



All People > Claudio Fahey > Claudio Fahey's Big Data Blog > 2016 > April > 11

Claudio Fahey's Big Data Blog



Real-Time Global Anomaly Detection in IoT with EMC Elastic Cloud Storage (ECS) - Part 2

Posted by Claudio Fahey in [Claudio Fahey's Big Data Blog](#) on Apr 11, 2016 9:51:41 PM

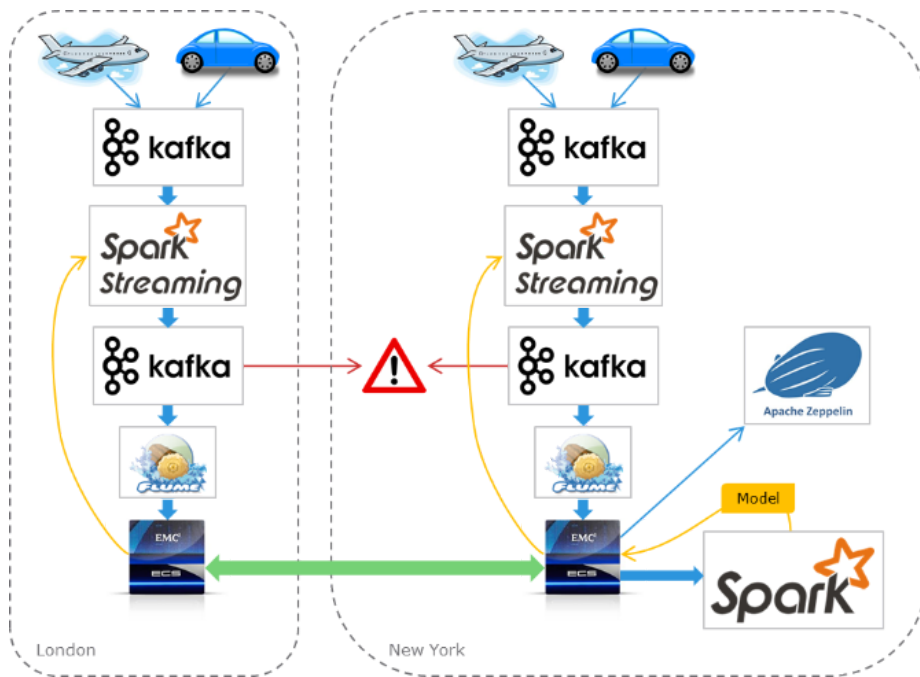
- [Part 1](#)
- [Part 2](#)
 - [Why Python?](#)
 - [Test Data Set](#)
 - [Data Generator](#)
 - [Kafka](#)
 - [Spark Streaming - Real-Time Anomaly Detection](#)
 - [Flume](#)
 - [EMC Elastic Cloud Storage \(ECS\)](#)
 - [Duplicate Records and At-Least-Once Semantics](#)
 - [Latency and Throughput](#)
 - [Spark - Machine Learning Model Training](#)
 - [Launching Spark](#)
 - [Unsupervised Random Forest](#)
 - [Hyperparameter Tuning](#)
 - [Interactive Machine Learning with Zeppelin](#)
 - [References](#)
- [Part 3](#)

Part 1

See [Part 1](#) of this blog series.

Part 2

In [Part 1](#) of this blog series, I presented the overall architecture of this system for performing near real-time global anomaly detection. In this part, I'll dig into some of the details.



The Python source code for this system is available from [this GitHub repository](#).

Why Python?

First, I'll assume that that you have decided on Spark for at least some of your data science tasks. Today, Spark applications can be written in Scala, Java, Python, and R. All of the coding that I needed to do for this system was done in Python. Why did I choose Python? According to a [recent survey by Stack Overflow](#), 63% of data scientists use Python. It's also extremely popular in the scientific computing, dev ops, operating systems, and with those of us who want to automate tasks of nearly any sort. [Do you still need more reasons?](#)

You will find that there are some things that can only be done in Spark using Scala but with each release of Spark, you'll be able to do more and more with Python.

