Machine Learning Blocks

by

Bryan Omar Collazo Santiago

B.S., Massachusetts Institute of Technology (2014)

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

Master in Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2015

Author .....
Department of Electrical Engineering and Computer Science
May 22, 2015

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

Accepted by .....
Albert R. Meyer
Chairman, Masters of Engineering Thesis Committee

Machine Learning Blocks

by

Bryan Omar Collazo Santiago

Submitted to the Department of Electrical Engineering and Computer Science
on May 22, 2015, in partial fulfillment of the
requirements for the degree of
Master in Engineering in Electrical Engineering and Computer Science

Abstract

This work presents MLBlocks, a machine learning system that lets data scientists explore the space of modeling techniques in a very easy and efficient manner. We show how the system is very general in the sense that virtually any problem and dataset can be casted to use MLBlocks, and how it supports the exploration of Discriminative Modeling, Generative Modeling and the use of synthetic features to boost performance. MLBlocks is highly parameterizable, and some of its powerful features include the ease of formulating lead and lag experiments for time series data, its simple interface for automation, and its extensibility to additional modeling techniques.

We show how we used MLBlocks to quickly get results for two very different real-world data science problems. In the first, we used time series data from Massive Open Online Courses to cast many lead and lag formulations of predicting student dropout. In the second, we used MLBlocks' Discriminative Modeling functionality to find the best-performing model for predicting the destination of a car given its past trajectories. This later functionality is self-optimizing and will find the best model by exploring a space of 11 classification algorithms with a combination of Multi-Armed Bandit strategies and Gaussian Process optimizations, all in a distributed fashion in the cloud.

Thesis Supervisor: Kalyan Veeramachaneni
Title: Research Scientist

Acknowledgments

I would like to thank many people in helping me complete this work.

First and foremost, I would like to thank my family. Their love and unconditional support has been essential not only for the completion of this work, but for any and all of my accomplishments. Adrián, Cándida, Xavier and Giomar, *los quiero mucho*.

Secondly, this work wouldn't have been possible with the help, guidance and support of Kalyan Veeramachaneni. Not only is he an excellent an advisor, shown by his assertive and knowledgeable guidance, but he is also a caring mentor and friend.

Lastly, I would also like to thank Cara and Arash in helping me improve this work. They were extremely helpful during the process.

Contents

List of Figures

Chapter 1

Introduction

We are currently living in a world that is being flooded by data. Every day huge amounts of data is generated by our many devices and platforms. Just as a simple example, user-generated content on Facebook amounts to about 500 terabytes in a *single* day. To put this in perspective, 1 gigabyte can store around 7 minutes of HD quality video, thus 500 terabytes is as much as 6 years of HD quality video; all generated in *one* day by *one* company.

With so much data, a natural question arises: "How can we use this data in meaningful ways to improve and impact our living?". How can we try to extract meaningful information/insights for analysis, or build complex algorithms to solve societal problems such as fighting fraud, assisting healthcare *via* personalized medicine and/or improving education. However, the sheer amount of data being collected brings about many challenges. For example, just storing or transferring it through a network can pose many obstacles. Handling the volume, given a *processing* module can be another challenge. While these two challenges are being solved by having newer and better infrastructure and software-as-a service paradigms one challenge still remains: the human data interaction bottleneck. Deriving insights from data is still a human driven endeavor. This is exactly the job of a data scientist; explore data, decide what questions to ask, what methods to use, and how to use them to ultimately build predictive models.

To work with data, derive insights, and build these systems, contemporary data

scientists undergo the same high-level steps. No matter if the application is for education or finance, the pipeline from raw data to good predictive models is very similar. This thesis carefully examines a data science pipeline and explores what can be automated and how interfaces can be provided for the automated modules. We call the automated components - MLBlocks. We begin by presenting a typical "Data Science Pipeline" in the next section and highlighting what parts of this pipeline are time consuming and human driven.

1.1 The Data Science Pipeline

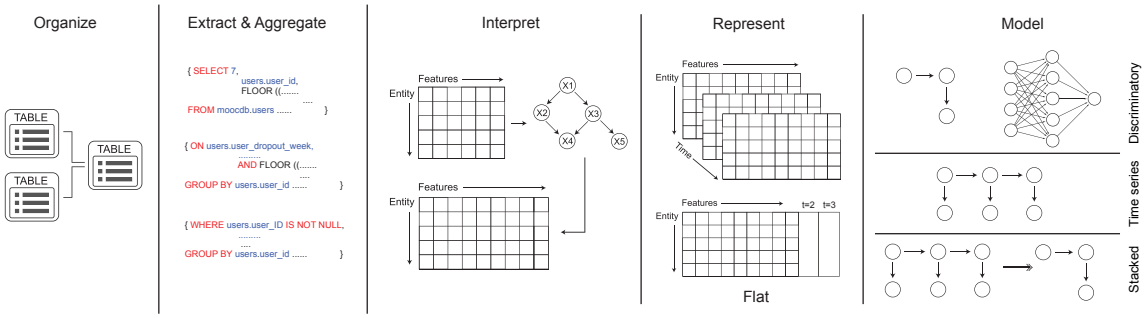

Figure 1-1: The Data Science Pipeline. Also included in Appendix A.

Figure 1-1 shows how we break down the data science pipeline in five steps. These are: "Organize", "Extract & Aggregate", "Interpret", "Represent" and "Model".

In the first step, the data scientist has to collect, extract and aggregate the subsets of the raw data that will be used for learning. Data is stored in many different ways: as images, text files, videos, relational databases, or log files, to name a few. For most applications, the data of interest has to be cleaned in some way, computed from a few sources, or simply integrated across different services. This, as the reader can imagine, is very application-specific.

After one has collected data and completed possible pre-processing steps, the data scientist goes through a process called "feature engineering", which is illustrated by the steps of "Extract and Aggregate" and "Interpret". The outcome of these step will be the Feature Matrix. The rows for this matrix are *samples*, and the

columns correspond to different *features* for each sample. Features can be thought of as "attributes" of the data, while a sample can be thought of as an "instance" or an example of the data. If, for example, the application is to build a system that can intelligently classify the species of a given flower, then samples can be different examples of flowers, and features can be particular attributes such as the number, color, and average length of the petals.

This feature engineering step is crucial for having good predictive systems. If a data scientist decides to use the wrong features, it may make the learning of a good model almost impossible. Imagine in the flower example above, the data scientist had instead computed features like the name of the person that planted the flower, or the time of the day when the sample was taken – it would be very hard for any algorithm to try to predict the correct species of a flower given these attributes.

The "Represent" step transforms the data into the structures and representations that subsequent machine learning approaches require. Finally, in the "Model" step the data scientist can start using standard machine learning algorithms to try to find the best model for the data. These techniques vary a lot, and every technique has its intricacies and special parameters. Trying out different techniques and searching through the space of parameter settings is a tedious process, and can take months of the scientist's valuable time.

Towards Automation

Now, considering this pipeline is very similar regardless of the machine learning problem that is being solved, a natural endeavor is to attempt to automate these steps. Indeed, there has been much work in attempting to automate some of these.

For example, in the "Organize" step, companies like Data Tamr[1] here in Cambridge are using breakthrough technology to help clients merge and organize many data sources into one consistent database. For the second step ("Extract and Aggregate") once humans decide what they would like to extract from the data, they can do so in a very efficient manner by utilizing Hadoop-based systems. However,

[1] http://www.tamr.com/

coming up with the ideas for extraction and aggregation still inherently require the intelligence and reasoning of a human.

Another step where the pipeline is dependent on the human, is the decision of "what high-level machine learning technique to use?". This is often a function of the type of data it is (e.g. *time series* or not), the data scientist's past experiences with similar problems, and the current set of tools available to him/her. In particular, the endeavor of "deciding what high-level machine learning technique to use" is expensive, and poses the typical data scientist conundrum that we present via an example.

Example 1. Data scientist Albert has been hired to make a machine learning system that can predict when a student will dropout of a Massive Open Online Course. He spends the first few weeks assembling the data, deciding what features to use, and extracting them using a Hadoop-based system. He then decides to solve it as a classification problem. It then takes him a few additional weeks to try out a Support Vector Machine (SVM) and tune its parameters. The best performance was achieved with a very specific SVM at around 82% accuracy, which is good, but not satisfactory, since predicting that everyone will drop out gives him 79% accuracy. Wondering how he can do better, he consults another data scientist who suggests that HMMs might be a great fit since it is a time series. This, however, means Albert has to spend possibly an additional week of trying out HMMs with different parameters; all to possibly recognize that there may be more options like: attempting to generate "state distribution features" and doing classification with them or do "clusterwise classification" to increase his SVM score.

To try different methodologies and report back to his manager, Albert faces the following challenges:

He has to piece together different disparate softwares: To do SVM he chooses *scikit* learn in python. For HMM he uses a C++ based library provided by the other data scientist. The library while learns the HMM efficiently does not have an efficient implementation of Viterbi algorithm which he can use to infer state probabilities. So he finds another tool that may allow him to do that. Using

different tools to explore these high-level techniques like discriminative modeling, generative modeling, or the use of synthetic features to boost performance we claim is an unnecessarily expensive way to use the data scientist's time.

For each technique he has to identify parameters and ways to tune them: For each modeling method he uses, he identifies the parameters that can be tuned. He consults fellow data scientists for suggestions and then ultimately resorts to "trying out" different tunable parameters for each, and a work that requires much less intuition than for example feature engineering.

Store intermediate data structures and write transformations: For every tool he downloads he has to write data transformation scripts such that he can pass the data in the input formats required.

These challenges in a data scientists' workflow motivated the work behind this thesis. We pose questions like: "How many high-level techniques [2]are possible for machine learning and predictive modeling?", "Is it feasible to build a generic software system that enables us to develop models from multiple high-level techniques?" , and "Is it possible to map every machine learning problem to this set of high-level techniques?". In particular, in this thesis we will present a software system called MLBlocks, short for Machine Learning Blocks, that aims to help the data scientist explore these different high-level techniques, which we call "*paths or lines*", in an efficient way and that is general enough to suit virtually any machine learning problem.

1.2 What is MLBlocks?

MLBlocks is a software system that the data scientist can use to explore the different high level techniques in the modeling phase. *Blocks* in MLBlocks refer to reusable software modules. For example, "discretize" is software module that is used to bin a continuous variable. It can be used to discretize variables before training an discrete

[2]By high-level technique we mean an end-to-end machine learning approach. For example classification is a high-level technique, HMM is another, clusteriwise classification is another.

HMM, or training a Bayesian network of variables. Multiple *blocks* are put together to construct an end-to-end high level technique. Some of what we refer to as high level techniques are the following classes of modeling techniques:

- **Discriminative Modeling**: Many models fall under this category; SVMs, Logistic Regression, Perceptron, Passive Aggressive are examples. In a probabilistic setting, they attempt to model the conditional probability $P(Y|X)$, where Y is the "label" (value you want to predict) and X is the sample under question.

- **Generative Modeling**: These models attempt to model the *joint* distribution of $P(X, Y)$. This has slightly different semantic implications and interpretations, but they are called "generative" because models of this form can generate not only the prediction values but the sample and features as well. Examples include Hidden Markov Models, Gaussian Mixture Models and Naive Bayes.

- **Clustering**: An unsupervised-learning technique that is not used directly to predict a value, but to find internal structure and patterns of a data. It groups "similar" data points together, where this notion of similarity can be parametrized.

- **Clusterwise Modeling**: A combination of the previous three techniques. The idea is to first cluster the data, and then learn a different discriminative or generative model for *each* cluster of the data. This often shows better performance, as in many settings the learning of different *types* of data should be done differently.

- **Abstract Feature Modeling**: Another common technique is to create additional features that may have little semantic meaning but strong predictive power. These we call abstract features and are usually derived from a machine learning model learnt over the data. The idea is to augment the data with these abstract features and do Discriminative or Generative Modeling in this new feature space.

These broad categories we claim include virtually every way of finding a good predictive model for a given feature matrix. Exploring these techniques is costly for a data scientist, and the principal purpose of MLBlocks is to make this exploration easy and agnostic of the application context.

1.3 Contributions

With MLBlocks we achieve the following important contributions:

- **A foundational representation of data**: We recognize that there is a foundational representation for any data problem and we present this representation and its variations in Chapter 3. We then proceed to present how data from two seemingly disparate problems can be mapped into this representation. This is shown in Chapter 5 and Chapter 6.

- **Automated pathways**: We achieve fully automated pathways. In our current software we have four different modeling blocks. These blocks are detailed in Chapter 3.

- **In-memory model exploration**: An important aspect of MLBlocks is that given the data in the representation we define, the user does not have to store intermediate data format while he/she explores multiple high level techniques.

- **Exposing parameters that can be tuned**: The system also exposes the parameters for each *block* and allows the data scientist to choose their value; enabling them to use a parameter tuning approach.

- **Extensible interface**: MLBlocks introduces a very extensible interface that permits new users develop their own high-level techniques (and *blocks*), which he/she can then reuse for additional problems.

1.4 Thesis Outline

We will present this work as follows. First we discuss related work in Chapter 2. Then, Chapters 3 and 4 present the complete software, while Chapters 5 and 6 will provide two real-world examples of how we used MLBlocks to build strong models after exploring the different blocks with different prediction problems.

Chapter 5 deals with the problem of predicting dropout in Massive Open Online Courses (MOOCs) and in particular shows the power MLBlocks has when solving different lead and lag problems for time series data.

Chapter 6 presents the problem of predicting the destination of a car, given its past trajectories. This will further the presentation of MLBlocks as a very general purpose machine learning system for data modeling.

Finally, Chapter 7 concludes with a summary of the results and contribution of this thesis.

Chapter 2

Related Works

Machine learning tools are now more prevalent than ever. As companies try to ease the challenges of managing and learning from "Big Data" for both business-minded personell and technically-versed data scientists, there is now a wide variety and slightly overwhelming range of products and services to choose from.

MLBlocks sits at a very particular place in this spectrum, as it is meant to be used by the data scientist in his search for the best way to model the data. This differs from the purposes and uses of many of the machine learning systems out there, as we discuss and present in this chapter.

2.1 The Data Scientist Toolkit

Although there are many products and tools for data scientists to use, many are very specialized. Examples of these are the Torch library in LuaJIT or CUDA-Convnet for working with fast, deep, and convolutional neural networks, or Tableau or D3.js for visualizations of data. But few tools share the general-purpose type of exploration, learning and experimentation involved in MLBlocks.

2.1.1 Weka and Other Open Source Tools

One product that provides this general, tunable type of learning is Weka[1]. Weka is short for Waikato Environment for Knowledge Analysis, and the name comes from an endemic bird in New Zealand and the institution where Weka was built, University of Waikato in Hamilton, New Zealand.

Weka is presented as a toolkit and workbench for data scientists to do "Data Mining." Its powerful features include its portability (it was written in Java), its visual Graphical User Interface (GUI), and its suite of modeling techniques.

Other Open Source projects in this line of work include ELKI, KNIME, MOA, and most notably Rapid Miner. ELKI focuses on unsupervised learning via clustering, KNIME focuses on the visualization aspects of exploring data, and MOA is the most similar to Weka (and was also developed in the University of Waikato). Rapid Miner is arguably the most popularly successful of these workbenches, being featured in news site like TechCrunch[2] and being adopted by industry leaders including eBay, Cisco, Paypal and Deloitte.

2.1.2 Rapid Miner

The capacities of Rapid Miner[3] lie in its amazing Graphical User Interface which can visualize data in different ways. It is geared for the non-programmer and thus has different level of abstraction than MLBlocks. It is also possible to model data in Rapid Miner using a suite of algorithms such as Naive Bayes, Nearest Neighbors, and others, by creating what they call a "Process" in a drag-and-drop direct manipulation UI. This allows Rapid Miner to quickly set up a specific pipeline if desired, but as we'll see, these features will be comparable to the exploration of model parameters and discovery of predictive aspects of the data supported in MLBlocks.

[1] http://www.cs.waikato.ac.nz/ml/weka/

[2] http://techcrunch.com/2013/11/04/german-predictive-analytics-startup-rapid-i-rebrands-as-rapidminer-takes-5m-from-open-ocean-earlybird-to-tackle-the-u-s-market/

[3] https://rapidminer.com/

2.1.3 BigML

A final product in this category is BigML[4]. As with Rapid Miner, its also designed for a higher level of abstraction of machine learning. BigML has a very nice way of visualizing decision tree models from the data, and several histograms and other graphs describing aspects of the data, like frequency of values in the features.

However, unlike in MLBlocks, the user must input data by uploading a file, which already constrains possibilities for questions and predictions. For example, BigML only allows you to choose a single column as a predictive feature, and if the data is a time series, it will be an expensive process to ask temporal questions like "how well can I predict such value in the feature, given only certain amounts of data from the past?". Also, the choice of modeling technique is very centered around Decision Tree models only, and fails to thoroughly explore the bigger space of modeling techniques and hyperparameters.

These products showcase the landscape of available options for data scientists and business analysts alike, and begin to hint how MLBlocks will be different from and comparable to these. In particular, across all the products presented above, the techniques for modeling data mostly fall into the categories of Classification, Regression and/or Clustering; which are all subsets of the blocks in our software.

2.2 Programming Languages and Libraries for ML

Of the products presented above, Weka seems to be the one most geared towards data scientists. Scientists in the fields of Machine Learning and Data Analytics have to use more specialized tools to do the fine-grained tuning and precise exploration required by their work. We present the most common ones below.

[4]https://bigml.com/

2.2.1 Python: Scikit-Learn and Others

Python is probably the most flexible language that has recently become one of the biggest players in scientific programming tools for data scientists. With the maturity of projects like Scikit-Learn, there has been a wider adoption of Python as a main language for fast prototyping and exploratory programming in data science.

Scientific computing packages like Numpy, SciPy and Pandas have made manipulation of data structures really efficient by leveraging the power of Python's API to C/C++ code. Similarly, libraries like Scikit-Learn have been designed to build on top of these and to support many machine learning algorithms to work "out of the box" in a very easy programming manner. Having these be Open-Source software helps considerably for their wide adoption and furthers their maturity and documentation as the strong communities behind them keep developing them.

There are too many Python packages to list, but we would like to make particular note of Orange[5], one of the few a data mining software suites with a GUI. This makes it look like the RapidMiner version for data scientists, as its focused on visualizing and exploring properties of the data, and not so much on different algorithms and modeling techniques. (A subtlety to note is that Orange has its own data structures and is not directly integrated with Numpy and/or Pandas.)

In general, most of the software packages provide raw tools for the data scientist to program his/her own experiments. We will see how MLBlocks leverages these raw tools to provide a higher level of abstraction for the data scientist.

2.2.2 R and Matlab

R is known to be the most powerful and specialized statistical programming language, and supports a vast library of statistics and machine learning algorithms. It provides great performance, but its different syntax provides a slight barrier to entry for programmers, as opposed to the more ubiquitous Python. Matlab also provides a specialized language and platform, and can provide very good performance. More-

[5] http://orange.biolab.si/

over, these two languages have inspired development in Python itself, as seen in the matplotlib library which was influenced by Matlab's graphical and plotting library, as well as the Pandas library, which was inspired by R's Data Frame structure.

Other languages that are also being designed and developed for the sole purpose of scientific and technical computing – for example the Julia[6] language project, developed here at MIT.

