Distributed Deep Learning Pipeline for Scalable Model Training

A Major Qualifying Project Report

submitted to the faculty of the WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Bachelor of Science by:

____________________  Dylan Fontana
____________________  Andrew Petit
____________________  Brendan O'Connor
____________________  Kartik Thooppal Vasu

Date: 3/13/2018

Approved By:

____________________  Professor Elke A. Rundensteiner, WPI, Advisor
____________________  Eric Gieseke, ACI Corporation, Mentor
# Table of Contents

Acknowledgements .................................................................................................................. 4

1. Executive Introduction ........................................................................................................ 5

2. Context .................................................................................................................................. 8

   2.1 Fraud .................................................................................................................................. 8

       2.1.1 Fraud and the Problems It Poses .............................................................................. 8

       2.1.2 Fraud Detection through Machine Learning .......................................................... 10

2.2 Technical Background ......................................................................................................... 13

   2.2.1 Hadoop ....................................................................................................................... 13

   2.2.2 Spark .......................................................................................................................... 13

   2.2.3 Livy ............................................................................................................................. 14

   2.2.4 Hive ............................................................................................................................. 15

   2.2.5 MongoDB .................................................................................................................. 15

   2.2.6 Machine Learning Libraries ....................................................................................... 16

2.3 Machine Learning Algorithms ............................................................................................. 18

   2.3.1 Decision Trees ............................................................................................................ 18

   2.3.2 Random Forests ........................................................................................................... 19

   2.3.3 Logistic Regression ...................................................................................................... 20

   2.3.4 Artificial Neural Network ............................................................................................ 21

2.4 Definition of Evaluation Metrics Used ................................................................................. 27

   2.4.1 Accuracy ..................................................................................................................... 27

   2.4.2 Precision ...................................................................................................................... 27

   2.4.3 Recall .......................................................................................................................... 27

   2.4.4 F1-Measure ................................................................................................................ 28

   2.4.5 False Positive Rate ..................................................................................................... 28

3. Proposed Solutions .................................................................................................................. 29

   3.1 A Production Ready Pipeline ........................................................................................... 29

   3.2 A Distributed Pipeline ..................................................................................................... 30

   3.3 A Hyperparameter Searching Pipeline ........................................................................... 30

   3.4 A Dataset Agnostic Pipeline .......................................................................................... 31

   3.5 A Pipeline of Many Algorithms ....................................................................................... 32

   3.6 A Performant Pipeline .................................................................................................... 32

4. Pipeline Design ...................................................................................................................... 34
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 Cluster and Pipeline Structure Overview</td>
<td>34</td>
</tr>
<tr>
<td>4.1.1 Cluster Structure</td>
<td>34</td>
</tr>
<tr>
<td>4.1.2 Pipeline Structure Comparison</td>
<td>34</td>
</tr>
<tr>
<td>4.2 Configuration, Storage, and Documentation</td>
<td>37</td>
</tr>
<tr>
<td>4.2.1 Introducing DotEnv</td>
<td>37</td>
</tr>
<tr>
<td>4.2.2 Storing Models to HDFS</td>
<td>38</td>
</tr>
<tr>
<td>4.3 Queue</td>
<td>39</td>
</tr>
<tr>
<td>4.4 Data Ingestion</td>
<td>39</td>
</tr>
<tr>
<td>4.5 REST API</td>
<td>41</td>
</tr>
<tr>
<td>4.6 Livy Service</td>
<td>44</td>
</tr>
<tr>
<td>4.7 Model Trainer</td>
<td>46</td>
</tr>
<tr>
<td>5. Deep Learning</td>
<td>47</td>
</tr>
<tr>
<td>5.1 TensorFlow and TensorFlowOnSpark Integration</td>
<td>47</td>
</tr>
<tr>
<td>5.1.1 TensorBoard</td>
<td>51</td>
</tr>
<tr>
<td>5.2 Hyperparameter Tuning</td>
<td>55</td>
</tr>
<tr>
<td>5.2.1 Implementation</td>
<td>56</td>
</tr>
<tr>
<td>5.2.2 Comparison</td>
<td>57</td>
</tr>
<tr>
<td>5.3 System Testing</td>
<td>58</td>
</tr>
<tr>
<td>5.3.1 Configuration Check</td>
<td>58</td>
</tr>
<tr>
<td>5.3.2 Pipeline Integration Testing</td>
<td>58</td>
</tr>
<tr>
<td>5.3.3 Running TensorBoard Instances</td>
<td>59</td>
</tr>
<tr>
<td>6. User Interface</td>
<td>60</td>
</tr>
<tr>
<td>6.1 User Experience</td>
<td>60</td>
</tr>
<tr>
<td>6.2 Visual Overhaul</td>
<td>63</td>
</tr>
<tr>
<td>7. Results</td>
<td>68</td>
</tr>
<tr>
<td>7.1 Pipeline Data Parallelism Scalability</td>
<td>68</td>
</tr>
<tr>
<td>7.2 Hyperparameter Tuning Tests</td>
<td>70</td>
</tr>
<tr>
<td>7.3 Parallel Training Tests</td>
<td>76</td>
</tr>
<tr>
<td>7.4 Flexibility Evaluation</td>
<td>76</td>
</tr>
<tr>
<td>8. Conclusion</td>
<td>79</td>
</tr>
<tr>
<td>8.1 Summary of Project</td>
<td>79</td>
</tr>
<tr>
<td>8.2 Team Experience</td>
<td>80</td>
</tr>
<tr>
<td>8.3 Recommendations and Future Work</td>
<td>81</td>
</tr>
</tbody>
</table>
9. References ..........................................................................................................................84
10. Appendix A: Installing the Pipeline .................................................................................90
    10.1 Setting up Hadoop and Ambari ..................................................................................90
    10.2 Setting up the Pipeline .............................................................................................92
11. Appendix B: Metadata Fields .........................................................................................95
12. Appendix C: Expanding the Pipeline .............................................................................98
    12.1 Adding a New Data Set to the Pipeline ....................................................................98
    12.2 Adding a New Metadata Field ..................................................................................100
    12.3 Adding a New Model Type .......................................................................................101
13. Appendix D: Data Flow within the Pipeline ..................................................................109
Acknowledgements

We would like to thank our advisor, Professor Elke Rundensteiner, for her support, guidance, and critical insight during this project. Additionally, we would like to thank our sponsoring mentor, Eric Gieseke, and our sponsoring company ACI Worldwide for the opportunity to take part in this project and the direction they provided in making the project a reality. Finally, we would like to thank those who have provided their insights towards the project from ACI Worldwide and from Worcester Polytechnic Institute for enabling this project. Our appreciation is boundless.
1. Executive Introduction

ACI and Existing Pipeline Background

ACI Worldwide, the Universal Payments company, is responsible for managing electronic transactions for over 5,100 organizations scattered throughout the globe. As such, the organization manages in excess of 14 trillion dollars in payments and securities each and every day (ACI Worldwide, n.d.). With this much money being overseen, some of the transactions are inherently fraudulent. To combat such transactions, 90% of online fraud detection platforms, ACI included, currently maintain millions of hard-coded rules to check for fraud, but these rules are not easily modifiable (Maruti Techlabs, n.d.). That is, they assume that every instance of fraud can be detected by a number of stipulations that remain consistent over time. In practice though, fraud is constantly adapting to bypass the rules that are put in place to prevent it. This combined with the fact that e-commerce is continuing to grow every year makes it increasingly time-consuming and costly to either update these rules to account for rapid changes in fraud or worse yet to manually check for fraud in larger datasets (Chaffey, 2017).

To alleviate these problems, machine learning models could be used to predict whether transactions are fraudulent without explicit hard-coded rules to check for fraud. These models work by taking large samples of pre-marked data to identify common patterns that exist in current fraud practices. With this information, the models can then be used to predict the likelihood that a current transaction is criminal (Mitchell, 1999). While useful, these models take substantial amount of time to create and train, therefore by the time a model is ready to be implemented, the dataset used to train the model could already be outdated (Wrobel-Konior, 2017). This makes the model potentially vulnerable to newer fraud attacks, leaving the initial issue of detecting constant change in fraud unaddressed.

WPI MQP 2016-2017

The previous MQP group worked on developing a potential solution to fraud detection utilizing machine learning algorithms. In addition, they also tried to substantially decrease the cost and amount of time these algorithms take to produce and train models. To do this, the group used a pipeline approach that allows users to choose, using an interactive interface, which machine learning algorithms to train on, which tuning parameters to use for training optimization, and which features to test. A comparison between a traditional static approach for producing machine learning models and the previous MQP group's pipeline approach can be found in Figure 1. Further, this system was set up so that deployment procedures could easily be implemented and automated in future revisions (Esco et.al, 2017).
Figure 1 Static model deployment (Left) vs. Pipeline deployment (Right). With the static approach, a model has to be created manually every time a user updates his/her input values. This new model is then deployed. With the pipeline approach, a request to update a model is sent to the pipeline which automatically updates the existing model and deploys the updated version (Esco et.al, 2017).

**Shortcomings of Prior Pipeline Project**

The previous pipeline had an excellent user interface that made it compelling to work with – especially receiving multiple iterations of review. Additionally, the architectural structure was present, providing a starting point for this project. Due to loss of data or improper version control, however, the pipeline was left in an incomplete state in need of fixing. The code backing the user interface had defects masked inside, so while not visual, caused problems later on. The usage of a CRON job as a means to transfer jobs from Queue to Spark was ineffective and non-portable but replaceable. The statistics about a model was not being reported correctly, presumably due to defects in computations. Perhaps the largest problem was the inability to run a model through the pipeline start to finish, certainly due to loss of data (and the non-portability of the pipeline due to many hard-codings of environment variables). So while overall the structure was present and the user interface superb, there was work left to be done to get the pipeline into a production ready state.

**Accomplishments**

While the prior group proved a pipeline approach adequately solved the financial cost of manually producing models using the traditional static approach, their pipeline was functionally limited and not ready for production. With this project, we have removed much of the inflexibilities, enabling the pipeline to be employed by ACI for production use cases. Additionally, through the usage of Hortonworks Hadoop, TensorFlow, and Apache Livy we have improved the performance of the pipeline by distributing the training of singular models (Data Parallelism) while training multiple models at the same time (Task Parallelism) resulting in speed increases upwards of 10x.
While achieving this, the team was able to achieve additional side goals such as the introduction of TensorBoard for visualization of model training, and the autotuning of model hyperparameters via the HyperOpt library for finding optimal models. As will be outlined in subsequent sections, much of the pipeline’s original infrastructure has been abstracted to support the addition and removal of new types of models, new data sets, new libraries, and varying environmental conditions. More details on the projects solutions can be found in Section 3.

**Paper Overview**

This paper is organized as follows: Section 2 provides a more comprehensive background of current fraud techniques and their effects on the industry as well as a formal description of all technologies and machine learning algorithms that will be used. Section 3 provides an overview of proposed solutions. Section 4 describes the pipeline’s structure and technologies, while Section 5 and 6 explore the Deep Learning and User Interaction aspects of the pipeline in detail. Section 7 discusses the project’s outcome. In Section 8 we recommend next steps for the pipeline. Finally, additional information can be found in the Section 10 through 12 Appendices.
2. Context

2.1 Fraud

2.1.1 Fraud and the Problems It Poses

With fraud and data breaches becoming increasingly prominent, many people and industries are recognizing the need for a larger emphasis on security. As the annual Nilson Report finds, in 2016 24.71 billion dollars were lost to fraud, a 13% increase over the previous year (Nilson, 2016). In the US alone, 31.8 million people had their cards breached in 2014, making up 47 percent of all global credit card fraud that year (Holmes, 2016). This trend is anticipated to have continuous growth, with experts approximately 27.69 billion in fraud for 2017, and 31.26 billion for 2018. In fact, since the year 2000 there has never been a decrease in global fraud (Kiernan, 2017), (Nilson, 2016).

![Global Fraud per year](image)

Since fraud is such a prevalent issue, new standards are being introduced. In the United States, cards are beginning to introduce what are called EMV security chips. Europay, Mastercard, and Visa have lobbied for the technology to be globally used because of its success. However, this technology has not prevented new account fraud, which has since gone up by 113% since this new technology was implemented (Kiernan, 2017). Additionally, electronic payment fraud – or fraud that occurs when the card is not present – has seen an increase, accounting for 45% of all fraud in 2014 (Holmes, 2016).
Table 1 Large Scale Data Breaches. 86% of identity theft victims experience fraudulent use of already existing accounts. Breaches such as these, nullify the security of a physical card. (Kiernan, 2017)

<table>
<thead>
<tr>
<th>Company</th>
<th>Year</th>
<th>Number of Accounts Affected</th>
</tr>
</thead>
<tbody>
<tr>
<td>CardSystems Solutions</td>
<td>2005</td>
<td>40 million</td>
</tr>
<tr>
<td>TJX Companies, Inc.</td>
<td>2006</td>
<td>94 million</td>
</tr>
<tr>
<td>U.S. Veterans Affairs</td>
<td>2006</td>
<td>17.5 million</td>
</tr>
<tr>
<td>Certegy</td>
<td>2007</td>
<td>8.5 million</td>
</tr>
<tr>
<td>Fidelity National Information Services</td>
<td>2007</td>
<td>3.2 million</td>
</tr>
<tr>
<td>Heartland Payment Systems</td>
<td>2008</td>
<td>134 million</td>
</tr>
<tr>
<td>Bank of New York Mellon</td>
<td>2008</td>
<td>12.5 million</td>
</tr>
<tr>
<td>Hannaford Bros. Supermarket Chain</td>
<td>2008</td>
<td>4.2 million (credit &amp; debit)</td>
</tr>
<tr>
<td>Sony</td>
<td>2011</td>
<td>12 million</td>
</tr>
<tr>
<td>Global Payments</td>
<td>2012</td>
<td>1 million</td>
</tr>
<tr>
<td>Target</td>
<td>2013</td>
<td>40 million (credit &amp; debit)</td>
</tr>
<tr>
<td>Home Depot</td>
<td>2014</td>
<td>56 million</td>
</tr>
</tbody>
</table>

The table above lists just a few of the large scale breaches of credit/debit card or account information. These data breaches are important because 86% of identity theft victims experienced the fraudulent use of already existing accounts, including credit cards and bank accounts. The problem with these breaches is that it nullifies the physical security of the card itself, since the attackers know all of the information for a credit card, as well as the information on the customer themselves.

**Current Fraud Problems**

Organizations can expect to lose on average 5% of their annual revenue to fraud (Warin, 2013). This causes companies to be required to invest in anti-fraud techniques in order to protect their revenue. The most widely used anti-fraud methods are rule-based: large sets of rules are used to categorize a transaction as fraud or not. General rules can be applied, such as: if multiple
large purchases happen within a small window of time, sequential purchases occurring large
distances from each other in a short period of time, or a transaction exceeding a certain amount
over the customer's average. Geographical locations subject to high rates of fraud may be
identified for active tracking in an effort to reduce activity in those areas (Palmer, 2013).
Specific rules can be set by the customer for most credit card companies, further tailoring the
rule set to their own needs. Once a rule has been triggered the transaction is marked for manual
review. Depending on the company, the customer may be asked to approve the transaction, or an
employee may inspect the transaction for further detail.

2.1.2 Fraud Detection through Machine Learning
Several fraud detection techniques exist beyond rule-based systems. Most recently,
machine learning algorithms have been adapted to either augment existing systems or replace
them altogether. Machine learning algorithms primarily aim to construct abstract models that
encode important patterns in the data. The effective implementation of these algorithms often
requires domain knowledge of law, economics, and business practices. Algorithms such as
artificial neural networks, decision trees, and random forests, among others have come to
generate great interest in fraud detection. For instance, the world's largest online payment
company, PayPal, and the FICO Falcon Fraud Manager, who report a 50% increase in
effectiveness from rule-based systems, use artificial neural networks (Crossman, 2016) (FICO,
2017).

Machine Learning for Fraud Detection
While possibly not as effective as neural networks at detecting fraud, Adam Langron, a
chief data scientist for Llyods Bank, and Alejandro Banshen, a chief data scientist for Easy
Solutions London, nevertheless report that random forests and decision trees have potential uses
in this application. Decisions trees, as Banshen notes, have the potential to locate features from a
dataset that are most relevant to fraud practices (Banshen 2017). Random Forests, Adam
Langron notes, are also useful for finding interesting features from a dataset, are quick to build,
and have very high discrimination rates (Langron 2015). Using these methods for feature
selection, as suggested in these testimonies, a Neural Network can then be trained on a smaller
subset of the original data set using only the selected features.

Neural networks, however, remain the choice algorithm for detecting fraud. Unlike rule
based systems that require specification of rules for the detection of fraud, neural networks learn
by example. This means that neural networks learn subtler patterns in the data than rule-based
systems. At a high level, two approaches to learning can be employed: supervised and
unsupervised. Supervised learning requires the training data to be classified prior to training—
most likely by a human operator. This then allows the algorithm, after training, to "learn" what to
predict for unclassified data. On the other hand, unsupervised learning does not require the
training data to be classified prior to training. It is used to find any underlying structure that might be present in the data.

**Supervised Learning**

Supervised machine learning, including supervised neural networks, have been extensively used for fraud detection (Estevez 2006). These networks, initially inspired by animal brains, "learn" to do different tasks by example. In the simplest scenario, neural networks for fraud detection require training data to be labeled as either fraudulent or not fraudulent. After training, they then learn to detect criteria for classifying new unlabeled data.

A neural network primarily aims to find a function that would, most generally, model the data distribution from the training data. To do this, it utilizes a cost function to determine how closely it 'fits' the true distribution of the data. For example, the cost function determines how far away the neural network was from classifying a fraudulent transaction as fraudulent. This allows the neural network to adjust the function or model that it is trying to approximate. The end goal of training is to be as accurate and generalizable to unlabeled data as possible in production use.

Supervised learning falls short, however, if either the size or the quality of training data is insufficient. That is, the training data must sufficiently represent the true population. While data is easily found, labeling this data is significantly more difficult. This process often requires human workers to manually identify individual data points. This can be a problem since models for fraud detection need to be retrained with the latest data at regular intervals to incorporate the latest behaviors of fraudsters. However, their simplicity and ease of use still prove supervised learning methods to be effective and justify their costs.

**Unsupervised Learning**

Unsupervised learning does not require labeled data. Unlike supervised learning algorithms, unsupervised learning can be used with large unlabeled datasets to extract valuable insights and patterns.

Unsupervised learning, like supervised learning, minimizes a cost function-- which can be any function of the data and the model's output. In unsupervised learning, however, the cost function is dependent on the task at hand as well as any other knowledge of the domain. While supervised learning has the problem of overfitting the data, unsupervised learning does not because the examples are unlabeled during training.

An example of unsupervised learning is clustering. Clustering aims to find 'groups' of data points with common characteristics or features. Such commonality might imply, in some cases, that the groups of data share a distinct property. Figure 3 shows an example of clustering in two dimensional unlabeled data. It shows the four different clusters (color-coded) that a clustering algorithm might find when “told” to look for four clusters.
Figure 3 Clustering, a form of unsupervised learning, attempts to place data points into groups as a means to distinguish one from another – just like the above example.

One example of a successful implementation of unsupervised learning for fraud detection models spending behavior with credit card transactions (Bolton. 2001). While unsupervised algorithms do not have the problem of labeled training data, they typically require domain knowledge of the dataset (Deshpande, 2015). This means a human with significant domain expertise must initialize these algorithms to gain prior information about the data, rather than discover clusters fully on its own.
2.2 Technical Background

2.2.1 Hadoop

Apache Hadoop is an open source platform created by the Apache Foundation to serve the purpose of distributed data storage and processing on clusters of nodes. Hadoop provides a means for storing data using the Hadoop Distributed File System (HDFS), which is made fault-tolerant by the automated replication of data across the cluster ahead of time. Additionally, Hadoop has the ability to scale to petabytes due to the locality of its computations on the cluster through the built in MapReduce Library (Hortonworks, 2017a). With the recent addition of the YARN centralized resource manager, Hadoop can now delegate multiple workloads simultaneously to newer data processing libraries such as Apache Spark (Hortonworks, 2017b).

