REAL TIME ANALYTICS WITH SPARK AND KAFKA

NIXON PATEL, Chief Data Scientist, Brillio LLC.

Predictive Analytics & Business Insights
September 15th, 2015
<table>
<thead>
<tr>
<th>Topic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Who Am I?</td>
<td>Kafka as Distributed, Fault tolerant RT Broker</td>
</tr>
<tr>
<td>What is Real-Time?</td>
<td>Real-Time Data Streaming Analytics Architecture</td>
</tr>
<tr>
<td>What is Spark?</td>
<td>What is Spark Streaming?</td>
</tr>
<tr>
<td>Spark Overview.</td>
<td>Spark Streaming Terminology.</td>
</tr>
<tr>
<td>Why Spark?</td>
<td>Discretized Stream Processing (DStream)</td>
</tr>
<tr>
<td>Why Spark Over Hadoop MapReduce?</td>
<td>Windowed DStream</td>
</tr>
<tr>
<td>How Does Spark Work?</td>
<td>Mapped DStream</td>
</tr>
<tr>
<td>What is Spark Streaming?</td>
<td>Network Input DStream</td>
</tr>
<tr>
<td>What is Kafka?</td>
<td>Architecture For RT Analytics with SS And Kafka</td>
</tr>
<tr>
<td>What is Mllib?</td>
<td>Technology Stack For Demo</td>
</tr>
<tr>
<td>What is Lambda Architecture?</td>
<td>Analytics Model</td>
</tr>
<tr>
<td>Data Flow in Lambda Architecture</td>
<td>Inferences Drawn With Tableau Dashboard</td>
</tr>
<tr>
<td>Spark Streaming as Real-Time Micro-Batching</td>
<td></td>
</tr>
</tbody>
</table>
WHO AM I?

- Chief Data Scientist @ Brillio
- SVP - Business Head Emerging Technologies (SMAC) @ Collabera 2014
- Also, A Serial & Parallel Entrepreneur 1994-2013

YantraSoft, A Speech Products Company 2009-2013
Yantrius, A Speech & ERP Solutions Company

YantraeSolar, A 5 MW Solar Photovoltaic plant to generate Electricity from Solar Energy 2009-20013


IIT BTech (Hons) - Chemical Engineering
NJIT MS Computer Science
Rutgers Master of Business and Science in Analytics--Ongoing
**WHAT IS “REAL TIME” (RT)?**

Like there is no true “Unstructured” data so there is no “RT”. Only “Near Real Time” (NRT)

NRT systems can respond to data as it receives it without persisting in a DB (Present)

“Present” could mean different for different scenarios—

- Options Trader it is in Milliseconds, Ecommerce Site it is Attention

---

- **ABILITY TO MAKE BETTER DECISIONS & TAKE MEANINGFUL ACTIONS AT THE RIGHT TIME**
- **DETECTING FRAUD WHILE SOMEONE IS SWIPING A CREDIT CARD**
- **TRIGGERING AN OFFER WHILE THE SHOPPER IS STANDING IN CHECKOUT LINE**
- **PLACING AN AD ON A WEBSITE WHILE SOMEONE IS READING A SPECIFIC ARTICLE**
- **COMBING & ANALYZING DATA SO YOU CAN TAKE RIGHT ACTION AT RIGHT TIME & RIGHT PLACE**
WHAT IS SPARK?

Brief Relevant Historical Information Of Spark

- 2002: Map Reduce @ Google
- 2004: Spark Paper by AMP Lab
- 2006: Spark 1.4.1 Release July 15, 2015
- 2008: Map Reduce Paper 2004
- 2010: 2006 Hadoop @ Yahoo
- 2012: Spark Paper by AMP Lab
- 2014: Apache Spark Top Level
- 2015: Spark 1.5 Preview Release in Databricks
## SPARK

### OVERVIEW

**Spark Stack**

<table>
<thead>
<tr>
<th>Spark SQL</th>
<th>Spark Streaming</th>
<th>Mlib (machine learning)</th>
<th>GraphX (graph)</th>
</tr>
</thead>
</table>

### Apache Spark

- **Spark SQL**
  - For SQL and unstructured data processing
- **MLib**
  - Machine Learning Algorithms
- **Spark Streaming**
  - Stream processing of live data stream

### Spark Stack

- **Spark SQL**
  - For SQL and unstructured data processing
- **MLib**
  - Machine Learning Algorithms
- **GraphX**
  - Graph Processing
- **Spark Streaming**
  - Stream processing of live data stream

### Runs Everywhere

Spark runs on Hadoop, Mesos, standalone, or in the cloud. It can access diverse data sources including HDFS, Cassandra, HBase, and S3.

