A Flexible Framework for Composing End to End Machine Learning Pipelines

by

William Xue

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of
Master of Engineering in Computer Science and Engineering
at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
June 2018

© Massachusetts Institute of Technology 2018. All rights reserved.

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

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

Accepted by ...........................................................
Katrina LaCurtis
Chair, Masters of Engineering Thesis Committee
A Flexible Framework for Composing End to End Machine Learning Pipelines

by

William Xue

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

Abstract

In this thesis, we aim to simplify the building of end-to-end machine learning pipelines while preserving the performance of such pipelines on real data. As a solution to this, we propose the MLBlocks framework, a system that allows an end user to obtain a pipeline with only data and a list of data science blocks. Once a pipeline is specified, a user can tune its hyperparameters, as well as fit and predictions, with minimal code.

When building MLBlocks, we first develop a data science block library that seamlessly integrates third party blocks without integration code, providing a foundation for users to start building data science pipelines. We then provide the MLPipeline framework that allows users to simply tie together these blocks and perform the aforementioned tuning, fitting, and predicting operations with the resulting pipelines.

Finally, we test the framework’s usability as well as its ability to preserve performance on real data by running several pipelines on various data modalities and by integrating MLBlocks into a larger scale project. Since we are able to replicate the pipelines already in use, we are able to obtain identical results while dramatically simplifying application logic. We conclude that MLBlocks succeeds in providing a simple but effective solution to making the construction of high-performing end-to-end pipelines both simpler and more accessible.

Thesis Supervisor: Kalyan Veeramachaneni
Title: Principal Research Scientist
Acknowledgments

First and foremost, I would like to sincerely thank my supervisor, Kalyan Veeramachaneni, for his continuous guidance and support throughout the project. Without Kalyan’s expertise and ideas, this project and thesis would not have been possible.

I would also like to thank my fellow DAI labmates Akshay Ravikumar and Laura Gustafson for helping with various aspects of this project, including design and implementation. I am also grateful to Carles Sala for helping to make MLBlocks production ready and improving code quality in general.

Additionally, I would like to acknowledge Xylem and the National Science Foundation for providing support in terms of generous funding for this project.

Finally I would like to acknowledge my friends and family for their continued support over the years.
# Contents

1 Introduction 15

1.1 Definitions .................................................. 17
1.2 Motivations for a pipeline creation framework ............ 18
  1.2.1 Composition of machine learning pipelines .......... 19
  1.2.2 Hyperparameter tuning .................................. 20
1.3 MLBlocks ..................................................... 22
  1.3.1 Goals ..................................................... 22
1.4 Thesis roadmap ............................................... 23

2 System design overview 25

2.1 Design goals ................................................ 25
  2.1.1 Non-goals ............................................... 27
2.2 High level design ........................................... 27

3 MLBlocks 31

3.1 A library of blocks ........................................... 31
  3.1.1 JSON specification ..................................... 33
  3.1.2 JSON parsers ........................................... 34
  3.1.3 Building the blocks library ............................ 35
3.2 The MLBlock abstraction ..................................... 38
  3.2.1 Virtual classes ......................................... 39
3.3 Contributing blocks ......................................... 40
3.4 Maintaining the library ..................................... 41
7.5 Results .............................................................................................................. 69
7.5.1 Ease of use .................................................................................................. 69
7.5.2 Feasibility of pipelines ............................................................................... 70

8 Future work ........................................................................................................... 71
8.1 MIT-TA2 integration ....................................................................................... 71
8.1.1 DARPA Data Driven Discovery (D3M) ......................................................... 72
8.1.2 MIT-TA2 system overview ......................................................................... 72
8.2 Automatically generated pipelines ................................................................... 73

9 Conclusion .............................................................................................................. 75
9.1 Key findings ..................................................................................................... 75
9.2 Contributions .................................................................................................. 76

A Figures .................................................................................................................. 79

B Tables .................................................................................................................... 89
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-1</td>
<td><code>MLBlocks</code> needs only a few simple inputs to create a pipeline.</td>
<td>21</td>
</tr>
<tr>
<td>2-1</td>
<td>The <code>MLBlocks</code> framework.</td>
<td>28</td>
</tr>
<tr>
<td>2-2</td>
<td><code>MLBlocks</code> pipeline creation flow</td>
<td>29</td>
</tr>
<tr>
<td>3-1</td>
<td><code>MLBlocks</code> blocks library.</td>
<td>31</td>
</tr>
<tr>
<td>3-2</td>
<td>A sample JSON file with specifications for a block(^1).</td>
<td>32</td>
</tr>
<tr>
<td>3-3</td>
<td>A sample <code>MLBlock</code> instance; analogous to the figure 3-2 JSON.</td>
<td>38</td>
</tr>
<tr>
<td>3-4</td>
<td>Integrating with and contributing to the <code>MLBlocks</code> block library.</td>
<td>41</td>
</tr>
<tr>
<td>4-1</td>
<td>The <code>MLPipeline</code>.</td>
<td>45</td>
</tr>
<tr>
<td>4-2</td>
<td>Initializing an <code>MLPipeline</code> with just one line of code.</td>
<td>46</td>
</tr>
<tr>
<td>4-3</td>
<td>Fitting and predicting with an <code>MLPipeline</code> on image data.</td>
<td>46</td>
</tr>
<tr>
<td>4-4</td>
<td>Obtaining hyperparameter values of an <code>MLPipeline</code>.</td>
<td>48</td>
</tr>
<tr>
<td>4-5</td>
<td>Updating hyperparameter values for an <code>MLPipeline</code>.</td>
<td>48</td>
</tr>
<tr>
<td>5-1</td>
<td>Format for the single-table datatype.</td>
<td>52</td>
</tr>
<tr>
<td>5-2</td>
<td>Format for the image datatype.</td>
<td>53</td>
</tr>
<tr>
<td>5-3</td>
<td>Format for the text datatype.</td>
<td>53</td>
</tr>
<tr>
<td>5-4</td>
<td>Format for the audio datatype.</td>
<td>54</td>
</tr>
<tr>
<td>5-5</td>
<td>Format for the multi-table datatype.</td>
<td>55</td>
</tr>
<tr>
<td>5-6</td>
<td>Format for the graph datatype.</td>
<td>56</td>
</tr>
<tr>
<td>5-7</td>
<td>Format for the multi-graph datatype.</td>
<td>57</td>
</tr>
<tr>
<td>7-1</td>
<td>The <code>DeepMining</code> workflow.</td>
<td>66</td>
</tr>
</tbody>
</table>
List of Tables

6.1  MLBlocks experiment pipelines ................................. 61
6.2  MLBlocks experiment pipelines ................................. 63
7.1  Lines of code to implement pipelines before and after MLBlocks integration ............................................. 69
B.1  Blocks in the MLBlocks library ................................. 89
Chapter 1

Introduction

As our everyday lives become more and more data driven, software tools and libraries are emerging to help solve real-world data problems. In just the past couple of months, companies like Intel and Google have open sourced production-ready tools for tasks such as natural language processing\(^1\) and deep learning in the browser\(^2\). As a result, today there exists a large ecosystem of software libraries that provide experts with a foundation to create end-to-end solutions to machine learning problems.

Concretely, an expert solves a problem by piecing together a data-driven solution using fundamental building blocks from libraries such as \texttt{scikit-learn}\(^1\) and \texttt{keras}\(^3\) to create pipelines. As a simple example, consider the problem of predicting whether a customer will buy a product. In this case, one might choose to create a pipeline with a featurization block and a random forest classifier\(^1\) to later use to fit and produce predictions on his data.

This is often an extremely complicated process, requiring carefully reading through the documentation, writing many lines of new code, and manually specifying the hyperparameters related to each transformation block, exposing them so that they can be effectively tuned. While recent growth in the quantity of building blocks allows experts to solve problems they could not have before, it also exacerbates this complexity and introduces several issues.

\(^{1}\)http://nlp_architect.nervanasys.com/
\(^{2}\)https://js.tensorflow.org/
\(^{3}\)https://github.com/keras-team/keras
First, a single library cannot solve all of the various machine learning problems. For example, scikit-learn cannot solve a majority of problems related to image datasets. Additionally, some libraries that are introduced today are extremely specialized, suitable only for problems like deep learning or other extremely specific functionalities. Inevitably, to solve a problem well, a data scientist\textsuperscript{4} must piece together elements drawn from multiple libraries.

Second, these libraries do not have a uniform API, so learning to use each of them involves a significant learning curve. It is uncommon for a data scientist—much less an average user—to possess the required knowledge necessary to connect building blocks from various domain-specific libraries together. Additionally, even if a data scientist manages to connect his desired blocks into an end-to-end pipeline, he must then make appropriate hyperparameter choices. While there exist tools to facilitate parts of this pipeline architecture construction, there is no end-to-end solution. As a result, many data science and machine learning tools are inaccessible to a large potential userbase. As more and more libraries are added, these problems will only be magnified.

To address these issues, we introduce MLBlocks, a lightweight library that unifies the vast quantity of disparate data science and machine learning libraries. MLBlocks provides a uniform API and focuses on making different machine learning tasks easy. Particularly, it makes it simple and intuitive for users to access building blocks from various machine learning libraries because of their integration with the MLBlocks system. Additionally, MLBlocks exposes hyperparameters for the entire pipeline. The end result is a one-stop-shop for solving problems for different types of data modalities, from image and text data to time series and relational data.

In this chapter, we lay the foundation for the MLBlocks system by defining key terms and laying out several compelling reasons to design an abstraction for end-to-end tunable machine learning pipelines.

\textsuperscript{4}We use the terms data scientist and developer interchangeably.
1.1 Definitions

Before we proceed, however, it is important to precisely define certain potentially ambiguous terms we will reference over the course of this thesis.

**Machine learning pipelines:** We first define a machine learning pipeline, or simply a pipeline for the sake of brevity, as an end-to-end solution to a data problem. When given input data, a pipeline should be able to output meaningful predictions. We note that a pipeline is defined as end-to-end, so the input data may not be preprocessed or featurized. In the case of solving a simple image classification problem, a machine learning pipeline may simply consist of a random forest classifier which performs classification, or it may include a histogram of ordered gradients (HOG) featurization step [3] in addition to the random forest classification. This precise definition of a pipeline is necessary because often we would like to tune the preprocessing and featurization steps of a pipeline along with the rest of the data-processing steps.

Input data does not necessarily have to be preprocessed, even in the case of training inputs for pipelines that require fitting over a training data set. However, we require the training target labels to have the same form as the output of the pipeline. That is, if a pipeline would output class predictions, then the training targets must also be class predictions. They cannot be passed in in any other form, such as class probabilities.