Hadoop clusters follow a Master/Slave architecture through the naming convention of NameNode/DataNode respectively. Within a cluster there is a single NameNode to serve as the file system manager and client access regulator. DataNodes manage storage attached to the nodes they run atop of, and execute operations on the data they store as requested by NameNode. Internally, files inserted into the system are broken into blocks, with each block being scattered and replicated across a series of DataNodes as delegated by the NameNode. DataNodes report which blocks they are storing to the NameNode through a block report message, while the NameNode tracks the health of a DataNode through heartbeat requests (if a heartbeat were to fail, the DataNode would be assumed down until the heartbeat is received) (Apache Software Foundation, 2017f). Clients can interact with data through the Java API or the FS Shell command line tool.

2.2.2 Spark

Apache Spark is an in-memory data processing engine suitable for machine learning or SQL workloads. Spark integrates with Hadoop YARN to operate over the entire cluster in place of alternative data processing engines such as Hadoop MapReduce. Spark processes data much faster than MapReduce by caching data in memory first. Any data that doesn’t fit in memory is then cached to disk, later to be brought into memory when Spark needs to reference it. Spark also includes MLLib for access to common machine learning algorithms (Hortonworks, 2017c).

Spark processes its data in Resilient Distributed Datasets (RDDs), which are fault-tolerant collections of elements that can be operated on in parallel — thus having an inherent ability for Data Parallelization as compared to Task Parallelization. RDDs support transformation actions to produce new datasets from the old, namely actions which produce a value based on a dataset. These RDD computations can then be persisted to memory, allowing them to be referenced later without recomputing (Apache Software Foundation, 2017i).

Alternatives to Spark include Apache Storm, MapReduce, and Apex. Apache Storm focuses on real-time data analysis in streams rather than batches, making it suitable for Task Parallelization on real time data (Apache Software Foundation, 2015). MapReduce is a slower
alternative to Apache Spark for batch data operations, due to its lack of memory caching (Apache Software Foundation, 2017c). Apache Apex, similar to Storm, focuses on stream processing rather than batch processing, but also adds a layer of fault tolerance (Apache Software Foundation, 2017a).

2.2.3 Livy

Apache Livy is a lightweight, programmatic and RESTful interface for submitting Spark jobs on a Hadoop cluster, (Apache Software Foundation, 2017k). Typically, job submission is handled through the usage of a shell script “Spark-submit.” This shell script allows users to interact with the cluster from an end node – a node that has a Spark client installed upon it. While that allows the manual triggering of Spark jobs, to automate the process this shell script must be called from either a programmatic subsystem call or a direct call to the shell. This creates a dependency on the host system for applications to run successfully.

Livy addresses this problem by serving the same functionality as Spark-submit but bundled as a REST enabled web server running on the cluster or called through programmatic interface. The same tasks and configurations as a Spark-submit command can be run through Livy. Livy simply takes these requests and forwards them to Spark, lifting the application dependency from host to cluster level. In addition, Livy can mimic the REPL clients Spark has for Java, Scala, and Python – allowing users to interact with Spark remotely through an interpreter in what they call a “Session.” This allows code to be evaluated and returned statement by statement, much like using a command line interpreter – but this time through a series of REST or function calls.

For non-session applications, Livy treats these applications as “Batches.” Batches track the progress, logs, and stages of a Spark application. Upon submission of a batch to Livy, a JSON is returned with a Batch ID and application status. The Batch ID may be used in future calls to Livy for tracking the progress of an application, obtaining log information, or stopping the application tracked by it.

Livy’s limitations come in how results and inputs are handled. Due to the middle-man nature of Livy, results from Spark applications cannot be returned to the caller like a Session can. Thus outputs must be communicated back to the caller in another manner such as REST or written to HDFS. Additionally, inputs beyond command-line arguments must be hosted on HDFS, since file upload is not supported. Therefore, the job to be run and its dependencies must already be on each node in the cluster or live on HDFS. Finally, support for Hive is not pre-established as is when run via Spark-submit, requiring the same configuration file Spark-submit uses to setup the Hive connection to be passed to Livy (dependency hosted on HDFS).
2.2.4 Hive

Hadoop allows data to be accessed in batches natively, in a low-latency NoSQL manner through Apache HBase, or in an interactive, relational manner with Apache Hive. The latter of the three allows data stored within the cluster to be queried using SQL. In Hive, data is stored in tables which can be subdivided into partitions. These partitions help break down how data is related to one another so queries can execute over a smaller subset of a table. Partitions can then be broken into buckets, again allowing data to be sampled faster or quicker queries. While optional, by defining partitions and buckets Hive can alter how it stores its tables to help subdivide large quantities of data. Using buckets and partitions in tandem with Hadoop’s in-memory caching allows queries to be executed in fractions of a second (Hortonworks, 2017d).

Notably, however, is that Apache Hive is not built for Online Transaction Processing (OTLP) workloads, but rather for data warehousing workloads or batch processing analytics (Apache Software Foundation, 2017b). OTLP refers to workloads generated from financial based applications processing consumer transactions, typically in the form of vast number of records being generated containing small amounts data in real time (IBM Corporation, 2010). This limitation becomes apparent after recognizing Hive does not allow new data to be inserted into existing partitions. If this functionality is desired, Hive’s Streaming API allows new data to flow into Hive, which is treated as small batches that become committed and immediately visible to all running processes (Apache Software Foundation, 2017j).

2.2.5 MongoDB

MongoDB is a NoSQL database that stores data in JSON-like documents, allowing fields to change from document to document and a flexible data structure (MongoDB, 2017f). NoSQL is a general term describing databases that store information in a non-relational manner, such as documents. This allows large horizontal scaling, where, as load increases additional servers can be added to maintain database performance by automatically rebalancing data across all servers (MongoDB, 2017d). Data is altered through the CRUD paradigm: Create, Read, Update, and Delete. Schemas provide structure to documents, enforcing a minimum amount of fields that need to be present (MongoDB, 2017c). Due to the nature of NoSQL, MongoDB serves as a good solution for metadata management (MongoDB, 2017b).

Another popular use for MongoDB is alongside Spark, which MongoDB provides a native connector for. When combined with traits unique to MongoDB such as secondary indexes and aggregation pipeline, Spark can operate on data with specific features. Data can also be co-located with Spark’s RDDs, removing additional overhead for moving data (MongoDB, 2017a). Use cases for Spark and MongoDB include serving relevant ads to users browsing and viewing videos, or building predictive failure models based on material samples from production lines (MongoDB, 2017e).
2.2.6 Machine Learning Libraries

2.2.6.1 MLLib & ML

MLLib is the original machine learning library used with Spark, which is based around RDDs for data manipulations. MLLib has been updated to include a secondary API, however, called ML that utilizes DataFrames instead of RDDs for greater cross-API flexibility and performance improvements. This has caused MLLib to enter maintenance mode, while ML has become the primary API receiving new features beyond the offerings of the MLLib API (Apache Software Foundation, 2017). For example, as of Spark 2.0 ML supports persistence for saving and loading models or even entire pipelines (Bradley, 2016). ML provides the structure for a full pipeline workflow, in the form of DataFrames to store data, Transformers to mutate DataFrames, Estimators to produce a model from a DataFrame, Pipeline to chain together transformers and estimators, and Parameter for specifying parameters across a Pipeline (Apache Software Foundation, 2017h).

ML’s current implementation wraps MLLib algorithms, so the two of them provide the same core machine learning algorithms (Apache Software Foundation, 2017g). This comes in the form of estimators for Classification and Regression through the usage of Decision Trees, Random Forests, Linear Support Vector Machines, and Multilayer Perceptron Networks (Apache Software Foundation, 2017d). The breadth of features provided by ML make it a good general purpose machine learning library, serving many different needs in one package. ML does not, however, provide support for Deep Learning methods such as Deep Neural Networks, Convolutional Neural Networks, or Recurrent Neural Networks - such behavior would come from a specialized library to work alongside ML. Additionally, there is support for K-Means clustering and the Frequent Pattern Growth algorithm (a parallel version of the latter is provided by MLLib) (Apache Software Foundation, 2017e).

2.2.6.2 TensorFlow

TensorFlow is an open source, Deep Learning library created by Google to make Deep Learning easier for the world (Google, 2017c). TensorFlow not only has optimizations for typical CPU based neural network training, but can also take advantage of CUDA on Nvidia GPUs to speed up the process (NVIDIA, 2017). The library provides an extensive API, including a high-level tf.contrib.learn API for learning and a distributed API for running TensorFlow over a cluster - with support for HDFS (Google, 2017a).

TensorFlow is able to operate in distributed mode on top of Spark thanks to Yahoo!’s TensorFlowOnSpark initiative, which wraps the Distributed TensorFlow API into a Spark compatible module. Current TensorFlow applications only need a few lines of code changed to become a Spark ready application (Yahoo!, 2017). This lets TensorFlow exploit the speed and coordination of a Spark cluster, while also providing the ability to parallelize the training of a single model, train different models in parallel, or tuning hyperparameters in parallel (Google, 2017a) (Hunter, 2017).
TensorFlow’s tf.contrib.learn API, as aforementioned, is TensorFlow’s high-level API for learning that serves as a way to abstract the subtleties of lower-level API endpoints. This high level API exposes many functionalities, some of which are experimental or undocumented general machine learning algorithms not officially supported (Google, 2017b). To this extent, TensorFlow remains a specialist in Deep Learning - the Learn API providing the means to build, evaluate, monitor, and export TensorFlow monitors. As of version 1.0, the Learn API also added official support for another high-level abstraction API - Keras (Sandjideh, 2017).

Keras serves as a means to abstract the low-level APIs provided by several Deep Learning frameworks, one of which is TensorFlow. Keras aims to make the step from concept to reality much quicker by focusing on user friendliness, modularity, extensibility, and Python integration. As a result, what can be challenging to build with a native Deep Learning API becomes quick, easy, and brief with the Keras abstraction layer (Chollet, 2017). Due to the nature of Keras being only a frontend to these frameworks, it is possible for Keras to work alongside Distributed TensorFlow - or anywhere that TensorFlow can be applied (Various, 2017).
2.3 Machine Learning Algorithms

2.3.1 Decision Trees

Decision trees serve as a means to approximate discrete valued functions, capable of handling noisy data and learning disjunctive expressions. A decision tree could be viewed as a series of If-Then statements a human might ask (Mitchell, 1997).

A decision tree’s representation can be thought of as a series of classifications. Starting at the root of the tree, each node tests the possible attributes, that instance can take on. In Figure 4, for example, the instance “Outlook” can take on the attributes “Sunny,” “Overcast,” or “Rain.” Each path down the tree would then represent a conjunction of these tests, which the tree itself is a disjunction of all of its paths (Mitchell, 1997). Appropriate problems for decision trees are those that have a discrete number of outputs, whose instances are represented by attributes with values, and can be described in a disjunctive manner. Decision Trees can also account for training data that has errors or missing values, making it a potential candidate when data is incomplete (Mitchell, 1997).

The most basic algorithm for decision trees is known as the ID3 algorithm, which constructs the tree top-down. The ordering of nodes is chosen based on which attribute tells us the most about the training data at that point. When a node is selected, the training data is split by that attribute for testing further down the path. The algorithm terminates when all attributes are accounted for. In a sense, this is a Greedy Search algorithm with no backtracking due to its prioritization policy. We can evaluate how well an attribute classifies the training data by looking at its entropy:

\[
\text{Entropy}(S) = \sum_{i=0}^{c} (-c \times \log(c))
\]
Where $c$ is one of $i$ attributes the training data can have. Entropy describes the proportion of examples containing this attribute, looking to classify how pure a categorization is. The better an attribute is at reducing entropy, the stronger it is at classifying the training data (Mitchell, 1997).

The shortfalls of decision trees come when there is error in the data or a lot of noise within the training examples, causing trees to be built incorrectly. We can resolve this by overfitting the tree and then pruning it afterwards by policy of Reduced Error Pruning or Rule Post-Pruning. Additionally, attributes with too many values can cause the tree to become very wide instead of balanced, leading to poor performance when trying to classify. This can be overcome with the use of variances of the entropy metric. Finally, continuous must be broken into discrete intervals, which can be unfavorable depending on the problem (Mitchell, 1997).

### 2.3.2 Random Forests

Random forest classification is an ensemble learning method that grows many decision trees. For classification, an input vector is sent down each one of the trees and the output with the most number of votes wins.

![Random Forest Diagram](image)

*Figure 5: A sample representation of a Random Forest Consisting of many trees, showing how multiple trees can lead to a more democratic approach towards finding a singular result. This leads to more possibilities evaluated compared to a decision tree.*

The growth of each tree includes sampling a training set of data with replacement at random from the original dataset. If the dataset includes $M$ input variables, a number $m << M$ is chosen and $m$ variables are selected at random. These $m$ randomly selected variables are then used to split nodes for each tree. Importantly, $m$ is kept constant for each tree. Each tree is then grown to the end without pruning.

The classification error for random forests mostly depends on the *correlation* between trees and the *strength* of each individual tree. Tuning the strength of each tree and the correlation
between each tree involves a tradeoff: reducing \( m \) reduces both correlation and strength (Breiman, 2001).

Random forests have several advantages, the most important of which is that it is unexcelled in accuracy among algorithms. It does not over fit data and it can handle several thousands of input variables without variable deletion. It also generates estimates of variable importance in classification. It can also be extended to unlabeled data, allowing unsupervised clustering and anomaly detection.

Due to sampling with replacement, about \( \frac{1}{3} \) of the original dataset is left out and is known as ‘out-of-bag’ data. This data is then used for getting an estimate for classification error after training as well as estimates of feature importances. This method of evaluating the random forest has proved to be unbiased in most cases.

Some datasets, like fraud-labeled datasets, have an imbalance in the number of fraud vs non-fraud cases. This sometimes results in off-balance prediction errors for each class. For example, in the case of fraud, the classifier might be more accurate in predicting non-fraud cases but less so in predicting fraud-cases. This error is typically eliminated by setting different weights for classes. If the weight of a class is high, the classifier decreases its error rate. Typically, the weight is set to the inverse of the class populations. However, an important consideration is that balancing class error, leads to an increase in overall error.

2.3.3 Logistic Regression

Logistic regression is a form of regression used on sets of data where the output must be dichotomous. Logistic regression is most useful when the output takes a binary value such as success and fail, good and bad, or fraud and not fraud. Logistical regression is based off of the logistical function, also known as the sigmoid function. The Logistical function is a function which maps any value on a line to an S shaped curve which falls between the value of 0 and 1 (Logistic Regression, 2017).

\[
y = b_0 + b_1 x
\]

\[
p = \frac{1}{1 + e^{-(b_0 + b_1 x)}}
\]

*Figure 6 Logistic Model compared to a linear model. In a logistic model a line is attempted to be fit such that it becomes a predictor. Having too many highly correlated inputs, however, can cause overfitting.*
Logistical regression uses input values (x) and combines them using weights (β) in order to predict an output value (y) (Brownlee, 2016). The equation is:

\[ p = \frac{1}{1 + e^{-(b_0 + b_1 x_1 + b_2 x_2 + \cdots + b_p x_p)}} \]

Notice how in Figure 6 the logistic model has the linear model inside of it, this is because the logistic model is simply a nonlinear transform of the linear model. The logistic model seeks to take the value that is output by a linear model and fit it to a curve ranging from 0 to 1. For example, being very close to 1 may mean that the inputs suggest that a transaction is fraud, while being very close to 0 suggests the transaction is not fraud.

In order to estimate the values of weights maximum-likelihood estimation must be used (Brownlee, 2016). In machine learning the training data is used to tune the values of β which then form the trained model itself, which is a list of the coefficients found during training. Maximum-likelihood estimation seeks to find estimates for all β such that the predicted probability of the output corresponds as closely as possible to the observed probability (Gareth, 2015). The Maximum-Likelihood function is as follows:

\[ L(\beta) = \prod_{i:y_i=1} p(x_i) \prod_{i:y_i=0} (1 - p(x_i)) \]

Other methods can be used as well, for example Newton’s method may be used to estimate the values of β (Shalizi, 2017), or iteratively reweighted least squares (Shalizi, 2017).

Logistical regression does make assumptions about the input data. As already discussed the output variable must be binary in order for logistic regression to be applicable. Logistic regression also assumes no error in the output, because of this it is important to remove outliers as well as possibly misclassified instances from the training data (Brownlee, 2016). The input and output are also assumed to have a linear relationship. If they do not, log transformations can possibly alleviate this issue (Brownlee, 2016). The input and output must have a linear relationship since the logistic regression model maps higher values to be closer and closer to 1, so if the relationship is not linear, there is no way to have a value be accurately mapped.

It is also important to remove highly correlated inputs. Having multiple highly correlated inputs can lead to an over fitting of the model, which while it may increase the validity over training data, it will make the model inaccurate to new instances of data it has yet to see. (What is logistic regression, 2017). The maximum-likelihood function may also not converge, causing the corresponding weights to not properly scale significant vs insignificant inputs.

### 2.3.4 Artificial Neural Network

An artificial neural network is a computing system made of a number of interconnected nodes (hereafter artificial neurons) that attempt to process data as a human brain would, namely
by looking at examples and learning from the outcomes of those examples (University of Wisconsin, 2007). Typical artificial neural networks are organized into a series of 3 different layers: input, output, and hidden. Each layer consists of a multitude of neurons which process input data, sum up that data using activation functions such as sigmoid or tanh, and return that data as an output. Furthermore, input data is weighted, letting each artificial neuron decide how important that input feature is to the model. In other words, not all features will have as much of an impact on the final prediction as others.

![Artificial Neuron Diagram]

*Figure 7 A simplistic representation of an artificial neuron that has exactly three layers. Neural Networks may have many more hidden layers depending upon the function of the network.*

The input layer acts as an interface between the network and an external data source. Depending upon the activation functions used in this network, input data may need to be normalized to be correctly processed before it can be handled by the network itself (examples of functions that require normalizing include sigmoid, tanh, and binary step). Once preprocessing is done, input layer neurons are typically responsible for calculating weighted connections between nodes using predetermined 'learning rules' that decide which patterns or features in the dataset matter the most for making solid predictions (University of Wisconsin, 2007). Each value from the input layer is then duplicated, multiplied by its respective weight, and sent to each neuron in the hidden layer (Steven Smith, 2011). It is important to note that input layer neurons do not use activation functions to generate output, they merely act as a means of sending weighted input data to the hidden layer where the bulk of processing is done.

In the hidden layer, neurons make predictions of whether or not certain elements may be true from the initial data imported into the neural network. For instance, if the network was trying to predict whether or not a transaction was fraudulent, one hidden neuron might be looking at the likelihood that a single transaction was consistent with a single known pattern of fraud. To do this hidden neurons use activation functions to calculate weighted sums of their input data to decide if they should send an output to a node in the next layer (Avinash Sharma V, 2017). If enough of the neurons in this layer detect possible fraudulence, an output layer neuron will likely fire off concluding that it was a case of fraud.
\[
\frac{1}{1 + \exp(-\sum_j w_j x_j - b)}.
\]

Figure 8 One of the more widely used activation function is the sigmoid function (featured above). This is used to transform hidden and output neuron input into output that can piped to the next layer.

Output layer artificial neurons complete the network and return a result to the external source. Each data point gets processed through to the output layer. The neuron in this layer with the closest value to 1 indicates the class of the processed data point.

2.3.4.1 Deep Neural Network

Deep neural networks (DNNs) are classified by having more than three layers of nodes within its network, distinguishing them from the simpler artificial neural networks (Introduction to Deep Neural Networks, 2017). While the inner workings of neural networks have been related to black boxes, it is known that DNNs work by inspecting a different aspect of the problem being solved in each layer. Layers further in the network can then process these high level aspects, extracting details from a problem that may not have been previously thought of. DNNs succeed in cases where the culmination of many pieces of information creates an important classification. (Exploring Deep Learning, 2017). Figure 9 shows the structure of a deep neural network, notice the three hidden layers included in this example.

Figure 9 Example deep neural network diagram, which can be considered more akin to a Multilayer Perceptron but with far more layers and neurons. Typically a network is considered deep if it contains 3 or more layers. (Exploring Deep Learning, 2017)
For example, image recognition is one of the problems that deep neural networks solve particularly well. For a neural network that must recognize objects in images, it is necessary to have inputs to the network that are easily interpretable by a computer. In order to do this, prior to training, all images are encoded into matrices that consist of the pixel density values as shown in Figure 10.