You can run Spark using its standalone_cluster mode, on EC2, on Hadoop YARN, or on Apache Mesos. Access data in HDFS, Cassandra, HBase, Hive

### Speed

- Run programs up to 100x faster than Hadoop MapReduce in memory, or 10x faster on disk.
- Spark has an advanced DAG execution engine that supports cyclic data flow and in-memory computing.

### Ease of Use

- Write applications quickly in Java, Scala, Python, R.
- Spark offers over 80 high-level operators that make it easy to build parallel apps. And you can use it interactively from the Scala, Python and R shells.

### Generality

- Combine SQL, streaming, and complex analytics.
- Spark powers a stack of libraries including SQL and DataFrames, Mlib for machine learning, Graph X, and Spark Streaming. You can combine these libraries seamlessly in the same application.
WHY SPARK

Spark unifies Batch, Real-Time (Interactive) & Iterative apps into a single Framework

Spark’s lazy evaluation of lineage graph reduces wait states with better pipelining
Spark’s Optimized heap use of large memory spaces
Use of Functional Programming model makes creation and maintenance of apps simple
HDFS friendly, Unified Toolset, Pipelined Oriented

Limitations of MapReduce Model

- MR Programming Model is very Complex & has high overhead in launch of new M/R
- Performance bottlenecks as Streams needs multiple transformation
- Most Machine Learning Algorithms are iterative as multiple iterations improves results
- With Disk based approach each iteration’s output is written to disk
WHY SPARK OVER HADOOP MAPREDUCE?

Hadoop Execution Flow

<table>
<thead>
<tr>
<th>Input</th>
<th>MR1</th>
<th>Tuples (on Disk)</th>
<th>MR2</th>
<th>Tuples (on Disk)</th>
<th>MR3</th>
<th>Tuples (on Disk)</th>
<th>MR4</th>
<th>Output Data on Disk</th>
<th>Output Data on Disk</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data on Disk</td>
<td>MR1</td>
<td>Tuples (on Disk)</td>
<td>MR2</td>
<td>Tuples (on Disk)</td>
<td>MR3</td>
<td>Tuples (on Disk)</td>
<td>MR4</td>
<td>Output Data on Disk</td>
<td>Output Data on Disk</td>
</tr>
</tbody>
</table>

Hadoop MR Record

<table>
<thead>
<tr>
<th>Data Size</th>
<th>102.5 TB</th>
<th>Spark Record</th>
<th>Spark 1 PB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elapsed Time</td>
<td>72 mins</td>
<td>23 mins</td>
<td>234 mins</td>
</tr>
<tr>
<td># Nodes</td>
<td>2100</td>
<td>206</td>
<td>190</td>
</tr>
<tr>
<td># Cores</td>
<td>50400 physical</td>
<td>6592 virtualized</td>
<td>6080 virtualized</td>
</tr>
<tr>
<td>Cluster Disk throughput</td>
<td>3150 GB/s (est.)</td>
<td>618 GB/s</td>
<td>570 GB/s</td>
</tr>
<tr>
<td>Sort Benchmark Daytona Rules</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Network</td>
<td>Dedicated data center, 10Gbps</td>
<td>Virtualized (EC2) 10Gbps network</td>
<td>Virtualized (EC2) 10Gbps network</td>
</tr>
<tr>
<td>Sort rate</td>
<td>1.42 TB/min</td>
<td>4.27 TB/min</td>
<td>4.27 TB/min</td>
</tr>
<tr>
<td>Sort rate/node</td>
<td>0.67 GB/min</td>
<td>20.7 GB/min</td>
<td>22.5 GB/min</td>
</tr>
</tbody>
</table>
HOW SPARK WORKS

