some `*args` and `**kwargs` as input, and returns a value.

  - `reduce_function`: Optionally, a function that takes in two values, and returns a result. For MapReduce computations, this function is generally associative.

With a simple interface, this abstraction should allow developers to introduce new computation frameworks into the DeepMining platform. At the same time, this abstraction suffices to implement many optimizations to search and scoring algorithms.

6.2 Implemented Computation Frameworks

To demonstrate the versatility of this abstraction, we implemented several computation frameworks using the `Compute` class. These include `SequentialCompute` (which uses Python), `PythonMultiProcessingCompute` (which uses the Pathos Multiprocessing library), `DaskCompute`, and `SparkCompute`. In this section, we discuss each of these implementations.

SequentialCompute: In the `SequentialCompute` class, we implement a simple sequential implementation of MapReduce in pure Python. This is useful for debugging purposes, while also demonstrating the feasibility and simplicity of the API.

PythonMultiProcessingCompute: In the `PythonMultiProcessingCompute` class, we implement a MapReduce framework using the Pathos Multiprocessing library. The framework performs computations using a pool of worker processes, where the user specifies the number of processes in the initialization function. This is particularly helpful for light to medium workloads, offering significant performance benefits without large overhead.

SparkCompute: In the `SarkCompute` class, we implement MapReduce using the Spark cluster-computing framework [10]. In the initialization function, users have the option to specify the number of partitions, as well as the location of the cluster: Spark can either run locally, or on an Amazon Elastic Compute Cloud instance. Spark is useful for high workloads, but incurs more overhead.

DaskCompute: In the `DaskCompute` class, we implement Dask, a lightweight task scheduler and parallel computing library for analytics [9]. Depending on the workload, Dask might perform better than Spark because it has less overhead.

6.3 Simplicity of Implementation

We demonstrate that implementing a new computation framework is simple, as the abstraction aligns well with existing computation frameworks. As an example, consider the implementation of `SequentialCompute` (see Figure 6-1), which demonstrates the

```
class SequentialCompute(Compute):
    """
    Class for simple sequential computations.
    """
    def run(self, arguments, map_function, reduce_function=None):
        output = [map_function(*args["args"], **args["kwargs"])
                  for args in arguments]
        if reduce_function is not None:
            output = reduce(reduce_function, output)
        return output
```

Figure 6-1: The implementation of `SequentialCompute`.

minimum work required to implement `run`.

Apart from initializing the framework and parsing the arguments, implementing a more complex computation framework requires little added effort. This is because most of these frameworks already support MapReduce computations in some form. Therefore, we achieve normalization without much effort, which enables us to easily switch between different frameworks. This simplicity is reflected in Table 6.1, which demonstrates that even implementing Spark or Dask requires substantially fewer than 100 lines of code.

Class	Number of Lines
`SequentialCompute`	13
`PythonMultiProcessingCompute`	25
`SparkCompute`	73
`DaskCompute`	20

Table 6.1: The number of lines needed to implement each of the computation frameworks.

6.4 Simplicity of Application

In this section, we demonstrate how users can use the `Compute` module to easily switch between different computation frameworks. To do so, consider the example in Figure 6-2, in which the `map` function performs simple arithmetic operations, and the `reduce` function sums the results. Note that the same computation can be seamlessly

```
def run_test(self, compute):
    args = [{"args": (i, i), "kwargs": {"test": 1}} for i in range(5)]
    reduce_func = lambda x, y : x + y
    def map_func(x, y, test=1):
        return x + y
    ans = compute.run(args, map_func, reduce_function=reduce_func)
    assert ans == 20

c = SequentialCompute()
run_test(c)

c = PythonMultiProcessingCompute(
    num_python_processes=10
)
run_test(c)

c = DaskCompute()
run_test(c)

c = SparkCompute(
    my_spark_directory="/Users/akshay/Documents/spark",
    spark_cluster_location=SparkCompute.LOCAL
)
run_test(c)
```

Figure 6-2: The implementation of a simple MapReduce computation using various computation frameworks.

integrated with each of the aforementioned computation frameworks. Apart from defining the `map` and `reduce` functions, defining a computation takes a few lines of code, and switching between different frameworks is as simple as replacing the initialization of the `Compute` object.

