Real-time Malware Detection With Scalable Deep Learning: DL4S-SparkV2 Spark Scala Project Technical Report and Documentation

By: Joojay Huyn May 16, 2016

1. Introduction

Directed by Professor Mohit Tiwari, the UT Austin ECE Spark Research Lab aims to build secure systems through novel architectures and program analyses. Because many of today's applications, spanning mobile devices to cloud servers, work with sensitive data, this lab strives to protect user data from malicious web and mobile applications, malicious cloud providers, and dastardly co-tenants in the cloud. As of the 2015/2016 academic school year, this lab showed interest in building a real-time machine learning pipeline to detect malicious activity on the computer networks monitored by the UT ISO (Information Security Office), which gathers data from all public domains in the state of Texas for security purposes. Based on the behavior and trace data of a certain process on the network, can the pipeline classify the behavior as malicious or benign? This high-volume and high-velocity data amounts to several terabytes per day. While the lab has not collaborated with the ISO team yet, Joojay Huyn, with the help of Professor Tiwari, Michael Bartling, and Mikhail Kazdagli, worked to build several pipelines to be tweaked for future collaboration with the ISO team during the spring 2016 semester.

When building a real-time system that employs machine learning to detect malicious activity over some network, engineering practices generally call for three pipelines.

1) A data collection pipeline: Figure 1 shows a diagram of the ideal architecture for the data collection pipeline that ingests data quickly into a temporary storage.
2) A model building pipeline: Figure 2 illustrates the ideal architecture for the model building pipeline, where a stored dataset supplies data to be explored and to build optimal machine learning models [1, 2, 3].
3) A model execution pipeline: Figure 3 depicts the ideal architecture for the model execution pipeline, which streams and processes events in real-time with the selected machine learning model.

Figure 1

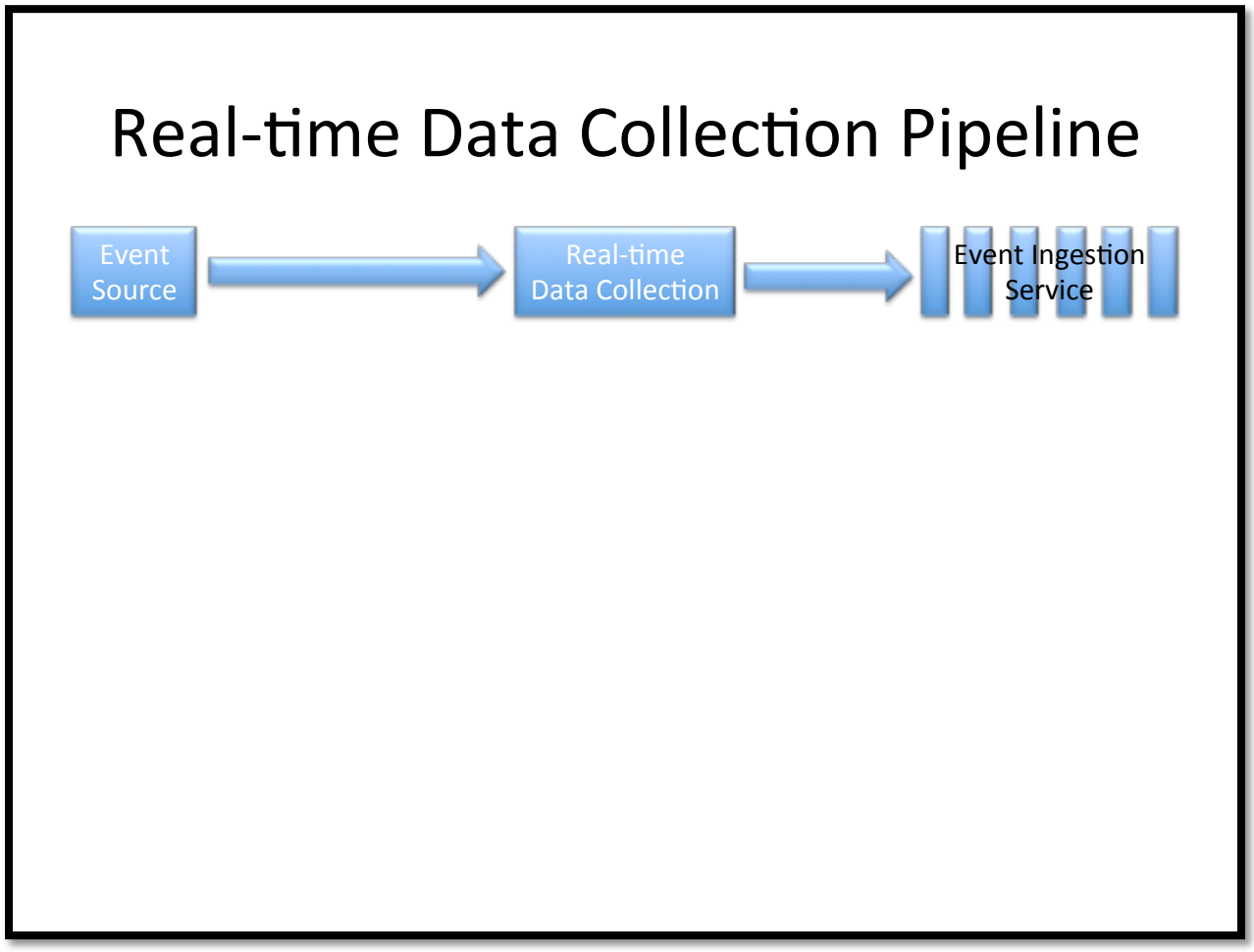

Real-time Data Collection Pipeline
Event Source
Real-time Data Collection
Event Ingestion Service