How Spark Works - SparkContext

Driver Program
sc=new SparkContext
rDD=sc.textfile("hdfs://...")
rDD.filter(…)
rDD.Cache

How Spark Works - RDD

Driver Program
sc=new SparkContext
rDD=sc.textfile("hdfs://...")
rDD.filter(…)
rDD.Cache

RDD (Resilient Distributed Dataset)

• Fault Tolerant
• Controlled portioning to optimize data placement
• Manipulated Using rich set of operators

Storage Types:
- MEMORY _ONLY,
- MEMORY_AND_DISK,
- DISK_ONLY,
...

Partitions of Data
Dependencies between partitions

HOW SPARK WORKS - RDD Operations

• Create a new dataset from an existing one.
  Lazy in nature.
  Executed only when some action is performed.
  Example - Map(func)

• Returns a value or exports data after performing a computation.
  Example -
  - Count()
  - Reduce(func)
  - Collect
  - Take()

• Caching dataset in-memory for future operations.
  - Store on disk or RAM or mixed
  - Persist()
  - Cache()

• A big collection of data having following properties
  - Immutable
  - Lazy evaluate
  - Cacheable
  - Type Inferred

RDD- Resilient Distributed Dataset
WHAT ARE THE CHALLENGES OF A REAL TIME STREAMING DATA PROCESSING?

- RTSD Processing challenges are Complex
- Collection of RT events coming @ Millions/Second
- Needs events correlation using Complex Event Proc
- Needs to Handle events with Low Latency
- Besides Vol & Var needs to handle Vel & Ver
- Needs Parallel processing of data being collected
- Needs Fault Tolerance
- Needs to handle data in distributed environment
- RTSD Processing challenges are Complex

1. Flexible Data Input
2. Data Fall Over Snapshots
3. Real-time, fault-tolerant, scalable, in-stream processing
4. Guaranteed Data Delivery

http://preview.tinyurl.com/o983fmw

APPLICATION
NOQL DATABASE
HADOOP
RDBMS
Lambda Architecture

LA satisfies the needs for a robust system that is fault-tolerant, both against hardware failures and human mistakes, being able to serve a wide range of workloads and use cases, and in which low-latency reads and updates are required. The resulting system should be linearly scalable, and it should scale out rather than up.

* http://lambda-architecture.net/ (Nathan Marz)
DATA FLOW IN LAMBDA ARCHITECTURE

1. All data entering the system is dispatched to both the batch layer and the speed layer for processing.

2. The batch layer has two functions:
   - managing the master dataset (an immutable, append-only set of raw data),
   - to pre-compute the batch views.

3. The serving layer indexes the batch views so that they can be queried in low-latency, ad-hoc way.

4. The speed layer compensates for the high latency of updates to the serving layer and deals with recent data only.

5. Any incoming query can be answered by merging results from batch views and real-time view

   - Seems Like a Complex Solution...
   - Besides, there is a general notion that Real-Time Layer is prone to FAILURE!
     - Why?
     - Computation happens in memory
     - A system Crash will erase the state of the Real-Time System
       - Bad Deployment
       - Out Of Memory
     - Delayed upstream data will give inaccurate metrics
SPARK STREAMING AS RT MICRO-BATCHING SYSTEM COMES TO RESCUE

• Since the Real Time layer only compensates for the last few hours of data, everything the real-time layer computes is eventually overridden by the batch layer. So if you make a mistake or something goes wrong in the real-time layer, the batch layer will correct it. -Nathan Marz  
http://preview.tinyurl.com/nugnz6t  Big Data: Principles and best practices of scalable real-time data systems

• Spark Streaming, A Micro Batch Layer on top of Core Spark corrects the fault tolerance issue
• Spark Streaming Provides Efficient and fault-tolerant state full stream processing
• Integrates with Spark’s batch and interactive processing

Wait A minute...

• Has the issue raised in component #1 of the Lambda Architecture addressed?

