

Real-Time Fraud Detection System

BIG DATA PROJECT

Under the noble guidance of Prof. David Belanger

Mukunth Rajendran



Stevens Institute of Technology
Hoboken, NJ 07307, USA.

Table of Contents

- I. Motivation
- II. Objective
- III. Introduction
 - Roadmap
 - Business Implication
- IV. Data Pipeline
 - 1. AWS Cloud Platform
 - 2. Python Flask API
 - 3. Kafka
 - 4. Model Building
 - 5. Spark Streaming
 - 6. Cloud watch
- V. Performance Evaluation
- VI. Analysis of Results
- VII. Conclusion and future work

Motivation

This project focuses on building a real-time fraud detection product. The federal trade commission estimates that 10 million people are victimized by credit card theft each year. These fraudulent cases require a lot of resources and time involvement. Often credit card companies end up paying for these losses. The aim is to build a product that diminishes these frauds. As it is rightly said, "Prevention is better than cure". This system follows the same policy.

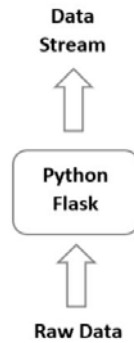
Objective

This project aims to build a scalable, multisource stream processing system with parallel processing capabilities to predict fraudulent transactions, acquire system information, and to deliver visualization of data.

Introduction

Data

The dataset is from Kaggle, it has 2 days of transactional data which is approximately 285000 records. It has 0.17% of fraudulent cases. 0.17% that is 492 records looks comparatively a small number, but don't underestimate these numbers as the loss per fraud case could be huge. The data has 28 anonymous attributes because of Banks' privacy issues. It has Amount attribute which shows the amount per transaction and Time attribute which is seconds elapsed between each transaction and the first transaction in the data. The data also has a class attribute which shows if the transaction was fraud or not.



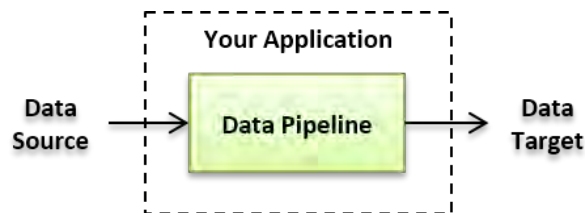
To mimic the real-time streaming data, python flask has been used to push the data to an API and hence making a continuous data stream which is further processed by the system. To further process the stream, micro batching technique is done with a window size of 1 sec. This will also help the system to support high velocity and high volume data.

Business Implication

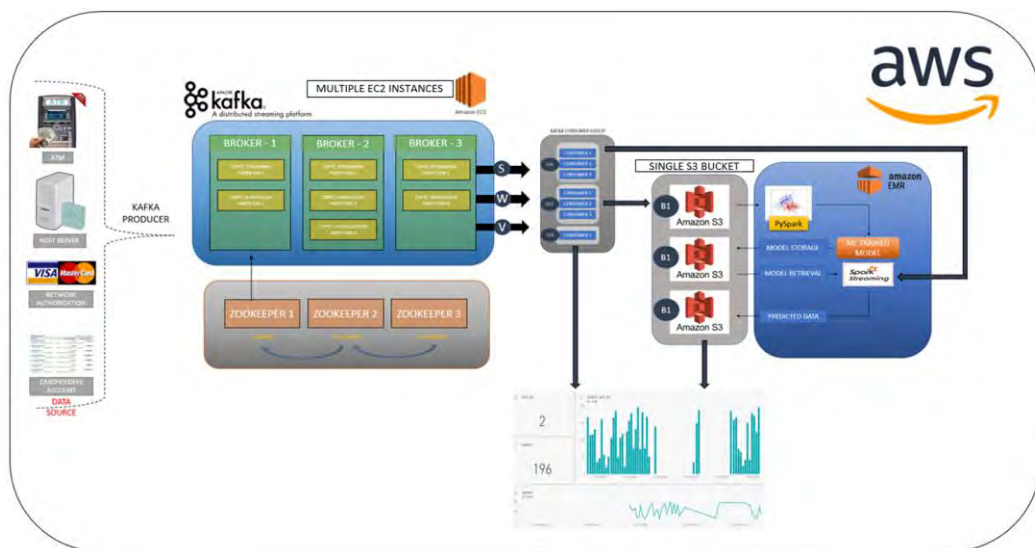
Every year, banks lose approximately 50 billion dollars towards dealing with fraudulent credit card activities. This system is designed to eliminate the usage of resources to deal with such cases. This is done by running the input values of a transaction against a machine learning model to retrieve results instantaneously while putting the transaction on hold. A confirmation request is sent to the customer in case it is predicted to be a fraud, if the customer confirms that the transaction was made by them, then the transaction is completed. With the implementation of this system, it can improve the customers' loyalty to the bank because the customer need not be worried about any unapproved transactions taking place in case their credit card was stolen. Moreover, when the components are hosted on AWS, the banks can pay based on their usage and allocate resources based on their requirements. As a result of the near real-time prediction, a notification is to be sent as a contingency plan in such cases. This scalable, high processing system can be used to aggregate multiple information sources and perform different operations based on the source. This being an architecture focussed project, is versatile and can take any type of data as the input and can perform the analysis.

Data Pipeline

A Data Pipeline consists of data processing elements connected in a series to perform set of actions, usually elements of the data pipeline are executed in parallel and in a time sliced manner. It automates the series of the processing and help us take decisions in real time using streaming data analysis.



Data flowing from different sources pass through data pipeline where the data is processed, replicated, stored, analyzed, and utilized by different consumers. In the prevailing market situation it is important to analyse and predict events with the data in real time and data pipeline helps us with it. Data pipeline helps us to gather all the data in one place in the same format. It also reduces the time and efforts involved in doing all the processes again for the future data, making it an automated process. Additionally, it can be reproducible later.



