ArchMLP: Machine Learning Platform

Aumit Leon
Middlebury College
Middlebury, VT
aleon@middlebury.edu

Mariana Echeverria
Middlebury College
Middlebury, VT
mecheverria@middlebury.edu

ABSTRACT

As Machine Learning research continues to expand, the tools available to practitioners grow in sophistication, abstracting away the rigorous implementation details that often define the development of complex mathematical models. While powerful, many of these tools are focused on model development and enhancement, giving little attention to the question of model deployment. While model development is an area of active research, deploying a trained model to production requires a clearly defined pipeline that manages assumptions and underlying conditions between the various stages of the model's lifescyle.

ArchMLP is a Machine Learning pipeline that creates an *analytical* setting and *operational* setting. The *analytical* setting focuses on model development and enhancement, while the *operational* setting is focused on model deployment. ArchMLP is designed as a microservices architecture via a Docker containerization schema [1]. The architecture of the platform ensures that functions are sufficiently isolated and maintains the clear functional contracts between various platform components necessary to guide a dataset from preprocessing to deployment. The functions that define ArchMLP's components are themselves deployed python modules, such that users can access our platform's functionalities outside of our architecture and run our functions in tandem with existing popular Machine Learning and data wrangling libraries. Our proof of concept illustrates ArchMLP's ability to take a dataset, run a specified set of generalized preprocessing steps, train a model, and deploy that model such that users can query it with parameters and receive predictions.

Keywords

ArchMLP; Machine Learning Platform; Docker; Containerization

1. INTRODUCTION

As Machine Learning research has expanded in the last decade, Machine Learning practitioners have been inundated with various Application Programming Interfaces (APIs), tools, and libraries that facilitate the development of sophisticated models. Libraries such as TensorFlow [2], Keras [3], and scikit-learn [4] provide a layer of abstraction above the mathematical implementation details that fuel statistical and machine learning algorithms. As a highly complex field, the democratization of Machine Learning requires many more layers of abstraction, with more attention to the entire pipeline. While existing libraries and APIs facilitate model development, it is often unclear how to go from a trained model to a model deployed in production. While a trained model is useful, the lack of solutions that attempt to guide a model from inception to production restricts the accessibility with which Machine Learning applications can be developed and deployed.

Outside of Machine Learning, the development and deployment of applications is often simplified by well established libraries and frameworks– the lifescyle of software is clearly defined, simplifying the process of developing, deploying, and debugging applications. This simplicity is missing within Machine Learning platforms. Developing Machine Learning applications requires complex implementations and compute-intensive operations, followed by infrastructure specific challenges that must account for model representation, model serialization and deserialization [5], as well as API functionality that wraps around the model.

While most widely used Machine Learning libraries and APIs are focused on the development of the model itself, there are comparably less solutions that provide generalized support around preparing a dataset for model development. Not all datasets are created equal– depending on the source of the data, most datasets will require expensive cleaning before they are ready to fuel a model. The cleaning of a dataset can include operations that remove or fill in missing values, filter data based on specific conditions, add features, and encode data, among others. After a dataset has been preprocessed, it can be used to develop a model. While many libraries assist in developing models, extra functionality around evaluating models would allow practitioners to gain a better understanding of their model's performance, along with actionable steps that can be taken to improve the model's performance.

Deploying a trained model after training requires a high level of scrutiny. A malfunctioning API means users can't query the model, which is a critical failure in the Machine Learning pipeline. The API functionality that wraps around the trained model should facilitate the model's interactions with the user. The well defined contract between the user and the model API should function such that the *user* provides a set of *parameters* to the the model API in the form of a *query*; upon receiving the query, the API runs the query against the trained model using the provided parameters, and responds to the query with the relevant prediction. This process can happen either programatically or via a user interface– the API simply needs to facilitate HTTP requests between the user and the model API.

In combining the training and development of a model with its deployment, ArchMLP provides users with a platform that guides a fresh dataset from preprocessing to user ready predictions. The platform is designed to provide users with the ability to preprocess a dataset, train a model using that dataset, and then deploy that trained model to production such that users can query the deployed model for predictions.