Not Yet!!!!
APACHE KAFKA COMES FOR RESCUE FOR FAULT

This is very well resolved by “The Log” from Jay Kreps
http://preview.tinyurl.com/qc43s5j
COMPONENTS OF REAL TIME DATA STREAMING ANALYTICS ARCHITECTURE

Lambda Architecture Stack

1. Unified Log - Apache Kafka
2. Batch Layer - Hadoop for Storage
3. Serving Layer - MySQL, Cassandra, NoSQL or other KV Stores
4. Real-Time Layer - Spark Streaming
5. Visualization Layer -- Tableau
WHAT IS SPARK STREAMING?

- Extends Spark for doing large scale stream processing
- Scales to 100s of nodes and achieves second scale latencies
- Efficient and fault-tolerant state full stream processing

Integrates with Spark’s batch and interactive processing

- Provides a simple batch-like API for implementing complex algorithms

These batches are called Discrete Stream (DStreams)

Credit: http://spark.apache.org/

Spark Streaming

Kafka
Flume
HDFS
ZeroMQ
Twitter

HDFS
Databases
Dashboards
**SPARK STREAMING TERMINOLOGY**

**DStreams**

- A Sequence of Mini-Batches, where each mini-batch is represented as a Spark RDD
- Defined by its Input Source - Kafka, Twitter, HDFS, Flume, TCP, Sockets Akka Actor
- Defined by a time window called the Batch Interval
- Each RDD in Stream contains records

<table>
<thead>
<tr>
<th>Each batch of DStream is replicated as RDD</th>
<th>RDDs are replicated in cluster for fault tolerance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Each DS operation result in RDD transformation</td>
<td>There are APIs to access these RDDs directly</td>
</tr>
<tr>
<td>RDDs from Batch and Stream can be combined</td>
<td></td>
</tr>
</tbody>
</table>

**Streaming Flow**

- Input data stream
- Batches of input data
- Spark Streaming
- Spark Engine
- Batches of processed data

**DStream to RDD**

<table>
<thead>
<tr>
<th>RDD @ time 1</th>
<th>RDD @ time 2</th>
<th>RDD @ time 3</th>
<th>RDD @ time 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data from time 0 to 1</td>
<td>Data from time 1 to 2</td>
<td>Data from time 2 to 3</td>
<td>Data from time 3 to 4</td>
</tr>
</tbody>
</table>
Run a streaming computation as a series of very small, deterministic batch jobs.
- Chop up the live stream into batches of X seconds.
- Spark treats each batch of data as RDDs and processes them using RDD operations.
- Finally, the processed results of the RDD operations are returned in batches.
DSTREAM PROCESSING

Run a streaming computation as a series of very small, deterministic batch jobs
- Batch sizes as low as \( \frac{1}{2} \) second, latency ~ 1 second
- Potential for combining batch processing and streaming processing in the same system
EXAMPLE: WINDOWED DSTREAM

By default:
Window = slide = batch
duration

Window

Slide

0s 1s 2s 3s 4s 5s 6s 7s
EXAMPLE: WINDOWED DSTREAM

The resulting DStream consists of 3 seconds micro-batches. Each resulting micro-batch overlaps the preceding one by 1 second.
**EXAMPLE: MAPPED DSTREAM**

- **Dependencies:** Single parent DStream

- **Slide Interval:** Same as the parent DStream

- **Compute function for time t:** Create new RDD by applying map function on parent DStream’s RDD of time t

```scala
override def compute(time: Time): Option[RDD[U]] = {
  parent.getOrElseCompute(time).map(_ .mapU(mapFunc))
}
```

Gets RDD of time t if already computed once, or generates it

Map function applied to generate new RDD
The above architecture uses a combination of batch oriented and streaming layer as in the case of Lambda architecture.

For the batch oriented architecture Hive which is an abstraction on top of MapReduce is used.

Scikit-learn is used to create the model from the tweets in HDFS after the labelling of the tweets has been done.

For processing the streaming tweets, Spark framework is used.

The sentiment of the tweet is figured out in Spark using Skikit Python Library and the same is stored in Tableau Dashboard.

