Project Number:1234

Flexible Infrastructure Supporting Machine Learning for Anomaly Detection in Big Data

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:

Erin Esco

Alexander Huot

Yihong Zhou

Ziyan Ding

Date: 03 March 2017

Approved By:

Professor Elke Rundensteiner

Assistant Professor Jian (Frank) Zou

Processor Eugene Eberbach

Table Of Contents

Abstract

The ever-changing landscape of both machine learning and the fields to which they apply make traditional model building methods insufficient for creating models that reflect the current state of the world. Automated model generation and deployment is quickly becoming the standard in financial technology. We designed and implemented a pipeline that supports the continuous creation, training, and deployment of models to reduce a six month process to a one hour task. We utilized Spark, Hadoop, and Hive to create a fault tolerant and scalable pipeline as a backend supported by a web application as the interface. The final architecture of our pipeline, the process of its implementation, and the evaluation of the Chronos Pipeline are described.

Acknowledgements

We would like to thank our advisors, Professor Rundensteiner, Professor Zou and Professor Eberbach, and ACI Worldwide for their continued guidance and support throughout the duration of this project. Additionally, we would like to thank our company sponsor, ACI Worldwide, and our mentor Eric Gieseke for the opportunity to work on this project and supporting us through its entirety. We greatly appreciate the time taken by Professor Brown to provide feedback on our pipeline as well as recommending reading to support our development. Finally, we would like to thank those who have provided feedback on our project both at ACI Worldwide and Worcester Polytechnic Institute.

1. Introduction

ACI Worldwide is a company responsible for more than $14 trillion in payments and securities daily. Their software offers the delivery of real-time immediate payment capabilities for their customers (ACI Worldwide, n.d.). Although a small fraction, some of these transactions are fraudulent. It is easy for fraudulent transactions to go unnoticed due to how fast transactions get processed. A large part of the fraud detection system is dictated by millions of hard coded rules, which is computationally expensive. Furthermore, it is extremely rigid, and thus does not adapt to the ever changing behaviors of people who commit fraud. This is problematic since fraud occurs in a variety of ways, such as credit card fraud or check fraud, and each type of fraud is committed in constantly changing ways.

A common approach is to generate machine learning models that are able to identify a transaction as fraudulent. Machine learning models are capable of taking labeled data of both fraudulent and non-fraudulent transactions and using them as input to a variety of algorithms to produce models that can classify future transactions (Mitchell, 1999). While these models are useful, creating just one model takes several months, when this has to be done manually. By the time the model is created, the rules they derived to determine if a transaction is fraudulent or not may no longer be applicable. In order to detect fraud, these models need to be produced faster and updated more often.

This problem is not specific to ACI Worldwide, as companies around the world are implementing machine learning in different ways to solve this problem. Paypal currently uses an 800 node hadoop cluster to experiment with and optimize their neural network for fraud detection. Additionally, Fair, Isaac and Company (Fico) offers a complete software system for companies to use that automates the machine learning and fraud detection process (Fico, 2017). In the race to fully automated systems, some fraud detection companies acknowledge the advantage of human input. Forter distinguishes itself through its combination of machine learning with human creativity by incorporating a human expert's opinion into the process without

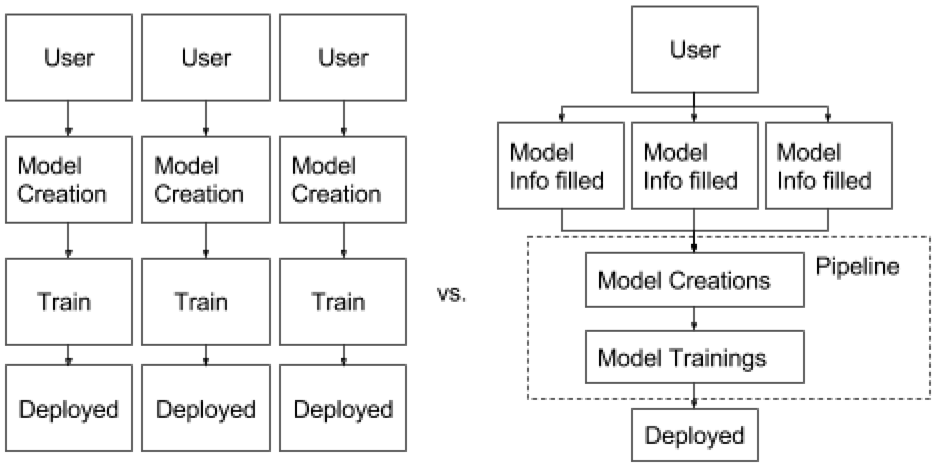


Figure 1.1. Static model deployment (Left) vs. a pipeline for models (Right). In static model deployment, whenever a new model is requested, there is a significant amount of time needed, and just for the one model, in order to get the model ready for deployment and then to deploy. In the pipeline, whenever a user requests a model, the requests are sent to the pipeline and automatically created, trained/retrained and deployed.

delaying the final output (Forter, 2017).

In this project, we also acknowledge the value added by including the ability to specify tuning parameters, among other customizations, without sacrificing the efficiency or speed. Our project explored the possibility of a system that would allow user input regarding the algorithms, tuning parameters, and features that define a model while fully automating (through a pipeline) the process following the initial user interaction. In order to create a system that accomplishes those two goals, we followed an iterative processes that began with ensuring individual components met their minimum requirements and then combed the pieces to produce a simple, viable product. From there, we continuously improved the system by soliciting feedback from our sponsors and advisors and ensuring that we always had a working system. Our solution allows users to define metadata for machine learning models without worrying about the deployment or training process. This automated training and deployment will reduce the current standard of a single monolithic model being carefully designed for six months to a pipeline that is only limited by the resources of the system. The pipeline created (hereinafter, "Chronos")[1] combines state of the art machine learning technologies and a component based architecture to produce a prototype that is modular, flexible, and scalable.

The pipeline boasts a user interface that serves as the interaction point between the user and the system. Here, users can create, view, and modify their models and the state of the system. Following the submission of a model to be created, the model metadata is stored through a Representation State Transfer Application Program Interface (REST API) that handles the communication between different interfaces (Rouse, Hannan, & Wilson, 2016), which is also the channel through which the Model Generator accesses the model metadata. The Model Generator uses the Spark ML Pipeline library and calculated features from a feature storage table in Hive[2], which is a framework for accessing data stored on a HDFS through SQL-like queries (Apache, 2017b). These capabilities make it an ideal candidate for our pipeline to store the data used to train and test the models before storing the models back into the model metadata storage. From here, the Scoring Engine can access the models ready for deployment and use them to produce an output indicating if a transaction is fraudulent, which is stored back to a Hive table. If desired, these results can be sent out of Hive in order for the next steps, such as inspection by fraud analysts. For more information about Hive, see Section 2.2.3.

The Chronos Pipeline is a sufficient proof of concept for a pipeline that combines user input and automated machine learning. Our testing indicates that it is a efficient and fault tolerant solution. The modularity of the pipeline makes it easy to plug in and out technologies to meet the needs of those who will be building models using this pipeline. This adds business value by both eliminating the need and financial cost of manually creating models as well as saving the

[1] Named after the Greek personification of time, as this pipeline is meant to greatly reduce the time taken for training models

[2] For more information about Hive, see Section 2.2.3.

company money by being current with trends in fraud through continuous and automated retraining and deployment of models.

This paper is organized as follows: Section 2 Background contextualizes the current state of fraud and gives insight on some of the technical background and related algorithms used in this project; Section 3 System Requirements explains the requirements outlined for this project; Section 4 describes the overall architecture, individual components and relationship between the Chronos pipeline elements; Section 5 details the infrastructure of Chronos and the chronological implementations of each pipeline component; Section 6 evaluates the performance of the overall and individual parts of the Chronos pipeline; Section 7 presents our conclusions and suggests possible future work in this space.

2. Background

2.1 Context

2.1.1 Fraud

2.1.1.1 State of Fraud

Public concern for fraud has risen dramatically over the past few years as a result of data breaches plaguing news headline. The 2015 Identity Fraud Study (Javelin Strategy, 2015) revealed that people who committed identity fraud bilked $16 billion from 12.7 million U.S. consumers during the previous year. Card fraud is one of the most prevalent malicious technological schemes. In 2014, card fraud accounted for a worldwide loss amounting to $16.31 billion on a total card sales volume of $28.844 trillion (PYMNTS, 2015). The United States alone hosts 47 percent of the worlds' card fraud and 31.8 million U.S. consumers had their credit card breached in 2014 (Holmes, 2016). Fraud continues to be an issue as technological advancements are always one step behind those who commit crimes.

The standard for transactions has recently migrated to cards that uses computer chips to authenticate chip-card transactions. Companies such as Europay, Mastercard, and Visa (EMV) support these technologies and have become the global standards as these chips decrease counterfeit card frauds. The chip-based system uses advanced cryptography to generate a unique code for each transaction. Authorization performed by dynamic data provides better security of account information than by the traditional static data stored in magnetic stripe cards. However, it isn't enough to counter the growing trend in electronic payments fraud (transactions where the card is not physically present) which accounted for 45 percent of credit card fraud in 2014 (Holmes, 2016) .

2.1.1.2 General Problems

Organizations lose 5 percent on average of their annual revenues to fraud, driving companies to enhance their anti-fraud techniques (Warin, 2013). One widely used fraud detection method is having business analysts set up specific criteria to flag transactions. Examples of those hard coded rules include multiple purchases in rapid succession, seemingly random large purchases, and online charges. All of these behaviors are likely to flag a transaction as fraudulent (Warnick, 2016). Moreover, companies monitor customers spending behavior to establish individual profiles. Once the customer deviates from their typical behavior, the company is notified to further investigate the situation. Some companies give the power to customers to set up their own preference or alert notification such as "sending me a text message if any purchase exceed $1000". Complaints and reporting from individuals also trigger the investigation of a potentially fraudulent transaction. Companies track fraud occurrences geographically to identify fraud "hot spots" and notify surrounding retailers and customers. (Palmer, 2013)

Data analysis is the norm in the current state of fraud detection, including clustering for modeling behaviors or transactions and classification for classifying people or transactions as fraudulent. (Zitter, 2016) Traditional data analysis has grown into big-data analytics, where engineers use cloud computing and machine learning to detect anomalies. (Zitter, 2016) These efforts have decreased the prevalence of successful fraud and more techniques are being developed to further the field of fraud detection.

2.1.2 Machine Learning for Fraud and Current Applications

Machine learning can extract meaningful insight from data by creating algorithms and models that reflect trends and behavior. It has been widely applied to the financial, healthcare, marketing and transportation industries as well as the government. In the financial sector, machine learning provides the advantages of investigating time series data or going through customer profiles and constructing models that are able to be retrained to fit user needs. Other interested stakeholders are banks, card companies and payment organizations who are equally invested in mitigating the prevalence of fraud.

More than 100 papers about using machine learning on fraud detection have been published worldwide in major conferences and major science magazines over last 20 years before 2013 (Mizes, 2013). Academic studies initiated the quantitative models and frameworks applied to multi-features transaction data and thus, assisted in pioneering the development of anti-fraud system in payment companies with data mining techniques and artificial intelligence supports.

2.1.2.1 Managing Risk through Simple Neural Networks

Paypal, the world's largest online payment company, is making its best effort to fight against online fraud. Rather than treating fraud as a legal problem like banks and government, Paypal views it as risk management problem. The company is currently adapting deep learning on its accumulating transaction data. Building upon a simple neural network, one of the most popular and easy to use machine learning models, Paypal extends it to a complex multi-layer model. Based on the experiment video the company published on 2014, the company tested a model that utilized R, H20 and a distributed file system on Hadoop with 2,4,6 and 8 layers and 1500 features. This model was able to achieve above 80% accuracy (Ramanathan, 2014). In addition to developing models, Paypal allows users to set advanced fraud management filters by themselves. The filter can monitor the live transactions to protect customers' capital.

2.1.2.2 Combining Models

Visa, an institute focusing on payment productions, is also developing its fraud detection system using machine learning models. The system is called Visa Advanced Authorization (VAA). According to the assertion of Rajat Taneja, the Executive Vice President of Technology and Operation in Visa, the VAA combined both a neural network and gradient boosting algorithm for a decision tree to make prediction on fraud. With more than 500 attributes to be considered for each transaction, the deployed model calculates the transaction score (which translates to the transaction's likelihood of being fraudulent) within less than one millisecond. Monitoring the real-time data, the system runs 56,000 transactions per second (Taneja, 2015).

2.1.2.3 Fraud Detection Service

Another example is FICO, providing analytical service for half of top 100 banks worldwide and famous for FICO® Score, a standard measurement of consumer credit risk. (Fico, 2017) It provides a fraud detection package called FICO® Falcon® Fraud Manager running on FICO® Falcon® Platform. Falcon® also runs on neural network based system. However, it not only runs real-time transaction data, but also can consider customer profiles so that models learn the customer's behaviors and their spending patterns, and thus, can become more accurate in detection fraudulent transactions.

2.2 Technical Background

2.2.1 Hadoop

Apache Hadoop is a software framework for both distributed storage and processing large datasets (Apache, 2017e). Hadoop is capable of storage through Hadoop Distributed File System (HDFS) and processing through MapReduce. On the top of Hadoop, YARN is the resource manager, controlling data storage on HDFS and application scheduling (Apache, 2017f). It can combine with additional software packages alongside Hadoop to build a developing ecosystem, such as Apache Spark, Apache Hive, Apache Hbase and Apache ZooKeeper(Bandugula, 2015； Apache, 2017b； Apache, 2017b； Apache, 2017d； Apache, 2017g).

All processes are orchestrated by the Namenode, which controls all Datanodes. Datanodes are responsible for storing data to blocks and duplicating data to prevent disk data failure. Advantages of HDFS include the robustness of cluster rebalancing and data integrity. It allows large data sets to be distributed over several machines using Namenodes to track the data trunks split between the Data Nodes on multiple servers.

For the MapReduce engine, JobTracker and TaskTracker are the core for monitoring cluster nodes. JobTracker farms out MapReduce tasks to specific nodes in the cluster and locates

TaskTracker nodes with available slots. (Apache, 2010) The TaskTracker spawns a separate java virtual machine process to do the actual computation work. (Apache, 2009)

2.2.2 Spark

Apache Spark is a data processing platform that is used for analytics on large data sets. (Bandugula, 2015) Apache Spark must be coupled with a way to manage clusters and distributed files. Examples of technologies for the former requirement include Spark's standalone cluster mode, Apache YARN, and Apache Mesos. These allow for a driver program to send tasks to worker nodes to handle tasks in parallel. Furthermore, distributing the data across nodes allows it to be processed in parallel. Resilient Distributed Dataset (RDD) is the core doing this job. It helps immutable collections of objects spread across a cluster. A few technologies that combine with RDD to handle distributed data are: Hive, Cassandra and Hbase. These aforementioned technologies work as data storage, but they can also assist Spark in processing distributed data in parallel quickly by building RDDs, avoiding putting data in disks and keeping them in memory. A notable downside of Spark is that by keeping data in memory, it runs the risk of losing any progress or data if it were shut off while performing computations.

Apache Spark is one of many cluster computing platforms alongside Apache Storm, Hadoop, and Apex. Apache Spark is different from Apache Storm in the way that it processes data. Apache Storm allows for streaming one event at a time (Apache, 2015a). Conversely, Apache Spark gets close to streaming by sending micro batches of events continuously to simulate a queue (Apache, 2015b). Apache Storm's streamed data is stateless and will process at least once. Conversely, Apache Spark processes events exactly once (Apache, 2015a). Additionally, Hadoop's MapReduce differs in how it is performed, but the notable distinction is that Hadoop's MapReduce uses more read and write operations which greatly impacts the time it takes to execute a given event (Bharadwaj, 2016). Finally, Apex combines stream and batch processing to support similar messaging and file systems, but graduated from the Apache Software Foundation on April 20th, 2016, so it hasn't been implemented in many large scale companies (Apache, 2017a; Apache, 2016).

2.2.3 Hive

Apache Hive is a framework for accessing data stored on a HDFS through SQL-like queries (Apache, 2017b). It allows data to be stored with more structure, mimicking a relational database by storing data in smaller components which combine to form larger components (Apache, 2017c). From smallest to largest granularity, the components are buckets, partitions and tables (Apache, 2017c). Each table is serialized and then stored in a directory in the file system and capitalizes on Hadoop's in-memory caching, offering sub-second query times as Hadoop's only SQL interface (Apache, 2017c). Its community and ongoing development make it the standard for storing relational data in Hadoop.

2.2.4 MongoDB

MongoDB stores JSON-like objects as records in its database. MongoDB is horizontally scalable through its use of sharding which is sharing data across machine clusters (MongoDB, 2017a). MongoDB uses Create, Read, Update and Delete (CRUD) functions for accessing and altering data (MongoDB, 2017b). Additionally, it is able to support schemas to ensure all relevant information about a document is present. A common use case for MongoDB is metadata management, which is relevant to storing information about machine learning models (MongoDB, 2017c). There is a MongoDB to Apache Spark connector which is able to locate RDDs to improve speed and performance.

2.2.5 Machine Learning Libraries

The MLlib is a package for a RDD based API. The algorithms supported by this package allow for consuming vectors, matrices, and distributed matrices (Apache, n.d.). There are a number of advantages to using this package. One of these advantages is that the package is user friendly. A new user could easily learn through the examples provided by the package how to run the various algorithms. This package is also scalable, allowing for the potential to grow the system. The main issue is that there are fewer classification algorithms capable of handling multi-class classification methods than other available packages, which are vital for fraud detection. The other issue is that this package is soon to be deprecated. At some point, improvements to a system using this package will be restricted simply due to the fact that Spark will no longer support it.

The ML is a package for a DataFrame based API (Apache, n.d.). The package is built on top of DataFrames which are Datasets organized by named columns, similar to a table in a relational database (Apache, n.d.). There are some interesting new uses supported in this package. For example, this package allows for sequentially combining machine learning algorithms. This means that one algorithm feeds its output directly into another algorithm's input, which has the potential to be redesigned to change transaction scoring. This package also provides a more user-friendly API than the MLlib package because of the use of DataFrames over RDDs. Due to these features as well as the fact that the MLlib package is soon to be depreciated, the ML package was chosen for this project.

2.3 Machine Learning Algorithms

2.3.1 Artificial Neural Network

Artificial Neural Network (ANN) is an algorithm inspired by the brain nervous system (Burger, n.d.). The network contains an input layer consisting of input nodes, hidden layer(s) for processing data from the input nodes, and an output layer for computing the final result (Burger, n.d.). When numerical information comes through a node in the input layer, there is an activation number associated with that node and undergoes a transition function which, combined with the relative strengths of the nodes, determines the output number from the node and the input to the next node. This explanation is linear, but in fact many nodes are combining to produce input values for one node and so on (Sayad, 2017).This method is good to use when the data set at hand is diverse and when trying to identify common regularities, or similarities between data points in the dataset. ANNs are also good at identifying and representing complex relationships where the relationships between data might be vague or difficult to understand (Burger, n.d.). The drawbacks of this algorithm include its complexity, the "black box" aspect of the hidden layer, and its tendency to overfit the model to the training data. ANNs are a popular choice for anomaly detection as they have been shown to have a low false positive rate, but a higher number of missed true positives, which is appropriate for some applications (Pradhan, 2012) . Artificial Neural Networks have become a popular choice in the fraud detection space (Patidar, 2011).

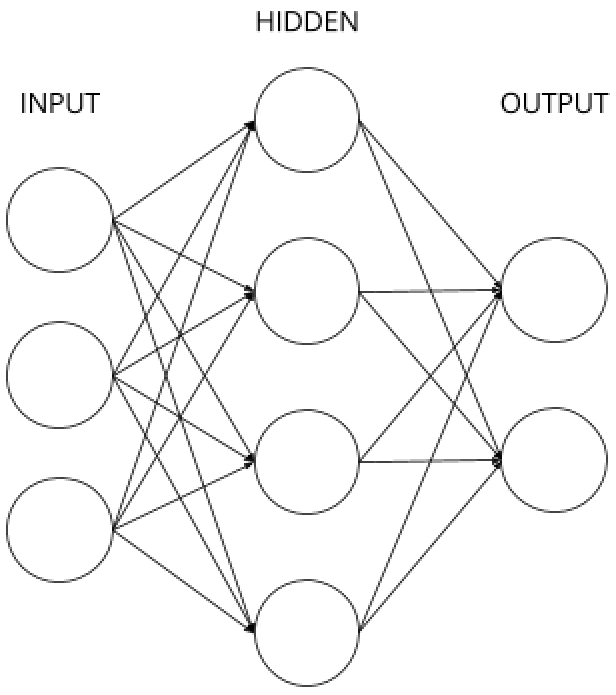


Figure 2.1. Diagram of Artificial Neural Network

2.3.2 Decisions Tree

Decision trees are a method of classification which generates sequences of choices to categorize data where each branch is a decision point learned from the training data and each leaf node is the classification of the data (Introduction to Data Mining, 2015). Decision trees aim to provide a process of how the classifications are made, where, starting from the top of the tree, each node represents a step in classifying the data, and the leaf nodes are the class that this data belongs to. Decision trees are simple to understand and interpret (Apache, n.d.). They are useful for determining the worst, best and expected values for different scenarios and can even be combined with other decision techniques.

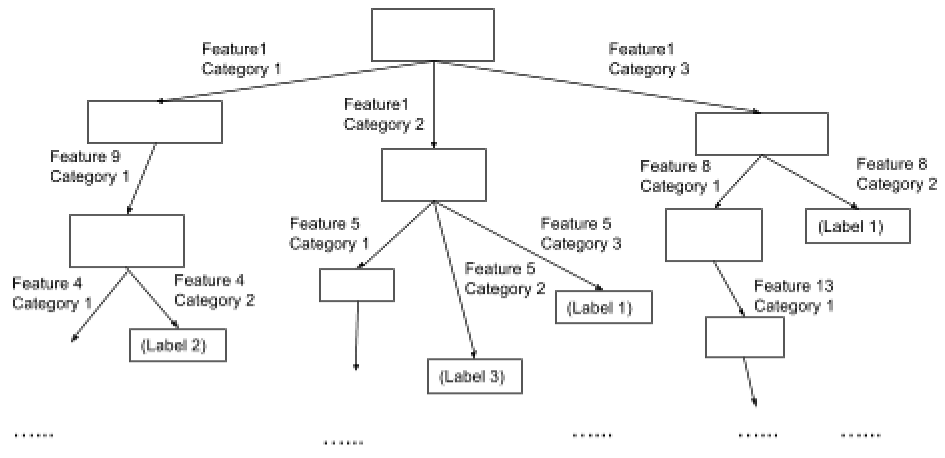


Figure 2.2. Diagram of Decision Tree example. Classification on attributes or features aggregates the label group.

One drawback is that this method tends to be biased in favor of attributes that are better defined in the tree, meaning that there are more nodes in the branch of the tree used to define these attributes. Complex calculations can also occur if many values are uncertain or if many outcomes are linked together. Such calculation is not ideal because the decision tree will not accurately reflect the actual distribution of records.