2. PROBLEM STATEMENT

In recent years, various Machine Learning methods have grown in popularity as they've been applied to increasingly complex problem spaces. The development and application of statistical learning methods has produced a dichotomous environment in which the challenges associated with developing a particular model are independent of those in applying the model. While this isolation produces a number of benefits for Machine Learning researchers and engineers alike, it has negative effects on the underlying pipeline that sees a model from its inception to its deployment. A model is only as useful as its applicability, and a modern pain point of developing applications dependent on Machine Learning methods is the lack of communication between the *analytical* and *operational* settings. The *analytical* setting requires advanced knowledge of mathematics and statistical modeling methods, while the *operational* setting necessitates an understanding of modern software infrastructures– especially as they relate to the deployment of large-scale data ingestion and processing applications [6]. Researchers focused on developing highly accurate models aren't aware of the ways that their models can be deployed to production, while engineers focused on deploying models are often forced to do so in an ad-hoc manner, developing application and API specific drivers for each individual model that needs to be deployed.

The *analytical* setting of Machine Learning is focused on the development and refinement of models through the optimization of some single metric or ensemble of metrics (i.e Accuracy, F1-Score, etc. [7]). While every problem space and associated dataset is unique and comes with a distinct set of challenges, certain practices could be generalized across domains and datasets. For example, while certain libraries such as *scikit-learn* [4] and *pandas* [8] provide functions that let users perform a combination of well defined tasks for preprocessing data, Python lacks a robust API that can automatically facilitate preprocessing via a set of well defined, sufficiently generalizable functions. The functions should be well defined in that they should make few assumptions on the format of the data, and should be generalizable so as to optimize for applicability across problem spaces and datasets. With few exceptions, preprocessing is almost always a required step in the development of Machine Learning models. Datasets are often messy, and require various forms of clearning before they are ready to be fed into analytical models. The *analytical* setting of our Machine Learning platform is cognizant of these requirements, and implements a modular preprocessing component packaged as a python module.

While distinct, the implementation of the *operational* setting is dependent on the assumptions and outputs of the *analytical* setting. Whereas the *analytical* setting is focused on the steps required to develop a model, the *operational* setting focuses on how to apply that model to real world applications. Given a trained model, this setting should be able to *interpret* the model, and package it into an API that allows users to interact with the model by providing parameters and receiving predictions. Because this setting is highly dependent upon model representations, traditional Machine Learning pipelines fail to recognize this interdependency, necessitating the development of ad-hoc solutions for individual deployments. The operational setting has a number of unique infrastructural challenges associated with it, including how to store a trained model, how to expose it to users, and how to manage the underlying dependencies required to support the prediction server.

The *analytical* and *operational* settings work in tandem to provide data scientists, machine learning practitioners, and engineers with the ability to develop models quickly, and then deploy those models to production with high confidence. In packaging the components of our platform into python modules, we get the portability of the Python Package Index (PyPI) [9], with the added benefit of users being able to use our module in parallel with other popular Machine Learning tools in python. In prioritizing modularity, our proposed microservices architecture containerizes each distinct setting and subcomponent so as to to create a clear architecture that can support the integration of more modules, and more functionality.

3. RELATED WORK

There are many platforms and libraries that facilitate model development such as Tensorflow [2], Keras [3], and scikit-learn [4]. Our analytical python modules were developed on top of functionality provided by pandas [8], numpy [10], and scikit-learn [4]– these libraries provide high level APIs for working with datasets and numerical data, and serve as the industry standard when it comes to working with data in python. The API design of these libraries was also used in the development of ArchMLP's proof of concept preprocessing module, such that operation decomposition attempts to combine multiple functions in a way that generalizes across Machine Learning use cases and applications.

While there are few open source, free to use platforms that provide the combined functionality of training models and deploying them to production, there has been promising research in this area, such as *MLBase* [11] and *MLI* [12]. *MLBase* is a a machine learning platform and *MLI* is the associated API that introduces high level differentiable programming abstractions that make it simpler to develop models. In our work, we attempt to recreate and build upon the ideas presented within MLBase by designing a microservices architecture that obviates the need for a singular API that connects all distinct containerized components.

The design of ArchMLP's microservices architecture was built upon the foundational ideas presented by Josh Wills at Midwest.io 2014. The perspective of moving "*From the Lab to the Factory*" [6] provided clear insight into the steps that industry leaders were taking in order to develop Machine Learning platforms in-house. ArchMLP's infrastrucrueal schema is based upon the *analytical* and *operational* settings that Wills' describes in his talk. We build upon these ideas by containerizing each component and implementing proper decomposition such that operations and their dependencies are isolated via containers, and communication between containers follows a strict, clearly defined contract of inputs and outputs.