AWS Cloud Platform

The entire data architecture was built on the AWS cloud platform. For this project we used paid services like multiple EC2 instances, snapshots, volumes, EMR clusters with 1 master and 2 nodes and cloud watch.

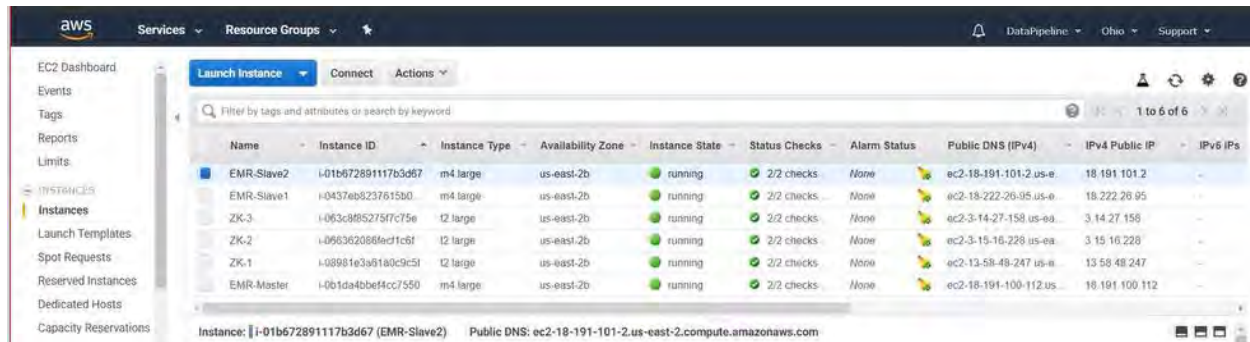
To begin with, a Flask API is created and used a serverless application - AWS lambda to create a Streaming API. The system mimics the data source with the help of Streaming data API, as shown in the image below. A row from the dataset, which is in JSON format is updated every 1 second.



A WebSocket protocol is used, providing a fully duplex communication channel over a single TCP protocol. It begins with a handshake between the API and the Kafka Producer, once the connection is setup, the kafka producer listens to the data from the URL. It is a Bidirectional communication, where the data can flow in both the directions.

The Kafka Producer was coded in Java. It listens to the streaming API once the connection is initiated and stops listening if the connection is disconnected.

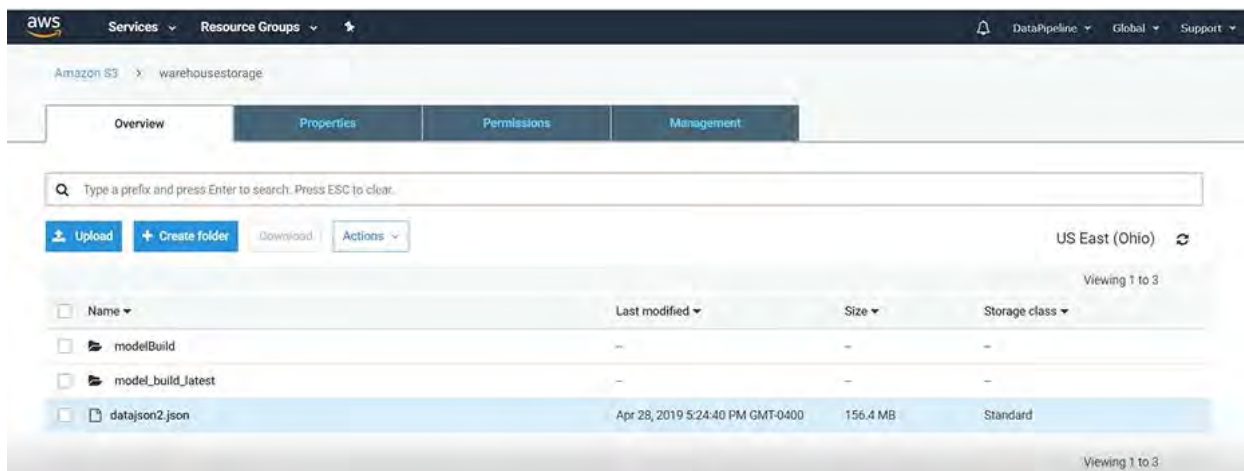
EC2 & EMR instances



The screenshot shows the AWS Management Console with a list of instances. The table includes columns for Name, Instance ID, Instance Type, Availability Zone, Instance State, Status Checks, Alarm Status, Public DNS (IPv4), IPv4 Public IP, and IPv6 IPs. The instances listed are EMR-Slave2, EMR-Slave1, ZK-3, ZK-2, ZK-1, and EMR-Master, all in a running state.

Name	Instance ID	Instance Type	Availability Zone	Instance State	Status Checks	Alarm Status	Public DNS (IPv4)	IPv4 Public IP	IPv6 IPs
EMR-Slave2	i-01b67289117b3d67	m4.large	us-east-2b	running	2/2 checks	None	ec2-18-191-101-2.us-e...	18.191.101.2	-
EMR-Slave1	i-0437eb8237615b0	m4.large	us-east-2b	running	2/2 checks	None	ec2-18-222-26-95.us-e...	18.222.26.95	-
ZK-3	i-063c895275f7c75e	t2.large	us-east-2b	running	2/2 checks	None	ec2-3-14-27-158.us-ea...	3.14.27.158	-
ZK-2	i-066362088aed1c6f	t2.large	us-east-2b	running	2/2 checks	None	ec2-3-15-16-228.us-ea...	3.15.16.228	-
ZK-1	i-08981e3a91a0c9c5f	t2.large	us-east-2b	running	2/2 checks	None	ec2-13-58-49-247.us-e...	13.58.49.247	-
EMR-Master	i-0b1da4bbe4cc7550	m4.large	us-east-2b	running	2/2 checks	None	ec2-18-191-100-112.us...	18.191.100.112	-

S3 Bucket



The screenshot shows the Amazon S3 console for a bucket named 'warehousestorage'. It displays a list of objects with columns for Name, Last modified, Size, and Storage class. The objects listed are 'modelBuild', 'model_build_latest', and 'datajson2.json'.