**Hyperparameters:** We now broadly define a hyperparameter as a parameter set by the data scientist when determining the pipeline architecture. These are distinct from parameters internal to the pipeline or learned during the pipeline training process. For example, in a convolutional neural network, the weights between nodes of a dense layer are model parameters, while things such as convolution kernel size are hyperparameters. Hyperparameters can be either fixed or tunable; this is typically determined by the data scientist, but in certain cases, the distinction is clear. For example, the input size to the HOG featurization step should be fixed, as the input size of the data passed into the step is always the same. However, the number of cells in each block of the image that HOG uses to create features can alter the quality of
the resulting predictions, so we can make that hyperparameter tunable. Orthogonal to this, hyperparameters can be either root or conditional. A conditional hyperparameter is tunable depending on whether another hyperparameter takes on a certain value, and a root hyperparameter is one that is not.

**Blocks:** Finally, we define a block as a fundamental building block of a pipeline. In the case of the aforementioned simple image classification pipeline, we define both the HOG featurization step and the random forest classification step to be blocks. In the context of this thesis, we treat the terms step and primitive as analogous to the term block. The only concrete requirement for a block is that it transforms data in some manner. For example, the HOG block transforms an image vector into a feature vector, and likewise, the random forest classifier transforms input data into class predictions. Notice that the requirement of having tunable hyperparameters is absent from this definition of a block. This is because some blocks do not need to be tuned, but are necessary for the pipeline to function. To visualize this, we can imagine a neural image classification pipeline where the final dense layer of the neural model outputs class probabilities for each class. A naive final step to convert class probabilities to class predictions may simply be an **argmax** function, which does not have any hyperparameters that need to be tuned, but is necessary for the pipeline to produce its expected outputs. Therefore, we must treat this step as a block.

### 1.2 Motivations for a pipeline creation framework

Our motivation for creating a new framework for composing and tuning pipelines stem primarily from accessibility. A solution that produces a fully specified and tuned end-to-end machine learning pipeline simply does not exist. In order to motivate the need for such a solution, we break down the pipeline construction process into two phases: pipeline composition and hyperparameter tuning.
1.2.1 Composition of machine learning pipelines

We imagine a scenario in which a user, such as a business, wishes to solve a data science problem using a dataset they possess. This entails composing a pipeline, beginning with selecting which blocks to include in the pipeline and writing the code to connect these blocks so that the data flows smoothly from input to output. Often, end users lack the expertise to perform these required steps; to create a pipeline that performs well on a given data set and a certain amount of domain knowledge is required. As a result, a user may take a variety of approaches in composing a pipeline, such as hiring professional data scientists or hosting data science competitions on websites such as Kaggle\textsuperscript{5}. This creates a barrier of entry to using machine learning pipelines to solve real-world problems. A user must possess either a baseline amount of generalized experience and/or specialized expertise in data science or the resources to employ experts with said experience.

Furthermore, due to a combination of both the decentralized nature and the sheer volume of data science and machine learning libraries, it is entirely possible that even a pipeline created by a data scientist is suboptimal. For example, a data scientist may be discouraged from using a certain library module as a pipeline block due to difficulty in integrating it with the rest of the pipeline. As many libraries are not designed to be integrated with others, this occurs with reasonable frequency. In an even more extreme example, a data scientist may simply be unaware of the library that best solves the problem.

Simplifying pipeline composition: We note that an abstraction that simplifies the construction of pipelines remedies these problems, both reducing the amount of knowledge required for an average end-user and making certain data science libraries more accessible to data scientists by creating already-in-place scaffolding and centralization of blocks. As we can see, the end result is better performing pipelines at a lower cost.

\textsuperscript{5}https://www.kaggle.com/
1.2.2 Hyperparameter tuning

After specifying the block layout of a pipeline, a data scientist must then select the pipeline’s hyperparameters. In this case, because the choice of parameters is so specific to the dataset in question, even experienced data scientists may not be able to choose appropriate hyperparameters to achieve the best performing pipeline. As a result, many systems for tuning hyperparameters—which use certain established algorithms to try different sets of hyperparameter combinations—are employed in practice. We describe two general hyperparameter tuning paradigms and highlight their individual strengths and weaknesses.

Library-specific tuners: Library-specific tuners are, as their name suggests, tuners that are designed to be used with specific libraries. One concrete example of such a tuner is the scikit-learn GridSearchCV\(^6\), which performs a brute-force search over all possible hyperparameter sets in a pipeline. This abstracts much of the code necessary on the part of the end-user to integrate the pipeline with the tuner, as the tuner can directly operate on the internals of the steps of the pipeline. However, this approach also assumes certain library-implementation details about each step, so it is impossible to integrate third-party libraries with these tuners; a sequential keras neural model cannot be directly used with scikit-learn’s GridSearchCV tuner. As a result, this places undesirable restrictions on a developer in developing a fully tuned and specified pipeline.

Black-box tuners: Black-box tuners lie on the other end of the spectrum from library-specific tuners in terms of advantages and disadvantages. As one would expect, these tuners abstract out the logic of tuning a pipeline from the pipeline-supplying end-user. Examples of black-box tuners include Spearmint [14], SigOpt\(^7\), and HPOlib [4] among others. Generally, these tuners pull input sets of hyperparameters into a scoring function which represents the pipeline, and output a set of hyperparameters that performs well on that function. This gives users the key advantage of flexibility in how they specify their pipeline. A user is able to connect arbitrary components in

---

\(^6\)http://scikit-learn.org/stable/modules/grid_search.html

\(^7\)https://sigopt.com/
his pipeline and evaluate it on multiple hyperparameter sets as long as he adheres to the scoring-function-as-an-input model of these tuners.

However, the lack of an API for specifying pipelines and exposing hyperparameters means that the onus is on the user to implement these things. This is an issue because different libraries expose hyperparameters in different ways. Moreover, in many libraries, hyperparameters are not exposed at all. For example, many models implemented with the *keras* library hard code hyperparameters in the model-building function. This results in a problem similar to the one we described before in section 1.2.1 of increased required developer load and expertise. As mentioned before, this creates a barrier of entry that we would like to reduce, if at all possible.

**Exposing hyperparameters:** We claim that an end-to-end pipeline abstraction allows us to achieve the best of both systems, as it specifies and exposes hyperparameters in a very clean and simple manner. That is, because our pipelines’ composition is specifically designed to be simple for the user, cleanly exposing hyperparameters in our pipeline abstraction allows us to easily integrate our pipeline with black-box tuners. In doing this, we gain all of the flexibility that comes with a black-box tuner while simultaneously minimizing developer load.

Tangentially, we observe that if we make our abstraction general enough, we can integrate with non-data science specific black-box tuners, such as Bayesian Tuning and Bandits (BTB)\textsuperscript{8}. This gives the developer much more freedom and accessibility when choosing how to tune his pipeline.
1.3 MLBlocks

The issues we outline in the above two phases of pipeline composition and hyperparameter tuning stem from a decentralized, component-based view of data science tools. While it is useful to abstract out the details behind these tools into separate libraries, different API specifications and standards coupled with the vast quantity of these tools make it difficult for an end-user to take full advantage of all the tools at his disposal.

The MLBlocks\(^9\) framework represents our solution to this overarching problem. Creating this solution required a two-pronged approach. First, we created various abstractions that enable an end user to create an end-to-end pipeline with hyperparameters properly exposed for tuning. Then, we implemented a system that facilitates the creation of a standardized library to unifies third party blocks via developer contribution. The end result is a framework in which an end-user can simply visit our library, pick out the blocks he would like to try, compose and tune his pipeline, and ultimately draw meaningful conclusions from his data.

1.3.1 Goals

At a high level, we aim to produce a system that allows users to easily create tunable end-to-end pipelines. We evaluate our framework’s success in achieving this through two primary metrics.

1. **Usability**: We want our system to be easy to use with respect to all parties involved. This means that an end-user should be able to select a set of blocks or an already fully specified pipeline, and, with minimal effort, attain a solution to his data science problem. To this end, the MLBlocks API should be clean and user-friendly.

2. **Accessibility**: We also want our system to help make the vast amount of data science tools that exist more readily accessible to both novice and experienced

---

\(^8\)https://github.com/HDI-Project/BTB
\(^9\)https://github.com/HDI-Project/MLBlocks
users alike. That is, we must easily be able to integrate both new and existing tools into our MLBlocks framework and library. A domain expert looking to contribute his expertise to our system for the benefit of the end user should be easily able to do so.

In outlining these goals, it is important to keep the scope of our problem in mind. We specifically note that the MLBlocks framework represents an end-to-end solution of pipeline creation, from block composition to hyperparameter tuning. While our end result is a fully tuned and specified end-to-end pipeline, we explicitly note that we do not aim to handle things such as data loading or pipeline recommendation. While these are both interesting potential additions to our work, they remain out of the scope of this project.

1.4 Thesis roadmap

We organize the rest of the thesis as follows:

Chapter 2 details the high-level design of the MLBlocks framework. We outline the abstractions of MLBlocks as well as how they connect to each other to form an end-to-end pipeline creation framework. Chapter 3 outlines the details of the MLBlocks framework, detailing each component of the system as well as how we build up the MLBlocks library. Items such as how we integrate with third-party libraries, how we maintain the MLBlocks library, and how a contributor might contribute a new block are all outlined here.

Chapter 4 details the user API of the MLBlocks system. We provide various code examples of how a user would use MLBlocks to update pipeline hyperparameters and produce results for data. Chapter 5 dives into one part of this API specification, detailing the specifics of the data inputs the MLBlocks system expects. We particularly illustrate how a user would process their data to use with MLBlocks.

Chapter 6 covers experiments run with real pipelines created via the MLBlocks framework on real data. In this chapter, we validate the utility of MLBlocks by showing that we can easily create pipelines that function on a variety of real datasets.
Chapter 7 further validates MLBlocks as a system in the real world by detailing the use of MLBlocks as a component in the DeepMining\textsuperscript{10} system. Particularly, we detail the ease of integrating MLBlocks into this system and the advantages we achieve by doing so, in the form of code readability, maintainability, and user-friendliness.

Chapter 8 outlines several potential future additions to the system to illustrate the potential of MLBlocks beyond the problem of pipeline creation. Finally, we end by tying this all together by summarizing our key findings and contributions in Chapter 9.

By the end of the thesis, we will outline an understanding of MLBlocks as a powerful tool for making data science more accessible to both novice users and data scientists alike.

\textsuperscript{10}https://github.com/HDI-Project/DeepMining
Chapter 2

System design overview

In implementing an end-to-end machine learning pipeline creation framework, users must be allowed to create their desired pipelines. Thus our framework needs to support a large number possibilities for pipelines. To address this, our framework provides a library of blocks derived from a variety of machine learning libraries.

Additionally, once a user composes a pipeline, he must be able to optimize the pipeline to produce reasonable results on his data. That is, it is necessary to have an abstraction that facilitates tuning of these pipelines. To this end, we introduce the MLPipeline, a pipeline abstraction facilitates easy specification and tuning.

In this chapter, we describe the architecture of the MLBlocks framework, largely consisting of the two components described above, as well as the motivations for the design decisions made.

2.1 Design goals

Block design: As previously mentioned, our MLBlocks framework must accommodate the needs of many users, producing reasonable pipelines for numerous data and problem types. To achieve this, we must enable integration of a large number of third-party machine learning libraries, which may be specified via a number of languages and frameworks.