The Python that comes standard on nearly all versions of Linux usually does not include the libraries that data scientist use most often - Numpy, Pandas, Scikit-learn, Matplotlib, etc.. Although these packages can be added to your default Python installation, the easiest way to get all of these libraries is by installing [Anaconda Python](#). There is a free community edition as well as subscription-based offerings.

Test Data Set

One of the challenges in building working data-centric systems that will be shared with the public is finding an applicable data set. After lots of searching, I landed on the [KDD Cup 1999 data set](#). This is a data set that was used in a competition to build a network intrusion detector. The data is essentially what you might expect to receive from a network router and contains fields such as the protocol type, number of bytes sent/received, packet flags, and session duration. This data set in particular is often used to develop anomaly detection systems, which is a key part of a typical network intrusion detection system.

Now if you saw the nice diagram shown in Part 1, you may have noticed that I showed planes and cars sending streaming data. And you are disappointed now because I'm working with network intrusion data. Yes, this network data from 1999 is not nearly as exciting but it is public (unlike your typical airliner's data) and as you will see that the bulk of this system will apply regardless of the nature of the data.

The KDD Cup 1999 data consists of nearly 5 million records in a CSV file. The first record is:

```
0,tcp,http,SF,215,45076,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,1,1,0.00,0.00,0.00,0.00,1.00,0.00,0.00,0,0,0.00,0.00,0.00,0.00,normal.
```

I won't describe the meaning of any of the fields here but if you are interested, the link above has additional details.

Data Generator

The first thing I needed was to build a data generator application that would use the KDD Cup 1999 data to send JSON messages to Kafka. The Python script **streaming_data_generator.py** does this. Note that this script and all other required files are in the [GitHub repo](#).

The script **streaming_data_generator.py** does the following:

1. Reads the entire KDD Cup 1999 CSV file into memory. It is 709 MB so this only takes about 10 seconds. This makes use of the excellent [Pandas](#) library to read the CSV file into an in-memory dataframe (not to be confused with a Spark dataframe).
2. Loop forever:
 1. Pick a random record from our dataframe.
 2. Add a few fields to make it look like a network device sent this as a real-time message. We add a timestamp, a unique UUID, an incrementing sequence ID, and finally the record number from our original dataframe. None of these new fields will be used by the anomaly detection algorithm.
 3. Convert the record to JSON.
 4. Send the record to the Kafka topic named sensor_messages.
 5. Sleep long enough to limit our send rate as desired. This is done with a token bucket.

The script accepts several options as shown below.

```
$ ./streaming_data_generator.py --help
Usage: streaming_data_generator.py [options]

Options:
  -h, --help            show this help message and exit
  --kdd_file=KDD_FILE   path for KDD data
  --kafka_zookeeper_hosts=KAFKA_ZOOKEEPER_HOSTS
                        list of Zookeeper hosts (host:port)
  --kafka_broker_list=KAFKA_BROKER_LIST
                        list of Kafka brokers (host:port)
  --kafka_message_topic=KAFKA_MESSAGE_TOPIC
                        topic to produce messages to
  --throttle_messages_per_sec=THROTTLE_MESSAGES_PER_SEC
                        Limit to specified rate
  --throttle_sleep_sec=THROTTLE_SLEEP_SEC
                        Seconds to sleep when throttled
  --one_sequential_pass
                        Send entire data set in order exactly once
```

