MSc Data Science
Department of Computer Science and Information Systems
Birkbeck, University of London, 2019

Project Report

Real-time Twitter Data Analytics and Sentiment Analysis: Big Data Software Application Development

Samuel McIlroy

Abstract

This project report details the design, implementation and analysis of a data intensive software application to acquire, store, process and visualise analytics from real time data generated as Twitter ‘tweet’ data from users in London. The application makes use of common distributed computing technologies for working with large volumes of data at scale, and web frameworks for presentation of the data to produce an end to end solution comprised generally of: 1) Scalable data pipelines to collect and store real time tweets across London. The pipelines are designed source and transform raw Twitter API JSON data and store the results for collection. 2) Scalable processing of the data collected, in real time, to produce key analytics. This will include maintaining a running count of key metrics (trending topics and users) and a sentiment analysis determining the ‘polarity’ and ‘sentiment’ of the text of each tweet as it is collected. 3) A web based front end to display the results showing the running totals and, as a centerpiece, a continually updating map of London showing the location and sentiment of each tweet. The project is developed and maintained in a Docker container so that the project, its codebase and its dependencies can be deployed as a single unit to any server or cloud based service allowing the application to scale to larger and larger volumes of data with minimal additional coding or development.

Contents

Structure of the Report ……...….. 5
Code Repositories …… 5
1. Introduction ……... 6
1.1 Problem Statement (Why this Problem Needs Solving)……. 7
1.2 Aims and Objectives …….. 7
1.3 Challenges, Volume of Data and Proof of Concept Approach ……... 8
2. Specification and Design…… 9
2.1 Specification ……. 9
2.2 Technologies to Implement at Each Stage ……. 9
2.2.1 Development Environment: Containerisation and Version Control ……. 9
2.2.2 Data Ingestion and Storage …… 13
2.2.3 Data Processing and Storage ……... 16
2.2.4 Front-end Web Interface, Charts and Sentiment Map ……. 18
2.2.5 Overall Design Diagram, Development Stages ……... 20
3. Implementation …… 21
3.1 Initial Setup, Development Environment and Docker ……. 21
3.1.1 Docker Image Creation …….. 21
3.1.2 Python, Java and Scala Versions …… 22
3.1.3 Apache Kafka and Apache Zookeeper Installation …… 22
3.1.4. Apache Spark Installation …… 22
3.1.5 Flask, Chart.js and Leaflet.js Installation …….. 23
3.1.6 DockerHub ……. 23
3.1.7 Visual Studio Code ……. 23
3.2 Data Ingestion …….. 24
3.2.1 Summary and Design Diagram …….. 24
3.2.2 Twitter Streaming API and Kafka Topics Creation …… 24
3.2.3 Tweepy Connection, Pulling JSON Tweets and Producing as Kafka Messages…. 25
3.3 Data Processing and Analysis …….. 29
3.3.1 Summary and Design Diagram ……. 29
3.3.2 Consuming Kafka Topics, Data Collection and Preparation ……. 30
3.3.3 Data Processing, Apache Spark Analytics ……. 31
3.4 Front-end Visualisations …….. 37
3.4.1 Summary and Design Diagram ……. 37
3.4.2 Flask Implementation ……... 38
3.4.3 Chart.js Visualisations ……... 40
3.4.4 London Sentiment Map, Leaflet.js …… 42
3.4.5 Page Design: Bootstrap and CSS …….. 45
4. Testing …….. 46
4.1 Testing: Data Ingestion …… 46
4.2 Testing: Data Processing and Analysis …… 49
4.3 Testing: Front End Visualisations …… 52
4.4 Testing: Summary …… 54

5. Results …… 55
5.1 Results: Real time data ingestion and storage …….. 55
5.2 Results: Real time data processing and storage …… 55
5.3 Results: Web based front end …… 56
5.4 Results: Summary …… 56
6. Evaluation and Future Work …….. 57
6.1 Evaluation: What went well? …… 57
6.2 Evaluation: Lessons Learned …… 57
6.3 Future Work …….. 58
Appendix A: Project Running Instructions for Examiners …….. 59
References ……..…….. 62

Structure of the Report

The report is structured in the following way:

1. An introduction to the project, why it has been chosen and the aims and objectives to be met

2. A description of the design and specification stages including details on the technologies to be implemented for development

3. A detailed overview of the implementation of the report including analysis and description of the code developed and functionality implemented.