2.3.3 Random Forest

Random Forest is an ensemble of decision trees. This method combines many decision trees together in order to reduce the risk of overfitting (Apache, n.d.). The decision trees are created by training each tree separately and providing some randomness to the training process by choosing only subsets of the training set data (Introduction to Data Mining, 2015). The output of a random forest model is a combination of the predictions of each decision tree. The decision trees used in this algorithm are created based on a random sample of the training set. The predictions are created by taking a majority vote of the predictions made by each decision tree. This method is a great way to use decision trees while also avoiding issues of

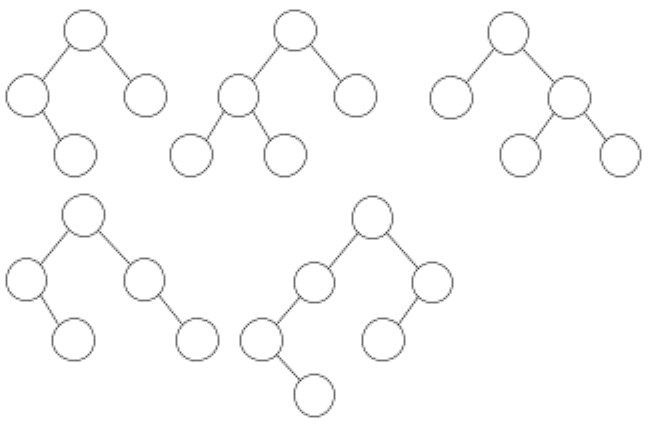


Figure 2.3. Diagram of Random Forest. Each tree has its own set of instructions for determining the result.

overfitting. The performance of this method improves monotonically with the number of decision trees used and runs efficiently for large datasets. In comparison to the decision tree method mentioned above, the same functionality is provided, but random forest runs faster because each tree is trained in parallel and on a subset of the training data. This method also includes the same drawbacks as decision trees, and can potentially overfit with noisy classification tasks. The Random Forest algorithm performs very well compared to many other classification algorithms, including the ANN algorithm.

3. System Requirements

We were tasked with building a pipeline that could generate[3], train[4] and deploy[5] machine learning models. These models would be automatically retrained to ensure that the models responsible for scoring incoming data reflect the current state of the world. This pipeline needed to have a user interface as the person generating models should be able to use the pipeline without a technical background. The pipeline was required to support multiple machine learning algorithms and models and use technologies that are scalable. Furthermore, it was imperative that the pipeline was comprised of microservices[6] to allow for a modular architecture to be flexible for future development.

The pipeline is available for customizing models and data. The user can add new machine learning algorithms to the pipeline with changeable tuning parameters. Data set also can be adjust with more or less features and attributes. Although these changes are adjusted behind the scene, they strongly impact model performance. Our pipeline needed to support a variety of different machine learning models, we identified the three best algorithms as: Decision Tree, Random Forest, Artificial Neural Network. The work supporting these choices can be found in Appendix B.

The iterative nature of our pipeline meant that requirements arose throughout development as opposed to being gathered prior to beginning the implementation. Given these tasks, we identified major requirements that needed to be upheld. The overarching purpose of the pipeline is to allow users to create models with parameters they define, so we knew that model flexibility was a requirement to be considered throughout the project (Requirement 1: Model Flexibility). When users create models, the process should be intuitive in order to avoid the user wasting their time or losing trust in the system (Requirement 2: Intuitive User Interaction). Additionally, the pipeline needed to be scalable to accommodate the incredibly frequent incoming transactions to ACI worldwide (Requirement 3: Scalable). Given this influx of data combined with the variability of user and system behavior and resources, the pipeline should not be easily susceptible to breaking (Requirement 4: Fault Tolerant). This pipeline serves as a prototype and the future implementation may need to use different technologies for a variety of reasons, so the pipeline needs to decouple technologies as much as possible to reduce the amount of time it would take to swap any component out (Requirement 5: Modular). If Requirement 5 is met, it will be easy to exchange technologies, but our pipeline should still utilize technologies that are most appropriate for the task at hand (Requirement 6: Appropriate Technologies).

[3] Create the model data using a set of input parameters (see Section 5).
[4] Create the model by running labeled data (data with the desired results already determined) through a Spark function
[5] To make the trained model ready for use in the Scoring Engine (see Section 5).
[6] Components of the pipeline that, while are not vital to the overall pipeline, allow for a simpler method of modifying any part of the pipeline. An example would be the REST API (see Section 5).

Finally, the pipeline supports building and deploying models from the beginning to the end. So Chronos needs to be flexible in terms of being able to support many models and it should produce accurate models (Requirement 7: Machine Learning). Table 3.1 is a summary table of these main requirements we identified:

Number	Label	Description
1	Model Attribute Flexibility	Models are flexible in their tuning parameters and defining characteristics
2	Intuitive User Interaction	It is clear to the user how to create and interact with models
3	Scalable	The pipeline is able to scale without extraneous effort or resources
4	Fault Tolerance	The pipeline is tolerant against invalid submissions and can handle errors in way that doesn't cause the system to go down
5	Modular	Pipeline is comprised of components that can be easily interchangeable
6	Appropriate Technologies	Despite ensuring each component is able to be easily swapped out, technologies should be the best fit for the job
7	Machine Learning Model Building Ability	Pipeline is able to support a variety of models and produce accurate output

Table 3.1. Pipeline requirements overview

In order to achieve these high level requirements in Table 3.1, we considered each component and broke down what the requirements were for that component to ensure all the requirements were satisfied. In Table 3.2, features of the pipeline will be labeled as an indication of where these requirements are met by the pipeline and specifically which of the requirements were met in that section. This table refers to the requirements in the context of whoever is interfacing with the pipeline as defined through gathering requirements. The "user" encompasses the person generating the models: a data scientist, a fraud analyst, or any other individual with access to the pipeline.

Component	Requirement
User Interface	● Allowed user to complete the following tasks (Requirement 2)

	○ Creating a model ○ Viewing all models ○ Viewing the queue of models ● Interface design makes the process of creating models intuitive (Requirement 2) ● No errors are introduced to the system by the user interface (Requirement 4) ● Interact with REST API
Model Metadata Storage	● Scalable (Requirement 3) ● Able to hold all information needed for a model (Requirement 6) ● Allow REST API to read and write to
REST API	● Handle CRUD operations for the connection to Model Metadata Storage (Requirement 6) ● Implement a management system for model training order and distinguishing others (Requirements 5, 6)
Model Generator	● Machine learning model generated from user input (Requirement 7) ● Handle potential failure from training and allow multiple (but limited) attempts (Requirement 4) ● Interact with REST API to get and update new models ● Interact with Data Storage in order to get transaction data for training
Data Storage	● Support for all rows and column types in the attribute and feature tables (Requirement 6) ● Preparation of data and tables (Requirement 6) ● Support write operations for data that needs to be added
Scoring Engine	● Allow scoring using multiple models (Requirements 3, 6) ● Allow for models to use a different set of features (Requirement 1) ● Score transactions and classify based on the probability that they are fraud (Requirement 7) ● Interact with REST API to get deployed models ● Interact with Data Storage in order to get transaction data for scoring

Table 3.2. Requirements of the pipeline per component

These requirements were considered and we continuously altered our pipeline to ensure they were met. The following sections expand on the requirement elicitation process of these requirements from either our sponsors or identified through technological need. The system requirements gathering process was tightly coupled with our implementation, so the iterative nature of our pipeline came naturally.

4. Overview of Chronos Architecture

4.1 Architecture Overview

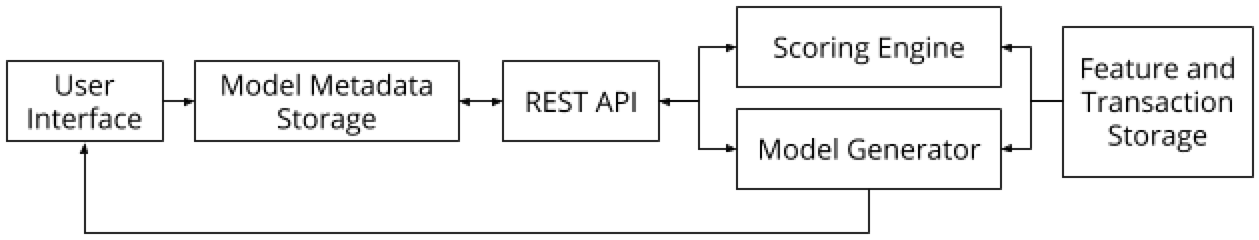


Figure 4.1. Final pipeline high level architecture overview.

The Chronos pipeline gives users the ability to define metadata surrounding models, train them, and deploy them. Additionally, this pipeline supports automated retraining to ensure that models are up to date. The user interacts with a user interface to create and view models whose metadata is stored in a database. The Model Generator takes the model metadata and features as input to create machine learning models where the parameters are defined by the user and generated using current data. Once a model is created, it can be deployed to the Scoring Engine which takes new transactions as input to produce a score for the transactions based on the desired output.

4.2 User Interface

The User Interface serves as the visual representation and interaction point between the user and the pipeline. It gives the user the ability to create models where they define the algorithm, output, and tuning parameters. Additionally, users can view the models they have created at a high level and also at a more detailed view which allows them to see the specifics of the model. Users can search for a model using any of the parameters displayed in the table as the search term. Finally, users can see the queue of models ready to be trained which allows for the state of the system and its models to be transparent to the user. The interface is flexible and offers a minimal design to ensure creating models is intuitive and there are no unnecessary distractions.

4.3 Model Metadata Storage

Our model metadata storage is a MongoDB database where each model record is stored as a JSON-like object. Each model has the same schema allowing both the User Interface to help define models so that other components of the pipeline can rely on these consistent attributes. Notably, our system is horizontally scalable due to the nature of MongoDB.

4.4 REST API

The REST API's main responsibility is to serve as a central connection to MongoDB for the User Interface (UI), Model Generator, and Scoring Engine. It supports the User Interface by providing functions that interface with the model metadata storage allowing the tasks of creating, viewing, updating, and deleting the models. For the Model Generator, the REST API has the ability to retrieve the next model for the Model Generator to train. This is performed by taking the first model from a priority queue which sorts the models based on their next train or retrain date. Finally, for the Scoring Engine, this API can return a set of models to be used for scoring by the Scoring Engine. Having this central coordinator is imperative in ensuring the system remains modular and component-based.

4.5 Model Generator

The Model Generator plays an essential role by producing the machine learning models. It gets the model information necessary to create the model from the REST API and uses it to construct a machine learning model trained on the data from Hive. The performance of constructed models after fitting data will be evaluated and users can see the evaluation results on User Interface. The retrain process is the same as first-time train and a new model with the same name but different id will be produced. Model Generator also allows models that fail to deploy during the training process to try multiple times and thus guarantees the resilience of broken or failed model manufacturing.

4.6 Feature and Transaction Storage

The Hive Database holds the data necessary to support the Model Generator and Scoring Engine. It provides both raw transaction data and calculated features for the system and maintains all newly created tables while the pipeline is running. An initial set of data and tables are already stored in Hive before the Model Generator and Scoring Engine start. This initial data is preprocessed so that the data can be ingested as features, but this is hidden from the users.

4.7 Scoring Engine

The Scoring Engine handles the scoring of incoming transactions using a set of deployed models. Using the provided models, a set of predictions for the transactions are created. These predictions are then combined by multiplying the predicted value with the corresponding model's normalized accuracy, and then added together. This total is used as a probability of fraud, in order to determine whether or not a transaction is fraud. Once the result has been determined, the Scoring Engine stores the results in Hive.

5. Methodology and Implementation

The initial stages of designing the pipeline were done individually, where each component was developed independently to perform the functions needed to have a minimum viable product. This meant that the Chronos pipeline simply needed to train and deploy a single model. Once each component was functional, we combined them to produce a pipeline that could support a single model through the workflow. All components were initially inflexible and had few features. We iterated on our initial design to produce a more flexible and scalable pipeline. The User Interface was developed through a continuous process of gathering requirements, producing a new version and soliciting feedback. The Model Generator and Scoring Engine grew more flexible to meet the needs (and future needs) of the employee building models. New components arose as tasks became more complex to decouple technologies. Once each component's, and thus the pipeline's, requirements were met the team shifted focus to testing.

5.1 Chronos Infrastructure

In initial discussions with our sponsor, we acknowledged to main tasks that pertained to the models: generating and deploying. The first design delegated these tasks to two separate pipelines as shown below:

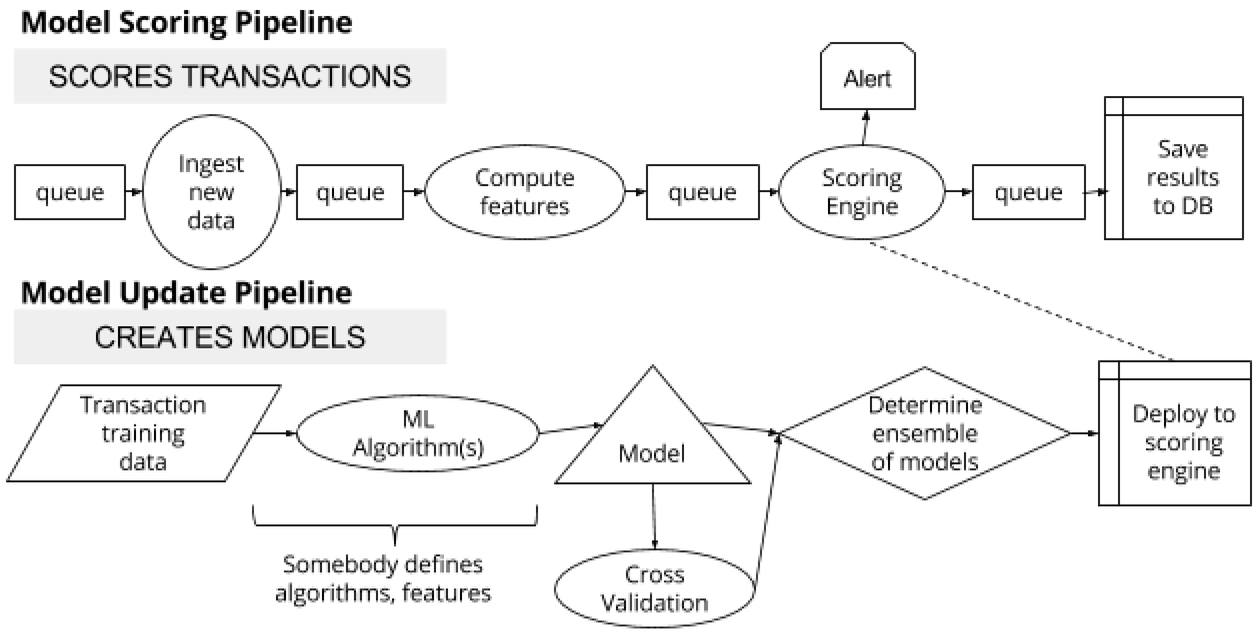


Figure 5.1. First pipeline design

Upon further investigation of the functional differences of these two and at the beginning of implementing this design we noticed that they aren't in parallel, but one after the other. It made sense to combine these into a single pipeline where the first half handles the training and generation of the models and upon completion sends it to the scoring engine where it is deployed to handle scoring incoming transactions. In order to ensure that the process of model generation was shared equally among models waiting to be trained, we created a scheduling system that, by looking at a priority queue, considers the retrain date of the model when deciding which model should be trained next. This queue manager sits alongside the REST API as a javascript file. We abstracted out many of the pipeline's connections by developing a REST API that served as a service to the Scoring Engine, User Interface, and Model Generator. Our final pipeline with the aforementioned improvements is shown below.

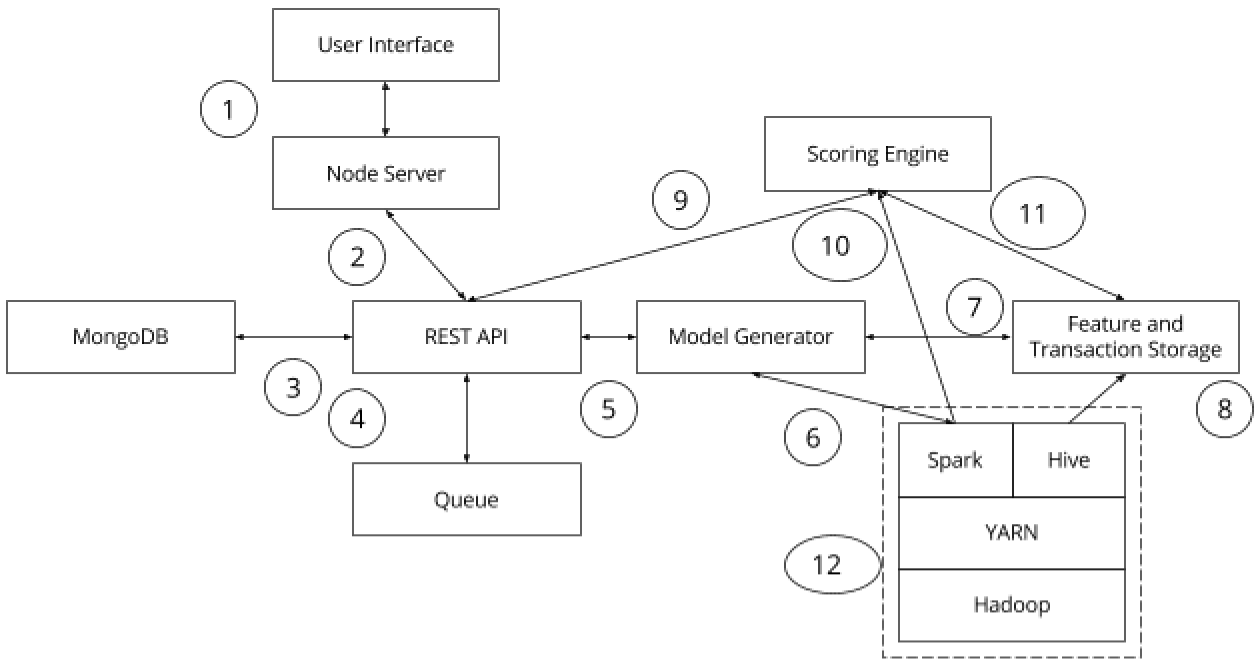


Figure 5.2. Pipeline diagram with connections labeled.

Each of these components have relationships with one or many other components. Each relationship is labeled and explained below.

1. User Interface and Node Server

When the node server is running and a call is made to view the main page, it serves up the relevant HTML, CSS, and JavaScript files. When the user interacts with buttons or forms on the user interface, the client-side javascript makes a request to the server side javascript on the node server which makes the appropriate API call.

2. Node Server and REST API

The node server makes calls to the REST API by making calls dependent on the task at hand. Below is a list of the action done by the user and the resulting REST API call.

Action	REST Call Function Prototype
Create model	`/createmodel?key=value&key=value&...` `(see REST API Section for full list of required key-value pairs)`
Update Model (User side)	`/updatemodelInfo?name=value&deployed=value&enabled=value&nname=value`
Delete Model	`/deletemodel?name=value`
Retrieve models	`/getmodel`

Table 5.1. Pipeline diagram with connections labeled.

3. REST API and MongoDB

Using JavaScript, the REST API forms a connection to MongoDB and then performs the MongoDB function calls needed. It does this through the Node.JS MonogoDB driver which abstracts the two technologies interfacing.

4. REST API and Queue

The REST API calls the queue through functions for dequeue, enqueue, and looking at the first element in the queue. When the current time is at or past the next model's train time, it is given as input to the model generator to be trained.

5. REST API and Model Generator

The Model Generator imports the module from rest.py. The rest.py will make the proper request to the REST API and return the response to Model Generator.

6. Model Generator and Spark

The pre-built Spark 2.0.2 package is required. The Model Generator imports the PySpark package to access the Python API. The Spark application runs using bin/spark-submit script in the package. It will load the libraries and submit the application to the cluster.

7. Model Generator and Database

The Model Generator gets data from the Features table in the Hive database. Based on the features requested by the user interface, the new model selects corresponding features from the AllFeaturesTable table which will be used as input during training.

8. Feature and Transaction Storage and Hive

The database is constructed by Hive, which runs on yarn and HDFS. Real data is stored in HDFS and can be accessed in table form through Hive. Table schemas and metadata of Hive database are kept on the servers' physical directories.

9. REST API and Scoring Engine

The Scoring Engine calls a function in the imported module, rest.py, which will make the appropriate request to the REST API. See section 5.4.2 for all of the function calls, their parameters, and what they return.

10. Scoring Engine and Model Generator

The scoring engine requires a few spark libraries in order to run. The imported libraries include all of the libraries for each machine learning algorithm used as well as the library for SparkContext, MLUtils, and Vectors.

11. Scoring Engine and Feature and Transaction Storage

The Scoring Engine creates a SparkContext which connects the Scoring Engine to Hive. The data to be scored is retrieved as a DataFrame and then mapped to a RDD. When the Scoring Engine is done scoring these transactions, a DataFrame containing the results is sent back to the Hive database.

12. Chronos Backend Infrastructure: Spark, Hive, Hadoop/Yarn

Spark and Hive run on an existing hadoop cluster. It connects to YARN for resource management and writes to HDFS by changing the Spark environment configuration in the downloaded Spark directory. Spark is started by executing ./sbin/start-all.sh before deployment. There are two deploy modes, *cluster* and *client*, to launch the spark application on YARN, which are distinguished by the parameter in the script which is used to submit the application. To launch it in cluster mode, the script is "./bin/spark-submit --master yarn --deploy-mode cluster".

To expand on the above reference to our solution and shed light on the technologies used and each component's purpose a summary of the pipeline's components are listed below.

Component	Technology	Responsibility
User Interface	HTML, CSS, JavaScript, Bootstrap, Dragula, NodeJS, Express	Allows users to create and view the state and performance of models

Model Metadata Storage	MongoDB	Persists model metadata, accessed through
REST API	NodeJS, Express	Stores, updates, retrieves model metadata and manages the scheduling queue
Model Generator	Spark, Hadoop	Generates models using the user defined tuning parameters as input. Checks the scheduler to see if any models are ready to run. Retry multiple times before model deployment fail.
Data Storage	Hive, Hadoop	Holding Raw Transaction Data, Calculated Features
Scoring Engine	Spark, Hadoop	Where deployed models live, scores incoming transaction using current models in engine

Table 5.2. Pipeline components

We built a modular pipeline that is capable of continuously training and retraining models. These models are defined by the user to ensure the user has the final say in the algorithm and tuning parameters that go into creating the model. As long as a model is enabled inside the pipeline, the model will continue to be added to the queue after updating, and therefore, will continue to be retrained. The models are retrained based on the the retrain frequency chosen by the user. These models will also be continuously selected and deployed to the Scoring Engine. This pipeline ensures that models never get old and no longer reflect the real world the automated retraining allows users to not even consider models once they have defined them.

5.2 User Interface

5.2.1 Iterations

In order to determine what the user interface needed to display, the main responsibilities of the pipeline and their intersection needed to be identified. These responsibilities were determined through conversations with our sponsor. The overall final results of these for the user interface conversations can be found in Table 3.2 in the User Interface section. At a high level, the pipeline is responsible for: training, storing and deploying models. A user is responsible for creating and monitoring their models. Those two responsibilities spawned the first to views: Create and View. The Create view is responsible for giving the user enough flexibility to create the models they want that still adhere to the specifications of the input to the pipeline (Requirements 1, 2). The View view is responsible for allowing users to see the models they created, their metadata, and their results. In the first version of the User Interface, these two views were developed. The User Interface generated and received requests to and from the model metadata storage through a REST API developed as a part of the pipeline.

To accommodate the user's ability to generate models, in our first implementation the user submitted a model by filling out fields specified through gathering requirements during meetings with our sponsor which are listed below. Because of the nature of this interaction, a form was used as the interface for users to create models. The form contained input validation to ensure that only valid information is passed to the REST API. It was built using HTML, CSS, JavaScript, Dragula and sat on top of a local NodeJS server which sent and received requests using the express library.