2.3 Generation 2.0 Tools

Aside from the exploration work that data scientists do, there is also the arguably equally challenging experience of making production-ready systems that can provide the power of machine learning to consumer-based products and services.

2.3.1 Google Prediction API

Google has become a leading supplier of production-ready services and application programming interfaces (APIs) that empower developers with their state-of-the-art infrastructure and software. In this wide range of services, machine learning is not missing.

Launched in September 2010, the Google Prediction API[7] is possibly the simplest way to conceive of Machine Learning as a Service (MLaaS). It works by exposing a RESTful interface that, after the developer has uploaded his data using Google Cloud Storage, he/she can call via the client libraries provided or through raw HTTP queries to get predictions. This in a sense, is very similar to using Numpy and Scikit to load a CSV file, train an SVM and have a webserver expose the SVM's prediction.

Its simplicity of use is key, but in the same vein it is not meant for exploratory data analysis and finding intrinsic valuable properties in the data for prediction. The complete modeling techniques are hidden from the user and they only support simple Classifications and Regressions. Flexibility in the type of prediction problems is also

[6] http://julialang.org/
[7] https://cloud.google.com/prediction/

limited by the form of the flat-file data uploaded and the user has virtually no control around the modeling techniques.

2.3.2 Amazon Machine Learning

As recently as April 9, 2015, another of the internet giants announced its MLaaS solution. Amazon's offering works in a very similar way to its Google counterpart. The user must also use an Amazon service (S3, Redshift or RDS) to upload a single CSV file with the data, and the label of the data must already be defined. Furthermore you can also only use simple Classification and Regression techniques with very little customization.

There is also functionality to set up a web service to make predictions in real time, but you would have to use one of Amazon's client services as it is not exposed thru a RESTful interface like Google's is.

Again, this service can be of use if the user has very little experience with machine learning and wants to set up a very simple machine learning service. But it is very hard to leverage any useful information or structure in the data that you might know prior to the construction of the model. It is also worth noting that the performance of this predictive system will rely heavily in the feature engineering done by the user and doing this right requires the maturity and intuition of a data scientist – the exact type of user that would benefit from a more flexible and tunable modeling strategy.

2.3.3 Azure Machine Learning

Microsoft entered the MLaaS space earlier this year with their new cloud-based predictive analytics service, Azure Machine Learning[8].

Azure however, does provide a greater flexibility in terms of custom implementation of algorithms. With support for custom algorithms implemented in R and Python by the user, as well as recent integration with Hadoop (as facilitated by

[8]http://azure.microsoft.com/en-us/services/machine-learning/

Microsoft's acquisition of Revolution R[9]), Azure is a much more flexible player in providing the MLaaS market space. Furthermore, like Rapid Miner and BigML, it integrates the ability to build your own pipeline, called "Experiments" in Azure, with a nice direct-manipulation centered GUI.

Moreover, it also provides simple parameterizations for the type of learning being done at each stage of your experiment, which is something not available from either Amazon or Google.

Although all of these products do a great job of making machine learning available in a production setting, and are highly scalable with simple approaches, they do not provide the user with the flexibility necessary to discover insights about their data, search efficiently through the space of modeling techniques, or formulate many types of prediction problems. These last few features are exactly the ones that MLBlocks focuses on providing. Although not meant for a production environment like these, it provides the tools for a data scientist to explore the data and find high-performing models across machine learning techniques, all in a highly tunable and automatable way.

[9]http://blogs.microsoft.com/blog/2015/01/23/microsoft-acquire-revolution-analytics-help-customers-find-big-data-value-advanced-statistical-analysis/

Chapter 3

Machine Learning Blocks Overview

In this chapter, we introduce MLBlocks at a high level. We first explain why we call our software *general* and present the main diagram that illustrates the software. Then, we dive into the specifics of how the different experiments that can be executed with MLBlocks work.

3.1 Why is MLBlocks general?

One of the biggest claims in our presentation is the generality of the MLBlocks software, in terms of both its suitability for virtually any data science problem, and its general support for modeling techniques.

The first claim is best borne out by from the presentation of a generalization of the standard Design Matrix. This generalization will also encompass time series data nicely – we call it the "Foundational Matrix," and we'll present it in section 3.3.4. The format will result in minimal change in the data scientist's current workflow. This is because he/she can still produce the standard Design Matrix if desired (with no change in workflow), or simply rethink the output format of his/her feature engineering process to leverage the complete power of MLBlocks.

The second claim, that MLBlocks supports virtually any modeling technique, follows from the extensibility of the MLBlocks software. This initial version of MLBlocks that we present only implemented four high-level techniques; Discriminative Model-

ing, Generative Modeling and two types of Synthetic Feature Modeling. However, MLBlocks provides a simple interface for developers to implement new high-level techniques, that may for example include Clustering, Clusterwise Classification or any other conceivable technique.

3.2 The MLBlocks Diagram

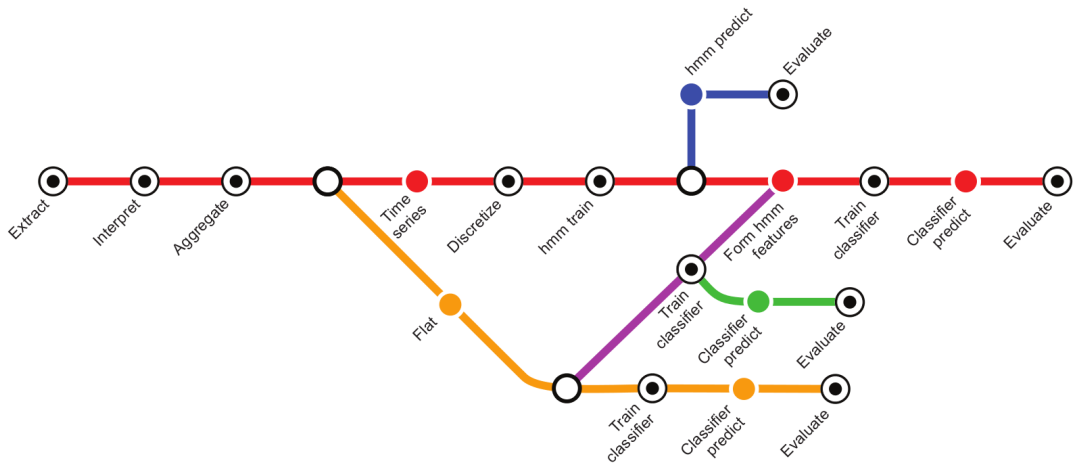

Figure 3-1: MLBlocks Diagram

As suggested by the diagram in Figure 3-1, we will take a look at the MLBlocks Software as a directed graph, where the different paths that one can take from the **Extract** node on the far left to the different **Evaluate** nodes on the right yield the different high-level techniques a data scientist can experiment with. Each node in the picture denotes a computational or transformational step incurred in that path's experiment, we call these paths "Lines" inspired by our very own public transportation system here in Boston, the MBTA.

Each **Evaluate** node concludes a high-level technique, and maps with one of the four techniques in MLBlocks.

Parameterizable Nodes

MLBlocks's is intended to be a very flexible piece of software that lets the user change many parameters within the many aspects of the experiments. In Figure 3-1, the nodes with black dots distinguish the steps that can be parameterized.

How this Chapter is Organized

This rest of this chapter will first discuss the common section for all "Lines" (which comprises the **Extract**, **Interpret**, and **Aggregate** nodes), and then goes to discuss each of the "Lines" one after the other. We then discuss each "Line" by presenting its nodes in a sequential fashion.

3.3 Extract, Interpret and Aggregate

MLBlocks' purpose is to help the data scientist explore the different modeling techniques available. Thus, the software itself is purposely decoupled from the feature engineering process as much as possible. However, any good machine learning pipeline, requires a thoughtful feature engineering procedure, and that's exactly what the **Extract**, **Interpret**, **Aggregate** nodes intend to capture.

3.3.1 Extract

Data storage forms used in machine learning vary greatly across application. Actually, most data storage systems were never designed with machine learning in mind. They were designed for other purposes, such as the performance of online queries, redundancy, availability, or simply readability or debugability.

Thus, for virtually every machine learning application, the very first step is to *extract* data from its raw format into a new repository and/or structure that is suitable for machine learning. This step is also reduces the possibly high complexity in the raw data, since many times you don't need *all* the raw data, but only a subset of it.

3.3.2 Interpret

After one has extracted the subset of data that is of interest for the predictive application, the data scientist has to undergo a process of thinking and reasoning about the exact formulation and representations of the data. This process, also known as "Feature Engineering", requires much intuition and knowledge about the specific domain of the application. Many times indeed, this is a joint effort between the data scientist and a domain expert on the application context.

3.3.3 Aggregate

After deciding on the interesting features, the data scientist has to then undergo the exact processing of the extracted data to new formats that encode the above feature engineering process and that can also be used for training models.

These previous two steps are the ones we propose to change slightly, in order to make the modeling phase much more general. We suggest the result of the interpret and aggregate phases be in a new format we call The Foundational Matrix, instead of the usual 2D Design Matrix.

3.3.4 The Foundational Matrix

To be able to run multiple and different types of experiments on the same data we needed to design a reusable, yet general, data format that could be the input format for MLBlocks. This data format needed to also be able to reflect previous feature engineering work done by the data scientist. We now present a very flexible data format for which we claim virtually any form of data, be it linked data, time series, text or image data, can map nicely to and still be able to encode the result of the feature engineering process.

This form of data is represented as a 3-dimensional Matrix we are calling "The Foundational Matrix," where the three axes of information are **Entities**, **Features** and **Data Slices**. The diagram in Figure 3-2 shows this data form, but with Time instead of Data Slices.

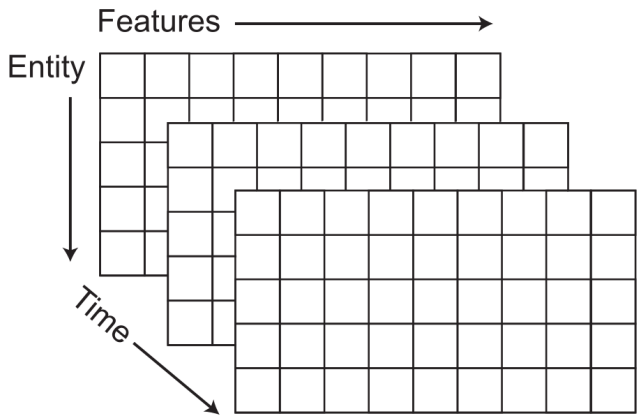

Figure 3-2: The Foundational Matrix

The **Entities** are the central notions or concepts behind the data. In many cases, these will follow from the application and data. Examples of Entities include Products in a products database, Students in a MOOC database, or Trips in a car database. A useful heuristic to detect the Entity in a dataset is to question, "with what axis should one separate the data when feeding it into an HMM as multiple observation sequences?". This generally yields a good Entity.

The second axis, labeled **Features**, should be more familiar to the data scientist and almost self-explanatory. Along this axis we see the feature engineering encoded per "sample" in the data.

Finally, and perhaps most conceptually challenging, is the axis that we are calling **Data Slices**. Initially, we designed it to be a time axis, but since the recording of data does not always happen in an equally-spaced time axis, we introduced the more general notion of a Data Slice, where the distance between 2D Entities x Features matrices need not be the same. A time axis in a time series is still a special case, but this schematic generalizes better to other settings where the intervals of time are not the same length.

3.3.5 An Example: Stopout Prediction in MOOCs

An example problem that we translated into this data form is the problem discussed in Colin Taylor's thesis of Stopout Prediction in a Massive Open Online Courses[**?**]. The goal is to predict if and when a student will "stopout" (the online equivalent of a dropout of a course), given data about his past performance. Here, an Entity is a Student, the Slices are the different weeks of data, and the Features, are quantities like average length of forum posts, total amount of time spent with resources, number of submissions, and many other recorded metrics recorded that talk about the student's performance on a given week.

3.3.6 Another Example: Standard 2D Design Matrix

There is one general mapping that already shows that a vast majority of datasets out there very easily and seamlessly work with MLBlocks. In particular, many datasets, such as the ones found in the famous UCI Repository and Kaggle competitions, come in what is also the most common output format of a feature engineering procedure, which is the n x d (samples x features) "Design Matrix." An example is shown in Figure 3-3 and is commonly denoted as X in the literature.

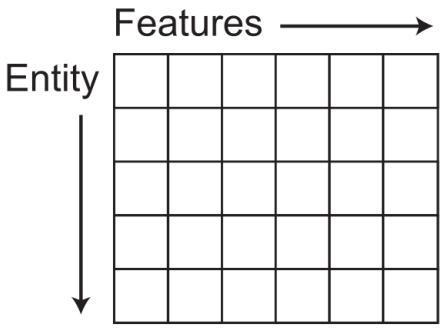

Figure 3-3: Standard Design Matrix in Machine Learning, where we've renamed the Samples axis with Entities.

We claim that this design matrix is simply a special case of the MLBlocks Foundational Matrix, where there is only one slice or one entity in the data. Hence, it can

be thought as a time series data of only one timestamp, or a time series data of only one entity. This suggests that the Entities and Data Slices axes are interchangeable in terms of the implementation, which we will show later is effectively the case.

Given this last example, MLBlocks Data Format already presents itself as a more general way of thinking about the data, and suggests virtually any problem in machine learning can be phrased this way.

MLBlock software starting node

We are now ready to present the four "Lines" in MLBlocks and how they work. It should now be clear how all the lines effectively start from the blank node to the right of the **Aggregate** node.

3.4 Orange Line: Discriminative Modeling

We first discuss the Orange line, which resembles Discriminative Modeling. Classification and Regression fall under this category and are arguably the most common Supervised Learning techniques in machine learning.

The Orange Line has four *blocks*: **Flatten**, **Train Classifier**, **Classifier Predict** and **Evaluate**. Please note how the last three nodes in this Line are actually the same three nodes as in the Green Line and Red Line. Although we will present them only in this Line, they work in the exact same way for the other two Lines.

3.4.1 Flatten Node: Lead and Lag Experiments

Part of the power of the MLBlocks Foundational Matrix is that it gives the user the ability to undertake Lead-Lag Experiments with time series data. These experiments are the equivalent of asking the questions: "How well can I predict j time (data) slices in the future, given i time slices from the past?". For example, suppose you want to train a model that will let a doctor know whether a patient will have a sudden increase in blood pressure in the next 10 minutes given data from the last 30 minutes. Similarly, suppose you would like to know how well you can predict the price of a

Figure 3-4: Orange Line: Discriminative Analysis

stock some j units of time from now, given data only from the last i time units. It would be nice to experiment with what values of j (lead) and i (lag) one can use to predict these problems well.

One way to carry out this type of training and experiment is to "flatten" the data – which is exactly what the **Flatten** node in the Orange Line does. We will produce a 2D Design Matrix from this node which classifiers can use for standard training, but which will also encode the question of how well we can make predictions given some lead and lag. In later sections, we'll also see how to also run such experiments with Hidden Markov Models (HMMs).

Formally, the transformation in this node goes as follows. Take a timeseries dataset $S = \{s_0, s_1, ..., s_{n-1} | s_t \in \mathbb{R}^d, t \in [0, n-1]\}$ with n slices in $\mathbb{R}^d$ as shown in Figure 3-5. Every slice s_t (also called an observation) has a label and d features. Then, given a lead j and a lag i, we will create a new 2d matrix with data samples F, where each new sample f_t is created by concatenating all the features of the slices in S from $t-i$ to t into a single vector, and setting the label of this vector to be the label of the slice at time $t+j$.

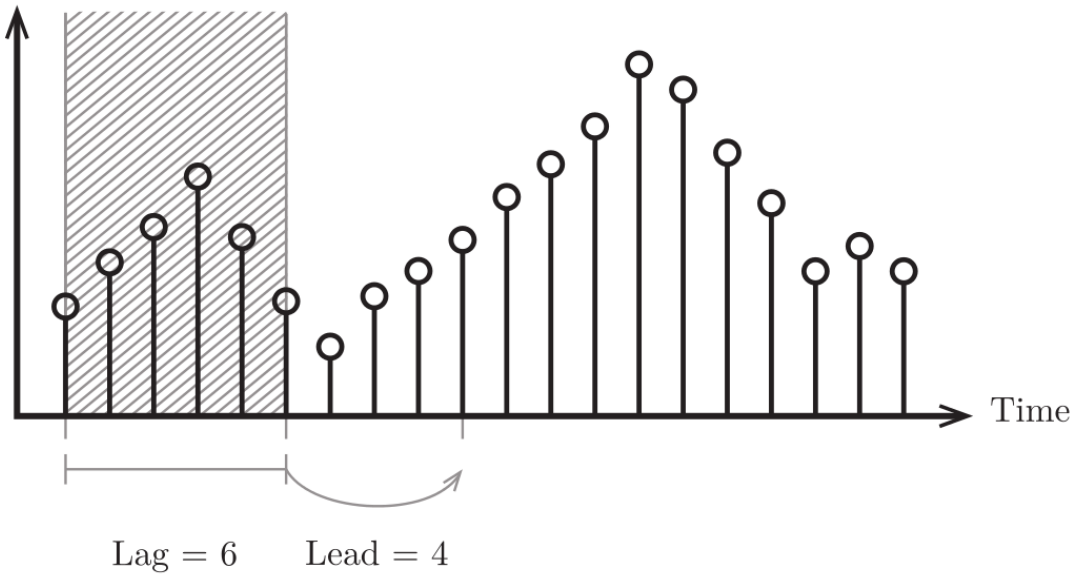

Figure 3-5: Lead and Lag Experiment where we use exactly 6 samples of data to predict the sample 4 timestamps in the future. Notice how this "window" can slide forward in time creating new prediction problems.

Hence, f_t will be the concatenation of $s_{t-i}, s_{t-i+1}, ..., s_t$ with the label of s_{t+j}. This further implies that $F = \{f_i, ..., f_{n-j}\}$ has size $n - j - i + 1$, as the operation doesn't make sense for timestamps lower than i or greater than $n - j$ (no samples before 0, or after $n - 1$). Also note how the new samples f_t have dimension $i * d$ and thus live in $\mathbb{R}^{i*d}$.

The definition above is a bit formal and maybe hard to grasp, but conceptually, we created a new Design Matrix where every new sample contains data (features) of i original samples, labeled with the label of j samples ahead. Then, it should be apparent how a classifier in the new space F will answer the type of lead-lag questions that we were interested in the first place, since classifying a sample correctly in the new space means using information of i samples in the past, to predict the label j samples in the future. This procedure is illustrated in Figure 3-6.

This node thus introduces the first two parameters in MLBlocks: the lead and the lag.

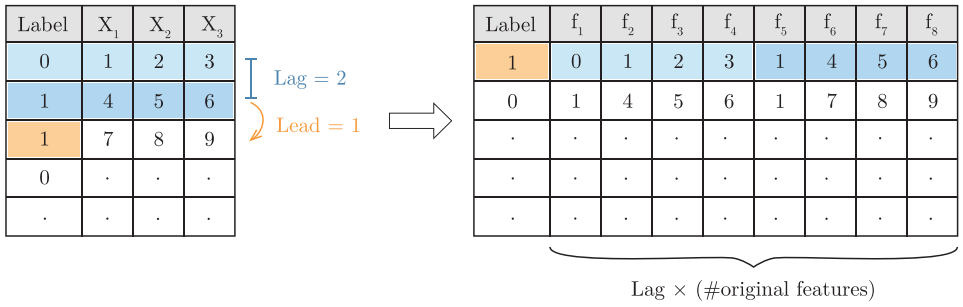

Figure 3-6: Process of Flattening input entity matrices.