Name	Last modified	Size	Storage class
modelBuild	-	-	-
model_build_latest	-	-	-
datajson2.json	Apr 28, 2019 5:24:40 PM GMT-0400	156.4 MB	Standard

Kafka

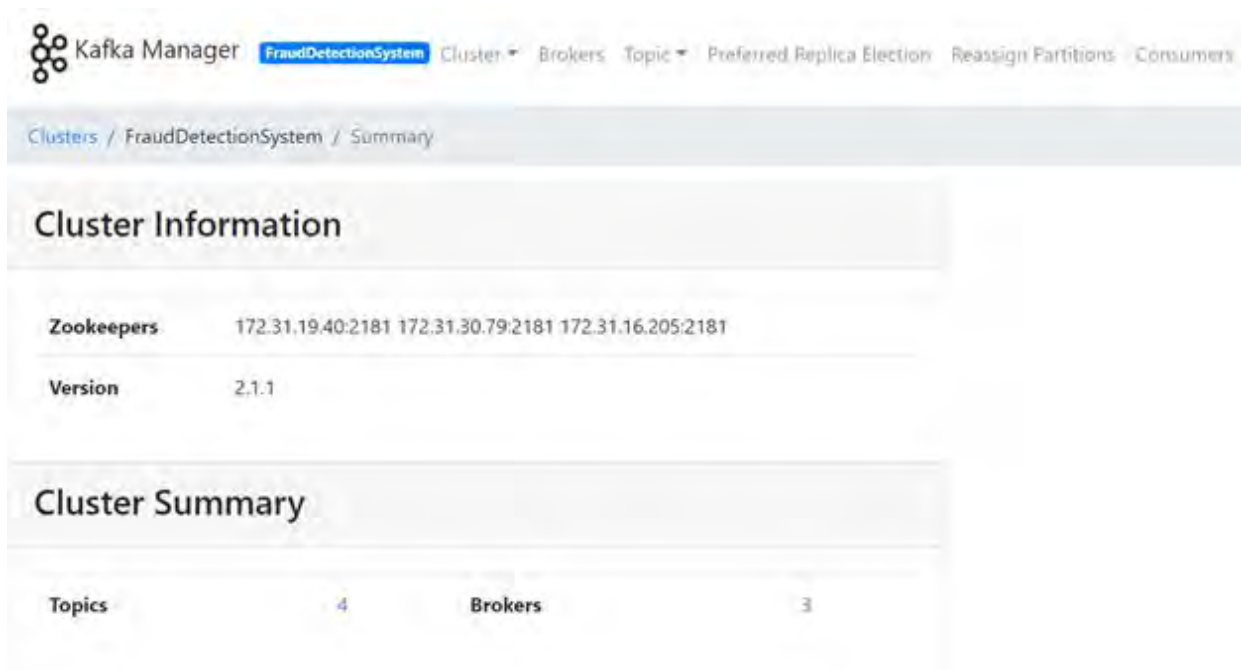
Kafka is used for building real-time data pipelines and streaming apps. It can horizontally scale, fault-tolerant, wicked fast and runs in production in thousands of companies. Kafka consists of Kafka producers, brokers, topics, partitions, and finally, Kafka consumers.

Kafka can't run without a Zookeeper, which is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services.

Zookeeper is implemented in the data pipeline in each EC2 instance. Combining these Zookeepers in multiple instance with each other is called "Ensemble of Zookeeper". If a leader - Zookeeper fails, a new leader is elected from the ensemble of zookeeper.

Kafka Manager and Kafka tools were used to get the GUI and understand the structure of the Kafka and it was installed on the AWS EC2 instances.

This Data pipeline was built towards providing fault tolerance and real time streaming data analysis, while handling the back pressure. Multiple brokers and zookeepers were used to handle any failure.



The screenshot displays the Kafka Manager web interface. At the top, the 'Kafka Manager' logo is followed by the cluster name 'FraudDetectionSystem' in a blue box. Navigation links include 'Cluster', 'Brokers', 'Topic', 'Preferred Replica Election', 'Reassign Partitions', and 'Consumers'. Below the navigation bar, a breadcrumb trail shows 'Clusters / FraudDetectionSystem / Summary'. The main content area is divided into two sections: 'Cluster Information' and 'Cluster Summary'. The 'Cluster Information' section contains a table with two rows: 'Zookeepers' with three IP addresses (172.31.19.40:2181, 172.31.30.79:2181, 172.31.16.205:2181) and 'Version' with the value '2.1.1'. The 'Cluster Summary' section shows a table with two rows: 'Topics' with the value '4' and 'Brokers' with the value '3'.

Cluster Information	
Zookeepers	172.31.19.40:2181 172.31.30.79:2181 172.31.16.205:2181
Version	2.1.1

Cluster Summary	
Topics	4
Brokers	3

Kafka producer listens to the streaming API, and consumes the updating data as messages and writes the message to a topic and the consumer listens to it. The Kafka System consist of multiple brokers running on multiple EC2 instance on AWS cloud.

