Initial Input

Based on the design from the assignment document for implementing this IDS. The program shall first ingest 2 files, “Events.txt” and “Stats.txt”. Below is a snippet of what the file contains.

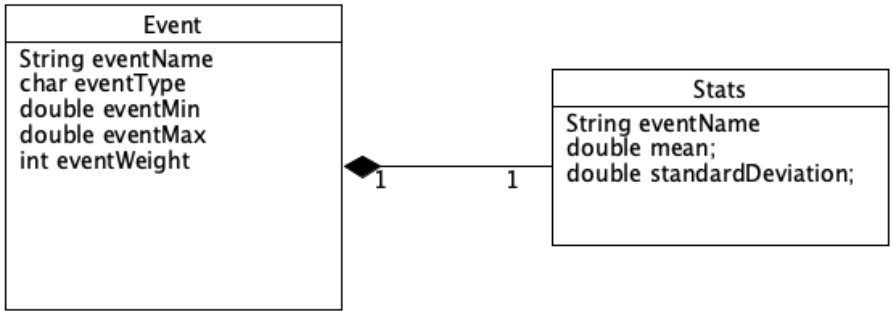
```
5
Logins:D:0::3:
Time online:C:0:1440:2:
Emails sent:D:0::1:
Emails opened:D:0::1:
Emails deleted:D:0::2:
```

Events.txt

```
5
Logins:4:1.5:
Time online:151:25.00:
Emails sent:10:3:
Emails opened:12:4.5:
Emails deleted:7:2.3:
```

Stats.txt

We can observe that in the events file, each row is an object of name, event type, min, max and weight, whist the stats file contains a corresponding event name with its corelated mean and standard deviation. Therefore, I decided to model 2 objects Events and Stats as plain old java objects as shown in the UML before.



The files can be passed as a program argument as such
java IDS Events.txt Stats.txt 5
and will be promoted with the log messages as shown below with each .toString() method for each class printed below for user verification as needed.

```
IDS is running...
Events Logged:
Event{eventName='Logins', eventType='D', eventMin=0.0, eventMax=0.0, eventWeight=3, stats=Stats{eventName='Logins', mean=4.0, standardDeviation=1.5}}
Event{eventName='Time online', eventType='C', eventMin=0.0, eventMax=1440.0, eventWeight=2, stats=Stats{eventName='Time online', mean=151.0, standardDeviation=25.0}}
Event{eventName='Emails sent', eventType='D', eventMin=0.0, eventMax=0.0, eventWeight=1, stats=Stats{eventName='Emails sent', mean=10.0, standardDeviation=3.0}}
Event{eventName='Emails opened', eventType='D', eventMin=0.0, eventMax=0.0, eventWeight=1, stats=Stats{eventName='Emails opened', mean=12.0, standardDeviation=4.5}}
Event{eventName='Emails deleted', eventType='D', eventMin=0.0, eventMax=0.0, eventWeight=2, stats=Stats{eventName='Emails deleted', mean=7.0, standardDeviation=2.3}}
```

Internally, the program will store them as an array of Events[ ] as each event consists of Stats.

```
Event[] initialEvents;
```

line 13: IDS.java

There can be potential inconsistencies such as having non-integer values for discrete events, 0 values for non-zero attributes like min,max or weights. For continuous events, a check for max value greater than 1440 is applied as there can't be more than 1440 seconds in a day. All the constraints were implemented in a method validateEvent( ).

```
public boolean validateEvent() {

    if (this.eventType == 'C') {
        if (this.eventMin < 0 || this.eventMax < 0 || this.eventWeight < 0) {
            return false;
        } else if (this.eventMin > this.eventMax) {
            return false;
        }
    } else if (this.eventType == 'D') {
        if(this.eventMax > 1440){
            return false;
        }
        if(intChecker(this.eventMin) == false
                || intChecker(this.eventMax) == false || intChecker(this.eventWeight) == false){
            return false;
        }
        if (this.eventMin < 0 || this.eventMax < 0 || this.eventWeight < 0) {
            return false;
        }
    } else {
        return false;
    }
    return true;
}
```

This method is then called for every event read from events.txt as show below.

```
for (Event e : initialEvents
) {
    if (e.validateEvent() == false) {
        System.out.println("Error in Event file.");
        System.exit( status: -1);
    }
```