6.5 Integration with DeepMining

Therefore, with the `Compute` abstraction, adding parallelization to the different aspects of DeepMining requires little effort. In fact, the following modules were first written without performance in mind, but were modified to use `Compute` objects with few changes to the core logic. In this section, we discuss how parallelization can easily be added to improve the `ScorePipeline` and `DeepMineSearch` classes.

Integration with `DeepMineSearch`: When searching through the hyperparameter space, one can evaluate candidate configurations in an embarrassingly parallel fashion.

The `SearchBTB` class, for example, was modified to include a `max_parallel` and

Without `Compute`

```
num_candidates = 0
while num_candidates < max_candidates:
    hp = tuner.propose()
    score = self.score(pipeline, hp)
    self.save(pipeline, hp, score, save=True)
    num_candidates += 1
```

With `Compute`

```
def score_pipeline_func(hp):
    return (hp, self.score(copy.copy(pipeline), hp))

def evaluate(hyperparam_arr):
    args = [{"args": [hp], "kwargs": {}} for hp in hyperparam_arr ↩
    ]
    results = compute_framework.run(args, score_pipeline_func)
    for hp, score in results:
        self.save(pipeline, hp, score, save=True)

num_candidates = 0
while num_candidates < max_candidates:
    hyperparam_arr = [tuner.propose() for _ in range(num_parallel ↩
    )]
    evaluate(hyperparam_arr)
    num_candidates += len(hyperparam_arr)
```

Figure 6-3: The integration of `Compute` into the `SearchBTB` class requires less than ten lines of code. If the tuner has a finite number of candidates, extra logic needs to be added in case `tuner.propose()` returns `None`.

`compute_framework` argument. Now, every time the number of proposed candidates reaches `max_parallel`, the batch can be evaluated in parallel. See Figure 6-3 for the implementation.

Integrating `Compute` with `SearchHyperband` was simpler, because the Hyperband algorithm already naturally evaluates a pipeline on a batch of hyperparameter configurations. See Figure 6-4 for the changes required.

Integration with `ScorePipeline`: In addition to search, `Compute` class was used to improve to `ScoreBLB` class, which performs Bag of Little Bootstraps (BLB). The algorithm creates s bags, each of which can be evaluated independently; therefore, the `Compute` object simply takes in the list of bags as its arguments, and `score_bag` as its map function. This integration also took less than five lines of code.

Without `Compute`

```
results = [self.score(pipeline, hp) for hp in T]

for hp, score in results:
    self.save(pipeline, hp, score, partition_size=int(r_i), save=True)
```

With `Compute`

```
def score_pipeline_func(hp):
    return (hp, self.score(copy.copy(pipeline), hp, partition_size=int(r_i)))

args = [{"args": [hp], "kwargs": {}} for hp in T]
results = compute_framework.run(args, score_pipeline_func)

for hp, score in results:
    self.save(pipeline, hp, score, partition_size=int(r_i), save=True)
```

Figure 6-4: The integration of `Compute` into the `SearchHyperband` class requires less than five lines of code.

6.6 Drawbacks of the `Compute` Abstraction

As demonstrated in the previous sections, integrating the `Compute` object to different aspects of DeepMining requires very few changes. However, unifying multiple frameworks under the same API, while increasing modularity, runs the risk of reducing performance and customizability. However, because the computations encountered in DeepMining are simple and embarrassingly parallel, it seems like most of these adjustments may be done by modifying the `Compute` class while maintaining isolation. For example, platform-specific parameters can likely be configured in the initialization function. In addition, any performance enhancements can likely be implemented in the `run` function.

Chapter 7

Performance

In this chapter, we run DeepMining on several datasets with different configurations. As we currently do not have enough pipelines and search algorithms to reliably assess accuracy, we instead focus on performance. As DeepMining, MLBLocks, and BTB evolve, there should be enough modules to conduct experiments involving accuracy.