Daniel Sarbe's talk at MesosConf 2017 provides further insight into how Apache Mesos was utilized in order to develop a private Machine Learning Platform at SDI [13]. Sarbe's talk and related work influenced ArchMLP's operational decomposition and containerization schema. Splitting the *analytical* and *operational* settings into containerized components helps remove interdependency issues, and keeps clean environments for both distinct settings.

In pursuing a component for model interpretability, LIME's generation of local explanations for Machine Learning classifiers provided insight into the ways in which we could potentially incorporate interpretability methods into our *analytical* setting [14]. While this component was not deployed as part of our proof of concept, it is part of our future work. Similarly *TFX* is a platform designed by Google that facilities the transition from the *analytical* setting to the *operational* setting [15]– while ArchMLP's infrastructure isn't based on TFX, the challenges encountered in the development of TFX have given insight into ArchMLP's future directions.

4. METHODS

4.1 Analytical Setting

The analytical setting is focused on generalized techniques for data preprocessing and visualization, model training, model evaluation and model interpretability. The data preprocessing component is a python module that supports standardized operations such as data cleaning, integration, transformation, and reduction. To build out this component, we generalized techniques used in data preprocessing libraries popular in R and Python. We explored functionalities provided by R's tidyverse [16] and ggplot2 [17], and python's scikit-learn [4], pandas [8], and numpy [10].

This component takes the location of a CSV file and returns four CSV files: train set with features (excluding the label column), test set with features (excluding the label column), train set with only the label column and test set with only the label column.

This component contains various methods of which three are required and six are optional. The three required methods allow the user to read the file by providing the source and split the data set into features and labels, and train and test sets. The other six methods serve different purposes: inspect entries of the dataset, remove desired variables, filter the data according to specific conditions, add features, perform one hot encoding approach for categorical variables and impute missing values.

For the model training component, we created a python script that takes the two CSV train files generated by the data preprocessing component. It uses the python module argparse [18] to collect the files and other arguments that would be used to train the model. The only required arguments are the two CSV files. The remaining arguments are parameters that the scikit-learn model accepts to customize the way in which the model is trained. The initial release supports two models, Random Forest and SVM.

To evaluate the performance of the model, we used the Cohen's Kappa metric. This metric tells us how much better the trained classifier is performing over the performance of a classifier that simply guesses at random according to the frequency of each class [19]. We decided to use this metric instead of accuracy (which tells us the proportion of correct results achieved by a classifier) because it gives a more accurate result when dealing with a data set that contains skewed classes. To compute Cohen's Kappa we use pyhton's scikit-learn [20].

Lastly, we created a model interpretability component in R to provide the user with more insight about the model. We incorporated the option to create feature importance