If you want to see the messages sent to Kafka, you can use the Kafka console consumer.

```
$ kafka-console-consumer.sh --zookeeper zk.example.com:2181 --topic sensor_messages --max-messages 1
{"num_access_files": 0, "src_bytes": 230, "srv_count": 26, "num_compromised": 0, "error_rate": 0.0, "urgent": 0, "dst_host_same_srv_rate": 1.0, "duration": 0,
"label": "normal.", "srv_error_rate": 0.0, "srv_serror_rate": 0.0, "is_host_login": 0, "wrong_fragment": 0, "uuid": "8238af16-9942-43d2-a541-1756929a37ef",
"service": "http", "error_rate": 0.0, "num_outbound_cmds": 0, "is_guest_login": 0, "dst_host_error_rate": 0.0, "dst_host_srv_error_rate": 0.0, "diff_srv_rate": 0.0,
"hot": 0, "dst_host_srv_count": 255, "logged_in": 1, "num_shells": 0, "dst_host_srv_diff_host_rate": 0.040000000000000001, "index": 1358500,
"srv_diff_host_rate": 0.19, "dst_host_same_src_port_rate": 0.01, "root_shell": 0, "flag": "SF", "su_attempted": 0, "dst_host_count": 117, "num_file_creations": 0,
"protocol_type": "tcp", "count": 8, "utc": "2016-04-08T22:59:44.978076", "land": 0, "same_srv_rate": 1.0, "dst_bytes": 2446, "sequence_id": 1654094,
"dst_host_diff_srv_rate": 0.0, "dst_host_srv_error_rate": 0.0, "num_root": 0, "num_failed_logins": 0, "dst_host_serror_rate": 0.0}
```

Kafka

Apache Kafka, in short, is a distributed and reliable message bus. Each data center will have its own Apache Kafka cluster to ensure functionality in the event of a communication failure between data centers. The service that maintains the persistent Kafka queues (among other tasks) is called the Kafka broker. Publishers are called producers and subscribers are called consumers.

Kafka is designed to store its data on the local disks of each node. In particular, it does not store any data on HDFS or any Hadoop-compatible file system. Kafka handles data protection by performing its own replication between the Kafka brokers.

Each Kafka cluster has the following topics in our system:

- **sensor_messages**: This topic will contain the messages sent by the sensors. In our case, the data generator will publish the synthetic data to this topic. It will be consumed by our Spark Streaming job.
- **enriched_data**: This topic will contain the enriched data. Each message in this topic will contain a single sample. In our case, Flume will consume the messages in this topic and persist them to reliable storage.
- **alerts**: When the anomaly detector predicts that a message is an anomaly, that message will be published to this topic. Our Zeppelin dashboard will continually show the most recent contents of the alerts topic.

Although Kafka will by default automatically create the topics the first time they are used, it will be important to manually create the topics ahead of time so that we can specify the optimal number of partitions and the replication factor. You should have enough partitions so that the topic can be parallelized across the broker and consumer instances. As a rule of thumb, the number of partitions will be dictated by rate of the slowest consumer. For example, if the topic receives 1000 messages/sec and your slowest consumer can process 10 message/sec, then you will need a minimum of 100 consumers and 100 partitions so that each consumer gets a distinct partition.

The replication factor of three will replicate the messages across three nodes, allowing any two to die before messages become lost.

To create the topics, use the commands below.

```
$ kafka-topics.sh --zookeeper zk.example.com:2181 --topic sensor_messages --create --partitions 4 --replication-factor 3
$ kafka-topics.sh --zookeeper zk.example.com:2181 --topic enriched_data --create --partitions 4 --replication-factor 3
$ kafka-topics.sh --zookeeper zk.example.com:2181 --topic alerts --create --partitions 4 --replication-factor 3
```

For non-production use, you may want to set the Kafka configuration property `log.retention.bytes` to something like 1 GB to put an upper-limit on the local disk space that Kafka uses.

Spark Streaming - Real-Time Anomaly Detection

A summary of the the Spark Streaming job is provided in Part 1. This job is implemented as the Python script **spark_streaming_processor.py** and it is executed by PySpark.

First, we must create a Spark Streaming context with the line below. Note that arguments are shown hard coded to help clarify the example code. Below tells Spark Streaming to create batches of 15 seconds which means every 15 seconds on multiples of 15 seconds (0, 15, 30, and 45 seconds after the beginning of each minute).


```
--kafka_alert_topic=KAFKA_ALERT_TOPIC
    topic to produce alert messages to
--kafka_enriched_data_topic=KAFKA_ENRICHED_DATA_TOPIC
    topic to produce enriched data to
--streaming_batch_duration_sec=STREAMING_BATCH_DURATION_SEC
    Streaming batch duration in seconds
--max_batches=MAX_BATCHES
    Number of batches to process (0 means forever)
```

The Bash script **spark_streaming_processor.sh** can be used to launch the Spark Streaming script with the correct parameters.