The neural network takes the matrix of pixel density values and transforms them into a different matrix. The result of this matrix transformation yields the input values of the next matrix transformation. The new matrices after each transformation are called "layers". These transformations occur until the output of the neural network, which typically produces the probabilities of the input being in each of the classes that the neural network is predicting.

There are many different variables in the creation of a DNN that must be adjusted such as the size of each individual hidden layer, the activation function, and even the number of hidden layers (Brownlee, 2016). These are considered “Hyperparameters” of the network, in which there is no one size fits all configuration for every DNN implementation. For example, when a DNN is attempting to transcribe multidigit numbers from photographs of addresses it was found that increasing the number of hidden layers lead to an increase in accuracy (Goodfellow et all, 2016).

This does not mean that simply adding additional hidden layers to a DNN will add to its accuracy every time. It is important to determine based on the specific situation what the best
configurations will be, so testing the configurations and trying different settings is important. While in some situation increasing the size of each hidden layer will add to accuracy, sometimes it will not.

Changes can also be made to parameters that are specific to training a model. For example, try different epochs or different batch sizes and compare the model accuracy to past configurations (Brownlee, 2016).

DNNs come with limitations, one of the largest being the amount and accuracy of the data needed to properly train. In comparison to training an ANN, DNNs require much more data in order to be accurate (Yao, 2017). If the data that is available to train is not enough, it may be worth it to either get more data, or to create more data by creating similar entries to ones that already exist in the data and make small changes (Brownlee, 2016).

### 2.3.4.2 Recurrent Neural Network

Recurrent neural networks are a particular type of deep neural networks that are useful in detecting patterns that might exist in sequences of data such as text, handwriting, and spoken word (Deep Learning 4j, 2017). They are often thought to be the most powerful type of currently implemented neural networks as they offer a way of storing earlier predictions in memory and can then draw upon these earlier predictions for enhancing future predictions later on.

To do this, recurrent neural networks add the ability for layers of neurons to send feedback up to previous layers. This gives neurons, particularly those in hidden layers, two separate inputs, present data and recent predictions that the network has already made. Having this data is extremely valuable in processes like text prediction, since it would be impossible to generate a good next guess without first knowing what the previous prediction and its connotation were (Britz, 2015). As seen in Figure 13 below, feedback is stored in memory for as
long as there is enough disk space for it. This means that feedback piped to an earlier layer will contain information not just about the last prediction, but also predictions made before it.

Each hidden layer, therefore, has output that is directly linked to previous hidden layer output. As such, to calculate this output, the model below can be used. In it, phi represents the sigmoid or tanh function, \( x_t \) is the time step, W is the weight matrix, and \( h_{t-1} \) is the previous time step modified by its own transition matrix, U (Deep Learning 4j, 2017).

\[
h_t = \phi(Wx_t + Uh_{t-1})
\]

In brief, training recurrent neural networks is very similar to training any type of deep neural network (i.e. training utilizes the backpropagation algorithm), but in recurrent neural networks parameters are shared by all time steps. Therefore, the output of each layer needs to account for previously calculated outputs in addition to the output given by a current step (Britz, 2017). This is done by modifying the backpropagation method slightly so that the output of a neuron sums up the output of previous time steps and modifies the current output gradient accordingly.

While recurrent neural networks show a lot of promise in speech recognition and related applications, they are inhibited somewhat by what is referred to as the "vanishing/exploding gradient problem". This problem, which persists through all gradient-based algorithms, is caused when a change in a network's output is so small or large that the network cannot learn the parameter effectively (Grosse, 2017). For example, in speech recognition if we want to determine what language a person might speak after reading in the text "I grew up in France... I speak fluent," the
context of France is necessary to determine that the language is probably French. But if the gap is large enough between France and the language that we want to predict, it is possible that the recurrent neural network may be unable to connect language to the speaker growing up in France (Olah, 2015). Nevertheless, with the invention of long short term neural networks (a variation of recurrent neural networks), this may no longer be a major issue.

### 2.4 Definition of Evaluation Metrics Used

#### 2.4.1 Accuracy

\[
Accuracy = \frac{True\ Positive}{Total\ Examples}
\]

In context of binary classification and our pipeline, accuracy is seen as the number of correctly classified examples (True Positive) out of all seen examples. Having a higher accuracy, then, is better as it shows the model is correctly labelling items. A well performing model, however, would not want to have a perfect accuracy of 100%, however, as it is indicative the model has over-fit to its training data. Over-fit meaning the model knows how to perfectly classify the examples it has seen, but given new examples, it would perform horribly because it only knows how to classify what it has seen – *it has not generalized a means to classify* (Mathworks, 2018).

#### 2.4.2 Precision

\[
Precision = \frac{True\ Positive}{All\ Positive}
\]

In binary classification and our pipeline, precision is the number of correctly classified examples (True Positive) out of all examples of that class (All Positive). Having a high precision is good, meaning the model classifying a specific class correctly. Too high of a precision may be a sign of overfitting, albeit not as telling as a high accuracy, given that the model may over-fit a specific label and become unable to generalize that label beyond the training examples.

#### 2.4.3 Recall

\[
Recall = \frac{True\ Positive}{True\ Positive + False\ Negative}
\]

Recall is the number of correctly classified examples (True Positive) over the number of correct examples (True Positive) and false classifications for that class (False Negative). Compared to precision, recall looks to understand if a model is classifying all relevant examples or if some are being missed – whether those classifications are right is a matter of precision. This helps us understand if a model is finding the right classes within a problem, which when combined with precision, then tells us if those results are of the right class. Having a high recall, then, is good.
2.4.4 F1-Measure

\[
F1\text{Measure} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}
\]

The F1-Measure is a combination of the precision and recall, which balances the two when computing a final metric. Having a high F1-Measure is indicative the model is precise, in that is classifies within each class well, and does not miss many examples when classifying.

2.4.5 False Positive Rate

\[
\text{False Positive Rate} = \frac{\text{False Positive}}{\text{False Positive} + \text{True Negative}}
\]

The False Positive Rate looks at how often a model classifies an example it shouldn’t as a specific class. A false positive can be thought of as “false alarms” or classifying an example as a specific class when it is not. A true negative is when a model correctly classifies an example as not this class. Thus, looking at the total wrong classifications over all the items that should not be this class tells us how often a model is wrongly identifying examples. Having a lower false positive rate is better.
3. Proposed Solutions

This project aimed to build upon the successes of the previous MQP group's machine learning pipeline to move it closer to a state where it could be pushed to production. In addition, it set forth in exploring model accuracy and visualization approaches for aiding in the development of higher performing models. The overall aim, thus, was to enable those of posterity to more effectively develop model where the project could not accomplish within its timeframe. To accomplish this, proposed solutions to all goals identified in the introduction are laid out in the following section.

3.1 A Production Ready Pipeline

The pipeline started as a strong proof of concept with most of the framework established, as shown later in Section 4.1. Specifically, the pipeline’s data flow design was sound and the user interface had gone through several iterations on its design. It scaled well horizontally due to the way it stored metadata. There were still problems that needed addressing, however, before it could be considered production ready.

The first problem was that the pipeline relied heavily on a CRON job (shell script) to coordinate tasks between the frontend and the cluster. While this is functional, it is difficult to maintain with time and involves several unnecessary intermediate scripts to communicate between the REST API, the metadata database, and Spark. The proposed fix to this was integrating Apache Livy for Spark, a REST based approach to submitting applications to Spark rather than using Spark’s shell script method. This would ensure reliability, transparency, simplicity, and maintainability that would all be welcome in a production environment.

The second problem is that the quality of the codebase was lacking in several areas. Across most scripts there were old code fragments left behind as comments, unnecessary print statements, and excessive lines of code leading to poor performance. Small regressions in functionality appeared across the pipeline resulting in most of it not being in a runnable state. Documentation was lacking, making maintenance impossible and the various component’s purposes hard to discern. Additionally, the version control repository was left scattered, with duplicate, outdated, or unused files that were checked in alongside the latest versions of the pipeline. Models were begin stored within the Git repository instead of HDFS. Lastly, the NodeJS server lacked proper routing, making the frontend and REST API overly complex. A comprehensive code review and restructuring has since taken place, with extensive documentation being left in the Github repository for posterity.

The third problem was many features and configurations of the pipeline were hardcoded into scripts, making them inflexible to new additions or changing environments. For example, the available model features and shape of the training data were specified in the Model Trainer rather than adapting to the input given by the metadata. Additionally, the location of model storage was hardcoded across the pipeline, which should have been placed into environment
configurations to make containerization and deployment simple. The NodeJS “DotEnv” framework has since been implemented, allowing these hard codings to be moved into environment variables where appropriate. All other cases were moved into the metadata or abstracted to function with varying inputs.

Lastly, there’s no evidence the pipeline’s code functions as intended. Each system relies on each other without any validation steps taking to ensure the pipeline works as it claims to. To address this, unit testing of the API, Livy (the Coordinator), and the Queue interface has been introduced to validate user level systems. To monitor the larger system, a series of system test examples were introduced to validate the cluster functions, specific libraries such as Keras works, and test networks on controlled data such as MNIST used to validate a specific algorithm is training as expected.

3.2 A Distributed Pipeline

In the initial iteration of the pipeline, the team was only capable of utilizing a cluster with a single node – effectively losing the performance gains that a Hadoop cluster brings. To qualify as a scalable pipeline, the previous iteration of the pipeline was brought into a multi-node cluster affectionately known as “Anton.” The structure of this cluster is defined in section 4.1.1 using Hortonworks Hadoop, its installation process documented in Appendix A for future replication.

With a cluster established, the next goal was to introduce training models in parallel, to best take advantage of additional executors should more be added from our initial cluster. This is known as “Task Parallelism,” and was implemented through our proposed replaced to the CRON job – the “LivyService.” The LivyService sole purpose is coordinating the training of multiple models at the same time. As models finish training, are due for a retraining, or new ones introduced the LivyService will push jobs to the cluster in parallelism – getting more done at the same time. Additional information on how the LivyService works can be found in section 4.6.

Finally, to increase the training speed of models we aimed to train the same model across multiple executors. This is known as “Data Parallelism.” To accomplish this, we integrated the Distributed TensorFlow to the cluster, allowing new models to be created and trained in a distributed manner. We used the TensorFlowOnSpark library provided by Yahoo to allow Distributed TensorFlow to work seamlessly with Apache Spark and Apache Hadoop. With this implementation, not only could multiple models train in parallel, but each model could be trained using multiple executors. To learn more about this integration, as well as Task and Data Parallelism, see Section 5.1.

3.3 A Hyperparameter Searching Pipeline

With the introduction of Task and Data Parallelism as outlined in section 5.1, the pipeline had the framework necessary to effectively search for better models for a given dataset. With this potential in mind, automated hyperparameter tuning was added to the pipeline – allowing for
optimal hyperparameters of a model to be chosen by the pipeline rather than requiring the user to manually input them.

Three different automated tuning algorithms – Grid, Random, and Bayesian search - were implemented using the HyperOpt and MLLib libraries. Grid Search, an exhaustive search algorithm, produces models for each different hyperparameter in a given range, and returns the most accurate model out of the bunch. Random Search, as its name implies, randomly chooses hyperparameters from a supplied range and produces a specified number of models and then returns the best produced model. Bayesian Search, the most effective of the three search algorithms, chooses which hyperparameters to test based on the accuracy of the previous model tested. A comparison between manual, grid, and random search is shown in Figure 14. For more information, see section 5.2 where each of these algorithms is discussed in depth.

In addition, the User Interface received new forms that allows users to specify regions the pipeline should attempt to produce models over (such as the range of number of layers to try, their sizes, activation functions to use, and more), giving users more freedom to control how narrow or wide a field the search algorithms operate on. These UI changes are highlighted in section 6.2.

![Figure 14 A comparison between manual, random, and grid search algorithms (Sharma, 2016).](image)

**3.4 A Dataset Agnostic Pipeline**

The pipeline’s prior implementation supported only a very specific dataset and no others. It had no documentation or support scripts to bring this data from local disk into the cluster. Without some support, future developers would struggle to work with the pipeline at all – data serving as the corner stone towards the entire process. In this iteration, we abstracted the hard coded assumptions about the data tables from model training (such as what columns existed, their data types, and where those tables could be accessed). We then created and documented various means to ingest different data shapes and types, successfully ingesting both in-house datasets as well as external datasets such as MNIST for system testing. Additionally, we
implemented the ability to test models on a separate dataset rather than just a train-test split, allowing common-ground comparisons to be made between models.

The inclusion of these features has helped address the excessive assumptions made by the prior iteration about the features in the data, where the data is, and what it is referred to as. Additional details on the changes we made can be found in sections 4.4, 4.5 and 4.7. Appendix C speaks more on how to ingest a new dataset.

### 3.5 A Pipeline of Many Algorithms

The pipeline was limited to Spark MLLib based machine learning algorithms only. This meant it could only train Random Forests, Decision Trees, or simple artificial neural networks. It could not take advantage of the latest and greatest machine learning libraries or techniques; nor could models be controlled with any fine-grained detail. Simply put, the opportunity to implement complex machine learning algorithms and libraries was missing from the architectural design.

To address this, the integration of Distributed TensorFlow allowed far more complex and controlled models to be introduced. Compared to the Spark MLLib library, TensorFlow does more than providing high level calls to the creation of specific algorithmic models (such as an Artificial Neural Network). TensorFlow allows the creation of custom models through the manipulation of Tensors, with control over each layer of the network and its properties (such as activation function). Higher level APIs can simplify this process where desired, while lower level ones allow fine-grained control over saving, creating, and training of models.

TensorFlow also comes bundled with the higher level, easier-to-use Keras API that allows easier model experimentation. As such, the Model Trainer was given a supporting package “ModelType” which allows any variety of machine learning algorithms or libraries to be called from the pipeline. The REST API and User Interface received updates to reflect these changes, allowing users to easily specify the exact ModelType they would like to make and the individual parameters that ModelType exposes. Appendix C outlines the ease of adding a new model, while sections 4.5 and 4.7 speak more on the REST API and Model Trainer changes made. Section 5 outlines more on TensorFlow and the tools brought with it.

### 3.6 A Performant Pipeline

After the extensive updates to the pipeline were addressed, the pipeline needed evaluation to fully understand its real-world ability. To accomplish this, we ingested a series of larger and larger datasets starting with the TFDescription Dataset (1 million records, 1,700 columns), then the RS Dataset (2 million records, 4,280 columns), and finally a series of RS subsets where features are normalized, selected by a Random Forest after normalization, and selected by PCA after normalization. Additionally, the MNIST handwritten digit dataset was ingested for use in
system control testing. Appendix C describes more on how these sets were ingested and transformed.

In addition, the evaluation of models in the prior pipeline had a broken scoring system with no means to monitor training in-progress. We fixed the scoring package, allowing all models to report their performance statistics such as accuracy, precision, recall, F1-Measure, and false positive rate. Alongside that, the integration of TensorBoard for TensorFlow has allowed model training to be monitored in real time for compatible ModelTypes – allowing deeper insight into how models are performing.

With these datasets and tools ready, we evaluated some of the new libraries introduced to the pipeline – TensorFlow (a Deep Neural Network implementation) as well as HyperOpt for Hyperparameter searching. Inspecting how this model performed, against which datasets, with what features and autotuning is documented in Section 7. We also examined the performance of the pipeline under a rich diversity of data and task parallelization workloads – such as the number of executors for a given model, or the number of models training at once. (The results from these tests can be found in section 7 of this paper.)
4. Pipeline Design

In this section, we will discuss in more detail the solutions that were introduced in Section 3. We will first provide information in Section 4.1 on how the current Hadoop cluster is structured, followed by how the various components of the pipeline interact. After which, in Section 4.2 and onward, we give details on each of those components and the aspects surrounding them. Along the way, we will cover additional, smaller features that may not have already been mentioned. What is provided in this section should be considered the current state of the pipeline.

4.1 Cluster and Pipeline Structure Overview

4.1.1 Cluster Structure

Prior to this project, the previous iteration of the pipeline was developed on a single node Hadoop cluster due to technical limitations. This project addresses this problem with the introduction of a 12 VM cluster powered by Hortonworks Hadoop. Each of these virtual machines is a single node on the cluster, two of the nodes are NameNodes, while the remaining 10 are DataNodes. One of the NameNodes takes the traditional role of a NameNode discussed in the Section 2.2.1. The secondary NameNode works as a failsafe, so if the first NameNode goes down for any reason, the data stored on the NameNode will not be lost and can be recovered from the secondary NameNode.

These NameNodes have HDFS client installed, which as discussed in 2.2.1 fills the role of the file storage system for the pipeline. The NameNodes have Spark 2 and Livy installed, allowing distributed, in memory computation to occur as described in section 2.2.2. With Livy, Spark jobs can be submitted through REST rather than command line – allowing for cleaner integration with other services generating Spark jobs.

The remaining ten nodes are set up as worker nodes. Eight of these nodes are strictly worker nodes without any client installations so from the view of a user they are just additional computing power available to the cluster. The final two worker nodes are the edge nodes to the cluster. They act as the access points to the cluster, allowing for users to access the cluster and interact with it. Users are authenticated at these edge nodes, preventing unwanted access while also tracking who is submitting jobs or acting upon the cluster.

4.1.2 Pipeline Structure Comparison

The prior pipeline implementation is detailed in length in the paper published by Esco et al (2017). This section will provide an overview of architecture comparisons, explaining changes made since then and how the system overall works together. For a more specific walkthrough of how data flows within the pipeline, see Appendix D.
The pipeline consists of three distinct systems: The NodeJS Webserver, the Spark Model Training, and Model and Metadata storage. The first handles requests to the system and dispatches to the necessary systems when making responses. The second utilizes the cluster to build, train, and evaluate models of the requested type (made from the Webserver). The final handles storing metadata describing models, serialized models themselves, and the data used to train the models.

In the prior iteration, as shown in Figure 15, the REST API lived as a separate process from the Express Server which was seen as excessive in this revision of the pipeline. Instead, by taking full advantage of Express, the REST API was merged into the User Interface server – thus making a single process for handling web requests. Thus, the Webserver maintains two simple
tasks: serve the user interface for job creation, and support pipeline operations through its REST API – additional details of which is provided in Section 4.5.

When a job is made via the interface, the cluster needs to be informed of it. To bridge this gap, a job service is needed to monitor the Job Queue, send items into the cluster as a Spark job, and monitor their progress. In the prior implementation, this service was a shell script (deemed the “Coordinator” in Figure 15). The usage of a CRON job was seen as bad practice (as described in Section 3.1), and as such, was removed in favor of our Livy Service. Additional reasoning and description on this change is provided in Section 4.6.

When a job is sent to the cluster, the secondary major system component kicks in: the Spark based Model Trainer. In the prior iteration the Model Trainer prepared data, trained models, evaluated them, and then stored those models on the local disk. Our iteration looked to break up the monolithic function provided in the Model Trainer by making its purpose more about dispatching models to be created, rather than handling it all on its own. Hence the ModelType package was introduced to hold model specific creation and evaluation code, leaving the Model Trainer to coordinate preparing the data for training, starting the model creation and evaluation, and then reporting and saving the resulting model and its performance. Additional changes are described in Section 4.7.

The final change made to the pipeline’s structure was in the Scoring Engine’s removal. In the prior iteration, this was a loose Spark Job that was not connected to the pipeline what-so-ever, meaning it had to be called manually. This script’s purpose was to evaluate a model again against a different set of data, but given it had no means of being executed from within the pipeline and did not function (it contained numerous errors preventing it from working), the item was removed. As described in Section 8.3, a future recommendation is to revisit this aspect of the pipeline to create a working system with an integration into the User Interface and REST API.
4.2 Configuration, Storage, and Documentation

Due to the prior hardcoding, it was only possible to run one instance of the pipeline at any time. Models could only be stored to a user’s local disk, and all model metadata could only be written to a single database pre-specified in the server code. While this worked well to prove the functionality of the pipeline, these hardcodings serve no place in production, and, as such, needed to be removed. The following sections describe what was done to the pipeline to configure the environment to improve portability.

4.2.1 Introducing DotEnv

Prior to this MQP, the pipeline had all environment configurations directly written in source code. These configurations included the database address at which each model’s metadata was written to, where training data could be found, what the data shape was, and even the port number at which the user interface ran – among others. By configuring the environment this way, the pipeline essentially assumed that several conditions were met, meaning that if someone attempted to run this pipeline on their own system, or with a different dataset, it may not work. If, for instance, a person wanted to store metadata to a database with a different address than the one listed in source code, that person would need to modify the code. This is generally considered very bad practice.