ARCHITECTURE FOR THE REAL TIME ANALYTICS WITH SPARK STREAMING & KAFKA

Batch Oriented Layer

Streaming Layer
TECHNOLOGY STACK USED FOR THE DEMO

Operating System

• Ubuntu 14.04 64-bit Server on Microsoft Azure

Languages

• Java 1.8
• Scala 2.10
• Python 2.7.6

Database

• MySQL 5.5.44

Big Data Software

• Apache Flume 1.6.0
• Apache Hive 1.2.1
• Apache Hadoop 1.2.1
• Kafka 2.10 0.8.2.1
• Apache Spark 1.4.1
The Kafka producer uses the Hosebird Client (hbc - https://github.com/twitter/hbc) developed by Twitter to pull the data for the topics of interest like the presidential candidates of 2016 elections. Hbc is Java HTTP client for consuming the Twitter’s Streaming API.

Once the tweets are pulled by Hbc, they are published to the Kafka topic where different subscribers can pull the tweets for further analysis.

The tweets are pulled and published in a real time. As soon as someone tweets on the relevant topics, the same is pulled by Hbc and published to the Kafka topic.
GETTING THE RELEVANT TWEET FOR 2016 PRESIDENTIAL ELECTIONS

Users tweet about politics, movies, sports and other interesting topics. On the StatusesFilterEndpoint the appropriate hashtags, keywords and Twitter User Id have to be specified as below to get the relevant tweets.

The StatusesFilterEndpoint is part of the Hbc API provided by Twitter. More details about the StatusesFilterEndpoint API can be found at https://goo.gl/hV89Sd.

The below are the hashtags and the keywords for which the Tweets are being pulled.

```java
```
//Create an instance of the Kafka Producer
Producer<String, String> producer = new Producer<String, String>(producerConfig);

//Create an instance of Kafka KeyedMessage which is passed to the Kafka Broker later
KeyedMessage<String, String> message = null;

try {
    //Populate the KeyedMessage with the topic, key and the message
    message = new KeyedMessage<String, String>(topic, "key", queue.take().trim());
} catch (InterruptedException e) {
    e.printStackTrace();
}

//Send the message from the producer to the broker
producer.send(message);
KAFKA BROKER

Batch Oriented Layer

Twitter → Kafka Producer → Kafka Broker

Kafka Broker → Flume → HDFS

HDFS → scikit-learn → model

Hive

Streaming Layer

Kafka Producer

ZK → Kafka Broker

ZK

scikit-learn

Tableau Dashboard

MySQL
• The Kafka Broker ([http://kafka.apache.org/](http://kafka.apache.org/)) provides a unified, high-throughput, low-latency platform for handling real-time data feeds. The design is heavily influenced by transaction logs. Kafka was developed by LinkedIn and is now a Top Level Project of the Apache Software Foundation.

• The previous slide mentions the Kafka producer. The Kafka consumers are the Flume and Spark Streaming in this scenario.
Flume is a distributed, reliable, and available service for efficiently collecting, aggregating, and moving large amounts of log data.

Flume has better connectivity to the Big Data ecosystem like HDFS and HBase when compared to Kafka, so it makes it easy to develop applications in Flume.

But, Flume doesn’t provide a reliable system when compared to Kafka. Kafka automatically replicates the events across machines and so changes of losing the data are minimized.

Flume is being configured to pull the messages from Kafka (http://flume.apache.org/FlumeUserGuide.html#kafka-source), pass them to the memory channel and the channel send it to the HDFS.

The next slide has the Flume configuration on how the different Flume components (Kafka Source, Memory Channel, HDFS Sink) are defined and chained together to the flow shown above.
FLUME CONFIGURATION FILE

# Sources, channels, and sinks are defined per agent name, in this case flume1.

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

# For each source, channel, and sink, set standard properties.

flume1.sources.kafka-source-1.type = org.apache.flume.source.kafka.KafkaSource
flume1.sources.kafka-source-1.zookeeperConnect = localhost:2181
flume1.sources.kafka-source-1.topic = twitter-topic
flume1.sources.kafka-source-1.batchSize = 100

flume1.sources.kafka-source-1.channels = memory-
# Other properties are specific to each type of source, channel, or sink. In this case, we specify the capacity of the memory channel.