While a Spark Streaming job is running, you can view the Spark UI by opening the YARN Resource Manager, navigating to your job, and clicking on the Application Master link in the Tracking UI field. The Streaming tab has several useful charts and metrics as shown below.



You'll want to make sure that the the scheduling delay is 0 ms for most batches. The average processing time will show you the lowest possible value that you should use for the streaming batch duration until pending batches start accumulating.

Flume

The standard method to write a Spark dataframe to Hadoop is:

```
df.write.json('/tmp/mydata')
```

Unfortunately, this method, as well as all other standard methods in Spark, are unable to append to existing files so instead they create a new file for each partition of the underlying RDD. You can specify an "append" mode in the above function call, but it actually writes a new file into an existing directory instead of appending to existing files. This is a limitation of the underlying Hadoop library that Spark uses to write to Hadoop. So if we have 15 second batches and 4 Spark executors, we would end up with more than 23,000 files in just one day. Unless your data rate is extremely high, this will likely incur excessive overhead to deal with so many relatively small files. An excessive number of relatively small files will use more Name Node memory (if using HDFS), directory enumerations will go slower, and batch jobs will have under-sized splits.

Flume can be used to avoid this problem very effectively. Flume can receive a stream of data from a variety of sources and write them to a Hadoop-compatible file system in an optimal way. We'll configure Flume so that it receives the enriched data from Kafka and writes them to ECS. Each Flume agent will write to only one file at a time. Flume will flush the file every few seconds to ensure that the file system has the latest data. Any readers of the data will see the latest data up the most recent flush. When the file reaches an optimal size (we use 128 MiB), it will close the file and begin writing to a new file.

The flume.conf file that accomplishes this is:

```
flume1.sources = kafka-source-1
flume1.channels = hdfs-channel-1
flume1.sinks = hdfs-sink-1

flume1.sources.kafka-source-1.type = org.apache.flume.source.kafka.KafkaSource
flume1.sources.kafka-source-1.zookeeperConnect = zk.example.com:2181
flume1.sources.kafka-source-1.topic = enriched_data
flume1.sources.kafka-source-1.batchSize = 100
flume1.sources.kafka-source-1.channels = hdfs-channel-1

flume1.channels.hdfs-channel-1.type = file

flume1.sinks.hdfs-sink-1.channel = hdfs-channel-1
flume1.sinks.hdfs-sink-1.type = hdfs
flume1.sinks.hdfs-sink-1.hdfs.writeFormat = Text
flume1.sinks.hdfs-sink-1.hdfs.fileType = DataStream
```

```

flume1.sinks.hdfs-sink-1.hdfs.filePrefix = enriched_data
flume1.sinks.hdfs-sink-1.hdfs.useLocalTimeStamp = true
flume1.sinks.hdfs-sink-1.hdfs.path = viprfs://repbucket1.ns1.site1/tmp/ecs_iot_demo2/enriched_data/%Y-%m-%d/site1
flume1.sinks.hdfs-sink-1.hdfs.rollCount=0
flume1.sinks.hdfs-sink-1.hdfs.rollInterval=0
flume1.sinks.hdfs-sink-1.hdfs.rollSize=134217728

```

The above Flume configuration works on Flume 1.5.2. If you are using Flume 1.6 or higher, you may want to use the Kafka channel instead of the Kafka source. See flume-1.6.conf for such a configuration..