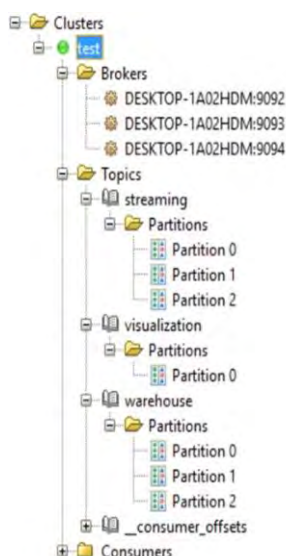
Brokers					
Id	Host	Port	JMX Port	Bytes In	Bytes Out
1	ec2-18-220-202-185.us-east-2.compute.amazonaws.com	PLAINTEXT:9092	-1	0.00	0.00
2	ec2-18-222-125-2.us-east-2.compute.amazonaws.com	PLAINTEXT:9092	-1	0.00	0.00
3	ec2-18-216-183-37.us-east-2.compute.amazonaws.com	PLAINTEXT:9092	-1	0.00	0.00

Initially, when a topic is created it is spread across all the brokers. The below image shows that “Streaming” topic is spread across multiple EC2 instances even though it was created in a single EC2 instance.

```

ec2-user@ip-172-31-19-40:~/kafka_2.11-2.1.0
[ec2-user@ip-172-31-19-40 kafka_2.11-2.1.0]$ bin/kafka-topics.sh --zookeeper 172.31.19.40:2181 --list
Streaming
[ec2-user@ip-172-31-19-40 kafka_2.11-2.1.0]$ bin/kafka-topics.sh --zookeeper 172.31.30.79:2181 --list
Streaming
[ec2-user@ip-172-31-19-40 kafka_2.11-2.1.0]$ bin/kafka-topics.sh --zookeeper 172.31.16.205:2181 --list
Streaming

```



Created multiple topics for individual streams of data - Streaming, Warehouse & Visualization. Streaming & Warehouse topic has 3 partitions and 3 replication factors, as our system works towards parallel processing. Visualization topic has only 1 partition and 1 replication factors and the stream is directed to PowerBI. There's another topic called 'Cosumer_offsets' which consists of the last data point, it helps the consumers in the consumer group to read from the last data point if the connection gets disconnected in between.

The image shows Partitions and Replication factors of the Streaming topic

Streaming

Topic Summary

Replication	3
Number of Partitions	3
Sum of partition offsets	0
Total number of Brokers	3
Number of Brokers for Topic	3
Preferred Replicas %	66
Brokers Skewed %	0
Brokers Leader Skewed %	33
Brokers Spread %	100
Under-replicated %	0

Operations

Delete Topic

Reassign Partitions

Generate Partition Assignments

Add Partitions

Update Config

Manual Partition Assignments

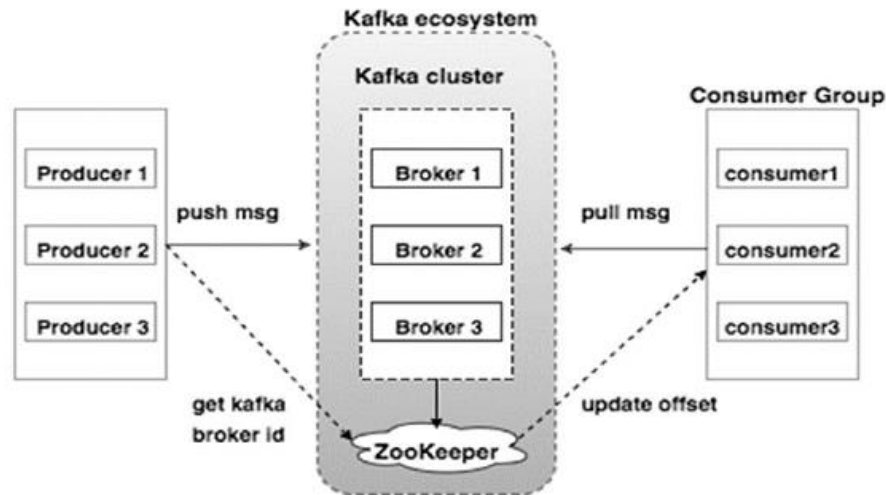
Partitions by Broker

Broker	# of Partitions	# as Leader	Partitions	Skewed?	Leader Skewed?
1	3	0	{0,1,2}	false	false
2	3	2	{0,1,2}	false	true
3	3	1	{0,1,2}	false	false

Created 3 Kafka topics for 3 different streams

```
ec2-user@ip-172-31-19-40: ~/kafka_2.11-2.1.0
[ec2-user@ip-172-31-19-40 kafka_2.11-2.1.0]$ bin/kafka-topics.sh --create --zookeeper 172.31.19.40:2181 --replication-factor 1 --partitions 1 --topic visualization
Created topic "visualization".
[ec2-user@ip-172-31-19-40 kafka_2.11-2.1.0]$ bin/kafka-topics.sh --zookeeper 172.31.16.205:2181 --list
Streaming
visualization
warehouse
```

Producer sends messages to Kafka in the form of messages. A message is a key-value pair. It contains the topic name and partition number to be sent. Kafka broker keeps records inside topic partitions. Records sequence is maintained at the partition level. Our system pushes data in a round robin manner to all the partitions to process the data parallelly.



This project uses kafka as the message queuing system, which publishes messages to the topics and their respective partitions, and the consumer subscribes to these partitions.

Model Building

The credit card transaction dataset is first converted into a streaming input using Python Flask and the data is published to the warehousing topic in Kafka. This warehousing stream is redirected to a storage in S3 through a Kafka consumer group that receives the data in the particular schema. The data table is stored in s3 using object storage and is continuously updated. The stored data in AWS S3 is used to build the prediction model. The incoming data is highly unbalanced with only 0.17% of fraudulent cases. To create an accurate model, the False Negative value should be minimized and the number of fraudulent cases predicted should be maximized. Accuracy is also an important factor in deciding the best model to use for predicting the streaming data. Machine learning models such as Logistic regression, Random Forest, Decision Trees and boosting algorithms like XGBoost followed by a tuned boosting model. The hyperparameters are tuned using GridSearchCV. Since the data is unbalanced, boosting algorithms tend to produce better results.