Figure 2

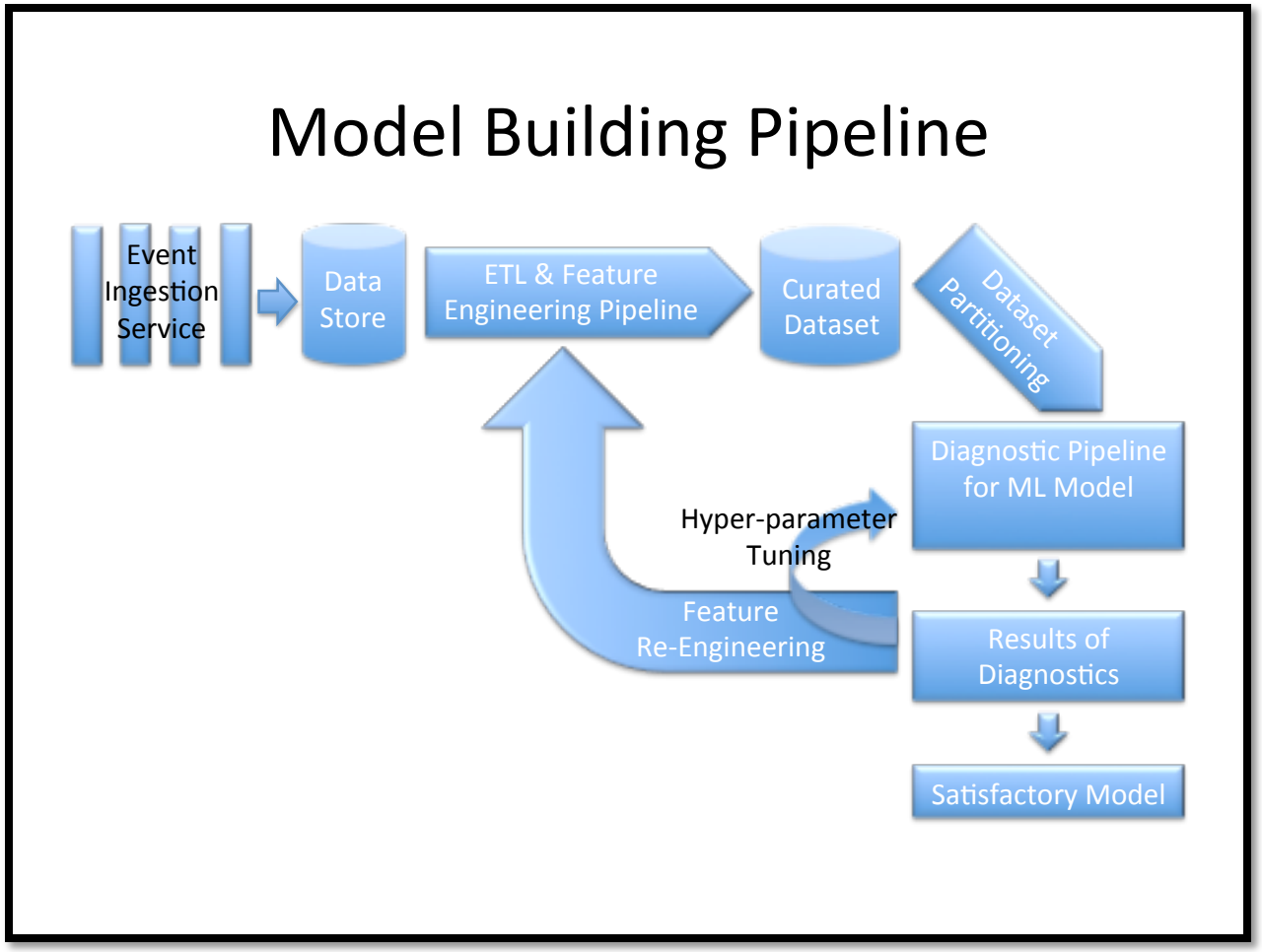
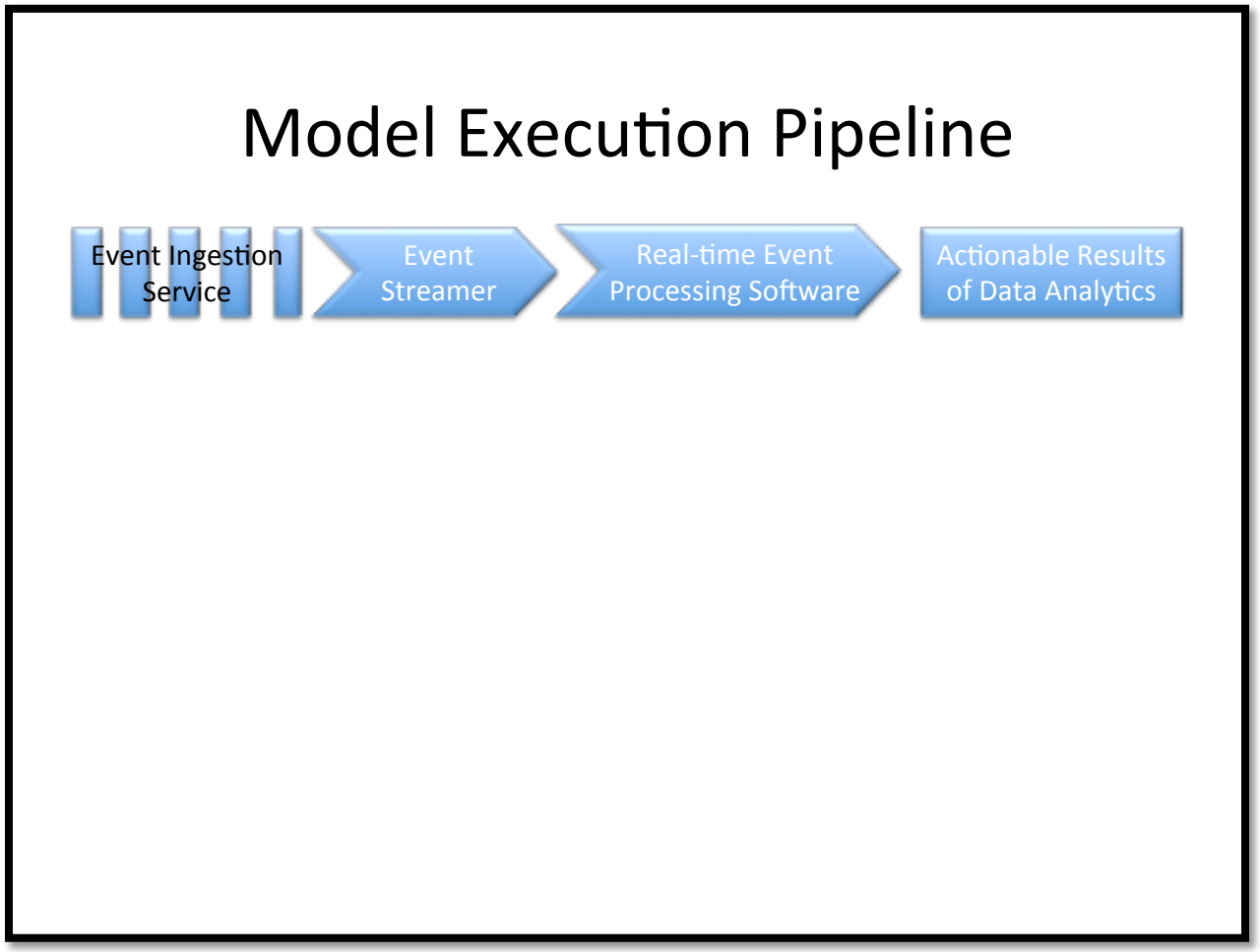