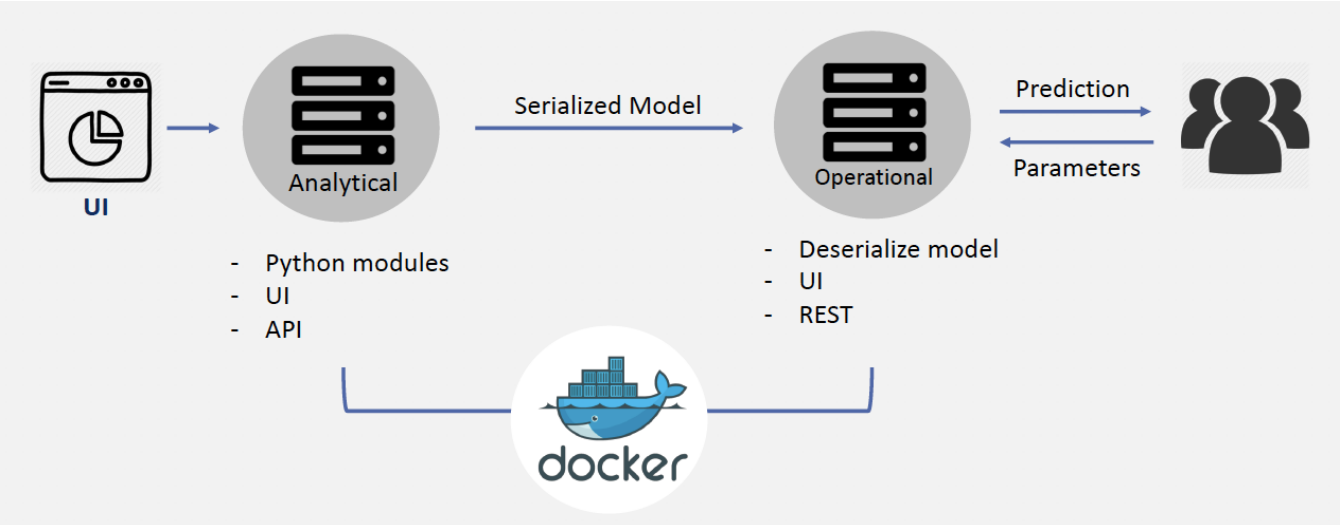

Figure 1: ArchMLP platform design. The UI allows a user to specify what operations they'd like to run which feeds into the *analytical* setting. This setting runs various operations and provides access to these individual operations via deployed python modules. The *analytical* setting produces a serializsed representation of a trained model that is consumed by the *operational* setting, which then serves as a prediction server that allows users to query the trained model with parameters and receive predictions in return.

graphs, partial dependence plots and a graph that shows the contribution of every variable to a final prediction. For these graphs, we used R's vip [21], pdp [22], and breakDown [23] packages, respectively.

All analytical modules are supported by CircleCI's continuous integration service [24], and new releases are managed by our own fork of the *python-semantic-release* module [25], configured to automatically upload new versions of our module to the Python Package Index. Continuous integration ensures that our primary branch remains clean, and adds room for automated tests for the integration of new features down the line.

4.2 Operational Setting

The *operational* setting is focused on taking taking a trained Machine Learning model and wrapping an API around it such that users can query the model with parameters and receive predictions based on the trained model as a response. The trained model must be in a serialized binary format that can easily be deserialized in order to accurately interface with queries. The premise of this design is presupposed by our containerization schema.

4.3 Containerization Schema

In order to facilitate the proper decomposition of our platform's functionalities, we designed a microservices architecture via a containerization schema that could ease the burden of cross conflicting dependencies for distinct components. The schema allows for each component to function independent of the other components; a modification or enhancement to any singular component shouldn't disrupt the functionality of the other components. The three components have strict contracts between one another that define inputs and outputs, and the internal implementation within each distinct container makes assumptions based on these contracts. For example, if a one container's contract with another is that it will always provide a CSV file to that other container, the code in the receiving container will assume that it will always receive a CSV. This schema can fail if contracts are broken– future work will focus on making this system design more robust.

Our architecture is composed of 3 distinct containers, as depicted in Figure 2. The first container is focused on data preprocessing, and sits within the *analytical* setting. This component accepts a user provided dataset, and produces 4 output CSVs– an x-train and y-train, as well as x-test and y-test. The test CSV's are used as cross validation when training the model in the downstream containers. This container hosts the preprocessing python module described in section 4.1, and includes all the dependencies the module requires to function. The dependencies are automatically installed when the image is built, and are available within the image when the container runs the preprocessing module. The container knows which preprocessing functions from our module the user wants to run, and executes these steps. After the module has completed running, it produces 4 output CSVs (x-train, y-train, x-test, y-test), and provides them as input to the train container, as depicted within Figure 2.

Similar to the *preprocess* container, the *train* container also sits within the *analytical* setting. The *train* image is built with all of the dependencies required to run our train script. The train component uses the x-train CSV and y-train CSV to train the model, and has the ability to validate against the x-test and y-test CSV files in order to improve the model. The model is trained using scikit-learn, and is saved using the python *dill* library [26]. After the model representation has been binarized by the *dill* library, it is passed to the third and final container.

The *prediction* container (depicted in Figure 2) sits in the *operational* setting. After receiving the binarized trained model representation from the *train* container, the *prediction* container wraps a RESTful API [27] around the model, facilitating interaction with web requests via the *flask* library in python [28]. Users are able to send POST [29] requests with a JSON file populated with parameter values, and receive a response from the *prediction* container that contains the prediction associated with the given parameter values.