The models built are tested with the dataset by splitting the dataset into train and test sets and the accuracies were compared. The best results were obtained from the tuned

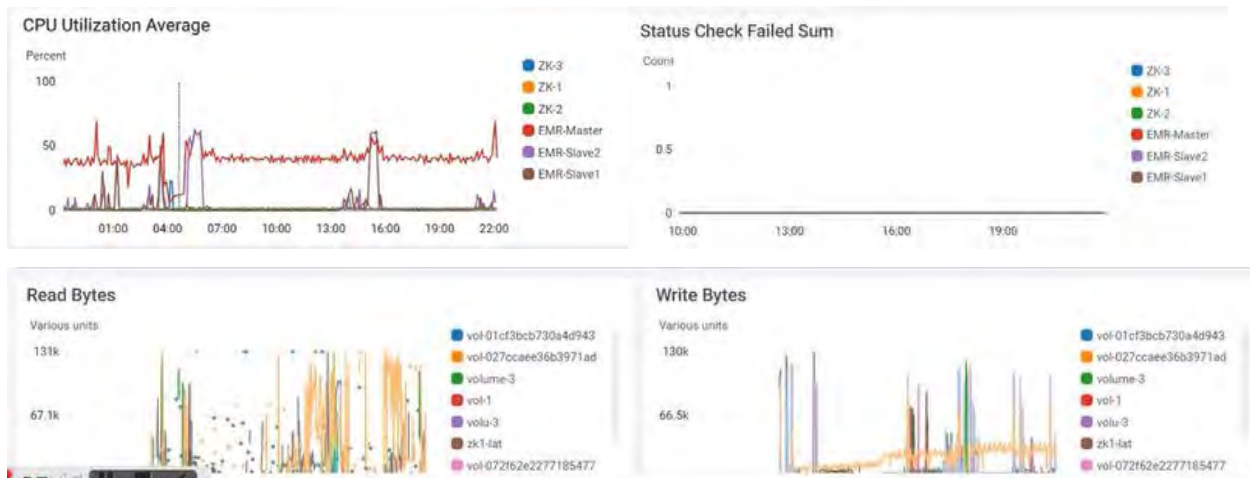
boosting model which is saved to AWS S3 after the training set is fitted. The code is written in PySpark that performs feature engineering and generates two new features: labels and features. This is obtained by applying OneHotEncoding to the dataset. After this process is done, different algorithms are applied. To obtain the false negative (FN) value, SparkSQL is used to run queries on the resulting dataset.

Streaming

The second Kafka consumer group gets the data from the streaming topic and loads the model that is stored in S3. The trained and stored model is applied to the incoming stream and the output is predicted using the model. The predicted output is then sent back to S3 for storage and further analysis. The schema for the incoming data is specified in this consumer group and the received data is decoded. After the decoding process, the data is converted to a SparkSQL dataframe consisting of only one row and the model is applied. The result is appended to the end of the row on the column named prediction and the probability is also provided. The streaming prediction is run on AWS EMR using a Spark Streaming program and is run on multiple clusters in parallel to improve the throughput. Amazon ElasticMapReduce (EMR) is used as a substitute for in-house cluster computing methods that run on local systems. Both the consumers are run on EMR instances and the model can be retrained when the accuracy reduces. The model is not trained for every new input in order to make the system time efficient. The model built can be retrained when required and the updated model can be overwritten at anytime.

CloudWatch

This system runs on AWS Cloud and the performance of the entire architecture was monitored using Cloud watch, in case the requirements of the entire system was more than or less than the current specification these metrics help to upgrade or degrade the web services.



Performance Metrics

Kafka was tested by pushing 50 records over a period of time to calculate the latency

```
[ec2-user@ip-172-31-19-40 kafka 2.11-2.1.0]$ bin/kafka-producer-perf-test.sh --topic streaming --num-records 1000 --throughput 10 --producer-props bootstrap.servers=172.31.19.40:9092,172.31.30.79:9092,172.31.16.205:9092 key.serializer=org.apache.kafka.common.serialization.StringSerializer value.serializer=org.apache.kafka.common.serialization.StringSerializer --record-size 1
52 records sent, 10.2 records/sec (0.00 MB/sec), 6.4 ms avg latency, 178.0 max latency.
51 records sent, 10.2 records/sec (0.00 MB/sec), 2.0 ms avg latency, 4.0 max latency.
50 records sent, 10.0 records/sec (0.00 MB/sec), 1.9 ms avg latency, 3.0 max latency.
50 records sent, 10.0 records/sec (0.00 MB/sec), 1.9 ms avg latency, 4.0 max latency.
50 records sent, 10.0 records/sec (0.00 MB/sec), 1.9 ms avg latency, 3.0 max latency.
50 records sent, 10.0 records/sec (0.00 MB/sec), 1.8 ms avg latency, 3.0 max latency.
50 records sent, 10.0 records/sec (0.00 MB/sec), 1.8 ms avg latency, 3.0 max latency.
50 records sent, 10.0 records/sec (0.00 MB/sec), 1.7 ms avg latency, 2.0 max latency.
51 records sent, 10.2 records/sec (0.00 MB/sec), 1.6 ms avg latency, 4.0 max latency.
50 records sent, 10.0 records/sec (0.00 MB/sec), 1.8 ms avg latency, 3.0 max latency.
50 records sent, 10.0 records/sec (0.00 MB/sec), 1.7 ms avg latency, 3.0 max latency.
50 records sent, 10.0 records/sec (0.00 MB/sec), 1.6 ms avg latency, 3.0 max latency.
50 records sent, 10.0 records/sec (0.00 MB/sec), 1.5 ms avg latency, 3.0 max latency.
50 records sent, 10.0 records/sec (0.00 MB/sec), 1.4 ms avg latency, 2.0 max latency.
50 records sent, 10.0 records/sec (0.00 MB/sec), 1.3 ms avg latency, 2.0 max latency.
50 records sent, 10.0 records/sec (0.00 MB/sec), 1.3 ms avg latency, 2.0 max latency.
1000 records sent, 10.004702 records/sec (0.00 MB/sec), 1.37 ms avg latency, 179.00 ms max latency, 2 ms 50th, 3 ms 95th, 3 ms 99th, 179 ms 99.9th.
```