Model Building Pipeline
Event Ingestion Service
Data Store
ETL & Feature Engineering Pipeline
Curated Dataset
Dataset Partitioning
Diagnostic Pipeline for ML Model
Hyper-parameter Tuning
Feature Re-Engineering
Results of Diagnostics
Satisfactory Model

Figure 3



The DL4S-SparkV2 Spark Scala project [16] approximates a machine learning model building pipeline as shown in Figure 2. This pipeline attempts to learn and classify the behavior of a process on the network as benign or malicious. The remainder of this report explains each component of the model building pipeline and designated future work for the overall project.

2. Model Building Pipeline

2.1 Event Ingestion Service

The event ingestion service, such as Apache Kafka, does not exist currently. While this component is out of scope of this project, we should include it for the sake of completeness. With the event ingestion service in place, the data can be made persistent by streaming it into a data store.

2.2 Data Store

To produce raw data, Michael Bartling of the UT ECE Spark Research Lab used a custom Intel Pin tool that intercepts system calls of generated malware and benignware. Then, he set up a MongoDB NoSQL database that collects the raw data of system call traces and machine instruction execution statistics.

In this project, the MongoDB database malobs2016 contains four collections: Ben, Mal, fs.chunks, and fs.files. However, the two collections of interest are Ben and Mal. (Think of a collection in MongoDB conceptually as a table in the relational database world) Recall that a collection contains a list of JSON documents. The Ben collection contains 123,089 documents, where each documents refers to the system call trace of a benignware process, and the Mal collection contains 64,451 documents, where each documents refers to the system call trace of a malware process. To obtain the compressed MongoDB backup files of the malobs2016 database, contact the UT ECE Spark Research Lab. These compressed files should lie in the /data/joojayhuyn-spring2016-malware-datasets/mongoBUfeb26_2016/ directory on sparkd.

After creating a MongoDB database with the "mongorestore" command [7], install the PyMongo distribution to access MongoDB through python, and start up a MongoDB instance by opening up a terminal and typing "mongod –dbpath <directory for data files>". Then, open another terminal, start up a Python shell, and type the following commands [8, 9] to view a sample document in the Ben collection.

```
> import pymongo
> from pymongo import MongoClient
> client = MongoClient()
> db = client["malobs2016"]
> collection = db["Ben"]
> collection.find_one()
```

Figure 4: Sample document in Ben collection that represents a benignware process system call trace:


```
{
        "_id" : ObjectId("56a909ff282cd34f1605eff7"),
        "DEVMODE" : "DEVMODE",
        "numSyscalls" : 1487,
        "type" : "Ben",
        "date" : ISODate("2016-01-27T18:18:39.794Z"),
        "numSeconds" : 1,
        "app" : "freesweep",
        "PIDs" : [
                11013,
                11011,
                11012
        ],
        "command" : "freesweep_benign",
```

```
        "rawID" : [
                ObjectId("56a909ff282cd34f1605eff1"),
                ObjectId("56a909ff282cd34f1605eff3"),
                ObjectId("56a909ff282cd34f1605eff5")
        ],
        "platform" : "Debian"
}
```

