PDSS CW2

B174502

Real-Time Distributed Fraud Detection System

Input

```
case class Transaction(transactionID: String, fromID: String,  toID: String,  amount: Float,
                       timestamp: Long)
RDD[Transaction]
```

Where each entry represents a transaction that is potentially fraudulent. The `timestamp` is a `Long` and is represented as the number of seconds since the Unix epoch

Output

```
RDD[Transaction]
```

Where each entry is a subset of the input containing the fraudulent transactions in question with a confidence score

Solution We can decide whether a transaction is fraudulent based on the transaction amount and the frequency of transactions

1. Define entry function

```
def detect_fraudulent_transactions(transactionsRDD: RDD[Transaction]): RDD[Transaction]
```

2. Reduce the input RDD to calculate the sum and squared sums of transaction amounts

```
val transaction_amount_stats = transactionsRDD
                      .map(x => (x.amount, x.amount*x.amount, 1))
                      .reduce((x,y) =>(x._1 + y._1, x._2 + y._2, x._3 + y._3))
```

3. Calculate mean and standard deviation of transaction amounts

```
/// Obvious calculation
```

4. From each transaction, do a self-join (cartesian join) to get all combinations of transactions and filter to get a strict ordering of transactions (T1 < T2) and map to get the time between transactions

```
val transaction_time = transactionsRDD.map(x => (x.transactionId,  x.timestamp))
val transaction_time_diff = transaction_time.cartesian(transaction_time)
                      .filter(x => x._1._2 < x._2._2)
                      .map(x => (x._2._1, x._2._2 - x._1._2))
```

5. Count the number of transactions within a minute window

```
val transaction_time_diff_in_min = transaction_time_diff
.map{case x => if (x._2 < 60) {(x._1,1)} else {(x._1,0)}}
.reduceByKey((x,y) => x+y)
```

7. Calculate the mean and standard deviation of the count of transactions within 1-minute

```
val transaction_time_stats_mean = transaction_time_diff_in_min
.map(x => x._2)
.reduce((x,y) => x + y) / transaction_time_diff_in_min.count()

val transaction_time_stats_std = math.sqrt(
transaction_time_diff_in_min.map(x => (x._2 - transaction_time_stats_mean)*(x._2 - transaction_time_s
  .reduce((x,y) => x + y) / transaction_time_diff_in_min.count()
)
```

9. Determine out-of-distribution transaction frequency and transaction amounts

```
val fraud_time = transactionsRDD
.map(x => (x.transactionId, x)).join(transaction_time_diff_in_min)
.filter(x => x._2._2 > transaction_time_stats_mean + 1*transaction_time_stats_std)
.map(x => x._2._1)
val fraud_amount = transactionsRDD
.filter(x => x.amount > transaction_amount_mean + 1*transaction_amount_std)
val fraud = fraud_time.union(fraud_amount).distinct()
```

Optimisations

- Persist RDDs by caching intermediate RDDs that are used across calculations
- Partition `transactionsRDD` with `transactionID` prior to first operation and then the final joins to get the fraudulent transactions
- Use an alternative method than `cartesian` to avoid a large join, perhaps a custom join that only joins the transactions close together in timestamp
  - We can only use range partitioning to partition this dataset according to the timestamp

Task 2

To cross-reference suspicious activities we can consider both location history for transactions. This can be done by considering a directed graph to find and suspicious transactions

1. Initialize using the entire transaction history
   (a) Initialise a node for each user
   (b) Initialise edges as the transaction with their edge weights initialised with:
      i. Transaction Frequency
      ii. Transaction Amounts
   (c) Initialise a clustered location graph for each user
      i. Cluster user locations based on -means
      ii. Each edge is a transaction from two different locations
2. User and Transaction Pattern Detection
   (a) Cycle Detection
      i. Find cycles within the graph where they are completed within a short-time frame
   (b) Flow Analysis
      3. Money flow into specific accounts that are unusual
      4. Splitting and merging of money across nodes
   (c) Node Metrics
      i. Look at if transaction connects to known suspicious users

3. Location Pattern Detection
    (a) Whether the newly created transaction has a high confidence score for being added to a location cluster

Distributed Log Processing for Anomaly Detection

Input

```
case class LogEntry(timestamp: Long, eventType: String, processId: String, responseTime: Long)
RDD[LogEntry]
```

The input is an RDD where each entry is a log where they contain the timings of the log (in seconds since Unix epoch), the type of the log and the process that generated the log.

Output

We can output all `LogEntry` which we consider to the unusual or suspicious. Since we will be adopting a window-based aggregation approach. All logs within a specific window will be outputted

```
RDD[(LogEntry)]
```

Solution

We should detect whether there are anomalies in logs based on the following:

- Sudden spikes in error rates
- Request time degradation across the same process

1. Define the entry function

```
def detect_unusual_log_patterns(logs: RDD[LogEntry]): RDD[LogEntry]
```

2. Map the RDD to transform timestamps to its corresponding window

```
val windowed_logs = logs.map(x => (x.timestamp / WINDOW_SIZE, x))
```

3. Detect error rate spike anomalies
4. Aggregate error count in each window and so, calculate the error rate in each window

```
val error_counts = windowed_logs
    .map(x => (x._1, (if (x._2.eventType == "ERROR") 1 else 0, 1)))
    .reduceByKey((x,y) => (x._1 + y._1, x._2 + y._2)).mapValues(x => (x._1.toDouble / x._2.toDouble))
```

5. Calculate the mean and standard deviation of the error rate

```
val error_count_mean = error_counts.map(x => x._2).mean()
val error_count_std = error_counts.map(x => x._2).stdev()
```

6. Detect the out-of-distribution error rate logs

```
val unusual_error_logs = error_counts
      .filter(x => x._2 > error_count_mean + 2 * error_count_std)
      .join(windowed_logs).map(x => x._2._2)
```

7. Detect response time anomalies
8. Reduce by the process key and calculate the mean and standard deviation of response times for this process

```
val process_logs = logs
      .map(x => (x.processId, x)).groupByKey()
      .mapValues(x => {
        val mean = x.map(_.responseTime).reduce(_ + _) / x.size
        val std = math.sqrt(
          x.map(y => math.pow(y.responseTime - mean, 2)).reduce(_ + _) / x.size
        )
        (mean, std)
      })
```

9. Detect the out-of-distribution response-time logs for either high or low response-times

```
val unusual_process_logs = process_logs.filter(x => x._2._2 > x._2._1 * 0.5)
      .join(logs.map(x => (x.processId, x))).map(x => x._2._2)
val unusual_logs = unusual_error_logs.union(unusual_process_logs).distinct()
```

Optimisations

- Persist intermediate RDDs by saving it in memory using `.persist()`
- Range partition by timestamps so that we can efficiently fetch windowed data in order while not being explicitly hashable

Task 2

1. Prepare log entries to be used as a timeseries
    (a) Group into time-based windows
    (b) Aggregate error counts and types within the windows
    (c) Baseline statistics for each window
2. Detect spikes in error rates
    (a) Calculate rolling statistics from window to window for error rates
    (b) Use statistical methods to identify strong deviations in error rate
    (c) Flag windows that are sufficiently deviated from typical behaviour