3.4.2 Train Classifier Node

Discriminatory Models in MLBlocks can be generated in two fundamental ways. One can either use a standard Logistic Regression, or Delphi, a distributed multi-algorithm, multi-user, self-optimizing machine learning system implemented in our Lab, that finds the best classification model by training and searching smartly through thousands of models in the cloud.

Delphi

Delphi was developed at the ALFA Lab by Will Drevo. It uses a combination of Multi-Armed Bandit Search and Gaussian Process Optimization to efficiently find the best model for classification.

In Delphi, different types of Models and Algorithms are seen as different "Slot-Machines" in the Multi-Armed Bandit Problem (MAB). The MAB Problem is a problem in probability theory that considers the question, "which slot machine should one try next?", given n slot machines and the previous rewards gained from trying them out. The idea is that, assuming every machine has a fixed distribution (or more generally that its behavior is defined by a Markov Process), one can envision an algorithm that suggest which slot machines to "try" in order to maximize the

rewards. Thus, Delphi uses ideas and algorithms from this problem to know what "type" of algorithm to iteratively try; whether it should be SVMs with Polynomial Kernels, Decision Trees with Gini Cuts, or K-Nearest Neighbors for example.

Then, after deciding which type of model, referred to as a "Hyperpartition" or "Frozen Set", Delphi will use Gaussian Process Optimization techniques to decide with what hyperparameters to "try" and train such type of model. Gaussian Process Optimization is the study of how to sample a function f that is assumed to be noisy and expensive to compute (as in the case of training a classifier), with the goal of achieving the maximum value of such function as quickly as possible. Thus, if you take $f(\vec{x})$ to be the cross-validation score of some K-fold evaluation, where $\vec{x}$ is the point in the parameter space of such a model (corresponding to a particular setting of the parameters), then it should be clear how the Gaussian Process Optimization techniques can be useful for Delphi.

Combining these two strategies makes Delphi able to find the model and parameter combination that performs best for any given 2D data set.

3.4.3 Classifier Predict and Evaluate Nodes

There are many metrics and prediction strategies one can use to evaluate models after they have been trained; MLBlocks supports a few of them.

The Logisitic Regression and Delphi will both do a random train/test split for which the user can specify the percentage of the split.

Finally, for binary classification problems, the user can decide to evaluate the models using the Area Under the Curve (AUC) of the Receiver Operating Characteristic (ROC). MLBlocks can also plot this ROC curve, if desired.

3.5 Blue Line: Generative Modeling

Next, we introduce the line for Generative Modeling. This is the Blue Line and Figure 3-7 shows how it goes through the **Time Series**, **Discretize**, **HMM Train**, **HMM Predict** and **Evaluate** nodes.

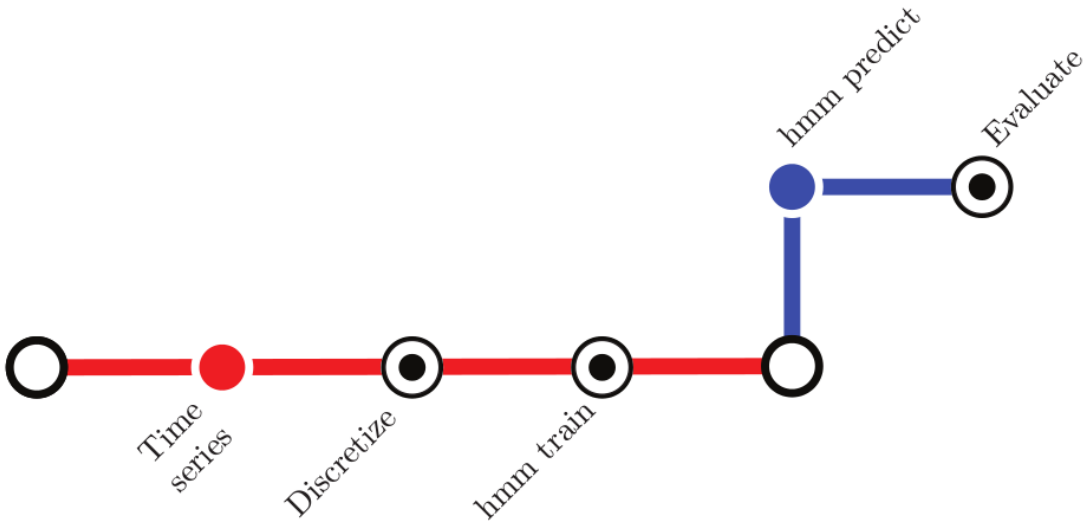

Figure 3-7: Blue Path: Hidden Markov Model

This line heavily uses techniques and ideas from Hidden Markov Models, so we provide a short background explanation.

Hidden Markov Models Background

Hidden Markov Models (HMMs) fall under Generative Models, as they can generate not only the label data, but the samples as well. They are of particular interest in machine learning because they provide a powerful way of modeling data – particularly data that is believed to be sequentially dependent on each other. As an example, consider a dataset where the goal is to detect invalid calls from a phone. If the phone was stolen at time t, it is very likely all the calls at time t, $t+1$, etc. are invalid which means they are far from independent of each other.

Markov Models are probabilistic versions of State Machines, and they model how a system changes from time to time, governed by some transition probabilities between a set of states. They are called *Markov* models because they contain the additional assumption that the transition between states depends only on the current state, and not on the history of states visited. This property is called the "Markov Property" and the idea is that the transition probability distribution at each state i does not depend on *how* the system got to state i.

In a *Hidden* Markov Model, the true state of the system is never revealed. However, we do have access to some "observation variables," which we assume to be related to the current state of the system, in some probabilistic way. In particular, we assume some "emission probabilites" that give the probability of an observation from some state i. Then, using exactly these "observations" is how we will model and predict the data; we will assume that our data was generated as a sequence of observed values in such HMM.

We can use HMMs for fundamentally two types of predictions. These are:

1. Finding the **state distribution** after seeing a sequence of observations.
2. Finding the **observation distribution** of the next observation after seeing a sequence of observations.

Note that they respectively correspond to finding $P(Y|\Theta)$ and $P(X|\Theta)$, where Y is the random variable denoting the state of the system, X is the random variable of the observations, and Θ is the vector containing all the parameters that fully characterize an instance of an HMM.

We will be using an HMM by first doing Maximum Likelihood Estimation (MLE) for its parameters. With such MLE estimates, we will predict the test data sequentially by using the second fundamental problem we listed, **observation distribution**, to get the most likely observation at the next step and using that as our "guess". These two steps are the "training" and "predicting" phases of our HMMs.

3.5.1 Time Series and Discretize Nodes

The first two nodes in the Blue Line are mainly preparation steps for the main **HMM Train** node, in a similar vein to the **Flat** node of the Orange Line.

Time Series

The Time Series node is intended to be just a simple helper function the goal of which is to transform the data into a format that is usable with whatever HMM tool

we choose (this transformation is almost trivial given a 3D MLBlocks Foundational Matrix and the specific HMM package we use.)

Discretization of Continuous Features

Hidden Markov Models have been mostly studied and understood in discrete spaces. That is, most of the models assume every feature in the dataset can only achieve values in a discrete space of B possible values (which are further assumed to be the integers from 0 to $(B-1)$).

Clearly, not all datasets are like this, but since many HMM packages (and in particular the HMM Package MLBlocks will be using) require this, we added the **Discretize** node to be able to discretize the data if needed.

The idea here is to "bin" the data. To do so, we will create b number of bins per feature, such that we can map original values to these "binned"/discretized values. Each bin will take the form of a range of values, and thus should be clear that if the bins cover $\mathbb{R}^1$, then any feature can be mapped to this discretized space.

MLBlocks uses a discrete version of HMM, so this step is necessary before we go on to the training phase.

3.5.2 HMM Train

The goal of tranining an HMM is to estimate its parameters as best as possible. These parameters are the transition probabilities between the states, the observation probabilities from each state, and a prior distribution on the states.

The HMM package that we use in MLBlocks uses the Baum-Welch algorithm to find a locally optimal solution. The Baum-Welch algorithm is an instantiation of the more general framework for estimating parameters, which is known as the EM algorithm. EM follows from the fact that the algorithm has two main steps, an Expectation step and a Maximization step. The EM as well as the Baum-Welch can only guarantee a local optima for the parameter settings.

3.5.3 HMM Predict

After we have successfully estimated HMM parameters Θ, we can start making predictions in the dataset. Predictions in HMM will also follow the sequential nature of HMM, in the sense that the prediction problems will follow the structure "what is the most likely observation at a time t given the data we have seen up to $t-1$?". Again, there is no reason the choices of how much data to use and what what time to predict shouldn't be parameterizable. In particular, they can be parameterized in the same way that we parameterized the Lead-Lag Experiments explained in the **Flat** node.

For our testing data set, we can use a user-specified lead j and lag i to predict in the following manner. First, we will feed the HMM with i samples starting at time 0, and ask for the most likely observation at time $i+j$. Then, we will move forward one timestamp, feed in samples from 1 to $i+1$, and ask about the most likely observation at time $i+j+1$.

HMMs, being generative models, will generate a complete distribution of the possible values the observation can take. That is, we will get a distribution of the values for each feature of the observation. Thus, if we want to extract the "most likely observation," we have to read the distribution and get values with the maximum probability of appearing, in every feature. Furthermore, in order to compare this with the true value suggested in the testing set, we will have to apply the same label logic that is expected from the set.

3.5.4 Evaluate

The Evaluate Node in the Blue Line will actually behave identically to the one in the Orange Line. We include this section for completeness, but AUC of the ROC and Accuracy metrics work in the same manner as in the Orange Line.

State Distribution Features

Before we move from the Blue Line into the next Lines, we discuss how we can use the first fundamental problem of computing the **state distribution** given an observation

sequence, as this will hopefully help motivate the next sections.

Interestingly, although this computation by itself can be of great value for some applications, these probabilities can also be used as a new feature space for learning as we explain below.

The idea is that these numbers, although they might look meaningless in terms of describing the data, must have some intrinsic relationship with the original data, because they were generated from it. Thus, one can envision using these as *features* themselves, effectively yielding the same problem in a different feature space.

This not only has the power to increase the prediction score by giving more dimensionality if combined with the original features of the problem, it can also be used as a powerful mechanism to protect the privacy of a sensitive dataset. This works because these values have very little semantic and contextual meaning for a human (they just look like a random distribution on the states of the HMM, which itself would have to be interpreted to contain explicit meaning), but can still encode predictive information from the dataset.

Using these "State Distributions" as features, is the motivation behind the last two lines MLBlocks supports; the Red and Green Lines. We call these synthetic features *sd-features*.

3.6 Red and Green Lines: Synthetic Features Predictions

As the Red and Green Line share the main computational step, we will be presenting them under a single section only. The images for the two Lines are presented in Figures 3-8 and 3-9.

It is worth noticing that although not explicitly denoted in the diagrams, the purple line is intended to present the "merging" of data from after the **Flat** node and after the **HMM Train** nodes. This corresponds to the creation of a new "merged" dataset, which combines features from the original set with the sd-features.

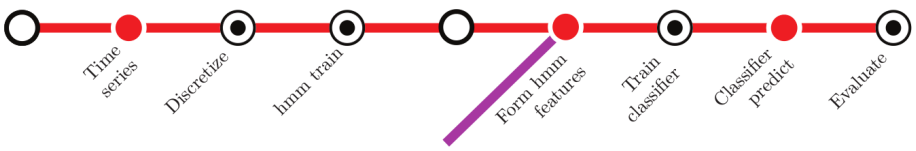

Figure 3-8: Red Line in MLBlocks.

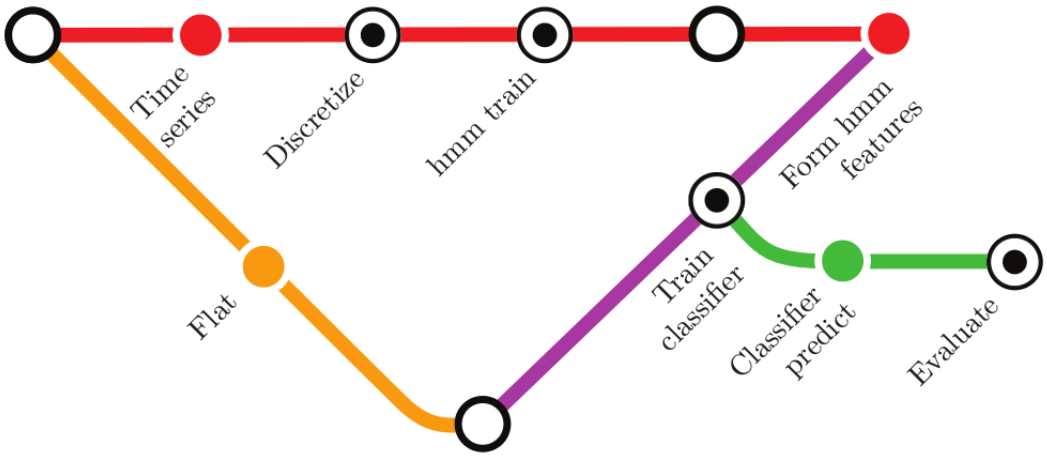

Figure 3-9: Green Line in MLBlocks.

The different way in which we "merge" these features is exactly and exclusively the difference between the Red and Green Lines. In particular, the Green Line will "merge" by simply concatenating the sd-features to the original dataset and passing this newly created dataset to the same **Train Classifier**, **Classifier Predict**, and **Evaluate** discriminative modeling triplet, as discussed in the Orange Line.

The Red Line will "merge" by applying the labels from the original dataset to the dataset created solely using the sd-features. Then it will pass this newly created dataset (completely in the sd-feature space) to the discriminative modeling triplet to find the best classifier it can find. Figures 3-10 and 3-11 show examples of how these merging procedures look in an example run.

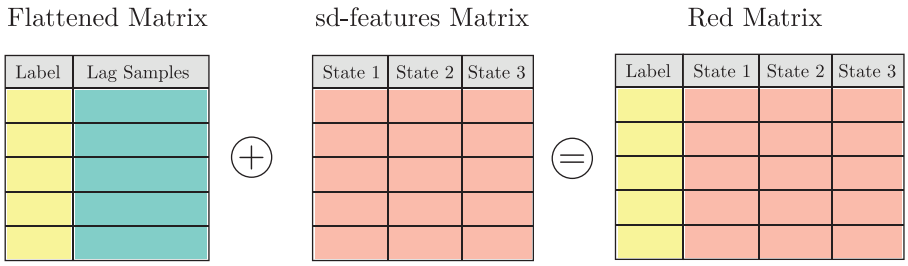

Figure 3-10: Example Dataset after "Merging" in the Red Line.

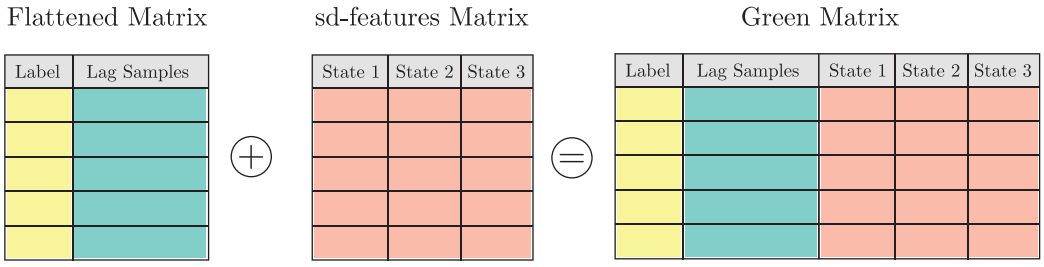

Figure 3-11: Example Dataset after "Merging" in the Green Line.

3.6.1 Form HMM Features Node

This node is responsible for generating the sd-features briefly discussed above. The final product will be k additional features for every sample in the flattened dataset (the result after the **Flat** node).

The first step to generating these n by k features is to train an HMM as we did in the Blue Path, but for the purposes of using its prediction of state distribution capabilities. After we have fully trained an HMM with k states, we will generate the k features of a flattened sample by getting the state distribution (k numbers) using the same lead-lag setup for which that flattened sample was generated. (That is, every flattened sample was generated in the **Flat** node by concatenating features of i samples in the past and the label of j samples in the future.) Then, when we create the sd-features for that particular sample, we will feed the same i observations into the HMM and get the state distribution of j timestamps in the future. This way,

there should be a one-to-one correspondence between samples in the flattened dataset and state distributions generated in this node; such that the merging described above is compatible.

Chapter 4

MLBlocks Architecture and Implementation

In this chapter we discuss the different design decisions and implementation details involved in developing MLBlocks. Let's start with the two basic notions required for using MLBlocks – its Command-Line Interface (CLI) and the Configuration File.

4.1 The MLBlocks CLI

The MLBlocks Software was designed to be used as a command-line tool that reads settings of an experiment from a configuration file and executes the experiment accordingly. This way, the workflow for MLBlocks is very simple: the data scientist can simply edit a configuration file in his favorite text editor and execute the experiment with a simple command in the command line.

This simple interface not only makes it easy to use by humans, but also gives it a very nice and desirable property, which is that of automation. It is a very easy to use interface for other programs, not even necessarily written in Python.

The complete Command-Line Interface (CLI) for MLBlocks cannot be simpler. Just one command executes a script, which reads the settings from a configuration file, the actual main entry point for MLBlocks. The exact command is simple:

```
python main.py
```

4.2 The Configuration File

The MLBlocks software is intended to be very flexible and highly tunable in terms of the experiment parameters. As suggested in the previous chapter, there are many parameters used by the different nodes and computational steps. The desire to have all of these parameters in one place, and one place only, further motivated the use of a single configuration file as main entry point. Moreover, having these many parameters be command-line arguments instead would have made for a cumbersome and complicated user interface.

We decided to use INI format for the configuration file for a variety of reasons. We were inspired by computer science's DRY principle ("don't repeat yourself"), and by the desire to find a format that could be easily edited by humans and computers alike. YAML was another strong contender as is is also very easy to edit by humans and computers, but Python's (the language of choice) standard ConfigParser module presented the most mature module to work with configuration files, and it primarily supports INI files. The only compromise made necesarry by this decision is that we have to stay within the section-key-pair structure that is inherent to the INI format, but this is more than sufficiently powerful for the simple type of configuration involved in MLBlocks.

With the choice of INI file format, the most intuitive use of its section-key-value pair nature was to use a section per parameterizable node in MLBlocks, and to let the corresponding key-value pairs be the parameters in the node. We also considered having a section per Line, but noticed that since some Lines share nodes together, this would have made some parameter definitions redundant. Thus, a section per node proved itself to be both more concise and still extendable to new Lines.

We now list all the tunable parameters in the MLBlocks Config File, presented section by section. There are six total sections: four that correspond to paremeterizable nodes, and two that deal with MLBlocks-wide parameters that are not specific to a particular Line. We start with these general two; MLBlocks and Data.