4.4 Proof of Concept

For the proof of concept, we trained a Random Forest and SVM using the *Synthetic Financial Dataset For Fraud Detection* from Kaggle [30]. This dataset contains simulated mobile money transactions based on a sample of real transactions extracted from one month of financial logs from a mobile money service implemented in an African country. We decided to use this data because we were particularly interested in fraud detection, and this was one of the few datasets that we found given the lack of publicly available financial-services datasets.

The Random Forest and SVM models are trained using *scikit-learn*, because scikit-learn provides a simplified model representation. *Scikit-learn* provides a convenient way to save models for downstream consumption.

After the data has been preprocessed and the model has been trained, the binarized pickled representation of the trained model is called within a *flask* server. The *flask* server can be tested by creating a JSON file with parameter values, and sending a CURL request like the following:

```
curl –H "Content–Type: application/json" –X
    POST ––data @instance29.json http
    ://0.0.0.0:5000/predict
```

In this particular case, given instance29, the server predicts 0 (non-fraudulent transaction):

```
8–12–10 20:47:27 [INFO] 172.18.0.1 – – [10/
    Dec/2018 20:47:27] "POST /predict HTTP
    /1.1" 200 –
Preparing to predict..
2018–12–10 20:48:04 [INFO] Predicted 0
```

5. RESULTS

While the containerization schema discussed in section 4.3 touches on the ways in which the *analytical* and *operational* settings function in isolation, ArchMLP combines both settings in an effort to complete a generalized Machine Learning pipeline. The discussion of our proof of concept build in section 4.4 illustrated that ArchMLP is able to provide users with the ability to preprocess a dataset, train a model on that dataset, and then deploy that model to production. While the proof of concept is limited to the Synthetic Financial Fraud dataset, ArchMLP's design allows for this functionality to be generalized beyond the particular dataset and models presented in this paper.

5.1 Analytical Results

The analytical setting allows users to quickly and accurately preprocess and train models. While this paper focuses on a particular dataset and set of models, these findings are easily generalizable to other datasets and other models. In deploying various components of the *analytical* setting as python modules on the Python Package Index, we ensure that our users are able to use our API without having to access the architecture defined in the *operational* setting.

In addition to functionality, the *analytical* setting was designed to facilitate the inclusion of additional modules and components. In our discussion of model evaluation within the training component in section 4.1, the use of Cohen's Kappa is just one example of the ways in which features are easily added to ArchMLP's modular architecture.

5.2 Operational Results

The Docker containerization schema defined in section 4.3 allows ArchMLP to extend beyond traditional monolithic architectures to support greater portability and performance. In isolating various functions of the platform within containers, ArchMLP ensures that the form and function of the platform's components are aligned. In addition to portability, the operational design of ArchMLP ensures that if a single component of the platform fails, other parts of the platform will not fail.

The *operational* setting within ArchMLP is able to accept queries from users and respond with accurate predictions based on a trained model representation, as discussed in section 4.4. The platform wraps a RESTful API around the trained model representation, which allows users to query the trained model directly. Abstracting away the need to programatically produce predictions makes the development and deployment of Machine Learning models much simpler.

In addition to allowing users to interface with a trained model via conventional web requests, ArchMLP has infrastructural fault tolerance. For example, if the platform is currently hosting a trained model in the *prediction* container, a failure in the *preprocessing* component or *training* component will not be propagated to the *prediction* container– ArchMLP will continue to accept and respond to user queries. Upstream failures are only propagated if a particular contract still needs to be satisfied– i.e, if the *prediction* container doesn't have a model to serve and the *train* component fails– ArchMLP will fail. These types of errors are mitigated by optimizing for modularity and being able to tune and improve various components without directly impacting others– a byproduct of ArchMLP's containerization schema.

5.3 System Design

ArchMLP's microservices architecture brings traditional software development practices to the world of Machine Learning applications. Whereas many tools optimize for model development and enhancement, ArchMLP augments those capabilities by building on top of the containerzation schema discussed in section 4.3. ArchMLP's user facing API and libraries are deployed as python modules that can be downloaded and used outside of the platform, but when used within the platform, the modules take full advantage of being able to sit within a single container dedicated to satisfying all dependency requests specific to that particular module. This isolation ensures that dependencies never conflict, components can be deployed in isolation, and the component hierarchy is well defined.

