**System for Generating Traffic Metrics**

Hello and thank you for checking out my project! This document includes a bit of background, a brief description of what the system does/how it works, and some instruction on how to get it all running.

In the Spring of 2022, I worked as part of a team with Dr. Junaid Zubairi, Dr. Syed Haider, Dr. Shahin Mehdipour-Ataee, and Dr. Sahar Idwan to develop an algorithm for providing detours around congested lengths of roads in urban areas. The algorithm uses tabular data as a .csv file containing block-by-block traffic metrics as source material. The repo for the project can be found [here](https://github.com/djhurtgen/Smart-City-Traffic-Guidance).

The original .csv file features metrics that were manually entered, one row at a time, modeling a snapshot of real traffic conditions taken from Google Maps over a small portion of Manhattan during rush hour. In an attempt to get practice with Apache Kafka and the AWS Cloud (specifically, DynamoDB and S3), I’ve returned to this aspect of the project to create a system that mimics the generation of data by onsite sensors — these would be IoT devices — which is then streamed in the form of a message queue and consumed by a program that periodically aggregates and averages the metrics to produce a new .csv file. A 2nd consumer simultaneously loads the data into an AWS DynamoDB database and S3 storage bucket for potential further analysis. The DynamoDB database could be used to analyze the data in a historical context (for example, “What does traffic tend to look like on 24th Street from 9 am to 10 am on Wednesday’s?”). The goal is to produce metrics that realistically reflect changing traffic patterns on a grid over time.

**Files:**

* kafka-traffic-simulator-client.py: this program should be run once after the Kafka zookeeper and Kafka server have been started to establish a Python client and create a topic for messaging
* kafka-traffic-simulator-producer.py: this program generates traffic metrics in the form of json objects, which are then converted to utf-8 and pushed into the message queue
* kafka-traffic-simulator-consumer.py: this program consumes the messages sent by the producer and periodically produces a .csv file that can be read in by the traffic guidance programs
* kafka-traffic-simulator-aws.py: this is also a consumer which uploads the messages into a DynamoDB database and S3 storage bucket
* traffic-metrics.py: this program houses the class used for DynamoDB object creation, it is imported by kafka-traffic-simulator.py
* manhattan.csv: this file is necessary as a baseline for the producer. The fields ‘avg\_speed’ and ‘num\_cars’ are set to average values for each block
* manhattan\_generated1.csv, manhattan\_generated2.csv: these are sample files output by the consumer. They were not produced as part of the same run.
* traffic-metrics.json: a sample file output by the aws consumer for loading into S3
* dynamodb\_query.png, dynamo\_results.png, s3\_objects.png: these are screenshots from AWS DynamoDB and S3, showing successful runs of the programs

**Data Flow Diagram:**

manhattan.csv 🡪 kafka-traffic-simulator-producer.py 🡪 message queue 🡪 kafka-traffic-simulator-consumer.py/kafka-traffic-simulator-aws.py 🡪 manhattan\_generated.csv/AWS DynamoDB and S3

**Overview of Simulated Metric Production:**

The heart of the traffic simulation algorithm lies within the inner loop of the producer. In summary, road segments are aggregated with their immediate neighbors (adjacent road segments where traffic is moving in the same direction) and the average speed of this aggregate is calculated. A new speed for the road segment is then produced using Python’s built-in ‘normalvariate’ function (a function that generates random numbers across a normal distribution), with mu = the average speed of the aggregated segments calculated above and sigma = 2. In this way, a relation is formed between adjacent road segments. From here, the number of cars on the road segment is determined thusly: the speed variable that was just produced is normalized on a scale of 0 to 1. A normalized ‘num\_of\_cars’ variable is produced as the absolute value of the normalized speed - 1:

This value is then ‘denormalized’ according to the maximum number of cars possible along the given road segment, which is:

In short, as speed increases along the segment, the number of cars drops. When the speed is at a maximum (the speed limit), the number of cars is 0. When the speed is at a minimum (5 mph), the number of cars is at a maximum (varies by length of road segment).

**Note:** For the aggregations to be properly formed, the original .csv file should have road segments grouped together and arranged by direction. For this particular .csv file (Manhattan), this means:

1. All 11th Avenue segments, traffic moving in one direction, i.e. south
2. All 11th Avenue segments, traffic moving in the opposite direction from above, i.e. north
3. Repeat for remaining avenues
4. All 23rd Street segments, traffic moving in one direction, i.e. east
5. All 23rd Street segments, traffic moving in the opposite direction, i.e. west
6. Repeat for remaining streets

**Instructions for Running the System:**

**Prerequisites:**

* An AWS account (this is only necessary to run the program ‘kafka-traffic-simulator-aws.py’… the other programs can successfully be run without it)
* I can’t speak to running Kafka on Windows or Mac platforms, I used a virtual machine running Ubuntu 22.04.3
* Java, version 8 or higher (required for Kafka)
* pandas module for Python… I would suggest creating a separate Python environment for the project. Pandas can be pip installed with:

**pip install pandas**

* kafka-python module for Python… Also installed in the project environment. Pip installed with:

**pip install kafka-python**

* boto3 module for Python (AWS SDK)… Also installed in the project environment. Pip installed with:

**pip install boto3**

\*only necessary to run ‘kafka-traffic-simulator-aws.py’

**Installing Apache Kafka:**

From the command line:

* Download Kafka:

**wget https://dlcdn.apache.org/kafka/3.4.0/kafka\_2.13-3.4.0.tgz**

* Apache recommends verifying the download. I used SHA256:

**gpg --print-md SHA256** **kafka\_2.13-3.4.0.tgz**

Compare the output with:

**sha256sum kafka\_2.13-3.4.0.tgz**

* Extract the .tgz file:

**tar -xzf kafka\_2.13-3.4.0.tgz**

**Running the System:**

From the command line:

* cd into the newly created Kafka folder:

**cd kafka\_2.13-3.4.0**

* Run the Zookeeper:

**bin/zookeeper-server-start.sh config/zookeeper.properties**

* Open another terminal window and run the Kafka server from the same Kafka folder:

**bin/kafka-server-start.sh config/server.properties**

* From here, one can run the system. Activate your Python environment (again, be sure pandas and kafka-python are installed, along with boto3 if AWS is being used) and run the programs in the following order (the producer and consumers must be run in separate terminals)…

1. kafka-traffic-simulator-client.py
2. kafka-traffic-simulator-producer.py
3. kafka-traffic-simulator-consumer.py
4. kafka-traffic-simulator-aws.py (AWS only)

* The consumers will have to be manually stopped once the producer has completed its loop. Otherwise, they continue running/’listening’ for incoming messages. Finally, stop the Kafka server (Ctrl + C) and Zookeeper (Ctrl + C).

**General Notes:**

* This is a simplified representation of an actual system. In reality, sensors would likely record speed and add to a total count of cars as each car passed. This would continuously create tens (or hundreds?) of messages per block per minute and necessitate a producer for each block. Though mimicking such a system would be very cool, I decided to scale things back to what you see here… messages for each block are sent simultaneously every 30 seconds by one producer. The consumer aggregates and averages after 10 runs (every 5 minutes) a total of 4 times. Any of these parameters can easily be changed by the user.
* I did my best to avoid hard-coding in these programs, but it’s possible that minor changes will need to be made when using different grids.
* David