[**MLBlocks**]

- **line**: Orange | Blue | Red | Green. The Line you want to take. :)
- **lead**: integer. Requires ≥ 0. To be used in all aspects of Lead- Lag experiments. Nodes that use this are the Flat node, the HMM Predict and the Form HMM features.
- **lag**: integer. Requires ≥ 1. To be used in all aspects of Lead- Lag experiments. Nodes that use this are the Flat node, the HMM Predict and the Form HMM features.
- **test_size**: float. Requires $0.0 < x < 1.0$. Will the percentage of the test set in the train-test split used in HMM Predict and Train Classifier with Logistic Regression; Delphi uses a Stratified K-Fold with K=10, and thus will ignore this.
- **num_processes**: integer. Positive Integer ≥ 1, number of processes to spawn in parallelized routines.
- **debug**: True | False. Controls the level of output in the console.

The next section requests information about the input data. It has five parameters.

[**Data**]

- **data_path**: string. Path to folder with CSV files (representing the Foundational Matrix) or path to a single CSV file (representing 2D Matrix).
- **data_delimiter**: string. Sequence to look for when reading input CSVs. In most CSVs should be "," (without quotes).
- **skip_rows**: integer. Requires ≥ 0. Rows to skip when reading CSVs. Useful for CSVs with headers.

- **label_logic**: lambda expression. This is the logic that will be used to get the label out of every sample. Needs to be a lambda expression as in Python, where the input is an array-like. Thus, if a label is a column i just have "lambda x: x[i]". It supports basic arithmetic operations.

The next sections are now specific to particular nodes and are named accordingly.

[**TrainClassifier**]

- **mode**: LogisticRegression | DelphiLocal | DelphiCloud. Discriminative Modeling Technique to use.

- **learners_budget**: integer. Requires ≥ 1. Only used for DelphiCloud; budget of learners to explore.

- **restart_workers**: True | False. Only used for DelphiCloud; whether to restart all the Cloud workers in Delphi. Will yield more throughput, as sometimes there are straggeler workers in Delphi.

- **timeout**: integer. Requires ≥ 0. Only used for DelphiCloud; seconds to wait for DelphiCloud to train. Will come back with results thus far.

[**Discretize**]

- **binning_method**: EqualFreq | EqualWidth. Strategy of compute bin cut-offs. EqualFreq makes the cut-offs be such that every bin has roughly the same number of instances in the dataset. EqualWidth just divides the range of that feature in equal length bins.

- **num_bins**: integer. Requires ≥ 1. Number of bins per feature.

- **skip_columns**: comma separated integers. e.g. 0,3,8. Indices to skip in Discretization process. If skipping columns, because say they are already discretized, you need to make sure they are discretized and 0-indexed.

[**HMMTrain**]

- **mode**: Discrete. To be used for switching type of HMM. Only "Discrete" is supported.
- **num_states**: integer. Requires ≥ 1. Number of states in the HMM.
- **max_iterations**: integer. Requires ≥ 1. Sets a maximum number of iterations for the EM Algorithm.
- **log_likelhood_change**: float. Requires > 0. Sets the stopping condition for the EM algorithm.

[**Evaluate**]

- **metric**: ACCURACY | ROCAUC | ROCAUCPLOT. Metric to use for evaluation. If ROCAUCPLOT will use matplotlib to plot ROC curve. ROCAUC* can only be used for binary classification problems.
- **positive_class**: 0 | 1. Required for ROCAUC* metric; ignored otherwise.

Putting this all together makes the MLBlocks config file. A concrete example can be found in Appendix B.

MLBConfigManager

To make the parsing of such a file modular, and self-contained, but also to provide a helper class for developers wanting to use MLBlocks in an automated way, we constructed an "MLBConfigManager" class that has two main methods; read and write. The read takes in a path to an INI file and will return a dictionary with all the key-value pairs already parsed and interpreted. The write takes in any number of the keyword arguments and an output path, and writes a MLBlocks formated config file to that path with the specified values, and default values for non-specified.

```
example_data/
 |--- student1.csv
 |--- student2.csv                    example_data/feature-matrix.csv
 |--- student3.csv
 ...
```

Figure 4-1: Supported input data format for MLBlocks. Left is MLBlocks Foundational Matrix, right is a standard 2D Feature Matrix.

4.3 Input Data for MLBlocks

MLBlocks' implementation supports two conceptually different forms of data, both of which match the Foundational Matrix explained in the previous chapter. These are:

- A path to a folder containing one file per **Entity**, in which every file is a CSV file representing a matrix where the columns are the **Features** and the rows are the **Data Slices**.
- A path to a CSV file containing a standard 2D Design Matrix. As explained in the previous chapter, this is a special case of the Foundational Matrix and will be treated as such.

Examples of these forms are shown in Figures 4-1.

4.4 Implemention Details

We decided to implement the MLBlocks Software primarily in Python, for a few reasons. For one, it has recently become one of the most powerful scientific programming languages, next to R and Matlab. Moreover, tools like Numpy and Scipy have matured to a stage that made the development of many of the numerical computations and data manipulations easier and much more efficient. Second, Python is the language of expertise for me the main developer, and for many MIT students and ALFA lab members. This way, if the project is extended, developers, will have a better experience.

4.4.1 Code Structure

As MLBlocks was going to be a codebase of considerable size in Python, an effort was made from the beginning to keep functions as encapsulated and modular as possible, as well as to minimize the amount of code used. This motivated many relevant decisions. For example, the most important pieces of the root folder of MLBlocks (shown in figure 4-2) are the "main.py" script and the "config.ini" file (which should already be familiar), and the "blocks/" and "util/" folders.

```
MLBlocks/
  |-- blocks
  |-- config.ini
  |-- main.py
  |-- mlblocks
  |-- README.md
  |-- tmp
  |-- util
```

Figure 4-2: Root folder in MLBlocks

The "blocks/" folder is responsible for the logic that connects all the components of each Line together. The folder contains one file per Line, which should contain the implemention of a child class from an abstract base class we defined called the BaseBlock class.

Block Abstract Base Class

One of the main design goals in MLBlocks was to create an *extensible* software, in the sense that it should be easy for new developers to add their own new Lines. This design goal, combined with the importance of the Line concept and the attempt to reduce duplication of code, made us decide to define a abstract base class using Python's Abstract Base Class (ABC) package. This then becomes a central building block in the software, and provides a guide for new developers to follow.

Util Module

The other main folder in the code base is the "util/" folder. This folder is a module that meant to group all the computationally expensive routines implemented in each of the nodes. These include the routines for flattening, discretizing and creating sd-features, as well as other less expensive but essential routines like the test-train splitting code and other very useful helper functions across the code base.

Each of these routines was implemented as a separate submodule, which we present in detail in the next section.

4.4.2 Computation Modules

All of these submodules where designed to follow an object oriented approach in their implementation. There aren't special arguments for following such programming paradigm, other than keeping the implementation style consistent across the code base.

But before we start listing each module, we note that most of these modules involved some transformational step that created intermediate representations that were required for future steps in the experiments. Because many of these could also be helpful for the data scientists as standalone representations, we created a "tmp/" folder in MLBlocks, in which all of these "computation modules" can save their intermediate outputs.

Intermediate Representations and tmp folder

The "tmp/" folder in MLBlocks will contain one new folder for each dataset run through MLBlocks. Inside each of these datarun folders, there will be the same structure of intermediate representations. These folders are:

- **discretized/**: Contains the same one-file-per-entity structure as the input data but with discretized values.

- **flattened/**: Contains the same one-file-per-entity structure as the input data but flattened with the given lead and lag. It also contains an additional file that appends all the entities together and adds the 'entity_id' column.
- **sd_features/**: The sole sd-features files per entity, given the lead and lag. Each row should contain k features, one per state in HMM.
- **green_features/**: The combination of the sd-features with the flattened files per entity.
- **red_features/**: The combination of the sd-features with just the label of the flattened files per entity.

Thus, MLBlocks can be used to compute any of these byproducts for a particular dataset.

We now list some implementation details for the "computation modules". Each sections is named to match the object implementing the functionality.

TrainTestSplitter

A common step not just in the MLBlocks software, but in machine learning systems in general is the splitting of a dataset into test and training subsets. In particular, we created a class that does this for the special Foundational Matrix in MLBlocks. Here, the split will happen by entity; a random subset of *test_size* entities are used for testing and the rest $(1 - test\_size)$ percent are used for training.

If the input data is a single 2D design matrix CSV file, however, the split will happen via samples (rows). That is, *test_size* random rows are selected for testing, and the rest for training.

Flattener

This class implements exactly the logic explained in the previous chapter. The key point is that it contains a for-loop of the size of the output space, along with a flag for whether or not to include the labels for the previous lag samples. This is still

not surfaced in the MLBlocks Config File, and it defaults to False; not to include the labels of the lag samples.

Discretizer

To implement this procedure, we use a combination of Numpy and a Data Mining package in Python called Orange. We first compile the complete training dataset into one in-memory matrix, so that we can compute the cutoffs with Orange on all the values for a given feature. Then, we use Numpy's *digitalize* function to apply these cutoffs to both the training data and the testing data, keeping in mind that the user could have required to skip columns for discretization.

SDFHelper

The computation of the sd-features is actually the most computationally expensive intermediate step. This is because, for every sample in every entity, you have to possibly feed all the previous observations, as well as execute the subprocess of calling the FastHMM package. At the same time, this procedure can be highly parallelizable, particularly across entities. Hence, we used the multiprocessing package in Python to implement the parallelization and speed up the process.

The parallelization will divide the entities equally among processors and will launch the computation of sd-features as explained in the previous chapter.

4.4.3 Training Modules

Two of the training strategies in MLBlocks involve integrating other projects in the ALFA Lab, in this case FastHMM and Delphi. They are both very good implementations, but may lack the maturity of open-sourced projects such as Scikit-Learn. Integrating them in a modular fashion that could abstract the details away from the user was indeed a challenge, but we believe we have been successful.

FastHMM and Python Wrapper

FastHMM is the package in MLBlocks that does the HMM training in the Blue Path. It was implemented by members of the ALFA Lab and it written in C++. This was a faster choice than other more mature packages written in Python and Matlab, especially when training HMMs which are also known to be computationally expensive.

Internally, it uses the LibArmadillo linear algebra library in C++, and so this becomes one of the dependencies of MLBlocks itself.

Moreover, the FastHMM provided a Python wrapper we were able to leverage with some modifications.

Integrating Delphi

Delphi's architecture involves a central "Master" node called the "DataHub", along with many worker machines we simply call "Workers". The DataHub is nothing other than a MySQL server that will contain information about all the active "jobs" or dataruns, and will also save the information about all the models trained by the workers.

The workers carry out the training, and behave by continually probing the DataHub for a HyperPartition and Model to train on. Then they write the results to the database and probe again. In the cloud we use CSAIL's OpenStack private computing cloud to deploy 80-100 worker nodes; locally, Delphi is executed in the user's computer with one worker process which talks to a MySQL server in the same computer. Thus, if the user desires to run Delphi locally, he/she is required to set-up the Delphi MySQL configuration.

In our instantiation of Delphi for MLBlocks, we used 11 algorithms in both deployments, as well as the "Hierarchical Search" strategy, described in Drevo's thesis.

Algorithms
Support Vector Machine
Logistic Regression
Stochastic Gradient Descent
Passive Aggressive
Deep Belief Network
K-Nearest Neighbors
Decision Tree
Extra Trees
Random Forest
Gaussian Naive Bayes
Multinomial Naive Bayes
Bernoulii Naive Bayes

Figure 4-3: List of 11 algorithms used by Delphi

Logistic Regression

The logistic regression algorithm that can be used in MLBlocks is the same one from the mature library Scikit-Learn.

4.5 MLBlocks Output

The principal medium for output in MLBlocks is the standard output in the command line. This keeps the usage of MLBlocks in one place and one place only, and still keeps it automatable, as this output can be piped into other programs. But to further this potential synergy between MLBlocks and other programs, MLBlocks also outputs a "results.json" file with details of the last-run experiment as well the best-achieved score in the specified metric.

Metric Evaluation Details

The output metric in MLBlocks is either a standard "accuracy" metric, or the Area Under the ROC curve. The AUC of the ROC is computed similarly across lines, but the "accuracy" metric varies slightly from Line to Line. In particular,

- Delphi uses the mean of a Stratified 10-Fold cross validation.

- Logistic Regression uses the mean accuracy on the given test data and labels.
- HMM Training uses the mean accuracy on the given test data and labels.

4.6 Extending to Additional Lines

We finish this chapter by providing a small guide into implementing new Lines in MLBlocks. The instructions are simple: assume without loss of generality that you want to create a new "Purple Line":

1. Implement a class called Purple in blocks/purple_.py, which inherts from the BaseBlock class provided in blocks/baseblock.py. (Note the trailing underscore is important and its purpose is to distinguish between any possible modules you yourself might need that might conflict in name.)
2. Make sure your class implements ___init___, transform_data, train, predict, and evaluate. You are encouranged to use BaseBlock's evaluate method by making sure you save the "accuracy", "y_true" and "y_score" attributes on your class by the time predict finishes.
3. Extend the MLBConfigManager class to parse any parameters you need and edit the sample_config.ini file.
4. Feel free to use the helper functions in the util folder and the global MLBlocks object to manage your paths, as well as to update the MLBlocks diagram.

That is it. Any user that now specifies "Purple" in the config file will be redirected to your code.

This concludes our discussion and presentation of the MLBlocks Software, and we continue by showcasing how we used MLBlocks to help us solve two real-world machine learning problem, predicting stopout in MOOCs and predicting the destination of a car.

Chapter 5

Using MLBlocks: Predicting Stopout in MOOCs

In this section we showcase how we used in MLBlocks to efficiently get results from a dataset on Massive Open Online Courses (MOOCs). In particular, this example showcases how efficiently we can do all sorts of Lead-Lag Experiments in MLBlocks, as well as get data behavior by casting different prediction problems under the different modeling techniques.

5.1 The Foundational MOOCs Matrix

We use the same example data for this problem as we discussed in section 3.3.5. This data was given to us in one flat file, where every 15 rows corresponded to 15 weeks of data for a given student. Therefore, a simple stream read that would separate every 15 lines into a single student file would already convert this data form into our MLBlocks Foundational Matrix.

The dataset contained 9,814 students, each with 28 features of data for the period of 15 weeks. It is a binary classification problem (which helped us showcase the AUCROC features of MLBlocks), where 0 is the "positive class" signifying stopout, and 1 means the student is still active that week.

5.2 Results

5.2.1 Lead and Lag Experiments

One of the most powerful features of MLBlocks is the ability to cast lead-lag experiments extremely easily. It is possible to ask all types of questions regarding the temporal nature of our dataset. In particular, we explored lead-lag properties of our data by doing a grid search on all paths with lead and lag values on 1, 3, 5, and 7 weeks. Additional parameters included discretizing the data using 5 bins of equal frequency, using a random 30% of students as test set and an HMM with only 4 hidden states.

We also fixed the classifier training on these experiments to the Logistic Regression mode to remove the complexity of the Delphi high dimensional search, and to have less moving parts when analysing the difference in behaviors across lines in MLBlocks. The results were as follows.

Orange Line

	Lag = 7	90.12%	91.46%	92.46%	93.43%
	Lag = 5	88.07%	89.16%	90.17%	90.25%
Accuracy Table	Lag = 3	85.57%	85.77%	87.02%	87.47%
	Lag = 1	81.74%	81.96%	82.84%	85.06%
		Lead = 1	Lead = 3	Lead = 5	Lead = 7

Blue Line

	Lag = 7	89.81%	88.77%	90.80%	98.30%
	Lag = 5	85.94%	87.28%	89.13%	91.48%
Accuracy Table	Lag = 3	81.86%	83.55%	87.23%	88.73%
	Lag = 1	76.53%	82.63%	84.52%	86.76%
		Lead = 1	Lead = 3	Lead = 5	Lead = 7

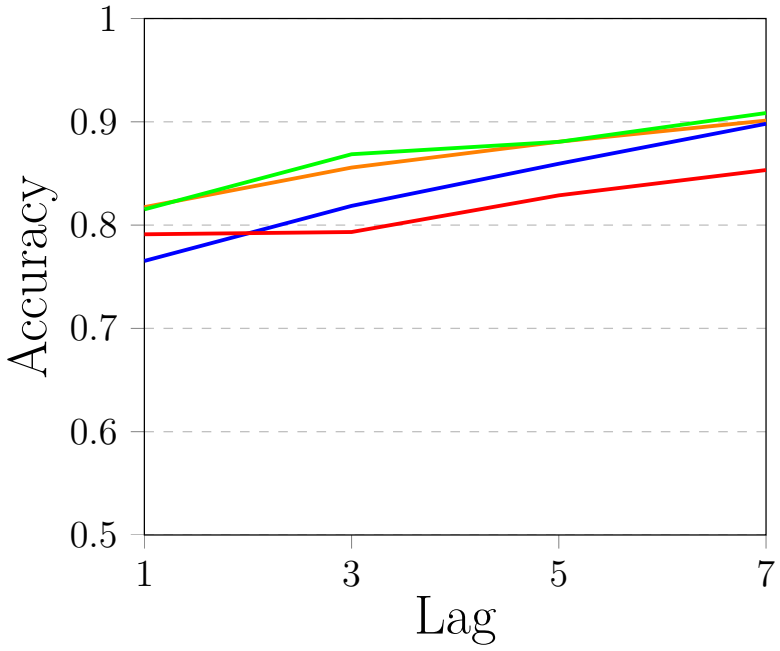

Figure 5-1: Performance for predicting stopout one week in the future (Lead = 1), as a function of weeks of data used from the past (Lag).

Red Line

Accuracy Table				
Lag = 7	85.33%	87.97%	89.95%	93.37%
Lag = 5	82.88%	85.34%	87.93%	89.57%
Lag = 3	79.33%	83.23%	85.82%	87.93%
Lag = 1	79.11%	79.46%	82.81%	85.49%
	Lead = 1	Lead = 3	Lead = 5	Lead = 7

Green Line

Accuracy Table				
Lag = 7	90.85%	91.92%	92.05%	93.12%
Lag = 5	88.06%	89.65%	90.28%	90.41%
Lag = 3	86.86%	86.37%	87.07%	87.32%
Lag = 1	81.52%	82.20%	83.13%	84.76%
	Lead = 1	Lead = 3	Lead = 5	Lead = 7

Analysis

As we can see from the tables, accuracy clearly increases with the increase of Lag – but what might be surprising is that it also increases with the Lead. This happens across all Lines. The reasoning behind the Lag is simply the intuition that with more data about the student, you are more likely to be able to predict well. Increase of accuracy with Lead has to do with the fact that after a student drops out, he/she stays that way. Thus, if the classifier can detect a dropout at time t, increasing the lead to ask about further time stamps makes prediction easier.

The Orange and Green Lines achieved the best performances. The performances are extremely similar, which might have been due to the fact that we used relatively few additional sd-features (4). Between the Red and the Green Lines the Red Line performed notably lower, highlighting the importance of the original features when predicting. However, if this lower performance is tolerable, then the exclusive use of sd-features might be of great interest because of how they encode what may be sensitive information in the original features. That is, producing powerful predictions with only four sd-features is encouraging and attractive because of their lack of semantic meaning.