Technology	Use
HTML	Language for creating web pages
CSS	Stylizing web pages
JavaScript	Populating tables with model information, handling click events for submitting and receiving information.
Bootstrap	Create responsive web page with interactive modals.
Dragula	Used to enhance user experience when choosing which features to include.

Table 5.3. Technologies used in front end.

Our pipeline doesn't use multiple data sets because the application is scoring transactions which come from a single data set. The primary concern with this new system is ensuring the models are trained on recent data, so the "data date range" parameter was added which refers to the date range over which the data is sampled from for input to the model. Additionally, the retrain frequency was added which indicates how often the model should be retrained. The automated retraining is an essential part of this pipeline and sets it apart from other pipelines where retraining is done manually. Giving the end user the opportunity to choose the date range of data gives them agency over the specificity of the data and is an advantage of the system. At this state and in future implementations, the drawback of the Chronos system is not giving the user the ability to choose the dataset that the model is trained on, but that can be implemented at a later date and isn't imperative to the needs of our current users.

Based on the initial requirements gathered through conversations with the customer, the first user interface contained the following fields: name, author, algorithm, output, features, data date range, and train frequency.

The justifications for why each parameter was included are listed below:

Parameter	Justification
Name	To allow users to identify the model in addition to its ID.
Author	To show who is responsible for the model. In future versions, users can see the models they created and others created.
Algorithm	The machine learning algorithm generating the model. Gives users a say in how the model is created.
Output	Allows users to decide what they want to be the output of the algorithm.
Features	Users generating models can add features they believe to be pertinent to the problem or subproblem they are solving.
Data date range	Changes with users' perception of the problem and allows them to choose between more historical vs. recent patterns.
Re-train frequency	To allow the users to determine how often the model is retrained on the newest data within the aforementioned data date range.

Table 5.4. Justification for parameters chosen by user.

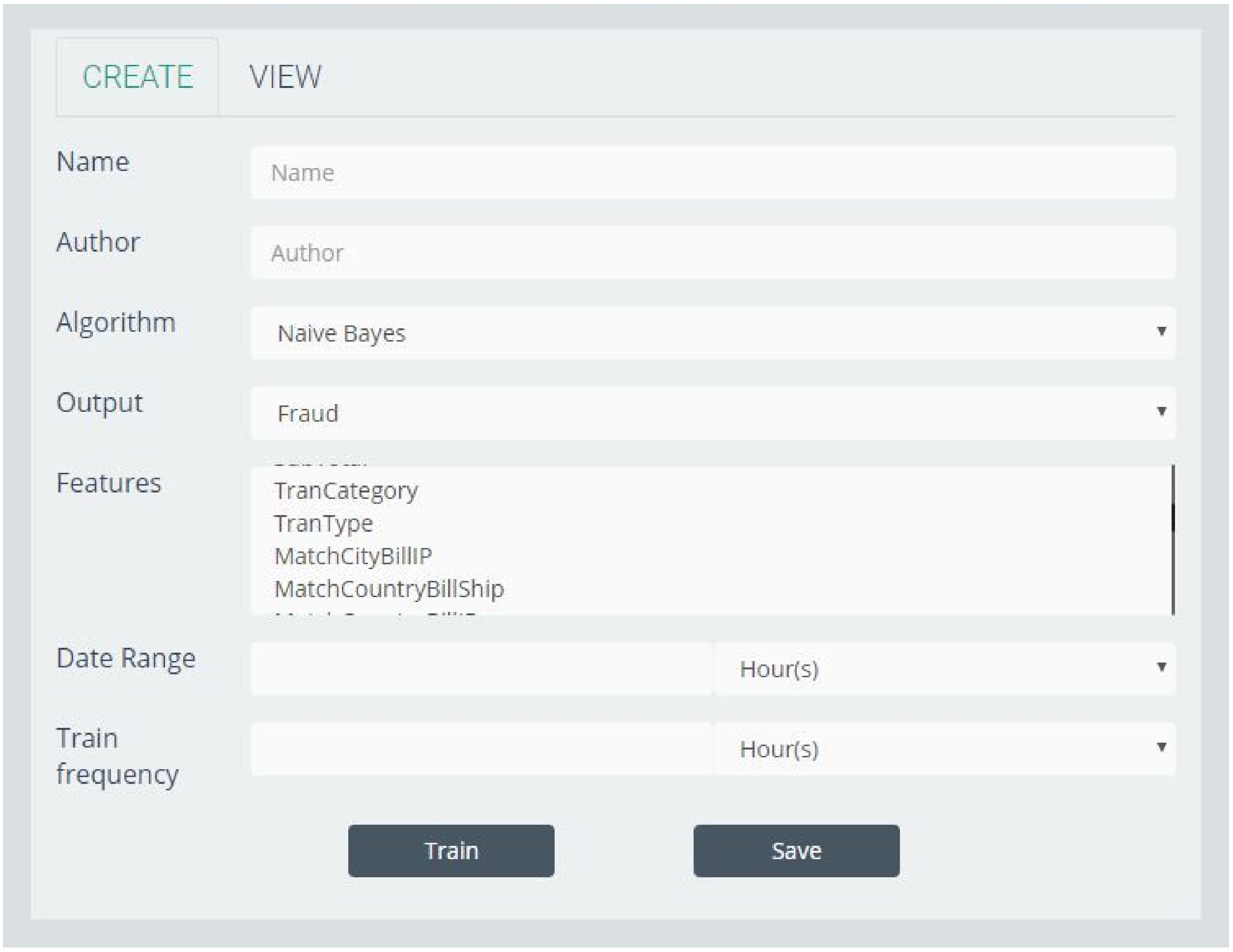
Skytree is a comparable system to ours and we wanted to ensure that our system had advantages that Skytree didn't offer. In the View view, users can view detailed attributes about a given model. Listed below are the attributes in the Skytree API model object versus those included in our model (Skytree, 2017).

Field	Skytree	Chronos Model
ID	X	X
Name	X	X
Dataset	X	
Data date range		X
Creation date	X	X
Updated date	X	X

Algorithm specific configuration	X	X
Algorithm	X	X
Performance measure	X	X
Retrain frequency		X
Accuracy measures	X	X

Table 5.5. Comparison of Skytree parameters vs. Our parameters.

These parameters, as well as conversations with our sponsor, were considered when creating the first version of the interface. The parameters would translate as input fields for the user to enter values. These fields were implemented as input in the create view which is featured below in the first version of the interface.

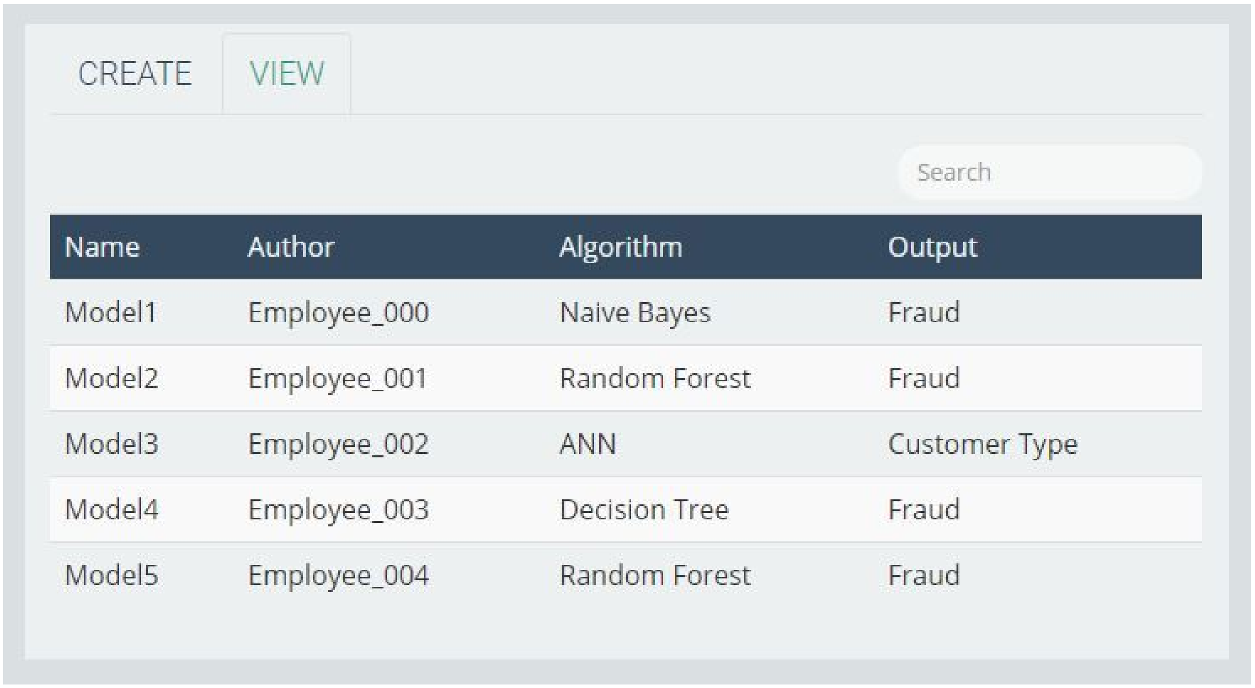


Name	Name
Author	Author
Algorithm	Naive Bayes
Output	Fraud
Features	TranCategory TranType MatchCityBillIP MatchCountryBillShip
Date Range	Hour(s)
Train frequency	Hour(s)

Figure 5.3. Create view for generating models.

On this page, users can enter values for the name, author, algorithm, output, features, data date range, and re-train frequency. The goal of this view was to create an easy to navigate form where the purpose of the parameters is clear. It achieves that goal by having a clean interface and logically ordered inputs. The parameters are displayed in an order that mirrors Skytree's order where it first handles high level metadata, then required and defining parameters followed by additional information.

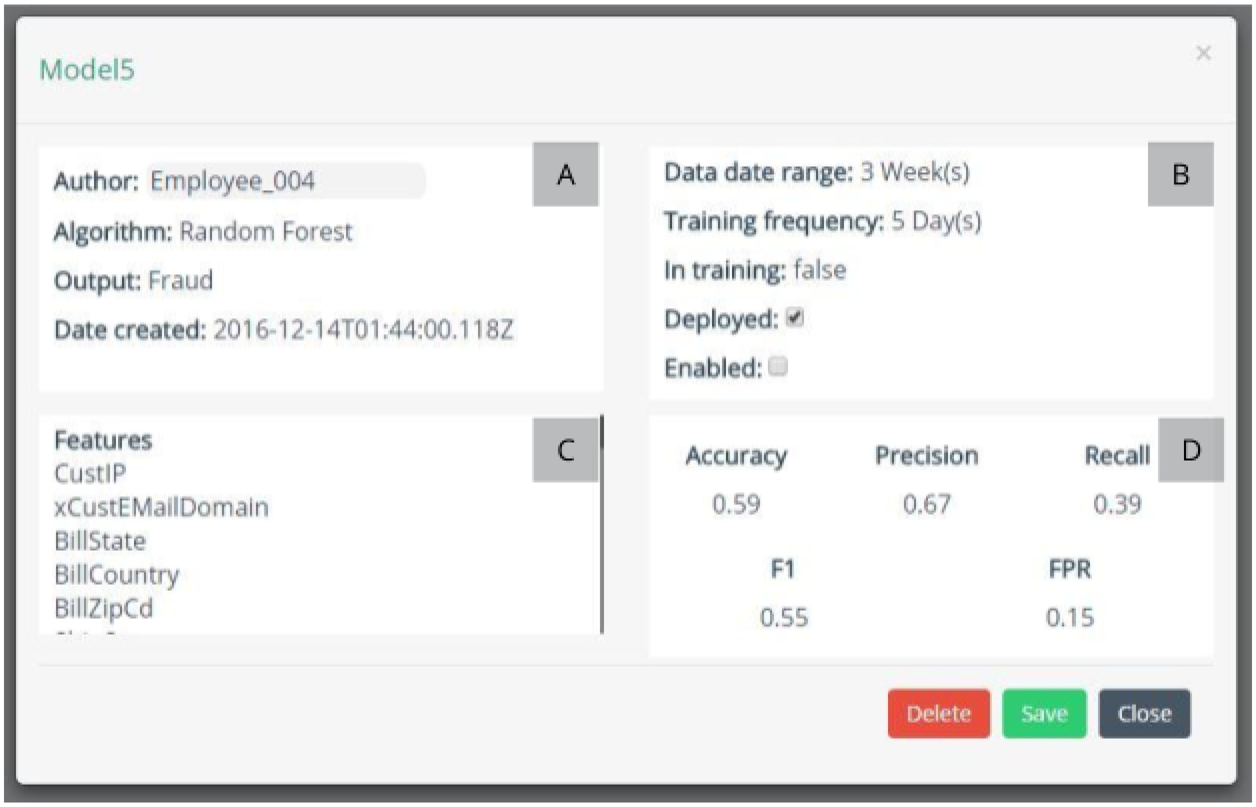
Following the form to enter the model's metadata is a button to train a model on that information. The next view allows users to see the collection of models in a table where the columns represent the basic attributes of a model (name, author, algorithm, output) and each row is a model. The output indicates the output of the algorithm: fraud or not fraud, customer type, etc. There is a search functionality that searches all table columns for a match.



Name	Author	Algorithm	Output
Model1	Employee_000	Naive Bayes	Fraud
Model2	Employee_001	Random Forest	Fraud
Model3	Employee_002	ANN	Customer Type
Model4	Employee_003	Decision Tree	Fraud
Model5	Employee_004	Random Forest	Fraud

Figure 5.4. All models general view.

Users can click on a row to see more information about a model. This includes the models' features, creation date, last train date, accuracy, precision, recall, f1 score, and false positive rate. The purpose of this view is to allow a user to see many models at once and to be able to easily navigate to the model they are interested in further investigating. It achieves that goal by making each model's metadata clear and easy to follow across a row with a clearly defined search bar. This page is laid in such a way that users can quickly visually navigate to the information they are seeking. When a user clicks on one of the models, a modal appears containing detailed information about the model. The purpose of the detailed model view was to give users an opportunity to see the finer details of a model, edit the metadata, and see how it performed in terms of accuracy during testing. This view fulfills that purpose by grouping similar information together into components and using intuitive buttons. The detailed model view is shown below.


Model5

Author: Employee_004
Algorithm: Random Forest
Output: Fraud
Date created: 2016-12-14T01:44:00.118Z

Data date range: 3 Week(s)
Training frequency: 5 Day(s)
In training: false
Deployed: ☑
Enabled: ☐

Features
CustIP
xCustEMailDomain
BillState
BillCountry
BillZipCd

Accuracy	Precision	Recall
0.59	0.67	0.39

F1	FPR
0.55	0.15

Figure 5.5. Detailed model view with basic and detailed information, features, and accuracy measures.

Each labeled section of the first version of the user interface is supported following the image.

Label	Item	Description
A	Basic Information	Basic, high level data used to identify the model. Author,algorithm, output of model, creation date
B	Detailed Information	More detailed information about the model that is more of an attribute and less of a defining characteristic. Data range, training frequency,
C	Feature List	List of features as input to the model
D	Accuracy Measures	Precision, accuracy, recall, f1 score and false positive rate of model, populated after model has been created and tested

Table 5.6. Components of the detail view and their descriptions.

We continued to expand on the user interface and added a drag and drop feature for the features because it was difficult to scroll through and select in such a small window. Following this first iteration, we conducted a critique of the interface from a Human Computer Interaction Expert. Following a review of the interface, Professor David Brown made the following comments:

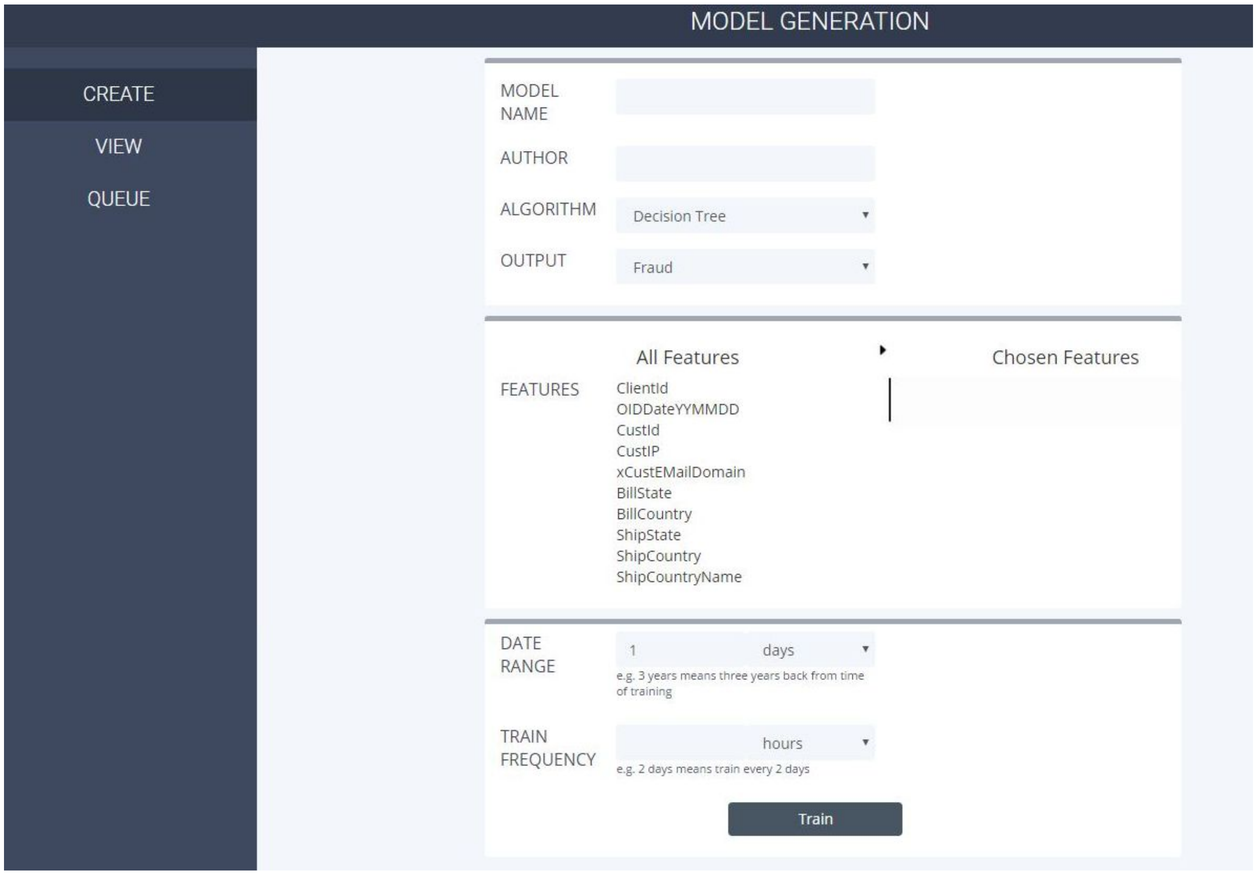
- The blue dialogue box is an odd placement, particularly because the "close" symbol is so far
- The title should be aligned to the left
- Ensure all things are lined up if related/equivalent
- Group items that are closely related
- Ensure that pre-filled boxes aren't redundant of their labels
- Keep drop down arrows near their labels
- Standardize error messages
- Ensure error messages pop up as early as possible

Table 5.7. Comments about improvements for the UI

A more detailed version of these comments can be found in Appendix C. Further conversations revealed that allowing the selection (and display later) of tuning parameters would make the pipeline more appealing to use, given the motivation of allowing users to fully customize their models. Additionally, as we developed a way of abstracting the order that models were trained in, our sponsor indicated it was important for the customer to see the queue of the models. Utilizing the new requirements and feedback, a second interface was developed that grouped similar items, better aligned equivalent items, and featured a navigation bar on the side. This feedback was implemented to generate the final solution which can be found in the following section where we show our final solution.

5.2.2 Final Design

The final User Interface is a web interface using HTML, CSS, JavaScript along with the Bootstrap and Dragula libraries for the client side rendering and page interactions. It runs on a node server which uses the express library to handle sending and receiving HTTP requests. Below are images of the final User Interface. This interface is the product of the above iterations that improved the ease of use through considering human computer interaction principles and fault tolerance through input validation (Requirements 2, 4) The first view is the create view where users can enter values surrounding the tuning parameters and other metadata of a model.


MODEL GENERATION

MODEL NAME

AUTHOR

ALGORITHM: Decision Tree

OUTPUT: Fraud

FEATURES

All Features
ClientId
OIDDateYYMMDD
CustId
CustIP
xCustEMailDomain
BillState
BillCountry
ShipState
ShipCountry
ShipCountryName

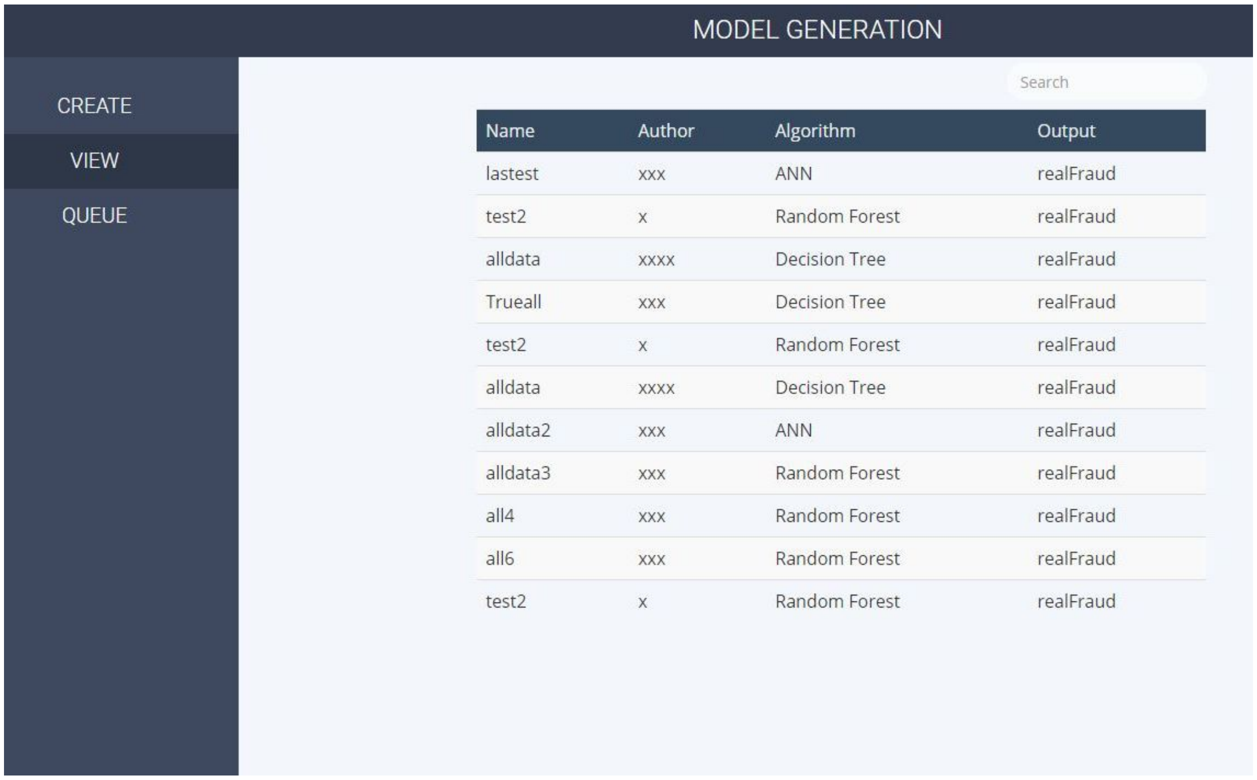
Chosen Features

DATE RANGE: 1 days
e.g. 3 years means three years back from time of training

TRAIN FREQUENCY: hours
e.g. 2 days means train every 2 days

Figure 5.6. The final create view for the system.