Various other details about the kafka was tested pushing 100 messages

```
[ec2-user@ip-172-31-19-40 kafka 2.11-2.1.0]$ bin/kafka-consumer-perf-test.sh --topic streaming --broker-list 172.31.19.40:9092,172.31.30.79:9092,172.31.16.205:9092 --messages 100
start.time, end.time, data.consumed.in.MB, MB.sec, data.consumed.in.mMsg, mMsg.sec, rebalance.time.ms, fetch.time.ms, fetch.MB.sec, fetch.mMsg.sec
2019-04-27 18:48:04:850, 2019-04-27 18:48:05:165, 0.0005, 0.0015, 500, 1072.6592, 15, 253, 0.0019, 1584.1270
[ec2-user@ip-172-31-19-40 kafka 2.11-2.1.0]$ bin/kafka-consumer-perf-test.sh --topic streaming --broker-list 172.31.19.40:9092,172.31.30.79:9092,172.31.16.205:9092 --messages 10000
start.time, end.time, data.consumed.in.MB, MB.sec, data.consumed.in.mMsg, mMsg.sec, rebalance.time.ms, fetch.time.ms, fetch.MB.sec, fetch.mMsg.sec
WARNING: Exiting before consuming the expected number of messages: Timeout (110500 ms) exceeded. You can use the --Timeout option to increase the timeout.
2019-04-27 18:48:09:859, 2019-04-27 18:48:20:223, 0.0076, 0.0007, 4551, 439.1162, 12, 10342, 0.0007, 439.6252
```

Operation	Time (sec)
Read/Write	29.74

Model	Accuracy	False Negative	Processing time
Logistic regression	0.976	55	25.01
Decision Tree	0.746	64	53.99
Random Forest	0.977	40	61.27
XGBoost	0.986	40	69.89
GridsearchCV	0.978	32	1789.23

Conclusion

In the end, the framework is successfully consuming the stream, processing it and predicting correctly. It has been found that as Kafka having a multi-publisher/consumer architecture proves to be the best infrastructure for this model and AWS EMR as the core system for running up the application and analyzing the stream. S3 is used for data warehousing and saving the model simplifies the overall system.

Future Scope

In the future, other processing frameworks like Apache Flink and Apache Storm can be used to process data in real-time and Apache Beam can be used to provide a unified model that performs preprocessing. The performance of the system can be improved using multithreading and reactive programming for further optimization.

Appendix

Creating a streaming data API using Python Flask

sampleA.py x

```
2  import json
3  from multiprocessing import Value
4  from flask import Response
5  from datetime import datetime
6  from time import sleep
7
8  counter = Value('i', 0)
9  app = Flask(__name__)
10
11  global index_add_counter
12  @app.route('/todo/api/v1.0/tasks', methods=['GET'])
13  def get_tasks():
14      def streamer():
15          while True:
16              with counter.get_lock():
17                  counter.value += 1
18                  print(counter.value)
19                  i = 0
20                  with open('creditcard.json') as json_file:
21                      data = json.load(json_file)
22                      for p in data:
23                          if(i == counter.value):
24                              yield "<p>{}</p>".format(p)
25
26                              # return jsonify({'tasks': p})
27                              i+=1
28      return Response(streamer())
29
30  @app.route("/time/")
31  def time():
32      def streamer():
33          while True:
34              yield "<p>{}</p>".format(datetime.now())
35              sleep(1)
36
37      return Response(streamer())
38
39  if __name__ == '__main__':
40      app.run(debug=True)
```


Kafka producer is initiated in multiple brokers running on multiple EC2 instances.
Installed java on EC2 instances to run the producer code.

```
1 package simple.kafka.tut1;
2 import org.apache.kafka.clients.producer.*;
3 import org.apache.kafka.common.serialization.StringSerializer;
4 import org.json.JSONObject;
5 import org.slf4j.Logger;
6 import org.slf4j.LoggerFactory;
7
8 import java.io.*;
9 import java.util.HashMap;
10 import java.util.Iterator;
11 import java.util.Map;
12 import java.util.Properties;
13
14 public class producerdemowithcallback {
15
16     public static void main(String[] args) {
17         final Logger logger = LoggerFactory.getLogger(producerdemowithcallback.class);
18
19         // Create Producer Properties
20
21         Properties properties = new Properties();
22         properties.setProperty(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "172.31.19.40:9092,172.31.30.79:9092,172.31.16.205:9092");
23         properties.setProperty(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());
24         properties.setProperty(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());
25
26         // create the Producer
27
28         KafkaProducer<String, String> producer = new KafkaProducer<String, String>(properties);
29         InputStream inputStream = producerdemowithcallback.class.getResourceAsStream("data.json");
30         BufferedReader bufferedReader = new BufferedReader(new InputStreamReader(inputStream));
31
32         String jsonString = null;
33         try {
34             jsonString = bufferedReader.readLine();
35         } catch (IOException e) {
36             {}
37         }
38
39         Map<String, String> map = new HashMap<>();
40         JSONObject jsonObject = new JSONObject(jsonString);
41         Iterator<String> keys = jsonObject.keys();
42
43         while (keys.hasNext()) {
44             String key = (String) keys.next();
45             String value = jsonObject.getString(key);
46             map.put(key, value);
47         }
48
49         for (String key : map.keySet()) {
50             System.out.println("key: " + key + " Value: " + map.get(key));
51
52             final ProducerRecord<String, String> record =
53                 new ProducerRecord<String, String>("warehouse", map.get(key));
54             // send data asynchronous
55             producer.send(record, new Callback() {
56                 public void onComplete(RecordMetadata recordMetadata, Exception exception) {
57                     // executes everytime a record is sent - success or exception
58                     if (exception == null) {
59                         // Record sent
60                         logger.info("Received new metadata. \n" +
61                             "Topic:" + recordMetadata.topic() + "\n" +
62                             "Partition:" + recordMetadata.partition() + "\n" +
63                             "Offset:" + recordMetadata.offset() + "\n" +
64                             "Timestamp:" + recordMetadata.timestamp());
65                     } else {
66                         logger.error("Error while producing", exception);
67                     }
68                 }
69             });
70             // Flush and close
71             producer.flush();
72             producer.close();
73         }
74     }
75 }
```