This suggested further examination by increasing the number of hidden states in the Hidden Markov model. And that's exactly what we do in the next section.

5.2.2 Further Examination

The results above suggest that the prediction problem isn't as hard because after a student drops out, he/she stays that way. This means that there are many "easy" prediction problems, where the classifier can see that the student dropped out in a previous week.

Thus, we decided to pre-process the data by removing all weeks after the stopout week. That is, we truncated every student file by only keeping all the weeks prior to the stopout, including the week of dropout itself. Although this removed the many "easy" 0 predictions, it yields a dataset that is mainly labeled as 1.

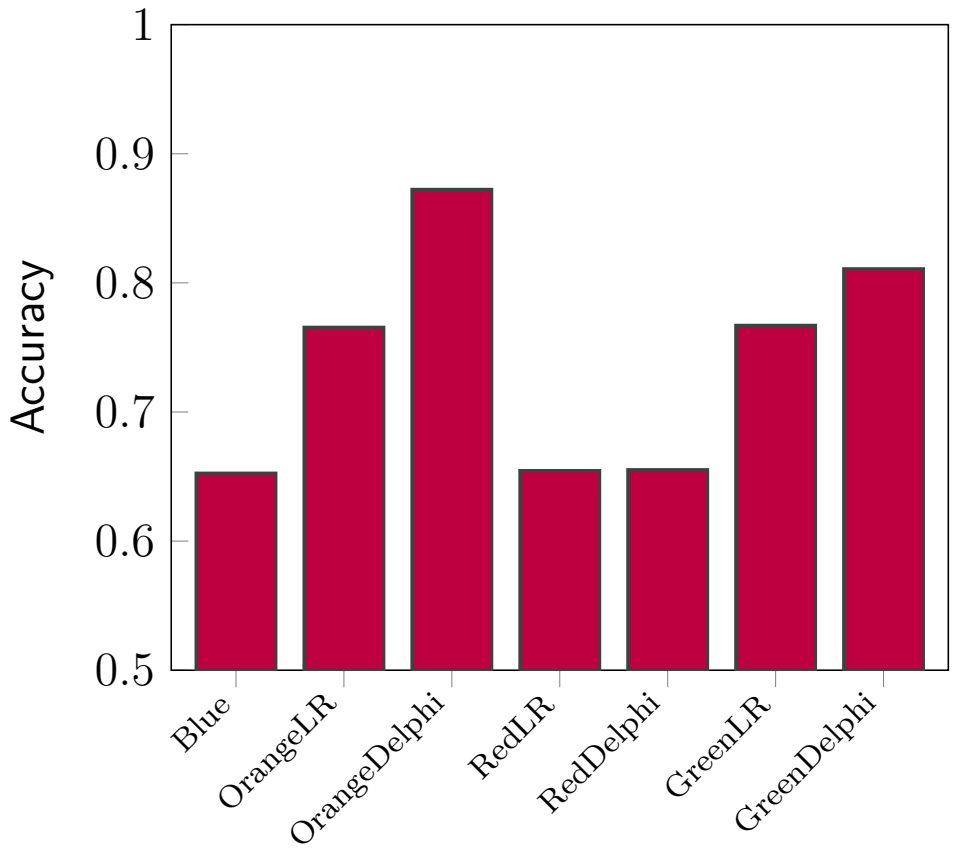

Figure 5-2: Performance of MLBlock Lines on the filtered MOOC dataset.

Thus, our final setup for this experiment was to use 29 hidden states, a log likelihood change stopping criterion of 1 capped at 30 iterations, equal frequency discretization into 5 values, and we focused on the particular problem of predicting stopout one week in the future using two weeks of previous data. That is, lag = 2, lead = 1. The results can be shown in Figure 5-2.

Analysis

From the above results it is easy to see how Delphi will always boost the performance of the Lines that pass through the TrainClassifier node. We take it for granted that Delphi can be treated as the golden standard of classification algorithms because of its intelligent and powerful search of parameters.

Furthermore, it is nice to see how again the Orange and Green Line where the

most powerful for this problem, matching the results of the initial exploration in the previous section. Finally, we reiterate the power MLBlocks gives, as further examination of results demonstrated that the best Orange Line estimator was a K-Nearest Neighbor algorithm with K = 14 and using a Minkowski distance metric; the best estimator for the Red Line was a K-Nearest Neighbor as well but with K = 18 and a Manhattan distance metric; while for the Green Line the best algorithm was a Passive Aggressive classifier with a very small C value of $2.55x10^{-}5$, 93 epochs of training, and using the Hinge Loss as loss function.

These results again show how MLBlocks can solve the data scientist conundrum illustrated in the Introduction chapter; as a automated run can yield this comprehensive report.

5.3 Conclusions

From this experimentation, we show a few things. One is that MLBlocks can be used very efficiently to get many lead-lag results by using its automation interface. The results for these 54 problems (4 lags, 4 leads and 4 lines), where all achieved within two days of computations by the execution of a single automated script. It furthermore very quickly illustrated temporal properties of our time series data, that we were able to interpret and can be used to further define the application of such machine learning system for predicting stopout.

It was also very easy to see the effect of synthetic sd-features in our context. If the data contained sensitive information, the domain expert could decide on whether he/she would like to take the trade-off of using synthetic features to encode data, with slightly less prediction power.

Chapter 6

Using MLBlocks: Predicting Destination

In this chapter, we discuss how we take another real world problem and used ML-Blocks to solve it. The problem consists of trying to predict the destination of a car using information about its past trajectories.

6.1 Motivation and Applications

This problem may seem a bit non-practical at first glance, as one may ask, if a driver already knows where he is going, why may it be useful if the car figures it out too? The answer is that most applications of knowing the destination involve having other systems build in top of this information.

That is, if the car knows where it is going, the user can not only skip the small step of entering the address in a GPS/GIS, but the car might be able to suggest better routing to avoid any current traffic events on the road. It may be able to set itself to use optimal fueling strategies for the destination, to suggest a place to eat or refuel on the way if needed, send an "I am on my way" message to attendees of the meeting, or perform many other targeted geomarketing or location-based social networking applications. It may be able to prepare the internal thermal conditions of the car to gradually match the changes outside if headed to a cooler or warmer place;

or it can be simply be used to detect abnormal behavior in the trips and notify the user of possible suspicious activity.

These are just a few of the benefits that become possible if the car knows the destination without having to wait for user input. Plus, as the field of driverless cars keeps maturing, this might indeed become a problem of greater interest.

6.2 Problem Definition

Before we proceed, it is important we accurately define the problem we will be tackling. We will be attempting to predict the destination of a trip, as soon as the user turns on his car. We will require that the only information used for the prediction is the car's past trajectories and possibly the conditions at the instant the car was started. Furthermore, note that this modeling is on a per-car basis.

The following sections will focus on how we started at the **Extract** node of the MLBlocks diagram and went all the way to push the the data across all different paths.

6.3 Raw Data

The dataset had recording of thousands of signals and readings from prototype cars that only a set number of employees had access to. How this data was collected is irrelevant to us, but its structure is not. We were provided with an example dataset that included data from 8 cars, collected across 5 months, for a total of 687GB of data.

This is a lot of data, especially considering it was formatted as a set of SQLite files, and SQLite is known for being a lightweight format designed to be used in mobile devices and other embdedded systems.

The data came in in the following directory tree structure:

```
data/
|--- CAR_1
```

```
|---|--- 2014-07-07
|---|---|--- 2014-07-07 21-10-12-001900 Partition 0.db
|---|---|--- 2014-07-07 21-10-12-104300 Partition 0.db
|---|--- 2014-07-08
|---|---|--- 2014-07-08 06-33-29-001900 Partition 0.db
|---|---|--- 2014-07-08 06-34-11-134100 Partition 0.db
|---|---|--- 2014-07-08 11-34-53-089500 Partition 0.db
|---|---|--- 2014-07-08 14-06-50-001900 Partition 0.db
...
```

where every .db file is a SQLite Database File. These files can be conveniently opened with a SQLite Database Browser to quickly navigate through their relational data structure.

The database schema of these .db files was not designed for machine learning, just as we had claim in section 3.3.1. Nonetheless, it gave a flexible original structure for reading such data. We now discuss the changes we went through in order to get the data in a format that we could easily work with.

To understand the exact schema of each of the .db files, we need to introduce a few important aspects of how the data was collected. The first is the idea of a 'Message'. A Message was the main abstraction data was recorded in its raw format. Messages would group batch of related readings together, which could be GPS readings, statuses on the media player, microphone input readings or internal car diagnosis messages.

The next important notion is that of a 'Channel'. This is the actual medium that recorded the readings. A Message contains readings from a number of Channels. For example, messages from ambient light data contain readings from 8 channels, where different channels are used to encode information about color of the light, intensity and other stats for the UI of such system.

Readings in a Channel come in any of 5 types; State Encoded, Double, Analog, Digital and Text, listed in order of descending frequency.

Finally, we note the subtle but important detail that a given Channel can appear in multiple type of Messages. This last statement will motivate us to define a new

notion of a 'Signal' that maps almost one-to-one with Channel, but lets us discern readings of a Channel that were used for different Messages.

6.3.1 Trip and Signal Abstraction

Defining good abstractions and concepts is a key part of any system design. In his influential book *The Mythical Man-Month*[**?**], Fred Brooks said that "Conceptual integrity is the most important consideration in system design". Thus, we took a closer look at what important concepts and abstractions we needed to make to be able to predict destination nicely.

The first one was that of a "Signal," as introduced in our last section. A Signal is simply the abstraction of a sequence of (timestamp, value) pairs for a given Channel and Message pair. This is the main building block of information that we get from the raw data, and it inherently imposes a times series structure that nicely matches how the data was generated.

One of the purposes of coming up with such a concept was to flatten the structure of Channels and Messages to make them easier to think and reason about, and to help us better discern readings that arose in different messages even though they used the same channel. With this definition, the raw data reduces to just "a set of Signals", which again is much simpler to think about as opposed to its complicated Message-Channel structure.

The second important concept that we define, and arguably the most important, is the concept of a Trip. This concept followed immediately from the problem definition of destination prediction and application. This translates the idea of a 'sample' in a machine learning context to our specific destination prediction problem.

Furthermore, it is imperative that we group them the Signals in terms of the particular Trip that they were recorded in. This was almost already done for us in the raw data, as the number of partitions (or .db files) mapped almost one-to-one with car trips. We say almost because only a subset of the partitions (.db files) were actually recordings of Signals from where the car started to where it was turned off. Some other partitions are still valid in light of the other JLR applications, but contain

other information – for example in the style of a "wake up" or "hello" signal from a system.

Having introduced these two important concepts we proceed to explain the experimental procedure of transforming this raw data to the MLBlocks Foundational Matrix, via the **Extract**, **Interpret** and **Aggregate** nodes. In particular, we note how we were able to focus on what effectively is the feature engineering process, thanks to the confidence that the MLBlocks software inspires in the modeling phase of the standard machine learning pipeline.

6.4 Experimental

6.4.1 Extract: Trip CSV Files

The data in its raw format is too big and complex to be of practical use when running experiments. Moreover, it contains much information that is of very little use when predicting destination.

Thus, as the first step in predicting destination, we decided to extract relevant signals and place them in a flat CSV file format that is much less complex, and is easy to manage and manipulate. In particular, we started by just exporting arguably the two most important signals in the destination prediction problem, the Latitude and Longitude signals.

To do this extraction we created some software infrastructure that could help us stream through the raw data structure that we outlined in the previous section and read of a particular signal (or subset of signals) of interest. We used this infrastructure to read all the files in the raw data structure, and create a new output directory with a similar Car > Date > File structure, in which the leafs or files of the directory tree where CSV files of the form:

$$Timestamp, Latitude, Longitude$$

Right off this linear scan, we note that the number of files reduced as, like we

discussed in the Raw Data section, not all partitions had information about trips. The way we detected this was simply by the presence of Latitude and Longitude. We also note, how extracting just these two signals out of the thousands provided, reduced to complexity of the data in size from the 687GB to 694MB. This process from partitions of raw data to trip CSV files is summarized in the following table.

Car	Number of Raw Partition Files	Number of Trips
Car 1	1101	704
Car 2	888	519
Car 3	639	258
Car 4	1007	555
Car 5	643	421
Car 6	990	494
TOTAL	**7079**	**3513**

By having this collections of CSV files, one per trip, we have successfully done the **Extract** node in the MLBlocks diagram, and are ready to move on into the **Interpret** node.

6.4.2 Interpret: Visualizing Trips

At this point, we thought it would be of value to start visualizing these GPS readings, in order to prepare ourselves to create the optimal Foundational Matrix for predictions.

In order to visualize our Trip CSV files, we used an online tool that leveraged the Google Maps API to visualize a series of latitude and logitude readings. We had to sample a subset of all the readings in a trip, as they were at times too many to process. This told us a first fact about the data – the high frequency for which we had GPS readings (in the order of milliseconds). After this sampling we were able to see trips like the ones in figures 6-1 and 6-2.

These visualizations gave us a better feel for the data, but also quickly motivated us to clean it, as shown in figure 6-2 there were many trips we call “Bad-Trips” that were made up of all bad readings.

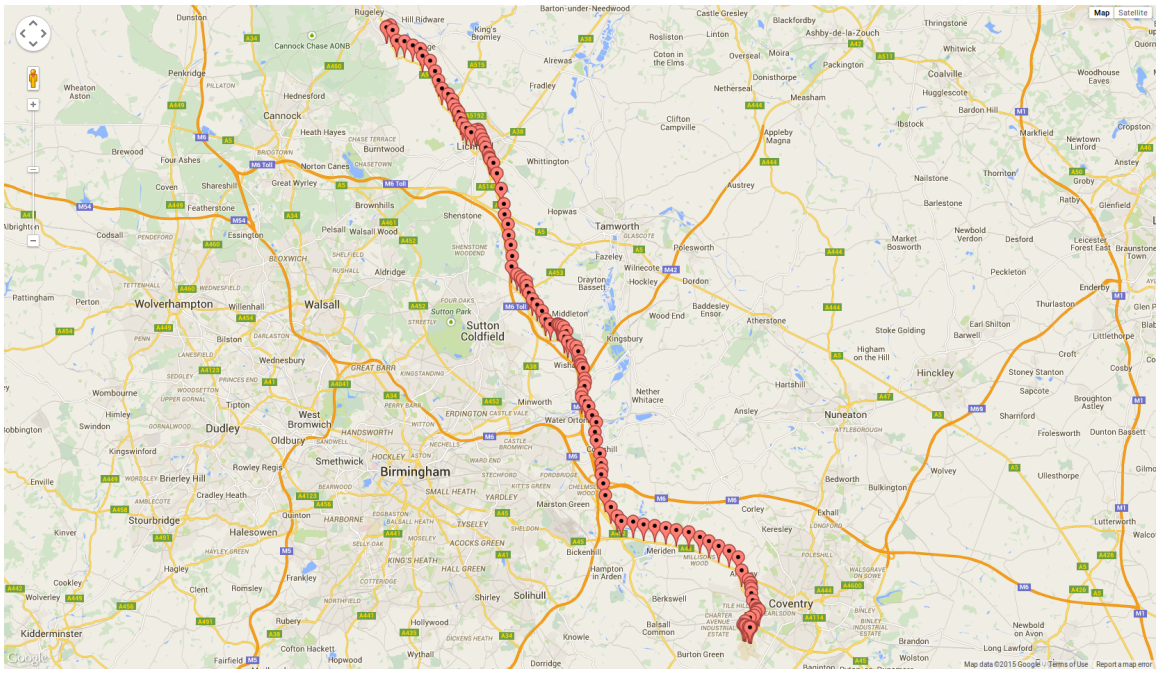

Figure 6-1: An example trip, sampled every 100th reading.

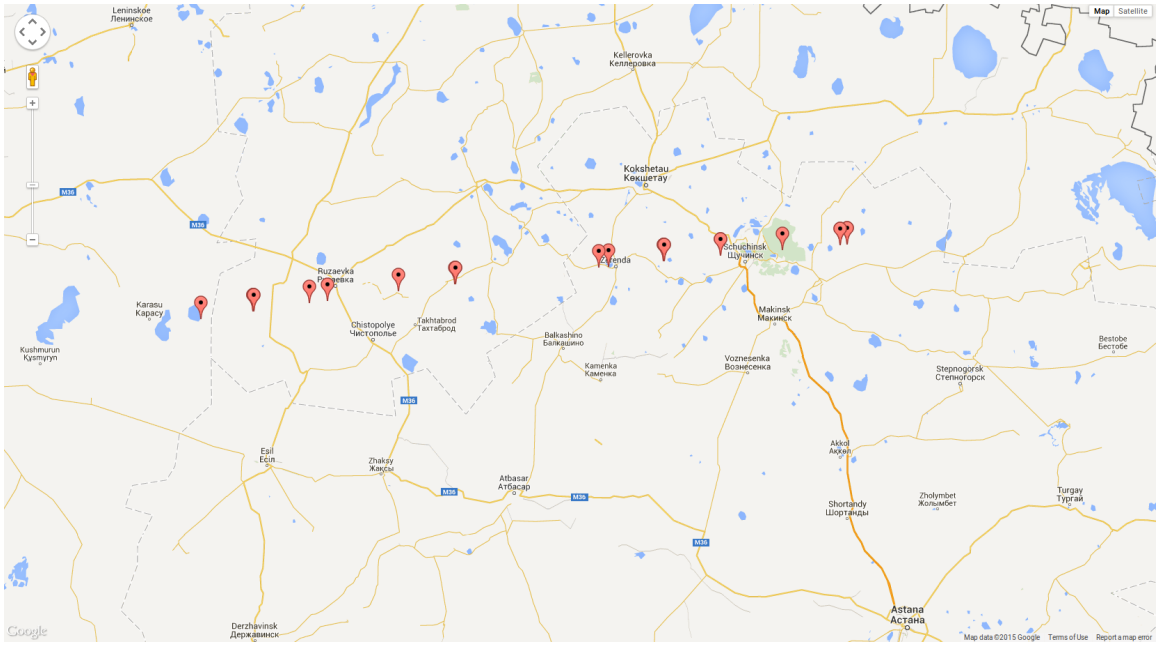

Figure 6-2: Another example trip.

Cleaning Trip Files

For cleaning the trips, we devised three main heuristics. The idea was to stream read through the Trip CSV files and throw away what seemed to be invalid readings. The

heuristics we used were:

- **Country Borders**: If a sample would be outside of the country borders for such car we considered it invalid. This would at worst throw away trips that were completely out-of-the-country, but there were none of this type.
- **0.0, 0.0 Readings**: 0 is a special value, and as such there were many readings in the middle of trips that suggested the car was at latitude 0, longitude 0, which is a bit unlikely given 0,0 is in the middle of the Atlantic Ocean.
- **High Velocity**: We also checked consecutive pairs of latitude and longitude readings for possible implications of unrealistic velocities. That is, if two consecutive readings would imply a velocity of more than 200mph we would deem this unlikely and throw away the readings as well.

These simple three heuristics were able to throw away 0.80%, 4.35%, and 0.01% of all the readings respectively. This resulted in 340 trips vanishing, leaving us with 3173 "clean trips". This process is also illustrated in Figure 6-3

6.4.3 Labeling Trips

Now that we had clean trip files, we started thinking about how to use the data for prediction. There is no reason to not use supervised learning techniques, as we have complete data, the problem of predicting the destination feels inherently supervised, and the techniques we support in MLBlocks are all based on supervised learning techniques. Thus, at this point we asked ourselves the question, what is the label in this data?