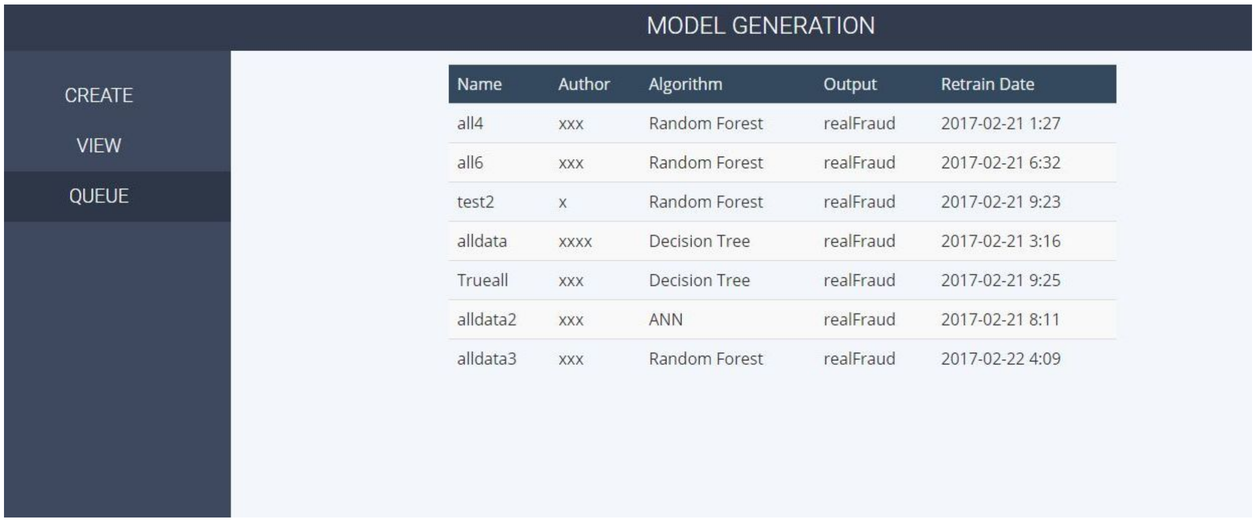
The tabs on the left separate the three main views to make it easy for the user to identify how to accomplish a task (Requirement 2). The next tab brings the user to the view tab which displays a searchable list of all models in the system. In this tab, the users can see some identifying information about each model including its name, author, and algorithm. This page can search for a model based on any of the four displayed values per model.



Name	Author	Algorithm	Output
lastest	xxx	ANN	realFraud
test2	x	Random Forest	realFraud
alldata	xxxx	Decision Tree	realFraud
Trueall	xxx	Decision Tree	realFraud
test2	x	Random Forest	realFraud
alldata	xxxx	Decision Tree	realFraud
alldata2	xxx	ANN	realFraud
alldata3	xxx	Random Forest	realFraud
all4	xxx	Random Forest	realFraud
all6	xxx	Random Forest	realFraud
test2	x	Random Forest	realFraud

Figure 5.7. The final general view for the system.

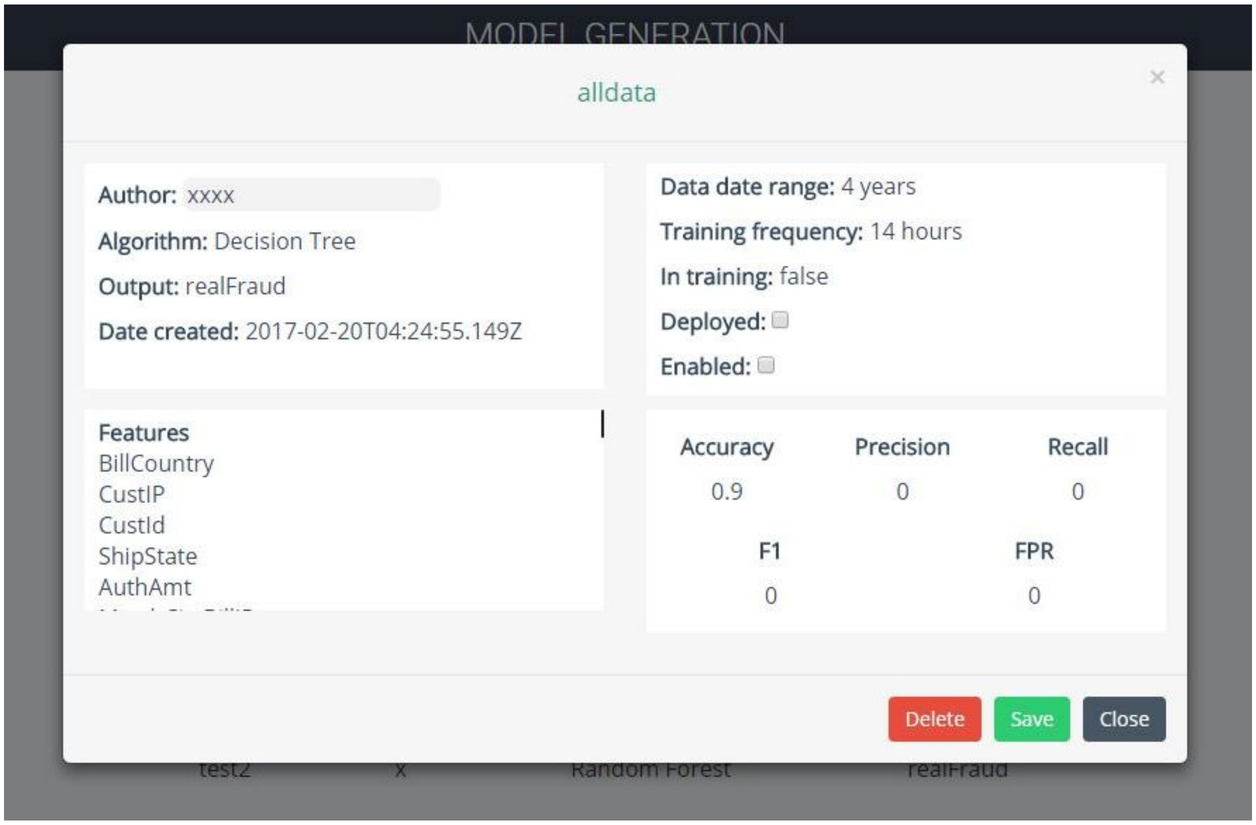
The third and final tab displays the queue which represents the order in which models will be trained or retrained. This was important in ensuring transparency between the system and the user. It contains the same four fields as the general view tab, but adds a retrain date, so it is evident when each model will train. At first glance, the queue view may appear to be very similar to the general view, but the queue view represents a subset of models included in the general view. It represents those models that are waiting to be trained and are therefore enabled. When the system scales up, this view will become more important to see which models are about to be trained and to track the system status in general.



Name	Author	Algorithm	Output	Retrain Date
all4	xxx	Random Forest	realFraud	2017-02-21 1:27
all6	xxx	Random Forest	realFraud	2017-02-21 6:32
test2	x	Random Forest	realFraud	2017-02-21 9:23
alldata	xxxx	Decision Tree	realFraud	2017-02-21 3:16
Trueall	xxx	Decision Tree	realFraud	2017-02-21 9:25
alldata2	xxx	ANN	realFraud	2017-02-21 8:11
alldata3	xxx	Random Forest	realFraud	2017-02-22 4:09

Figure 5.8. The final queue view for the system.

The next three images show the result of clicking on any model in either the "view" view or the queue view. They differ in their algorithms and therefore their tuning parameters, so they each have a slightly varied display (Requirement 1). In this first view, a Decision Tree algorithm is shown, which has no tuning parameters, so it shows only the basic information about the model.



Author: xxxx
Algorithm: Decision Tree
Output: realFraud
Date created: 2017-02-20T04:24:55.149Z

Data date range: 4 years
Training frequency: 14 hours
In training: false
Deployed: ☐
Enabled: ☐

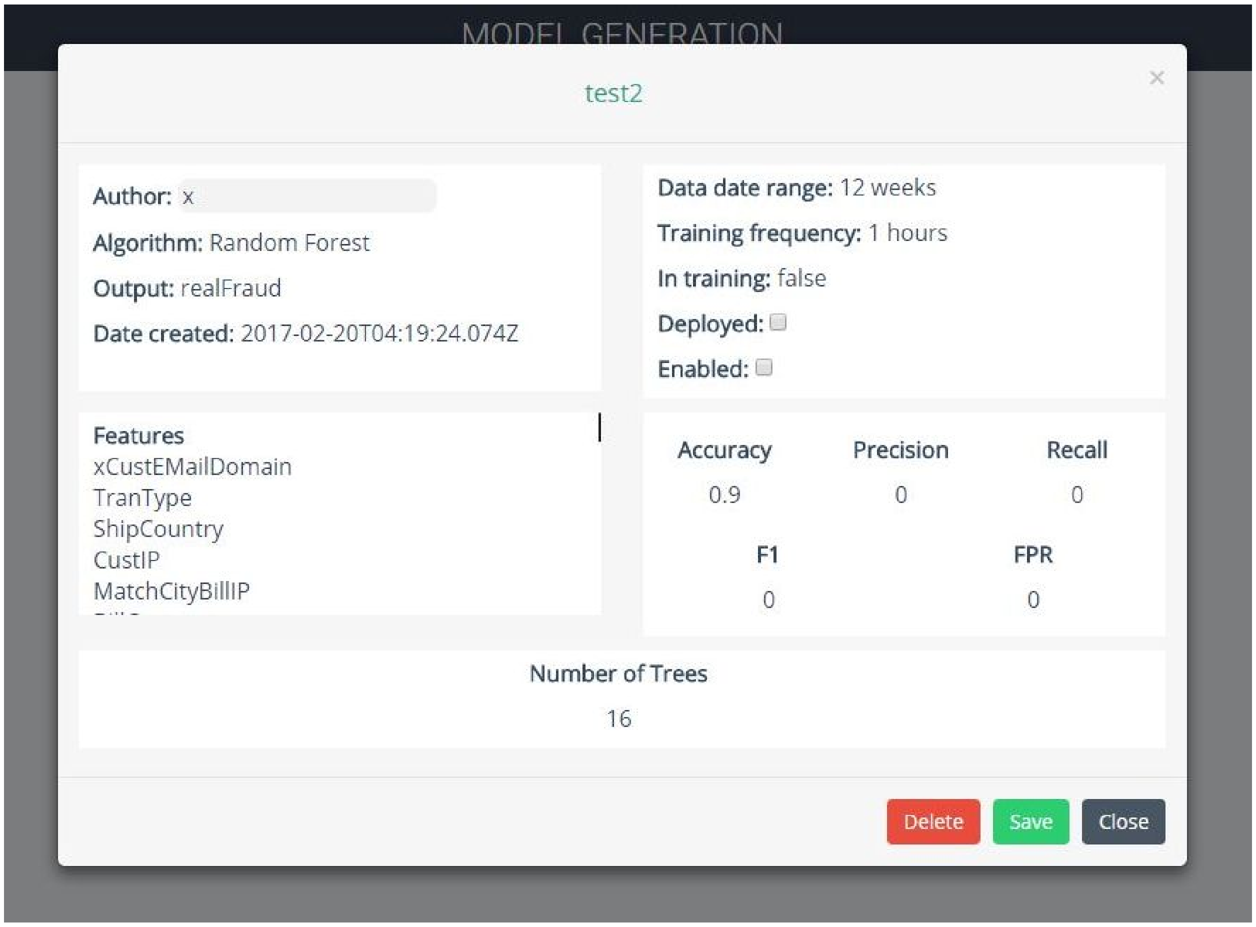
Features
BillCountry
CustIP
CustId
ShipState
AuthAmt

Accuracy	Precision	Recall
0.9	0	0

F1	FPR
0	0

Figure 5.9. The final Decision Tree detailed view for the system.

Next, this view displays the results of clicking on a Random Forest model. This model only has the number of trees as its parameter in addition to the other general attributes whose values are displayed here. Notably, here you can see that the user has the option to delete a model or save an updated author name, deployed status, or enabled status.



Author: x

Algorithm: Random Forest

Output: realFraud

Date created: 2017-02-20T04:19:24.074Z

Data date range: 12 weeks

Training frequency: 1 hours

In training: false

Deployed: ☐

Enabled: ☐

Features
xCustEMailDomain
TranType
ShipCountry
CustIP
MatchCityBillIP

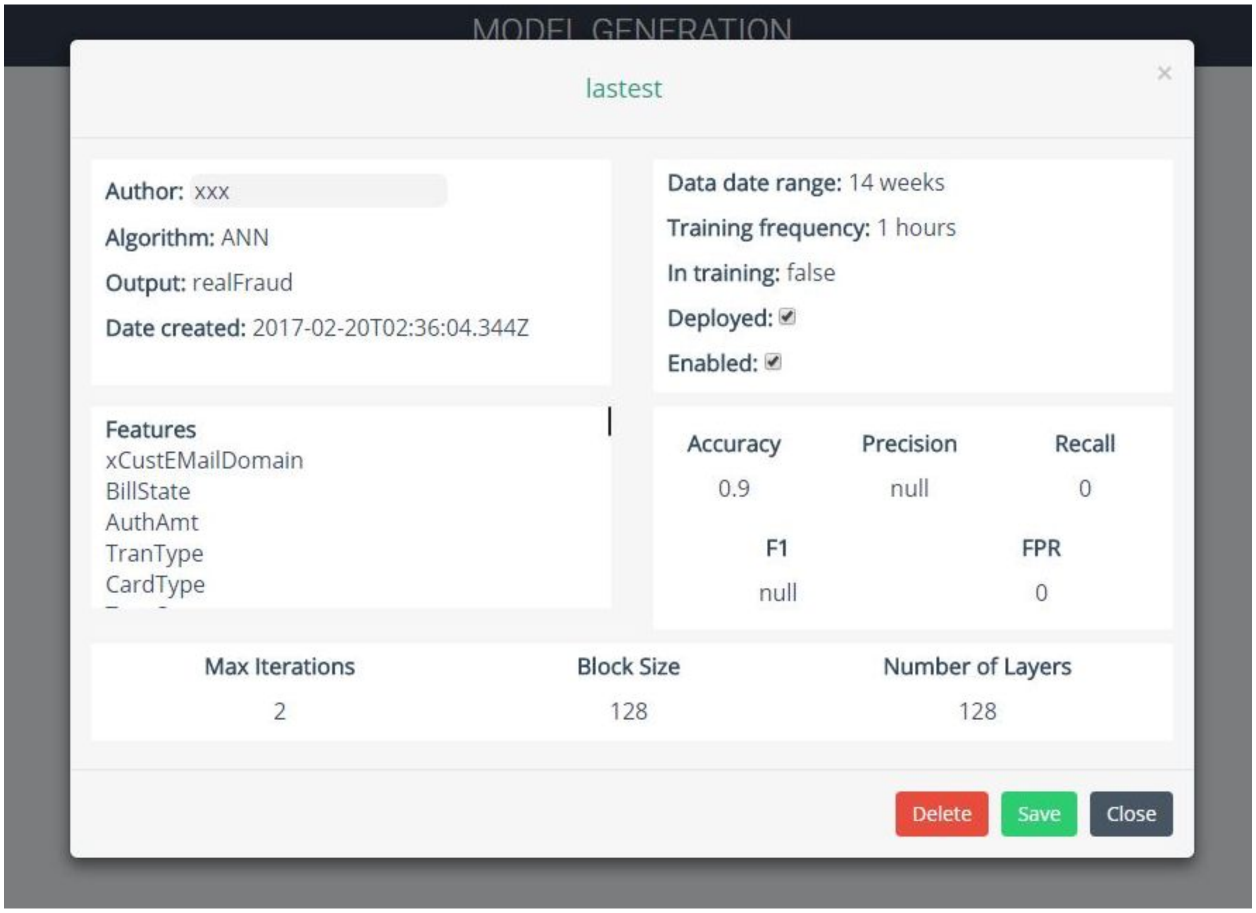
Accuracy	Precision	Recall
0.9	0	0

F1	FPR
0	0

Number of Trees
16

Figure 5.10. The final Random Forest detailed view for the system.

Finally, this view shows the algorithm with the most tuning parameters: the Artificial Neural Network. This view shows the following tuning parameters: maximum iterations, block size, and number of layers. The only relevant tuning parameter that is hidden is the number of nodes per hidden layer, but we intentionally excluded that out of fear of bloating the user interface.



lastest

Author: xxx
Algorithm: ANN
Output: realFraud
Date created: 2017-02-20T02:36:04.344Z

Data date range: 14 weeks
Training frequency: 1 hours
In training: false
Deployed: ✓
Enabled: ✓

Features
xCustEMailDomain
BillState
AuthAmt
TranType
CardType

Accuracy	Precision	Recall
0.9	null	0

F1	FPR
null	0

Max Iterations	Block Size	Number of Layers
2	128	128

Figure 5.11. The final Artificial Neural Network detailed view for the system.

Figures 5.6 to 5.11 represent the final views for the user interface. This interface contains all the requirements and parameters identified as necessary to allow the user to create models for the pipeline. The create view was the one that changed the most throughout the iterations because it was the one we felt was most important to be intuitive to ensure we got buy in from the potential users. The detail views show their ability to adapt its view based on the model at hand. In addition to displaying the system, we added input validation to ensure no faulty data could enter the pipeline from the User Interface (Requirement 4). An in-depth evaluation of the user interface can be found in Section 6.2.

5.3 Model Metadata Storage

5.3.1 Iterations

When deciding the schema of our objects for MongoDB, we wanted to ensure that we were capturing all the information without bloating the model. The model had to support the user interface among other components, but the attributes were derived from the conversations about the user interface as those directly translated to what attributes a model should contain. Our initial model metadata in MongoDB is shown here:

```
{
"_id" : ObjectId,
"task" : String,
"name" : String,
"author" : String,
"dateNum" : int,
"dateSpan" : String,
"dateFrequencyNum" : int,
"dateFrequencySpan" : String,
"algorithm" : String,
"output" : String,
"features : String[],
"file_location": String,
"created" : Date,
"last_trained" : Date,
"deployed" : String,
"enabled" : String,
}
```

Following this implementation, the User Interface requirements grew to include showing tuning parameters and performance metrics. Additionally, we added attributes for fault tolerance purposes to keep better track of the state of the model that the metadata defined. Our iterations proved that the following additional attributes were necessary through conversations with our sponsor and ensuring our models could support the User Interface design principles we were aiming to achieve.

Attribute	Reasoning
Accuracy	Performance metric
Precision	Performance metric
Recall	Performance metric
F1	Performance metric
FPR	Performance metric
inTraining	Indicates if that model is currently in training to avoid collisions
numTrees	Random forest only ,indicates number of trees for the random forest

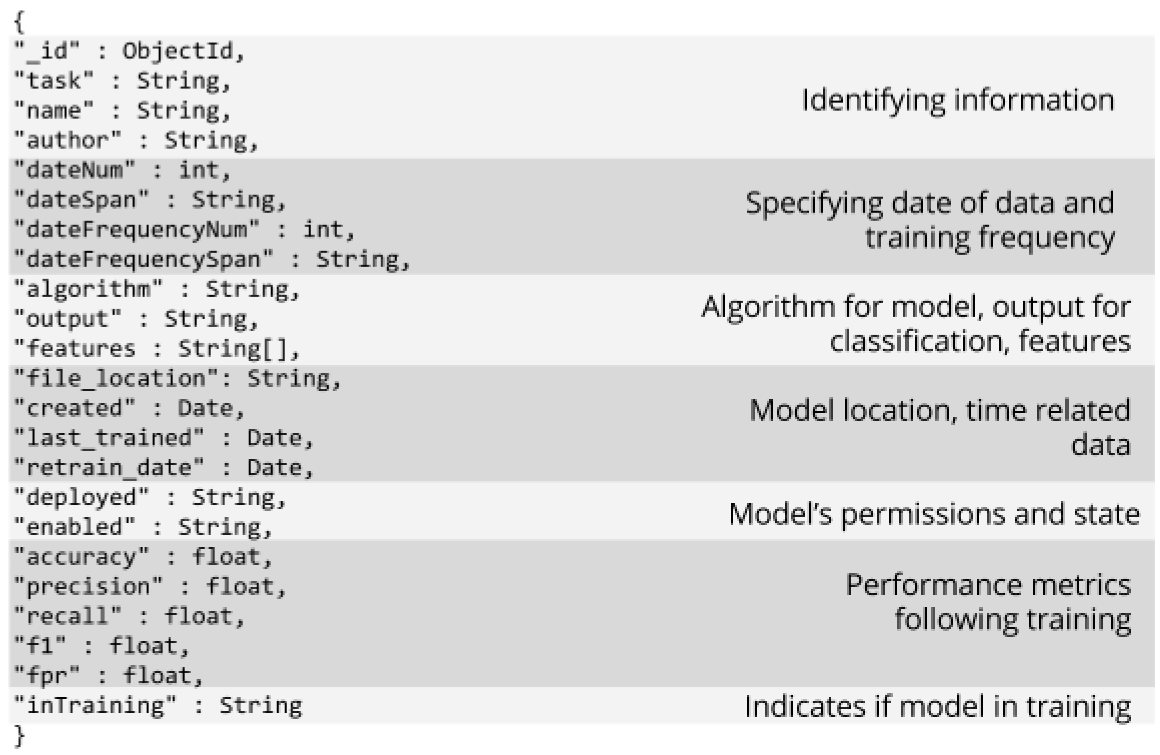
blockSize	Artificial neural net only, indicates block size for the algorithm
maxIter	Artificial neural net only, indicates maximum iterations
layerNum	Artificial neural net only, indicates number of layers
layerNumNodes	Artificial neural net only, array indicating how many nodes per layer

Table 5.8. List of attributes that were added in the second implementation

The final MongoDB object used to hold this metadata is a combination of the old attributes and the ones above identified throughout the implementation. A final view and description of all attributes can be found in the final design section.

5.3.2 Final Design

Metadata about models is stored per object as a record in MongoDB. Below is the schema for the metadata of a model and an explanation of the attributes:



```
{
"_id" : ObjectId,
"task" : String,
"name" : String,
"author" : String,
"dateNum" : int,
"dateSpan" : String,
"dateFrequencyNum" : int,
"dateFrequencySpan" : String,
"algorithm" : String,
"output" : String,
"features : String[],
"file_location": String,
"created" : Date,
"last_trained" : Date,
"retrain_date" : Date,
"deployed" : String,
"enabled" : String,
"accuracy" : float,
"precision" : float,
"recall" : float,
"f1" : float,
"fpr" : float,
"inTraining" : String
}
```

Identifying information
Specifying date of data and training frequency
Algorithm for model, output for classification, features
Model location, time related data
Model's permissions and state
Performance metrics following training
Indicates if model in training

Figure 5.12. Schema for metadata of model

In the following table, each attribute is described in more detail. This shows how the attributes appear, for use when writing queries, as will be discussed in the REST API section.