4. Details of the tests completed to ensure that the project meets its aims and objectives

5. Details of to what extent the deliverables of the project have been achieved and aims and objectives satisfied

6. A personal evaluation of the work and reflection on the process, including lessons learned and opportunities for future work

Code Repositories

All code produced is available on the the private GitHub repository:
https://github.com/sammcilroy/msc_project_smcilr01

The Docker container image developed to run this project is available on the private DockerHub repository: **sammcilroy/london_twitter:latest**

Access to both of these repositories can be given on request. Instructions on running the project through the Docker container is provided in Appendix A

1. Introduction

This project aims to utilise modern development in open source distributed and cloud computing technologies around the processing of large volumes of data efficiently to create an end to end solution for the collection, analysis and visualisation of Twitter data including a running total of key trends and users and a real-time sentiment analysis which aims to gain insight into the general ‘happiness’ levels of Londoners.

“It is estimated that the amount of data generated during this year, 2019, will surpass all data that has been collected in the last 500 decades combined [1]. Each day the internet search and social media giants alone collect and store petabytes of user generated content, as approximately 3.7 billion active internet users log in around the world [2].

The need to derive meaning and use cases from this vastly expanding data has led to rapid developments in distributed and cloud computing technologies to handle data processing at increasing scale. The principal challenge has been that as large volume data storage becomes cheaper and easier, relatively slow pace in CPU speed improvements has meant that the volume of data becomes the bottleneck, rather than the clock speed of the CPU in more traditional single machine development, if the data is to be used in a meaningful way and processed in a reasonable time-frame. The need to distribute data processing over many machines for parallel processing then becomes key, leading to the term ‘data-intensive’ software applications [3].

While just some years ago data processing at this scale would have been expensive and difficult due to the number of machines required and the relatively niche software engineering skill-sets required, modern advances in the cloud computing Infrastructure as a Service (IaaS) approach of providers (e.g. Amazon Web Services, Google Cloud Platform and Microsoft Azure) allowing cheap and easy access to rented computing clusters, and the development of, and industry preference for, free and open source (FOSS) software platforms for distributed computing and big data analytics (Apache Spark etc.) access to hundreds of machines, large scale storage and industry level software is available cheaply and easily to large companies, start ups and single end users alike. [3]” [38]

1.1 Problem Statement (Why this Problem Needs Solving)

I have chosen to produce this project as it will allow me to simulate the end to end development of a data software product which demonstrated the scalable analysis of real-time streaming data analogous to larger products which are developed by software engineers supporting data analytics teams in the real world.

“Using Twitter data to map trends and sentiment across London areas gives both access to the required amount of ‘big’ data to complete a software application in this area, and the opportunity to solve an interesting problem. Providing analysis and access to popular trends, user sentiment and opinion, especially geographically, allows us to quantify human feelings and key metrics regarding their environments, situations and their daily lives. Access to such compiled and visualised data is of interest to data scientists, social scientists, corporations and marketing organisations, urban planners and more. [4]” [38]

1.2 Aims and Objectives

The overall objective of the project is to demonstrate the development of an industry standard software application to provide access to visualisations and results from large scale data processing and to show the practical application of modern technologies and processing data at scale.

The aims and objectives of the project in brief:

- The implementation of distributed and cloud technologies to develop, as close to industry standard as possible, the data pipelines required for both real time streaming processing of data from the Twitter API as needed for the development of the subsequent visualisation front-end.

- The application of real time data processing and sentiment analysis.

- The implementation of a front-end visualisation/mapping of the data/sentiment analysis being performed on the streaming data. The front-end will show the results of the analysis in real-time as the data is collected and processed.