Additional validation is added such as mismatch length in both files and invalid program parameters

```
    try {
        eventsFilePath = args[0];
        statsFilePath = args[1];
        days = args[2];
        baseDays = Integer.parseInt(days);
    } catch (Exception e) {
        System.out.println("Error in input parameters.");
        System.out.println("Parameters are as such: IDS Events.txt Stats.txt Days");
        System.exit( status: -1);
    }
    initialEvents = ReadEventTypesFile(eventsFilePath);
    initialStats = ReadEventStatsFile(statsFilePath);
    if (initialEvents.length != initialStats.length) {
        System.out.println("Error in length for both files.");
        System.exit( status: -1);
    }
```

Activity Simulation Engine and the Logs

2.1 Generating events approximately consistent with the particular distribution.

To ensure an event generation engine that can create logs that is consistent with the given distribution in Stats.txt, I utilized the Random library in java with the .nextGaussian() api. By multiplying the output of this method with the sum of the mean and standard deviation of the associated event, we can closely generate simulation logs that can fit in the stored distribution so long as the value is within the original constraints of the min and max values of an event. Additionally, if an event is discrete the value will be rounded to the next closest integer value. The implementation is as such:

```
4 usages  kikoken831
public double getValueWithinDistribution() {

    Random rand = new Random();
    while (true) {
        double value = stats.getMean() + stats.getStandardDeviation() * rand.nextGaussian();
        // round the value if it's discrete
        if (eventType == 'D') {
            value = Math.round(value);
            if (value >= eventMin && (value <= eventMax || eventMax == 0.0)) {
                return value;
            }
        }
        if (value >= eventMin && (value <= eventMax || eventMax == 0.0)) {
            return value;
        }
    }
}
```

2.2 File name and log output conventions

For the initial activity generation, all log files will be pre-fixed with “BaseActivityLog<Day number>.txt” where the day number is iterated over the number of days given in the initial program parameters. Within each log file, the data is shown abiding with the similar pattern of ‘:’ delimiters. Each log row is an recurrence of either a single discrete event, or a single row of the total value of continuous event prefixed by a randomly generated timestamp with the collection being sorted. Shown below is an example of the activity log generated.

```
01-13-40:Logins:1
01-15-30:Time online:121.75
01-28-21:Emails deleted:1
01-38-34:Emails sent:1
02-42-46:Emails opened:1
06-56-41:Emails deleted:1
09-17-13:Logins:1
09-37-31:Emails sent:1
10-12-30:Emails deleted:1
10-22-51:Logins:1
10-51-12:Emails sent:1
13-34-18:Emails sent:1
16-31-59:Emails sent:1
18-03-39:Emails deleted:1
18-04-02:Emails deleted:1
18-06-19:Emails deleted:1
18-49-10:Emails sent:1
21-03-07:Emails opened:1
21-24-09:Emails sent:1
23-10-00:Emails opened:1
```

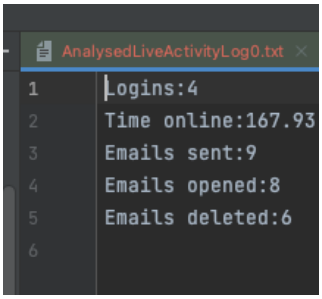
Analysis Engine.

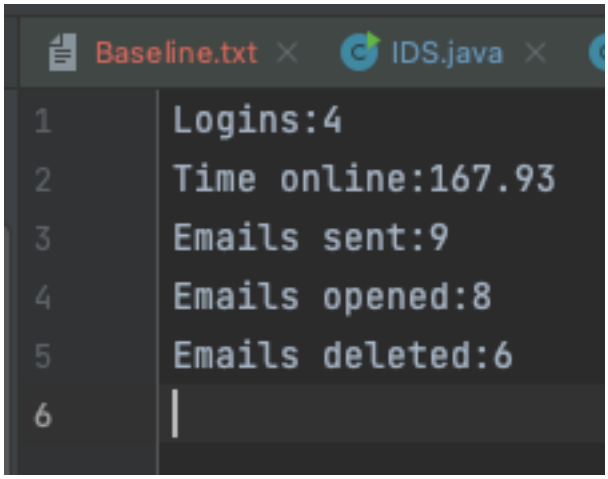
Upon the execution of the analysis engine, the output generated will be as such:

```
Activity Engine created.
Activity Engine finished.
Analysis Engine created.
==================Analysis Engine Report==================
Event Name: Logins Mean :4.2 Standard Deviation:1.17
Event Name: Time online Mean :164.39 Standard Deviation:30.51
Event Name: Emails sent Mean :9.2 Standard Deviation:3.76
Event Name: Emails opened Mean :9.2 Standard Deviation:3.6
Event Name: Emails deleted Mean :6.2 Standard Deviation:1.17
Analysis Engine finished.
```

In the background, the engine iterates all the “BaseActivityLog” files and gets the totals of every event type during the first iteration, with the sum of all values it will iterate again to generate the mean for each event type. Now with a “Map” of the mean of each event, the program will iterate all the log files again this time calculating the standard deviation for each day with the mean for reference.