To fix the issue, this MQP pulled all environment configurations out of source code. In place of these hard-coded environment configurations, we inserted environment variables that could be adjusted with a separate .env configuration file that any user of the pipeline could make. Once an .env configuration file is made, variables written in it are imported to the server code using the framework Dotenv. A sample .env configuration file is shown in the image below.
These env files get injected at runtime, allowing containerization. To address hardcodings that weren’t simple configurations but fundamental flaws (such as the list of features to choose from in the frontend, and what Hive table those were in) additional REST API endpoints were added. These endpoints pulled hardcoding from the front and backend, consolidating them into one location – essentially a façade pattern. This allows the frontend to now have selectable datasets, in addition to a list of features that changes with that dataset. Suggestions on how to further improve this is documented in Section 8.

### 4.2.2 Storing Models to HDFS

With the prior evolution of the pipeline, all models were stored to local disk space. While this functions as one might expect, it was seen as more desirable to store models to HDFS on the cluster because of HDFS’s integration with Ambari Server that makes browsing file directories clean and simple. This also meant models were stored in a redundant manner rather than at a

```bash
1 # Please fill in the blanks (denoted with <VALUE_NAME>).
2 # The same values provided mimic what the final result should look like
3 # Server - General
4 MONGO_URL=mongodb://<hostname>:<port>/<database>
5 PIPELINE_PORT=8090
6 # Note, model home should NOT have the extra '/' at the end of the url - its
7 # assumed missing.
8 MODEL_HOME=hdfs://<hostname>(master):<port>(8028)/hdfs/path/to/model/storage
9 # Livy service
10 LIVY_IMPORT=http://<hostname>:<port>
11 LIVY_JOB=pyspark/hdfs/to/model_trainer.py or your job file.
12 # Livy service - python dependencies to your job file
13 # For each python zip dependency needed by model_trainer and friends,
14 PYFILE_CUSTOM_1=/path/on/hdfs/to/PYFILE_1.zip
15 PYFILE_CUSTOM_2=/path/on/hdfs/to/PYFILE_2.zip
16 PYFILE_CUSTOM_3=/path/on/hdfs/to/PYFILE_3.zip
17 # Livy Service - file dependencies to your Job. One example is the required
18 # Hive_site.xml file. Please list key-value starting with "FILE_CUSTOM" so
19 # they all get parsed.
20 FILE_CUSTOM_1=/path/on/hdfs/to/hive_site.xml
21 # Spark / HDFS Env Vars. These are used by the livyservice to tell the
22 # executors where to find necessary jars/bin for connecting to HDFS from
23 # within spark & tensorflow. These are also used by the tensorboard
24 # functionality of the server, to set the variables it too needs access to for
25 # using HDFS. These have been set to their typical values.
26 HADOOP_PREFIX=/usr/hdp/current/hadoop-client/
27 HADOOP_HDFS_HOME=/usr/hdp/current/hadoop-hdfs-client/
28 LIB_HDFS=/usr/hdp/2.6.2.0-205/usr/lib/
29 LIB_JRE=/usr/java/default/jre/lib/amd64/server/
30 # Tensorboard: Provide a command separated list of ports you'd like to be used
31 # for opening tensorboard instances. Note the more you provide here, the more
32 # instances that can be opened.
33 TENSORBOARD_PORTS=7770,7771,7772,7773,7774,7775
34 # Testing
35 # Set these parameters for the testing services
36 LIVY_SERVER_FACADE_PORT=fake_port
```

Figure 16 Sample .env file, showing how a large amount of configurations have been extracted and documented in a single file. These files can change on a per-user basis, allowing multiple pipelines to run at once with different focuses.
single point of failure and they could be accessed from anyone who could reach the cluster. As such, new environment variables were added to fix driver errors preventing HDFS interaction, paths describing where jobs can be found included, and the pipeline’s code updated to correctly save to HDFS. Now MLLib and TensorFlow can save and open files on HDFS, in addition to TensorBoard having the ability to directly interact with models stored there.

4.3 Queue

The queue was among one of the first things that was updated and improved on during the process of creating the production ready pipeline. The original implementation of the queue held all of a models metadata inside of it, which duplicated the copies found within the MongoDB database. Requests for metadata were taken straight from this Queue file, resulting in a confusion on the source of truth and potentially conflicting metadata entries should one change but not the other.

The decision was made to change the queue to instead hold only two pieces of information instead of everything. This would be a model's id and it date for next training, allowing the bare minimum needed for the Queue to function. The Queue was then abstracted into an interface, rather than operations directly on the file backing it occurring. This now allows the Queue to be abstracted into other forms if desirable – be it a database table holding a list of items to train next or some other form. This interface now defers to the MongoDB database as the single source of truth, meaning what the database has on file for a model is the sole copy.

The queue exposes the following methods in order for the pipeline to interact with it.

- **Enqueue**, which adds a new model to the queue and inserts it into the correct position in the queue.
- **FindNext**, which returns the first model in the queue ready for training, but does not remove it from the queue. This allows Task Parallelism to occur.
- **Dequeue and Remove**, which returns the first model in the queue or the given model id from the queue, respectively. This is used after a model’s training has completed successfully and is saved.
- **Get**, which returns a sorted array of every model currently in the queue – mostly used for testing purposes to validate queue behavior.

4.4 Data Ingestion

Data ingestion has seen an incremental improvement in the current iteration. Originally, the pipeline ingested data through a helper script that would parse a specific dataset into a predetermined Hive table hardcoded into the pipeline. This ingestion script assumed a very specific dataset would be used, describing which columns within it were numeric and which were strings. While this was a satisfactory start, the script was in a defunct state and left the pipeline unable to ingest any other form of data.
In the new iteration, there was more than one type of data to bring in. The original data was a single CSV time-series dataset of 1,734 transactions with 52 columns describing each record. A new dataset was introduced, again in time-series format, but with 1 Million records covering the span of two days and 223 columns describing each transaction. This data came in an offset format rather than comma delimited. A second new dataset was introduced later in the project, albeit a non-time-series dataset of 2.2 Million records with 4,820 unnamed pre-computed features describing each transaction – this time carat delimited. Beyond transactional data, system test data was desired to validate known models were training right for specific datasets. For this, the MNIST dataset was ingested as well.

Due to all of these datasets having different sizes, contents, type, and format meant the ingestion scripts and dataset flexibility would need improvements for the pipeline to become compatible. This was addressed through a series of abstraction scripts used to break apart the previous monolithic script. The first step was to abstract the ingestion process’s understanding of a dataset’s features and types. A new metadata parser and schema generator accomplish this task. The first is responsible for consuming a description file about a dataset, converting it into a Python dictionary describing the columns, types, and offsets (if present) about the ingesting data. The latter then takes this information and translates it into the Hive-specific datatypes necessary for creating the proper table.

The next step is to bring the raw data, with its descriptions, into a Hive table for feature conversion. These higher-level scripts are not designed to be comprehensive, given the wide variety of datasets and formats to encounter, but instead to serve as a framework for quickly building ingestion scripts composed of these smaller elements. For example, a raw transaction ingestion script was introduced to consume a CSV or convert an offset file to CSV to then bring into Hive. An alternative script for other forms of delimited files was introduced to allow custom delimiters to be specified. At the same time, it is also possible to combine these scripts just as easily – showing that the lower-level-helper scripts have served their purpose well in making ingestion easier.

As a final step, should it still be needed, raw data brought in that needs to be converted into “training compatible” can be done with the construct features script. “Training compatible” is nothing more than ensuring string types become indexed into numeric types, and any columns containing enough invalid data to cross a threshold are discarded. This step is not always needed, depending on the dataset’s nature. For example, our largest dataset consisted of purely precomputed and pre-validated features so it did not need any conversion.

This overall structure leads to a more flexible system that can be adapted to ingest a wider range of datasets with less development effort. Discussion in the later section “Model Trainer” will cover how the pipeline has been enabled to handle these new datasets. Additionally, specific steps to bring a new dataset in has been outlined in Appendix C.
4.5 REST API

The REST API is a series of REST services that are mainly responsible for connecting the pipeline with our Mongo Database and local queue file. To be specific, most of these services are directly responsible for creating, updating, deleting, and retrieving model information stored on the database with a few other REST endpoints used to update the Queue. As such, these services hold two major purposes: providing an easy way to update model information from the cluster and to gather information inputted from the user interface, making the REST API vital for user interface and pipeline interfacing. Table 2 lists the routing for each REST API service, and gives a brief explanation as to what each of the services does.
### Table 2

A list of all exposed REST endpoints that the pipeline exploits and their usage. The newly introduced clean routing has brought the API to be more REST-like while also enabling easy expansion.

<table>
<thead>
<tr>
<th>Route</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>/api/model/create</td>
<td>Adds a new model to the mongo database and inserts it into the Queue.</td>
</tr>
<tr>
<td>/api/model/delete</td>
<td>Deletes an existing model from the mongo database and the Queue.</td>
</tr>
<tr>
<td>/api/model/getall</td>
<td>Gets a list of all models in the mongo database.</td>
</tr>
<tr>
<td>/api/model/:id</td>
<td>Gets model information from the mongo database for the model with the given id number.</td>
</tr>
<tr>
<td>/api/model/retrain</td>
<td>For a model that needs to be retrained, creates a blank metadata file with an updated retrain date for the next training.</td>
</tr>
<tr>
<td>/api/update/modelinfo</td>
<td>Updates the author, deployed status, and enabled status of a model with a given id number.</td>
</tr>
<tr>
<td>/api/update/stats</td>
<td>Updates the various different stats (f1 score, accuracy, precision, hyper-parameter values, etc.) of a given model.</td>
</tr>
<tr>
<td>/api/update/training</td>
<td>Sets the training of the given model id to the given status and the time that this status was arrived at.</td>
</tr>
<tr>
<td>/api/features/get/:tableName</td>
<td>Returns an object describing the given table trainable columns, as well as the method in which its data should be obtained.</td>
</tr>
<tr>
<td>/api/features/list</td>
<td>Returns a list of all feature tables available for the user to train over.</td>
</tr>
<tr>
<td>/api/queue</td>
<td>Returns the current state of the queue.</td>
</tr>
<tr>
<td>/api/TensorBoard</td>
<td>Creates a new TensorBoard process, registering it with the global mapping of processes to IDs.</td>
</tr>
</tbody>
</table>

### Restructuring

While most current REST API functionality was in place with the prior evolution of the pipeline, much of it was either overly complicated, non-compatible with the new Livy Service, or extremely verbose. In addition, the old REST API was needlessly separated from the NodeJS
server that managed access to the user interface, which complicated routing for each of the services that existed at the time.

Revisions started with merging the server that manages the user interface with the REST API. To do this, a proper routing structure had to be created that would map each of the REST services to their own distinct path. This proved somewhat tedious, however, as the existing REST API had all REST services included in a single JavaScript file, which meant that we needed to manually go through the file and separate each service into its own JavaScript file. Once this was done, the ExpressJS framework was used to set up routing in a clear, maintainable, and concise manner.

**Unit Tests**

Unit tests were performed using Mocha and Chai. Mocha is a JavaScript test runner designed to prepare and run Unit Tests in an asynchronous manner. Powering the Unit tests is the Chai assertion engine – used to validate conditions during testing including HTTP calls. To evaluate code coverage, the Istanbul code coverage package was added. The three major user interaction portions of the pipeline were unit tested: the REST API, the Queue, and the Livy Service, while the remainder of the cluster was covered by larger scale system tests.

For testing the REST API a testing database was added to the MongoDB instance, allowing unit tests to create and modify metadata safely away from the production set. Unit tests worked with the Queue directly or through the REST API where appropriate to test the creation of, updates to, and deletion of models in the database. Queue unit tests looked to inspect that the proper order was maintained and that the information retrieved about models was accurate. Livy Service tests sought to ensure items were processed from the queue correctly and updated through the REST API accordingly. To control interactions with Livy during unit tests, a facade server was created. This allowed predetermined outcomes to be issued for specific models during testing to inspect all edge conditions.
4.6 Livy Service

Our implementation of Livy serves as a replacement to the CRON service used in the prior evolution of the pipeline. Originally, the way tasks were communicated from the web interface to Spark was via a CRON job. This CRON job would run as a background service looping every few seconds, checking for changes to the local queue file. If an item entered the queue, the CRON service would dispatch a series of Python scripts that would then run a spark-submit job to train the item in the Queue.

There were a few faults with this implementation. First, the usage of CRON could prove unreliable: if an error occurred when training a model, obtaining a model from the queue, or submitting a model the error could not be propagated back to the user to diagnose. Additionally, CRON runs as a system service, which requires maintenance and monitoring of its own for checking errors. Finally, since CRON runs locally on a host, it cannot communicate with any external systems. This forces the Queue system to live locally with the CRON job such that the CRON job can reach it (i.e., persistent to disk). Overall, CRON presents a solution that is inflexible and complex.

Figure 17 The flow of data through the newly created Livy Service. Living as a forked process from the Web Server the Livy Service can run multiple models at the same time to achieve Task Parallelism.

In the new implementation, our “Livy Service” (shown in Figure 17) lives alongside the Web Interface and API Server as a NodeJS subprocess. This forked process can now programmatically work with the Queue, rather than communicating through File I/O. Additionally, it can take advantage of the new environmental configurations introduced to the pipeline – specifying what job to run on Hive, the dependent files on Hive, and any executor configurations that should be forwarded (such as HDFS driver paths). Perhaps it serves as a testament to the pipeline’s level of abstraction that, in order to implement Task Parallelism, the only part of the pipeline that needed editing was the Livy Service. In other words, the backend could achieve Task Parallelism without change.
Once the Web Server is started and its configuration is loaded, the Livy Service is forked to immediately check for in-progress items. If nothing is known to be training, it will then programmatically ask the Queue for the next item to train. If there is an item available and ready to be trained, the model’s metadata is pulled from the MongoDB database via the REST API and sent to the configured job on Spark. The loop will continue up to the configured number of concurrent jobs the cluster should run. During each iteration from then on, the Livy Service will check the status of any running Spark jobs. Due to the asynchronous NodeJS environment, each of these loops occur at their own pace, resyncing back at the start to ensure no single thread gets too far ahead of another.

If a job is found to complete successfully, the service moves to train the next item in the Queue. If the job has failed, the Livy Service can take action on that failure; either by requesting additional error information to pass to the user or (in the current implementation) attempt to retrain the model up to the configured number of times. Should the model fail too many times, its metadata is marked as bad in the database and it is dequeued. This behavior is fully customizable in the Livy Service, should an alternate implementation be desired.

Through the usage of Livy, we solve CRON’s shortcomings. First, errors are received and handled by the system rather than left for the user to find. Second, the programmatic connection established between the Livy Service and the Queue allows the two to communicate directly rather than through File I/O. Additionally, this removes the need for the service to run on shell, making the entire implementation agnostic to the host it is run upon. The Queue’s implementation is free of any constraints caused by CRON, allowing it to be backed by database, file, memory, or other system. Finally, as a result of the programmatic connection to the Queue, future implementations are open to extract the Livy Service from the Web Server by changing this interface (be it REST or another means). In the previous File I/O and CRON implementation, this would have been a far less trivial task.

Overall, the introduction of Livy creates a more flexible implementation that can work with both Spark and other sub-systems of the pipeline fluidly.
4.7 Model Trainer

The Model Trainer portion of the pipeline required many improvements. This portion of the pipeline takes metadata for a model, and turns it into an actual machine learning model, saving it to disk, and reporting the model’s accuracy, precision, recall, false positive rate, as well as F1-Measure (as defined in Section 2.4) to the REST API for updating in the metadata database.

In the original implementation of the pipeline, models were stored on disk in a folder alongside the Model Trainer script. This was changed with the current implementation where the Model Trainer instead stores trained models to HDFS. Changes also needed to be made with respect to how data was prepared for training. With the new pipeline, the frontend allows users to be able to select how much data from a dataset that they would like the model to use for training, be that a number of rows, or a number of hours if the data is time series. They can also specify which test dataset to use when evaluating a model. Both of these selections had to be implemented in the Model Trainer, since it has the responsibility of creating the dataframe for the model to be trained and tested over.

Another improvement that was made in the current version of the Model Trainer is that all of the specific scripts for the different model types are now stored in a separate ModelType package. This allows the Model Trainer to dispatch jobs, after performing preparation stages, to specific ModelType implementations to be trained and evaluated. From a developer perspective, this keeps the implementation clean and more easily maintainable, which is necessary to make the pipeline production ready.

Dispatching to ModelType is done by inspecting the metadata of a model for the algorithm label, which determines the specific ModelType to run. For example, if the metadata is for an ANN, then the Model Trainer will specifically get the number of hidden layers, the batch size, and the number of epochs. The Model Trainer will have already created a Spark job, and will then hand that specific ModelType the Spark job, as well as the data and all of the additional metadata required by the model in order to be trained and evaluated.

Each separate ModelType returns the accuracy, precision, recall, F1-Measure, false positive rate, and the trained model. Once these statistics and the model are returned, the Model Trainer sends the trained model to HDFS for storage. It then updates the metadata for the model in the MongoDB database for its performance stats and location of the saved model on HDFS.
5. Deep Learning

5.1 TensorFlow and TensorFlowOnSpark Integration

One of the primary goals of this project was the addition of deep learning neural network models to the pipeline. Specifically, we wanted the pipeline to be able to train deep neural networks such as multilayer perceptrons by leveraging frameworks dedicated to the training of deep neural networks. Several frameworks found in our research included Caffe, SparkNet, and TensorFlow. The three were considered against their feasibility for integrating into Spark and the pipeline, in addition to how easily extensible they are.

Caffe (Convolutional Architecture for Fast-Future Embedding) is a deep learning framework that was first developed as a research project at UC Berkeley (Caffe, 2018). It has since moved to being an open-source project on GitHub. Similarly, SparkNet is a deep learning framework that supports distributed training of neural networks (SparkNet, 2018). SparkNet was also developed as a research project. Finally, TensorFlow is a software library open-sourced by Google that is primarily used for machine learning applications. It provides several utilities for performing dataflow programming.

During our research, we found that in comparison to SparkNet and Caffe, TensorFlow had more extensive documentation. Unlike Caffe and SparkNet, TensorFlow was primarily built for industry applications rather than as research projects. TensorFlow has a larger, more active community of developers working on it as shown by their GitHub repository—not to mention the corporate backing to fund its development. The reliability in its documentation and development is a critical business decision to ensure the library remains supported for future years coming. Finally, TensorFlow has TensorBoard, which is a visualization tool that can be used to actively debug and understand TensorFlow programs—a unique feature unavailable to others.

Having made the decision to choose TensorFlow, we needed to integrate it to work alongside Spark—as the pipeline operates on a Spark cluster. Using a lightweight framework developed by Yahoo, TensorFlowOnSpark, we were able to create Distributed TensorFlow programs that could operate on top of Apache Spark seamlessly with other operations. This removed the need to have dedicated servers for TensorFlow, allowing us to use our Apache Hadoop DataNodes. TensorFlowOnSpark operates lightly on top of Spark through three, high level steps:

1. The startup process allocates DataNodes to serve as workers for TensorFlow. These workers are given the program’s main function in addition to having listeners attached for data/control messages.
2. The Data Ingestion step manages getting data to executors:
a. TensorFlow’s **Reader**, **QueueRunners**, and **Saver** mechanisms are used to read/write data files directly from HDFS. This is used when restoring a model or saving one to disk.

b. Data taken from Hive is read into an RDD using **Spark SQL** operations. This DataFrame is turned into an RDD, which is passed to TensorFlowOnSpark. Inside, it is broken down into chunks to be fed into a Multithread-safe Joinable Queue instance.

c. Using TensorFlow’s `feed_dict` system, data taken from the TensorFlowOnSpark queue is fed into each worker one batch at a time. It is during this step that RDDs are broken down into Numpy format to be compatible with TensorFlow.

3. Finally, the shutdown procedure frees executors back to the cluster. This step occurs after data is written out to HDFS or sent to any callers.