This report will outline the technologies used and document the application and its development and testing processes.

1.3 Challenges, Volume of Data and Proof of Concept Approach

“Before development of the project began, there were challenges to the proposed analytics across all London twitter users. The principal challenge identified is the volume of data that can be acquired from the Twitter API. In general, the Twitter API limits the amount of tweets that can be captured to a 1% sampling of the data. This means that the tweets I can collect to analyse do not represent the entire population.

To estimate the volume of data I can expect, I conducted a one hour test of streaming data using my own computer and a simple connection to the API through the Tweepy python library. During this test I was able to capture 1753 tweets in a one hour period. Assuming a constant rate during the day, for simplicity, this is approximately 42,000 tweets per day, or around one tweet every two seconds.

This volume of data gives sufficient volume to implement the project and to provide an adequate sampling of data to create analysis and mappings which are representative of the population as a whole. While it would be possible to use alternative more traditional approaches to handle this data, I have decided on on the implementation of a ‘proof of concept’ approach where if the volume of data were to be increased, to the full Twitter population for example, then the same approach could be taken with as little alterations as possible.” [38]

2. Specification and Design

This chapter will begin by outlining the specification of the project application which will serve as the deliverables/contract of the application. It will then go on to detail the technologies chosen to be implemented at each stage and the overall design of the application to ensure the specification is met.

2.1 Specification

The application developed will include/implement the following features:

- **Real time data ingestion and storage** of as many tweets from within the London area as the Twitter and their public streaming API will allow. Tweet data should contain:
  - JSON representations of tweets
  - restricted to London coordinates, by use of a geographical boundary coordinate box
  - location/coordinates data must be included for any map based visualisations
  - text data must be included for any sentiment analysis
  - text data must be included for extraction of trending topics, hashtags, and tagged 'influential' users (@ tags).
  - Data should be held in short or long term storage as 'streams' to be processed by subsequent stages
  - Data 'streams' should be held either in long term storage or in short term storage,for example stored in a message broker system, i.e. an appropriate Apache Kafka topic

- **Real time data processing and storage** of tweets as events/messages as soon as they are collected.
  - Data should be made available as 'streams' to the processing stage.
  - The processing should produce the following three outputs
    - dataframe: running total of the top ten hashtags used in all tweets processed so far
    - dataframe: running total of the top ten users tagged in all tweets produced so far
    - transformed data: each tweet should be processed to extract polarity and sentiment from the text
      - re-stored as new 'stream' and handled as other streams above

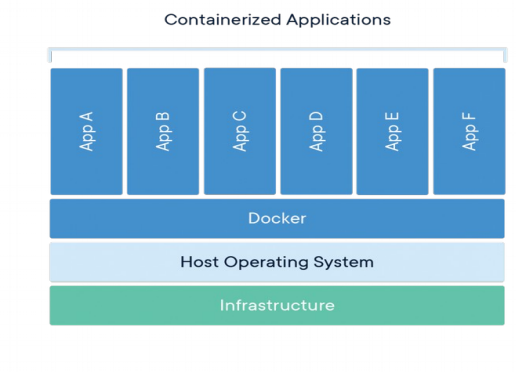
- **Web based front-end**, create and update visualisations to show the data being processed in real time. Should provide:
  - real-time bar-chart: top ten trending topics
    - defined by topics tagged by #
  - real-time bar-chart: top ten tagged 'influential' users
    - defined by users tagged by @
  - sentiment map of London
    - should show coordinates/user location and attached sentiment/polarity value for each tweet as processed, as close to real-time as possible (allowing for processing time)
    - should give an indication of sentiment levels and tweet volume across London
  - appropriate organisation, navigation and visual elements to create an efficient user interface/ access to the visualised data

2.2 Technologies to Implement at Each Stage

2.2.1 Development Environment: Containerisation and Version Control

Containerisation, Docker

"A Docker container is an executable software package that hosts together everything required for an application to run, including the OS, software tools, and dependencies. [5]" [38]