flume1.channels.memory-channel-1.type = memory
flume1.channels.memory-channel-1.capacity = 10000

flume1.channels.memory-channel-1.transactionCapacity = 100

#Specify the HDFS Sink properties

flume1.sinks.hdfs-sink-1.channel = memory-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 = tweets
flume1.sinks.hdfs-sink-1.hdfs.useLocalTimeStamp = true
flume1.sinks.hdfs-sink-1.hdfs.path = hdfs://localhost:9000/user/analysis/tweets
flume1.sinks.hdfs-sink-1.hdfs.rollCount=10
flume1.sinks.hdfs-sink-1.hdfs.rollSize=0
flume1.sinks.hdfs-sink-1.hdfs.rollInterval=0
Batch Oriented Layer

Streaming Layer

HIVE
The tweets in HDFS are in JSON format. Hive which provides a SQL layer of abstraction is used to for analysis of the JSON tweets in a batch oriented fashion.

Hive out of the box can only understand csv, tsv and other types of data, but not the JSON data. So, a custom JSON (SerDe - Serializer Deserializer) has to be used for the same. A SerDe allows Hive to read the data from a table, and write it back to HDFS in any custom format.

https://cwiki.apache.org/confluence/display/Hive/SerDe

The queries through Hive are batch oriented in nature, so Spark Streaming has been used as discussed in the coming slides.
Spark streaming makes it easy to build scalable fault-tolerant streaming applications.

The main advantage of Spark streaming is that it lets reuse of the same code for batch processing as well as processing the streaming data.


The Spark Streaming program has been developed in Python and uses Scikit learn to figure out the sentiment of the tweet in a real time fashion and populate the same in the MySQL database.
#Create the Spark Context and Spark Streaming Context

```python
sc = SparkContext(master = "spark://Demo-Ubuntu:7077", 
appName="PythonStreamingKafkaWordCount")

ssc = StreamingContext(sc, 1)
```

#Get the tweets from Kafka, Spark Streaming is a consumer to the Kafka broker

```python
zkQuorum, topic = sys.argv[1:]

kvs = KafkaUtils.createStream(ssc, zkQuorum, "spark-streaming-consumer", 
{topic: 1})
```

#Figure out the sentiment of the tweet and write the same to MySQL

```python
kvs.map(lambda x: x[1]).map(classify_tweet).pprint()
```
EXAMPLE - GET HASHTAGS FROM TWITTER

\[
kvs = \text{KafkaUtils.createStream}(\text{ssc}, \text{zkQuorum}, "\text{spark-streaming-consumer}", \{\text{topic: 1}\})
\]

new DStream - a sequence of distributed datasets (RDDs) representing a distributed stream of data

Twitter Streaming API

tweets

DStream

batch \_at\_t

batch \_at\_t+1

batch \_at\_t+2

stored in memory as an RDD (immutable, distributed dataset)

Kafka Topic

ZooKeeper
EXAMPLE - GET HASHTAGS FROM TWITTER

kvs.map(lambda x: x[1]).map(classify_tweet).pprint()

Transformation: the classify_tweet python function calculates the sentiment of the tweet and writes to MySQL

tweets DStream

Tweet sentiment is calculated and put in MySQL database.
In order to classify tweets as positive or negative, we built a model using the Random Forest classifier in Python’s scikit-learn package.

- Used a publically available dataset of pre-classified tweets as our training data set.
- Extracted n-grams of the text (uni-grams, bi-grams and tri-grams), and created dummy variables from them, where 1 indicates that an n-gram is present in a tweet.
- Examples of some n-grams are on the right. Here, the count indicates the number of times a string appears.

Using this, we can compute the TF-IDF to select important words:

\[
TF(t) = \frac{\text{Number of times term } t \text{ appears}}{\text{Total number of terms in the document}}
\]

\[
IDF(t) = \log_e\left(\frac{\text{Total number of documents}}{\text{Number of documents with term } t \text{ in it}}\right)
\]
The data set contained around 1500 variables, out of which the Random Forest selected a maximum of 100 in each tree. There were 30 trees in total (this was the point at which the error rate converged).