In addressing this, we aim to make it easy for expert contributors to integrate
their new and existing libraries with our system. This yields two concrete challenges and goals:

1. **Interface familiarity**: Data science and machine learning experts would be glad to contribute their work, but may not be comfortable programming in certain ways. For example, an expert on Bayesian optimization may not have experience with the dataflow programming idioms used in tensorflow models. This inherently limits currently existing pipeline modeling systems to exclude certain useful libraries that may not comply with their specification. In designing our library, we must provide a uniform block interface that a vast majority of experts are familiar with.

2. **Developer load**: An expert may simply not have the time or be willing to spend the effort to go through an integration process often involving refactoring their code or writing specific software just for the sake of integration. Our library must minimize the amount of language and implementation-specific integration code a contributor must write to contribute a block.

At the same time, it is also necessary for our library to integrate well with the rest of the **MLBlocks** system. This includes items such as making hyperparameter information readily available.

**Pipeline design**: Once a user has access to all of the blocks he needs, he then must then create his pipeline. This process is often nontrivial and it requires some level of domain expertise to achieve reasonable results. We aim to develop a new pipeline abstraction that facilitates this process. Along these lines, we outline two concrete goals to measure our success:

1. **Ease of composition**: It must be simple for a user to select his blocks and create a pipeline from them. Concretely, this means that we primarily aim to minimize the amount of code the user must write to create his pipeline. At the same time, however, we must also allow users some degree of flexibility in selecting blocks. For example, in many cases, a user may compose his pipeline
solely from blocks present in our **MLBlocks** blocks library, but for reasons such as security, it may not always be practical for a user to do so. Therefore, we must achieve a balance between these two goals.

2. **Performance**: Once a user has connected his blocks to form a pipeline, he must then be able to achieve reasonable predictions using his data. It goes without saying, then, that we must provide basic functionality to fit to a pipeline and to produce predictions from it. This allows us to evaluate pipelines on a variety of metrics. In addition to providing basic functionality, we aim to optimize pipeline performance by allowing end-to-end tuning of pipelines. This entails exposing all tunable hyperparameters of the pipeline to our tuning mechanism in as flexible a way as possible.

2.1.1 **Non-goals**

We do not currently aim to perform any data ingestion, cleaning, or formatting with the **MLBlocks** system. That is, we expect data in certain formats and do not aim to provide functionality to convert data into these formats.

Additionally, we do not currently aim to automate the composition of our end-to-end machine learning pipelines, as that is out of the scope of our current system. However, as a fully-automated system is perhaps the simplest way for a user to obtain an end-to-end model that performs well on his data, we must consider this when designing our abstraction. We would like our abstraction to be extensible so that it may provide such automation capabilities in the future.

2.2 **High level design**

At a high level, the **MLBlocks** system consists of two main components: a library of blocks and the **MLPipeline** abstraction. These two components are connected via a collection of parsers. We describe each of these components and how they interact with each other at a high level below. A diagram of our high level design is shown in
Our system first consists of a library of blocks which can be used to compose a pipeline. These blocks are specified via JSON\(^1\) files containing both information about the functions used to operate on given data as well as various hyperparameter metadata. These JSON files allow for easy integration with various third-party libraries while making the hyperparameters of each block explicitly accessible.

The **MLPipeline** is a modeling class that represents an end-to-end pipeline, composing all steps from featurization to prediction. Currently, it is implemented in Python 3 [12]. An **MLPipeline** is composed of **MLBlock** objects, which are Python representations of blocks. Like our JSON blocks, these contain various hyperparameter metadata, storing this information in specified data container objects called **MLHyperparams**. Additionally, they provide basic functionality to fit on and produce results from data.

Our JSON block files are parsed and loaded into **MLBlocks** at runtime by one of various parsers maintained in the **MLBlocks** framework. It is worth noting that we do not persist said **MLBlocks** in the form of Python code; we simply specify an **MLBlock** as a virtual class and load all the necessary information from JSON files upon initialization. With that said, once loaded, each **MLBlock** Python object contains all of the information specified in the JSON including linked functions for operating on given data.

To create an **MLPipeline**, a user simply must specify a list of block names as well as an optional topological ordering of said blocks. The **MLPipeline** then uses the appropriate parser to load each JSON into **MLBlocks** and can now operate on given

---

\(^1\)[http://www.json.org/]
data. The MLPipeline creation flow is shown in figure 2-2.

We provide more detailed discussion of the advantages and disadvantages each component offers in Chapter 3.
Chapter 3

MLBlocks

In this chapter, we discuss the detailed design of the MLBlocks framework. We begin by describing the block library, which stores its meta information in JSON files. In doing so, we examine our MLBlocks parsers, which use these JSON files into MLBlock objects. We then discuss the MLBlock, which is an in-memory Python representation of a block. Finally, we describe the MLPipeline, which ties together MLBlock objects together to be able to perform operations on data.

3.1 A library of blocks

In this section, we outline the details of design of blocks library, as shown in figure 3-1.

This block contains the specification for a random forest classifier and links to scikit-learn code. As we can see, it links the scikit-learn fit and predict methods to our JSON and exposes hyperparameters.
Figure 3-2: A sample JSON file with specifications for a block

```json
{
    "name": "rf_classifier",
    "class": "sklearn.ensemble.RandomForestClassifier",
    "fit": "fit",
    "produce": "predict",
    "fixed_hyperparameters": {
        "class_weight": null
    },
    "tunable_hyperparameters": {
        "criterion": {
            "type": "string",
            "values": ["entropy", "gini"]
        },
        "max_features": {
            "type": "float",
            "range": [0.1, 1.0]
        },
        "max_depth": {
            "type": "int",
            "range": [2, 10]
        },
        "min_samples_split": {
            "type": "int",
            "range": [2, 4]
        },
        "min_samples_leaf": {
            "type": "int",
            "range": [1, 3]
        },
        "n_estimators": {
            "type": "int_cat",
            "values": [100]
        },
        "n_jobs": {
            "type": "int_cat",
            "values": [-1]
        }
    },
    "root_hyperparameters": ["criterion", "max_features", [...],
    "conditional_hyperparameters": {}"
}
```
3.1.1 JSON specification

We specify meta information about each block in our library using a JSON file. An example of which is shown in figure 3-2. The specification has the following information:

Methods, library and class: It specifies the library, the class and its corresponding fit and produce methods. This allows us to decouple the logic for fitting and producing data from a model from the rest of the hyperparameter specification. This is useful because it allows us to potentially reuse the hyperparameter specification of the JSON. For example, if a user wants to use a different implementation of an already existing classifier, he can simply replace the class and fit and produce methods with the appropriate information for his implementation.

Hyperparameters: We first notice that there are several fields for fine-grained hyperparameter specification. Each hyperparameter present in the fixed and tunable hyperparameter fields is identified by name and maps to a value. Each tunable hyperparameter additionally has a type and range or values attribute. This information is generally needed when tuning hyperparameters. Separately, we note that we can identify which hyperparameters are root and which hyperparameters are conditional. Hyperparameters mentioned in the list of root hyperparameters need to specified for a pipeline to be fitted to data, while conditional hyperparameters only exist when certain values are picked for the root hyperparameters. This gives us coverage over all of the hyperparameter variants we identify in section 1.1, to allow for the full hyperparameter specification of a block.

We are aware that while our JSON specification tends to provide enough information to create blocks in most use cases, different libraries may need more information to create blocks. Therefore, we allow extensions to our JSON specification. For example, because keras needs information about various layers to instantiate a model, its JSON specification differs slightly from our standard JSON. An example keras JSON is shown in figure A-1.

Advantages of using JSON: We primarily choose to do this using JSON as it offers
a variety of advantages. It integrates well with most modern languages, reducing the amount of code needed to integrate a block with our framework, and it is both portable and extensible, allowing it to be extended and potentially be used in other systems in the future. Additionally, however, it is widely-used and language-agnostic, facilitating easy understanding of specification of blocks; a developer does not need to be familiar with any particular programming paradigm or the intricacies of a programming language to be able to use or contribute a block.

3.1.2 JSON parsers

To use specifications described in the JSON files and translate them into objects in code, we use what we call the block-parser paradigm. In our case, this means that in parallel to maintaining a repository of JSON specification of blocks, we maintain a collection of parsers. Concretely, a parser takes in a JSON file in our specified format, and outputs an MLBlock object, which we will describe in section 3.2.

This primarily allows us to take advantage of the fact that in many libraries, each block follow the same API. For example, in scikit-learn and many other libraries, each block provides a fit and predict method and hyperparameters are specified via class constructor. keras blocks are composed by stacking layers and compiling a model. Because the logic for creating and using a block is often the same across one or many libraries, we do not need to rewrite or even wrap entire libraries to integrate them with our system; we simply need to write one parser that serves many blocks. This makes integrating third-party libraries with our system relatively simple.

We currently provide several default JSON parsers that function with popular machine learning libraries such as scikit-learn and keras. However, just as we give users and contributors the flexibility to specify custom JSON formats in special cases, we allow them to specify custom parsers. We further describe our base parser and custom parsers in section 3.1.3.

Language agnosticity: We note one particular advantage with regards to the JSON block-parser model that stems from the fact that JSON specifications are relatively language agnostic. While we currently only support Python, our design makes it
relatively simple to port our library to another language. One would simply have to write a new JSON parser for said language, potentially change the class, and fit and predict methods. The hyperparameter specifications would remain the same. Depending on the language and parser implementation, it is possible to provide a thin translating layer (e.g. Jython for Python to Java translation) in the parser layer, in which case it may not even be necessary to change the JSON files at all.

3.1.3 Building the blocks library

To this point, we have described basic mechanisms for integrating third-party libraries with the MLBlocks architecture via the JSON block-parser model. One might notice, however, that we have not made explicitly clear how we will build up our fully integrated library of JSON blocks.

In building our MLBlocks blocks library, and to a lesser extent our parser collection, we adopt an open-source model. Concretely, we begin by providing several blocks that we can use to construct basic pipelines for a variety of data types. The base blocks of our library at the time of this writing are enumerated in table B.1.

However, for more particular use cases and for pipelines that provide better performance, we allow users to contribute new blocks to our library. This approach has the key advantage of allowing us to collect and leverage the knowledge of domain experts around the world. In the following sections, we discuss the process of facilitating easy open-source contribution in order to achieve coverage over a large number of third-party libraries. We now discuss in detail how a contributor would integrate his block or block library with our system.

**Base JSON and MLParse**: In facilitating the integration of third-party data science libraries with MLBlocks, we primarily aim to allow users to contribute to blocks by writing and contributing JSON files. Concretely, this means that we aim to minimize the amount of Python code required to contribute blocks. To accomplish this, we provide a base JSON format in addition to a base parser, called the MLParse. The base JSON is analogous to the one shown in figure 3-2.