![Figure 18 How TensorFlowOnSpark operates. Notice how the master provides each worker with a function to run, after which it then feeds each worker data that it can receive in that function to operate on. Each worker trains on a shared model on HDFS configured by TensorFlow.](image)

With TensorFlowOnSpark resolving the integration problem, that leaves TensorFlow itself to unravel. TensorFlow provides an API at different levels of complexity for implementing machine learning application. At higher levels, TensorFlow provides a DNNClassifier that allows for initial experimentation. However, in order to distribute training, a lower level implementation is required since the high-level implementations like DNNClassifier do not include support for distributed training. This involves specifying the tensors at each layer of the TensorFlow graph and their corresponding shapes and activation functions. While this requires more work, it empowers the user to take fine-grained control of their model.
Figure 19 Task Parallelism (Top) compared against Data Parallelism (Bottom). Notice that in Task Parallelism, multiple models are trained on multiple machines, but in Data Parallelism, many machines handle the same model.

When working with TensorFlow Distributed there are further considerations to make. At a high level, there are two different ways to parallelize training of models: Data Parallelism and Task Parallelism. While the latter assigns different parts of the model itself to different nodes, the former only distributes data across the cluster. Data Parallelism involves training mini-batches of data on different worker nodes that share the same model. We chose to implement Data Parallelism since it can be used to train models asynchronously (see Figure 18), allowing worker nodes to run independent of each other. Figure 19 depicts Data and Task Parallelism in contrast.
### Asynchronous vs Synchronous Data Parallelism

In synchronous, all machines train the model and wait before updating. In asynchronous, all models contribute their changes independently—due to the fact updates are commutative (Schlag, 2016).

Data Parallelism can then be broken down into two different ways to distribute the architecture of the neural network itself. They are:

- **Between-graph Replication**
  
  Each worker gets a copy of the graph. Workers only have in common the variables that are updated at each step (biases and weights) in the backend. By doing this, all the workers train the same model without having to share a single copy of the graph across all the workers. This is the current implementation of a deep neural network on our system.

- **In-graph Replication**
  
  In this scenario, a single graph is created on the distributed master. This graph includes all of the replicas included in the worker devices. This is simpler than between-graph replication but the graph can get very complex as the number of workers change.

  Because in-graph replications are not scalable when the number of workers increases, the between-graph replication was chosen for our TensorFlow multilayer perceptron model.

### HDFS File Storage

In order to maintain consistency with the ML and MLLib models on our pipeline, it was important that we stored our TensorFlow models on HDFS. This meant that the “model” or the common variables that were being updated during each step of training by all the worker nodes...
had to be stored on HDFS. We accomplished this by passing the HDFS file path into our TensorFlow DNN script and configuring the executors to point towards HDFS. In contrast, our Spark ML models are saved in the higher level “Model Trainer” scripts after they are trained, which allows them to simply use Spark to gain HDFS access.

By pushing the model storage responsibility to the lower level TensorFlow script, it was possible to access the model variables during the training of models. This was important for serving the model's accuracy metrics to the front-end as well as to the TensorBoard functionality, which is detailed below.

### 5.1.1 TensorBoard

TensorBoard is a visualization framework that helps visualize neural network models both during and after the training of a model. This helps in debugging and understanding model training easily. TensorBoard can be used by simply launching the TensorBoard program and pointing it to the location of the model on the file system— in our case, HDFS. Our implementation launches a TensorBoard instance by forking off a child process when a user requests it from the front-end of the pipeline.

With the current configuration of the TensorFlow neural network models, TensorBoard initially opens with the following visualized summaries of biases and weights in each layer (input layer, hidden layers, output layer):

1. Mean
2. Minimum
3. Maximum
4. Standard deviation

Additionally, the following metrics are also visualized in the summaries:

1. Cross entropy loss
2. Accuracy

All of these summaries enhance a user's ability to debug and analyze any model during training which saves time by revealing lower-level details about a model than a 'black-box' algorithm.
Another feature of TensorBoard is its ability to visualize the graph of the different working parts of a neural network model. With this interactive visualization, a user can easily understand the different interacting modules within a TensorFlow neural network model. This provides a lot of value for debugging larger and more complex models.
In addition to the line chart that displays the summaries during each step of training, TensorBoard also visualizes the distributions for the following parameters:

1. Pre-activations at each layer ($W \ast X + b$)
2. Values after activation
3. Biases
4. Weights

Figure 23 In addition to the 5-Figure summaries, distributions are plotted of each tracked scalar. This enables monitoring how values behave relative to the rest, pointing out potential outliers.

For a more information-dense view of some of the parameters, a 3D graph is also drawn for the histograms of the following vectors:

1. Pre-activations
2. Activations
3. Biases
4. Weights

This allows the user to view all the values in each layer contained in the above vectors, monitoring how they change with time and their overall distribution shape.
Figure 24 Histograms are also reported, to monitor how distributions change with time as the network trains. As time can be a factor in the training of networks, histograms provide insight that may not have been exposed before.

Figure 25 The Projector, when configured correctly, can be useful to see how inputted data sits in a 3-Dimensional space after PCA is run on it. This opens opportunities for identifying clusters in the data visually.
Given the close tracking of scalars, it is possible to watch accuracy and cost of a model change with each step trained over.

TensorBoard significantly adds to the value of the pipeline because previously there had been no way to observe model training. A model would have been sent in, with the only methodology for monitoring it being log file inspection. Through TensorBoard, a door is opened allowing the model’s intimate components to be carefully watched beyond just the metrics reported back at the end of training. With the addition of TensorBoard, it is possible now to visualize metrics like loss and the weights of the neural network between each step of training, allowing users to reformulate their workings should they notice anomalies.

### 5.2 Hyperparameter Tuning

Hyperparameter tuning, much like its name suggests, is the process of optimizing input parameters used by machine learning algorithms to best fit models for a given dataset. Input parameters that could be optimized, for example, include the number of trees to be used when training with random forests or the number of hidden layers that should be used when training with deep neural networks. Traditionally, this process would be done using a combination of cross-validation and grid search, but this has proven to be both extremely time-consuming and costly. As such, several new approaches have been created to reduce the time demands for Hyperparameter tuning without sacrificing accuracy. These approaches include random search and Bayesian Optimization Algorithms such as Gaussian Processes or Tree of Parzen Estimators. This MQP explores grid search, random search and Tree of Parzen Estimators as potential solutions to Hyperparameter tuning and each are discussed in the following paragraphs.
Grid search in combination with cross-validation is the most expensive of all the potential solutions explored. This process, like all other potential solutions, involves building many different models and comparing them against each other with the model that minimizes the specified loss function best being the one returned. The difference with grid search, however, stems from the fact that it is an exhaustive search. Meaning that grid search will train all possible models made by a combination of pre-specified parameters. This creates a large search space, with each trial taking exceedingly long. It is not until all models are done training that the one minimizing the loss function (inverse f-score for us) is returned. Ultimately, this will lead to a best solution for the range of hyperparameters given, but will cost the most amount of time to complete. With this being stated, other solutions that are faster that maintain similar accuracy would be ideal in place of grid search.

Random search is a process that aims to solve the efficiency issues that plague grid search. To do this, rather than create all possible models in a pre-specified range, random search randomly chooses a number of parameters to compare in trained models. As might be expected, this dramatically expedites the process of searching for hyperparameters, but in doing so, potentially misses finding the most optimal hyperparameters for a model.

Lastly, Tree of Parzen Estimators chooses hyperparameters to test based upon how the previous test performed, thus making an educated guess to narrow the search space of hyperparameters after each trial. As such, this allows Tree of Parzen Estimators to get an accurate prediction without sacrificing time, as not all hyperparameters will need to be checked to determine which the best is. As such, many studies - this MQP being among them – determine that using Tree of Parzen Estimators often works well to find hyperparameters that best optimize a model.

### 5.2.1 Implementation

Each of the above methods to calculate hyperparameters has been implemented into the pipeline using various different open-source libraries provided by the MLLib and HyperOpt frameworks for Python. Unfortunately, however, MLLib’s rendition of grid search is not readily compatible with TensorFlow, and, as such, grid search has not yet been implemented to compute hyperparameters for a model that is being trained with a TensorFlow deep neural network. Given the poor performance Grid search holds on the algorithmic level compared to the alternatives, this is seen as a non-issue. Other than this exception, grid search, Tree of Parzen Estimators, and random search all can be utilized to compute hyperparameters for each training algorithm available in the pipeline.

With all of these methods, the biggest hurdle for implementation comes down to adequately choosing what ranges of input parameters each method should look through for an optimal solution. If the range is too small, for instance, there runs the possibility that all values in the chosen range are poor selections for the parameter, meaning that the model, even with hyperparameter tuning being done, could conclude with a low f-score. Conversely, if the range is too large, Hyperparameter tuning could either take a very long period of time to complete (grid
search) or result in poor selections (random, Bayesian). Either way, an adequate range needs to be chosen for implementation.

### 5.2.2 Comparison

Hyperparameter tuning methods have been well explored over the past ten years, and a plethora of results comparing different tuning methods exist from numerous reputable sources. One such study, *A Stratified Analysis of Bayesian Optimization Methods*, directly compares the pros and cons of using grid search, random search, and Bayesian algorithms to determine if there is any notable benefits and drawbacks from using one versus the other. The below images captures some of their results.

![Comparison of Hyperparameter tuning algorithms](image)

Notably, the study did not focus on time, but rather on top-performing models. As can be seen, Bayesian algorithms such as Tree of Parzen Estimators consistently outperform both Grid and Random Search algorithms.

---

*Figure 27: A comparison of the various Hyperparameter tuning algorithm implementations. HyperOpt has shown to be one of the top performers. More importantly, however, Bayesian algorithms have shown to be the most effective at finding the best models.*
5.3 System Testing

When developing in a distributed manner, it is often troublesome to determine a specific source of error. Unlike with single host applications, distributed applications often have log files in multiple locations, errors originating from many different process, and uncertainty is whether problems originate from the User’s program or some configuration error. To remove the uncertainty and validate specific parts of the pipeline/cluster are configured correctly, these system tests were introduced.

5.3.1 Configuration Check

One of the most common datasets that is often used as a standard for neural networks is the MNIST hand-written digits dataset. This dataset contains 60,000 image examples to train models over. Each data point is an image of 28x28 pixels which means that the data contains 784 (28x28) features.

![Random Sampling of MNIST](image)

This test dataset served as a control variable for us during varying stages of the project. For example, when first integrating TensorFlowOnSpark we chose to copy their provided example of training a distributed MNIST model to our cluster. Using this example, we could run the training manually to validate executors were configured properly to receive and operate on data. Later, we loaded this test onto HDFS to help configure Livy, validating we could launch and train on Spark in cluster mode rather than just a client-side Spark-submit command.

5.3.2 Pipeline Integration Testing

Once the necessary configuration is established on the cluster for training TensorFlow models, the next step is to make sure that the model trains properly when loaded into the pipeline itself. This lead to one of the first major improvements in the pipeline itself – the ability to work
with other datasets beyond a specific time series set. Since MNIST is agnostic to time, and its format a different set of columns and types, new ingestion scripts and user-selections were added. The frontend received the ability to select non-time series data and the backend was abstracted to work with whatever features the user had selected.

With the improvements made, we could then utilize this dataset during the training of our own created models (such as the TensorFlow Multilayer Perceptron) to validate we had constructed the network correctly. Since the MNIST dataset is a well-studied set, we were able to train known models that achieved a specific performance both on a single host and in our distributed program to validate they both achieved similar accuracies. In the model we chose, a single hidden layer of 500 nodes with the ReLu activation, we expected an accuracy of 92% - which was achieved.

5.3.3 Running TensorBoard Instances

In order to ensure that TensorBoard can look at the location of the model on HDFS, TensorBoard instances can be spawned from VMs that can point to the HDFS location. By doing so, and observing the corresponding spawned TensorBoard instance, it can be deduced that model locations on HDFS are discoverable and that the data is being stored into these locations accurately when training a model on a well-known dataset like MNIST. These tests proved the most trying yet valuable, as there was little help from error logs as to how the TensorBoard was unable to interact with HDFS. Tests such as this one enlightened the necessary environmental configurations each executor must have in order for HDFS to become readily accessible.
6. User Interface

The pipeline’s main point of interaction is through its web-based interface. Since the prior iteration, this interface has received several updates in visual aesthetic and underlying programming cleanup. A large defect backlog has been addressed, including adjusting visual alignments and limiting the state being tracked in the browser. In particular, the latter caused corruption of the Queue and database. Other behind-the-scene changes included simplification of the underlying DOM and removing excess structural and form elements. The most significant changes, however, come in the form of the visual and user experience aspects.

6.1 User Experience

To reflect additional features added in model generation, the user interface received additional controls and flexibilities. As pictured in Figure 29 the new TensorFlow algorithm was added as a selectable option. Underneath this, the fine grain model form appears in the same layout as the MLLib based models. This allows the user to set parameters such as steps of training, the size of each training batch, the learning rate the model should use, and the activation function to apply at each layer. Layer customization received a refresh, allowing the user to specify as many hidden layers as they wish and the size of each.

With the addition of this form for Hyperparameter selection, an “auto tune” toggle was added to aid in optimizing these values. As depicted in Figure 30 activating this toggle removes the additional fields from the User’s view, making it apparent they wish to have the Model

Figure 29 Additional dialogues appear in the interface, giving access to new model types and Hyperparameter tuning algorithms.
Trainer handle fine-tuning their layer sizes, activation, step count and other parameters. While not pictured - as the feature was added after the visual overhaul - an additional dialogue is presented to the user where they can select which method to auto tune parameters – be it random, grid, or Bayesian based search methods.

New datasets of varying formats were also integrated. This lead to the changes in the REST API as previously outlined, enabling the frontend to dynamically update based on the type and format of data the user selected. For example, as shown in Figure 29 the right image shows the MNIST dataset selected – a non-time-series dataset. Previously the pipeline did not support this kind of data, as it assumed the user would want to choose a date range to select their data over. To increase flexibility, this was reminded by using information passed from the REST API about the selected dataset to determine what way the user should select their data. In the event of a non-time-series dataset like MNIST, the user can select the amount of data they wish to train over. If it is all, they are prompted to enter ‘-1’ as a signal to the Model Trainer. As shown in Figure 30 however, if the user has a time series based dataset then the option reverts to its previous “back-N-days” behavior to determine the time range of data to train over.
In addition to the data selection changes, Figure 31 also shows changes made in the way users select their features. Previously, features were a static list encoded into the interface. This has been replaced with a more flexible REST based approach where selectable features the user is allowed to choose from is sent from the REST API after dataset selection occurs. This lets the frontend adapt based on the dataset. Should feature selection not make sense, such as the case of MNIST, the dataset’s metadata in the REST API can be configured to prevent the user from selecting features or labels. If the user should selection features and a label, however, the valid options can be specified server side based on the REST API and displayed as shown in the figure.
The model view also received a new feature: the addition of Launching and Closing TensorBoard instances for TensorFlow enabled models. Figure 32 shows the new button placements, living alongside the previous bank of buttons. These follow the same styling as the others to soak into the interface. Upon pressing one of these, a request is made to the REST server to lookup or spawn a new TensorBoard instance for the model. The response given to the interface redirects the user to a new tab pointing at the TensorBoard web page. This allows the user to inspect neural networks more deeply, taking advantage of powerful visualization tools exposed by TensorBoard right from the web interface. As an addition, the ports these TensorBoard instances run on are configurable via environment variables - allowing the number of instances to be controlled.

Overall, the tighter integration with the REST API this pipeline iteration, as made apparent, has brought greater flexibility to the interface. Most of these changes maintain the excellent job the previous MQP group had done on the user experience, keeping its usage of spacing and intuition intact, while expanding the functionality for future usage.

6.2 Visual Overhaul

In response to the new TensorBoard integration and Deep Learning shift, the pipeline received a visual upgrade to bring it more in line with the styling of TensorBoard. Since TensorBoard serves as the new hub for model monitoring, inspection, and debugging in the pipeline we wanted to ensure consistency across both interfaces for a cohesive user experience. In its previous state, the pipeline’s dark blue colors left the two tools feeling like polar opposites.
(which comes as no surprise, given blue is the complimentary color of orange). After discussions with the sponsor, it was decided the pipeline should be brought more in line with TensorBoard – the brighter orange serving as a means to increase the flair, enthusiasm, and continuity of the pipeline.

Figure 33 The TensorBoard interface, monitoring simple statistics about the currently training model’s individual layers and final output.
As shown in Figure 34 the new pipeline’s homepage aims to maintain much of the prior’s implementation, with changes heavily coming in the form of visual aspects such as color and font. Specifically, the greatest change was the location of the navigation bar to reflect that of TensorBoard’s, which is shown in Figure 33. As identified by the previous MQP group, it is important to maintain related items near one another to ensure the user could understand how to accomplish simple tasks. Thus, the concept of tabs was maintained, but simply lifted to the top of the page to maintain continuity as a user viewed both TensorBoard and the pipeline (Esco et al, 2017).

Additionally, the font of the pipeline was changed to use Roboto for headings and Open Sans for inline text. This reflects the same fonts used in TensorBoard, helping to maintain a smooth transition between tools as the User swaps between them. Colors were updated, using the bold orange highlights as a means to draw attention to separate elements of the interface as TensorBoard does. In particular, the user’s attention is drawn immediately to the navigation pane – grounding their sense of direction in the interface. This gives concreteness to navigating the three views established by the prior MQP group. Smaller amounts of the same orange are then used elsewhere in the page, as seen in TensorBoard, to show where different parts of the form start. Next, buttons and toggles were changed to use the saturated blue TensorBoard uses for clickable elements, in effort to keep the user’s intuition the same for interacting with either page. Finally, the background colors were changed for a high luminance blue to shades of gray. Using
a more neutral, low luminance tone such as gray increases the contrast on of the page, which makes identifying interactive elements within the form much simpler.

These changes were reflected to the View and Queue pages, as shown in Figure 35. As indicated by the prior group, the View and Queue pages are very similar; however, the Queue page is a subset of the View page showing only models currently preparing to be trained as compared to all models created. We recognized the importance of keeping transparency with the user, so as such these views maintained their original functionality but simply received the same font and color updates (Esco et al, 2017).

The final view, the model details, was also given an update from the prior iteration. As indicated by the prior group, this view will adapt to reflect the parameters of the specific type of model it is displaying. This functionality has been maintained in the exact same manner. In addition to the color and font updates, however, the new TensorBoard buttons received an update.
to reflect their importance on the page. The individual sections of the UI also received a standardized sizing and a drop shadow, as a means to bring emphasis to them through contrast. The layout was also cleaned up (see the “Displayed” and “Enabled” toggles) by updating the underlying code to use CSS Flexboxes. This lets the elements shift on the page dynamically based on the number and size of other elements – so if future iterations add more statistics they can fit with ease. This change also reduced the amount of complexity in the code, as an added benefit.
7. Results

7.1 Pipeline Data Parallelism Scalability

One of the major draws of the pipeline is reducing training time through the usage of multiple executors. It was important to determine how effective additional executors are at decreasing training time, and at which point the training time would plateau. A series of trainings we run with sequentially increasing executor count to determine the effectiveness. To ensure executor counts were the only difference, each model was trained with the exact same specifications and over the course of several epochs to ensure enough time was spent training a model.

We decided for this section of testing to use the MNIST data set, due to its certainty in what accuracies a model of certain specifications should be producing. The model tested on was a Tensorflow based Multilayer Perceptron of one hidden layer consisting of 800 neurons, activated using the ReLu function. Batches of 128 were fed in over 5 epochs. The AdamOptimizer was used with a base learning rate of 0.001. Given these specifications, the accuracy of the model should be around 96% for every one of our trials. Trails varied using 2, 4, 6, 8, 10, 12 and 14 executors. Each trial was run three times, ensuring the recorded data was precise in the event of an outlier training.

The results of these tests can be found in Figure 37. Given the accuracies for models trained remained within a margin of error of the expected percentage (given the asynchronous distributed manner) the results show a positive correlation that only impacts time – not accuracy. Increasing the number of executors increases the training speed for a model.
It should be noted, however, the red outlier to the trend in Figure 36. This outlier’s implementation does not operate through the Distributed TensorFlow API, due to the need for more than one executor for the API to operate. Thus, its implementation is with a different API, optimized for single threaded performance. The outlier’s training result, however, is as expected for single threaded performance. It is important to distinguish the benefits of distributed training as compared to single threaded. Distributed training allows the handling of larger datasets than single threaded implementations – for example, the single threaded implementation would not be able to handle our larger datasets of 55GB or more, given the entire dataset must fit in memory. The benefit of distributed training is that only the data of a single batch must fit in memory.