- Split the data set in an 80:20 ratio for training and testing, and then trained the RF model on the 80% dataset. The 20% dataset was used for testing the results (sample is shown below).
- Using K-fold cross validation (with K=5), we...

<table>
<thead>
<tr>
<th>Sentiment</th>
<th>Negative</th>
<th>Positive</th>
<th>Count</th>
<th>Prediction Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Negative</td>
<td>3866</td>
<td>770</td>
<td>4636</td>
<td>83.4%</td>
</tr>
<tr>
<td>Positive</td>
<td>1261</td>
<td>4101</td>
<td>5362</td>
<td>76.5%</td>
</tr>
<tr>
<td>Count</td>
<td>5127</td>
<td>4871</td>
<td>9998</td>
<td></td>
</tr>
<tr>
<td>Result Accuracy</td>
<td>75.4%</td>
<td>84.2%</td>
<td></td>
<td>79.7%</td>
</tr>
</tbody>
</table>
RANKING CANDIDATES ON THEIR SIMILARITIES

1. A similarity matrix is a transposed n x n matrix of n candidates, where each candidate has a score with respect to another.

2. A candidate has the maximum score with himself, and a gradually decreasing score with candidates less like him/her.

3. The scoring used some of the following metrics:
   - Overall popularity of the candidate
   - The positive sentiment towards the candidate

<table>
<thead>
<tr>
<th>NodeName</th>
<th>Affiliate Party</th>
<th>Candidate1</th>
<th>Candidate2</th>
<th>Relationship</th>
<th>LineX</th>
<th>LineY</th>
<th>CircleY</th>
<th>Similarity_Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mark Everson</td>
<td>Republican</td>
<td>Mark Everson</td>
<td>Jim Gilmore</td>
<td>Mark Everson --&gt; Jim Gilmore</td>
<td>4600</td>
<td>6100</td>
<td>6100</td>
<td>3.14285714286</td>
</tr>
<tr>
<td>Ted Cruz</td>
<td>Republican</td>
<td>Ted Cruz</td>
<td>Donald Trump</td>
<td>Ted Cruz --&gt; Donald Trump</td>
<td>4100</td>
<td>6400</td>
<td>6400</td>
<td>4.0</td>
</tr>
<tr>
<td>Ted Cruz</td>
<td>Republican</td>
<td>Ted Cruz</td>
<td>Marco Rubio</td>
<td>Ted Cruz --&gt; Marco Rubio</td>
<td>4100</td>
<td>6400</td>
<td>6400</td>
<td>5.0</td>
</tr>
<tr>
<td>Chris Christie</td>
<td>Republican</td>
<td>Chris Christie</td>
<td>Chris Christie</td>
<td>Chris Christie --&gt; Chris Christie</td>
<td>5600</td>
<td>6600</td>
<td>6600</td>
<td>6.0</td>
</tr>
<tr>
<td>Rick Perry</td>
<td>Republican</td>
<td>Rick Perry</td>
<td>Mark Everson</td>
<td>Rick Perry --&gt; Mark Everson</td>
<td>5000</td>
<td>2300</td>
<td>2500</td>
<td>4.01851851852</td>
</tr>
<tr>
<td>Rick Perry</td>
<td>Republican</td>
<td>Rick Perry</td>
<td>Martin O'Malley</td>
<td>Rick Perry --&gt; Martin O'Malley</td>
<td>5000</td>
<td>2300</td>
<td>2500</td>
<td>5.11111111111</td>
</tr>
<tr>
<td>Rick Perry</td>
<td>Republican</td>
<td>Rick Perry</td>
<td>Jim Gilmore</td>
<td>Rick Perry --&gt; Jim Gilmore</td>
<td>5000</td>
<td>2300</td>
<td>2500</td>
<td>5.00255102041</td>
</tr>
<tr>
<td>Mike Huckabee</td>
<td>Republican</td>
<td>Mike Huckabee</td>
<td>Mark Everson</td>
<td>Mike Huckabee --&gt; Mark Everson</td>
<td>5300</td>
<td>3800</td>
<td>3800</td>
<td>4.01851851852</td>
</tr>
<tr>
<td>Mike Huckabee</td>
<td>Republican</td>
<td>Mike Huckabee</td>
<td>Jim Gilmore</td>
<td>Mike Huckabee --&gt; Jim Gilmore</td>
<td>5300</td>
<td>3800</td>
<td>3800</td>
<td>3.00255102049</td>
</tr>
<tr>
<td>Mike Huckabee</td>
<td>Republican</td>
<td>Mike Huckabee</td>
<td>George Pataki</td>
<td>Mike Huckabee --&gt; George Pataki</td>
<td>5300</td>
<td>3800</td>
<td>3800</td>
<td>4.5</td>
</tr>
<tr>
<td>Mike Huckabee</td>
<td>Republican</td>
<td>Mike Huckabee</td>
<td>Lincoln Chafee</td>
<td>Mike Huckabee --&gt; Lincoln Chafee</td>
<td>5300</td>
<td>3800</td>
<td>3800</td>
<td>3.5</td>
</tr>
<tr>
<td>Ted Cruz</td>
<td>Republican</td>
<td>Ted Cruz</td>
<td>Mark Everson</td>
<td>Ted Cruz --&gt; Mark Everson</td>
<td>4100</td>
<td>6400</td>
<td>6600</td>
<td>4.02222222222</td>
</tr>
<tr>
<td>Ted Cruz</td>
<td>Republican</td>
<td>Ted Cruz</td>
<td>Martin O'Malley</td>
<td>Ted Cruz --&gt; Martin O'Malley</td>
<td>4100</td>
<td>6400</td>
<td>6600</td>
<td>3.14285714286</td>
</tr>
</tbody>
</table>
1. The output of the analytic model, and the classification of tweets into positive and negative sentiments, are stored in a table on the database.