A Docker Container [5]

The project will be developed and be able to run wholly inside a single Docker container/image rather than on a local machine or virtual machine environment. This will have several advantages:

- All of the requirements for the project, programming language versions, Apache Spark and Kafka environments and versions, can be packaged and maintained without conflicting with the host OS environment or each other

- The docker image will contain the required Spark and Kafka environments for the project. Apache Spark and Apache Kafka have multiple versions which can cause conflict and dependency mismatches if not carefully maintained. This is especially true when using Python development libraries rather than the native JVM/Scala implementations as there are additional steps required to communicate between Python and the JVM.

- The container/image can be run on any machine with docker installed without having to set up new development environments

- A Docker based deployment is lighter on resources as it runs on the host OS' resources rather than having to virtualise a complete system.

- Once complete the project will be contained in a single image/file. It can easily be shared and more easily deployed to servers and cloud environments. This will be especially useful if the project were to be employed to process larger volumes of data, for example the full stream of Twitter API data if it were available. [6]

Programming Languages

“The project will be developed in the Python programming language. I am most experienced in Python programming and the majority of the open source Apache based technologies I have considered have robust Python implementations.

Apache Spark, which will be employed during the data processing stage, especially is known for its Python support and ‘first class’ treatment of Python as a second language. Spark was originally coded in Scala and there are documented speed advantages to using Scala over Python as Python requires additional steps to communicate with the JVM. However, these speed advantages are becoming less relevant as development and strong support for Python continues. [7]” [38] Apache Kafka also has well documented python support through the Kafka python library [8] and Flask provides a user friendly and simple python based framework for work on the front-end interface. [9]

“There are also advantages to using the same language throughout the whole project as much a possible. Readability of code and documentation will be easier for example when using a single language.” [38]

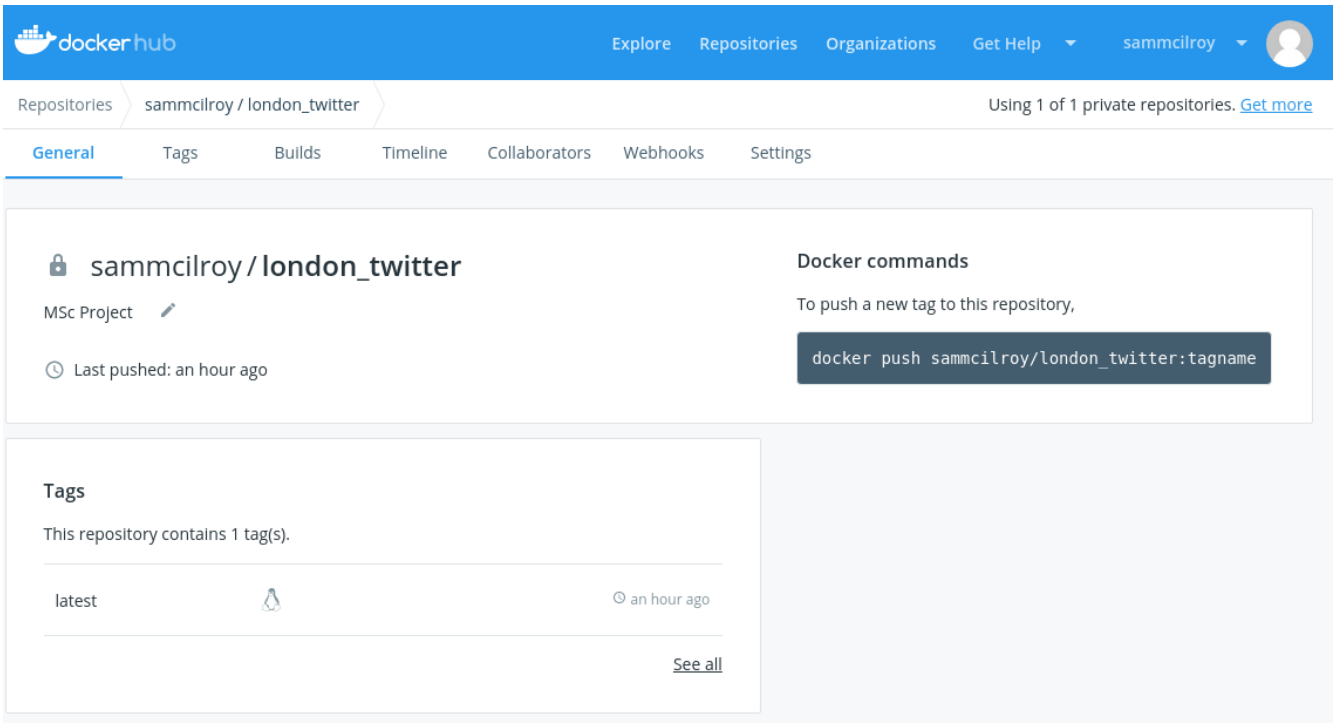
Additional languages required will be restricted to the front-end development stage. HTML, CSS, and JavaScript (through visualisation frameworks) will be utilised to create the visual elements.