The directory listing below shows how Flume outputs the data. The file currently being written has a .tmp extension. If there are several Flume agents, you will see a .tmp file for each one.

```

-rw-r--r-- 1 flume hdfs 128.1 M 2016-04-10 03:07 enriched_data/2016-04-10/site1/enriched_data.1460246408056
-rw-r--r-- 1 flume hdfs 128.1 M 2016-04-10 03:28 enriched_data/2016-04-10/site1/enriched_data.1460246408057
-rw-r--r-- 1 flume hdfs 128.1 M 2016-04-10 03:49 enriched_data/2016-04-10/site1/enriched_data.1460246408058
-rw-r--r-- 1 flume hdfs 9.8 M 2016-04-10 03:50 enriched_data/2016-04-10/site1/enriched_data.1460246408059.tmp

```

We can check that the file system has up-to-date data with the following command.

```

$ hadoop fs -tail viprfs://repbucket1.ns1.site1/tmp/ecs_iot_demo2/enriched_data/2016-04-10/site1/enriched_data.1460246408059.tmp ; date -u
... "utc": "2016-04-10T04:00:44.935864", ...
Sun Apr 10 04:00:53 UTC 2016

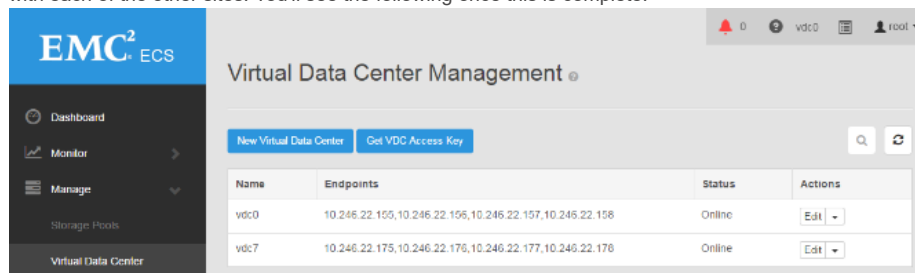
```

You may notice that Flume is configured to dump the records from each site (data center) in a site-specific directory (site1). This is simply a way to help organize our directory structure. On a geo-replicated file system such as ECS, this is not strictly necessary.

EMC Elastic Cloud Storage (ECS)

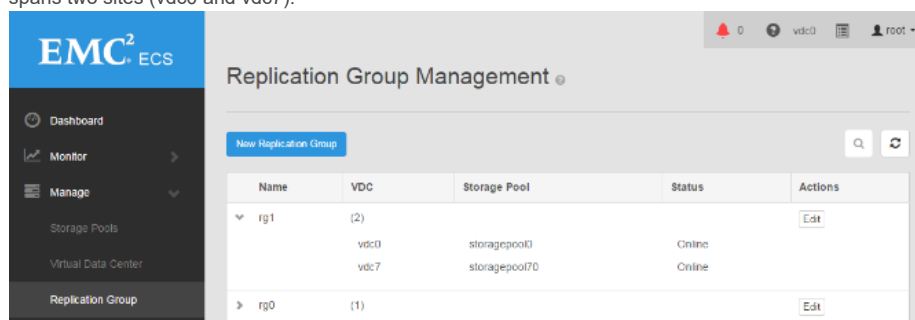
A summary of ECS is provided in Part 1. For our use case, we read and write files on ECS using the viprfs protocol. This is provided by a Java Archive file called viprfs-client.jar that implements the Hadoop-compatible File System (HCFS) interface. With a few exceptions, all Hadoop applications that can use HDFS can use an HCFS such as viprfs-client.jar.

Once you have an ECS cluster installed at a minimum of two sites, you'll configure one of the sites (called a Virtual Data Center or VDC) to communicate with each of the other sites. You'll see the following once this is complete.



Name	Endpoints	Status	Actions
vdc0	10.246.22.155, 10.246.22.156, 10.246.22.157, 10.246.22.158	Online	Edit
vdc7	10.246.22.175, 10.246.22.176, 10.246.22.177, 10.246.22.178	Online	Edit

Next, you'll create a replication group that will be used to group multiple VDC's and their storage pools. In the screen shot below, the replication group rg1 spans two sites (vdc0 and vdc7).