2. The sentiment matrix is also stored as another table on the database.

3. The two tables are linked by the Candidate’s name.

4. Whenever a new tweet is streamed in real-time, both tables are updated.
TABLEAU DASHBOARD

1. A connection is created between the database and Tableau, which gets refreshed in real-time when the database gets updated.

2. The front page gives an overview of tweet counts, tracks the daily number of tweets, most popular issues being discussed, and the number of tweets for each candidate.
Selecting Democrats filters the word cloud and candidate plots.

Clicking on the positive or negative portion of the party’s pie chart, will drill down each of the graphs.

Here, we select the Positives of the Democrats, and can see their daily progress, their most important issues, and how their candidates are faring.
TABLEAU DASHBOARD

• If we click on a word in the word cloud, we are able to view all candidates who are talking about that issue.

• We can even see the positive and negative reception of the candidate towards that issue.

Clicking on Jobs displays all candidates talking about Jobs.
The second page is network graph of candidates, where each one is linked to the other based on their similarity score.

The score ranges from 0 to 6, with the links varying from red to green.

Using the buttons on the top right, we can see the candidates that are most alike.

Display the similar candidate links. Here, Joe Biden is similar to Lincoln Chafee and Jim Webb, but dissimilar to Hillary Clinton.
Clicking on a candidates bubble, displays all the other candidates that are similar to him.
Similarly, we can change the view and see the candidates most unlike a particular candidate, by toggling the button on the top right. If we select Dissimilar candidates, and select Jeb Bush, we see his information displayed, and all the candidates most
THANK YOU

Predictive Analytics & Business Insights Team
All Attendees
Everyone on the Next slide....
REFERENCES AND CITATION FOR THIS PRESENTATION

My personal thanks to Praveen Sripati, Jim D’Souza, Sandeep Devagiri, Deepak Vinoth, Divya, Sanisha, Mayank Pant, Srinivas Guntur and rest of team at Brillio who made this presentation and demo possible.

3. databricks.com/blog/2014/11/05/spark-officially-sets-a-new-record-in-large-scale-sorting.html
8. jjmalina/pygotham-2015
11. http://www.slideshare.net/spark-project/deep-divewithsparkstreaming-tathagatadassparkmeetup20130617