Version Control

Version control will be an important part of the development process which will help to maintain:

- A well documented history of the development
- The ability to revert to previous versions of code when things go wrong
- Storage of the code and Docker images
- The ability to share code with Birkbeck markers

All of the code will be stored in a private GitHub repository which is made available to my supervisor and can be made available to other Birkbeck staff members on request. The Docker image of the project will be maintained on a private DockerHub repository which will server both as version control, allowing several versions of the image to be maintained during development, and as a general host of the image/project allowing it to be shared and 'pulled' in its entirety to other computers/servers as needed. [11]



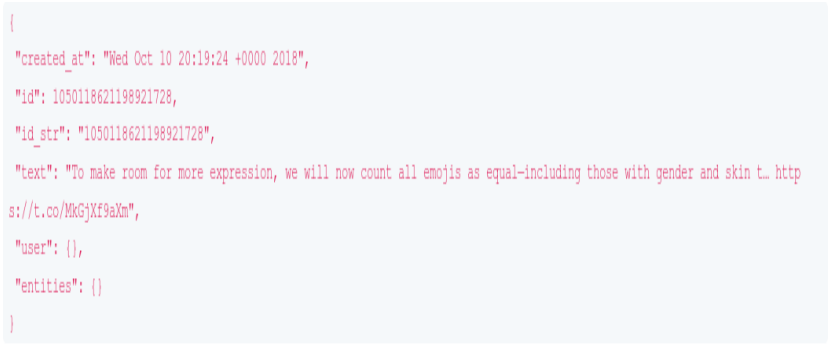
2.2.2 Data Ingestion and Storage

Twitter API

"Twitter provide a standard Streaming API for developers. This will allow me to collect tweets as they are posted. In order to access the APIs users must apply and be accepted as a developer. I have applied and have access to the APIs.

Twitter currently limits the standard Streaming API to a 1% sample of the streaming tweets queried. To navigate this restriction, some aspects of the project may have to use samples or more limited data but should still allow the aims to be implemented, at least as proofs of concept which could be bettered with less restrictive access. Premium APIs are available to remove some of these restrictions but these are cost prohibitive to implement in this project. [12]" [38]

Tweets can be consumed as 'Tweet Objects' which contain all of the data contained in the tweet in an organised format which can be extracted as needed for the project. [13]



Tweet contents mapping to JSON object, example from Twitter API Documentation [13]

During the data ingestion stage the project will pull 'Tweet Objects' in JSON in their entirety from the Twitter API to an Apache Kafka topic, where the data processing stage will employ analytics on individual aspects of the tweet by extracting the relevant JSON fields. Twitter provides a dictionary/reference table to map JSON elements to aspects of each tweet which will be used to ensure the correct data is being analysed.