Name	VDC	Storage Pool	Status	Actions
rg1	(2)			Edit
	vdc0	storagepool0	Online	
	vdc7	storagepool70	Online	
rg0	(1)			Edit

Finally, you'll create a bucket in this replication group.

The screenshot shows the EMC² ECS console interface. On the left is a navigation menu with options: Dashboard, Monitor, Manage (expanded), Storage Pools, Virtual Data Center, Replication Group, Authentication, Namespace, Users, Buckets (selected), File, and Settings. The main panel displays the configuration for a bucket named 'repbucket1'. Fields include: Name (repbucket1), Namespace (ns1), Replication Group (rg1), and Bucket Owner (hdfs). There is a checkbox for 'Set current user as Bucket Owner' which is unchecked. Below these is a 'Bucket Tagging' section with an 'Add' button and a table with columns 'Key', 'Value', and 'Actions'. At the bottom, there are toggle switches for 'Quota' (Disabled) and 'File System' (Enabled).

Once the bucket has been created, you'll be able to access it from Hadoop by using the FQDN as shown below.

```
$ hadoop fs -ls viprfs://repbucket1.ns1.site1/
```

Because this bucket belongs to the replication group that spans multiple VDC's, the same bucket can be accessed from site 2 and it is guaranteed to reflect the exact same state of the file system at all times. ECS is not just "eventually consistent." It is strongly consistent although the consistency guarantee can be relaxed if you choose in order to provide read-only access in the event of a communication failure between sites.

Once you start writing data to the replicated bucket, you can view the replication status.

The screenshot shows the 'Geo Replication' status page in the EMC² ECS console. The left navigation menu includes: Hardware Health, Node & Process Health, Chunk Summary, Erasure Coding, Recovery Status, Disk Bandwidth, and Geo Replication (selected). The main panel shows tabs for 'Rate and Chunks' (selected), RPO, Failover Processing, and Bootstrap Processing. Below the tabs is a table showing replication status for 'rg1'. The table has columns: Replication Group, Write Traffic, Read Traffic, User Data Pending Replication, Metadata Pending Replication, and Data Pending XOR. The values for 'rg1' are: 17.77 KB/s, 17.54 KB/s, 0.0 B, 0.0 B, and 0.0 B respectively.

To get started with ECS, you can download the [Community Edition of ECS](#).

Duplicate Records and At-Least-Once Semantics

One common difficulty in streaming systems is the possibility of lost or duplicate records. The possibility of lost records can be virtually eliminated with an effective acknowledgement and retry mechanism at each point in our system. For instance, if a network link goes down during the transmission of a message, the sender will not get an acknowledgement from the receiver and it will know to resend the message. Or in the case of a Spark job, a failed task will be detected and retried.

The more difficult problem to overcome is that of duplicate messages. If a network link goes down exactly at the end of a message transmission so that the message is completely received but the acknowledgement to the sender is lost, then the sender will resend the message and the receiver will get the same message twice. In the case of a Spark job, a task may fail after some or even all of the task's side effects have occurred.

A system that guarantees that no message will be lost but allows for duplicate messages is said to have at-least-once semantics. A system that guarantees that no message will be lost and that no message will be duplicated is said to have exactly-once semantics.

Kafka, Spark, and Flume are each designed to have at-least-once semantics, meaning that duplicates message may occur. There are sophisticated ways to eliminate duplicates in Kafka and Spark in some situations but in our use case of anomaly detection, we accept the possibility of duplicates. Duplicate messages coming into the real-time anomaly detector will, at worst, result in duplicate alert messages. In the batch job to train the model, duplicate records will have virtually no impact on the resulting model. If we actually did want to eliminate the impact of duplicates, we could simply drop the duplicates when reading the data. For instance, one line in Spark would accomplish this.

```
df = df.dropDuplicates(["uuid"])
```

Of course, this is not a trivial operation so you should be certain that it is important before doing this.