Attribute	Type	Description	Example
task	string	The task being performed on the model	Default: “create”
name	string	The name of the model	“model1”
author	string	The name of the user who created the model	“Brian”
dateNum	int	The number of unit of time measurements in dateSpan to use for training data	30
dateSpan	int	The unit of measurement used for time in training data	“days”
dateFrequencyNum	int	The number of units of time measurements in dateFrequencySpan for how often model is retrained	30
dateFrequencySpan	int	The unit of measurement for time training frequency	“days”
algorithm	string	The algorithm used to generate the model	“decision tree”
output	string	The desired output from the model	Fraud, Customer segment
features	array	The features used for creating the model	[feature1,feature2]
file_location	string	The location of where the model is stored, in order to get the model later	“/home/store”
numTrees	int	The number of trees in a Random Forest model	5 Required only for Random Forest algorithm
maxIter	int	The maximum number of iterations made in an ANN model	100 Required only for ANN algorithm

blockSize	int	The size for stacking input data in matrices in an ANN model	128 Required only for ANN algorithm
numLayers	int	The number of hidden layers in an ANN model	3 Required only for ANN algorithm
layerNodeNums	array	Contains the number of nodes at each hidden layer in an ANN model	[1,2,3] Required only for ANN algorithm
created	date	The date when the user created the model information	By default, the date is created when CreateModel is called
last_trained	date	The date that the model was last trained on	Default: null
retrain_date	date	The date that the model is ready for training or retraining	By default, the retrain date is the created date
deployed	string	Tells whether or not the model is deployed	Default: false (values can only be true or false, as a string)
enabled	string	Tells whether or not the model is enabled	Default: true (values can only be true or false, as a string)
accuracy	int	The accuracy of the model, written as a decimal	Default: null
precision	int	The precision of the model, written as a decimal	Default: null
recall	int	The percentage of all instances of “not fraud” labeled	Default: null

		correctly, written as a decimal	
f1	int	The mean value of precision and recall, written as a decimal	Default: null
fpr	int	The percentage of all instances of “fraud” labeled correctly, written as a decimal	Default: null
inTraining	string	Tells whether or not the model is being trained	Default: false (values can only be true or false, as a string)

Table 5.9. All model attributes.

The values of these attributes change over time depending on the state of the model and vary between models per their user-specified parameters. MongoDB remained the technology of choice for the duration of the model metadata storage development due to its ability to scale horizontally and hold all relevant model information (Requirements 3, 6).

5.4 REST API

5.4.1 Iterations

The first implementation of the REST API contained the basic CRUD operations (Create, Read, Update, Delete). For the CreateModel function, a model was created using parameters provided by the query parameters (Requirement 1). The list of attributes belonging to a model are listed in the table below. For Read operations, the REST API supported GetNewModel and GetModel. GetNewModel was used to get a new model for training. This function would first check to see if there were any new models ready to be trained. If so, that model was returned. Otherwise, GetNewModel checked to see if there were any models ready for retraining. If there are, the chosen model is disabled and a new model is created with the same initial parameters, then this new model is returned. If no models are ready for retraining, GetNewModel returned “false”. GetModel got at most five deployed models and returned those models as an array for the Scoring Engine (Requirement 3). The next function was UpdateModel, which updated a model with the given model name with the provided accuracy and file location. This function also updated the deploy status to “true”. Finally, the REST API had DeleteModel. This function deleted the model from MongoDB with the specified model name.

Attribute	Type	Description	Example

task	string	The task being performed on the model	Default: “create”
name	string	The name of the model	“model1”
author	string	The name of the user who created the model	“Brian”
dateNum	int	The number of unit of time measurements in dateSpan to use for training data	30
dateSpan	int	The unit of measurement used for time in training data	“days”
dateFrequencyNum	int	The number of unit of time measurements in dateFrequencySpan for how often model is retrained	30
dateFrequencySpan	int	The unit of measurement for time training frequency	“days”
algorithm	string	The algorithm used to generate the model	“decision tree”
output	string	The desired output from the model	Fraud, Customer segment
features	array	The features used for creating the model	[feature1,feature2]
file_location	string	The location of where the model is stored, in order to get the model later	“/home/store”
created	date	The date when the user created the model information	By default, the date is created when CreateModel is called
last_trained	date	The date that the model was last trained on	Default: null
deployed	string	Tells whether or not the model is deployed	Default: false (values can only be true or false, as a

			string)
enabled	string	Tells whether or not the model is enabled	Default: true (values can only be true or false, as a string)
accuracy	int	The accuracy of the model, written as a decimal	Default: null
inTraining	string	Tells whether or not the model is being trained	Default: false (values can only be true or false, as a string)

Table 5.10. Original set of parameters in model metadata

In the next iteration, two new functions were added called GetAll and UpdateModelInfo to support actions identified as requirements in the user interface: viewing all models and updating them. GetAll returned every model in MongoDB, including the disabled models. UpdateModelInfo allowed for updating a model's name, author, deploy and enable status when also provided the chosen model's name (Requirement 1). As for other improvements, models included the statistical parameters: precision, recall, f1, fpr. At this point, it was noted our current system for grabbing the next model was insufficient and would favor new models. The solution to this problem was implementing a scheduling queue where the items are the enabled models waiting to be trained. GetNewModel used this queue in order to determine which model to attempt to train or retrain. The queue allowed for every model to have a chance to be called next for training or retraining, rather than waiting for all of the new models to be trained first. We discussed the possibility of a system where a user sees their own models to represent how the system would behave once scaled. A new file called authors.txt was also added that stored all of the authors of models so that GetModel could get the array of models for a specific author.

The method of using a model's name for identifying models in these functions were replaced by the _id parameter generated by MongoDB. This change was made because a model name does not necessarily have to be unique. Models were also given new parameters that were used to allow the user to control the algorithm parameters for training a model. Unlike previous implementations, the model data will not necessarily have all of these parameters listed but rather just the parameters necessary for the specified algorithm (Requirement 1). Another parameter called retrain_date was also added to hold the date for when a model needs to be trained or retrained. The queue became a priority queue, which sorts the models based on the retain_date. GetNewModel was modified to just check the first model in the queue. This function will try to train or retrain the model only if current date is or past the retrain date. UpdateModelInfo was improved to also update the queue and authors.txt as necessary if a model's information is updated. A few new functions were also added. Two of the functions, ModelStatus and UpdateTraining, are designed to help with the CRON Job running for the Model Generator. ModelStatus returns the deployment status of the specified model and UpdateTraining is meant for if the ModelGenerator fails, in which case, the function changes the deployment status to "failed". Our sponsor indicated it would be valuable for the queue to display on the user interface so the third function, GetQueue, returns the queue as an array. For more information about the completed REST API, see the Final Design section.

5.4.2 Final Design

Many of the components of this project are interconnected via a REST API. This API handles receiving and sending information between the user interface, queue, and model generator. Table 5.9 shows all of the attributes for a model. Any attribute not given a default in the table is an attribute specified when creating a model.

The following table is an overview of all of the functions. This table shows an example of how to run each of the functions, as well as what the result is from running the function. Any alternate or optional parameters that could be used in the URL can be found in the previous tables.

CreateModel	
URL Example	`/CreateModel?name=m1&author=Ben&dateNum=30&dateSpan=days&dateFrequencyNum=30&dateFrequencySpan=days&algorithm=Random Forest&output=fraud&features=[f1,f2]`
Result	This function does not return anything, but adds an entry to mongoDB with the provided model information and all other attributes for a model set to their default values.
UpdateModel	
URL Example	`/UpdateModel?idi=23aff3124e&file_location=/home/store&accuracy=0.5`
Result	This function does not return anything, but updates the enabled entry in mongoDB with the matching model ID as the one provided in the function call. In this case, the file location and statistics are updated using the information from the query.
UpdateModelInfo	

URL Example	`/UpdateModelInfo?id=23aff3124e&deployed=false&enabled=false&nname=model1`
Result	This function does not return anything, but updates the enabled entry in mongoDB with the matching model ID as the one provided in the function call. In this case, the model is disabled (enable set to "false") and the model is no longer deployed. Also, in the case of this example, the name of the model will be changed to "model1"
DeleteModel	
URL Example	`/DeleteModel?id=23aff3124e`
Result	This function does not return anything, but deletes every model in mongoDB with the matching model ID as the one provided in the function call.
GetModel	
URL Example	`/GetModel`
Result	This function returns an array of model entries from mongoDB. Each entry is a dictionary object. Only the enabled models for a single user are returned, determined by the server-side list of authors stored in authors.txt.
GetNewModel	
URL Example	`/GetNewModel`
Result	This function returns one model entry for training or retraining. The model entry is a dictionary object. Returns "false" if there is no model in MongoDB or if there are no models ready to be trained or retrained.
GetAll	
URL Example	`/getall`
Result	This function returns every model entry in MongoDB.
ModelStatus	
URL Example	`/modelstatus?id=23aff3124e`
Result	This function returns the deployment status of the specified model
UpdateTraining	
URL Example	`/updatetraining?id=23aff3124e`

Result	This function updates the inTraining parameter of the specified model to "fail"
GetQueue	
URL Example	`/getqueue`
Result	This function returns the queue

Table 5.11. Table of function call examples and their results.

Below is the table of the parameters used for the CreateModel function. Every attribute listed is required for the function. This function takes all of the parameters specified by the user and enters them into MongoDB as a new set of model data (Requirement 1).

Attribute	Type	Description	Example
name	string	The name of the model	"model1"
author	string	The name of the user who created the model	"Brian"
dateNum	int	The number of unit of time measurements in dateSpan to use for training data	30
dateSpan	int	The unit of measurement used for time in training data	"days"
dateFrequencyNum	int	The number of units of time measurements in dateFrequencySpan for how often model is retrained	30
dateFrequencySpan	int	The unit of measurement for time training frequency	"days"
algorithm	string	The algorithm used to generate the model	"decision tree"
output	string	The desired output from the model	Fraud, Customer segment
features	array	The features used for creating the model	[feature1,feature2]

numTrees	int	The number of trees in a Random Forest model	5 Required only for Random Forest algorithm
maxIter	int	The maximum number of iterations made in an ANN model	100 Required only for ANN algorithm
blockSize	int	The size for stacking input data in matrices in an ANN model	128 Required only for ANN algorithm
numLayers	int	The number of hidden layers in an ANN model	3 Required only for ANN algorithm
layerNodeNums	array	Contains the number of nodes at each hidden layer in an ANN model	[1,2,3] Required only for ANN algorithm

Table 5.12. Table of required attributes for CreateModel.

Below is the table of the parameters used for the UpdateModel function. Every attribute listed is required for the function. This function updates the model that has just been trained or retrained by updating the information about where the model is stored, the statistics of the model, the last date trained and the retrain date.

Attribute	Type	Description	Example
_id	string	The id of a model (this value is generated automatically by MongoDB)	"23aff3124e"
file_location	string	The location of where the model is stored, in order to get the model later	"/home/store"
accuracy	int	The accuracy of the model, written as a decimal	0.5
precision	int	The precision of the model, written as a decimal	Default: null
recall	int	The percentage of all instances of "not fraud" labeled correctly, written as a decimal	Default: null
f1	int	The mean value of precision and recall, written as a decimal	Default: null

fpr	int	The percentage of all instances of “fraud” labeled correctly, written as a decimal	Default: null

Table 5.13. Table of required attributes for UpdateModel

Below is the table of the attributes used for the UpdateModelInfo function. Every attribute listed is required for the function. This function is intended for use on the user side, and allows for the information stored for the model to be changed (Requirements 1 and 2).

Attribute	Type	Description	Example
_id	string	The id of a model (this value is generated automatically by MongoDB)	“23aff3124e”
deployed	string	Tells whether or not the model is deployed	Default: false (values can only be true or false, as a string)
enabled	string	Tells whether or not the model is enabled	Default: true (values can only be true or false, as a string)
nname	string	(optional) The new name for the model	“newmodel1”
nauthor	string	(optional) The new name for the model author	“Brian”

Table 5.14. Table of required attributes for UpdateModelInfo.

Below is the table of the attributes used for the DeleteModel function. Every attribute listed is required for the function. This function searches for the specified model by the ID of the model, and deletes the model information from MongoDB.

Attribute	Type	Description	Example
_id	string	The id of a model (this value is generated automatically by MongoDB)	“23aff3124e”

Table 5.15. Table of required attributes for DeleteModel.

Below is the table of the attributes used for the ModelStatus function. Every attribute listed is required for the function. This function searches for the specified model by the ID of the model, and returns the deployment status of that model.

Attribute	Type	Description	Example
_id	string	The id of a model (this value is generated automatically by MongoDB)	“23aff3124e”

Table 5.16. Table of required attributes for ModelStatus.

Below is the table of the attributes used for the UpdateTraining function. Every attribute listed is required for the function. This function searches for the specified model by the ID of the model, and updates the inTraining status of the model to “failed".

Attribute	Type	Description	Example
_id	string	The id of a model (this value is generated automatically by MongoDB)	“23aff3124e”

Table 5.17. Table of required attributes for UpdateTraining.

5.5 Model Generator

5.5.1 Iterations

The original technology considered for producing models was Skytree. Skytree is a compact integral machine learning pipeline itself, so we decided it was redundant to integrate it into our pipeline. It runs on the self designed HDFS and Spark as a complete commercial product. Skytree runs independently on a virtual machine and web application in a server which requires a lot of storage. Skytree is similar in nature to the entirety of our project, so we didn’t include it as a component.

Later on in the development, we added more algorithms to support a variety of models, which were identified as important by our sponsor (Requirement 7). The next algorithms to be supported were Artificial Neural Network and Random Forest. Additionally, the Model Generator was altered to take in the tuning parameters for each of these algorithms as specified in the user interface.The code structure was changed to separate files in order to easily add new algorithms (Requirement 2). The code for each algorithm is now stored in a separate file. Additionally, extracting the model information from the UI and the data preparation were separated from the algorithm training part of the code. They were also in an independent file to work as a dispatcher that can choose corresponding algorithm from an algorithm pool and send it user’s parameters.

Finally, in order to prevent application failure during train or retrain, a script is written to rerun the spark submission command (Requirement 4). The user can determine the number of times a model will attempt to train. This improved the robustness of whole pipeline.

5.5.2 Final Design

The Model Generator generates a model with user-specified parameters, running on top of a hadoop cluster with YARN as the resource manager. It interacts with mongoDB through the RESTful API to extract the model metadata defined by a user and utilizes the dataset from the Hive database to train a model using the user-specified machine learning algorithm.

A cron job runs every X hours to check if there is a model that needs to be trained. Cron is a daemon that only needs to be started once and it remains dormant while not in use. The cron job doesn't occupy memory while dormant and will start up on the next defined time even if the previous execution fails. The model generator will be activated to train a model if a new model's metadata is returned through REST API.

The returned object contains the desired algorithm's name, feature names, the date range and the model's output. The model Generator is flexible in that the output is not limited to a predetermined value, but can be used to predict any column in the transaction data. The model's output indicates the column that user wants to predict. The Model Generator ingests the data from Hive using Spark SQL. It selects the column whose name matches the feature names and output specified by the user and stores them in a Spark DataFrame structure.

For further data preparation, the Spark transformation operation was used to combine the list of feature columns into a single vector column and change the name of the column user wants to predict to 'label'. The returned DataFrame is split into a training dataset and a test dataset with predefined proportions. Model Generator then implements user-selected algorithm to train the model on the training dataset. The generated model is then tested on the testing dataset and a python object Evaluator is designed to measure the model's predictive accuracy.

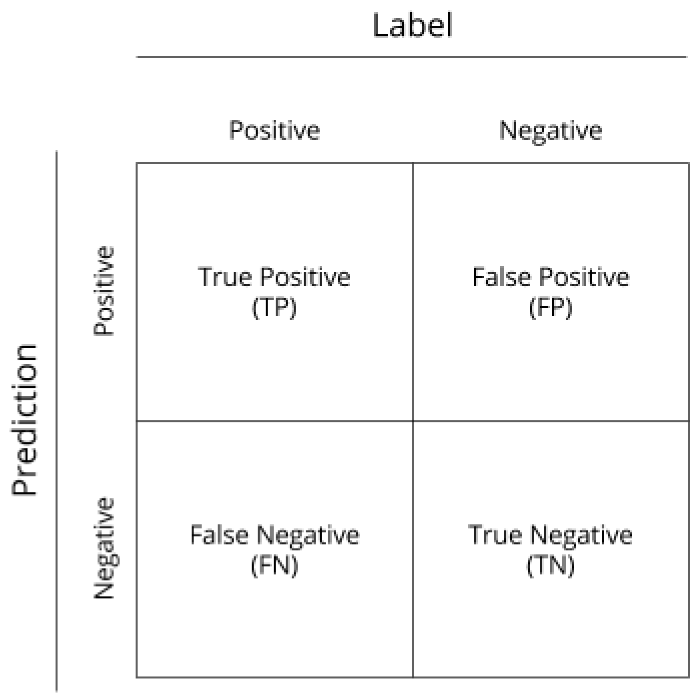


Figure 5.13. Notation Clarification

Five most commonly used statistically matrices are selected for the measurement and notation and their calculations are stated as above (Requirement 7). TP is predicting positive and label is positive; FP is predicting positive but label is negative; FN is predicting negative and label is negative; TN is predicting negative but label is positive. TP and FN represent correct label whereas FN and FP show errors.

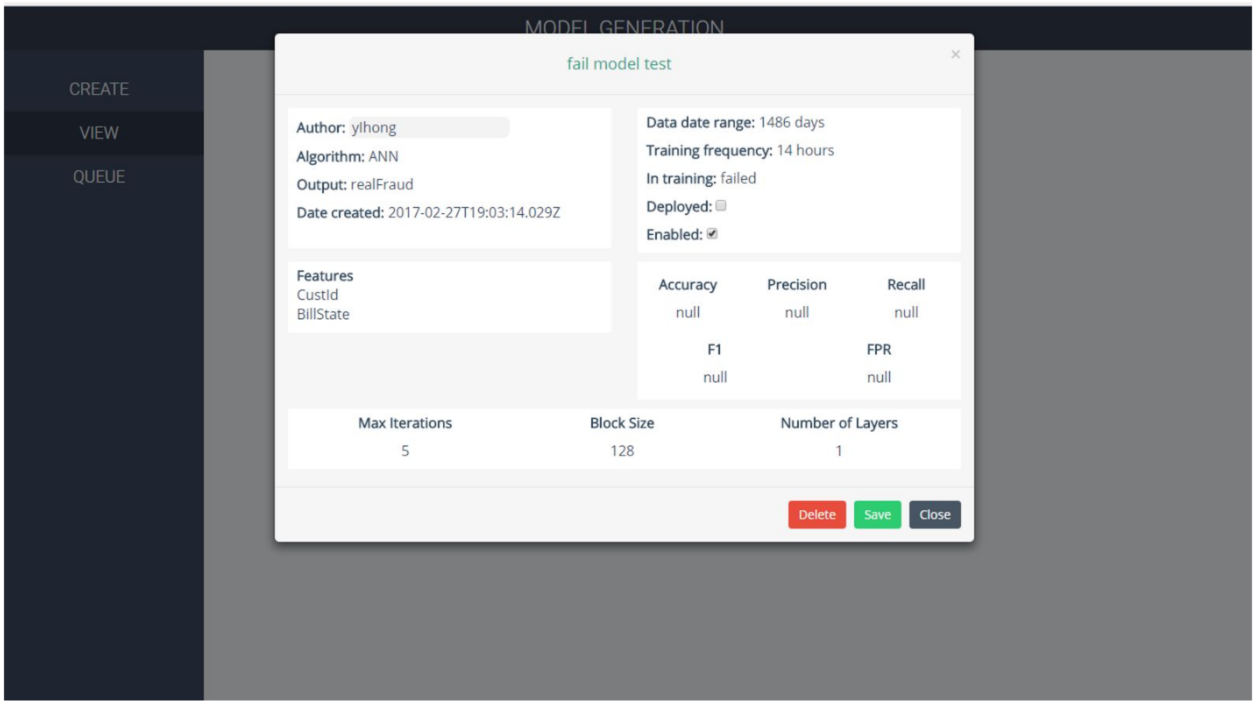
Name	Equation	Description
Accuracy	$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$	The fraction of correct prediction over the total number of predictions.
Precision	$\text{Precision} = \frac{TP}{TP + FP}$	The fraction of actual positive among those predicted as positive, which measures the rate that retrieved instances are relevant.
Recall	$\text{Recall} = \frac{TP}{TP + FN}$	Also known as true positive rate or sensitivity, the fraction of actual positive among those labeled as

		positive, which measure the rate of relevant instances that are retrieved.
False Positive Rate	$\text{False Positive Rate} = \frac{\text{FP}}{\text{TN} + \text{FP}}$	The fraction of those falsely predicted positive instances among those label as negative.
F1-Measure	$\text{F1} - \text{Measure} = \frac{1}{\frac{1}{\text{Recall}} + \frac{1}{\text{Precision}}}$	The harmonic mean of recall and precision.
Area Under Receiver Operating Characteristic (ROC) Curve	$AUROC = \int_0^1 \frac{TP}{P} d\left(\frac{FP}{N}\right)$ (Apache, n.d.)	A ROC curve plots (recall, false positive rate) points at different threshold settings (Apache, n.d.).
Area Under Precision-Recall Curve	$AUPRC = \int_0^1 \frac{TP}{TP+FP} d\left(\frac{TP}{P}\right)$ (Apache, n.d.)	A P-R curve plots (precision, recall) points at different threshold values (Apache, n.d.).

Table 5.18. Binary Classification Evaluation Matrices

The trained model is associated with the file address at which it is stored and performance metrics are written back to mongoDB through the REST API.

The Model Generator is also responsible for trying multiple times to train a model if it fails. Spark servers could be interrupts at any point in the model's training, so the Model Generator allows multiple attempts to rerun the model (Requirement 4). If it has attempted to train more times than allowed then the InTraining attribute on the model will be change to "failed".



fail model test

Author: ylhong
Algorithm: ANN
Output: realFraud
Date created: 2017-02-27T19:03:14.029Z

Data date range: 1486 days
Training frequency: 14 hours
In training: failed
Deployed: ☐
Enabled: ☑

Features
CustId
BillState

Accuracy	Precision	Recall
null	null	null

F1	FPR
null	null

Max Iterations	Block Size	Number of Layers
5	128	1

Figure 5.14. Model deployed Failure

5.6 Feature and Transaction Storage

5.6.1 Iterations

5.6.1.1 Storage Tool

Here is a table comparison based on the features our project supports prior to finalizing the required functionality (Requirement 6).

	Hive	**HBase**	**Cassandra**
Description	Data warehouse software for querying and managing large distributed datasets	Built based on the concept of BigTable. Has cell-level access labels and a server-side programming mechanism	Built based on the idea of BigTable and DynamoDB.
Database model	Relational DB	Wide column storage	Wide column storage

SQL	SQL-like(HQL)	Master-Slave NoSQL databases	NoSQL databases
Advantage	● Good for batch processes, running periodically (maybe in terms of hours or days). ● Analytical queries ● Mainly used for ETL and data warehousing purpose. ● Can present Cassandra and HBase as a "table" that can be "queried" via Hive's language	● Support low latency calls. ● Good for Heavy reads and less Write applications ● Linear Scalability for large tables and range scans ● Fast lookup and random access ● Real-time querying	● Supports low latency calls. ● Good for single-row queries or selecting multiple rows based on a Column-Value index
Disadvantages	● Not as popular as other two engines ● Loss all the goods of real-time processing ● All the disadvantage of relation DB	Not good for Classic transactional applications or even relational analytics and the data that need to be aggregated, rolled up, analyzed cross rows	Not good for transactional operations (Rollback, Commit) and relational data Range-scan is not as good as Hbase
Example		Facebook messenger	Twitter, Travel portal

Table 5.19. Comparison of different storage tools.