It will also generate an analyzed log file named “AnalysedBaseActivityLog” file with the summation of each event type for the day and each day. It will also generate a “Baseline.txt” file consisting of the statistics for every event as displayed in the output above.

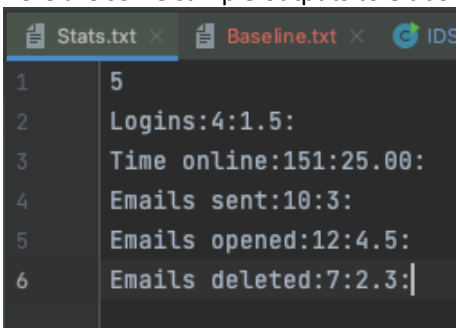




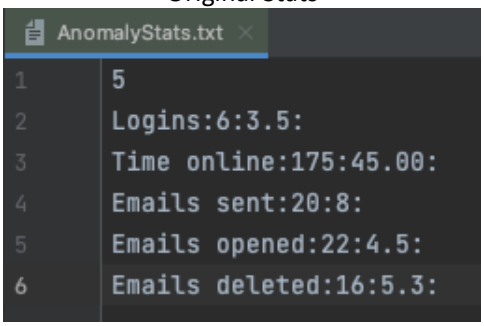
(*We can observe that the distribution values shown are close to what was displayed in Stats.txt which reinforces the fact that the method of simulating the distribution is working.*)

Alert Engine

In practice, the Alert Engie is to check consistency between the live data while referencing its stored baseline statistics. To process the live data, the user is prompted the terminal for a statistics file and the number of days. As this assignment calls for the generation of simulated “live” data as well, a variation of “Stats.txt” has been created with inflated mean and standard deviation values to force the creation of anomalies in the activity engine. Here are some sample outputs to elaborate:

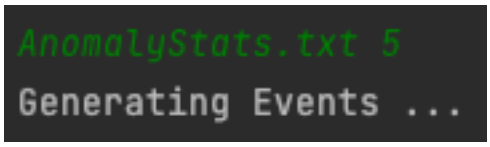


Original Stats

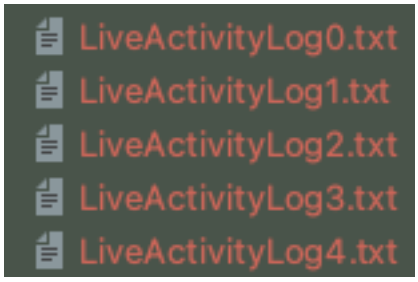


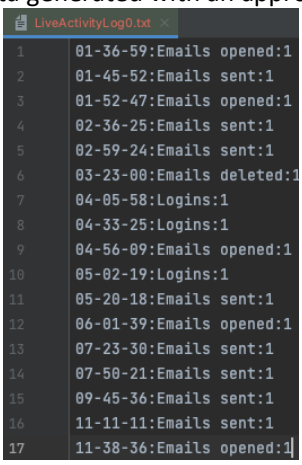
Anomaly Stats

Upon giving the valid file parameters, the program will display:



And the file system will show live data generated with an appropriated file name of “LiveActivityLog”.





The alert engine will then iterated for X days as given in the input and generate the logs in the terminal as such:

```
==================Live Analysis Engine Report===================
==========Day 0 Anomaly threshold: 18==========
Event: Logins Value: 3.0 Mean: 4.2 Delta/Anomaly Counter: 1.2000000000000002 Std: 1.17
Event: Time online Value: 135.93 Mean: 164.39 Delta/Anomaly Counter: 28.45999999999998 Std: 30.51
Event: Emails sent Value: 18.0 Mean: 9.2 Delta/Anomaly Counter: 8.8 Std: 3.76
Event: Emails opened Value: 8.0 Mean: 9.2 Delta/Anomaly Counter: 1.1999999999999993 Std: 3.6
Event: Emails deleted Value: 6.0 Mean: 6.2 Delta/Anomaly Counter: 0.20000000000000018 Std: 1.17
Day 0 Anomaly score: 5.417348608837972
Day 0 Anomaly score: 5.417348608837972 is less than the threshold: 18 No anomaly detected
==========End of Day 0==========
```