In particular, we run DeepMining on the following datasets and pipelines:

- The traditional image pipeline (which runs histogram of oriented gradients (HOG), then a random forest classifier), tuned on the MNIST dataset.

- The random forest classifier, tuned on the Wine dataset, a multiclass classification problem provided by `scikit-learn`.

- The random forest regressor, tuned on the Boston house-prices dataset, a regression problem provided by `scikit-learn`.

We run these experiments on an Ubuntu 14.04 machine with 16 cores and 16 gigabytes of RAM. In summary, we notice that on the provided workloads, Spark performs poorly, while the Python multiprocessing library offers significant speedups. We plan to run more conclusive tests as more modules are implemented in DeepMining.

7.1 Effect on Search

Tables 7.1, 7.2, and 7.3 demonstrate the results when running the BTB search algorithm on the MNIST, Wine, and Boston datasets. We vary the `Compute` instance used, along with the number of evaluations to run in parallel, represented as the `num_parallel` argument.

In all three cases, we find that using the Python multiprocessing library offers almost a 3x speedup when `num_parallel` is 100. In addition, we find that Spark performs slightly worse than a pure sequential implementation: this seems to indicate that Spark has too much overhead, but perhaps running on a larger dataset would demonstrate a better speedup.

Tables 7.4, 7.5, and 7.6 demonstrate the results when running the MNIST, Wine, and Boston datasets using the Hyperband search algorithm. Here, we notice similar results, although Spark performs slightly worse, and using Python multiprocessing instead brings a 2x speedup. Note that Hyperband does not need a `num_parallel` attribute, because it already evaluates configurations in batches.

7.2 Effect on Scoring

Table 7.7 effects of Bag of Little Bootstraps (BLB) by tuning the traditional image pipeline on the MNIST dataset. Here, we notice that using Python multiprocessing offers a 2x speedup, while Spark performs approximately 10% worse than sequential.

Compute	Num. Parallel	Time (s)
`SequentialCompute`	10	146.63
	20	146.944
	50	147.719
	100	147.957
`PythonMultiProcessingCompute`	10	73.592
	20	56.7
	50	57.197
	100	50.737
`SparkCompute`	10	157.978
	20	146.245
	50	143.173
	100	142.485

Table 7.1: The results for the MNIST dataset and BTB search algorithm.

Compute	Num. Parallel	Time (s)
`SequentialCompute`	10	58.822
	20	59.249
	50	59.982
	100	60.285
`PythonMultiProcessingCompute`	10	30.553
	20	30.578
	50	31.19
	100	21.787
`SparkCompute`	10	81.481
	20	69.77
	50	67.379
	100	67.037

Table 7.2: The results for the wine dataset and BTB search algorithm.

Compute	Num. Parallel	Time (s)
`SequentialCompute`	10	74.884
	20	74.884
	50	74.763
	100	75.976
`PythonMultiProcessingCompute`	10	39.157
	20	38.708
	50	39.996
	100	28.495
`SparkCompute`	10	76.101
	20	63.031
	50	59.821
	100	58.858

Table 7.3: The results for the Boston dataset and BTB search algorithm.

Compute	Time (s)
`SequentialCompute`	229.599
`PythonMultiProcessingCompute`	110.145
`SparkCompute`	295.111

Table 7.4: The results for the MNIST dataset and Hyperband search algorithm.

Compute	Time (s)
`SequentialCompute`	161.149
`PythonMultiProcessingCompute`	81.943
`SparkCompute`	206.817

Table 7.5: The results for the wine dataset and Hyperband search algorithm.

Compute	Time (s)
`SequentialCompute`	270.437
`PythonMultiProcessingCompute`	137.177
`SparkCompute`	332.055

Table 7.6: The results for the Boston dataset and Hyperband search algorithm.

BLB?	Compute	Time (s)
Yes	`SequentialCompute`	593.509
Yes	`PythonMultiProcessingCompute`	301.660
Yes	`SparkCompute`	652.783
No	-	136.416

Table 7.7: The results for the MNIST dataset and BTB search algorithm, with and without Bag of Little Bootstraps.