To view a sample document in the Mal collection, type:

```
> collection = db[“Mal”]
> collection.find_one()
```

Figure 5: Sample document in Mal collection that represents a malware process system call trace:

```
{
        "_id" : ObjectId("56b55a61282cd365beee14d3"),
        "date" : ISODate("2016-02-06T02:28:49.961Z"),
        "args" : "exec_env",
        "rawID" : [
                ObjectId("56b55a61282cd365beee14c2"),
                ObjectId("56b55a61282cd365beee14c4"),
                ObjectId("56b55a61282cd365beee14c6"),
                ObjectId("56b55a61282cd365beee14c8"),
                ObjectId("56b55a61282cd365beee14ca"),
                ObjectId("56b55a61282cd365beee14cc"),
                ObjectId("56b55a61282cd365beee14ce"),
                ObjectId("56b55a61282cd365beee14d0")
        ],
        "PIDs" : [
                786,
                784,
                787,
                781,
                788,
                785,
                782,
                783
        ],
        "command" : "vim_malicious",
        "numSyscalls" : 5996,
        "type" : "Mal",
        "DEVMODE" : "DEVMODE",
        "platform" : "Debian",
        "numSeconds" : 11,
```

```
        "app" : "vim"
}
```

Here, the JSON documents of each collection have almost identical key-value pairs or schemas. Note that: 1) the sample document in the Mal collection has an “args” key, while the sample document in the Ben collection does not 2) the value of the “rawID” key in each document refers to a list of ids as its value. This list of ids refers to a document id that lies in the fs.files collection, which contains special grid fs files. The raw contents of a grid fs file in the fs.files collection lies in the fs.chunks collection.

Continuing from the Python commands from above, type the following [10, 11, 12] to view the sample raw data of a grid fs file (you may have to install the gridfs package, but it might come with pymongo already):

```
> import gridfs
> gridFSInstance = gridfs.GridFS(db)
> doc = collection.find_one()
> gridFSFileId = doc[“rawID”][0]
> gridFSFile = gridFSInstance.get(gridFSFileId)
> gridFSFile.read() # Warning: this will print out a lot of text to the terminal
```

Figure 6: Sample raw data (stored in fs.chunks) of a grid fs file in the fs.files collection: (In the raw data below, the collection is “Ben”)

```
18:18:43.844411 close(255)              = 0 <0.000005>\n18:18:43.844445
rt_sigprocmask(SIG_SETMASK, [], NULL, 8) = 0 <0.000004>\n18:18:43.844465
rt_sigaction(SIGTSTP, {SIG_DFL, [], SA_RESTORER, 0x7f1b7ca6c180}, {SIG_DFL, [], 0},
8) = 0 <0.000004>\n18:18:43.844485 rt_sigaction(SIGTTIN, {SIG_DFL, [], SA_RESTORER,
0x7f1b7ca6c180}, {SIG_DFL, [], 0}, 8) = 0 <0.000004>\n18:18:43.844504
rt_sigaction(SIGTTOU, {SIG_DFL, [], SA_RESTORER, 0x7f1b7ca6c180}, {SIG_DFL, [], 0},
8) = 0 <0.000004>\n18:18:43.844528 dup2(3, 0)              = 0 <0.000004>\n18:18:43.844545
close(3)              = 0 <0.000004>\n…
```

A newline character (“\n”) separates data and metadata for each system call trace. The first system call is rt_sigprocmask, the second system call is rt_sigaction, the third system call is rt_sigaction, etc….

Hence, a process, benignware or malware, can be represented as a sentence or sequence of system calls. Referencing the above JSON documents for examples, figure 4 refers to a sequence of 1487 system calls, and figure 5 refers to a sequence of 5996 system calls. In this dataset, a total of about 331 distinct system calls exist. One can add another system call named “other” to this set to catch unrecognized system calls in the trace. For more information regarding the data collection pipeline, semantics of the MongoDB malobs2016 database and dataset, or the raw MongoDB backup files, contact the UT ECE Spark Research Lab or Michael Bartling.

Future work includes setting up an MPP (massively parallel processing) relational database as data store to collect the raw data. With a highly scalable architecture, the MPP

relational database understands standard ANSI SQL queries and commands delivers query results with high performance, enabling one to leverage this speed to perform extensive data exploration and feature engineering on the dataset.

2.3 ETL and Feature Engineering Pipeline

Recall that the system call trace dataset, known as the malobs2016 dataset, contains a total of 123,089 benignware processes and 64,451 malware processes, where each processes corresponds to a sequence of system trace calls. In the /data/joojayhuyn-spring2016-malware-datasets/mongoBUfeb26_2016/ directory on sparkd, the Ben.txt file represents each benignware process system trace call as a sequence. In other words, the Ben.txt file contains 123,089 lines, where each line contains a sequence of system calls, each call separated with a comma; likewise for Mal.txt. The Ben.txt and Mal.txt files represent each system call as a number from 0 to 331 for data compression reasons, and the MichaelBartlingDataSchema.txt file in the DL4S-SparkV2 project directory maps each system call to a number.