We note that a third-party block only needs to provide a constructor that takes in
hyperparameters in addition to the basic required methods\(^2\) in order to integrate with these base abstractions. This is an extremely common way many libraries specify their blocks; for example, every \texttt{scikit-learn} block follows this format. Therefore, by having these base abstractions in place, we allow easy integration with a large number of libraries. If a developer writes a library that follows this method of specifying hyperparameters via class constructor in each block, he can already integrate all of his blocks into our library without having to write any additional Python code to do so.

**Custom parsers:** In many cases, a third-party library does not follow the specification defined in the base JSON and \texttt{MLParser}, but still follows a more or less uniform API across the library. An example of this is \texttt{keras}. While the \texttt{keras} library does provide fit and produce methods, to construct a \texttt{keras} block, a user needs to stack layers and compile a model. Because the \texttt{MLParser} expects block model construction to be as simple as passing hyperparameters into a constructor, we cannot use it with \texttt{keras}.

In these cases, we encourage users to write their own custom parsers and define a custom consistent JSON format, following the base JSON specification as much as possible. The \texttt{keras} JSON format is shown in figure A-1. As we can see, while we specify additional fields to facilitate model construction, we expose hyperparameters in much the same way as the base JSON specification.

Implementation-wise, we require that custom parsers subclass the \texttt{MLParser} base class. In subclassing \texttt{MLParser}, we only require one method be overridden:

- \texttt{build\_mlblock()}: returns an \texttt{MLBlock} instance with the appropriate hyperparameter lookup tables initialized as well as all required external and internal instance methods set.

In implementing this method, we note that in addition to specifying how \texttt{fit} and \texttt{predict} are linked, we must specify the logic for one internal \texttt{MLBlock} method:

\(^2\)The only required method is the produce method, which performs actual transformations to input data. However, many blocks must be fit to the data before data is produced, so we allow specification of this as well. Implementation-wise, if no fit method is specified, during a fit call of the pipeline, nothing happens at that step.
update_model(): updates the MLBlock based on the stored hyperparameter values in the MLBlock's lookup tables. Because each library may update models in different ways, the user must specify this logic.

It is worth noting that our default parsers cover a majority of popular blocks used in practice and that each parser is designed to serve a large number of blocks. Thus, we do expect the number of custom parsers, and thus the amount of user-required integration code, to be relatively small.

Custom functions: We note that it is not always the case that we are integrating existing third-party libraries such as scikit-learn into our system. Often, users write their own custom functions to perform operations such as feature extraction on certain pieces of data. Our architecture enables users to use said custom functions locally. For example, a user who wants to use his own custom audio feature extraction functions can first create a JSON and link his local custom function to the class, fit, and produce fields of the JSON. The user can then simply specify the path to the directory containing the JSON when composing a pipeline. We encourage users who create custom functions to contribute them back to our library along with their associated JSON files when possible.

A concrete example of Python code and JSON specification for a HOG featurization step are shown in figures A-2 and A-3. As we can see, the full delimited module path to the HOG code is specified in the "class" field, and the make_hog_features method is linked to the "produce" field of the JSON. We also can observe that we do not specify any method to link to the "fit" field, indicating that the HOG step should not be fit.

In other cases, a custom function may be the easiest way for a contributor to integrate his third-party library with his blocks without needing the overhead of a custom parser. This tends to be the case for smaller third-party libraries.

In these cases, however, we would still like users to contribute back their custom blocks. A custom step for one user’s application may be a solution for another user’s problem. Therefore, we store custom functions in our library to allow users to contribute their linked block code in addition to the corresponding JSON file.
Functions vs parsers: We provide two mechanisms for integrating third-party libraries with MLBlocks: we can either import the library and write custom function wrappers around it, or we can write a parser to create MLBlock objects that link to the library blocks. Therefore, there may be some ambiguity in how a contributor should integrate his third-party library.

To clarify this ambiguity, we propose that developers should use whichever mechanism results in the least contributed code. Concretely, this means that custom functions should be written when a contributed library is small, and consists of only a few blocks. In this case, it is often easier for a developer to write custom function wrappers around his blocks to support the fit and produce API of the base JSON And MLParser.

However, in cases where a library consists of a large number of blocks with a consistent API, a custom parser for the entire library should be written. This involves less developer load than writing a custom function over each block in the library, and minimizes the amount of code necessary to contribute the library.

3.2 The MLBlock abstraction

The Python representation of a block, the MLBlock, shown in figure 3-3. Currently we implement this via Python 3, but the interface’s design is such that we can implement it in other languages in the future. MLBlock instances are instantiated using the JSON parsers described in section 3.1.2.
An MLBlock instance contains two lookup tables, providing quick access to fixed and tunable hyperparameters, respectively. For fixed hyperparameters, we simply map hyperparameter names to values. For tunable hyperparameters, we map hyperparameter names to MLHyperparam objects, which are thin wrappers around tuning information that provide certain utility functions such as random initialization. It also contains a model, which serves as an entry point to the code that performs operations on data, as well as an external-facing interface for said operations.

We choose to keep the API for MLBlocks relatively simple to reduce developer load when creating an MLBlock. As this is the case, we only support two user-facing methods for MLBlock: fit and predict. We briefly detail these below:

- fit(X, y, **kwargs): fits the block step to the training data X, and the training targets y. Also takes in any other required arguments for fitting. For example, a random forest classifier may take in sample weights when fitting.

- produce(X, **kwargs): produces new data from the input data X. As with fit, this may take in any other required arguments for producing data. The output data is often predictions, as in the case of a random forest classifier, but generally, the produce method transforms data.

By simply specifying hyperparameters and the implementation of these three functions, we can obtain a tunable single step that can be seamlessly integrated into a full machine learning pipeline.

### 3.2.1 Virtual classes

While the interface for an MLBlock is relatively simple, we instantiate an MLBlock object in a unique manner. To describe this, we introduce an idea of a virtual class. A virtual class is a class that is instantiated at runtime. That is, when we initialize an instance of the class, in addition to supplying class attributes, we also define the method implementations.

In the particular case of an MLBlock object, we define the fit, predict, and update_model methods when we create the object. We accomplish this by taking
advantage of the fact that Python allows method overriding by assignment and patch the method implementations onto the MLBlock objects on creation. This instantiation is all accomplished using our collection of JSON parsers, described in section 3.1.2.

We note the alternative possible design here, in which for each block, we specify a base class and API. In this design, for a user to implement a new block, he would subclass said base class and implement the logic to perform operations such as fitting and predicting via Python code. To implement a library, he would have to do this for every block in the library.

We note that our design gives us several advantages over this alternative approach in particular. Primarily, this allows us to avoid creating and storing language-specific block code, keeping block information in the JSON files. This has the following advantages:

- It allows us to keep the size of our library relatively manageable both in terms of storage and readability.
- The main advantage, however, is that we are able to keep the specification of block loosely coupled from any language-specific implementation of blocks. It allows us to use the JSON files elsewhere and develop a new library from scratch.
- Finally, in case we want to add more functionality to MLBlock abstract class, it allows us to do that easily, by only changing one class and adapting the parsers. Compare that to updating all the class files if each block was written as a class of its own.

3.3 Contributing blocks

With everything mentioned so far in mind, we can now illustrate how a machine learning expert would go about contributing a block once he has written a library or custom function. A flowchart for this process is illustrated in figure 3-4.

As we can observe, the number of steps required to contribute a block is minimal. Moreover, we argue that each of the steps required is minimal. If a developer has
3.4 Maintaining the library

One concern related to integrating a large number of third-party contributions is that we maintain the quality of the \texttt{MLBlocks} library. This concretely means three things. First, any updates to our repository must adhere to the already-established format of the rest of the repository. This is important to keep our library as easy-to-understand as possible for both end-users and contributors. Second, each contribution must reliably function in the way a user would expect. As a very basic example, we must be able to load a contributed JSON into an \texttt{MLBlock} object. Finally, updates to already-existing components of our system must not break established functionality. For example, many users may be using a particular block in a variety of pipelines, so any updates to that block must preserve basic functionality. We describe several techniques that we use to achieve these goals below.

3.4.1 Code format

In addition to having a fixed JSON format for JSON files to adhere to, we also keep consistent formatting of the Python code that composes the \texttt{MLBlocks} framework.
From a contributor’s perspective, this means that all contributed custom parsers and functions must adhere to our format specifications.

We choose the PEP8\(^3\) format for our project because it is officially supported by the developers of Python and it is widely used. This formatting requirement is enforced via flake8\(^4\) and Travis CI\(^5\) when a new contribution is made. Therefore, we encourage contributors to lint their code before contributing, and only accept contributions that adhere to the PEP8 style guidelines.

### 3.4.2 Testing

In the current MLBlocks repository, we address this by requiring tests for each individually contributed block. For ease of integration, we provide various test templates for different data types. These tests load toy data, format them into the expected formats, and run the initialized blocks on them. An example test for a random forest classifier is shown in figure A-4.

Each test follows a basic layout defined in our test template. We first test that the specified block loads properly from the base JSON. In doing so, we particularly check that an initialized MLBlock contains the hyperparameters as specified by the JSON file. We also check that it contains the appropriate step, as specified by the constructor argument. We then test the basic fit and predict functions of the pipeline as a whole, which tests the fit and produce method of the block.

We note that the functions not tested by this layout are common to all pipelines and blocks. While fit and predict are supplied via soft linking in JSON block specification, methods such as those to update hyperparameters and models function the same way for all pipelines. Thus, we cover these with our general MLBlocks unit tests.

All unit and end-to-end block and pipeline tests run every time a new block or pipeline is contributed via tox\(^6\) and Travis CI. This helps guard against potential breakages of our system as our library is updated.

\(^3\)https://www.python.org/dev/peps/pep-0008/
\(^4\)https://github.com/PyCQA/flake8
\(^5\)https://travis-ci.org/
\(^6\)https://tox.readthedocs.io/
We note that we are mainly interested in testing the functionality of blocks and pipelines and not performance. This decision is made largely in the interest of keeping test runs relatively short. For example, certain neural models may need to train on tens of thousands of training examples to obtain optimal results. Furthermore, we have little control over how users will tune hyperparameters, and hyperparameter choice itself largely affects performance. We accomplish this by limiting the amount of data fed into each pipeline during the tests and by not tuning the hyperparameters of each pipeline at all.

In the future, we may introduce more comprehensive, automated end-to-end tests that fully train and tune certain pipelines containing the blocks, but for the sake of initial validation of blocks, basic functionality tests suffice.

3.4.3 Manual Review

While we are able to check basic functionality of contributed pipelines and blocks, we still require a brief manual review of any contributions before adding them to our MLBlocks library. This is particularly important in special cases such as custom JSON formats for certain libraries. As we maintain our repository on GitHub\textsuperscript{7}, this is done in a relatively straightforward manner. We simply review pull requests for contributions and merge them into our library.

\textsuperscript{7}https://github.com/HDI-Project/MLBlocks
Chapter 4

MLBlocks User API

We now describe the user API of MLBlocks. Essentially every user interaction with MLBlocks is done through the MLPipeline class. The MLPipeline allows users to operate on data and expose hyperparameters for tuning. We describe the design and API of the MLPipeline class and provide several usage examples to illustrate how a user would create a pipeline with the MLBlocks system in detail below.