The answer to that question is that the label for every trip is its last latitude-longitude reading in the sequence. This is clearly the destination of the trip and hence the label. However, we note that in our application, trying to predict both the latitude and longitude would make it a complex Multi-Label Learning problem. Then, in order to address this, our initial idea was to cluster the labels via KMeans and use these clusters as the labels.

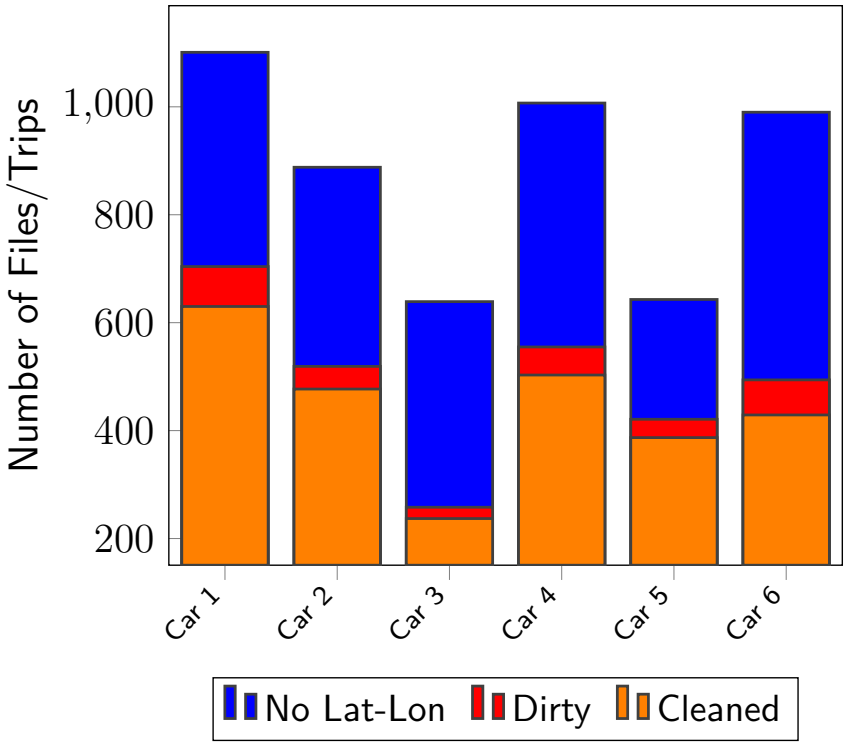

Figure 6-3: Summary of trip cleaning.

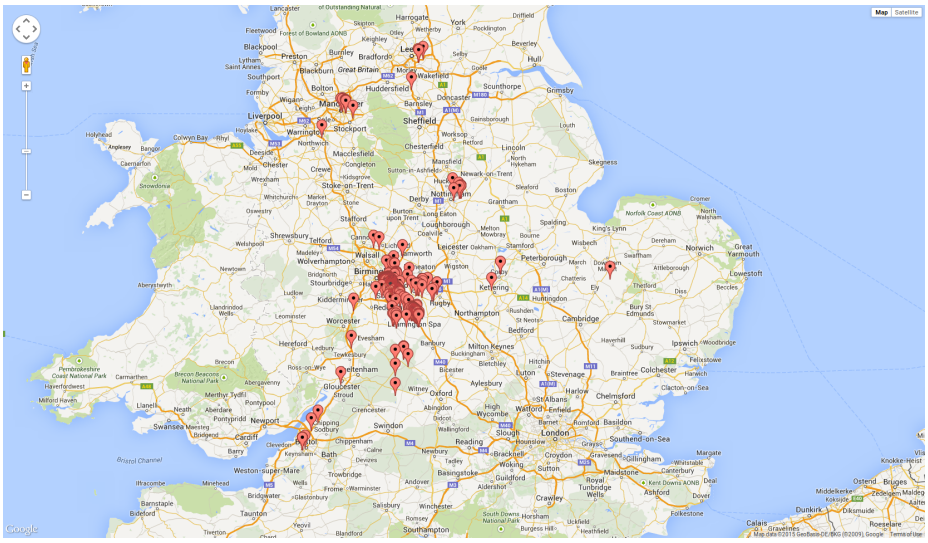

Figure 6-4: Endpoints for a given car across all 5 months of driving.

We started by extracting all the endpoints for a given car and running KMeans and KMedoids clustering algorithms on these. However, these algorithms didn't perform too well, because in cases where you have long trips to sparse destinations, this would make the mean somewhere in between these places, which is not a sensical destination.

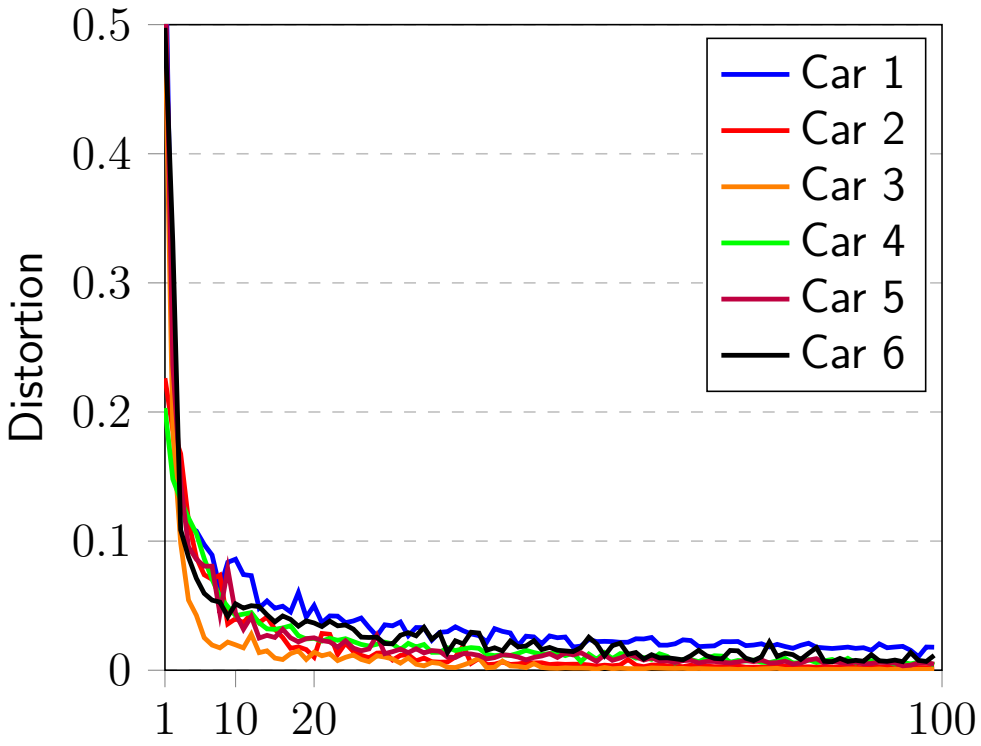

Figure 6-5: Distortion in KMeans clustering of car endpoints.

The KMedoids also had the downfall that if the user had meaningful destinations that were close to each other, it would find points in between that could lower the entropy of clustering, but didn't again match the actual destination labels we wanted.

These were helpful however in understanding what a typical "good" K value for the clusters is across cars. In particular, Figure 6-5, suggested that, with a rough "elbow method", $K = 10$ or $K = 20$ seemed ideal values. That furthermore means that most cars have around 10 to 20 meaningful destinations they frequently visit.

This suggested that a more specialized way of clustering could lead to interesting results into labeling the data. We however, took a different direction, and the idea would be to *encode* the location data in different ways by mapping it to a more coarse representation, that would effectively yield a different form of clustering.

6.4.4 Encoding Location

We implemented three encodings that we now present. The first two encodings are different ways of imposing a grid on the map, and mapping every latitude and longitude to the new discretize cell ids. The last one will entail considering the roads in the city as a graph, where every road is an edge, and every intersection a node.

L-Gridding Encoding

This encoding of the data involved simply truncates the latitude and longitude to L decimal places. This parameter effectively controls the size of the grid. The drawbacks to this encoding are that in different parts of the world 1 degree in latitude or longitude doesn't measure the same distance (because of the shape of the earth), and so the "cells" are not of equal size. Example of this gridding is shown in figures 6-6 and 6-7, and we call this type of encoding L-gridding.

It is worth noting that 1 latitude degree is roughly equal to 110.574 km, and thus 2 decimal places (L = 2) in precision yields roughly a 1km by 1km grid.

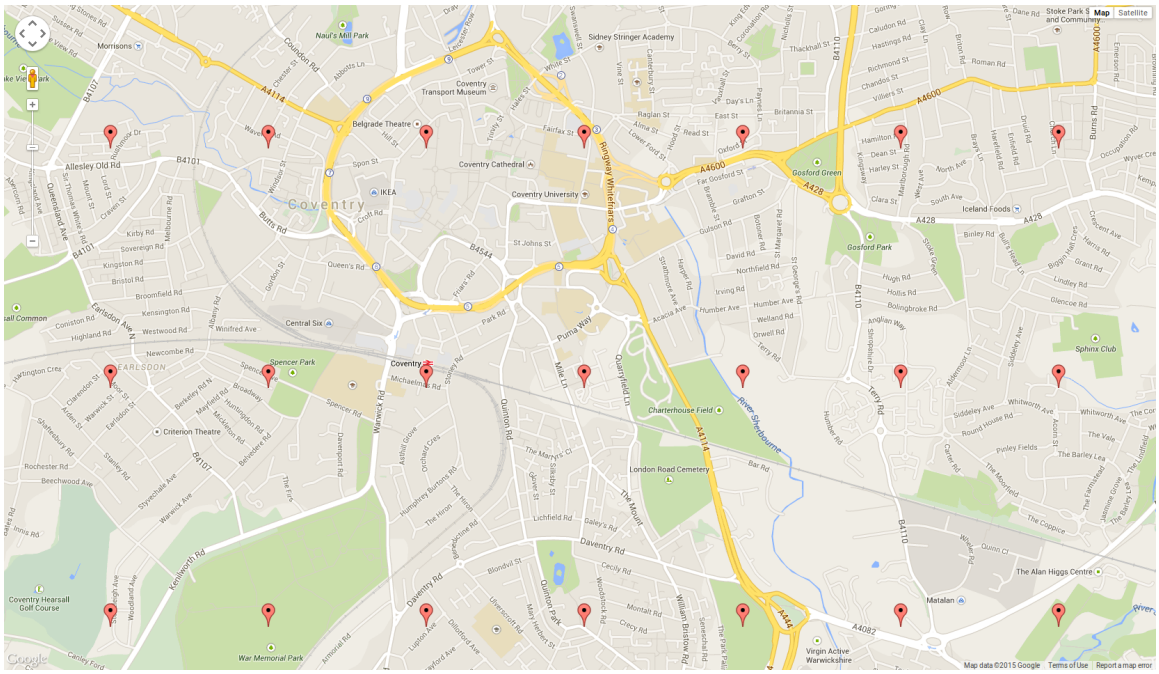

Figure 6-6: L-Gridding for L = 2

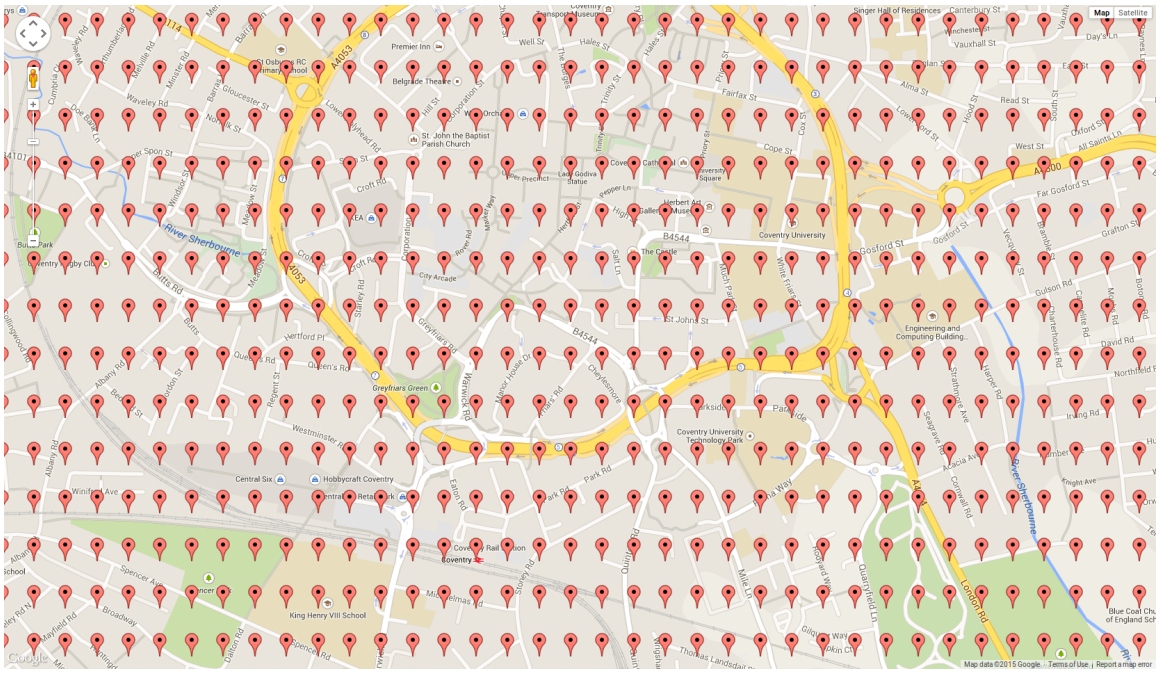

Figure 6-7: L-Gridding for L = 3

R-Gridding

The next encoding of the data is effectively equivalent to the one above, but with more control over the size of the grid by creating "squared" cells. Here, the parameter to control the grid was simply the side-length of a cell in meters. This way its easier to think and reason about and all the cells have the same width and height. We call this R-gridding and R the parameter of the meters in a cell edge. The imagery for R = 1000 and R = 100 is very similar to Figures 6-6 and 6-7, respectively.

Graph Representations: Node and Edge

Finally, the last encoding was to think of the map as a graph with respect to the roads. For this, we used an Open Source Project[1] by one of the Post Docs in our Lab, Ignacio Arnaldo. The project is called OpenStreetMap2Graph and does exactly that, it uses OpenStreetMap data (which is Open Source Map data) and converts it to a graph. It only supports a select number of cities, but among them is Birmingham, UK, which is the area where most of our cars drive around. A visualization of this

[1] http://ignacioarnaldo.github.io/OpenStreetMap2Graph/

graph representation is visible in Figure 6-8.

Figure 6-8: The city of Birminham in its graph representation. It contained 1,047,873 nodes and 1,106,965 edges.

With this data structure we can now map our raw latitude and longitude readings in two ways. One is to get the sequence of nodes traveled by the car, and the other is the sequence of edges. These representations will be very interesting under the light of the Blue Line in MLBlocks as they are a perfect fit for HMM Analysis.

To compute these representations first create a KD-Tree of all the nodes and linearly scanned the trip files finding the closest node in the graph to the latitude and longitude. For edges, we then scanned the newly created list of node-ids file, in search for consecutive nodes with edges in between them. We acknowledge this is an approximation, but one that can yield interesting results nonetheless.

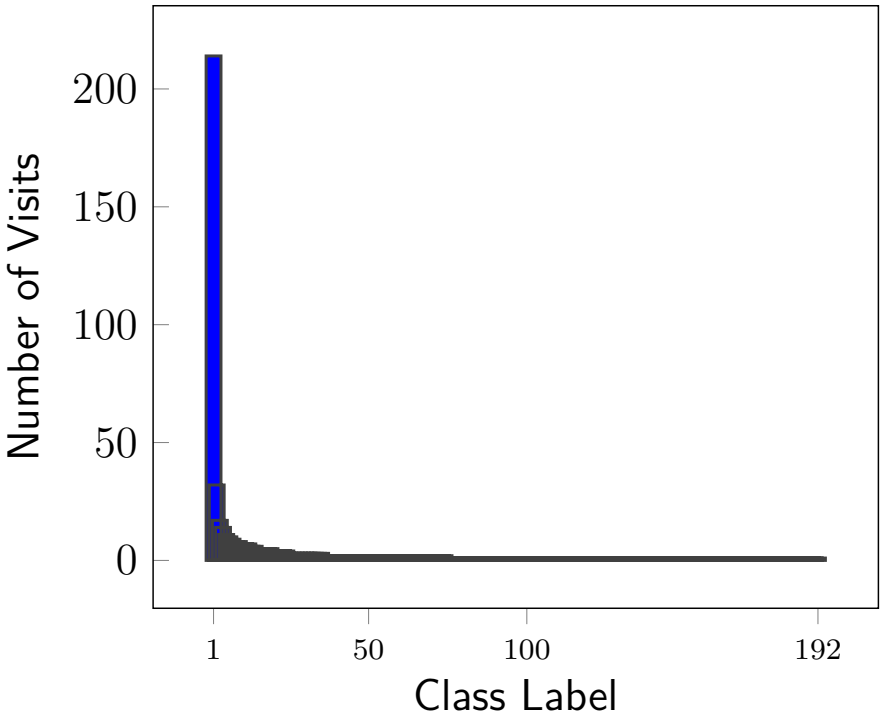

Figure 6-9: Labeling of trips for Car 1 with no clipping.

Gridding Results

The results of this gridding procedure were captured by histograms representing the "frequency dictionary" or binning of the endpoints for a given car. These looked like the one in Figure 6-9.

The results brought an idea that refined our problem definition slightly. That is, the frequency of visits has somewhat of an exponential decay, and in the case of 6-9 around 150 locations are visited less than 5 times. So this made us question again, how realistic it is to predict these destinations. In particular, we thought that if we would "clip" such a graph, and amount all the locations after some top locations K to a special "not top location" class, this would improve the prediction capabilities and result in a better product.