Kafka producer listening to streaming API running in the localhost:5000



```
1 package com.rsmps;
2
3 import java.io.BufferedReader;
4
5 @SpringBootApplication
6 public class RsvpApplication {
7
8     private static final String STREAMING_ENDPOINT = "localhost:5000";
9
10    public static void main(String[] args) {
11        SpringApplication.run(RsvpApplication.class, args);
12    }
13
14    @Bean
15    public ApplicationRunner initializeConnection(ComputeData computeData) {
16        return args -> {
17            computeData.readAndSendData("data.json");
18        };
19    }
20 }
```

Zookeeper Configuration and kafka configuration with the corresponding instances and ports.

Kafka-Properties

```
dataDir=/tmp/kafka-logs-1
clientPort=2181
maxClientCnxns=0
initLimit=5
syncLimit=2
tickTime=2000
# list of servers
server.1=172.31.19.40:2888:3888
server.2=172.31.30.79:2888:3888
server.3=172.31.16.205:2888:3888
```

zk1

```
# The id of the broker. This must be set to a unique integer for each
broker.
broker.id=1
# A comma seperated list of directories under which to store log files
log.dirs=/tmp/kafka-logs-1
advertised.host.name=ec2-18-220-202-185.us-east-2.compute.amazonaws.com
log.dirs=/tmp/kafka-logs-1
# add all 3 zookeeper instances in here
zookeeper.connect=172.31.19.40:2181,172.31.30.79:2181,172.31.16.205:2181
zookeeper.connection.timeout.ms=6000
```

Installing the Kafka producer dependencies in Java.

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
5   <modelVersion>4.0.0</modelVersion>
6   <groupId>com.rsrvps</groupId>
7   <artifactId>CollectRsvpsAndSendToKafka</artifactId>
8   <version>0.0.1</version>
9   <packaging>jar</packaging>
10  <name>CollectRsvpsAndSendToKafka</name>
11  <description>Collecting RSVps from Meetup endpoint and send them to Kafka</description>
12  <parent>
13    <groupId>org.springframework.boot</groupId>
14    <artifactId>spring-boot-starter-parent</artifactId>
15    <version>2.0.2.RELEASE</version>
16    <relativePath />
17  </parent>
18  <properties>
19    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
20    <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
21    <java.version>1.8</java.version>
22    <spring-cloud.version>Finchley.BUILD-SNAPSHOT</spring-cloud.version>
23  </properties>
24  <dependencies>
25    <dependency>
26      <groupId>org.springframework.boot</groupId>
27      <artifactId>spring-boot-starter-websocket</artifactId>
28    </dependency>
29    <dependency>
30      <groupId>org.springframework.cloud</groupId>
31      <artifactId>spring-cloud-stream</artifactId>
32    </dependency>
33    <dependency>
34      <groupId>org.springframework.cloud</groupId>
35      <artifactId>spring-cloud-stream-binder-kafka</artifactId>
36    </dependency>
37    <dependency>
38      <groupId>org.springframework.boot</groupId>
39      <artifactId>spring-boot-starter-test</artifactId>
40      <scope>test</scope>
41    </dependency>
42  </dependencies>
43  <dependency>
44    <groupId>org.springframework.cloud</groupId>
45    <artifactId>spring-cloud-stream-test-support</artifactId>
46    <scope>test</scope>
47  </dependency>
48  <dependency>
49    <groupId>org.json</groupId>
50    <artifactId>json</artifactId>
51    <version>20180813</version>
52  </dependency>
53  <dependencyManagement>[]
54  <build>
55    <plugins>
56      <plugin>
57        <groupId>org.springframework.boot</groupId>
58        <artifactId>spring-boot-maven-plugin</artifactId>
59      </plugin>
60    </plugins>
61  </build>
62  <repositories>
63    <repository>
64      <id>spring-snapshots</id>
65      <name>Spring Snapshots</name>
66      <url>https://repo.spring.io/snapshot</url>
67      <snapshots>
68        <enabled>true</enabled>
69      </snapshots>
70    </repository>
71    <repository>
72      <id>spring-milestones</id>
73      <name>Spring Milestones</name>
74      <url>https://repo.spring.io/milestone</url>
75      <snapshots>
76        <enabled>false</enabled>
77      </snapshots>
78    </repository>
79  </repositories>
80  </project>

```

Saving and Importing the model from S3

```
In [47]: M cv.write().overwrite().save("s3://warehousestorage/model_build_latest_cv")
```

```
In [50]: M from pyspark.sql import SparkSession
from pyspark.ml.classification import LogisticRegressionModel

spark = SparkSession.builder.appName('model-build').getOrCreate()