4.1 The MLPipeline abstraction

To tie different MLBlock objects together into an end-to-end pipeline, we create the MLPipeline class. This class provides functionality that allows the steps in a given pipeline to be tuned end-to-end together regardless of implementation details (such as source library).

Implementation-wise, at a high level, we keep a lookup table of MLBlock name
to MLBlock instance. This allows us quick access to each MLBlock instance and the information that each MLBlock instance contains. Additionally, we maintain a separate data structure for mapping a topological ordering of MLBlock steps, although we only support linear orderings currently.

We also provide several ways of specifying MLPipeline and several functions for MLPipeline to operate on data and expose hyperparameters for tuning. We describe these along with sample usage of MLPipeline in further detail in chapter 4.

### 4.2 Pipeline specification

```python
pipeline = MLPipeline(['HOG', 'random_forest_classifier'])
```

Figure 4-2: Initializing an MLPipeline with just one line of code

We initialize an MLPipeline by passing a list of names of JSON files into the MLPipeline constructor. By default, we assume that these JSON files are located in our repository’s block library.

Currently, we only officially support specifying blocks in a pipeline in a linear order. That is, the first block of a pipeline feeds its output into the second block, and so on, until we reach the final step, where we make predictions. However, we have scaffolding in place for a user to specify a topological ordering of steps for non-linear pipelines in the future.

### 4.3 Fitting and predicting

```python
mnist = fetch_mldata('MNIST original')
X, X_test, y, y_test = train_test_split(mnist.data, mnist.target, train_size=1000, test_size=300)
pipeline.fit(X, y)
predictions = pipeline.predict(X_test)
```

Figure 4-3: Fitting and predicting with an MLPipeline on image data
We provide several methods for actually performing operations on data for an entire `MLPipeline`. We briefly outline these below:

- **`fit(X, y, fit_params)`**: Iteratively fits each step to the output of the previous step and the training labels. The first step’s input is simply the training data, specified in an array-like format. We specify any step-specific fit parameters via the `fit_params` argument, which is a mapping of (step name, fit parameter name) to fit parameter value. This allows for a fine-grained specification of the input to each particular `MLBlock`, and allows us to take advantage of the full functionality of each `MLBlock` model. For example, we can support multi-table steps by passing in auxiliary tables to those steps via `fit_params`. Of course, if no steps require no auxiliary parameters, then we can simply choose not to specify `fit_params`.

- **`predict(X, predict_params)`**: Iteratively applies the produce method of each `MLBlock` step to the transformed output of the previous step. As with fit, the input to the first step is simply the data to predict on. Also similarly, we specify any step-specific predict parameters via the `predict_params` argument, which is analogous to the `fit_params` argument in the fit method. We note that in a pipeline, the final step should produce predictions, so this method is named appropriately.

We also provide two other methods that support batch loading of data: `fit_generator`, and `predict_generator`. These methods function similarly to the `fit` and `predict` methods described above, but instead of taking array-like data structures, they take in generators which output batches of data: `(X, y)` in the case of `fit_generator`, and `X` in the case of `predict_generator`. This allows the user to create pipelines for large data formats that do not fit into memory, such as audio files, as long as the `MLBlock` steps in the pipeline support incremental fitting. Additionally, this approach abstracts the logic of loading batches of data out of our system, giving the user full control over said implementation. This is desirable, as in many cases,
the data a user has is unique and said user may want to perform the loading in a specific way.

### 4.4 Exposing hyperparameters

```python
>>> tunable_hp = image_pipeline.get_tunable_hyperparams()
>>> print(\n    
    \n    .join(map(str, tunable_hp)))

Hyperparameter: Name: num_orientations, Step Name: HOG, Type: int,
  → Range: [9, 9], Value: 9
Hyperparameter: Name: num_cell_pixels, Step Name: HOG, Type: int,
  → Range: [8, 8], Value: 8
Hyperparameter: Name: num_cells_block, Step Name: HOG, Type: int,
  → Range: [3, 3], Value: 3
Hyperparameter: Name: criterion, Step Name: rf_classifier, Type:
  → string, Range: ['entropy', 'gini'], Value: entropy
Hyperparameter: Name: max_features, Step Name: rf_classifier,
  → Type: float, Range: [0.1, 1.0], Value: 0.9134616693335704
Hyperparameter: Name: max_depth, Step Name: rf_classifier,
  → int, Range: [2, 10], Value: 2
Hyperparameter: Name: min_samples_split, Step Name: rf_classifier,
  → Type: int, Range: [2, 4], Value: 3
Hyperparameter: Name: min_samples_leaf, Step Name: rf_classifier,
  → Type: int, Range: [1, 3], Value: 3
Hyperparameter: Name: n_estimators, Step Name: rf_classifier,
  → Type: int_cat, Range: [100], Value: 100
Hyperparameter: Name: n_jobs, Step Name: rf_classifier, Type:
  → int_cat, Range: [-1], Value: -1
```

Figure 4-4: Obtaining hyperparameter values of an MLPipeline

```python
hp_dict = {('rf_classifier', 'max_depth'): 9}
image_pipeline.set_from_hyperparam_dict(hp_dict)
```

Figure 4-5: Updating hyperparameter values for an MLPipeline

The hierarchical architecture of our MLPipeline allows us to access hyperparameters easily by looking up the MLBlock to which the hyperparameter belongs via our MLPipeline-level lookup table, and subsequently looking up the hyperparameter in
one of the two MLBlock-level lookup tables. However, for convenience, we provide several methods to access hyperparameters, which we detail as follows:

- **get_fixed_hyperparams()**: Gets all of the fixed hyperparameters of each MLBlock contained in the MLPipeline as a mapping from (step name, hyperparameter name) to hyperparameter value.

- **get_tunable_hyperparams()**: Gets all of the tunable hyperparameters of each MLBlock contained in the MLPipeline as a list of MLHyperparam objects. Note that because an MLHyperparam object contains information about the step a tunable hyperparameter belongs to, we do not need to specify the step name externally.

Additionally, we provide methods for updating the hyperparameters of the pipeline that take inputs analogous to the outputs of the corresponding get methods.

- **update_fixed_hyperparams(fixed_hyperparams)**: Updates the fixed hyperparameters of the MLPipeline given fixed hyperparameters to updates specified as a mapping from (step name, hyperparameter name) tuples to values to update to. Fixed hyperparameters not specified in the input are not updated.

- **update_tunable_hyperparams(tunable_hyperparams)**: Updates the tunable hyperparameters of the MLPipeline given tunable hyperparameters to update specified as a list of MLHyperparam objects. Like with fixed hyperparameters, tunable hyperparameters not specified in the input are not updated.

Often, applications that use MLBlocks may not want to create MLHyperparam objects in order to update tunable hyperparameters. We address this by introducing an additional method:

- **set_from_hyperparam_dict(tunable_hyperparams)**: Updates the tunable hyperparameters of the MLPipeline given tunable hyperparameters to update specified as a mapping from (step name, hyperparameter name) tuples to values to update to. This is useful because, in many cases, a user may be only
interested in updating the value of a tunable hyperparameter rather than any of the other metadata (such as range or type).

Note that this means that the models of the MLBlock objects containing the hyperparameters to be updated are updated in addition to the hyperparameters themselves, which is why we need users to specify the logic to update a model in the `update_model` method when creating an MLBlock object in a parser.

These methods allow us to abstract the lookup and update logic away from the user, which contributes to the ease-of-use of our system in general.
Chapter 5

Data integration

As we aim to provide a uniform API for the variety of third party libraries that we integrate with, it makes sense to provide a uniform data specification in MLBlock.

We achieve this at a high level by requiring numpy [10] array inputs to the MLPipeline fit and predict methods. One might notice that by doing this, a contributed block that expects to be the first step of a pipeline and does not correspond to this API must be changed in order to integrate with our library. As we want to minimize the amount of integration effort required, this is less than ideal. To remedy this, we introduce several data converters that convert between our input numpy arrays and the data types these blocks expect as inputs.

Even with this uniform high-level specification, however, data is often messy in practice, and numpy arrays are very general. This means that it may be unclear to a user how to format his data as numpy arrays.

In this chapter, we describe several concrete data types and how they integrate with MLBlock.

5.1 Data types

We address seven data types commonly found in data science problems. These data types are as follows: image, text, audio, single-table, multi-table, graph, and multi-graph. We expect most datasets found in real world use cases to fall into one
of these seven categories, as they are reasonably broad.

By addressing each of these individual data types, we aim to achieve a high degree of coverage. That is, a user with no domain knowledge should simply be able to come to our repository, pick a series of blocks appropriate for their data type, and produce reasonable performance on their data.

We shall describe these data types and how our system integrates with them in more detail in the following sections.

5.2 Single-table

![Figure 5-1: Format for the single-table datatype.]

The single-table datatype is already provided as a matrix, so no special transformations are necessary, as we can see in figure 5-1.

An example of a single table dataset would be the iris dataset, which consists of a table of flower attributes, such as sepal width.

We note that, in addition to requiring no special transformations, most real world datasets can be converted to a single-table data format reasonably easily. For example, almost all relational data that can be contained in a single table can be represented in the single-table data format, hence the name. Thus, in integrating with MLBlock, most data can simply be passed in without needing special preprocessing.

5.3 Image

In general, for image pipelines we expect the input format to be that of an array, with the first index mapping to the number of samples, as shown in figure 5-2. This can be simple RGB values loaded from image files, or a flattened binary representation in the case of grayscale images.
A famous example of image data is the MNIST dataset, which consists of grayscale images of handwritten digits.

As an example, we describe how a user might preprocess a grayscale image dataset. For each grayscale image, we can simply accept input as a binary representation of the image. A user would first load the grayscale binary values of each pixel in the image as an array with the same dimensions as the image. He would then flatten the image into a one-dimensional array, with length equal to the product of the dimensions of the image. These flattened arrays can be composed to form one larger array for the entire dataset, with the first index mapping to the number of samples.

In the case of a colored image, we can carry out a similar procedure for each RGB channel before concatenating them into an array with three rows, each of which has a length equal to the product of the dimensions of the image. We note that this means that each image does not need to be mapped to a flattened array—the only requirement is that the first index maps to the number of samples.

5.4 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, these should be concatenated, as shown in figure 5-3.
An example of text data would be a collection of movie reviews, labelled with positive or negative sentiment.

We expect most text preprocessing to be fairly straightforward based on this description. It is worth noting that each text example being represented as a concatenated string is an invariant that holds regardless of how long a text example is. In the case of sentences, we may expect this concatenated to be relatively short, and in the case of books, we may expect it to be longer. In both cases, all text in the example should be concatenated.

5.5 Audio

![Audio Icon]

\[
\begin{bmatrix}
[3,4,...],[3,...],[4,...],... \\
\vdots
\end{bmatrix}
\]

Figure 5-4: Format for the audio datatype

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, as shown in figure 5-4.