One thing of note as well with the single threaded run of the training is that it got a much lower accuracy than the distributed version. The single threaded run achieved an accuracy of only 82% while the distributed runs were getting accuracies around 96%. This could be caused by the implementation of the single threaded itself, such as the difference in batch size discussed earlier.

Thus, depending on the data use case, the need for distributed training compared to single threaded suddenly changes. More executors are not necessarily always better – but a key takeaway from the test is that the pipeline can utilize both single executor and multi-executor models, while training multiple models simultaneously. Additional exploration in selecting single threaded versus multi-threaded models based on data sizes could prove fruitful for optimizing executor usage. For big data purposes, however, distributed training is beneficial.

To explain the convergence towards training times past 12 executors requires taking a deeper look at what goes into a training. Diminishing returns result from training time converging to 7 minutes and the preprocessing steps taking approximately 3 minutes in this MNIST example. This latter preprocessing part also does gain as much of a performance increase with the addition of executors, thus creating a baseline.
Time to train broken down into parts

Figure 38 A chart showing the time to train the standard MNIST model broken down into the three major groups that make up the training. The actual training of the model, the testing of the model on the test set, and moving data to the different executors. Notice how the “Moving Data” values do not change between trials, and the “Training” value does not decrease linearly.

7.2 Hyperparameter Tuning Tests

One of the major features of the pipeline is the ability to set a model training to use Hyperparameter tuning instead of setting the parameters by hand. To evaluate the ability of the Hyperparameter tuning to find a well performing model in a reasonable timeframe, we ran tests over our known dataset (MNIST) and then our unknown dataset (RS). Through starting with MNIST, we can validate a model known to be well performing found or better. Then we could explore the RS dataset, to determine what kind of well performing model could be found in addition to determining if reducing features using either PCA or Random Forest importance could help.

The first time running MNIST Hyperparameter tuning the ending accuracy was 28.16% and took 2 hours and 8 minutes to complete over 20 model evaluations (the number of models to attempt within the search space before stopping). This first test’s poor performance was the result of an ill-defined hyperspace in which to select Hyperparameters from. In other words, since a high performant model was not within the search space the Hyperparameter tuning algorithm could not find a high performing model. Thus, we learned the ability to adapt the Hyperparameter search space is a necessary use case, and as a result, we implemented the ability to set the hyperspace when requesting to tune hyperparameters automatically. The follow up test, with an adjusted space straddling the known model to be well performant (but wide enough to
find a better model) was able to find a model performing at 97.35% accuracy. This surpassed our known model, consisting of a single layer with 4,500 nodes activated by ReLu. The time to find this model was only 4 hours and 45 minutes.

Moving into the various form of RS sets (that is, the entire RS dataset, a subset selected by Random Forest importance, and a dimensionality reduction done through PCA). Using the full dataset first, it was decided to run an additional 30 evaluations (a total of 50) in hopes of finding a strong performing model. It was found through increasing the number of evaluations, and due to the far wider RS dataset (4,820 columns compared to 784 in the MNIST set) that the training time took much longer per model. MNIST models took 15 minutes to train each; however, RS models took an hour and 10 minutes. Thus, the total time for searching the Hyperparameter space took approximately 58 hours for all 50 models on the full RS dataset.

Due to an error in cluster configuration causing a failure to allocate an executor for TensorFlow on Spark, this first RS Hyperparameter training was cut short 32 hours in. Even with the training cut short, however, it was possible to examine the training logs to determine the most accurate model it had found so far. At the 32-hour mark, only 55% of the way done, Hyperparameter tuning had found a model with 91.1% accuracy on the RS test 1 dataset. Since the training was cut early, it is impossible to know the parameters this model was trained using. Additionally, the search space was not recorded as the introduction of user specified search spaces was in development during the progress of this test.

After running additional tests, the team decided to loop back and run the full RS set through another round of Hyperparameter tuning. In 72 hours the best model that was found was a batch size of 500, sigmoid activation function, a learning rate of .001, and 6 hidden layers. The hidden layers had the sizes: 400, 50, 350, 450, and 50. The accuracy found was 91.16%. The team decided to run this same model again by itself and to give it 50 epochs of data to train over. This version of the training ended with an accuracy of 94.01% a plot of the accuracy and loss of this run can be seen below, as well as a table with all statistics for the final model.

<table>
<thead>
<tr>
<th>Statistics for 50 epoch full dataset</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Accuracy</td>
<td>94.01%</td>
</tr>
<tr>
<td>Precision</td>
<td>94.96%</td>
</tr>
<tr>
<td>Recall</td>
<td>94%</td>
</tr>
<tr>
<td>F1</td>
<td>94.32%</td>
</tr>
<tr>
<td>FPR</td>
<td>11.74%</td>
</tr>
</tbody>
</table>
In time for the Random Forest and PCA RS tests, the user could define the search space. For both of these tests, a hyperspace consisting of the following was used: 50 evaluations, a maximum of 10 hidden layers, batch size between 200 and 400 with a step size of 50, a hidden unit range of 50 to 500 with a step of 50, and an epoch range of 3 to 7 with a step size of 1.

The model found in the Random Forest based test had 85% accuracy, taking 7 hours and 15 minutes to run evaluations. The model that was found was a batch size of 128, sigmoid activation, a learning rate of 0.001, 2 hidden layers of size 100 and 200. The accuracy and loss charts final model of the Hyperparameter tuning can be found below, as well as statistics for the final model. While there is some noise and spikes in these charts, the accuracy chart does show an upward trend, and the loss shows a downward trend. It is possible the training set needs to be more tailored to the designed model for better performant training, or a longer epoch evaluation to smooth out the curve as examples become seen more and more.

<table>
<thead>
<tr>
<th>Statistics for Random Forest Hyperparameter tuning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Accuracy</strong></td>
</tr>
<tr>
<td><strong>Precision</strong></td>
</tr>
<tr>
<td><strong>Recall</strong></td>
</tr>
<tr>
<td><strong>F1</strong></td>
</tr>
</tbody>
</table>
The final Hyperparameter test to be run was the PCA version of the full RS dataset. Hyperparameter tuning finished in 11 hours and 10 minutes. The best model it was able to find in its 50 evaluations was as follows: batch size of 200, sigmoid activation function, a 0.001 learning rate, 7 hidden layers, with sizes of 200, 100, 50, 100, 250, 100, and 250. This model achieved an accuracy of 88.22% using the PCA test dataset, suffering a similar issue as the Random Forest test with spiking. It is notable, however, that the fluctuation in accuracy and loss is more controlled as compared to the Random Forest set, and as such, it may be that PCA could prove more effective at training in fewer epochs. The accuracy and loss charts for the final run of the PCA Hyperparameter tuning can be found below, as well as a table for model statistics.
Looking at the three runs of the hyperparameter tuning on the different datasets the team decided to run an additional test on the PCA dataset. Using the same parameters as were found in hyperparameter tuning the team trained that model over 70 epochs instead of the number of epochs used in hyperparameter tuning. The accuracy found for this model was 90.41%, and took 2 hours to train. The accuracy and loss charts can be seen below, as well as a table for model statistics.
Statistics for PCA 70 epochs training

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Accuracy</td>
<td>90.41%</td>
</tr>
<tr>
<td>Precision</td>
<td>91.6%</td>
</tr>
<tr>
<td>Recall</td>
<td>90.4%</td>
</tr>
<tr>
<td>F1</td>
<td>90.89%</td>
</tr>
<tr>
<td>FPR</td>
<td>26.40%</td>
</tr>
</tbody>
</table>

While these specific three runs of Hyperparameter tuning might not create perfect models, they do prove the concept that even with just 11 hours, and no knowledge of what might work, a model can be created on these datasets with 85%+ accuracy in a relatively short amount of time. Finer grained tuning of the model type and training could produce more accurate models, but the high baseline demonstrated here is promising there is room to grow for a well-performing model. Additionally, these tests have shown the effect of data width can have on training time, showing that narrower datasets such as PCA or Random Forest can run faster than much wider sets. Feature selection remains valuable for timing, in addition to accuracy.
7.3 Parallel Training Tests

Another feature that was tested was training models in parallel. This meant having multiple models training at the same time, and recording if anything changed when increasing the number of concurrent trainings. The experiment that was run used the standard MNIST model we discussed in section 7.1 again. The first stage was to set the max concurrent jobs in the Livy Service to one (via environment configuration), train the model using three executors, and record its accuracy and timing. This served as the control for subsequent trials. Each trial thereafter used the same model specifications, but increased the max concurrent jobs to 2, 3, 4 and 5. In each trial, the corresponding number of models was submitted simultaneously via multiple webpage requests. The results are shown below.

![Graph showing the relationship between number of parallel model trainings, and the time for all models to complete training. Time holds steady for 1-4 parallel models because all complete in the same amount of time. Five models take 72 minutes because of a Spark configuration restricting a single user to submitting four jobs at once – thus the fifth trial had to train the last model after the other four.](image)

Thus, for models using three executors, the training time held constantly for 1 through 4 executors. This is expected given the executor availability of the cluster. When running five models, however, it was found that a Spark configuration restricted a single user to submitting at most four jobs at once, thus causing the last model of the trail to be trained after the first four. If this configuration was corrected, the testing shows that so long as there is enough memory and executors within the cluster the number of models that can train concurrently scales linearly. Running models in parallel does not affect training time of others, nor their final accuracy.

7.4 Flexibility Evaluation

Evaluating the flexibility of the pipeline requires identifying the key areas where the pipeline should be expanded or contracted. These areas should be simple to expand upon,
meaning they require little effort relative to other possible changes. In addition, these areas should be abstracted from those in which they interface, preventing dependency on preexisting conditions should one area need to change dramatically.

The areas defined in need of flexibility are as follows. The datasets in which the pipeline train over should not require any specific shape or schema, allowing the pipeline to expand beyond just fraud cases. The usage of the MNIST dataset, an image based dataset, is a good example of dataset flexibility. The details behind the ease of adding a new dataset is covered in Appendix C, but at a high level simply requires the dataset to be ingested into Hive, string indexed, and the corresponding endpoint in the REST API updated to reflect its existence. It is possible to take this flexibility a step further through the usage of Livy Sessions to query for what tables exist and columns within – thus removing the need for this façade pattern. Due to all interaction with data stemming from the REST API (in terms of selecting data) and the Model Trainer (for processing it) the locality of how data is operated on is contained well – requiring few edits on how data is processed should they even be necessary.

The concrete implementation of the Queue interface is the next area of interest. Allowing the Queue to sit behind an interface removes dependency on how the queue exists, should that be a file on disk, an object in memory, or a table in a database. This defines a contract by which the Web Service can interact with the Queue, allowing its implementation to persist by the needs of the business. In its current form, the Queue is backed by a file. Should the Queue need to be hosted in a diskless location, however, the new interface allows its implementation to be swapped without affecting any other systems.

Adding a new field to the metadata should be simple, such that new behaviors within the Pipeline are specified without sweeping changes to the entirety of it. How to add a new metadata field is covered Appendix C, however as an overview, a new metadata entry is added by editing the Create endpoint of the REST API. How the metadata field is populated is up to the developer, but the simplest means is to add a new field in the user interface – another change requiring two additional file edits. This consolidates metadata creation and manipulation to the frontend only, allowing the remainder of the pipeline to take action on specific metadata document fields without having to update any of their systems. Additionally, the usage of NoSQL to store metadata allows specific documents to grow and shrink based on what fields they need – not all fields must be present within a document for success. The few required fields are distinguished in the create endpoint, to ensure they are added to all documents.

The types of models the pipeline trains over should be easily added to, allowing new types of models to be experimented on without altering existing ones. The introduction of the ModelType package addresses this concern, removing the implementations from the Model Trainer into their own subpackages for easy version controlling and expansion. Adding a new Type of model is covered in Appendix C, with the gist requiring edits to the Create endpoint and frontend (such that the user can select the new model type and its metadata be created). Next, the creation of the model should be implemented in the ModelType package. Finally, the Model Trainer should be edited to send jobs for that Model to its implementation. This is the most
impactful change that can occur across the pipeline, yet the addition of TensorFlow and Keras based models have shown the ability for the pipeline to take new model types on easily. A minor trade-off but with the gain of easier experimentation.

The *environment* the pipeline runs within is the final flexibility point. The pipeline should be deployable to a variety of host and system configurations without needing to have its core code edited. This has been addressed by the introduction of an environment configuration file, which can be supplied with each deployment of the pipeline. This allows the systems within the pipeline to have their settings changed by editing a single file, rather than each reference throughout the entire system. Additionally, this now enables alternate versions of the pipeline to exist on the same set of executors – as demonstrated throughout this project’s development. This enables a test environment, a production environment, and a development environment to all utilize the same hardware – with the stipulation that they will still need to share their executors. Using the Spark YARN manager overcomes this stipulation, by serving as a secondary Queue for jobs on Spark; holding jobs until executors become available for the taking.

Overall, the introduction of these changes have made the system more robust and adaptable. While there are still further improvements to be explored, such as using Livy Sessions for data set information gathering or reducing the impact of adding a new model type, the important aspects of the pipeline remain flexible. New models and data can be tested without requiring vast rewrites of code, and changes to underlying hardware will remain agnostic to the pipeline.
8. Conclusion

This section will go over the team’s concluding thoughts on the project. Including a list of our accomplishments in completing this project, as well as our thoughts on the team experience, and finally our own recommendations on where to go next with this project for whatever team takes over after us.

8.1 Summary of Project

One of the major project achievements was producing a production ready pipeline. This meant that a lot of work had to be done in taking the pipeline from the previous group and bringing it to a production ready state, as well as moving from a single node cluster to a multi node cluster. A large chunk of hard coding was removed in favor of environment variables and proper information sharing between components. The queue management was rewritten to utilize the database as a single source of truth for metadata, rather than storing a duplicate copy within the queue. These changes opened a way to making the pipeline more modular, shifting its focus to a metadata driven system that will adapt to what the metadata says rather than force a specific behavior.

The REST API received restructuring that detangled its endpoints from a single monolithic file format into a cleanly routed structure. Complications and inefficiencies in database interactions were reduced and simplified. For example, a single insert and update operation was being performed as an insert, then find, then update – when a single call was only needed. Over usage of MongoDB calls was a typical problem, resulting in highly nested code. Where possible, calls were reduced, and where not, JavaScript Promises were brought in to reduce the nested-ness of the code.

The implementation of Livy itself was a large part of creating a high-quality pipeline. The Livy service replaced a CRON job that the old pipeline made use of, this change allowed for monitoring of training jobs, as well as the ability to submit jobs to a multi node cluster, instead of the previous single node cluster.

The system to ingest data into the pipeline also needed to be updated. The existing method was a broken script heavily tailored to an unnamed dataset the previous team had received. We have since scrapped this script and started over, creating a suite of Python scripts to help format and ingest datasets of different types. While they do not encompass every dataset possible (such a task would be extreme), they abstract what is possible to support ingesting offset and delimited based file formats – as seen in the RS and TFDescription dataset. Appendix C provides additional details around this suite.

The Model Trainer also was made more flexible to accommodate different model types, allowing for easy addition of model types such as a TensorFlow ANN implementation. Finally, the Model Trainer had to be changed to save models to HDFS instead of the local file system. This allows models to be more flexible, so storage is not limited to only being on a local machine.
In addition, the scoring engine was updated inside of the Model Trainer, with these scores being sent back to the REST API for storage. Hyperparameter tuning was also added to the Model Trainer as an option selectable in the frontend. This means that instead of the user hand selecting every input parameter, the Model Trainer will try out different combinations itself, searching for the most accurate set of parameters.

The frontend of the pipeline also saw significant effort throughout the project. Many of the changes made in the backend of the project had to have changes made to reflect them in the frontend. For example, with the addition of support for multiple data sets, a drop down had to be added to allow a user to select the dataset they would like to train over. Visual glitches were also cleaned up and fixed throughout the project, as well as a design change was made to emulate the UI found in TensorBoard for the pipeline frontend itself, creating a more cohesive look throughout the frontend.

Beyond infrastructure based changes, the project made several strides in expanding the possible machine learning methods for the pipeline to utilize. The introduction of the ModelType package (and its accompanying changes) has allowed the pipeline to run any concrete ModelType implemented. With TensorFlow integrated, this means a host of machine learning methods can be brought in – as demonstrated by our TensorFlow Deep Neural Network and Recurrent Neural Network implementations. Easier to prototype models are made possible via the higher level Keras API TensorFlow brings, or finer grained control with its core API. This has shown that the pipeline is compatible with a variety of models, be it lower level and also high-level abstractions.

Perhaps the most critical takeaway, then, is that while the pipeline is capable of turning out accurate models – its unrivaled flexibility holds uncapped potential. Should a model not be implemented of the type desired, a new one is easily added without major rewriting. Should a new approach of evaluation be desired, it can be achieved down to the model-specific level through the ModelType package. The major limitations no longer lie in implementation, but in the libraries in-use.

Future changes will not break prior models thanks to the NoSQL approach. The usage of HDFS makes models safe from loss. Finally, the introduction of TensorBoard has opened opportunities towards the exploration of stronger models with visual feedback – be it those models are for fraud or other business applications.

### 8.2 Team Experience

Our group worked effectively together, completing project tasks in a timely manner, and supporting each other along the way. Many MQP projects need to move in to D term, but with how efficiently the group worked, we were able to complete the project within the allotted 3 terms to complete the project.

A large part of this success is due to the use of Trello. Members always knew the tasks they had to work on for the next checkpoint. Cards on Trello were assigned to members with due dates attached, members had to make sure they completed each of their assigned cards by each
card’s deadline. Another practice that helped to add to the success of the team was holding frequent meetings. On top of the once a week meeting with the advisor at WPI, and the longer meetings held once a week at ACI, the group met a minimum of two other times during the week to check in on current tasks, and determine the next steps. This constant feedback loop, as well as system of check-ins helped to push the project along at the pace it did.

Group members worked to their strengths, and made sure they completed tasks to the best of their ability. That said some group members had stronger technical skills, and were therefore able to contribute more to the project.

8.3 Recommendations and Future Work

This team did complete a large amount of work, laying the groundwork for many interesting directions for future work. Some features or stretch goals were unable to be met. Also, while finishing the project other future goals were discovered, that a future team could work through as yet another iteration of the machine learning pipeline.

**Alternative Use Cases**

While the pipeline was originally designed for detecting credit card fraud, the smooth integration of the non-fraud related MNIST image dataset has shown the pipeline is capable of using datasets for use cases beyond just fraud. One topic discussed during the project was trying to mine other data from the credit card transactions using the pipeline, such as if you could predict buying habits of users for targeted advertising or the classification of new, growing trends based on what items are purchased in a time period. Investigating these other applications could prove beneficial.

**Deployment of Trained Models**

The pipeline was originally envisioned as a means to not only train models, but then prepare them for running inference (or accepting new input automatically) based on some trigger. For example, a REST API call to the pipeline for a specific model and dataset would take that pre-trained model and run inference on that specified dataset. This functionality was brought into the metadata as the “Deployed” field, but the previous group never fully implemented the rest. While out of scope for this current project, we recommend finishing this aspect of the pipeline through the addition of a few more REST points and Spark scripts that would load an existing model into TensorFlow from HDFS, run inference on the noted dataset, and return the result. This would fully enable the pipeline for taking action on new data, rather than just training models.

**Implement a Model Manager**

The current implementation is capable of storing models on HDFS for a long time coming. With that comes the stipulation of how to handle a growing number of models. First,
excessively old models would need to be removed from the system. Adding a reaping service such that occasional jobs are submitted to Livy to clear out outdated, unused models from HDFS would resolve this problem. Since the metadata knows when a model was last trained, and it knows where those models are stored on disk, the implementation of such a service would be considered a relatively easy task. Second, additional user interfaces in the web page should be added to better manage the organization of models and the ability to reassign them to new tasks. Currently the models are listed, which will not work well when the number of models becomes large. Adding the ability to store models inside of folders, in a hierarchical structure, could resolve this problem. Given the pipeline can manipulate HDFS (as seen in the TensorflowDNN ModelType) and TensorFlow has full access to HDFS from its HDFS integration, this would be an easier task as well.