Hbase and Hive support the functionality we need for this project. After comparing the two technologies, we decided to choose Hive. Below are the justifications for choosing Hive:

1. Hive is good to use in non real-time situations. Our project does not involve real-time querying. Though the Scoring Engine would run better if it could stream transactions in real-time, we are more focused on the proof of concept that the Scoring Engine can score transaction data. In this project, all data comes from a database and no new data feeds into the Scoring Engine.
2. Hive is good for batch processing. This project does not require fast lookup and random access. Our goal is simply to preprocess data and feed it to the Model Generator in batches. Hive is good for data storage that doesn't update too often, which is the case for our pipeline.

3. ACI is uses a relational database for their transactions. Hive is relational database. Hbase is not a relational database and it uses wide column storage. Hbase is also not good for classic transaction application or even relational analytics where these are Hive's strength.
4. Hive can map to HBase and Cassandra (Requirement 3). Using Hive's language can query Hbase and Cassandra as "table". If this project wants to further expand to using other database or getting information from table using HBase or Cassandra, Hive is a good spring board.
5. Hive is enough for this project. The pipeline we are creating does not require many write operations. Only batch modification when clearing data is required and this is done before running Model Generator.

5.6.1.2 Hive Tables

When considering the layout of our tables, one table which we can extract all information user needed for model training from would be the best to reduce bloat. Also, there is not enough variation or relationships to justify an additional table. The original design for Hive storage only contains the Transaction Table and Dimensional Table. The transaction table stores all the raw transaction data, with features calculated based on single transactions.

There were two initial prototypes for the dimension table. The first version uses a sparse table and treats the Dimension type as the primary key so that different instances can be aggregated based on Dimension types. Some of the instances have attributes that other instances don't have and this will make most of the table contain "NULL" cell entries. Additionally, the sparse table is not good for batch operations and for training. Hive needs to spend time examining each "NULL" space and the Model Generator has to deal with meaningless "NULL" values in each column. The second version is more complicated. The type of each instance forms a table and there is a large table to sum up all instance types. It is good for an individual instance as the targeted table allows free design for each instance and avoids filling missing values with "NULL". Though with better storage efficiency, this version has a drawback of sophisticated data manipulation. To get the information of an instance requires user to jump between multiple tables.

After beginning the implementation, the team emphasized the purpose of a Dimension table and how to make the Dimension more efficient at data storage and at the same time can be quickly manipulated by the Model Generator.

After discussing the roles and responsibilities of the tables, we decided the Dimension table should play a role of helping to calculate the features for the Transaction table, rather than be a data source for Model Generator. The Dimension table's job is to calculate the average, sum and standard deviation of multiple time intervals. These statistics are going to help with computing the difference between population and sample values for a single transaction in the

Transaction table. In conclusion, all features and attributes are going to be in the Transaction table and it will be the final table from which users select features sent into the Model Generator.

Our advisors suggested that there is no need to calculate the difference between the overall performance and single performance of each transaction. As long as there is a difference between the transaction rows, the machine learning model can distinguish them. Therefore, The information provided by Dimension table as a features table and Transaction table without features are enough.

While implementing both tables, the team focused on cleaning up data. The sample data set has around 3800 data and 495 attributes. Not all 495 attributes are important and some of them are missing data. Additionally, because machine learning algorithms only accept data in doubles and in a feature vector form, all strings had to be converted to categorical doubles, even numeric columns in types of int, float and bigint had to convert to doubles and then add the user features into a vector form.

In this process, 2 more tables are constructed for each converting step. A new AllFeaturesTable Table is the final version before selecting user features for training. During this time we changed strings to categorical doubles and cleaned up unhelpful columns, delete rows with “NULL” is the first idea. However, deleting rows with “NULL” cuts off most of the data set and feeding such a data set to machine learning algorithms is meaningless, so filling “-999” was an idea to maintain diversity of the dataset, though it may weigh too much while training and impact the training accuracy.

Hive also stores the result of the model’s scoring from the Scoring Engine. The Scoring Engine result was originally added to the Transaction table. However, there is no need to append a new column as scoring result column and modify every slot when new results come in because it takes long time to do the modification in Hive and users cares more about overall transaction and the Scoring Engine score. So transaction id and its corresponding score, without features and attributes, will be left in result table of Scoring Engine.

5.6.2 Final Design

Both the transaction and feature data are stored in a Hive database, which runs on top of HDFS. HDFS is able to store large files across multiple machines. Orchestrated by the Namenode and Datanodes, HDFS is a favourable place for data storage. Spark SQL can access data stored in HDFS through Hive to get both raw transaction attributes and calculated features. Combining Spark with HDFS, both new data and old data, both attributes and features could be sustained, manipulated with ease.

5.6.2.1 Transaction Storage

The transaction table’s real data is stored in HDFS with its Hive database metadata in physical space. All the transaction data was historical data provided by ACI worldwide, Inc.

With 495 attributes, including name, address, transaction id, date, time, chargeback and manual fraud labels, transaction data are all raw data stored in transaction table. The data quantity can be easily increased or decreased for the user needs.

After the scoring engine scores the corresponding transactions, it produces a new table called “SEresults” for the usage of score checking. The new table needed is to reduce the inconsistency of data on the same table while getting data from old feature table, providing data for Model Generator, and writing and updating new score results.

Attributes from raw transaction….. OID, Customer Name, Address, Merchants, Date, Time….. Chargeback, manual fraud label
…. Total 495 attributes here

Table 5.20. Transaction table structure.

f_oid	Predictions for Model 1	Predictions for Model 2	...	Predictions for Model n	Probability	Scoring Engine Results
….						“fraud”
….						“not fraud”

Table 5.21. SEresults table structure.

5.6.2.2 Feature Storage

All features that can be ingested in the table are stored in dimension tables and feature tables in Hive.

Dimension tables are constructed based on the user selected time interval to compute the average, sum, and standard deviation of transaction amounts and the number of transactions. Four dimension tables are created using Spark SQL and stored in Hive. The sum, average, standard deviation of transaction amounts and number of transactions are calculated over one, seven, thirty, and ninety days. For example, in the dimension table with time span set as seven days, for each card number, it computes the sum , average, standard deviation of transaction amounts and number of transactions happens in past seven days, which consists of today and the previous six days.

Feature tables store the features derived directly from the transaction data’s attributes. It converts different types of attributes such as strings and missing values to numerical values that can be ingested in the model using StringIndexer provided by the Pyspark library. Spark SQL is used to read the transaction data into a Spark DataFrame. To ensure the data isn’t too sparse, a percentage is set to determine if the columns in the DataFrame containing enough information. This percentage represents the amount of data the frame contains . The default percentage is 0.5 meaning that fifty percent or more of the data must not be empty.

A new DataFrame will be constructed by selecting only attributes in the transaction data DataFrame where the percentage of the not null data is larger than the valid data percentage. Then in this new DataFrame, the null value in the string column will be filled by 'NA'. A series of computations converts all string columns in this DataFrame to numeric features and carries over the original numeric columns. The numeric features' columns' name is same as the original attribute columns' name plus 'f_' in front. For example, a string column in transaction data DataFrame named "Report" will be converted to "f_Report" with datatype double. After multiple DataFrame transformations, it yields a DataFrame containing both attributes and numeric features. Then the numeric features, which must include "f_OIDDateYYMMDD", and HashCardNo, will be selected to form a DataFrame and this DataFrame is then written back to Hive and is stored as a Hive table as FeaturesTable. All columns' types in FeaturesTable are numeric except HashCardNo which is the same as the one in transaction data DataFrame. The HashCardNo and "f_OIDDateYYMMDD", which represents date, are selected because they are used to outer joining dimensional tables and FeaturesTable.

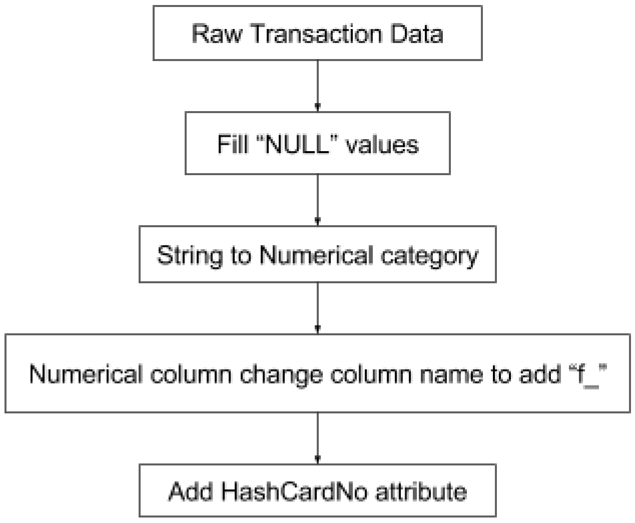
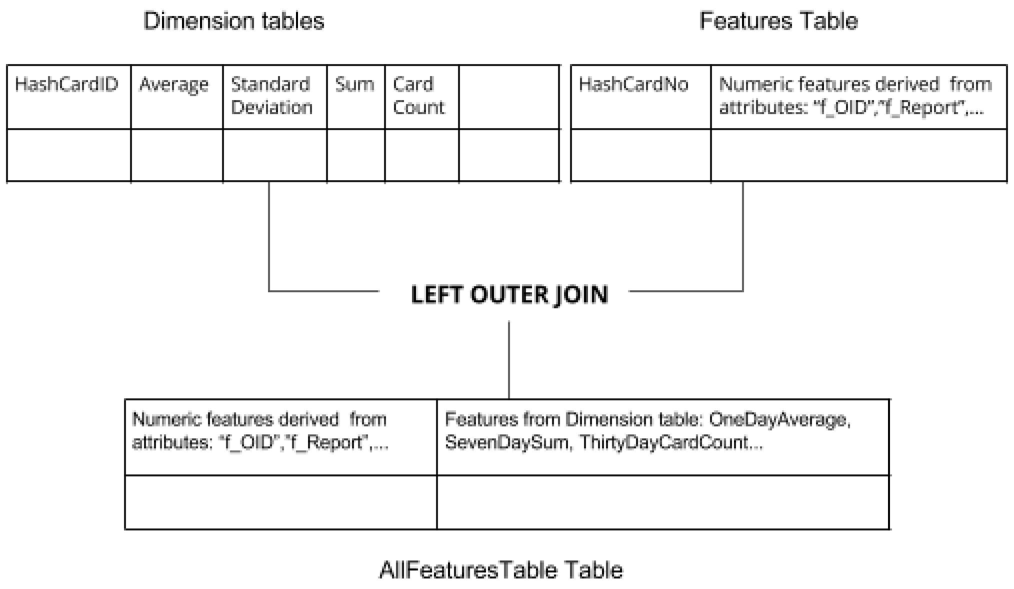
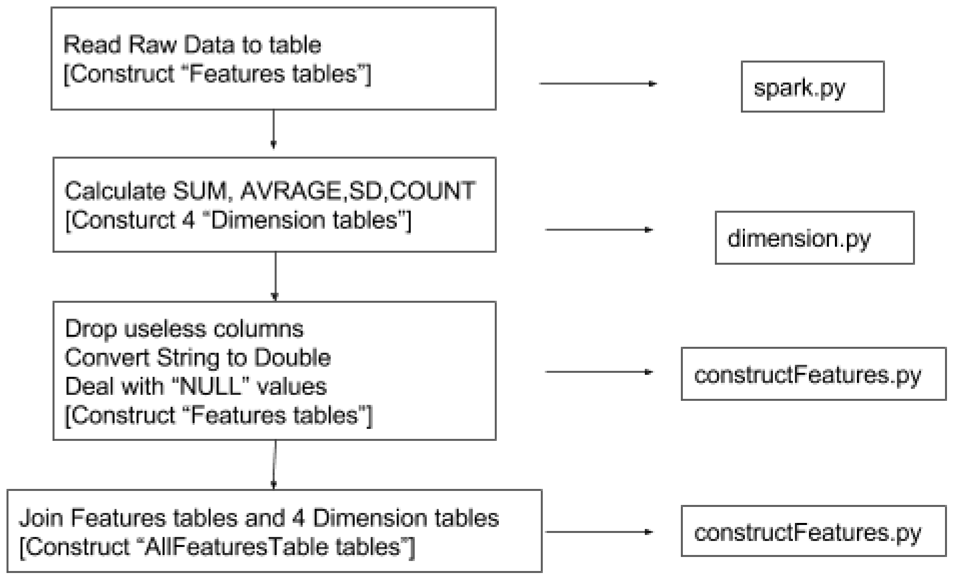

Raw Transaction Data
Fill "NULL" values
String to Numerical category
Numerical column change column name to add "f_"
Add HashCardNo attribute

Figure 5.15. Data clearing manipulation

The AllFeaturesTable is constructed by executing a left outer joining one feature table(FeaturesTable) and dimension table(DimensionOneDay, DimensionSevenDay, DimensionThirtyDay, DimensionNinetyDay) where they have equivalent hash code number and same transaction date. This table will store all the features that could be used during training. The null value in columns in the DataFrame after outer joining are filled by 0. An additional empty column "manualFraudLabel", which is used for the business analysts manual label indicating whether the transaction is fraudulent, is then appended to the AllFeaturesTable.


Dimension tables
Features Table
HashCardID
Average
Standard Deviation
Sum
Card Count
HashCardNo
Numeric features derived from attributes: "f_OID","f_Report",...
LEFT OUTER JOIN
Numeric features derived from attributes: "f_OID","f_Report",...
Features from Dimension table: OneDayAverage, SevenDaySum, ThirtyDayCardCount...
AllFeaturesTable Table
Figure 5.16. Creating AllFeaturesTable

The Figure 5.17. is the summary of preprocessing data through all calculation or formatting steps. Started from reading raw data into a table, the preprocessing calculated features, cleaned and formatted datas and gathered all information into one table. The right side are the corresponding code files.


Read Raw Data to table
[Construct "Features tables"]
spark.py
Calculate SUM, AVRAGE,SD,COUNT
[Consturct 4 "Dimension tables"]
dimension.py
Drop useless columns
Convert String to Double
Deal with "NULL" values
[Construct "Features tables"]
constructFeatures.py
Join Features tables and 4 Dimension tables
[Construct "AllFeaturesTable tables"]
constructFeatures.py
Figure 5.17. Data Preprocessing workflow

5.7 Scoring Engine

5.7.1 Iterations

The Scoring Engine initially worked only for very small datasets, but failed when using the pipeline test datasets. The error that occurred while using Spark's foreach function which loops through every row in an RDD and performs an operation on the row. The issue was caused due to a memory issue. The code required a function to run on each line of the RDD, which can quickly overload the VM's call stack, causing the error. This was an unforeseen limitation and required us to change the design of the Scoring Engine.

The Scoring Engine was rewritten to use DataFrames instead. Spark is designed to handle operations on DataFrames more efficiently, so the Scoring Engine can run with large data sets. This also allowed for better interaction with Hive, as the transaction data is retrieved in the form of a DataFrame. The general method of the Scoring Engine remained relatively the same as the original RDD design, except for a few changes. Rather than loading all of the model information first and then creating a set of predictions, the set of predictions were generated one at a time. The information needed to create the predictions for one model is collected and then used to create that model's predictions, with this process repeating for each model. This reduced the need for arrays as storage. The only data stored in an array are the accuracies and the predictions. The accuracies are stored in order to normalize them, so that for larger numbers of models in the Scoring Engine, there will always be a probability of fraud that is out of 100% (Requirement 3). The predictions are still combined, but use Spark DataFrame operations instead. Likewise, creating the probability of fraud uses DataFrame operations to perform the calculation on each row. Finally, the results are based only on the weighted voting method. The first check to see the majority vote was dropped due to the chance that multiple models with low accuracies could all incorrectly vote fraud, in which case the Scoring Engine would produce in incorrect result for that transaction.

5.7.2 Final Design

The Scoring Engine populates itself with models by calling GetModel() from the REST API. The scoring engine is currently designed to use five models at most, but could allow for more models, as discussed in section 6.1.1 (Requirement 3). If no models are available, the scoring engine terminates and logs with a system message saying that there are no models. When models are available, for each model, the needed information is collected, including the algorithm, file location and accuracy. The accuracy is stored in an array for use at a later step. Next, the model is loaded from storage. The features and data are also loaded using the current model's features list. Then, the predictions that the model makes are generated and stored as a DataFrame in a local variable array called dfpredictions where each entry corresponds to a prediction (Requirement 7). These steps are repeated for each model retrieved. Once each model has created a set of predictions, the accuracies are normalized by taking each accuracy as a fraction of the sum of all accuracies.

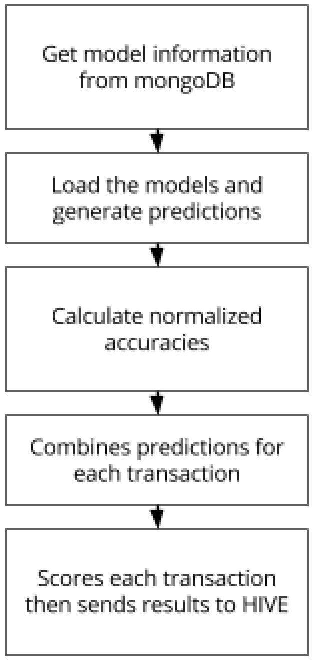

Get model information from mongoDB
Load the models and generate predictions
Calculate normalized accuracies
Combines predictions for each transaction
Scores each transaction then sends results to HIVE

Figure 5.18. Flowchart of the Scoring Engine's process.

Array	Purpose	Example	Use
ac[]	Stores the accuracy of each model	ac[0] corresponds to the accuracy for the first model taken from mongoDB	Needed for calculating the score for each model
dfpredictions[]	Stores the predictions of each model (an DataFrame containing the prediction for each transaction)	dfpredictions[0] corresponds the set of predictions for the first model	Needed for generating the score for each transactions using all models

Table 5.22. Arrays used in scoring engine.

The next step that the scoring engine takes is to group all of the predictions together. To do this, each set of predictions are joined together into a single dataframe using the transaction's order ID to validate that the corresponding predictions line up. Then, the Scoring engine calculates the probability of fraud. This is done by using a weighted voting calculation. Each prediction is multiplied by the corresponding model's accuracy and then all of the predictions are added together. This total is then subtracted from one, since "fraud" is associated with a value of zero by the models created in the Model Generator, in order to get the probability of fraud for each transaction, as a percent.

Finally, the scoring engine evaluates the transaction. For a transaction with a probability greater than or equal to 0.5, the Scoring Engine determines that the transaction is “fraud”. All other transactions are then determined to be “not fraud”. These results are added as a new column to the DataFrame and the results are. These results are then recorded in Hive.

5.8 Putting It All Together: The Chronos Pipeline Workflow

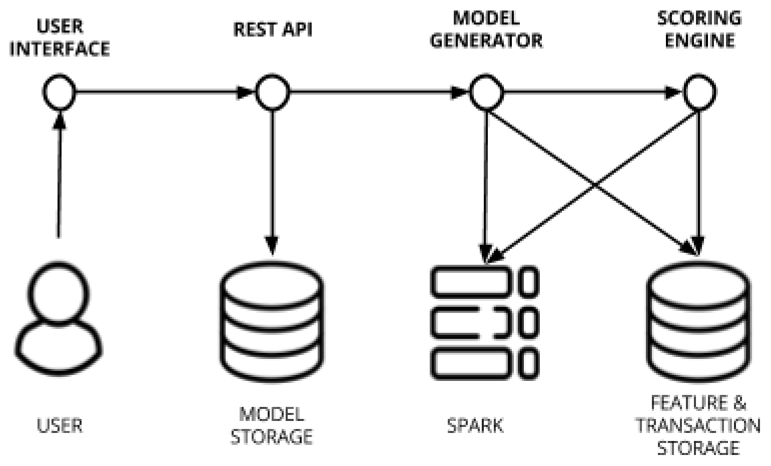


Figure 5.19. Flow of Chronos pipeline

To contextualize the above detailed technical solutions of each component, in this section we will give an overview of the workflow of the pipeline. The figure above shows the components the model passes through on the top and the technology that helps support them on the bottom. Starting at the UI, a user creates a new model. The user will specify the set of requirements for the model through a form, and then submit the model. The REST API receives the request to create this new model, and adds the model metadata to MongoDB and the model training queue. The Model Generator continuously queries the queue for a new model to be trained. The REST API finds the next model ready to be trained or retrained and returns that model to the Model Generator. Once it is the submitted model’s turn in the queue, the Model Generator uses the parameters specified by the user as input to define the data to gather for training, the features that will be used, and the algorithm and its respective tuning parameters. From here, the model will be deployed to the Scoring Engine where it will be waiting for incoming transactions to score them based on the output defined by the user (our main motivation is fraud). The user can view this model’s state in the system through the user interface and update or delete it as well as create additional models.

Our Chronos pipeline meets all of our pipeline requirements. The models generated by our pipeline have a wide variety of parameters, many of which can be altered, allowing for flexible models. Through reviews with others including a HCI expert, we created a User

Interface that is intuitive for any user with some understanding of the process of creating a model. For scalability, Hive is capable of mapping to Hbase or Cassandra, allowing for the expansion of storing data. In addition, the pipeline can be modified to allow any number of models to be used while running the Scoring Engine. This pipeline has also incorporated fault tolerance in a number of ways. The User Interface is designed to expect certain inputs and will prompt the user to make any necessary changes to the input. To avoid issues with multiple calls to the REST API and thus, the query (which is capable of breaking, see section 6.3.1), try-catch statements were added. Also, when running the Model Generator, there is a test to make sure that a model's inTraining status is updated in case the Model Generator fails. For modularity, in the event that any section, such as the UI, Scoring Engine or Model Generator need to be swapped out for something else, the REST API allows for this without affecting the other portions of the pipeline. Through research, we determined that using MongoDB and Hive were the most appropriate technologies for our pipeline and using web technologies that supported JSON like models was the best fit to increase the cohesion between components. Finally, our pipeline is capable of supporting models using our chosen algorithms, and could easily be updated to include more algorithms. A summary table of the requirements and how they were met is shown below.

Number	Label	How Requirement is met
1	Model Flexibility	● Multiple parameters for customizing models ● Ability to modify many of the model parameters later
2	Intuitive User Interaction	● Easy to use ● Multiple ways to interact with models
3	Scalable	● Hive can map to HBase and Cassandra (allowing for expansion of storing data) ● Can allow for any number of models in the Scoring Engine
4	Fault tolerance	● UI checks for valid inputs ● REST API uses try-catch statements to handle multiple calls without causing errors ● Model Generator updates inTraining in case it fails with model still inside
5	Modular	● Can replace UI, Model Generator or Scoring Engine without affecting the rest of the pipeline
6	Appropriate Technologies	● Through research, decided to use MongoDB and Hive

7	Machine Learning	● Supports models being generated using our chosen algorithms, and can be made to easily support more ○ Can create models from user input ○ Can score transactions using these models

Table 5.22. How we met the system requirements

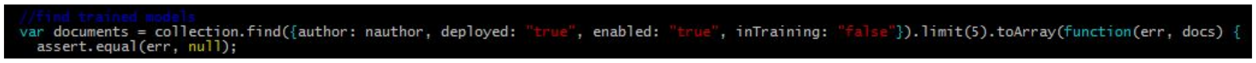
6. Evaluation

In this section, we discuss the results of our work. Overall, we were able to implement a pipeline that met all of the requirements that arose prior to and during our implementation. Next, we wanted to test our pipeline to identify shortcomings and areas for improvement. The tests varied on the component given their different responsibilities; the user interface can't be evaluated in the same way as the database. We looked at the pipeline not only at its technologies' limitations, but the pipeline's ease of use and the ability to easily extend its capabilities as those less quantitative results are just as important to the evaluation of our pipeline.