One example of an audio dataset is the Urban Sounds dataset, which consists of labeled audio files of various sounds that occur in an urban setting.

To preprocess an audio file, two steps are often required. The first step is to load the data into an array representation. This can be done with a number of tools, including the popular `scipy.io.wavfile`. The user should then give structure to this loaded data by creating segments of audio. Sample code to do this is provided in figure A-1.
5.6 Multi-table

The multi-table datatype refers to matrix data that is augmented with extra matrix data, possibly sharing a primary key. In this case, \( X \) refers to primary matrix, we let \texttt{fit_params} contain the auxiliary matrix data, as shown in figure 5-5. As we can see in the figure, the two tables are related via the highlighted entries.

One example of a multi-table dataset is one of retail store data, where the primary table stores transactions, and an auxiliary table stores information about the products involved in each transaction.

We note that there may be some ambiguity as to what constitutes a primary matrix and what constitutes auxiliary matrix data. For example, when using transaction and product tables to predict whether a customer will buy an item, it may be unclear which should be the primary table. We define the primary table as the one whose rows correspond to the labels. That is, in the above example, if the labels are whether a customer will buy an item or not, the primary table should be primarily keyed by customer. If multiple tables satisfy this requirement, then we leave this to user discretion, as the choice of tables should not affect the functionality of the pipeline results.

We also note that \texttt{fit_params} does not have the requirement that any data passed in must be a numpy array. Thus, a user can preprocess any auxiliary data how he expects and can pass it into \texttt{fit_params} in the format desired. This choice of format is often dependent on the first block used in the pipeline, and so we leave this decision to the user.
5.7 Graph

The graph datatype refers to matrix data that is augmented with a graph, depicting the relationship between samples in the matrix. For this datatype, we let $X$ refer to the primary matrix, while `fit_params` contains the graph, represented as a Python dictionary, as shown in figure 5-6. As we can see, the primary matrix, graph, and graph dictionary are related by nodes denoted by highlighted elements. A subsequent row in the primary matrix, could, for example, correspond to another node. We note nodes in the graph do not have to be the primary key of the primary matrix.

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

We observe that there may be some confusion with regards to what the graph structure passed into `fit_params` should be in particular. While in general, we allow flexibility with regards to the data passed into `fit_params`, we recommend usage of the GML data format [6]. We choose GML as the data format of choice because it is used as an input to a variety of graph data blocks such as `networkx`. An example GML format is shown in figure A-6. Note that this means that we recommend passing in loaded dictionary representations of GML rather than filenames identifying GML files. Note also that depending on the blocks used, other dictionary representations of graphs may also be used.
5.8 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, as shown in figure 5-7. We observe that the primary matrix is related to nodes in both graphs via the highlighted elements. A subsequent row of the primary matrix could correspond to nodes in either graph. As with single-graphs, nodes in the graph do not have to be primary keys of the primary matrix.

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

While we do allow flexibility in specifying these graphs, once again we recommend usage of the GML data format, as we do for single-graph datasets.
Chapter 6

Experiments

In this chapter, we perform experiments to validate MLBlocks’ ability to easily produce a pipeline that solves a user’s particular use case.

To this end, we test our system’s ability to perform on various data modalities. In doing so, we gain confidence that a user will be able to implement a pipeline that solves his particular data science problem given his particular dataset.

6.1 Experiment setup

6.1.1 Datasets

In evaluating our system on different data modalities, we must choose appropriate datasets that may represent datasets found in the real world. At the same time, it is useful if the datasets are relatively well known.

With these criteria in mind, we select five different datasets to represent five different data modalities.

- **Wine (single-table):** The wine dataset [5] contains a single table of 178 instances of observations about wine. The table consists of 13 attributes, and each instance is mapped to one of three types of wine. The goal of the dataset is to classify new instances a particular type of wine.
• **Instacart (multi-table):** The Instacart dataset\(^1\) contains information about three million Instacart orders in multiple tables. We are primarily interested in using order information stored in an orders table along with product information stored in a products table to predict whether a new user will buy a product or not.

• **MNIST (image):** The MNIST dataset [9] consists of 70,000 examples of handwritten digits, each mapped to the true value of the digit drawn. The goal of the dataset is to identify the number present on new images of handwritten digits.

• **20Newsgroups (text):** The 20Newsgroup dataset [8] is a collection of approximately 20,000 newsgroup documents, each mapped to one of 20 newsgroups. The goal of the dataset is to identify which newsgroup a new document belongs to based on the contents of the document.

• **UrbanSounds (audio):** The UrbanSounds dataset [13] is a collection of 1302 sound recordings each mapped to a particular urban sound label identifying the sound. For example, a sound could be identified as a "car horn." There are ten such classes of labels. The goal of the dataset is to classify new sound recordings as one of the ten classes.

We note that we select datasets that are generally conducive to classification problems. To address any concerns about what types of problems we can solve with MLBlocks, we test one one more dataset that is serves a regression problem, rounding our total number of datasets tested to six.

• **Boston housing (single-table):** The Boston housing dataset\(^2\) contains a single table of 506 instances of various information about Boston suburbs. The table consists of 12 attributes, and each instance is mapped to a median housing price, which is a continuous value. The goal of the dataset is to predict median housing prices for new instances of towns, as described by attributes.

\(^1\)https://www.instacart.com/datasets/grocery-shopping-2017
\(^2\)http://archive.ics.uci.edu/ml/datasets/Housing
In testing how well MLBlocks performs on various data modalities, it goes without saying that one must compose pipelines that serve each of these data modalities via the MLBlocks framework. Concretely, we test eight different pipelines on our six different datasets, each representing a different data type.

We describe these pipelines as follows.

- **Random forest classifier (Wine)**: A simple pipeline consisting of just a random forest classifier.

- **Random forest regressor (Boston housing)**: A simple pipeline consisting of just a random forest regressor.

- **Deep Feature Synthesis (Instacart)**: A pipeline consisting of Deep Feature Synthesis (DFS) [7] followed by a random forest classifier. DFS is a primi-

<table>
<thead>
<tr>
<th>Pipeline</th>
<th>Dataset</th>
<th>Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random Forest Classifier</td>
<td>Wine</td>
<td>Random Forest Classifier</td>
</tr>
<tr>
<td>Random Forest Regressor</td>
<td>Boston housing</td>
<td>Random Forest Regressor</td>
</tr>
<tr>
<td>Deep Feature Synthesis</td>
<td>Instacart</td>
<td>DFS Random Forest Classifier</td>
</tr>
<tr>
<td>Traditional Image</td>
<td>MNIST</td>
<td>HOG Random Forest Classifier</td>
</tr>
<tr>
<td>Traditional Text</td>
<td>20Newsgroups</td>
<td>Count Vectorizer Array Conversion TFIDF Transformer Multinomial Naive Bayes</td>
</tr>
<tr>
<td>Neural Text</td>
<td>20Newsgroups</td>
<td>Tokenizer Sequence Padder LSTM Class Probabilities Converter</td>
</tr>
<tr>
<td>Audio</td>
<td>UrbanSounds</td>
<td>Audio Featurization Random Forest Classifier</td>
</tr>
</tbody>
</table>

Table 6.1: MLBlocks experiment pipelines
tive developed by Featurelabs\(^3\) which automatically derives a large number of features across multiple relational datasets.

- **Traditional Image (MNIST):** A pipeline consisting of a histogram of ordered gradients (HOG) featurization step followed by a random forest classifier. This represents a traditional approach to solving image problems.

- **Traditional Text (20Newsgroups):** A pipeline consisting of a count vectorizer followed by an array conversion step followed by a TFIDF transformer followed by a Multinomial Naive Bayes classifier. This represents a traditional approach to solving text problems.

- **Neural Text (20Newsgroups):** A pipeline consisting of a tokenizer followed by a sequence padder followed by an LSTM. The LSTM consists of an embedding layer followed by a dropout layer followed by a convolution and maxpooling layer. The output of this is directed into an LSTM layer, which is then fed to a dense layer to produce class probabilities. This output is then fed into a step to convert class probabilities to labels.

- **Audio (UrbanSounds):** A pipeline consisting of an audio featurization step followed by a principal component analysis (PCA) step followed by a random forest classifier. This audio featurization step is based loosely on the featurization proposed in SenseML [2]. It extracts features such as energy\(^4\), spectral entropy, and spectral flux.

We summarize this pipeline structure in table 6.1. All blocks are present in the MLBlocks library, as listed in table B.1. All pipeline definitions and experiment code can be found in the MLBlocks-Demos repository\(^5\).

---

\(^3\)https://www.featurelabs.com/

\(^4\)https://en.wikipedia.org/wiki/Sound_energy

\(^5\)https://github.com/HDI-Project/MLBlocks-Demos
<table>
<thead>
<tr>
<th>Pipeline</th>
<th>Dataset</th>
<th>Average F1 Score</th>
<th>Best F1 Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random Forest Classifier</td>
<td>Wine</td>
<td>0.965</td>
<td>1.000</td>
</tr>
<tr>
<td>Random Forest Regressor6</td>
<td>Boston housing</td>
<td>0.829</td>
<td>0.926</td>
</tr>
<tr>
<td>Deep Feature Synthesis</td>
<td>Instacart</td>
<td>0.840</td>
<td>0.904</td>
</tr>
<tr>
<td>Traditional Image</td>
<td>MNIST</td>
<td>0.890</td>
<td>0.922</td>
</tr>
<tr>
<td>Traditional Text</td>
<td>20Newsgroups</td>
<td>0.701</td>
<td>0.884</td>
</tr>
<tr>
<td>Neural Text</td>
<td>20Newsgroups</td>
<td>0.111</td>
<td>0.166</td>
</tr>
<tr>
<td>Audio</td>
<td>UrbanSounds</td>
<td>0.297</td>
<td>0.335</td>
</tr>
</tbody>
</table>

Table 6.2: MLBlocks experiment pipelines

6.2 Experiment procedure

Once we have obtained our datasets and pipelines, we now proceed to our experiment. We briefly describe this procedure as follows.

As a first step, we split a dataset into training and evaluation sets, with an approximate 80-20 train-evaluation split. Once we have done this, we then fit the appropriate pipeline to our training set and training labels. We then make predictions on the evaluation set and compare them to the evaluation labels.

We run this experiment for each of our datasets for 100 iterations each, recording the F1 score of each pipeline on the dataset for each run.

6.3 Results

We present the results of our experiments in table 6.2. As we can observe, on the majority of our pipelines and datasets, we achieve reasonably positive results. Particularly, we can see that for all pipelines except the neural text and audio pipelines, even the average F1 score is reasonable. This shows us that our choice of blocks was appropriate for the problem type.

More importantly, we see that in all cases the best F1 score is significantly greater than the average F1 score. While this may seem like an obvious result, this gives
us confidence that our pipelines can be tuned to improve performance. As we can recall, each run of our iteration is only different in terms of the initialization of hyperparameters, which is done randomly. Thus, we can interpret the average F1 score as the performance of untuned pipelines; on average, our pipelines are randomly initialized and untuned.