To represent this raw unstructured data in Mal.txt and Ben.txt in an appropriate data format, such as a vector, for a machine learning classifier, two approaches come to mind: the vector space model and bi-grams model. The vector space model represents a document as a distribution of word occurrences. Assuming that the set of possible words is 10,000, then this model represents a document x as a vector of 10,000 features or dimensions, where feature 1 corresponds to the number of occurrences of word 1 in document x, feature 2 corresponds to the number of occurrences of word 2 in document x, etc…. To apply the vector space model to this research problem, represent a process (a sequence of system trace calls) as a vector of 332 features. (As stated above, this dataset has a total of 331 distinct system calls plus a 332^{nd} system call named "other" to catch unrecognized system calls in the trace) Each of the 64,451 malicious processes or data instances receives a class label of 1, and each of the 123,089 benign processes or data instances receives a class label of 0.

The bi-grams model takes the vector of 332 system calls and expands it into a vector of all possible pairs of the 332 calls, including pairs of the same call. In other words, the new vector has 332^2 or 110,224 features. To populate this vector, create a window of size 2 and iterate down the sequence of system calls associated with a given data instance. Note that the bi-grams model has vectors with larger dimensions than those in the vector space model. While this representation may cause overfitting, the bi-grams model does take the ordering of system calls somewhat into consideration while the vector space model completely disregards ordering.

While this project only reduced the malobs2016 dataset to a vector space model, which contains 187,540 vectors of 332 dimensions, future work includes experimenting with a bi-grams model, which would contain 187,540 vectors of 110,224 dimensions.

2.4 Curated Dataset:

As a result of this entire process of extraction, transformation, loading, and feature engineering, a newly curated dataset contains two csv files, Ben.csv and Mal.csv, where both files have 332 columns. Mapped to a certain system call, each column represents the number of times that system call occurred in the execution of the process. For example, the 0^{th} column of row 12 represents the number of times the "setdomainname" system call occurred in process 12. The Ben.csv file has 123,089 rows, and the Mal.csv file has 64,451 rows. To obtain this curated

dataset, contact the UT ECE Spark Research Lab. (The files should lie under the /data/joojayhuyn-spring2016-malware-datasets/mongoBUfeb26_2016/ directory on sparkd) Note that these csv files do not have a header row that contains a schema.

Future work should include a schema in the header row of Ben.csv and Mal.csv. To view the schema, open up the MichaelBartlingDataSchema.txt file in the DL4S-SparkV2 project directory. To clarify, setdomainname refers to the 0^{th} (or 1^{st} column), lseek refers to the 1^{st} (or 2^{nd} column), etc…. The order of these system calls corresponds to the order outputted by a Python script that uses a set data structure. In other words, the Python hash code function for strings and sets orders these system calls in the MichaelBartlingDataSchema.txt file by hash code.

2.5 Apache Spark, ETL, and Data Partitioning

Due to the potentially high volume (and velocity when considering the model execution pipeline), this project uses the AWS web services, Apache Spark software and the Scala programming language to conduct data analytics. According to the creators of the Apache Spark software and authors of *Learning Spark* [15], "Apache Spark is a cluster computing platform designed to be *fast* and *general purpose*. On the speed side, Spark extends the popular MapReduce model to efficiently support more types of computations, including interactive queries and stream processing". To increase performance, Spark has the ability to run computations in memory; for complex applications running on disk, Spark is also more efficient than MapReduce.

A versatile tool, Spark covers many tasks that previously required separate distributed systems, including batch applications, iterative algorithms, interactive queries, and streaming. Using the same computing engine to process these tasks, Spark enables users to easily and inexpensively combine different processing operations, often vital in production data analysis pipelines. Highly accessible, Spark offers "simple APIs in Python, Java, Scala, and SQL, and rich built-in libraries". Integrating with other Big Data tools, "Spark can run in Hadoop clusters and access any Hadoop data source, including Cassandra".

Spark contains several closely integrated components, including a fast and general-purpose computational engine at its core that schedules, distributes, and monitors applications of many computational tasks across a computing cluster. This core powers many higher-level and specialized components, such as SQL and machine learning. Other components include Spark SQL (package for working and querying with structured data), Spark Streaming (enables processing of live streams of data), MLlib (library containing machine learning functionality), GraphX (library for manipulating graphs), and Cluster Managers.

In my personal experience, the Scala programming language is the most natural language to use with the Apache Spark platform. Furthermore, not every language is created equally in Apache Spark, and the Spark Scala API offers more functionality than its Python or Java API. In fact, many engineers speculate that Scala will dominate as the data analytics programming language and will succeed Java.

The Spark programs in this project read from input csv files, Ben.csv and Mal.csv, and transform this data into appropriately formatted Spark objects, such as an RDD (resilient distributed dataset) of Labeled Points or DataFrames, a structured data representation containing rows and a schema [4, 5, 6]. Recall that a resilient distributed dataset represents a collection of values of the same type distributed across different nodes, and the resilient adjective describes how users can set the replication factor of this distributed dataset. Next, the program splits this

RDD or DataFrame into a training set (~80%) and test set (~20%). Then, the main programs train machine learning models, such as SVMs, Logistic Regression, and Multilayer Perceptrons (MLPs) [1, 2, 3], on the training set and evaluate these models on the testing set with performance metrics such as the area under the ROC curve [17] and classification accuracy rate. The ETL Scala object in the ETLUtils package contains utility functions to assist in these ETL operations. Think of the ETL Scala object as a static class that the following contains static methods.

mBartCSVToPairRDDVec:

This function takes as input two csv files, Ben.csv and Mal.csv, that represent the malobs2016 dataset in a vector space format. Then, it converts these two files to a pair of RDDs of Vectors, where both RDDs represent a set of maliciously classified and a set of benign classified feature vectors respectively. Hence, each RDD corresponds to a csv file. One RDD corresponds to the Ben.csv file, and the other the Mal.csv file.