**Monitoring Existing Models**

With the deployment of models comes the issue of maintaining them as well. Beyond just organizational enhancements, adding a system to monitor model performance would be advised. This system would determine if a model is performing poorly during routine tests, which if so, would then either retrain the model with more up-to-date data or discard the model altogether – depending on the desired use case. This service could be a continuous Spark job running on the cluster that occasionally checks models in deployment, or perhaps more appropriately, a service similar to our Livy Service which can use the metadata as a means to help track model performance.

**Additions to ModelTypes**

Towards the end of the project, work was begun on Convolution Neural Networks, Recurrent Neural Networks, as well as Residual Neural Networks. Of these, only the Recurrent Neural Network was able to get a full implementation with enough time to test and evaluate. It would be a good idea to consider these models for fraud detection, if they prove to be valuable their full implementation and integration into the pipeline would be a good addition. The code for these models reside in their own ModelType implementations, waiting to be finished. It is also fully possible to implement many other types of models beyond these, the limitations of which would be what is possible with the TensorFlow API.

**Evaluation of Supervised Versus Unsupervised Learning**

While the pipeline is primarily designed with supervised training of models, and some preliminary testing done to find strong performing models, it would be advised to go much deeper into this area. The further training of models with differing hyperparameters and implementations to find higher performant models will be needed before deployment of any models to production. The pipeline in its current state is a tool for this exploration, allowing new models to be brought in and tweaked with relative ease, while monitoring them in TensorBoard can give more insight than a few reported statistics. It may also be worth exploring the
performance of unsupervised learning methods further, albeit this exploration may be more appropriate outside of the core pipeline. The additional infrastructure work needed to support unsupervised training should not be justified until single computer tests show it is worth exploring further. Snippets from throughout the pipeline could be reused to help this initial testing.

**Addition of Graphics Accelerated Computing**

The cluster used in this project was made up of virtual machines. A potential improvement to the cluster as well as the pipeline would be to move from using virtual machines to using actual hardware. Gaining access to actual hardware, especially GPUs could greatly help the speed of training on the cluster. Using GPUs though would take some work, as the specific ModelTypes related to TensorFlow would need to be explored to ensure they can take advantage of GPUs “out of the box” or if they would need to explicitly ask for GPU usage. Further research on the GPU integration of TensorFlow would easily answer these questions.

**Containerization for Deployment**

Finally, the containerizing the Web Service would complete the abstraction we set forth to create with it – allowing the container to be deployed anywhere with the configured environment variables injected in. Most of the heavy work appears to have been done, thankfully, so this last step should be an easy grab.
9. References


Retrieved February 25th, 2018 from https://www.slideshare.net/albahsen/2017-stacking-csdtacorrea.


James, G., Witten, D., Hastie, T., & Tibshirani, R. (2017). An introduction to statistical learning: with applications in R.


10. Appendix A: Installing the Pipeline

10.1 Setting up Hadoop and Ambari

Verify disks are setup right

You will need to have enough room on your VM’s root, home, var, and tmp folders to house HDP data and development data. Having about 100GB for the root folder and 20GB for the other three appeared to have worked well for us. In the event you are given VMs with the disk space you need but not allocated to the folders listed above, here are the commands to fix that:

- Show partitions and their sizes: `lsblk`
- Grow partition to a fixed size: `sudo lvextend –L 21G /dev/VolGroup00/<LV_Drive>`
- Update after growing: `sudo xfs_grow /dev/VolGroup00/<LV_Drive>`

Adding a common user

You will want to have one unified one across the entire cluster for Ambari to operate over. This user must have passwordless sudo privileges and passwordless SSH abilities for Ambari to operate smoothly. For us, we called this user **horton** and set him up by running these commands on each VM:

- Add New User, no password: `sudo useradd horton` & `sudo passwd horton`
- Sudo: `sudo visudo` then insert `horton ALL=ALL NOPASSWD:ALL` after the `%wheel` group declaration. Save and quit with `:wq`

Next from your node you are treating as master, you’ll want to generate a new SSH key without a password while logged in under this user. Once you’ve generated it, copy the PUBLIC rsa key to each of the other hosts using `scp` or `ssh-copy-id`. Either command words.

Download and Install Ambari

You can follow along with the [official documentation](#) from step 4 and onwards. Here is a summary of steps you’ll need to take to setup the cluster on REHL 7.0 servers. Please note if your VMs do not have internet access, you’ll need to follow the command line version of the installation guide (see BitBucket for some details).

1. Login as **horton** on your to-be-master node. We use this since we don’t have root access but **horton** is a sudoer. You’ll use wget to download the Ambari repo into yum’s repo list: `wget -nv http://public-repo-1.hortonworks.com/ambari/centos7/2.x/updates/2.5.2.0/ambari.repo -O /etc/yum.repos.d/ambari.repo`
2. Install Ambari-server: `sudo yum install Ambari-server`
3. Run the Ambari setup with **Ambari-server setup**. This will prompt you how to set things up, which you can stick with the defaults for most of it. The only deviation is when it asks what user to run as DO NOT pick root. You want to enter your user –
horton. Additionally, select “Yes” when asked to “Setup the remote database connection properties”
4. Start Ambari (If there are errors, check BitBucket troubleshooting guide): Ambari-server start

The Cluster Wizard
Now that your Ambari server is installed and running, you’ll want to configure the cluster through the browser interface. This interface should be running at port 8080 on your master node. The default login information for admin is “admin” and “admin”. You’ll want to change this when updating your users.
1. You’re greeted by a setup cluster wizard. Launch the wizard. You’ll want to follow the steps it provides, giving the cluster a name (pick something fun, you’ll be getting to know the cluster very well), and choose a recent version of HDP (we used 2.6.2, but note you’ll want to stay up-to-date for bug fixes!). Ensure the repo you choose to download is the one matching your operating system, redhat7.
2. Once at the host selection screen you must enter the Fully Qualified Domain Name of each VM in your cluster. This can be found by running “hostname –f” on each VM, albeit you should notice a trend after the first few.
   a. You’ll need to provide the PRIVATE ssh key that you generated earlier for Horton.
   b. Make sure you provide the username, horton.
3. The next screen will verify the hosts are all setup right. If you get an error, check the BitBucket Troubleshooting guide – otherwise most errors at this stage have to do with improperly setting up Horton and passwordless Sudo/SSH.
4. Once verified you’ll have to choose services. You can leave everything checked, but we opted to keep things slim by unchecking all services and selecting what we needed: Spark2, HDFS, Hive, and Livy. Ambari will prompt you to select their dependencies when you hit continue.
5. On the next page you’ll have to assign masters. We chose to shift a lot of the servers to our Primary and Secondary NameNodes (Masters) to keep the Master-client nature clear.
6. The following screen assigns what clients to give to what nodes. You should have a few nodes serve as Edge Nodes – these are where you install all the clients so your users can interact with the cluster easily (things like pySpark for Spark-submit, or beeline for running SQL commands through Hive). You’ll also want to assign a server to be your Livy Server – this should probably be one of your NameNodes. The rest should all be DataNodes.

Configuration
Finally, it is time to fix your configurations. There’s a ton of them, and Ambari will help guide you – be sure to address all warnings and red errors. You’ll need to setup passwords and
remove paths that include “var” or “tmp” – as for some reason those are not desired yet they are the default paths. The most important part is to make sure under the MISC tab you turn off UID Management. REHL doesn’t like the cluster assigning User IDs when making all the necessary supporting users.

If you are unsure what to set the memory options to, I highly suggest reading and using this documentation’s utility. Other settings to adjust include under Spark2 setting the CSRF Protection to false, to allow jobs to be sent to Livy without permissions. Further exploration on this may be desired if you need that. You will also want to increase the livy.session.timeout to prevent long running jobs from being killed by Livy. We set this to 5 days (in milliseconds) but it might be advisable to go higher.

10.2 Setting up the Pipeline

Adding Developer / Other Users for Ambari

In order to interact with any part of the cluster, each person needs a user – otherwise all interaction is done through Admin and that could result in some damage. To resolve this start by SSH’ing into any node on the cluster. Run the following to build a folder for each user:

- Switch to the HDFS user: `sudo su hdfs`
- Make the user’s folder: `hdfs dfs –mkdir /user/<username>`
- Set the owner: `hdfs dfs –chown <username:hadoop> /user/<username>`

Next you’ll navigate into the Ambari webserver and enter the “Manage Ambari” screen via the top-right drop down menu. You’ll then navigate to users, and create a local user. Fill out the form and navigate to the Roles tab on the left. Provide each user with the necessary privileges level – we deemed this as “Cluster User” but Ambari can provide more details on the different levels.

You’ll also want to repeat this for the Admin user, so they can access browser utilities such as “Files View”. With this completed, each user can now interact with Hive, HDFS, and Ambari from their own unique user without risk of hurting the cluster.

Preparing the Environment

This section will cover how to prepare each node in the cluster for running the various systems of the pipeline itself. This means preparing the webserver on a single node, and then preparing the executor environments on all nodes (including both masters). The node you choose to run the frontend webserver will need NodeJS and MongoDB installed, otherwise ALL nodes in the cluster should receive the python installation and the corresponding dependencies outlined below. If your nodes do not have an internet connection, see BitBucket guide on “Adding More VMs Later On” which talks about setting this up offline – prepare for a very fun time.
• For the Webserver Node: As per the NodeJS website instructions you will need to use curl to download a LTS version of node. As of writing that was 6.0, but please keep this up to date to gain performance and syntax updates. These commands are:
  1. curl --silent --location https://rpm.nodesource.com/setup_6.x | sudo bash -
  2. yum -y install nodejs

• For the Webserver Node: As per MongoDB’s instructions for Version 3.4, you’ll want to download and install MongoDB server. You can actually install this on any node you want, just know which node so you have the URL to point the Webserver’s ENV at. After installation, start the service with sudo service mongod start. You are done.

• For ALL nodes: You will want to have Python3 with Pip3 installed for TensorFlow support. The default version on REHL is 2.4.5 with Pip2. Sadly, REHL does not include Python3 in the base yum packages, so you’ll need to go through their EPEL repository. Instructions are here, but you can follow this summary on each node:
  1. Wget/Curl the latest EPEL RPM for REHL 7 from here. The exact link is in the fine print on the page, so please find it. It may be an updated link, hence why the indirect hyperlink provided in this document. Technically you could just run yum install epel-release to avoid this step, but that gave us conflicts for unknown reasons. You may opt to try the command before manually installing the RPM file.
  2. Install the downloaded RPM: yum install <downloaded file>
  3. Install python3 and pip3: yum install python34 python34-pip

Once you have prepared the NodeJS environment and all the python environments the final step is to give ALL nodes the python dependencies they need to run the various subsystems of the pipeline. This includes TensorFlow and TensorFlowOnSpark (for obvious reasons), Requests for posting to the Webserver, and Hyperopt for Hyperparameter autotuning. Simply run pip3 install TensorFlow TensorFlowonSpark requests hyperopt to get all of these.

Installing the Pipeline’s Code from BitBucket

Congrats, you are in the final step. In this step we will be taking the cloned pipeline repository and sending its various parts to where they must go on the cluster for the pipeline to function. Be wary of where you start putting these files, as you will need to know where they go and what node they go on for accessing them or setting up the environment config.

For the webserver, you will copy the “frontend” folder from BitBucket to the node you installed NodeJS on. It does not matter where you place this on the node, what’s more important is that you remember what node it is. Since this will be launching a server for you to access, you will need to enter the node’s IP and the port configured in the webserver to access it.

• Clone folder to Node: scp -r frontend <NodeIP>:<DestPathOnNode>
Before setting up anything further with the frontend, we must prepare the rest of the pipeline. The reasoning being is that once we finish the remainder of the pipeline, you’ll return to the frontend folder you just copied to fill in the environment configuration file (.sample-env or .env). This file will ask you about several paths to where you copy the rest of the pipeline!

First, SSH into a node that has Spark-submit abilities (ie an edge node). You will need to obtain the file “hive-site.xml” from /etc/Spark2/conf/hive-site.xml”. Upload this file to HDFS, either using the Ambari file view from your VDI (after SCP’ing the file to your VDI) or via the hdfs command: `hdfs dfs –put hive-site.xml /<path on hdfs>`. It need not matter where you put it, but remember the HDFS path. This is one of the require environment variables.

Next, create a directory on HDFS, again through either Ambari File View or the hdfs command. What’s important about this directory is not where it is, but that it should be owned by the hadoop group and give read/write access to everyone. This is what we call the “Model Store” folder and is where all your models are serialized to disk for long term storage. This is another environment variable the frontend server needs, so remember where you choose to store this on HDFS. To make this directory via HDFS command, run:

- `hdfs dfs –mkdir /<path on hdfs>`
- `hdfs dfs –chown <user>:hadoop /<path on hdfs>`
- `hdfs dfs –chmod u+rwx,g+rwx,a+rwx /<path on hdfs>`

Finally, you need to store all files related to your Spark Job on HDFS, such that Livy can access and submit them. This would be the “model_trainer.py” – our main job, and its supporting packages; scoring, utilities, and ModelType. It doesn’t matter where these are on HDFS, just remember where you put them as each path is a different entry in the environment configuration file for the frontend. After SCP’ing the Model Trainer, utilities, and scoring folders to a node, you’ll run the following:

- Put our Main/“Livy Job” onto HDFS: `hdfs dfs –put model_trainer.py /dest/Path/On/Hdfs/`.
- Next, zip the scoring, utilities, and ModelType packages. For Livy to use a python package not installed on each executor it must be zipped and supplied as a supporting file. Hence run: `zip –r ModelType.zip ModelType` (repeating for utilities and scoring).
- Finally, use the `hdfs dfs –put` command to place the three zip files onto HDFS.

At this point you may return to the node with the frontend folder you cloned earlier. You’ll need to install the dependencies by running `npm install` while inside the frontend folder. Then you will edit the .sample-env file, filling in all the blanks with the file paths and URLs you just setup. Each of the fields are explained inside the sample file, so it should be straightforward. Once done, rename the file to just .env. You can now start the frontend command: `node server.js`
11. Appendix B: Metadata Fields

Below is a table of all the metadata fields employed by the pipeline. These, specifically, are the fields found within the MongoDB documents that define what a model is and how it behaves in the pipeline. For additional information in how to expand upon these, see the relevant section in appendix C.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Found in?</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>All Models</td>
<td>The name of the model, for human legibility</td>
</tr>
<tr>
<td>Author</td>
<td>All Models</td>
<td>The author of the model when it was created in the frontend.</td>
</tr>
<tr>
<td>Algorithm</td>
<td>All Models</td>
<td>This string field is used to determine what ModelType the Model Trainer should use to build model backed by this Metadata object. Some examples include “Random Forest” or “TensorFlow DNN”</td>
</tr>
<tr>
<td>Output</td>
<td>All Models</td>
<td>This is the string name of the column in the selected dataset that is considered the label. For example, this might be the digit’s actual number in the MNIST dataset. Used by Model Trainer when selecting the data from Hive.</td>
</tr>
<tr>
<td>Features</td>
<td>All Models</td>
<td>This is an array of column names selected from the frontend. The Model Trainer uses this list to build its SQL select statement – hence why it is important to specify column names correctly in the features REST endpoint. See future work suggestions.</td>
</tr>
<tr>
<td>Dataset</td>
<td>All Models</td>
<td>This is the name of the Hive Table to take data from, used by the Model Trainer when building its SQL statement.</td>
</tr>
<tr>
<td>TestDataSet</td>
<td>When selected in frontend</td>
<td>When this field is present in the dataset, it signals the Model Trainer to select a separate test dataset to use for evaluation from Hive. We don’t enforce having this table having the same shape as the Training set – we assume the model creator is capable of selecting a compatible set correctly. Adjustments to the features REST endpoint could enforce this. If it’s omitted in the frontend, this field is set to ‘false’ as a flag for the model_trainer.</td>
</tr>
<tr>
<td>SplitPercent</td>
<td>When not using a test dataset</td>
<td>If a separate test data set is not being used, this field is instead supplied by the user to specify how much percentage of their data they want to use for training. The remainder will be used for testing.</td>
</tr>
</tbody>
</table>


<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accuracy, Precision, Recall, f1, fpr</td>
<td>Filled in by the Model Trainer after scoring. These statistics (Accuracy, Precision, Recall, F1-Measure, and FalsePositiveRate) are what is currently recorded and reported. <strong>If you have an alternate evaluation method, then you can add in the value!</strong></td>
</tr>
<tr>
<td>dateFrequencyNum</td>
<td>All Models. This is the retraining frequency number component. Ie if the user said to train every “5 Hours”, this field is the “5”. This is used to compute the next retrainings.</td>
</tr>
<tr>
<td>dateFrequencySpan</td>
<td>All Models. This is the retraining frequency span component. Ie if the user said to train every “5 Hours”, this field is the “Hours”. This is used to compute the next retrainings.</td>
</tr>
<tr>
<td>Deployed</td>
<td>All Models. This is currently unused, albeit referenced by parts of the REST API should it be used. Currently the vision is as a means to signal a model is currently in production inferencing within the pipeline.</td>
</tr>
<tr>
<td>Enabled</td>
<td>All Models. This is currently unused, albeit referenced by parts of the REST API should it be used. Currently the vision is as a means to signal a model should be taken down – perhaps not yet in production by is preparing.</td>
</tr>
<tr>
<td>Retrain_date</td>
<td>All Models. This is the field used to signal when the model should be next trained, filled in by the create endpoint rather than the user. This field is updated each time a retraining occurs using the dateFrequencies to compute the next date.</td>
</tr>
<tr>
<td>Last_trained</td>
<td>All Models. This field is informational, simply tracking when a model was last successfully trained.</td>
</tr>
<tr>
<td>File_location</td>
<td>All Models. This is the location where the model should be stored. For a yet-to-be-trained model, this is simply the base directory where a model should be serialized to. After training, a directory is created here to serialize the model into, at which point this field is updated alongside the model’s score to signal exactly where the model is on disk.</td>
</tr>
<tr>
<td>In Training</td>
<td>All Models. This field is informational. It is “OK” or “Failed” depending on whether or not a model succeeded to train or failed all together.</td>
</tr>
<tr>
<td>tune</td>
<td>When set in frontend. If the user opted to set their hyperparameters via autotuning, then this field is set to the Hyperparameter selection algorithm desired – ie “Bayesian, Random, or Grid”.</td>
</tr>
<tr>
<td>dateNum, dateSpan, dateCol, hourCol</td>
<td>When a model is time series based. If the model being trained is based on time stamps, then these fields hold the information necessary to select data from the Hive Table based on time. HourCol is only provided if the dataset supports Hours.</td>
</tr>
<tr>
<td><strong>recordNum</strong></td>
<td>When a model is non-time series based</td>
</tr>
<tr>
<td><strong>Model Specific Fields</strong></td>
<td>Depends on Model Chosen.</td>
</tr>
</tbody>
</table>
12. Appendix C: Expanding the Pipeline

12.1 Adding a New Data Set to the Pipeline

The pipeline uses Hive tables to store its data for training and testing, meaning if your data is within the cluster’s Hive database then the pipeline has the ability to reach it for training and testing. The only requirement to alert the pipeline of this new table is to edit the features endpoint of the REST API, such that you list the table name in list.js and setup the columns / type of range selection you want in the accompanying get_features_for_list.js endpoint. This is all you have to do if your data is already in Hive, since the Model Trainer builds a SQL statement around the columns and table name provided to it from the Model’s Metadata object, the Model Trainer will handle the rest without need of changing anything else. If this table receives any updates with new rows, then those rows will be available for upon the next training of a model when a new SQL statement is built.

More importantly is ingesting a given dataset into Hive. While there is no formal mechanism for this – given the various shapes and sizes data can come in – we have created a series of helper scripts inside the data_ingestion folder to assist. The important scripts to pay attention to are those beginning with “ingest” and the construct_features.py. The remainder files within this directory are either support scripts for these higher order scripts, or alternatives worth trying if one doesn’t work perfectly.