#LOAD MODEL
model_load=CrossValidator.load("s3://warehousestorage/model_build_latest_cv")
```

Logistic Regression

```
In [32]: M from pyspark.ml.classification import LogisticRegression
model = LogisticRegression(featuresCol='features', labelCol='label')
model_fit = model.fit(train_xg)
predictions = model_fit.transform(test_xg)

In [33]: M from pyspark.sql import SQLContext
sqlContext = SQLContext(sc)

In [34]: M from pyspark.ml.evaluation import BinaryClassificationEvaluator
evaluator = BinaryClassificationEvaluator()
print('Accuracy', evaluator.evaluate(predictions))

('Accuracy', 0.9767626633911445)

In [35]: M sqlContext.registerDataFrameAsTable(predictions, "pred")
bad_vals=sqlContext.sql("SELECT COUNT(*) FROM pred WHERE Class=1 AND prediction=0")
bad_vals.show()
num_ones=sqlContext.sql("SELECT COUNT(*) FROM pred WHERE prediction=1")
num_ones.show()

+-----+
|count(1)|
+-----+
|      55|
+-----+
```

Decision Tree

```
In [36]: M from pyspark.ml.classification import DecisionTreeClassifier
dt = DecisionTreeClassifier(featuresCol = 'features', labelCol = 'label', maxDepth = 3)
dtModel = dt.fit(train_xg)
predictions = dtModel.transform(test_xg)

In [37]: M evaluator = BinaryClassificationEvaluator()
print('Accuracy', evaluator.evaluate(predictions))

('Accuracy', 0.7462866110757335)

In [38]: M sqlContext.registerDataFrameAsTable(predictions, "pred")
bad_vals=sqlContext.sql("SELECT COUNT(*) FROM pred WHERE Class=1 AND prediction=0")
bad_vals.show()
num_ones=sqlContext.sql("SELECT COUNT(*) FROM pred WHERE prediction=1")
num_ones.show()

+-----+
|count(1)|
+-----+
|      64|
+-----+
```

Random Forest

```
In [39]: M from pyspark.ml.classification import RandomForestClassifier
rf = RandomForestClassifier(featuresCol = 'features', labelCol = 'label')
rfModel = rf.fit(train_xg)
predictions = rfModel.transform(test_xg)

In [40]: M evaluator = BinaryClassificationEvaluator()
print('Accuracy', evaluator.evaluate(predictions))

('Accuracy', 0.9773283570279322)

In [41]: M sqlContext.registerDataFrameAsTable(predictions, "pred")
bad_vals=sqlContext.sql("SELECT COUNT(*) FROM pred WHERE Class=1 AND prediction=0")
bad_vals.show()
num_ones=sqlContext.sql("SELECT COUNT(*) FROM pred WHERE prediction=1")
num_ones.show()

+-----+
|count(1)|
+-----+
|      40|
+-----+
```

Gradient boosting

```
In [42]: M from pyspark.ml.classification import GBTClassifier
gbt = GBTClassifier(maxIter=10)
gbtModel = gbt.fit(train_xg)
predictions = gbtModel.transform(test_xg)

In [43]: M evaluator = BinaryClassificationEvaluator()
print('Accuracy', evaluator.evaluate(predictions))

('Accuracy', 0.9862111752094065)

In [44]: M sqlContext.registerDataFrameAsTable(predictions, "pred")
bad_vals=sqlContext.sql("SELECT COUNT(*) FROM pred WHERE Class=1 AND prediction=0")
bad_vals.show()
num_ones=sqlContext.sql("SELECT COUNT(*) FROM pred WHERE prediction=1")
num_ones.show()

+-----+
|count(1)|
+-----+
|      40|
+-----+
```

Parameter tuning

```
In [45]: M from pyspark.ml.tuning import ParamGridBuilder, CrossValidator
paramGrid = (ParamGridBuilder()
             .addGrid(gbt.maxDepth, [2, 4, 6])
             .addGrid(gbt.maxBins, [20, 60])
             .addGrid(gbt.maxIter, [10, 20])
             .build())
cv = CrossValidator(estimator=gbt, estimatorParamMaps=paramGrid, evaluator=evaluator, numFolds=5)
# Run cross validations. This can take about 6 minutes since it is training over 20 trees!
cvModel = cv.fit(train_xg)
predictions = cvModel.transform(test_xg)
evaluator.evaluate(predictions)

Out[45]: 0.9733658262714183

In [46]: M sqlContext.registerDataFrameAsTable(predictions, "pred")
bad_vals=sqlContext.sql("SELECT COUNT(*) FROM pred WHERE Class=1 AND prediction=0")
bad_vals.show()
num_ones=sqlContext.sql("SELECT COUNT(*) FROM pred WHERE prediction=1")
num_ones.show()

+-----+
|count(1)|
+-----+
|      32|
+-----+
```

Streaming code:

```
In [350]: M from pyspark.ml import Pipeline
from kafka import KafkaConsumer
consumer = KafkaConsumer('streaming', bootstrap_servers='172.31.30.79:9092')
for msg in consumer:
    # print(msg.value)
    varlatest = json.loads(msg.value)
    dflatest = sc.parallelize([varlatest])
    dflatest = sqlContext.createDataFrame(dflatest)
    # model load.transform(df).show()
    # print(type(dflatest))
    # dflatest.show()
    cols=dflatest.columns
    pipeline = Pipeline(stages = operations)
    pipelineModel = pipeline.fit(dflatest)
    dflatest = pipelineModel.transform(dflatest)
    selectedCols = ['label', 'features'] + cols
    dflatest = dflatest.select(selectedCols)
    predictions = model.load.transform(dflatest)
    predictions.show()
```