First, this function converts the rows of a csv file to an RDD of strings, where each string is 332 numeric comma-separated values. Naturally, it follows that the function converts this RDD of strings to an RDD of Vectors, where each Vector contains 332 dimensions. Then, this function combines all the RDDs of Vectors into one large RDD (of Vectors) for scaling purposes because good practices recommend scaling the features of all feature vectors in the dataset. Specifically, this function builds a scaling model from the aggregated RDD with the StandardScaler API provided by Spark, which standardizes features by scaling to unit variance or removing the mean using column summary statistics on the samples in the dataset. Standardization can improve the convergence rate during a machine learning optimization process and also prevents against features with very large variances exerting an overly large influence during model training. After creating the scaling model, this function scales all (in this case two) RDDs of Vectors and returns a pair of RDDs of scaled Vectors.

Future work for this function includes re-implementing how the function reads in input and the application of the scaling model. Currently, I hardwired this method to read the input of two data files of interest. However, this assumption may not hold true for future projects; instead, this function should have the ability to read a list (of variable length) of data files. Second, notice that the function processes the entire dataset (the set of data that falls into the training, cross-validation, and test set) to build a scaling model. Is this methodology of scaling appropriate? Or should the function build a scaling model by processing only the training set (~80% of the entire dataset), apply this model to RDDs of Vectors assigned to this training set, and save this same model to later transform RDDs of (never seen before) Vectors assigned to the test set? These two methodologies deserve some investigation, especially from a statistical perspective. Perhaps one is recommended or preferred over the other?

mBartCSVToRDDLP:

This function takes as input two csv files, Ben.csv and Mal.csv, which represent the malobs2016 dataset in a vector space format. Then, it converts these two files to an RDD of Labeled Points. A Labeled Point object represents a feature vector with an assigned class label.

First, this function calls the mBartCSVToPairRDDVec function to transform the two csv files of interest to a pair of RDDs of Vectors, where each RDD corresponds to a collection of

feature vectors for its assigned class label. Then, this function simply wraps a Labeled Point object around each vector. In other words, it transforms an RDD of Vectors to an RDD of Labeled Points. A Spark object, a Labeled Point represents one feature vector and has two parts, a class label and its corresponding vector. Then, the function aggregates all RDDs of Labeled Points into one large RDD, which contains Labeled Point objects of all class labels, and returns this RDD. Note that machine learning classifiers in the Spark mllib (not ml) library expect data to come in the format of an RDD of labeled points.

Future work involves reimplementing part of this function to read in a list of csv files, not just two files.

mBartCSVToMLDataFrame:

This function takes as input two csv files, Ben.csv and Mal.csv, which represent the malobs2016 dataset in a vector space format. Then, it converts these two files to a DataFrame, a collection of rows with a schema. The DataFrame Spark object represents structured data much like a relational database.

First, this function calls the mBartCSVToPairRDDVec function to transform the two csv files of interest to a pair of RDDs of Vectors. Then, this function wraps a Row object around each Vector object, thus transforming an RDD of Vectors to an RDD of Rows. While a Row object simply represents a collection of objects, the Row object in this scenario must follow a strict format. In this format, the Row has two parts, a class label and its corresponding feature vector. In fact, this function creates a schema of two named columns: the first column, called "label", is of type DoubleType, and the second column, called "features, is of type VectorUDT; the Row must conform to this schema. Then, the function creates a DataFrame with the schema object and RDD of Rows. The DataFrame created in this function must strictly adhere to this schema format because the machine learning classifiers in the Spark ml (not mllib) expect a DataFrame and schema with two columns, one called "label", and the other called" features".

Future work involves reimplementing part of this function to read in a list of csv files, not just two files.

getTrainTestSetsFromRDDLP and **getTrainTestSetsFromDataFrame:**

These functions randomly split an input RDD of Labeled Points or an input DataFrame into a training set (~80%) and test set (~20%) for the purposes of training and evaluating machine learning classifiers. Perhaps the getTrainTestSetsFromRDDLP function should be renamed to getTrainTestSetsFromLPRDD?

getTrainCVTestSetsFromRDDLP and **getTrainCVTestSetsFromDataFrame:**

These functions randomly split an input RDD of Labeled Points or an input DataFrame into a training set (~60%), cross-validation set (~20%), and test set (~20%) for the purposes of training and evaluating machine learning classifiers. Perhaps the getTrainCVTestSetsFromRDDLP function should be renamed to getTrainCVTestSetsFromLPRDD for clarity.

2.6 Diagnostics

The term diagnostics here refers to a debugging or tuning process. Suppose the machine learning classifiers does not perform as expected or hoped. Several methodologies exist to investigate where this problem exists and how to fix this problem. Perhaps the problem lies in biased or faulty data, perhaps some of the machine learning parameters must be tuned to yield a better performance, or perhaps the machine learning pipeline performs slowly for some reason The diagnostic process can prove challenging, and many projects require engineers to create a custom diagnostic process tailored to their own process. Often, engineers combine several different methodologies to debug their machine learning pipeline. Some of the programs in this project run simple diagnostic tests, such as 10-fold cross validation, consistent with the diagnostic tests that other PhD students have used in the UT ECE Spark Research Lab.