The Ingest_* scripts are our examples of how to bring in a given dataset to a hive table. The Ingest_transactions script is designed to bring in a fixed width format file given a description of the dataset (a metadata file), the data file itself, and the name of the table you want ingest the data into. This script works by converting the fixed offsets into a CSV, then building a Hive compatible schema from the metadata file (essentially mapping each entry into its corresponding type), and finally uploading the CSV to Hive using native Spark 2.0 methods. The construction of the schema is handled by the schema_generator support script and the parsing of the metadata file is handled by the metadata_parser support script.

The other two ingest_scripts work similarly, with the ingest_delimitedfile serving as a means to bring in a data file that is delimited by some character specified in the arguments. This script doesn’t have to convert to CSV before uploading, but does the same steps as the ingest_transaction script otherwise. The Ingest_mnist is an example of handling an entirely different data format, parsing and converting into a CSV, then bringing into Hive. The takeaway is: once the data is in a delimited format, it can be easily brought into Hive with native methods. There are other calls to Hive that can be made if you’d like to avoid converting to a delimited file, but we did not find a need to explore those options when data could be easily converted. Naturally, once a set has been ingested, we’d expect new data to be inserted directly to the Hive Table rather than going through additional ingestion. An example of running one of these scripts might include:

- Spark-submit ingest_transaction.py <description file> <data file> <tablename>
Ingesting the transactions is not enough, however, as the pipeline expects all data to be numeric in order for training in a Neural Network. While the project evolved, several scripts were created to help handle these problems, as they became known, which is why there are multiple steps to this. As a future effort, it may be worth merging the functionalities of these scripts into one, to prevent the need of making intermediate Hive Tables or running multiple scripts.

First, construct_features will convert an existing data table in Hive to be String Indexed via MLLib Pipeline’s StringIndexer step – meaning all strings in the table will be converted to a numeric type to represent that string. In addition, it will drop columns that do not meet the given threshold of valid data (meaning too much data was missing from too many data points). If your data table already has these properties, then this script is not necessary to run on it. Otherwise a minor side effect of the script is that it will append an “f_” to each column name when saving into the specified table. This was due to an early requirement from the prior MQP group’s implementation of the pipeline that has since been lifted – in your version of these scripts you are welcome to remove this step.

Second, in order to normalize the datasets, normalize_two_datasets.py and normalize_dataset_old.py were used for the newfeatures dataset and for the RS dataset respectively. If you were to normalize a different dataset, the above scripts can be adapted. One thing to note is that before the fitting of a StandardScaler model on the datasets, it is important to exclude categorical features including the target feature of the data. After doing this, it is required that a map operation maps the newly transformed features. The "withStd=True" argument specifies that the features be transformed to unit standard deviation and the "withMean=True" specifies that the features be transformed with mean centering. Finally, the transformed features, are mapped, along with the previously excluded categorical variables, to a new dataframe which can now be saved into HIVE.

Additionally, feature selection methods such as PCA and the important features from random forest classifications were used for experimentation.

PCA, like our StandardScaler model above, requires that we exclude our categorical variables. Similarly, our target variable must also be excluded as it is an unsupervised method. Currently, the pca_new.py and pca_old.py scripts are used for transforming the newfeatures and the RS Set respectively. Both PCA models are constructed in a manner that returns 10 principal components (k=10). Depending on the size of the dataset, the number of rows that you can perform PCA on varies because of computational limitations. For example, the RS Set could be reliably trained on 500 data points. After fitting the PCA model to the training dataset only and then transforming both the training and testing datasets, they can then be written as tables into HIVE after mapping the previous categorical variables.

Random Forest scripts were random_forest_new.py and random_forest_old.py. These scripts can be adapted to new datasets similar to the PCA and normalization scripts. These scripts iteratively train 10 different random forest models collecting the model's feature importances at
each step. Next the `constructFeaturesDict` function is used to extract the feature importances that are stored as SparseVectors into a clean dictionary form to aid further processing. Next, the `generateTopFeatures` function creates a "master dictionary" with all the features and their corresponding weights added across all of the models. From here, the top 20 features are extracted and then selected from both the train and test datasets and stored into HIVE.

With your data now ingested, you are welcome to edit the features endpoint as mentioned at the start of this section. After doing so, the frontend will have been made aware of the table’s existence and users can start selecting it to add into their metadata documents. As some concluding notes on this, if you specify your dataset as time series by providing an hour and/or date column you will need to adhere to one of the two support date formats currently in the pipeline. These are YYYYMMDD and YYMMDD, as specified in the `utilities/date.py` file. If you wish to support other formats, you will need to update this file to handle the cases where it receives a different date format (specifically inside `previousNDays()`).

Additionally, when running any of these ingestion scripts you will be working with Spark2 and Python 3, requiring you to export the following environment variables before execution: `SPARK_MAJOR_VERSION=2` and `PYSPARK_PYTHON=python3`. You will also need to run these scripts with additional memory, given the processing demands they include. This can be done by specifying the memory explicitly, for example:

- `Spark-submit --driver-memory=15G --executor-memory=15G construct_features.py`

### 12.2 Adding a New Metadata Field

The pipeline is metadata driven – understanding this will help you recognize all of its potential and flexibility. At each step the metadata serves as the single source of truth – allowing the pipeline to adjust very specific aspects of it depending on what is stated inside the metadata. This enables specifying where a specific model should be saved, individual parameters of it, associating a score with the model, and specifying individual parameters about the model (such as the features it will use, the data it should draw from, the test set it should evaluate over, and model-specific parameters). If at any point you find something should be more customizable about a model, you can adjust the metadata to have an additional field specifying this customization and then adjusting the pipeline itself to draw from this metadata field. Almost every aspect of the pipeline will see a copy of the metadata, making this a very trivial task.

Before reviewing the steps to add a new metadata field, however, please review the current metadata as explained in Appendix B. Remember, the benefit of using MongoDB is a NoSQL style database – meaning we can add or omit fields from a document at any point without having to update the table shape. Hence, we have included the “Found In” column to help hint when a field might be specified.

**To add a new metadata field, you must:**
1. Update the frontend’s Main.html to allow the user to set the metadata.
2. Update the frontend’s Main.js, such that the request made to REST API to make the metadata object includes the new field.
3. Update the REST API’s create endpoint to include the new field, so it can be accepted and stored into Mongo.
   i. If this is a field that should be changed for the next retraining, be sure to update the retrain endpoint as well.
   ii. If this is a field that should be changed after a model has been trained, update or add a field in the update group of endpoints to set the field.

At this point you now have introduced a new field into the metadata object. If you inspect documents within MongoDB, all newly created models will contain this field to the specification you have defined – so if the field is only set for certain models in the frontend, then it will only exist for certain models in the database. How you now act upon this metadata field is up to you – as stated before, most systems within the pipeline will see a copy of this field. So, for example, if you want to have a ModelType behave differently you would want to adjust the concrete ModelType within the ModelType package.

12.3 Adding a New Model Type

Adding a new model type is very straightforward process – don’t over think it! Included within this section is a template of a blank model type based on TensorFlow, albeit if your ModelType does not use TensorFlow then it may vary (For examples in MLLib, see the Random Forest or Decision Tree Model Types. For examples in Keras, see the Keras branch on BitBucket – noting that Keras models are VERY similar to TensorFlow models).

If you wish to use a whole new library, you will need to make sure all executors have the dependencies installed upon them and you will need to ensure there is a compatibility with receiving data from a Spark RDD – this is what TensorFlowOnSpark does for TensorFlow (in addition to some distributed support). If you wish to use something such as PyTorch you will need a mechanism to receive and operate on a Spark RDD and a way to support the distribution of PyTorch (unless you want a single node model training, which is acceptable). It is a similar story for almost any Python library.

Otherwise, adding a new type of model under Model Type is simple:
1. Update the frontend’s main.html and main.js to support the selection of your new algorithm name. The Algorithm metadata field is deterministic of what ModelType is selected for creation, hence why you want to update the systems using it to add a new ModelType.
2. Update the Create REST endpoint should you have any additional fields with your model type – such as BatchSize or LearningRate. If you don’t have extra fields, you can omit this step.

3. In the Model_Trainer.py you should edit the “model_generator()” method’s switch statement to include your newly made ModelType, thus calling your concrete implementation in the ModelType package.

4. At this point you now implement your model inside the ModelType package. There are no further “General Steps” to take.

If you wish to add a new TensorFlow model, perhaps a new Neural Network type for this example, you’d want to copy the TensorFlow DNN to serve as a template. Everything about the way data is processed, fed into the model, the construction of the model, and the training / testing of the model is controllable. In the following I will walk you through how the code works, so if you implement your own model you will know what to change.

1. The Method called from Model Trainer

The inception point into your ModelType serves as a preprocessing step for your Model’s Training and testing. Anything you do here is not Distributed TensorFlow code. The first few lines of code of the “TensorFlowDNN” method in “TensorFlowDNN” are typically not something you should change, as they setup some needed configurations regarding executor count and what packages that part of the code requires.

The first line of code you should look at in the TensorFlowDNN method is how we preprocess our given RDDs such that they are in the format we want for training:

```
# Consumes a row of the dataframe, converts to tuple format
def toNumpy(record):
    feature = np.array(record.features)
    ary = [0 for i in range(0, model_args['classes'])]
    ary[int(record[label])] = 1
    labels = np.array(ary)
    return (feature, labels)

# Turns DF to RDD, then transforms into TensorflowOnSpark format.
formattedTrainData = trainingData.rdd.map(lambda x: toNumpy(x))
formattedTestData = testData.rdd.map(lambda x: toNumpy(x))
```

Figure 48 Before we start Distributed TensorFlow, we need to format our data such that TensorFlow can work with it - that is NumPy arrays and One-Hot Labels

In this example we’ve taken the RDDs and converted each row into a numpy array followed by a one-hot encoding of its label, returning a tuple of the X value and the Y value. This is critical note #1: We return pairs of Features and its Label because the values returned from here serve as the input to our models later on from the Feed_Dict method we will be reviewing in
a moment. In MOST cases you should not change the output format of this function, since the feeding mechanism relies on this layout. You are, of course, welcome to change the way data is fed into your models and we will talk more on that later. Just note that for most model types, you don’t need to do that.

Next, we fetch the arguments we need to prepare the TensorFlow cluster. This is the epochs we are training over, so that TensorFlowOnSpark knows how many times to feed the training dataset into your model. We also clear out the directory where the model is about to be saved to should it already exist – this is NOT the modelStore folder, but rather this specific model’s SUBFOLDER inside modelStore. Why is this needed? It’s not – unless you are debugging the same model over and over, which at that point, since it’s the same exact model it will be saving to the same exact folder. Think of this as a convenience, not a requirement:

```
# Provide additional parameters not inputted by caller
model_args['mode'] = "train"
model_args['featureSize'] = np.array(trainingData.select('features').first()).shape[1]
tensorboard = False

# Compute enough epochs to run the requested number of steps for the amount of data we have. We will always at least run one epoch.
epochs = math.ceil((model_args['steps']*model_args['batchSize'] / training_size))

# Clear summaries that may already exist for a model (applies if there was an error previously and so we are training the same ID model)
clear_summaries(model_args['save_address'])
```

Figure 49 We set a few last-minute arguments to feed into our Distributed TensorFlow code, as well as specify how many epochs we need TensorFlowOnSpark to feed into our training. The Clear summaries method is simply convenience.

Finally, we start training. This is where TensorFlowOnSpark reserves our Distributed TensorFlow cluster out of our Spark Executors and starts training. When training ends the cluster is shutdown and a new one is built, this time with inference data. Realize the most important part of these lines is the “main_fun” argument – that’s the Distributed TensorFlow code we’ll be running (and reviewing in just a moment):

```
# Run the main function to workers, then runs the training and stopphase
cluster = TCluster.run(spark_context, main_fun, model_args, num_executors, num_ps, tensorboard, TCluster.InputMode.SPARK, log_dir=model_args['save_address'])
cluster.train(formattedTrainData, epochs)
cluster.shutdown()

# Run inference
model_args['mode'] = "inference"
cluster = TCluster.run(spark_context, main_fun, model_args, num_executors, num_ps, False, TCluster.InputMode.SPARK, log_dir=model_args['save_address'])
labelRDD = Cluster.Inference(formattedTestData)
predictions = model(rdd)
evaluate_results = eval_metric(labelRDD, predictions)
evaluate_results = str(evaluate_results)
cluster.shutdown()
return evaluate_results
```

Figure 50 Start the training cluster, then afterwards swap to testing mode, and run the cluster again. Our Distributed TensorFlow Code will adjust as needed. Notice testing returns an RDD with the predictions the cluster made - we can then evaluate this.
Notice we set the “mode” to be “inference” – this is NOT some special argument; it is ours! To take advantage of the same code for training and testing (but to do something different in testing “mode”) we provide this argument to the main_fun. Any arguments in “model_args” gets forwarded to the main_fun, so add in extra flags and such to that object! Otherwise, after inference we run our evaluation using the function defined at the top of this method:

```
def evaluate_ann(label_rdd):
    metrics = MultiClassMetrics(label_rdd)
    return metrics.accuracy, metrics.weightedPrecision, metrics.weightedRecall, metrics.weightedFMeasure(), metrics.weightedFalsePositiveRate, None
```

Figure 51. Our custom evaluation of the labelRDD. This is wholly up to your implementation - we just chose to use Spark’s MultiClassMetrics to compute Accuracy and such.

This is where you can customize how your model is scored. When we talk about the Distributed TensorFlow code, we will talk about how this “LabelRDD” is built. You can build it anyway you want, the key point is that once this RDD is returned, you can then run any kind of evaluation you want on it from here. Here is critical note #2: For us, that is simply computing Accuracy, Precision, Recall and such. For you that might be making a subprocess call to an executable (found in PATH on each executor), in which you have designed a system to feed an RDD into that executable. It might also be some custom algorithm you have implemented in Python. The choice is wholly yours. Whatever you return is what the Model Trainer will receive, at which point you can configure in the Model Trainer how it is sent back to the REST API to update the Model’s Metadata in the database.

2. Distributed TensorFlow and the Main_Fun

Now, what is this Main_Fun and what is it used for? This is how TensorFlowOnSpark works – it does all the heavy lifting in preparing our Spark Executors to work in a Distributed TensorFlow Environment, and then it calls this function where our code is executed as if it is running on a Distributed TensorFlow cluster (which for all intents and purposes it truly is!). This Main_fun has nothing specific to it from TensorFlowOnSpark other than two lines of code used to fetch the Distributed TensorFlow cluster and server instance and to feed data into it. That is Critical Note 3: everything inside your Main_Fun is Distributed (Or Not Distributed) TensorFlow code, not a single line is specific to anything else. If it runs in TensorFlow, it will run here.

The first line of code you should consider is the feed_dict, everything else beforehand is standard or how you choose to preprocess your passed arguments. The feed_dict is how data from our Spark RDD is received by TensorFlow:
Figure 52 Feed_dict is the mechanism to take a batch from the RDD and convert it to a format compatible with training.

TensorFlowOnSpark will take the data we passed to the cluster.train() or cluster.inference() call (hence why we preprocess it into NumPy form) and send it into the feed_dict system – this is a series of “feeds” (batches) which we retrieve with a TensorFlowOnSpark call during training. Each batch is then sent through the feed_dict function to build the structures needed for feeding input and labels. At this point in execution, the data you are receiving is no longer an RDD or Spark related. It is the tuples we converted our data to with the toNumpy function – see why we needed tuples now? We can pull apart our features and labels in the batch to build into the form TensorFlow wants: an array of inputs and an array of labels.

The next few lines are what distinguish the workers from the master in the Distributed TensorFlow cluster. This is also what setups between graph training using the replica_device_setter and sets the cluster instance taken from TensorFlowOnSpark. Again this is not something you’d really want to change unless you’re exploring other types of distributed training within a model. In that event, this is exactly what you are looking for:

Figure 53 The cluster is the first thing established within TensorFlow. Here we setup between-graph distributed training, while preparing to tell workers what to do specifically.

Everything within this ‘with’ statement is what defines our TensorFlow model. This is Critical Note #4: This is where you define your model, how it has structured and built. Everything you place in here is unique to your model, preparing the TensorFlow graph for training and testing. There are many examples out there, including our Keras and TensorFlowDNN example. Use these as guides to help define your model. Notice that the only non-essential parts of what is in here are the Summaries – those are what is logged to TensorBoard. The next line of code in which you should stop tweaking things is the saver mechanism and supervisor preparation for training:
At this point, your model is defined and we prepare how it will be saved. We also will prepare the Supervisor, which watches over training and inference.

The saving mechanism uses the built-in TensorFlow Saver, which we reference in the training step to save checkpoints of the graph to our logdir (which is our HDFS file location passed from all the way back in model_trainer). The Supervisor is what watches over training and inference—notice the only difference is the omission of the init_op since we do not want to initialize our graph all over—we will be using the existing model located in the logdir for inference. The point of the Supervisor is to bring our defined graph into the Distributed Session, so each worker gets the graph in preparation for training.

Finally, we have arrived at the training and testing portion of all this code. This, again, is not something you would change typically unless you want extra steps to be taken or a different output from inference. Here is the code:
Recall the feed_dict function we defined at the very start. The first line within this block is the accompanying TensorFlowOnSpark call mentioned – calling “TFNode.DataFeed()” will prepare the feeding system for train or testing. This feed object is what we then draw from until it is empty, calling “next_batch” with our batch size. We process the batch into the format TensorFlow wants for X/Y in Feed_Dict. Then we give this dictionary into the session.run() call – hence why it’s called the feed dictionary. Session.run() will run the operations we defined for our model, the train_op being our optimizer (such as Adam, Stochastic Gradient Descent, or others). The summary_op is what writes our TensorBoard summaries for visualizing, and the Global Step is what tracks how far in we are to training.

In the event we are inferencing and not training (as in, testing) notice we instead run the label and prediction ops defined as part of our model. This gives us back an array of labels and predictions made on the batch, which we zip together and feed into the output of the Tf_feed. This is the final Critical Note: if you want to change the RDD that comes back as a result of TFCluster.inference() then what you send into tf_feed.batch_results() is your call to change!
At this point, your model is done. You have setup the rest of the pipeline to take your new ModelType, you’ve defined your concrete implementation, and are now ready to deploy. Just as you did when originally deploying the pipeline, you simply zip your new ModelType package and replace the copy on HDFS with this version.
13. Appendix D: Data Flow within the Pipeline

At a high level, the first step in the pipeline occurs when a user is interacting with the web-based user interface. The user interface prompts a user to specify the details of the model that they want to train. This web-based user interface interacts with the REST API that acts as the API for the metadata database and for the model queue that is stored in the filesystem. When a user specifies a model and requests that the model be trained, first the model's metadata is entered into the MongoDB metadata database via the REST API. Second, the model is enqueued into the queue that is stored in the filesystem. The queue contains the model ID of the model to be trained.

The livyservice which directly interfaces with the REST API then grabs the model from the head of the queue. Once the livyservice has the model ID for the next model to be trained, it makes a request to the REST API for other model metadata. When this request is fulfilled by the metadata database, the livyservice sends this information to Livy.

At the next step, Livy has model metadata information that is needed to train the models on the backend. Here, Livy can launch the Spark job that will make use of the backend machine learning model implementations to train and test models. When the Spark job is launched, the Model Trainer is then able to, from the model's metadata, recognize the appropriate model to be trained: decision tree, random forest, TensorFlow DNN, etc. It is also at this step, that the Model Trainer has to gather the data to train over. Based on the model's metadata that it receives from Livy, ModelTrainer is able to recognize the specified dataset, and make a request to HIVE in order to load the corresponding dataset into the training process.
While models like Decision Tree and Random Forest have premade MLLib implementations that can be called directly because the distribution of the training process already occurs, it is not the same for Tensorflow models. In order to distribute Tensorflow models across a Spark cluster, TensorFlowOnSpark acts as an entity that manages the distribution of the training process on the Spark cluster. Additionally, in the TensorFlow implementation, regular checkpoints of the training process are made at specified points into HDFS. This allows Tensorboard to visualize the model's training process.

Once a model has finished training, it then goes through a testing process, either on a portion of the same dataset from which training examples were extracted, or on a separate dataset with an identical schema. When the testing process has finished, the model's performance metrics: accuracy, F1-Measure, precision, recall, and false-positive rate are reported back to the Model Trainer. Finally, at this step, the ModelTrainer submits the performance metrics of the model to the REST API which then synchronizes the model's updated metadata between both the metadata database and the web-based user-interface from which performance metrics are accessible.