On the other hand, running our pipelines for 100 iterations with random hyperparameter initialization corresponds to random search over our hyperparameters. After enough iterations, we can expect that our best performing hyperparameter combination represents a tuned pipeline, and the best F1 score is representative of the results a real-world application of our pipelines would obtain on the data. Because our simulated random hyperparameter search tuning mechanism does give us noticeable performance gains, we conclude that pipelines created with MLBlocks can be tuned.

Regardless, the most important result from these experiments is that we show that we can simply compose various pipelines with various blocks from different third-party libraries with MLBlocks. We can then run these pipelines and obtain results on real-world datasets of a multitude of data modalities. In doing this, we see that MLBlocks can serve as an end-to-end pipeline creation framework on real-world problems.
Chapter 7

Application

In this chapter, we illustrate the use of \texttt{MLBlocks} in the wild by describing its application in the \texttt{DeepMining}\textsuperscript{1} system. In doing so, we demonstrate the ease of integration of an already existing system with \texttt{MLBlocks} and validate \texttt{MLBlocks} as a tool to be used in the real world.

We primarily choose to integrate with the \texttt{DeepMining} system because \texttt{MLBlocks} provides much of the facility it requires. As we shall see, \texttt{MLBlocks} simplifies much of the \texttt{DeepMining} application logic by allowing easy specification of pipelines and exposition of hyperparameters over an entire pipeline.

7.1 DeepMining overview

Developed by the Data to AI lab\textsuperscript{2} at MIT, \texttt{DeepMining} provides end-to-end AutoML capabilities, producing tuned pipelines from data science problems. We note that the goal of the \texttt{DeepMining} system extends the scope of \texttt{MLBlocks}. While \texttt{MLBlocks} also aims to produce pipelines, \texttt{DeepMining} is given access to problem metadata such as how a user wants a pipeline evaluated, and provides guarantees that the produced pipeline is tuned. In contrast, \texttt{MLBlocks} is only given actual data points and labels and only produces a pipeline that \textit{can} be tuned; \texttt{MLBlocks} provides no guarantees

\footnote{\url{https://github.com/HDI-Project/DeepMining}}

\footnote{\url{https://dai.lids.mit.edu/}}
that the produced pipeline the user employs on his data is tuned.

## 7.2 DeepMining system architecture

At a high level, the DeepMining architecture consists of five core modules working in conjunction. Three of these modules are native to DeepMining. They provide functionality to score pipelines, search over pipeline and hyperparameter spaces, and parallelize the entire computation process. We refer to these modules as Search, Score, and Compute respectively.

Additionally, DeepMining depends on the Bayesian Tuning and Bandits (BTB) library to perform hyperparameter tuning. Finally, DeepMining requires a module for specifying and modelling pipelines, which our MLBlocks library provides.

At a high level, a developer would use DeepMining as follows. This workflow is illustrated in figure 7-1. The numbers in the figure correspond to the steps of the workflow as described below.

The developer begins by initializing the Search module with a dataset, problem type, and scoring function. This module ties largely leverages and ties together the rest of the module to search over a space of hyperparameters and pipelines.

1. The developer specifies a list of candidate MLPipeline objects to consider to the Search module.
2. The **Search** module uses **BTB** to propose various hyperparameter configurations for the candidate **MLPipelines**. **BTB** is called from within the **Search** module.

3. The **Search** module assigns the scores to each potential pipeline and hyperparameter configuration via the **Score** module. These scores are passed to **BTB** to train tuners and selectors and propose new hyperparameter configurations.

4. The **Score** module leverages the **Compute** class to evaluate multiple pipeline and hyperparameter configurations in parallel.

5. The **Search** module also parallelizes computation with the **Compute** class by evaluating pipelines on independent bootstraps or subsamples of the input dataset.

Once the **Search** module has evaluated a sufficient number of pipeline and hyperparameter configurations, it can then output the best performing configuration as a tunable pipeline. The developer can now use this pipeline to perform real predictions on his data.

In integrating **MLBlocks** with **DeepMining** as the pipeline specification module, we primarily need to integrate with **BTB** and the **Score** module. We describe this process below.

### 7.3 Bayesian Tuning and Bandits (BTB)

One primary goal of the **MLBlocks** platform is to allow easy integration with third-party tuners. In the **DeepMining** system, we integrate with **BTB**. Particularly, we pass in hyperparameter information and a performance score into **BTB** and obtain suggested hyperparameter values to try next in return. These values are specified as mappings from hyperparameter names to hyperparameter values. We then can use these values to update our pipeline’s hyperparameters, re-score, and iteratively tune our pipeline to optimize the quality of our predictions.
We see that MLBlocks makes this process extremely simple. We can simply take our pipeline, obtain our tunable hyperparameters from our pipeline, package the values as well as various metadata such as hyperparameter ranges into BTB’s hyperparameter abstraction, and use that as input into BTB. Upon receiving values to update to, we can simply use the MLPipeline `update_from_hyperparam_dict` method to update our hyperparameter and use the predict method to obtain predictions with the new hyperparameters. We can then obtain any of a number of performance scores with these predictions and repeat the tuning process until we get our desired quality of results.

We abstract out all of the code necessary to deal with interacting with hyperparameters in a pipeline with our MLPipeline. A developer only needs to implement code for scoring and interacting with BTB.

### 7.4 Score

In addition to updating the hyperparameters of our pipeline, we must also be able to score our pipeline as users desire. With MLBlocks, obtaining a score is as simple as first obtaining predictions by sequentially calling `fit` and `predict` of our MLPipeline. We can then pass in these predictions along with evaluation true labels to obtain our pipeline's score.

We note that by choosing to provide methods to produce predictions rather than producing a certain kind of score in our MLPipeline, we leave the description and implementation of any scoring logic to the user. This integrates seamlessly with DeepMining, as the DeepMining Score module is given autonomy to use user-input scoring functions and apply whatever scoring logic necessary for the user’s particular use case.
Table 7.1: Lines of code to implement pipelines before and after MLBlocks integration

<table>
<thead>
<tr>
<th>Pipeline</th>
<th>Old LOC</th>
<th>New LOC (MLBlocks)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random Forest</td>
<td>22</td>
<td>1</td>
</tr>
<tr>
<td>Traditional Image</td>
<td>87</td>
<td>46</td>
</tr>
<tr>
<td>Traditional Text</td>
<td>59</td>
<td>1</td>
</tr>
<tr>
<td>CNN Image</td>
<td>99</td>
<td>23</td>
</tr>
</tbody>
</table>

7.5 Results

We have so far qualitatively described the ease of integration of MLBlocks with DeepMining. We now justify the utility of this integration from a user’s point of view via several heuristics, both quantitative and observational.

7.5.1 Ease of use

We measure ease of use quantitatively in terms of lines of code, as shown in 7.1. Particularly, we measure the amount of Python code required by a developer to implement a pipeline in DeepMining before and after the MLBlocks integration.

Before the MLBlocks integration, a user would have to write the connection and hyperparameter exposition code of his pipeline from scratch. As we can observe, by leveraging MLBlocks and only specifying JSON files, in many cases, we can simply define our MLPipeline in one line as we described in figure 4-2.

In the case of the Traditional Image pipeline, the user has to write a custom function for the histogram of ordered gradients (HOG) featurization step. In the case of the convolutional neural network (CNN image), the user has to write some code to specify the number of classes in the data as well as the optimizer and loss function to used during training. In both cases, this code must be provided by the user, as no system can know the user’s preferences for these.

Regardless, our MLBlocks abstraction dramatically reduces the number of lines of code required by the end user in specifying pipelines for use with DeepMining; that is, it is easier for a user to provide pipeline inputs to DeepMining due to our MLBlocks
integration. In doing this, it is also worth noting that a user can specify a larger number of candidate pipelines and obtain better results on his data.

7.5.2 Feasibility of pipelines

We also note that by when integrated with MLBlocks, DeepMining can provide users with access to pipelines than it would have otherwise not have. Particularly, DeepMining used the scikit-learn Pipeline abstraction before the MLBlocks integration. It was able to create pipelines, such as the CNN image pipeline listed in table 7.1, from different modules such as keras via special scikit-learn wrappers around these modules.

However, in cases where scikit-learn does not provide a wrapper around a library, we observe that a user cannot use that library with DeepMining. MLBlocks rectifies this by allowing integration with any library, allowing users to create their own pipelines to address their problems with fewer restrictions.
Chapter 8

Future work

8.1 MIT-TA2 integration

We would like to integrate with the MIT-TA2 system in the future. Developed in conjunction by teams at MIT and Featurelabs, this system provides end-to-end AutoML capabilities, producing predictions from data science problems. This system is primarily designed to produce results as a part of the DARPA Data Driven Discovery (D3M) program\footnote{https://www.darpa.mil/program/data-driven-discovery-of-models}, and, as this is the case, is subject to various D3M specifications.

We primarily choose to integrate with MIT-TA2 for two reasons. First, MIT-TA2 also uses Bayesian Tuning and Bandits (BTB) to perform hyperparameter tuning, which we have already integrated with, as noted in section 7.3. This gives us a reasonable amount of confidence that MLBlocks will be easy to integrate, providing benefits to the system as a whole. Secondly, the MIT-TA2 system has achieved results on a large number of datasets. Thus, by integrating with MIT-TA2, we would like to evaluate MLBlocks as a real-world system on these datasets as well.

We provide a brief description of the D3M program and its requirements, and give a high-level overview of the TA2 system, illustrating where exactly MLBlocks would integrate to in the system, below.
8.1.1 DARPA Data Driven Discovery (D3M)

The D3M program is a competition sponsored by DARPA in which various institutions compete to create systems that can produce reasonable predictions on a large variety of problems.

The competition is split into three levels, or technical areas. In Technical Area 1 (TA1), teams focus on contributing new blocks that perform well on certain types of data. In Technical Area 2 (TA2), teams aim to compose the TA1 blocks into end-to-end systems that can produce predictions from data science problem data, specified in a D3M-specific format. In Technical Area 3 (TA3), teams aim to provide intuitive user interfaces to the TA2 backends. The MIT-TA2 system, as its name suggests, competes in TA2.

8.1.2 MIT-TA2 system overview

![MIT-TA2 architecture](image)

Figure 8-1: MIT-TA2 architecture
At a high level, the MIT-TA2 system does several things. First, it receives a D3M file as input, and parses this to determine problem metadata as well as training and test data and labels. It then uses the problem metadata to determine which of several hard-coded pipelines to use. Once the pipeline is decided, the TA2 system then performs several iterations of hyperparameter tuning, performing cross-validation and saving the best-performing pipeline after each iteration.

To integrate with MIT-TA2, we will likely implement the MIT-TA2 pipelines in the MLBlocks framework. This would involve creating JSON blocks for the blocks in these hard-coded pipelines and composing MLPipelines from these. We then can use these in conjunction with BTB and MIT-TA2’s other components to produce predictions, as illustrated in figure 8-1.