Of the three driver programs in the MBartlingSparkML, only the MLPDriver performs some sort of diagnostic test. The SVMDriver and LogRegDriver programs simply train an SVM model and logistic regression model on a training set given the default parameters provided by the Spark API. Then, these drivers measure the performance of the SVM and logistic regression model by returning each model's area under the roc curve (AUROC) and classification accuracy rate on a never-before-seen test set. However, one can easily add a diagnostic component for the LogRegDriver via the Spark APIs spark.ml.tuning.CrossValidator or spark.ml.tuning.TrainValidationSplit. Regarding the SVMDriver, it is unclear whether Apache Spark provides APIs to conduct diagnostic tests on the SVM model in the spark..mllib.classification API. If not, one must manually write the diagnostic test.

The MLPDriver program runs a 10-fold cross-validation test [1, 2, 3] to select the architecture that performs best on a validation dataset using the AUROC as an evaluation metric. This program must return the AUROC metric to keep consistent with the work of other PhD students in the UT ECE Spark Research Lab. In other words, other students have trained and tuned machine learning models with 10-fold cross validation, and these models return the AUROC metric as well. With consistent diagnostic methodologies and performance metrics, one can fairly compare the results of the multilayer perceptron model in the MLPDriver program with machine learning models of other student researchers. As stated in the source code, this program unfortunately cannot take advantage of the cross-validation and diagnostics API provided by the spark.ml library. Why?

Currently, the spark.ml.evaluation.MulticlassClassificationEvaluator class does not support the AUROC metric, and problems occur when evaluating the spark.ml.classification.MultilayerPerceptronClassificationModel object with a spark.ml.evaluation.BinaryClassificationEvaluator object [6]. When classifying data instances in a test dataset of type DataFrame, the MultilayerPerceptronClassificationModel returns a DataFrame with a schema of three columns: label, features, and prediction. Prediction refers to the multilayer perceptron model's final predicted class label of type double of a data instance. However, the BinaryClassificationEvaluator constructor expects a DataFrame with a raw prediction column, where this raw prediction column represents a vector of predicted probabilities for each class label. This issue forces me to convert the resulting predicted DataFrame of the multilayer perceptron model back into an RDD of pairs of doubles. In this RDD, the first double represents the model's final predicted class label of the instance, and the second double represents the actual class label of the instance. The spark.mllib.evaluation.BinaryClassificationMetrics constructor accepts an RDD of pairs of

doubles and can return an AUROC metric for the multilayer perceptron model's prediction for a testing set.

Wait a minute! If the multilayer perceptron model ultimately returns a final predicted class label for a given data instance (and not a probabilistic score or vector of probabilities for each class label), how can the Spark API correctly calculate the AUROC? In other words, the multilayer perceptron model returns a final predicted class label (1.0 or 0.0), resulting in the creation of only one point, not a collection of points to form a ROC curve. While academic and research papers have created formulas to measure the area under the ROC curve given only one point of the ROC curve, the Spark API source code initializes the curve with 2 points already. These points are (0,0) and (1,1) or the bottom left and top right corners of the ROC curve graph respectively. Therefore, adding a third point created by the multilayer perceptron model's prediction would create a rough ROC curve of three points, and the Spark API uses the trapezoidal rule to calculate the area under this rough ROC curve. Note that if the ROC curve is convex, then the trapezoidal calculation provides a lower bound on the actual AUROC. Hence, the actual AUROC would be better.

Future work suggests that one could attempt to create an automated diagnostic pipeline or manual diagnostic pipeline involving data visualizations.

3. Results

The log files in the SampleLogs folder (in the DL4S-SparkV2 project directory) show the evaluation results of running the classifiers, SVM, Logistic Regression, and Multilayer Perceptron, on a never-before-seen test set. Executed on a Spark AWS EC2 cluster, the programs that run and evaluate these classifiers are located under the MBartlingSparkML directory of the DL4S-SparkV2 project.

Spark EC2 cluster:

- 1 master node
- 4 worker nodes
- All nodes running Amazon Linux OS
- All nodes of type m4.xlarge machine
- Each node has 4 cores
- Each node has 14.4 GB of memory
- BLAS library NOT installed

Classifier Hyper-Parameters

Classifier	Learning rate α	Opt. Algo	# of iter.	λ	Regularization Type	Use of Intercept	Other Parameters
SVM	1.0	SGD	100	0.01	L2	False	N/A
Logistic Regression	N/A	LBFGS	100	0.0	L2	True	Decision boundary = 0.5
Spark MLP	N/A	LBFGS	100	N/A	N/A	True	See below

DL4J MLP	N/A	LBFGS	100	0.01	L2	N/A	See below

Spark MLP and DL4J MLP:

- 1 hidden layer
- 332 units in input layer
- 150 units in hidden layer
- 2 units in output layer
- Sigmoid activation function in hidden layer
- Softmax activation function in output layer
- MLP seed for weights initialization: 1234L
- Spark MLP documentation is not clear whether regularization is supported
- DL4J documentation is not clear whether it uses intercepts/bias units in the network
- DL4J seed for weights initialization: 12345