6.1 Chronos Overall

6.1.1 Assessment of the User's Ability to Complete Tasks

In our context, flexibility is defined as giving the user all the capabilities the need to accomplish a reasonable task. Currently, Chronos' biggest limitation is the limitation of the number of models deployed. In order to increase the number of models that can potentially be sent to the Scoring Engine, GetNewModel in the REST API would need to be updated. In order to do this, at line 581, the line shown in the figure below needs to be updated by changing the number in limit() to the desired number of models. Alternatively, limit() can be removed from this line to allow any number of models by one author to be used in the Scoring Engine.



```
//find trained models
var documents = collection.find({author: nauthor, deployed: "true", enabled: "true", inTraining: "false"}).limit(5).toArray(function(err, docs) {
  assert.equal(err, null);
```

Figure 6.1. The line of code that determines how many models can be sent to the Scoring Engine

6.1.2 Ease of Plugging in New Technologies

It was important throughout our development to be considerate of the fact that some technologies may need to be swapped out in the future, so we needed to consider the time associated with those tasks and the technical difficulty. The User Interface can be altered in any way as long as it is able to connect to the REST API. In order to change From MongoDB to another database, the REST API needs to be updated. At the beginning of the REST API code, the code used for setting up the connection to MongoDB would have to be replaced by the connection setup for the new database. At the end of the code, the listeners would have to be updated to connect to the new database. Finally, in each of the main functions (those described in the final design section), the code used to access the model data in MongoDB would need to be updated to access the model data in the new database. Although this switch could be time intensive, it is localized to this one part of the code because the REST API serves to abstract the database connection.

If a section of the back-end of the pipeline needed to be changed, such as changing the Model Generator or Scoring Engine, then only the connection to the REST API needs to be updated. The new code needs to import rest.py in order to use the functions from the REST API. In the case where these functions need to produce a different output, such as expecting only one model in the new Scoring Engine, then the corresponding functions need to be updated as well. In these cases, the corresponding functions need to update the line using collection.find() to use or not use the limit() argument (see Figure 6.1 above for details). Our REST API really lends itself well to ensuring that we don't overly couple our components.

6.2 Chronos User Interface

The User Interface was evaluated on two broad areas: the core principles of human computer interaction and its fault tolerance. For the latter, the fault tolerance was more focused on the grand scheme of the pipeline as opposed to human computer interaction and alerting the user of any errors. There, we evaluate the potential for the user interface to submit invalid information that could negatively impact the pipeline.

6.2.1 Evaluated against Human Computer Interaction Principles

The final User Interface was analyzed against these ten core principles of Human Computer Interaction (Nielsen, 1995; Nielsen, 1994). We listed both the benefits and drawbacks to ensure the analysis was fair and unbiased. This analysis also considered the feedback from the Human Computer Interaction expert we met with, Professor David Brown.

Principle	Benefits	Drawbacks
Visibility of system status	● Queue shows order of models to be trained ● Model shows in training, enabled, deployed	● No comprehensive list of what is deployed/enabled
Match between system and the real world	● After refinement, terminology matches real world vernacular	
User control and freedom	● User is able to customize models within reason	● User is pretty confined to defined model creation procedure
Consistency and standards	● Same words, layout, terminology used throughout	● Error handling is a bit inconsistent

Error prevention	● Error handling for reasonable use cases such as user inputting wrong type or empty data	● Error messages aren't immediately visible (usually until form is submitted)
Recognition rather than recall	● Views all visible from every page	● Have to go to model detail to delete
Flexibility and efficiency of use	● Input is limited by system availability (algorithms, output, features)	● Creating a model only asks for required information
Aesthetic and minimalist design	● No extraneous information	● Detailed model view is a bit busy
Help users recognize, diagnose, and recover from errors	● Error messages appear following a submitted error	● Error messages could appear earlier

Table 6.1. Core principles of a UI

6.2.2 Examining Fault Tolerance through Sanitizing Bad User Input

When considering a User Interface, the fault tolerance is embodied by the user's ability to input invalid information. When the user has too much control, they are able to submit input that will not result in valid output at best or could cause the pipeline to break at worst. Below is a table listing every interaction between the user and the models and an analysis of the flexibility versus the input's ability to prevent invalid inputs.

	Flexibility	**Avoiding Errors**
Field: Model Name	● Input type is text (any text)	● Rejects empty box
Field: Author Name	● Input type is text (any text)	● Rejects empty box
Field: Algorithm	● Limited to algorithms available	● Dropdown box limits input
Field: Output	● Limited to output available (currently only fraud)	● Dropdown box limits input

Field: Features	● Choose as few or as many features as desire	● Drag and drop limits input
Field: Data Date Range	● User has choice over the amount of time, but is somewhat limited	● Input limited to number input and drop down
Field: Train Frequency	● User has choice over the amount of time, but is somewhat limited	● Input limited to number input and drop down
Submit (Train) Button		● Only submits when input fields are valid
Edit Author	● Any text	● Limited to text field (to add: not empty)
Save Edited Model	● Any text	● Need to add: input validation on edit author (to add: not empty)
Delete Model	● Delete model when in detail view, delete any model	● Handles deletion using ID

Table 6.2. Analysis of interactions between user and models.

The interface fields give the user the flexibility to enter a range of information while ensuring the information is valid to avoid putting the pipeline in an error state. By including the input validation so early on the pipeline, we avoid more costly mistakes and give the user more time to correct their mistakes.

6.3 REST API

6.3.1 Assessing Fault Tolerance through Input Sanitization

The first set of tests involved making sure that every function worked as expected. This included testing the handling of bad inputs. If a required field is missing, the field is filled in with null. For other inputs, there are no errors. However, the REST API does not evaluate the parameters, so the API assumes that the parameters have been entered in correctly and will proceed as usual. The expectation is that the other parts of the pipeline, such as the UI, are responsible for evaluating the input parameters.

6.3.2 Behavior when Handling Simultaneous Requests

This API is capable of running multiple requests at once. The test for this included four calls to the REST API to create models, where each call was made immediately after the previous call. While running the test for the first time, the test failed, meaning that an error occurred. The error was due to having multiple requests try to load and save the queue at the same time. For the second test, try-catch arguments were added to the functions pertaining to the queue. The API successfully created all four models, as well as added the author's name to authors.txt and added the model to the queue.

6.3.2 Speed Testing Results

The next set of tests involved testing the speed of the functions in the REST API that are called from other sources. This test involved taking the average running time of three instances of each function. For all of the functions, there was only one model in the database. The exception was CreateModel(), in which the database was empty while testing. The results are shown in the table below.

CreateModel		**GetNewModel**	
Speed	49.33 milliseconds	Speed	6.33 milliseconds
UpdateModel		**GetAll**	
Speed	10.66 milliseconds	Speed	5.66 milliseconds
UpdateModelInfo		**ModelStatus**	
Speed	12.66 milliseconds	Speed	3.66 milliseconds
DeleteModel		**UpdateTraining**	
Speed	11.66 milliseconds	Speed	7.66 milliseconds
GetModel		**GetQueue**	
Speed	4.66 milliseconds	Speed	1.66 milliseconds

Table 6.3. Result of testing the speed of the functions in the REST API

6.4 Model Generator

Recording a Baseline and the Impact of Different Algorithms and their Parameters on Training Times

To see the performance of Model Generator, models with a variety of algorithms were constructed and random but reasonable parameters settings were chosen to do test them on their speed and accuracy. Fitting different data sets also affected model performance so in the following scenario and with 20 plus features, models were set up:

Algorithm	Data set size	Attributes
Decision Tree	3864 data (4 years +1 months date range)	
Random Forest(1)	3864 data (4 years +1 months date range)	numOfTree: 7
Artificial Neural Network(1)	3864 data (4 years +1 months date range)	Max Iteration: 2 Block size: 128 Num of Hidden layer: 2 Hidden layer 1 node: 47 Hidden layer 1 node: 27
Random Forest(2)	3864 data (4 years +1 months date range)	numOfTree: 37
Artificial Neural Network(2)	3864 data (4 years +1 months date range)	Max Iteration: 2 Block size: 128 Num of Hidden layer: 1 Hidden layer 1 node: 100

Table 6.4. Models used for testing

In first set of models were created to fit whole sample dataset with four years and one month data range. Decision Tree, Random Forest and Artificial Neural Network were all the list. Because Decision Tree algorithm did have parameter to alter, only one model for this algorithm is enough. Random Forest and Artificial Neural Network each had two models with different parameters. For Random Forest, one of numOfTree variable is smaller than features number selected by the user and one variable is greater than features number selected. For Artificial Neural Network, one is with 2 hidden layers using random prime numbers as layer nodes and the other one is 1 layer using 100 nodes. We are expecting different results from Neural Network with different layers and different nodes numbers.

Algorithm	Data set size	Attributes
Decision Tree	220 data(5 months data)	
Random Forest(1)	220 data(5 months data)	numOfTree: 7
Artificial Neural Network(1)	220 data(5 months data)	Max Iteration: 5

		Block size: 128 Num of Hidden layer: 2 Hidden layer 1 node: 47 Hidden layer 1 node: 27
Random Forest(2)	220 data(5 months data)	numOfTree: 2
Artificial Neural Network(2)	220 data(5 months data)	Max Iteration: 5 Block size: 128 Num of Hidden layer: 1 Hidden layer 1 node: 47

Table 6.5. Second set of models used for testing

In second set of models were created to fit whole sample dataset with five months data range. Decision Tree still maintained same setting. One of Random Forest kept old tree number. Another Random Forest model shrunk to smaller number to see the accuracy performance. For Artificial Neural Network, Max iteration increased to get better accuracy. One layer Neural Network reduced the 47 nodes was expected to see the reduce impact on accuracy.

The accuracy and speed of testing result in the first test cases are following:

Algorithm	Train Duration	Accuracy
Decision Tree	2 minutes 55 seconds	0.941558
Random Forest(1)	2 minutes 24 seconds	0.943548
Artificial Neural Network(1)	1 minutes 57 seconds	0.946138
Random Forest(2)	2 minutes 23 seconds	0.931848
Artificial Neural Network(2)	2 minutes 13 seconds	0.939471

Table 6.6. Test results of first test set

The accuracy and speed of testing result in the second test cases are following:

Algorithm	Train Duration	Accuracy
Decision Tree	2 minutes 04 seconds	0.988636
Random Forest(1)	2 minutes 05 seconds	0.980392
Artificial Neural Network(1)	2 minutes 03 seconds	0.987314

Random Forest(2)	2 minutes 03 seconds	0.978261
Artificial Neural Network(2)	2 minutes 26 seconds	0.990825

Table 6.7. Test results of second test set

In first table, train durations were overall longer than the train duration in second table. While datasize increased more than 17 times, the pipeline train time only increased about 30 seconds. The second test set was small so accuracy was higher but may overfit. There were not much difference between parameter settings and even three algorithms. The accuracy was greatly affected by datasize. However, we still can see Artificial Neural Neural Network perform a little better than Random Forest generally.

6.5 Chronos Backend Infrastructure

The Chronos Backend Infrastructure is built on Hadoop and Spark. Hadoop performance is hard to test but can be partially reflected from the performance of Model Generator and Feature and Transaction Storage. However, Spark performance is more easily testable is our implementation. We wanted to evaluate its performance to see its accuracy, speed, and fault tolerance in the context of our pipeline. Below is our list of tests:

1. Test date range calculation correctness using testDate.py
2. Test efficiency of calculating using Dimension.py
3. Test efficiency of converting categorical string to double using constructFeature.py
4. Test efficiency of constructing whole table using constructFeature.py
5. Test cluster node go down
6. Test the amount of times spark will try to rerun a model on failure
7. Test incorrectly formatted data
8. Test number of requests before resources are used up

6.5.1 Accuracy of Chronos Backend Infrastructure Calculations

1.Test date range calculation correctness using testDate.py

Our pipeline uses date ranges to gather a subset of data, so this test explores the accuracy of the date ranges it actually gathers. Date.py contains all functions calculating data ranges for computing features and selecting customized dataset. The functions includes calculating the date of previous N days or after N days. The test cases used 5 dates with different data ranges involving spanning days, months and years for each function.

The data for the test cases is below:
Start date of 160527, spanning 10 days
Start date of 160527, spanning 30 days (span a month)

Start date of 160527, spanning 100 days (span multiple months)
Start date of 160527, spanning 300 days (span a year and multiple months)
Start date of 160527, spanning 1400 days(span multiple year and multiple months)

All of the test results of each function are correct and they are the same with manually calculated results.

6.5.2 Speed of Chronos Backend Infrastructure Calculations and Processing

2. Test: efficiency of calculating using dimension.py

Dimension.py calculates all the features from the transaction table with various date ranges. The program produces 4 tables: one day feature, seven day feature, thirty day feature and ninety day feature.

When first running dimension.py, all tables need to be constructed. But after running for the first time, it is available more quickly for later runs. The old tables are saved and new features are inserted into old tables which saves running time. The first run requires about 44 minutes to construct all tables, even though there is only 220 transactions data in the table. The following runs could be as quick as 1 minute for inserting new calculations. If running all data in the sample data set, it still takes about 1 minute and 22 seconds to process based on the old table.

3. Test efficiency of converting categorical values using constructFeature.py

Converting categorical string to numerical values (double) is an important step for data cleaning. The data sent to Model Generator must be a double. Converting speed heavily is correlated with the size of dataset.

Data set size	Time to construct table
220 data	32 minutes
640 data	41 minutes and 52 seconds
1927 data	55 minutes and 13 seconds
3846 data	1 hour 4 minutes and 10 seconds

Table 6.8. Conversion efficiency test with different data size

4.Test efficiency of integrate all features and attributes use constructFeature.py, join_all_features function

Our pipeline constructs a feature table which isu used for training. Below are the times it took to construct that table with varying dataset sizes.

Data set size	Time to construct table
220 data	1 minutes 13 seconds
640 data	1 minutes 20 seconds
1927 data	1 minutes 29 seconds
3846 data	1 minutes 42 seconds

Table 6.9. Join efficiency test with different data size

The select and join performance for this final features table isn't linearly related to the data set size, though it the time does increase negligibly with dataset size. The average speed for the join_all_features function, no matter the size of data set, is about 1 minute and 30 seconds. There is pretty uniform and stable result for a function.

6.5.3 Fault Tolerance

5. Testing Cluster Behavior when a Node Goes Down

In the cluster, there is only one node running. If the node goes down, the cluster is unable to be used. Hadoop and Spark need to be restarted. The specific command used for restarting hadoop and spark is in User Guide. Given the fault tolerance of Spark, a cluster with more than one node would not face this shortcoming as the tasks would be rerun on a different node.

6. Test the amount of times spark will try to rerun a model on failure

To ensure a constantly failing model doesn't monopolize the system, there is the RERUNMAX variable to determine how many times a model that failed to be trained can try to train again. Currently, the default RERUNMAX is set to retry 3 times. One chance for formal run and three chances are for retrials. This can be changed to other numbers. If model deployment is unsuccessful, the spark-submission command will rerun until model is deployed or reaches RERUNMAX. The tests go through from changing RERUNMAX from 0 to 5. The running result is the same as the expectation:

RERUNMAX	Model deployment success Total run needed	Model deployment fail Total run needed
0	1	1 (no retry chance)
1	<= 2	2 (1 retry chance)
2	<=3	3 (2 retry chance)

3	<=4	4 (3 retry chance)
4	<=5	5 (4 retry chance)
5	<=6	6 (5 retry chance)

Table 6.10. Testing the maximum number of times a model can attempt to be deployed

7. Testing Incorrectly Formatted Data

The pre-processing of the data is done before starting the Model Generator. The incorrectly formatted data would be sent as "NULL" when reading into the Transaction table. When running construct Feature.py to clear and format the dataset, "NULL" columns would be automatically dropped. This is not going to affect overall model performance. Due to this check, there is no worry of encountering incorrectly formatted data.

8. Test number of requests before resources are used up

With the current configuration, the size of the tmp directory limits how many spark submissions can be processed. The average is about 24 spark submission requests that could be done. At the 25th try, YARN can no longer find the resources for Spark and loops at the "Accept" status. When encountering this situation, YARN needs to restart. The specific command used for restart YARN is in User Guide.

6.6 Feature and Transaction Storage

These three tests of the Hive database evaluate the efficiency, storage required and the ease of plugging in a new database technology. The speed test was conducted by running a script that moves data to a hive table. The storage used by Hive and the number of tables in Hive is calculated and checked on the HDFS. Easy plugin and plugout with Hive were also analysed in this section.

6.6.1 Speed of Feature and Transaction Storage

The team ran spark.py to test the speed of reading data to a table. This program only involved reading data from a csv file to a Hive table with 495 fixed columns. Started from submitting spark application, the whole process, including launch task executors and cleaning block memories, took about 1 minutes to finish. The dataset is not big, with 3846 data points and 495 features.

6.6.2 Storage Space Required

There are eight tables right now on HDFS in total size of 10.8 MB. Each table with average of about 1 MB size. Because the team used a sample dataset of 3846 with 495 features, this space usage is reasonable. The space in HDFS still has about 23TB free space.

The data types of transaction data are defined to form a table. If data type are wrong when coming into table, the whole column will be seen as “NULL” and will be dropped in the data clearing step.

6.6.3 Ease of Replacement

Hive can be easily replaced by any other HDFS based database. Because Mongodb is not running on HDFS, Mongodb cannot take advantage of data and file distribution. Hive can map to Hbase so Hbase wide column tables can be construct by Hive. HBaseStorageHandler is the tool to register Hbase tables to Hive metastore and to do the column mapping. This tool also allows Hive to use HQL(Hive query language) to access and manipulate data in Hbase. Cassandra handler also released for Hive and maintained by Datastax. Its function is similar to HBaseStorage for map from Hive.

6.7 Scoring Engine

6.7.1 Speed of Scoring Engine

Testing the speed of the scoring engine was done in two ways. The first set of tests involved evaluating the speed of the scoring engine with only one model. A model was sent to the scoring engine three times in order to get an average running time. This was done for the ANN, Random Forest, and Decision Tree algorithms. The second test involved a similar process, except that in this test, five models were used instead of one to determine the performance of the Scoring Engine with multiple models. These models were two Decision Tree models, two ANN models and one Random Forest model. In both tests, only the code created for the Scoring Engine was evaluated. While running the code involved extra time for starting up Spark and saving the results table to Hive, these tests are already covered in the previous results sections. For more information about the models and data, see Table 6.6. The results are shown in the table below. These tests suggests that each algorithm requires about the same amount of running time and, as suggested by the five model test, do not greatly affect the overall runtime when more models are added.

Decision Tree

Speed	43.81 seconds
Random Forest	
Speed	40.70 seconds
ANN	
Speed	39.37 seconds
Five Models	
Speed	69.26 seconds

Table 6.11. Result of testing the speed of the Scoring Engine on individual models and a set of five models

6.7.2 Scalability Assessment through Speed Tests

In order to test the scalability of the Scoring Engine, the average speed of each model was determined for a four month dataset and a four year dataset. For each test, the model was used for scoring three times, and then the average speed was calculated. This scalability test was also applied to a set of five models. As with the previous test involving five models, the models chosen were two Decision Tree models, two ANN models and one Random Forest model. Just like in the previous test, only the code specifically created for the Scoring Engine was evaluated. For more information about the models and data, see Tables 6.6 and 6.7. The results are shown in the table below. While there is a noticeable increase in time, considering the difference between the number of transactions in each set, this is not that large of an increase in running time.

Decision Tree	
Speed (5 Month Set)	43.81 seconds
Speed (4 Year + 1 Month Set)	78.45 seconds
Random Forest	
Speed (5 Month Set)	40.70 seconds
Speed (4 Year + 1 Month Set)	76.27 Seconds
ANN	
Speed (5 Month Set)	39.37 seconds

Speed (4 Year + 1 Month Set)	74.13 seconds
Five Models	
Speed (5 Month Set)	69.26 seconds
Speed (4 Year + 1 Month Set)	193.67 seconds

Table 6.12. Result of testing the speed of the Scoring Engine on a four month dataset and a four year dataset.

7. Conclusions and Future Work

7.1 Conclusions

The goal of this project was to build a pipeline that would automate the process of training, storing, and deploying machine learning models while still giving the tuning power to the user specifying the model's parameter. The pipeline created for this project meets all of the criteria. A user can connect to the UI and design a model using a wide set of parameters, including the tuning parameters for their specific model algorithm. The models created or automatically trained, retrained and scored by the pipeline.

This UI allows the user to have significant control over the process. As already mentioned, there are a wide selection of parameters that the user can select in order to better control the creation and deployment of the model, such as the algorithm used, the amount of data to train on, and how frequent retraining should be. In addition, the user is provided control over a model's enabled status and deployed status. A user can even easily delete the model if they do not want it anymore. The UI also shows users the statistics of trained models as well as the training status of the model. This flexibility allows for ease of use, which will be appealing to potential users.

The Model Generator is capable of efficiently training models. The CronJob that continuously calls the Model Generator was designed to handle Spark crashing, in order to make the code more fault tolerant. The only issue is that the models seem to have to high of an accuracy. No matter how the models were modified, including changing the number of features or tuning parameters, the models always had an accuracy above 90%. This could possibly be because the test data was mostly fraud rather than closer to a fifty-fifty mix of fraud and not fraud transactions.

The Scoring Engine is capable of efficiently scoring transactions using multiple models. The Scoring Engine runs on the same CronJob as the Model Generator, and will not affect the pipeline in case it crashes. Currently, the scoring engine uses a weighted voting system for scoring each transaction, where the predictions for each transaction are multiplied by the corresponding model's accuracy and then added together. The accuracies are normalized, which is a potential issue for scoring using one model, since this would mean that the model is treated as being 100% accurate. This might possibly be undesired if the pipeline is intended to be used by only having one model at a time.

Overall, this pipeline will be a useful tool for ACI. Due to the constant need of models for detecting fraud, as well as the need to consistently update the models, this pipeline will help by automating the process. This will allow for the focus to shift away from always working on the next version of the model and just allowing this pipeline to handle the job of detection of fraud. In conclusion, this pipeline served as an adequate proof of concept and met all of its requirements.

7.2 What We Learned

Throughout the duration of this project, we have learned lessons we hope to share on best practices we have identified for pipeline development. Regarding the actual process of development, we found our pair-per-component development to be incredibly valuable. This model assigns each component a main programmer and a support, so that no one monopolizes the programming or knowledge of a single component. Later on as testing and debugging became more prevalent, having a second expert on any given component ensured that no one person would be blocked.

In terms of development, the biggest takeaway is the amount of planning that we should have spent more time on when considering what our system would look like. We kept our requirements colloquial and never formerly defined them, which made it difficult to develop a pipeline off of a conversation. Had we formalized our meetings and requirements gathering, our end goal would have become more apparent earlier on. Additionally, when building a pipeline like this there is a certain amount of technical preparations that need to be made for machines to support the operations. We were unaware of this and in turn the size of our hadoop cluster was hindered and we met occasional hiccups due to lack of storage. In addition to technical planning, planning by ensuring we had a comprehensive understanding of the material would have helped us avoid some incorrect assumptions or picked more appropriate technologies.