Chapter 8

Conclusion and Future Work

Over the course of this year, we achieved the design goals we had in mind: to take the DeepMining system and divide it into modular, isolated parts that can easily be extended or modified. For the most part, however, we focused on functions that were already present in the original iteration of DeepMining. As DeepMining evolves, we hope to further this goal: to both add new abstractions that aid the automated machine learning process, and to implement new subclasses of each of these abstractions. In doing so, we increase the scope and robustness of DeepMining.

In this chapter, we discuss future work that could improve the next iteration of DeepMining, and speculate on how these features might be implemented.

8.1 Supporting More Forms of AutoML

The machine learning process can be automated on several levels: as discussed in Chapter 1, these include feature engineering, architecture search, and hyperparameter search. While the current iteration focuses on the latter, the DeepMining abstraction could easily allow for other forms of AutoML. We discuss these in the following subsections.

8.1.1 Feature Engineering

To reiterate, feature engineering is the process of taking a dataset and generating salient features from it. The more relevant the features are, the more accurate the resulting model may be.

Feature engineering could be easily implemented by creating a `DataObject` class in DeepMining. In addition to storing `X_train`, `y_train`, `X_val`, and `y_val`, the `DataObject` class could integrate with a module that reads in this data, along with metadata about the problem, and generates more salient features. For example, Deep Feature Synthesis [4], created by the Data to AI group, could be integrated with DeepMining.

8.1.2 Architecture Search

The hardest problem that needs to be solved is finding the pipeline search space. There are two levels in which this can be done: first, the system can search through a list of predefined pipelines, and decide which ones are compatible with the dataset and problem. On a deeper level, because the MLBlocks library provides composable primitives, the system can use these primitives to automatically construct pipelines that might be feasible. This problem is much more difficult, because the search space is exponential. In addition, it requires domain-level expertise to annotate each primitive, and to decide when two primitives can and should be composed.

Architecture search could be easily integrated into the `DeepMineSearch` class: specifically, instead of accepting the `search_space` variable, which represented the list of `MLPipeline` objects to grade over, the `__init__` function could instead infer the search space using the dataset, problem type, and scoring function.

8.2 Implementing More Modules

The abstractions in DeepMining were intended for any developer to contribute new modules. Another goal is to invite developers to use the platform, and contribute

new `Compute`, `DeepMineSearch`, and `ScorePipeline` modules. While increasing the breadth of the DeepMining platform, this also helps test the extensibility and flexibility of the current abstraction. In addition, it helps receive feedback about the system design as a a whole. The same holds for MLBlocks and BTB, as contributing to those modules has a direct benefit on DeepMining.

Appendix A

API Tables

`__init__(self, *args, **kwargs)`	
Purpose	
Initializes a particular scoring method.	
Arguments	
The arguments are specific to the parameters of the computation framework. These might include the number of subsamples, size of each subsample, etc.	
Outputs	
This function initializes the instance, but does not return anything.	
`score(self, pipeline, X_train, y_train, X_val, y_val, score_func=None)`	
Purpose	
Evaluate the pipeline on the given train data, validation data, and scoring function.	
Arguments	
`pipeline`	The `MLPipeline` object to be evaluated, with hyerparameters already set.
`X_train, y_train, X_val, y_val`	The training data, training labels, validation data, and validation labels, represented as `numpy` array-like objects.
`score_func`	The scoring function to use, which takes in `X` and `y` and returns a score. If `score_func` is `None`, uses a default scoring function.
Outputs	
The `run` function runs the `map_function` on the list of arguments. If `reduce_function` is `None`, then return the outputs from the map procedure, Otherwise, run `reduce_function` on the outputs and return the result.	

Table A.1: The API for the the `ScorePipeline` class.