Defining contracts between the *preprocess*, *train*, and *prediction* containers keeps the implementation clean, and ensures that assumptions are understood by all relevant components of the platform. A breech of contract between any 2 containers should cause the system to fail, and that is the current behavior of the platform. ArchMLP's contracts also allow for simple debugging: if the *preprocessing* container only produces 3 output CSVs as opposed to 4, then the *train* container's operations will fail. While the error would have been detected in the *train* container, the stack trace would note the insufficient input CSVs, which means

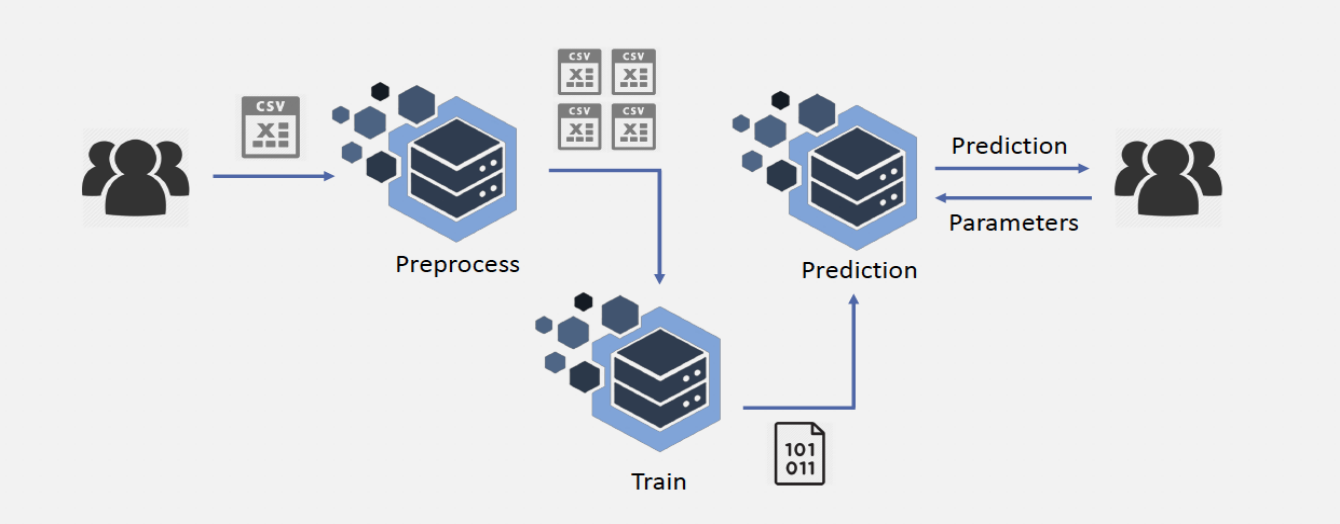

Figure 2: Docker containerization schema. The *preprocess* and *train* containers are in the *analytical* setting, while the *prediction* container is in the *operational* setting.

there was an issue in the *preprocess* component that needs to be fixed. ArchMLP was designed to be robust to change with room for more components and features– the system design facilitates this level of portability via a well defined containerization schema.

6. DISCUSSION

ArchMLP provides a clear pipeline from data preprocessing to model deployment. The *analytical* and *operational* settings are able to interface with one another through clearly defined contracts, and ensure that the functionality of individual components is isolated. Machine Learning libraries are complex tools with many shared dependencies– ArchMLP's containerization schema ensures that each component's dependencies are isolated. The portability of containers themselves provides more advanced users with the ability to deploy various components of the platform to different infrastructures at their own discretion. The *train* container could possibly be enhanced with GPU capabilities, and thus, should probably be deployed to a compute infrastructure that has GPUs available.

Containerizing software maintains its portability– the only requirement is that the underlying system has the Docker runtime installed. Thus, a user that wants to configure ArchMLP with their specific system has the freedom to do so, regardless of infrastructural details. In addition to launching the various containers for use, users can also directly install ArchMLP's deployed python libraries without using the containers– thus, ArchMLP's preprocessing module can be used in tandem with other popular Machine Learning libraries and frameworks.