Thus, we decided to introduce the new parameter K to represent the number of locations we want to predict. These locations will be the most frequently visited locations (which we also call "top locations"), and this will in turn yield a $K + 1$ multi-class problem since we will lump together all **not-top-K** locations to a single

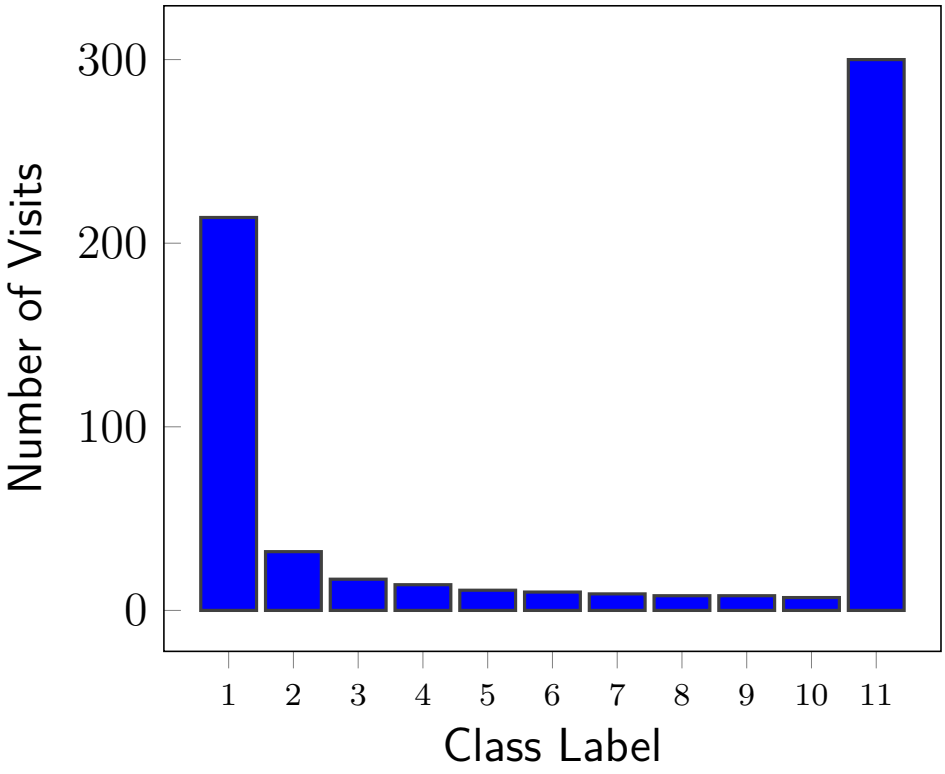

Figure 6-10: Labeling of trips for Car 1 with K = 10 clipping.

additional class.

The larger the K the harder but more interesting the problem. With this parameter K the different histograms are represented by the Figures from 6-10 to 6-15.

Further investigation of these, effectively "clusters", demonstrated that the class 1, would always correspond to a residential area, and class 2 and 3 universities/schools and workplaces. We redact examples for the sake of privacy.

Having these top K "cells" be labels, we can now label every trip with the corresponding label of its destination cell, and are ready to do the final step of feature engineering and creating our MLBlocks Foundational Matrix.

6.4.5 Feature Engineering

We decided to start by doing the special case of the MLBlocks Foundational Matrix in which there is a 2D Design Matrix per car, where the rows are Trips, and the columns are trip features. We computed a total of $8 + K$ features, without counting

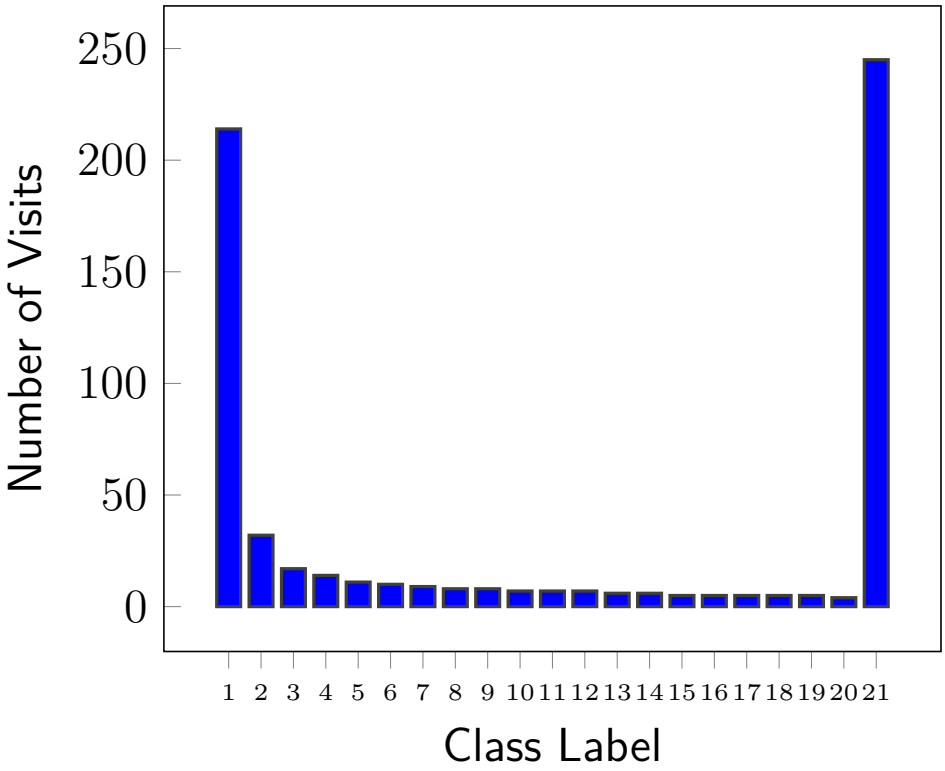

Figure 6-11: Labeling of trips for Car 1 with K = 20 clipping.

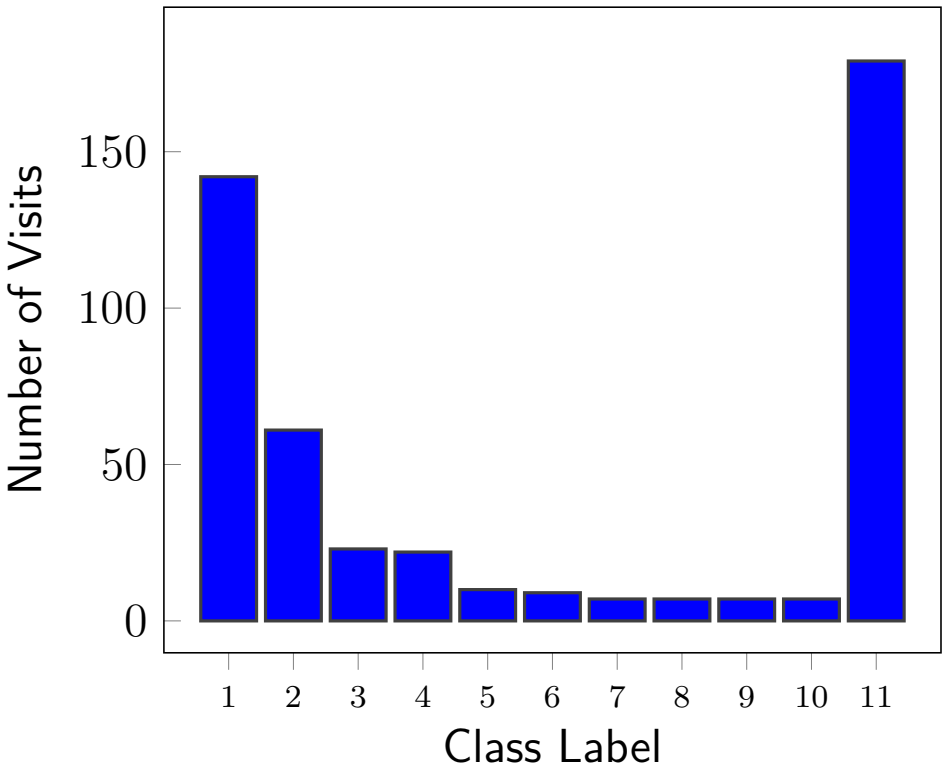

Figure 6-12: Labeling of trips for Car 2 with K = 10 clipping.

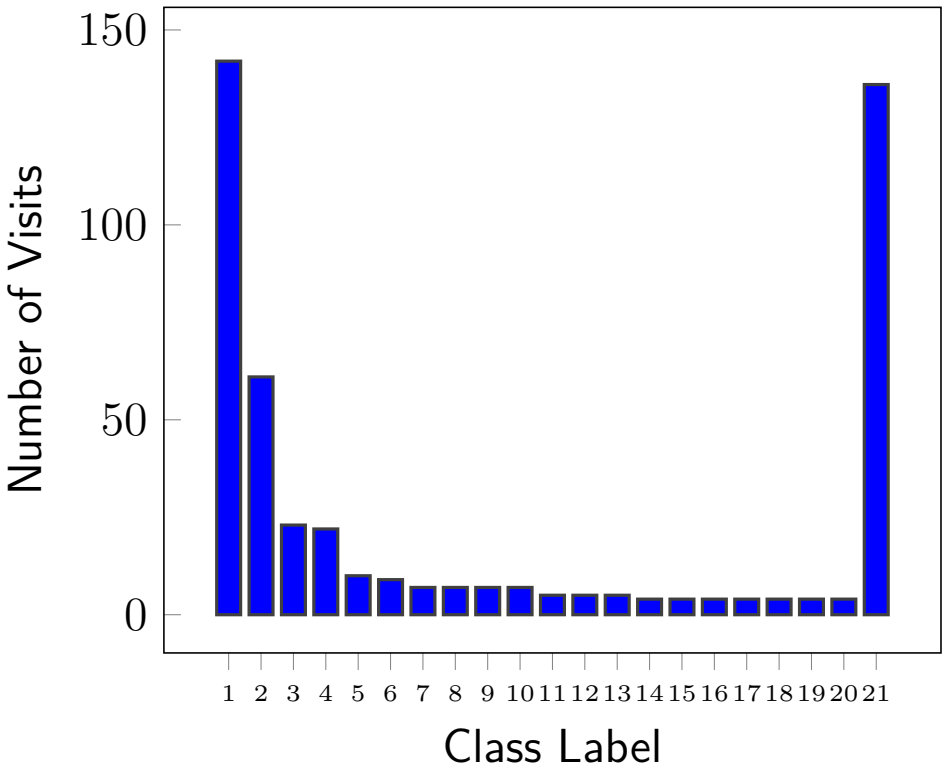

Figure 6-13: Labeling of trips for Car 2 with K = 20 clipping.

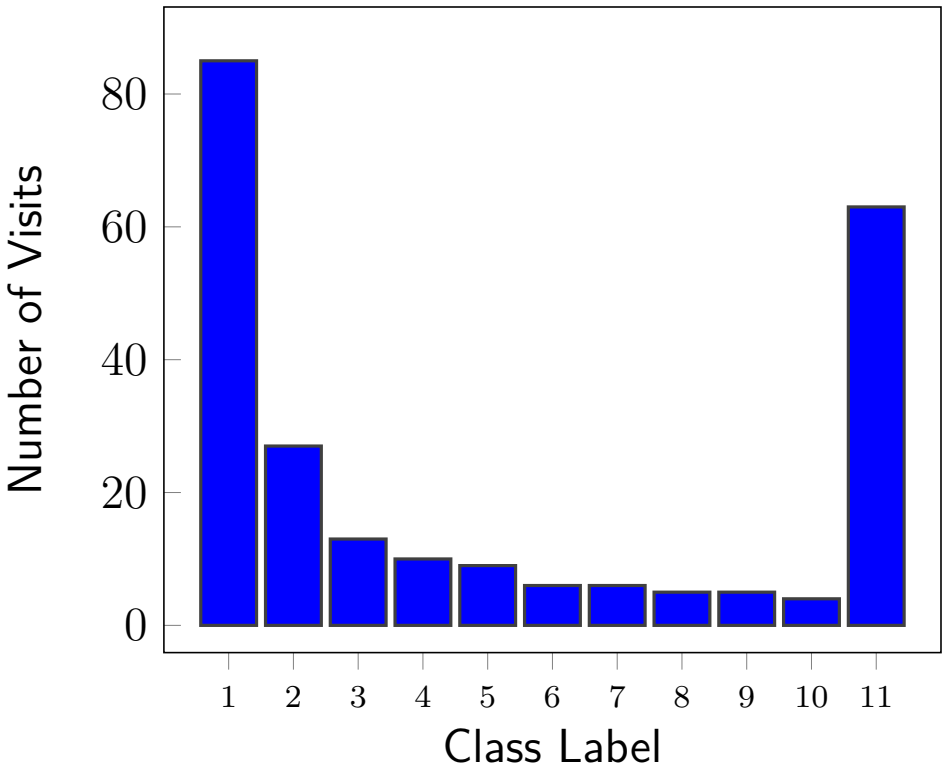

Figure 6-14: Labeling of trips for Car 3 with K = 10 clipping.

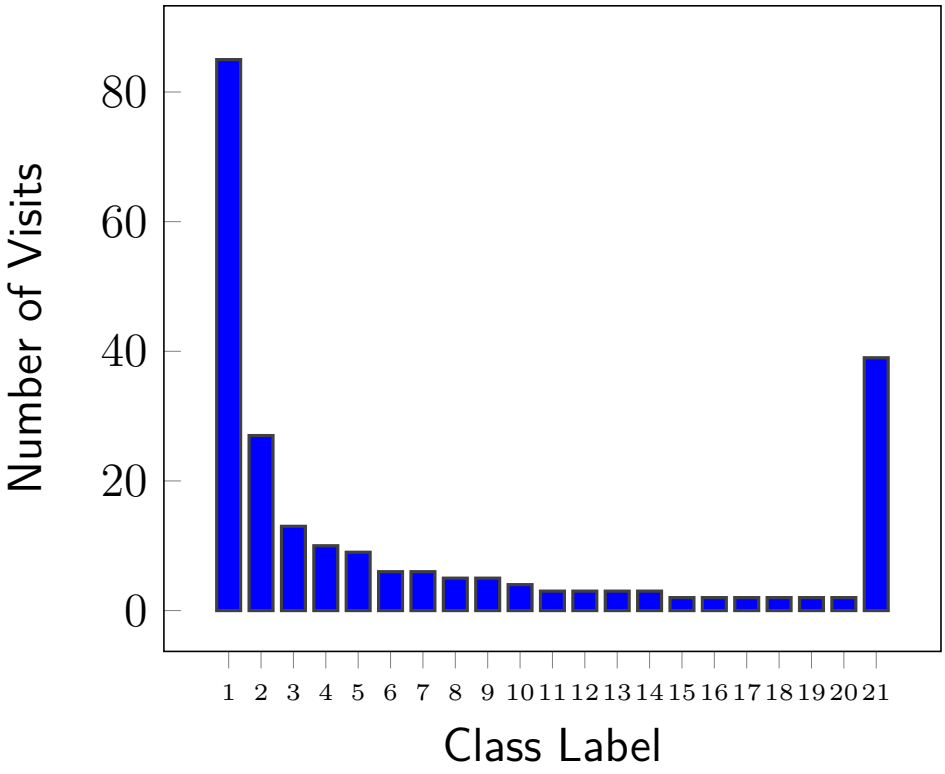

Figure 6-15: Labeling of trips for Car 3 with K = 20 clipping.

the label. To simplify our procedure we only computed these using the R-gridding representation, but the transformation with the other representations should be clear.

Start Location Features

- x_1: The x coordinate of the cell where the car was turned on.
- x_2: The y coordinate of the cell where the car was turned on.

Time Features

- x_3: Day of the Week. This is an integer from 0 to 6 inclusive, where Monday is 0 and Sunday is 6.
- x_4: Month. This is an integer from 1-12 inclusive, corresponding to the month in which the trip is happening.

- x_5: Day of the Month. Ranges from 1 to the number of days of that month in that year.
- x_6: Hour of the Day. This is the hour when the trip is starting, ranges from 0-23 inclusive.
- x_7: Minutes of the Hour. Ranges from 0-59 inclusive.

Distribution Features

These were $K+1$ features that represented the conditional distribution on the top-K locations (and the special not-top-K location), given that the trip has started in this location. That is, for these features, we looked at all the previous trips that the user had taken that had started in this same cell. Then, we took the distribution of all the destinations in those trips. So effectively, every feature f_i is the proportion of times you have been to class i, of all the trips that also started in the current trip's location.

- x_i: Conditional Probability of going of going to class i given the start location of this trip. For integers $i \in [1, K+1]$

6.5 Results with MLBlocks

The feature matrices created above are a perfect fit for the Orange Line in the ML-Blocks software. We inputted them and achieved the following results.

6.5.1 Orange Line

Examining the R and K Parameters

We used two main parameters to phrase the problem of destination prediction. These were the size R of the grid, and the number of locations to predict K.

We did the variation of R on 10, 100 and 1000, and K on 10, 20 and ALL (meaning no-clipping; using all the unique destinations the car has been to). We fixed the car

in order to see better the variation of R and K, and averaged the results over 10 runs of the Logistic Regression of the Orange Line. This summary is shown in the following table:

Car 1	Clipping at 10	Clipping at 20	All Destinations
10m x 10m grid	80.42%	74.02%	2.43% (446 classes)
100m x 100m grid	78.94%	72.91%	43.91% (191 classes)
1000m x 1000m grid	67.88%	61.21%	48.83% (126 classes)

In order to put some perspective in these numbers is important to consider some sort of naive benchmark. In our case, a good benchmark comes from considering the classifier that always predicts the most common label in the data. This classifier would perform as follows:

Car 1 Benchmark	Clipping at 10	Clipping at 20	All Destinations
10m x 10m grid	81%	73%	3% (446 classes)
100m x 100m grid	48%	39%	34% (191 classes)
1000m x 1000m grid	36%	36%	36% (126 classes)

The high percentages in the 10m by 10m with clipping suggested such problems are trivial, since the class that corresponds to "not-a-top-k" location is too big. That is, with such a high precision (10 meters), most top-location cells have only about 10 visits because of how distributed and segmented the space is.

With this benchmark however, we see big improvements across the 100m by 100m grid. This result also matches the intuition that 100m by 100m cells can better capture the size of a house and the most frequent parking spots (unless the parking spaces are big). This suggests this grid is worth exploring further, since although the 1000m by 1000m prediction problems also show the behavior, they are less interesting to predict.

Another interesting note is that for this car, the benchmark exhibits a 36% across all clippings in the 1000m by 1000m. This happened because for this particular car, the "home" location (most visited 1000m by 1000m cell) accounted for 36% of the visits, and so is not affected by clipping.

Finally, it is interesting to see how the clipping strategy makes the accuracies decrease as a function of R, while without this the numbers increase. This happens because the introduction of the "not-top-k" class works against R. The smaller the R, the easier the problem is with the "not-top-k" class; with a larger R the "not-top-k" class has less effect. Another way to think about it is that, the stronger this clipping is, the classifier has less choices (and chances) to be mistaken.

Orange Logistic Regression

Now that we have a rough picture of the behavior of R and K, we decided to explore further using the complete power of MLBlocks by examining all the cars under the 100m by 100m grid. The results are as follows:

K	Car 1	Car 2	Car 3	Car 4	Car 5	Car 6
10	77.35%	69.09%	64.85%	80.20%	65.86%	79.84%
20	73.54%	64.61%	62.57%	76.28%	59.05%	72.63%
ALL	44.81%	46.36%	53.85%	43.51%	46.37%	44.49%

Where its important to note that with no K, this is a multi-class problem with a very large number of classes. In particular, the cars had 192, 111, 54, 168, 88, and 137 classes respectively (in the same order as in the table). Therefore accuracies of 40% are not trivial since the random guessing in this setting would yield less than 1% accuracy.

Orange Delphi Cloud

Doing the same analysis but levering the complete power of Delphi to search over the 11 algorithms listed in 4-3. The experiments where ran by giving Delphi a budget of 100 learners; combined with an MLBlocks automation script we wrote, we were able to compile these results in less than an hour. The results are summarized in Figures 6-16, 6-17, 6-18, and 6-19.

	Algorithm	Test Performance
Car 1	k-Nearest Neighbors	0.8412
Car 2	Deep Belief Networks	0.7622
Car 3	Logistic Regression	0.8142
Car 4	Logistic Regression	0.7972
Car 5	Random Forests	0.7500
Car 6	Logistic Regression	0.8837

Figure 6-16: Performance for K = 10.