8.2 Automatically generated pipelines

The simplest way to provide users with pipelines that perform well on their data is to automatically generate such pipelines and provide them to the user. In this case, no domain knowledge at all is required on the user end. As briefly outlined in section 2.1.1, we do not aim to automatically generate pipelines in MLBlocks.

However, we note that the architecture of MLBlocks makes it possible to add such functionality in the future. We also note that we can leverage JSON’s ability to store structured information to store the information required to automatically compose pipelines. As a concrete example, we may be able to annotate the HOG step as a featurization step suitable for image pipelines. In addition to making our JSON files more descriptive to the user, this information can then be used to propose reasonable pipelines for various datasets. From there, we can use a recommender system to recommend one of our proposed pipelines to the user.
Chapter 9

Conclusion

9.1 Key findings

The MLBlocks framework successfully accomplishes our goal of providing a framework to allow the user to easily specify end-to-end tunable pipelines. Additionally, at the beginning of this project, we outlined two concrete goals in the form of usability and accessibility. Here, we discuss MLBlocks in relation to these goals.

MLBlocks allows users to easily specify an end-to-end tunable pipeline and use it to achieve results on particular datasets. To specify a pipeline in the MLBlocks framework, a user only needs to specify an ordering of blocks. They can then immediately tune the pipeline or simply fit and predict on their data. Particularly, we saw that the DeepMining system was able to easily integrate with MLBlocks, leveraging the MLBlocks API to perform operations to tune hyperparameters and fit and predict on data. MLBlocks dramatically simplified the pipeline component of the system while maintaining results.

MLBlocks makes various data science and machine learning libraries accessible to users by unifying them under one interface. The MLBlocks system provides the capability to easily integrate numerous third party libraries and provide a consistent interface with them. Particularly, we can integrate all of the scikit-learn and keras libraries without writing any new Python code. We only need to specify JSON files containing hyperparameter information among various other metadata.
We further illustrate this accessibility by implementing base pipelines that produce reasonable predictions for various data science problem types. In doing so, we integrate a large number of libraries from different third-party sources easily and show that our unified API can produce results with these integrated libraries to serve a number of purposes.

9.2 Contributions

In this thesis, we:

1. Designed a library for collecting third-party blocks under one umbrella via open source contribution of JSON files.

2. Designed an abstraction for exposing key information and functionality of machine learning pipelines. Particularly, the `MLPipeline` abstraction allows us to easily perform hyperparameter-tuning of end-to-end pipelines and to use these tuned pipelines to fit and predict on data.

3. Implemented a base parser, allowing us to integrate blocks from various popular third-party libraries, including `scikit-learn`, without having to write any integration-specific code. To integrate a block, we only need to provide a JSON file.

4. Introduced and implemented a `keras` JSON format and parser as a proof of concept for the integration of nonstandard or custom libraries into the `MLBlocks` framework. Additionally, we are now able to integrate any `keras` block by simply providing a JSON file.

5. Set up an open-source block contribution framework, with linting, automated testing, and continuous integration.

6. Implemented several pipelines for several different data types and the blocks that they are composed of. This serves to show the utility and robustness of the `MLBlocks` framework in the real world.
7. Integrated MLBlocks with the DeepMining system, illustrating the simplifications and utilities MLBlocks provides in a larger, more ambitious context.

We conclude that the MLBlocks framework allows novice users to easily specify end-to-end tunable pipelines that perform well on their particular data science problems.
Appendix A

Figures

```json
{
    "name": "lstm_text",
    "class": "keras.models.Sequential",
    "fit": "fit",
    "produce": "predict",
    "fixed_hyperparameters": {
        "pad_length": 1500,
        "conv_activation": "relu",
        "dense_units": 1,
        "dense_activation": "linear",
        "optimizer": "keras.optimizers.Adam",
        "loss": "keras.losses.mean_squared_error"
    },
    "tunable_hyperparameters": {
        "num_top_words": {
            "type": "int",
            "range": [1000, 40000]
        },
        "embedding_size": {
            "type": "int",
```
"range": [100, 500],

"dropout_percent": {
    "type": "float",
    "range": [0.1, 0.75]
},

"conv_kernel_dim": {
    "type": "int",
    "range": [3, 10]
},

"pool_size": {
    "type": "int",
    "range": [2, 10]
}

"root_hyperparameters": ["num_top_words", "embedding_size", "dropout_percent", "conv_kernel_dim", "pool_size"],

"conditional_hyperparameters": {},

"layers": [
    {
        "class": "keras.layers.Embedding",
        "parameters": {
            "input_dim": "num_top_words",
            "output_dim": "embedding_size",
            "input_length": "pad_length"
        }
    },
    {
        "class": "keras.layers.Dropout",
        "parameters": {
        }
    }
]
"rate": "dropout_percent"
}
{
 "class": "keras.layers.Conv1D",
 "parameters": {
 "filters": "embedding_size",
 "kernel_size": "conv_kernel_dim",
 "activation": "conv_activation"
 }
},
{
 "class": "keras.layers.MaxPooling1D",
 "parameters": {
 "pool_size": "pool_size"
 }
},
{
 "class": "keras.layers.LSTM",
 "parameters": {
 "units": "embedding_size"
 }
},
{
 "class": "keras.layers.Dense",
 "parameters": {
 "units": "dense_units",
 "activation": "dense_activation"
 }
}
Figure A-1: A sample *keras* block
```python
import math

import numpy as np
from skimage.feature import hog

class HOG(object):
    def __init__(self, num_orientations, num_cell_pixels,
                 num_cells_block,
                 img_dimension_x=None, img_dimension_y=None):
        self.num_orientations = num_orientations
        self.num_cell_pixels = num_cell_pixels
        self.num_cells_block = num_cells_block
        self.img_dimension_x = img_dimension_x
        self.img_dimension_y = img_dimension_y

    def make_hog_features(self, X):
        """Call the transform function of the HOG primitive.
        NOTE: Get a "ValueError: Negative dimensions" with some
        settings of the hyperparameters.
        ""
        if math.sqrt(X.shape[1]).is_integer():
            # We can set dimensions if the image is square
            img_dim = int(math.sqrt(X.shape[1]))
            self.img_dimension_x = img_dim
            self.img_dimension_y = img_dim
```

else:

    if not self.img_dimension_x or not self.img_dimension_y:
        raise Exception("Must specify image dimensions for non-square image")

def make_hog(image):
    image = image.reshape((self.img_dimension_x,
                           self.img_dimension_y))
    features = hog(image,
                    orientations=self.num_orientations,
                    pixels_per_cell=(self.num_cell_pixels,
                                     self.num_cell_pixels),
                    cells_per_block=(self.num_cells_block,
                                     self.num_cells_block),
                    block_norm='L2-Hys',
                    visualise=False)

    return features

return np.apply_along_axis(lambda x: make_hog(x), axis=1, arr=X)

Figure A-2: Histogram of ordered gradients (HOG) Python code
Figure A-3: Histogram of ordered gradients (HOG) JSON block
```python
from sklearn.datasets import load_iris
from sklearn.metrics import f1_score
from sklearn.model_selection import train_test_split

from mlblocks.ml_pipeline.ml_pipeline import MLPipeline

print(""
================================================================================
Testing Random Forest Classifier
================================================================================
"")

iris = load_iris()
X, X_test, y, y_test = train_test_split(
    iris.data, iris.target, train_size=120, test_size=30)

rf_classifier = MLPipeline(["random_forest_classifier"])

# Check that the hyperparameters are correct.
for hyperparam in rf_classifier.get_tunable_hyperparams():
    print(hyperparam)

# Check that the steps are correct.
expected_steps = {'rf_classifier'}
steps = set(rf_classifier.steps_dict.keys())
assert expected_steps == steps

# Check that we can score properly.
print("\nFitting pipeline...")
rf_classifier.fit(X, y)
print("\nFit pipeline.")

print("\nScoring pipeline...")
predicted_y_val = rf_classifier.predict(X_test)
score = f1_score(predicted_y_val, y_test, average='micro')
print("\nf1 micro score: %f" % score)
```

Figure A-4: Sample block test code for a random forest classifier
```python
from scipy.io.wavfile import read

def segment(f, window_size = 2000, percent_overlap = 0.1):
    
    """
    params:
    window_size = size of window in milliseconds, float
    percent_overlap = amount of frame overlap for segments, float
    """
    samp_freq, data = read(f)
    num_channels = data.ndim
    num_samples = data.shape[0]
    start_intervals = []
    segments = []

    window = int(samp_freq * window_size/1000.0)
    offset = int((1-percent_overlap) * window)
    num_segments = num_samples // offset

    for i in range(num_segments):
        x = i * offset
        start_intervals.append(x)
        segment = data[x:x+window]
        segments.append(segment.reshape(len(segment),
                                        num_channels))
    return samp_freq, start_intervals, segments, num_channels

Figure A-5: Sample code to segment audio files
```
```python
graph [
  node [
    id 0
    label "1"
  ]
  node [
    id 1
    label "88160"
  ]
  node [
    id 2
    label "118052"
  ]
  node [
    id 3
    label "161555"
  ]
  edge [
    source 1
    target 2
  ]
  edge [
    source 2
    target 3
  ]
  edge [
    source 3
    target 1
  ]
]
```

Figure A-6: Sample GML graph
# Appendix B

## Tables

<table>
<thead>
<tr>
<th>block</th>
<th>Third-party library</th>
<th>Data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random Forest (RF) Classifier</td>
<td>scikit-learn</td>
<td>multiple</td>
</tr>
<tr>
<td>Random Forest (RF) Regressor</td>
<td>scikit-learn</td>
<td>multiple</td>
</tr>
<tr>
<td>Multinomial Naive Bayes (NB) Classifier</td>
<td>scikit-learn</td>
<td>multiple</td>
</tr>
<tr>
<td>Principal Component Analysis (PCA)</td>
<td>scikit-learn</td>
<td>multiple</td>
</tr>
<tr>
<td>Count Vectorizer</td>
<td>scikit-learn</td>
<td>text</td>
</tr>
<tr>
<td>TFIDF Transformer</td>
<td>scikit-learn</td>
<td>text</td>
</tr>
<tr>
<td>Sequence Padder</td>
<td>keras</td>
<td>multiple</td>
</tr>
<tr>
<td>Tokenizer</td>
<td>keras</td>
<td>text</td>
</tr>
<tr>
<td>Text LSTM</td>
<td>keras</td>
<td>text</td>
</tr>
<tr>
<td>MNIST CNN</td>
<td>keras</td>
<td>image</td>
</tr>
<tr>
<td>Deep Feature Synthesis</td>
<td>dfs</td>
<td>multi-table</td>
</tr>
<tr>
<td>Class probability converter</td>
<td>custom function</td>
<td>multiple</td>
</tr>
<tr>
<td>Histogram of ordered gradients (HOG)</td>
<td>custom function</td>
<td>image</td>
</tr>
<tr>
<td>Audio featurizer</td>
<td>custom function</td>
<td>audio</td>
</tr>
</tbody>
</table>

Table B.1: Blocks in the **MLBlocks** library
Bibliography