Classifier Results

Classifier	Completion Time on AWS Spark Cluster with HDFS	AUROC	Classification Accuracy Rate
SVM	~29 seconds	0.8388	0.8886
Logistic Regression	~59 seconds	0.9826	0.8886
Spark MLP	~3.2 hours	0.9149	0.9160
DL4J MLP	24 minutes	0.5	0.6545

Spark MLP:

- Uses 10-fold cross validation on 3 architectures to train 31 models
- ~6.1935 minutes to train one model

DL4J MLP:

- For some reason, it takes 24 minutes to train one model
- For some reason, it performs poorly on the test set
- The code in the driver program MBartlingDL4J.MLPDriver compiles
- I have not spotted what is wrong with the code…it should have about the same performance as the Spark MLP driver program

Classifier	Local Machine (Mac OS X)	AUROC	Classification Accuracy Rate
SVM	~68 seconds	0.8378	0.8876
Logistic Regression	~108 seconds	0.9820	0.8876
Spark MLP	> 31 hours (Estimated)		
DL4J MLP	java.lang.OutOfMemoryError: Java heap space		

Spark MLP:

- Runs very slowly because training dataset is not cached

- After 60 minutes, the program has not managed to even train one model
- The program uses 10-fold cross-validation to train 31 different models
- If training dataset is cached, program does not execute properly
- If a program attempts to cache an RDD that does not fit in memory, program must resort to persisting RDD in disk storage

DL4J MLP

- Will not even run due to an "Out of memory" exception

4. Future Work

This section recommends a few possible experiments and directions for future work:

- Incorporate an event ingestion service, such as Apache Kafka or Spark Streaming, into the data collection pipeline
- Change data store from MongoDB to an MPP relational database. Apache Kudu seems to be an open source MPP relational database [18].
- Leverage SQL to conduct data exploration and feature engineering
- Experiment with bi-grams representation of data
- Experiment with potentially relevant signal processing techniques, such as fourier transformations, wavelets, and spectral analysis, to represent data
- Use Apache Spark to connect with data store via MongoDB-Spark connector or JDBC (for MPP relational database)
- Code refactoring for efficiency and readability purposes
- Install BLAS library to accelerate matrix computations and pipeline performance
- Attempt to leverage GPUs (on AWS?) to accelerate performance with Deep Learning libraries that communicate with Apache Spark (not obvious)
- Investigate automated vs. manual diagnostic pipeline
- Tweak other hyper-parameters of multilayer perceptron, such as the regularization parameter and seed for weight initialization
- Experiment with other deep-learning models, such as recurrent neural networks, which can be used to classify variable-length sequences of words. Building RNN models require large datasets.
- Experiment with other machine learning models, such as Hidden Markov Models
- Build a testing suite to increase confidence in code performance and facilitate debugging
- Build a model execution pipeline, which leverages Apache Kafka and Spark Streaming, to analyze in real-time incoming data by feeding it to a selected pre-built machine learning model for classification

5. Conclusion

As of May 16, 2016, the DL4S-SparkV2 project has accomplished three milestones not achieved in the lab previously. This project sets up a model building pipeline of a simple deep learning network, a multilayer perceptron specifically. Furthermore, this pipeline runs on an AWS Spark EC2 cluster and is infinitely scalable in theory. Lastly, the deep learning network performs relatively well and yields an area under the ROC curve of ~0.915, which breaks the 0.9

barrier. Previously to my understanding, other machine learning classifiers tuned in the lab could only achieve an area under the ROC curve of ~0.85. This suggests that a multilayer perceptron and other deep learning networks perform better than basic machine learning classifiers, such as logistic regression and support vector machines. Further tuning of other deep learning parameters (more complex architectures, seeds for weight initialization and regularization) and feature re-engineering to represent the raw data as vectors with higher dimensions , such as a bigrams model, could even increase the performance of a multilayer perceptron.

6. References

1. Stanford Coursera Machine Learning Course (Andrew Ng): https://www.coursera.org/learn/machine-learning/

2. Stanford CS 229 Machine Learning Course (Andrew Ng): http://cs229.stanford.edu/

3. University of Toronto Coursera Neural Networks for Machine Learning Course (Geoffrey Hinton): https://www.coursera.org/course/neuralnets

4. Apache Spark Documentation: http://spark.apache.org/docs/latest/

5. Apache Spark MLlib Documentation: http://spark.apache.org/docs/latest/mllib-guide.html

6. Apache Spark Scala API Documentation: http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.package

7. MongoDB Documentation: https://docs.mongodb.com/manual/

8. PyMongo Documentation: https://api.mongodb.com/python/current/

9. PyMongo Tutorial: http://api.mongodb.com/python/current/tutorial.html

10. GridFS Documentation: https://docs.mongodb.com/manual/core/gridfs/

11. GridFS for Python Documentation: http://api.mongodb.com/python/current/api/gridfs/

12. GridFS for Python Tutorial: http://api.mongodb.com/python/current/examples/gridfs.html

13. DL4J Website: http://deeplearning4j.org/

14. DL4J Java Documentation: http://deeplearning4j.org/doc/

15. *Learning Spark* (O'Reilly books Publishing) by Holden Karau, Andy Konwinski, Patrick Wendell, and Matei Zaharia

16. DL4S-SparkVS project directory: https://github.com/joojayhuyn/DL4S-SparkV2.git

17. ROC Curve: https://en.wikipedia.org/wiki/Receiver_operating_characteristic

18. Apache Kudu: http://getkudu.io/