It will state the event name with its value count for the day, the mean of it’s baseline, standard deviation of it’s baseline and the delta or anomaly counter. For each day a sum of the anomaly score is shown which is computed with this formulate implemented as such:

```
double valueDouble = Double.parseDouble(value);
double delta = Math.abs(valueDouble - s.getMean());
System.out.println("Event: " + eventName + " Value: " + valueDouble + " Mean: " + s.getMean() + " Delta/Anomaly Counter: " + delta + " Std: " + s.getStandardDeviation());
if (delta > s.getStandardDeviation()) {
    anomalyScore += e.getEventWeight() * (delta/s.getStandardDeviation());
}
```

In the event that the anomaly score is larger than the threshold the final line of the output will state that “an anomaly has been detected” instead of “No anomaly”, an example of such an output:

```
Day 3 Anomaly score: 34.85170030914712 is greater than the threshold: 18 Anomaly detected
==========End of Day 3==========
```

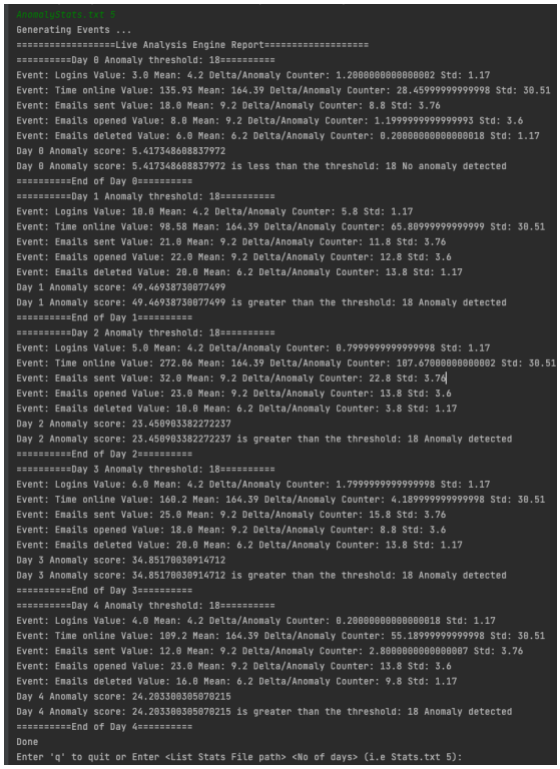
Once the alert engine has completed running, the program will await further file inputs or a prompt to quit the entire program.

```
Enter 'q' to quit or Enter <List Stats File path> <No of days> (i.e Stats.txt 5):
```

Proof of execution:

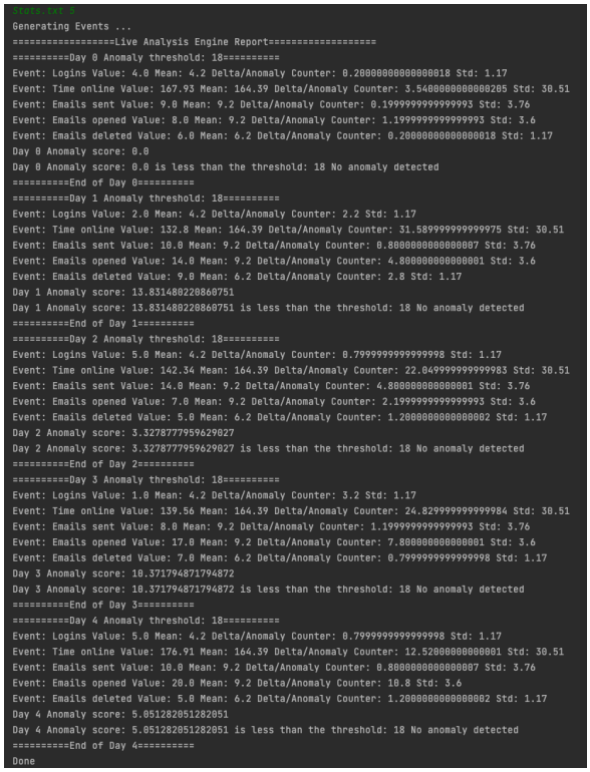
As shown previously, the inflated values stored in "AnomalyStats.txt" should trigger more detection as the activity engine will generate more data that will skew more into triggering more alerts. This can be shown in the difference in running the alert engine with "AnomalyStats.txt" as compared to "Stats.txt" which was the baseline file.

AnomalyStats.txt



4/5 days with anomalies

Stats.txt



0/5 days with anomalies