ArchMLP's system design was intended to provide room for continuous improvement and development. Future work on the platform includes the development of a user interface that can interact with the *preprocessing* component, depicted in Figure 1. A user should be able to pick a dataset, and choose what preprocessing options they want to run on that dataset, where each option is a different function within ArchMLP's preprocessing module. The user should also be able to directly specify what model to train, and with what parameters. Similarly, future work should also include the development of a user interface that facilitates interaction between the *prediction* container and the user– the user should be able to specify parameters and query the model directly from the interface, and then receive a prediction promptly.

Beyond data preprocessing, model interpretability, automated model enhancement, and automated hyperparameter tuning are all areas of future work that could enhance ArchMLP's *analytical* setting. The *operational* setting should continue to focus on integrating various APIs and libraries. The development of a model serialization format specifically for ArchMLP or API specific drivers that can work with various model formats would benefit the platform by providing access to a wider range of model development tools. The system design should continue to isolate functionality between containers, and maintain the notion of well defined contracts between communicating containers.

7. REFERENCES

[1] Docker: Enterprise Container Platform https://www.docker.com/

[2] TensorFlow: An open source machine learning framework for everyone. https://www.tensorflow.org/

[3] Keras: The Python Deep Learning Library. https://keras.io/

[4] Scikit-learn: Machine Learning in Python. http://scikit-learn.org/stable/

[5] TensorFlow Basic Serving https://www.tensorflow.org/serving/serving_basic

[6] Josh Wills, *Building a Production Machine Learning Infrastructure*. Midwest.io, 2014.

[7] C. Goutte and E. Gaussier *A Probabilistic Interpretation of Precision, Recall and F-score, with Implication for Evaluation*

[8] Pandas: Python Data Analysis Library https://pandas.pydata.org/

[9] PyPI: Find, install and publish Python packages with the Python Package Index. https://pypi.org/

[10] Numpy http://www.numpy.org/

[11] T. Kraska, A. Talwalker, J. Duchi, R. Griffith, M. Franklin, M. Jordan, *MLbase: A Distributed Machine-learning System.* Conference on Innovative Data Systems Research, 2013.

[12] E. Sparks, A. Talwalkar, V. Smith, J. Kottalam, X. Pan, J. Gonzalez, J. Gonzalez, M. Franklin, M. I. Jordan, T. Kraska, *MLI: An API for Distributed Machine Learning.* International Conference on Data Mining, 2013.

[13] Daniel Sabre, *Building a Highly Scalable Machine Learning Platform Using Apache Mesos.* MesosCon, 2017.

[14] M.T Ribeiro, S. Singh, and C. Guestrin, *"Why Should I Trust You?" Explaining the Predictions of Any Classifier*, 2016.

[15] D. Baylor, E. Breck, H. Cheng, N. Fiedel, C. Foo, Z. Haque, S. Haykal, M. Ispir, V. Jain, L. Koc, C. Koo, L. Lew, C. Mewald, A. Modi, N. Polyzotis, S. Ramesh, S. Roy, S. Whang, M. Wicke, J. Wilkiewicz, X. Zhang, M. Zinkevich. *TFX: A TensorFlow-Based Production-Scale Machine Learning Platform*. KDD 2017 Applied Data Science.

[16] Tidyverse: R packages for data science https://www.tidyverse.org/

[17] ggplot2: https://ggplot2.tidyverse.org/

[18] argparse: Parser for command line options, arguments and sub commands https://docs.python.org/3/library/argparse.html

[19] *The measurement of observer agreement for categorical data*, Biometrics, 1977.

[20] Cohen's kappa: a statistic that measures inter-annotator agreement https://scikit-learn.org/stable/

[21] vip: Variable Importance Plots https://cran.r-project.org/web/packages/vip/index.html

[22] pdp: Partial Dependence Plots https://cran.r-project.org/web/packages/pdp/index.html

[23] breakDown: Model Agnostic Explainers for Individual Predictions https://cran.r-project.org/web/packages/breakDown/index.html

[24] CircleCI: https://circleci.com/

[25] Python Semantic Release: https://github.com/relekang/python-semantic-release

[26] Dill: Extension of python's pickle module for serializing and deserializing python objects https://pypi.org/project/dill/
[27] Rest: REpresenational State Transfer https://restfulapi.net/
[28] Flask: web development one drop at a time http://flask.pocoo.org/
[29] HTTP POST method: https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/POST
[30] Synthetic Financial Datasets for Fraud Detection: https://www.kaggle.com/ntnu-testimon/paysim1