Any project considering using a variety of technologies needs to prioritize not only the technologies' capabilities but also their ease of integration with other existing parts of the system. Initially, we considered technologies solely because of how they added up on a advantages and disadvantages list and didn't consider the technical requirement to implement them into our system. For example, the MongoDB / JSON Objects / JavaScript pipeline felt very intuitive and even though some things felt easier to do with other technologies, the implementation time ended up making them less appealing options.

7.3 Future Work

When running the pipeline, we mainly used only one of our virtual machines. We were unable to get the fully distributed hadoop cluster setup due to various issues with the virtual machines, including some of the virtual machines crashing. As a result, we never got to test using Spark on a true distributed cluster. Therefore, we recommend that the pipeline gets tested on a fully distributed hadoop cluster in order to test the performance of the pipeline. We believe that using such hadoop cluster will improve the performance and efficiency of the pipeline, as Spark has better performance on such cluster. We also recommend switching the pipeline to work for streaming transactions instead, as this is the most likely format ACI Worldwide would be dealing with the data.

As for individual sections of the pipeline, we have some recommendations for them as well. For the UI, we recommend that more machine learning classification algorithms and their corresponding tuning parameters get added to the pipeline. This will allow for the user to have a larger variety of algorithms to choose from. To make these changes, the REST API, Model Generator and Scoring Engine will need to be updated in addition to the UI. For the Model Generator, we recommend conducting tests on how to train the models in order to get an appropriate accuracy. Our tests data contains mostly fraudulent transactions, and it is possible that there is not enough information to accurately predict the "not fraud" cases which are less common. Due to the fact that fraud is rare, it is possible that the current pipeline would only detect cases where the transactions are not fraud. The tests need to make sure that the models are capable of detecting both cases, rather than just fraud or not fraud. Finally, for the Scoring Engine, we recommend making a determination in how you would like the pipeline to work. If the intention is for multiple models to be used in the pipeline, than the current system works. However, allowing only one model to be in the Scoring Engine at a time will result in the accuracy of that model being set to 100%. In this case, the code for normalizing the accuracies would need to be modified.

Citations

ABOUT US | FICO®. (2017, February 12). Retrieved February 26, 2017, from http://www.fico.com/en/about-us#at_glance

ACI Worldwide. About ACI. Retrieved February 26, 2017, from https://www.aciworldwide.com/about-aci

Apache. Classification and regression. Retrieved February 26, 2017, from http://spark.apache.org/docs/latest/ml-classification-regression.html

Apache. Data Types - RDD-Based API. Retrieved February 26, 2017, from http://spark.apache.org/docs/latest/mllib-data-types.html

Apache. Evaluation Metrics - RDD-based API. Retrieved February 26, 2017, from http://spark.apache.org/docs/latest/mllib-evaluation-metrics.html

Apache. Machine Learning Library (MLlib) Guide. Retrieved February 26, 2017, from http://spark.apache.org/docs/latest/ml-guide.html

Apache. Spark SQL, DataFrames and Datasets Guide. Retrieved February 26, 2017, from http://spark.apache.org/docs/latest/mllib-data-types.html

Apache. (2009, September 20). TaskTracker. Retrieved February 26, 2017, from https://wiki.apache.org/hadoop/TaskTracker

Apache. (2010, June 30). JobTracker. Retrieved February 26, 2017, from https://wiki.apache.org/hadoop/JobTracker

Apache. (2016, April 20). Apex Incubation Status. Retrieved February 26, 2017, from http://incubator.apache.org/projects/apex.html

Apache. (2015a). Apache Storm. Retrieved February 26, 2017, from http://storm.apache.org/

Apache. (2015b, March 19). 5 reasons why Spark Streaming's batch processing of data streams is not stream processing -. Retrieved February 26, 2017, from http://sqlstream.com/2015/03/5-reasons-why-spark-streamings-batch-processing-of-data-streams-is-not-stream-processing/

Apache. (2017a). Apache Apex Open Source Stream & Batch Processing Platform. Retrieved February 26, 2017, from https://www.datatorrent.com/products-services/apache-apex/

Apache. (2017b, January 03). Home - Apache Hive - Apache Software Foundation. Retrieved February 26, 2017, from https://cwiki.apache.org/confluence/display/Hive/Home

Apache. (2017c). Apache Hive. Retrieved February 26, 2017, from https://hortonworks.com/apache/hive/
Apache. (2017d, February 17). Apache Hbase. Retrieved February 26, 2017, from https://hbase.apache.org/

Apache. (2017e, January 26). Apache Hadoop. Retrieved February 26, 2017, from http://hadoop.apache.org/

Apache. (2017f, January 20). Apache YARN. Retrieved February 26, 2017, from https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html

Apache.(2017g) . Apache Zookeeper. Retrieved February 26, 2017, from https://zookeeper.apache.org/

Bandugula, N. (2015, June 23). The 5-Minute Guide to Understanding the Significance of Apache Spark | MapR. Retrieved February 26, 2017, from https://www.mapr.com/blog/5-minute-guide-understanding-significance-apache-spark

Bharadwaj, S. (2016, March 28). What is the difference between Apache Spark and Apache Hadoop (Map-Reduce) ? Retrieved February 26, 2017, from https://www.quora.com/What-is-the-difference-between-Apache-Spark-and-Apache-Hadoop-Map-Reduce

Burger, J. A Basic Introduction to Neural Networks. Retrieved February 26, 2017, from http://pages.cs.wisc.edu/~bolo/shipyard/neural/local.html

Fico. (2017, February 27). FICO® Falcon® Fraud Manager | FICO®. Retrieved February 28, 2017, from http://www.fico.com/en/products/fico-falcon-fraud-manager

Forter. (2017). Forter | Accurate Fraud Protection & Detection for E-Commerce Sites. Retrieved February 28, 2017, from https://www.forter.com/

Holmes, T. E. (2016, February 02). Credit card fraud and ID theft statistics. Retrieved February 26, 2017, from http://www.creditcards.com/credit-card-news/credit-card-security-id-theft-fraud-statistics-1276.php

Javelin Strategy. (2015, May 3). $16 Billion Stolen from 12.7 Million Identity Fraud Victims in 2014, According to Javelin Strategy & Research [Press release]. Retrieved February 26, 2017, from https://www.javelinstrategy.com/press-release/16-billion-stolen-127-million-identity-fraud-victims-2014-according-javelin-strategy

Liaw, A., & Wiener, M. (2002). Classification and regression by randomForest. R news, 2(3), 18-22. Retrieved February 26, 2017, from http://ai2-s2-pdfs.s3.amazonaws.com/6e63/3b41d93051375ef9135102d54fa097dc8cf8.pdf

Mizes, H. (2013, December 3). Card Fraud. Retrieved February 26, 2017, from http://www.cs.rochester.edu/~kshen/csc296-fall2013/lectures/Mizes_CardFraud.pdf

Mitchell, T. M. (1999). Machine learning and data mining. Communications of the ACM, 42(11), 30-36.

MongoDB. (2017a). Introduction to MongoDB. Retrieved February 26, 2017, from https://docs.mongodb.com/manual/introduction/

MongoDB. (2017b). MongoDB CRUD Tutorial. Retrieved February 26, 2017, from https://docs.mongodb.com/v3.0/applications/crud/

MongoDB. (2017c). Metadata and Asset Management. Retrieved February 26, 2017, from https://docs.mongodb.com/ecosystem/use-cases/metadata-and-asset-management/

Nielsen, J. (1995, January 1). 10 Heuristics for User Interface Design: Article by Jakob Nielsen. Retrieved February 27, 2017, from https://www.nngroup.com/articles/ten-usability-heuristics/

Nielsen, J. (1994, April). Enhancing the explanatory power of usability heuristics. In Proceedings of the SIGCHI conference on Human Factors in Computing Systems (pp. 152-158). ACM.

Palmer, K. (2013, July 10). How Credit Card Companies Spot Fraud Before You Do. Retrieved February 26, 2017, from http://money.usnews.com/money/personal-finance/articles/2013/07/10/how-credit-card-companies-spot-fraud-before-you-do

Patidar, R., & Sharma, L. (2011). Credit card fraud detection using neural network. International Journal of Soft Computing and Engineering (IJSCE), 1(32-38).

Pradhan, M., Pradhan, S. K., & Sahu, S. K. (2012). Anomaly detection using artificial neural network. International Journal of Engineering Sciences & Emerging Technologies, 2(1), 29-36.

PYMNTS. (2015, August 05). Global Card Fraud Damages Reach $16B. Retrieved February 26, 2017, from http://www.pymnts.com/news/2015/global-card-fraud-damages-reach-16b/

Ramanathan, V. (2014, December 02). Venkatesh Ramanathan, Paypal - Fraud Detection Using H2O's Deep Learning. Retrieved February 26, 2017, from https://www.youtube.com/watch?v=RqkheMI3Ciw

Rouse, M., Hannan, E., Wilson, S. (2016, December 08). RESTful API. Retrieved February 26, 2017, from http://searchcloudstorage.techtarget.com/definition/RESTful-API

Sayad, Dr. Saed (2017). Artificial Neural Networks. Retrieved February 26, 2017, from http://chem-eng.utoronto.ca/~datamining/dmc/artificial_neural_network.htm

Skytree (2017). Skytree REST API. Retrieved February 26, 2017, from http://pages.skytree.net/rs/855-VWM-226/images/SkytreeRESTAPI.pdf

Tan, P., Steinbach, M., & Kumar, V. (2015). Introduction to Data Mining. Dorling Kindersley: Pearson.

Taneja, R. (2015, May 28). Trend to Watch Closely: Data and (Deep) Machine Learning. Retrieved February 26, 2017, from https://www.linkedin.com/pulse/trend-watch-closely-data-deep-machine-learning-rajat-taneja

Warin, F., Diamant, M., & Vretsona, O. (2013, August). How to use company data efficiently to detect fraud and corruption. Retrieved February 26, 2017, from https://www.financierworldwide.com/how-to-use-company-data-efficiently-to-detect-fraud-and-corruption

Warnick, M. (2016, February 02). 7 reasons your credit card gets blocked and 7 tips for handling it. Retrieved February 26, 2017, from http://www.creditcards.com/credit-card-news/7-reasons-credit-card-blocked-tips-for-handling-1282.php

Zitter, L. (2016, August 24). How Credit Card Companies Fight Fraud. Retrieved February 26, 2017, from http://www.investopedia.com/articles/personal-finance/080416/how-credit-card-companies-fight-fraud.asp

Image Credits
By the Eyecons. N.d. Data base, data storage, server, storage icon [Online image]. Retrieved from https://www.iconfinder.com/icons/1370034/data_base_data_storage_server_storage_icon#size=128

Webalys. n.d. Database, streamline icon [Online image]. Retrieved from https://www.iconfinder.com/icons/185097/database_streamline_icon#size=128

Atlas. n.d. Account, man, person, profile, user icon [Online image]. Retrieved from https://www.iconfinder.com/icons/1540176/account_man_person_profile_user_icon#size=128

Appendix

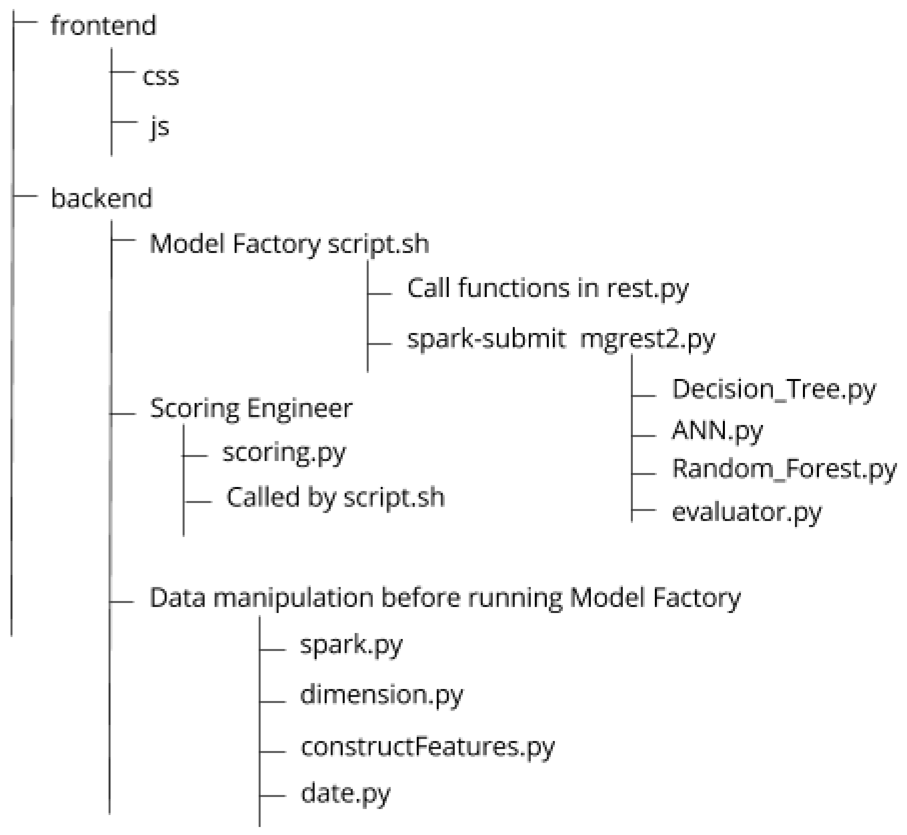
Appendix A: User Guide

Pipeline

Software Requirements

- Node
- Dragula
- Express
- Bootstrap
- JQuery
- Spark Version 2.1.0
- Hadoop Version 2.7.3
- Hive

File Structure



Order of Initialization

1. MongoDB
2. REST API
3. User interface
4. Hadoop
5. Spark
6. Model Generator

Initializing Hadoop and Spark(Step 4 and 5) can be done before Front end initialization, but initializing Model Generator must be after REST API running. Make sure REST API is running before running Model Generator

REST API

Navigate to the folder containing restserver.js

```
node restserver.js &
```

User Interface

Navigate to the folder containing server-express.js

```
node server-express.js &
```

Chronos Backend Infrastructure

Hadoop

Navigate to hadoop/sbin folder containing shell command

```
./start-all.sh
```

This command runs both dfs and yarn

For stop Hadoop service, go to the same folder and run

```
./stop-all.sh
```

For stop Yarn service, go to the same folder and run

```
./stop-yarn.sh
```

For start Yarn service, go to the same folder and run

```
./start-yarn.sh
```

Spark

Navigate to spark/sbin folder containing shell command

```
./start-all.sh
```

For stop spark service, go to the same folder and run

```
./stop-all.sh
```

Hadoop and Spark Configuration

HDFS is constructed through the port 127.0.0.1:9001 with data duplication of 1, which means all data will be copy once. Namenode is distributed under directory /mqp/hadoopinfra/namenode. Datanode is distributed under directory /mqp/hadoopinfra/datanode. As we are using only one node, the dfs.namenode.handler.count and dfs.datanode.handler.count are all set as default.

Yarn is the resource manager, tracking down the namenode and datanode activities. Specific configuration of Yarn is in ~/hadoop/etc/hadoop/yarn-site.xml. All values for each property in Yarn is calculated by yarn-tuning-guide.xlsx from cloudera, using one worker hosts as cluster size. Specifying Worker Host Configuration, Worker Host Planning and Cluster Size.

Mapreduce configuration are set as default, because the calculated mapreduce results in the document met the lowest requirements.

In our setting, Spark connected to Yarn, which ran on the top of hadoop. In client mode, Spark runs driver and submits application to Yarn, Yarn resource manager launches Spark application master. Spark application master and Spark driver request resources from Yarn. If there are resources available, Spark application master will launch container via Yarn NodeManager which launches Spark Executors. After Spark Executors launched, Spark driver will be register with Executors and launch tasks for the Executors. All Yarn containers runs on HDFS.

Model Generator

Navigate to ModelGenerator/refractory folder under /mqp

To start model generator run

```
crontab -e
```

This will open up the script of cronjob
setup time interval for running ./script.sh in cronjob

Right now it is `*/1 * * * * ./script.sh >> logAddress(optional) 2>&1`, which means it runs every minute and save the running log to ModelGenerator directory.

To close the Model Generator

```
crontab -e
```

Comment out the line of running ./script.sh in cronjob

To read csv sample data in the home directory to Hive table, run under /mqp/modelGenerator/refractory

```
~/spark/submit --master yarn --client spark.py
```

Feature and Transaction Storage

To construct Dimension Table, run under /mqp/modelGenerator/refractory

```
~/spark/submit --master yarn --client dimension.py
```

To construct Features Table, run under /mqp/modelGenerator/refractory. Go to main function of constructFeatures.py. Comment out the join_all_features_table() and leave construct_feature_table() for running. You can change valid data percentage to keep columns with more valid data, and change time range for dataset.

```
~/spark/submit --master yarn --client constructFeatures.py
```

To construct AllFeaturesTable Table, run under /mqp/modelGenerator/refractory. Go to main function of constructFeatures.py. Comment out the construct_feature_table() and leave join_all_features_table() for running. It joins all calculated features and attributes to one table, from which customer will select attributes they want for final dataset.

```
~/spark/submit --master yarn --client constructFeatures.py
```

Note: For now, Spark and Hadoop runs on one machine in pseudo distributed mode. The system only has one node, but it distributes all files and applications to a huge storage mount. Spark sometimes don't have enough space for execute the task in Yarn(All spark tmp files, hadoop tmp files, yarn tmp files are under /tmp directory, which increases dramatically and could occupy the 5G after about 25 times spark submit trials). If Spark get stuck in the middle, like lost task executors, or stuck at the "accept" state, turn off cronjob, restart Hadoop and Spark following the command above if necessary.

Scoring Engine

To start Scoring Engine, actually the command is contained in the script.sh under /mqp/modelGenerator/refractory, so if script.sh running in the cronjob, Scoring Engine is started.

crontab -e
This will open up the script of cronjob
setup time interval for running ./script.sh in crontab

Right now it is */1 * * * * ./script.sh >> logAddress(optional) 2>&1, which means it runs every minute and save the running log to ModelGenerator directory.

To close the Model Generator

```
crontab -e
```

Comment out the line of running ./script.sh in cronjob

Creating a Model

1. View begins at Create View
2. Fill out form of fields below

Parameter	How	Explanation
Model Name	Fill-in	Name of model
Author	Fill-in	Who is creating the model
Algorithm	Dropdown	Algorithm used to generate model
Output	Dropdown	Categorical output of model
Features	Drag and drop	Features used to train model
Data date range	Fill-in + Drag and Drop	Date range of data to train the model on
Re-train frequency	Fill-in + Drag and Drop	How often to re train the model

3. Once the form is filled with valid information, hit the “Train” button and view your model in the list of models, as a detailed view, or in the queue

Viewing a Model

In the view tab

1. Go to the “View” tab
2. Click on any model you wish to view

In the queue tab

1. Go to the “Queue” tab
2. Click on any model you wish to view

In either case, clicking on the model will bring you to the model detail view.

Appendix B: Comparison of Algorithms

Classification Algorithms Comparison

	Algorithm	Advantages	Disadvantages
1	Artificial Neural Network	● Good with diverse data and capturing regularities ● Deal with complex relations	● A black box, hard to interpret the solving process ● It is not probabilistic ● Prone to overfitting
2	Hidden Markov Model	● Sequence analysis ● Produce probabilities ● Flexible to allow unknow states	● Expensive in computation time and memory
3	Multilayer Perceptron Classifier	● Capture non-linearly relation ● Computation speed is high ● Do not require assumption about statistical distribution	● Same as ANN
4	Naive Bayes	● Simple and super fast ● When independence is valid, this method is faster than others and needs less training	● Assumes independence between predictors ● Bad at classifying if category not included in training

		● Produce probability	
5	Decision Tree	● Are simple to understand and interpret. ● Help determine worst, best and expected values for different scenarios ● Can be combined with other decision techniques.	● For data including categorical variables with different number of levels, information gain in decision trees are biased in favor of those attributes with more levels. ● Calculations can get very complex particularly if many values are uncertain and/or if many outcomes are linked.
6	Random Forest	● Combine results from multiple decision trees to avoid overfitting ● Performance improves monotonically with the number of trees ● Provides the same functionalities as decision trees ● Faster by training trees in parallel ● Runs efficient on larger dataset	● Have been observed to overfit with noisy classification/regression tasks ● Include the drawbacks the decision has
7	Support vector machine	● Less overfitting ● Efficient in high dimensional spaces	● Spark only supports linear kernel function ● Computation expensive, speed is slow
8	Gradient-Boosted Trees (GBTs)	● Capture non-linearities and feature interactions ● Handle category features	● Training in sequence could result in low computation speed ● Performance decrease as the number of trees becomes larger ● Works better on smaller tree
9	Isotonic Regression	● Does not assume any form for the target function	● Not applicable for the large scale of the transactions because it has to be maintained as increasing trend always ● It is a monotonic regression

Appendix C: Notes from User Interface Review

This test began with an overview of the pipeline and its purposes and then a discussion of the human computer interaction principles and how they might apply to our project (Nielsen, 1994).

Principle	Comments
Visibility of system status	Not sure state of system, visibility of things you can do The more you can put out without confusion, the better it is
Match between system and the real world	Ask the data scientists terminology
User control and freedom	-
Consistency and standards	-
Error prevention	-
Recognition rather than recall	-
Flexibility and efficiency of use	Not very important for this task, don't want too much flexibility Expert users want to make sure they can do things quickly

Aesthetic and minimalist design	Key idea: everything you put on the screen is information
Help users recognize, diagnose, and recover from errors	Flag errors, point out where the error is, say what the error is, suggest how to fix it, helpful!

Notes before analysis:

- Consider the user and what they want
- Important question: what the features are impacts the output
- Are you assuming DS knows where all the data knows?
- Expectations: Make sure what they expect is realized
- Task analysis
    - If there were no user interface you would still want to consider what things would a user wanna do as a series of actions
- Group related items
- Consider using some kind of diagram to represent what the user will do
    - Finite State Automata diagram, state transition diagram
    - This will reveal which things are more frequent

Analysis of User Interface:

- Odd Place for the blue “confused” box, field is large, cancel is in far right
- Would move “model generation” title over to the left, line it up with left
- Lining things up is important-- looks simpler when lined up
- Closer things are more related in the eyes of the user, so make sure you are mindful of this
- Dummy entries vs real entries--hard to tell the difference of dummy prefilled values, keep all boxes initially empty
- Features are not in line, looks odd
- Arrows for inputs are a long way off from their text
- What does an author's name looks like? (username, name, etc.)
- You could make algorithm and output fields smaller because they are far from their drop down arrows which are small and it is hard to hit a small target
- Looking at ANN, shows its subsidiary information
- Scrolling should be there all the time for tuning parameters
- Error messages for entering invalid information should pop up earlier if possible, before the user hit submit
- The interface should give you error when you try to increase and its hit the max as opposed to capping out with no message

- Number boxes are very far between label and box
- Many components are equal size in a way that is misleading (i.e. not related, but same size)
- Hidden layers should be children to tuning parameters
- Colons in second indent of labels should maybe go, but they are a consistent
- Change “Name” to “Model Name” for clarity
- Typo on author label
- Make error messages look the same for consistency
- Put an arrow between the two feature boxes to make it more clear you drag, make chosen features box darker, All / Chosen - label boxes and make the labels distinguishable from names of features maybe italics or lighter font
- Scroll bar on the right is darker , make it consistent
- Even field names are far away from the field names that they apply to, make it smaller
- Clarify the retrain date
- One thing that might be an issue is how easy it is to read for people who don’t have good vision, add contrast
- Label underneath should match initial value (if we are giving an example and the box has “3 days” filled in, use that to give the example
- Date range goes negative on the last two date ranges