Latency and Throughput

This system as described is considered near real-time. The latency between an anomalous message being sent to Kafka and the alert being generated is about 5 seconds. The primary reason for this is that we are using Spark Streaming in mini-batches to perform the enrichment and Random Forest evaluation. For real-time applications that require sub-second latency, consider exporting your model to PMML and using a high-performance evaluator such as [Openscoring.io](#).

Each component of this system is designed to be horizontally and linearly scalable. The system's throughput can be increased linearly simply by adding more Kafka brokers, YARN Node Managers (for Spark), Flume agents, or ECS nodes.

Spark - Machine Learning Model Training

A summary of the the Spark batch job is provided in Part 1. This job is implemented as the Python script **batch_model_builder.py** and it is executed by PySpark. For our use case, it is important that our machine learning model consider all of the data from each of our data centers. For this reason, we will run this job at just one data center but it will read the enriched data that originated from each of our data centers. This job should run daily or weekly.

One of the advantages of using Spark is that the API for streaming is nearly identical to that of traditional Spark batch jobs. The first thing our Spark batch job does is read the enriched data that was placed in our ECS bucket by the Flume job at each data center.

```
df = sqlContext.read.json("viprfs://repbucket1.ns1.site1/tmp/ecs_iot_demo2/enriched_data/*/*")
```

Since ECS guarantees a consistent view of the data, this Spark job will see the entire data set up to the most recent flush by each of the Flume agents.

Next, the feature vector is created just as in our streaming job. The feature vector is then sent to our unsupervised Random Forest machine learning training algorithm. After waiting for a long time (approximately 3 hours when training on all 5 million records), it saves our model to the same replicated ECS bucket.

For the updated model to be used by the Spark Streaming job at each site, the Spark Streaming jobs will need to reload the model from the ECS bucket. This is not automated in the current implementation of this demo so the Spark Streaming jobs will simply need to be restarted.

Launching Spark

To launch this job, you can use `batch_model_builder.sh`. This executes the following:

```
$ spark-submit \
--master yarn \
--deploy-mode client \
--driver-memory 4g \
--num-executors 4 \
--executor-cores 2 \
--executor-memory 8g \
--packages com.databricks:spark-csv_2.10:1.4.0 \
batch_model_builder.py \
--input_data_path ${ENRICHED_DATA_PATH}/*/* \
--data_format json \
--model_path ${MODEL_PATH}
```

Increase the executor parameters as desired to increase the resources available to the job. This job will use all of the resources that you provide it.

Unsupervised Random Forest

Most successful machine learning algorithms for this purpose are based on decision trees. Decision trees are essentially a series of questions such as "is `src_bytes < 50`" and "is `duration < 3.5`". After each question, either another question is asked or a final value is returned as the prediction (anomaly or normal). The Random Forest algorithm uses a group (ensemble) of decision trees and aggregates the results with a simple majority vote (one tree, one vote!). Each tree in a Random Forest is trained on a random subset of the rows (samples) and columns (features).

Training a decision tree or an ensemble of them normally requires that the samples be labeled. The label is simply whether the sample is an anomaly or normal. In our use case, we don't have labels so we move forward by assuming that all of the samples that we have ever seen are normal. We can then generate what we hope are anomalous samples by taking the value for field 1 from a random sample of our data, taking the value for field 2 from a different random sample of our data, and repeating this for each field and each new anomalous sample that we want to generate. This works surprisingly well.

The function to generate the synthetic anomalous samples is `supervised2unsupervised()` and is from [Getting Started With Apache Spark](#).

The code to perform the model training is as follows:

```
unsupervised_forest = supervised2unsupervised(RandomForest.trainClassifier, fraction=0.1)
model = unsupervised_forest(features_rdd,
    numClasses=2,
    categoricalFeaturesInfo={},
    numTrees=10,
    featureSubsetStrategy='auto',
    impurity='gini',
    maxDepth=15,
    maxBins=50)
```

Hyperparameter Tuning

You will notice that there are several parameters that were passed to our model training function (numTrees, maxDepth, etc.). These are called *hyperparameters* since they are not directly based on our data. (The *parameters* are essentially the questions and predictions in our decision trees.) What are the best values of these hyperparameters? This is one of the more difficult questions to address in machine learning. The general approach is to try a variety of different values and then to pick the one set of values that gives you the best model. So you may try numTrees=100, maxDepth=30 for one pass and then numTrees=30, maxDepth=15 for another pass. If you wanted to try every possibility with numTrees={30,100,300}, maxDepth={15,20,25,30}, and maxBins={25,50} you would need to try 3*4*2=24 combinations. If each one took 3 hours, that's 3 days of processing!