`score(self, pipeline, hyperparams, partition_size=None)`	
Purpose	
Runs a pipeline on a given hyperparameter set, and a particular partition size of the data.	
Arguments	
`pipeline`	The `MLPipeline` object to consider.
`hyperparams`	The hyperparameters to pass into the pipeline, represented a dictionary from name to value.
`output_dir`	Where to save logs during the search procedure. Defaults to `saved_models/`.
`partition_size`	If provided, only evaluates the pipeline on a subsample of the data. This is passed into the `ScorePipeline` object.
Outputs	
Returns the score as a `float`, as evaluated by the `scorer`.	
`save(self, pipeline, hyperparams, score, partition_size=None, save=False)`	
Purpose	
To save information about a particular trial. We give developers control over when they do this, in case they don't want to store information about every single candidate.	
Arguments	
Same as the `score` function, except users also provide the `score` for that trial.	
Outputs	
This function pickles the `MLPipeline` object, saves it in `output_dir`, and adds information to the history of trials.	

Table A.2: The API for the developer-facing methods in `DeepMineSearch`.

`__init__(self, *args, **kwargs)`	
Purpose	
Initialize a particular search algorithm.	
Arguments	
`scorer`	The scoring method, represented a subclass of `ScorePipeline`. For example, this might refer to Bag of Little Bootstraps (`ScoreBLB`) or cross-validation (`ScoreCV`).
`X_train, y_train, X_val, y_val`	The training data, training labels, validation data, and validation labels, represented as `numpy` array-like objects.
`score_func`	The scoring function to use, which takes in `X` and `y` and returns a score. If `score_func` is `None`, uses a default scoring function.
`search_space`	The list of pipelines to tune, represented as a list of `MLPipeline` objects.
`output_dir`	Where to save logs during the search procedure. Defaults to `saved_models/`.
Outputs	
This function initializes the search instance, but does not return anything.	
`search(self, *args, **kwargs)`	
Purpose	
Initialize a particular search algorithm.	
Arguments	
This depends on the scoring algorithm. For example, a user might specify the number of candidates to consider, the number of candidates to run in parallel, which tuner or selector to use (if using `BTB`), etc.	
Outputs	
The `search` function does not return anything, but it stores the best pipeline in the `best_pipeline` attribute. In addition, information about previous candidates is stored in `output_dir`.	

Table A.3: The API for the user-facing methods in `DeepMineSearch`.

`__init__(self, *args, **kwargs)`	
Purpose	
Initialize a particular computation framework.	
Arguments	
The arguments are specific to the parameters of the computation framework. These might include the number of threads, location of the cluster, etc.	
Outputs	
This function initializes the instance, but does not return anything.	
`run(self, arguments, map_function, reduce_function=None)`	
Purpose	
Initialize a particular search algorithm.	
Arguments	
`arguments`	A list of arguments in the form `[{'args': *args, 'kwargs': **kwargs}]`.
`map_function`	Any function that takes takes `*args` and `*kwargs` as input, and returns a value.
`reduce_function`	Optionally, a function that takes in two values, and returns a result. Generally, this function should be associative.
Outputs	
The `run` function runs the `map_function` on the list of arguments. If `reduce_function` is `None`, then return the outputs from the map procedure, Otherwise, run `reduce_function` on the outputs and return the result.	

Table A.4: The API for the `Compute` class.

Bibliography

[1] Data-driven discovery of models (d3m). `https://www.darpa.mil/program/data-driven-discovery-of-models`.

[2] Feature labs. `https://www.featurelabs.com/`.

[3] A. Anderson, S. Dubois, A. Cuesta-Infante, and K. Veeramachaneni. Sample, Estimate, Tune: Scaling Bayesian Auto-tuning of Data Science Pipelines.

[4] J.M. Kanter and K. Veeramachaneni. Deep Feature Synthesis: Towards Automating Data Science Endeavors.

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

[6] 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.

[7] E.R. Sparks, A. Talwalkar, D. Haas, et al. Automating Model Search for Large Scale Machine Learning.

[8] J.T. Springenberg, A. Klein, et al. Bayesian Optimization with Robust Bayesian Neural Networks.

[9] Dask Development Team. Dask: Library for dynamic task scheduling. 2016.

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

[11] B. Zoph and Q. V. Le. Neural Architecture Search with Reinforcement Learning. *ArXiv e-prints*, November 2016.