	Algorithm	Test Performance
Car 1	Passive Aggressive	0.7989
Car 2	Passive Aggressive	0.7552
Car 3	Passive Aggressive	0.7571
Car 4	Passive Aggressive	0.8310
Car 5	Stochastic Gradient Descent	0.7241
Car 6	Logistic Regression	0.8062

Figure 6-17: Performance for K = 20.

	Algorithm	Test Performance
Car 1	Passive Aggressive	0.5925
Car 2	Passive Aggressive	0.5700
Car 3	Passive Aggressive	0.6000
Car 4	Passive Aggressive	0.5675
Car 5	Extra Trees Ensemble	0.5775
Car 6	Passive Aggressive	0.6589

Figure 6-18: Performance for no K. That is, predicting ALL destinations.

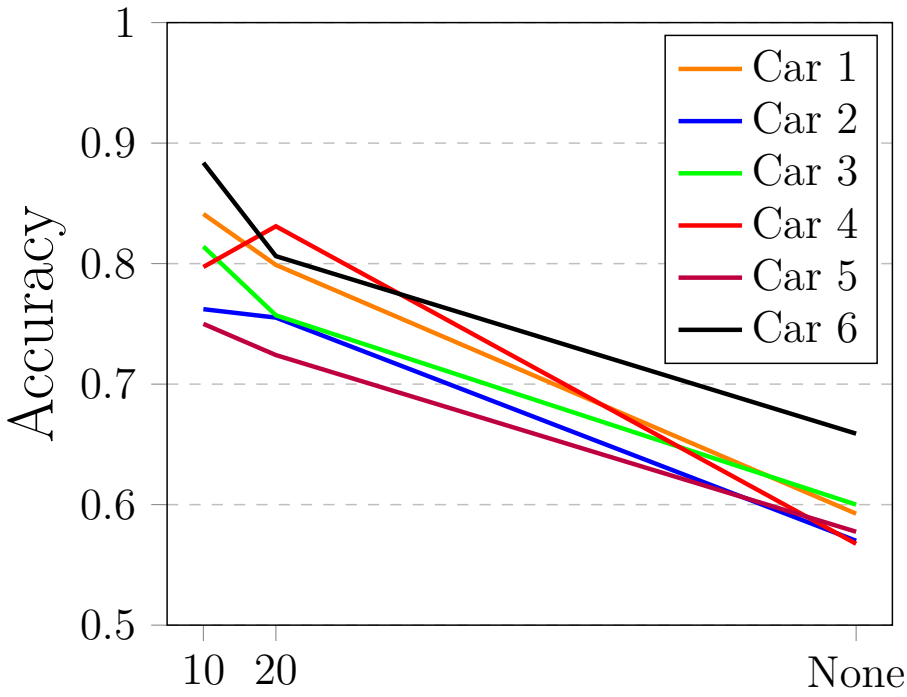

Figure 6-19: Performance of cars as a function of K. The right end of the graph marks no-clipping; note however that different cars have different number of labels in such scenario.

Analysis

The above results match the notion that with increasing K (number of labels), the problem becomes harder. This indeed happens as the number of classes the classifier is predicting is much higher, and it can't easily use the fall-back "not-top-K" class. It is actually very encouraging to note that we can achieve around 60% accuracy for problems of more than 100 classes, around 200 at times.

Another interesting note is that increasing K from 10 to 20, didn't affect much much the prediction scores. In general, the average difference in accuracy was about 4%. This seems like a good trade-off, considering that doubling the amount of destinations we support in the predictions lowered the accuracy less less than 5%. If tolerable, it would be interesting to see performance at say $K = 30$ and compute the percentage of the trips that fall under the top 30 locations. It might indeed be the case, that for most cars 30 can definitely include all the meaningful locations to predict, while after that most locations are very distinct and might feel random; which

furthermore poses the question of how predictive are these destinations any how? Is it possible that we can predict say a trip to a coffee shop you randomly wanted to try, but most likely never go again?

It is also encouraging to see that there was little correlation of performance scores with amount of data we had from the cars. That is, the car with most data, Car 1, performed in the middle of the pack, and the cars that performed best and worst are not clearly in any end of the spectrum of amount of data. This might softly indicate that the Figure 6-19 is showing the variance under a broader overall predictive bound on destination.

Future Work

The overall results also suggests that with more sophisticated clustering or as simple as increasing the size of the grids from $R = 100$ to $R = 200$ might yield better results.

In the same veins, the complete above analysis was merely done for the grid-encoded data. Another direction that is still under exploration is the graph representations with MLBlocks. These are a great fit for the Generative Modeling Line in MLBlocks as they inherently have a timeseries form, since one can see the data as a *sequence* of edges or nodes, which again are a great fit for the Blue Line. In particular, the Foundational Matrix in this setting can be constructed by having the entities be the different trips; and the features can be simple (timestamp, edgeid) or (timestamp, nodeid) tuples. Here we would be leveraging completely the notion of a **Data Slice** as the distance in time between the different samples is not necessarilly the same.

As a small preview of this work, we include the corresponding "frequency dictionary" on the space of all nodes and edges in Birmingham, UK.

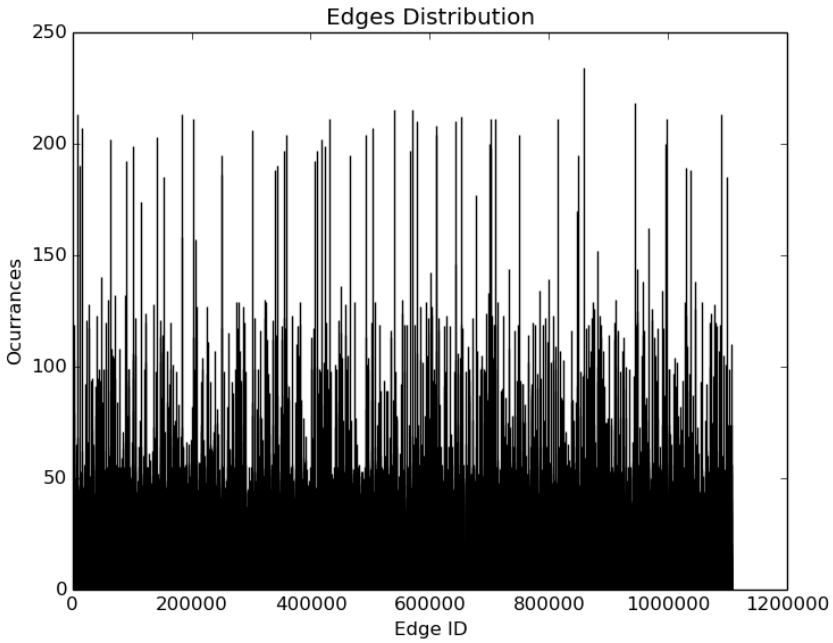

Figure 6-20: Frequency histogram of Edges for a given car.

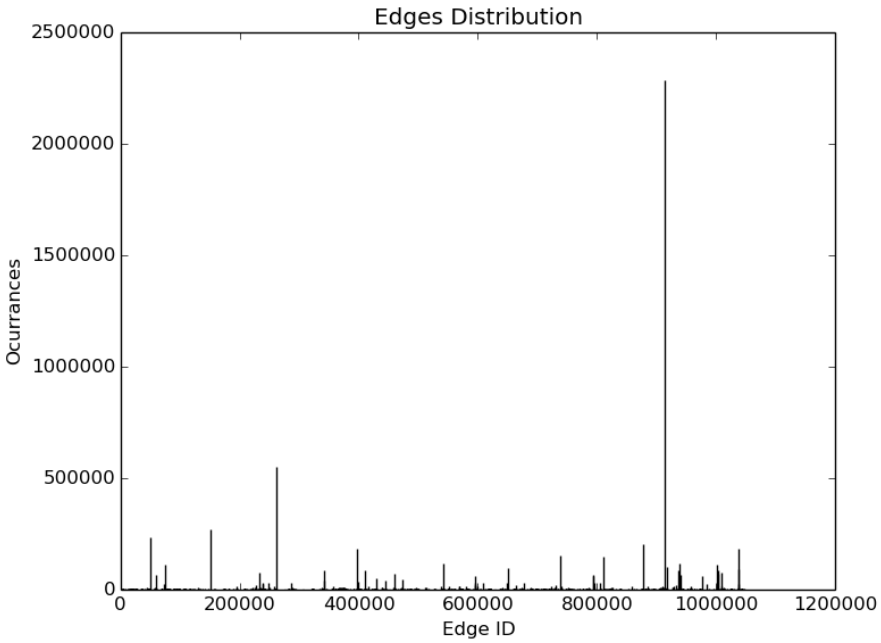

Figure 6-21: Frequency of nodes for a given car.

Chapter 7

Conclusions

In this thesis, we have presented the MLBlocks software from end-to-end, and given two real world examples of how it can be used. We showed how MLBlocks can help the data scientist improve his workflow considerably by removing the complexity of the data modeling phase and permitting the data scientist to focus solely on feature engineering.

7.1 Summary of Results

We have shown how virtually every data science problem can be mapped to the MLBlocks Foundational Matrix, a data structure that is reusable for the many techniques in MLBlocks and is more general than the standard Design Matrix. We have also included a list of steps for extending MLBlocks functionality to additional high-level techniques.

MLBlocks was presented as a strong mechanism to examine prediction performance of time series data as a function of how far in the future one would like to predict and how much data from the past one should use. This was particularly apparent in the MOOC example, where we also pointed out how to use synthetic features to encode possibly sensitive information without losing too much predictive power. All these experiments were automated using MLBlocks's ConfigManager class, and were able to generate comprehensive reports overnight.

With the Destination Prediction problem, we showed, when discriminative modeling is desired, MLBlocks leverages the capabilities of Delphi to return the best-tuned model from a space of 11 classification algorithms. This again posed MLBlocks as a very powerful tool, as the data scientist can take any standard Design Matrix and get the best performing algorithm along with its parameters, in a matter of minutes. We also automated this analysis, to get results for 18 different problems (6 cars and 3 K values) in less than an hour.

The MLBlocks Recipe

One of the main contributions of this thesis, is the creation of an efficient recipe of work for the data scientist; that permits him/her to focus on Feature Engineering, arguably the phase where most of the intuition and domain knowledge is necessary.

This recipe can be summarized as follows:

1. Extract relevant data from raw format.

2. Think about how to map the data to the Foundational Matrix.

3. Perform a Feature Engineering Pipeline that produces the Foundational Matrix.

4. Use MLBlocks to search the modeling space of models with ease

The power of having such a recipe is that, its only a slight change in the current overall workflow of the data scientist (bullets 1 - 3 still happen with the less general 2D Design Matrix), but it makes the search of techniques much easier. Now, instead of the data scientist have to formulate his feature matrix differently to use HMMs, experiment with synthetic features, or use simple discriminative models; he/she can just formulate his data once and use MLBlocks to explore all these techniques with simple changes to a configuration file.

7.2 Future Work

This thesis also inspired much work that can be very interesting to pursue. Many of which revolve around improvements to the MLBlocks Software itself. We discuss some of these.

Extending to Additional Blocks

It can be very interesting to implement additional functionality as new "Blocks" or "Lines" in MLBlocks. These could very well implement functionality like Clustering, Clusterwise Classification or any other imaginable high-level technique.

Improvements to Current Blocks

Some Blocks in MLBlocks can be extended further in functionality. For example, the Blue Line of Generative Modeling, only supports a single type of HMM which supports only discrete values. One can very well envision MLBlocks to also support continuous HMMs, or even more generally different types of Bayesian Networks analysis; all of which could be parametrized, by a single "mode" parameter similar to the Discriminative Modeling Line.

Also, the lead and lag notions in MLBlocks follow a "sliding-window" approach, which yields many prediction problems. One could make this window fixed so that there is only one prediction problem per entity (use the first lag samples to predict lag+lead).

Parallelization of Computation Nodes

To make MLBlocks highly scalable, one can very well envision parallelized instantiations of the computational nodes in MLBlocks. Some of these are already parallelized via the spawn of multiple processes in the host computer like the Forming of HMM features and the HMM Predict. But one could take this further and consider distributing the computation across a cluster of computers in the cloud.

Sampling MLBlocks Workflows

MLBlocks, being highly parameterizable, hints on the idea that one could *learn* how to optimally set these parameters. Consider sampling the space of MLBlocks tunable parameters and using its output to guide the sampling search. This in essence is to abstract the learning even higher and use MLBlocks programatically to search the optimal settings of parameters at the "Workflow" level; thus effectively yielding the results that Delphi achieves at a discriminative modeling setting of finding the best model, but on a much broader space, where the result yields the best high-level technique and parameters for such dataset.

This work is very exciting, and we have already started some development on this endeavor as well. MLBlocks already writes its best score to a file, that a library created by one of the members of the ALFA Lab can read to feedback a procedure that tunes MLBlocks parameters by modeling it as a Gaussian Process.

Appendix A

Figures

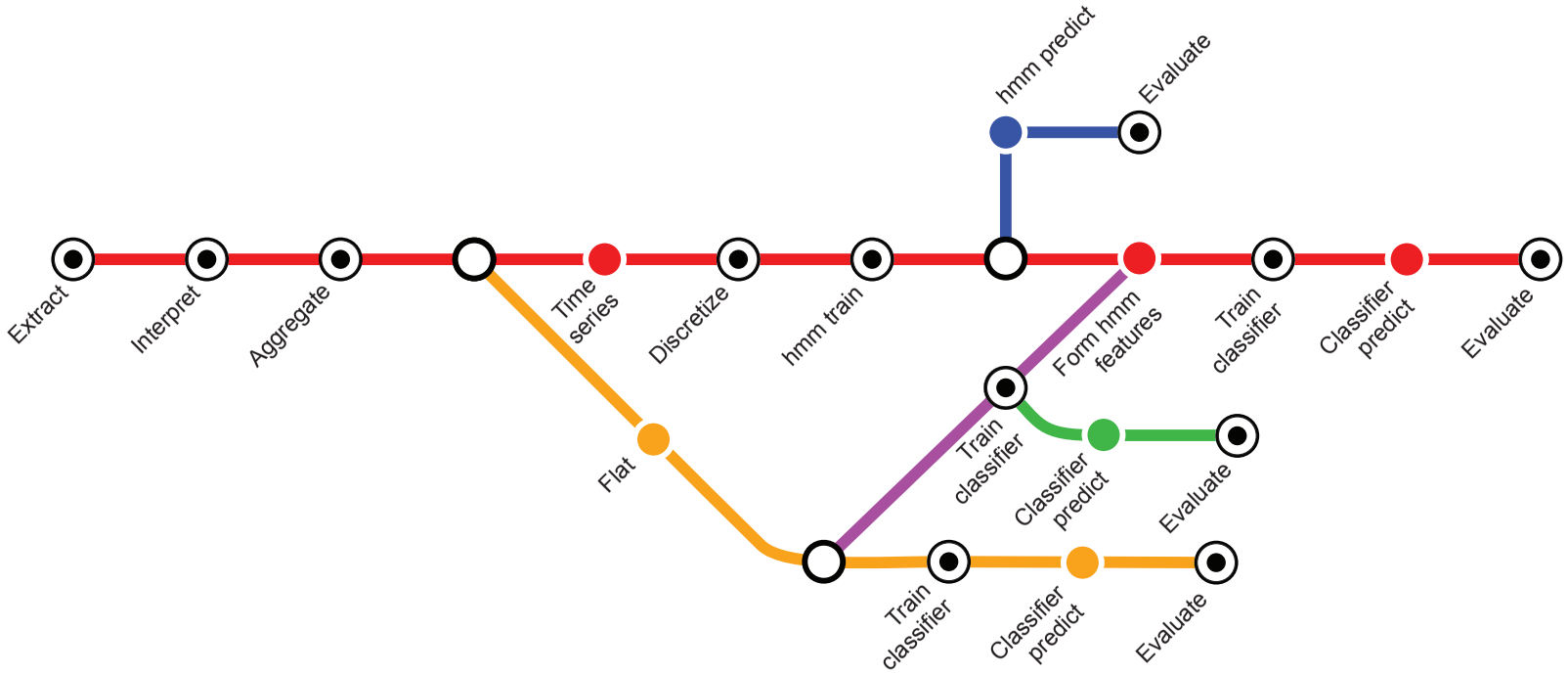

Figure A-1: MLBlocks Diagram

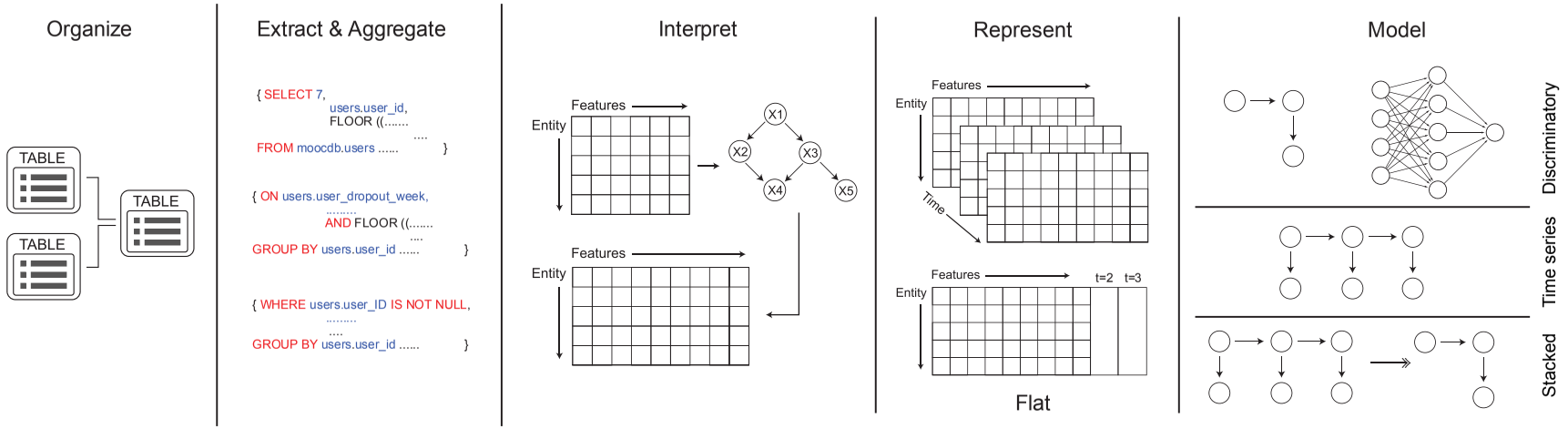

Figure A-2: Data Science Pipeline

```
[MLBlocks]
line=Orange
lead=0
lag=1
test_split=0.3
num_processes=12
debug=False
[Data]
data_path=example_data/
data_delimiter=,
skip_rows=1
label_logic=lambda x: x[0]
[TrainClassifier]
mode=LogisticRegression
learners_budget=5
restart_workers=False
ping_back_seconds=60
[Discretize]
binning_method=EqualFreq
num_bins=5
skip_columns=None
[HMMTrain]
hmm_type=Discrete
num_states=4
max_iterations=10
log_likelihood_change=.0000001
[Evaluate]
evaluation_metric=ACCURACY
positive_class=0
```

Figure A-3: Example Config File in MLBlocks.