Of course, you'll also need to quantify how good each model is. This is normally done by holding out 20% of the samples, training on the remaining 80% of the samples, and then comparing the model's prediction on the held out 20% to the actual values. This can even be done five times by holding out a different 20% of the samples. This is called 5-fold cross validation and it would bring us to 15 days of processing - time to buy more hardware!

For those interested and willing to do a lot more learning, the grid search method described above can be replaced or supplemented with a more intelligent optimization method such as one that uses [Yelp's MOE](#).

That said, the batch_model_builder.py script makes no attempt to tune the hyperparameters or to measure the quality of the model. That is left as an exercise for someone who has more hardware and/or time than me.

Interactive Machine Learning with Zeppelin

In addition to using [Apache Zeppelin](#) for building dashboards and exploring the data, as described in Part 1, it can also be used to perform machine learning tasks interactively. Since it supports PySpark, the code we type into PySpark can be easily copied to our Spark Streaming or Spark batch job. We could even put the code into a common Python library (e.g. common_pyspark.py) that is imported by multiple applications.

Load our machine learning model

```
model_path = 'viprfs://reputbucket1.nsl.site1/tmp/ecs_iot_demo2/model'
model = RandomForestModel.load(sc, model_path)
```

Finished

Took 8 seconds (outdated)

Build the feature vector from the data previously loaded

```
features_sdf = build_features_vector(messages_sdf).cache()
```

Finished

Took 1 seconds (outdated)

View the feature vector

```
show_pandas_dataframe(pd.DataFrame(features_sdf.map(lambda row: row.features.toArray()).take(5)))
```

Finished

Took 1 seconds (outdated)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	
0	0	1032	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	511	511	0	0	0	0	1.00	0.00	0	255	255	1.00	0.00	1	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	261	7	1	1	0	0	0.03	0.06	0	255	7	0.03	0.07	0	0	1	1	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	111	4	0	0	1	1	0.04	0.08	0	255	4	0.02	0.06	0	0	0	0	1	1
3	0	1032	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	509	509	0	0	0	0	1.00	0.00	0	255	255	1.00	0.00	1	0	0	0	0	0
4	0	1032	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	509	509	0	0	0	0	1.00	0.00	0	255	255	1.00	0.00	1	0	0	0	0	0

Apply the model to our feature vectors to detect anomalies

```
features_rdd = extract_features(features_sdf)
predictions_rdd = model.predict(features_rdd)
features_and_predictions_rdd = features_sdf.rdd.zip(predictions_rdd)
anomalies_rdd = features_and_predictions_rdd.filter(lambda x: x[1] <= 0).map(lambda x: x[0])
anomalies = anomalies_rdd.collect()
```

Finished

Took 56 seconds (outdated)

```
show_pandas_dataframe(pd.DataFrame([row.asDict() for row in anomalies]), 2)
```

Finished

	count	diff_srv_rate	dst_bytes	dst_host_count	dst_host_diff_srv_rate	dst_host_errror_rate	dst_host_same_src_port_rate	dst_host_s
0	21	0	471	84	0	0	0.01	1
1	1	0	256	255	0	0	0.00	1

References

- [GitHub Repository](#)
- [Getting Started With Apache Spark](#)
- [KDD Cup 1999 data set](#)
- [EMC Elastic Cloud Storage \(ECS\)](#)

Part 3

See [Part 3](#) of this blog series.

(2 ratings)

0 Comments

There are no comments on this post

© 2018 Dell Inc. or its subsidiaries. All Rights Reserved.