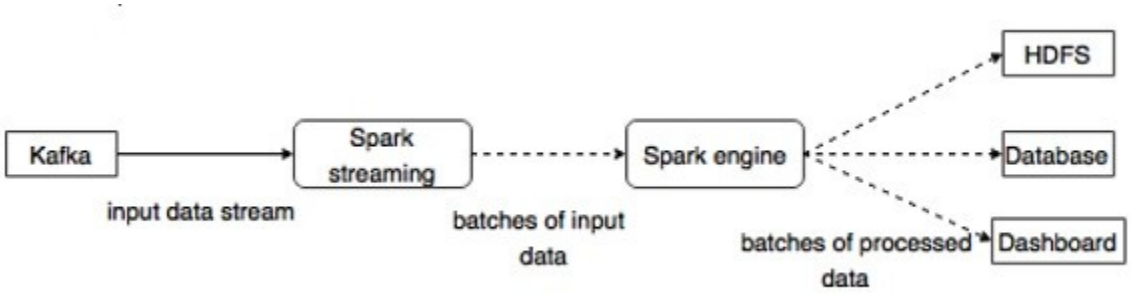
Tweepy Python Library

The Tweepy Python Library is self described as ‘an easy to use Python library for accessing the Twitter API’. [14] Tweepy allows live/streaming connections from Python to the Twitter API via OAuth and the Twitter provided developer access tokens. Tweepy is used in the project to connect to the API during the data ingestion stage, filter by London region by using a bounding box or coordinates, and prepare the data for storage in Apache Kafka.

Apache Kafka

“In terms of the data streamed from the Twitter API, each tweet and its accompanying metadata can be seen as a message, or event. A message broker, such as Apache Kafka is designed to sit between the data processing stage and Twitter API. The broker can be set to listen for specific messages, such as a tweet with the corresponding London coordinates, and pull and store that message for later processing.

For example, in a typical streaming data pipeline, Apache Kafka might serve as the message broker to store data for and serve data to Apache Spark for processing:



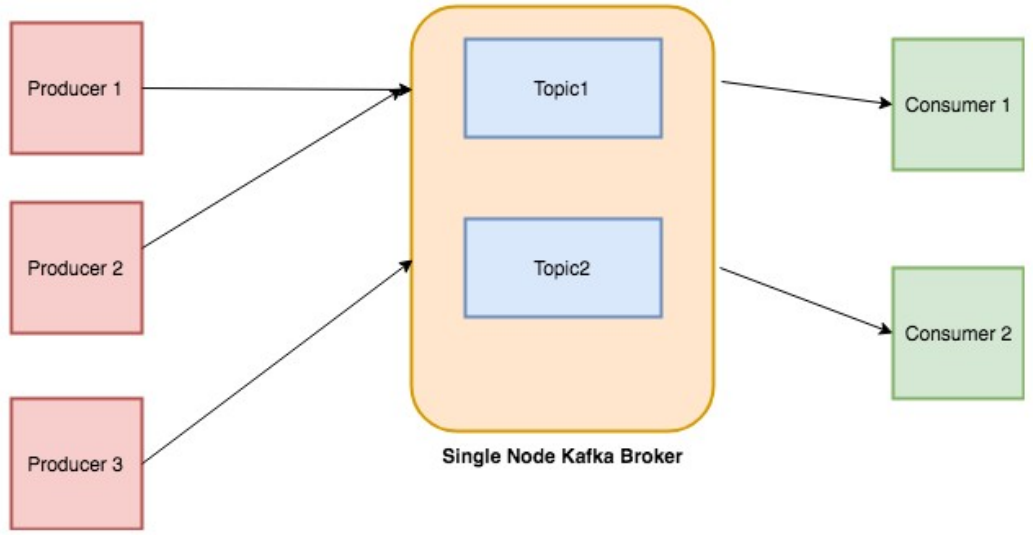
Apache Kafka serving as a message broker, feeding streaming input data to Spark Streaming [15]

A message broker can be beneficial in that records can be stored there until it is picked up for processing, into Apache Spark for example. For instance, if you wanted to make changes or tweaks to the data processing then the message broker could continue to run and collect the data. If data was simply ‘pulled’ into the data processing as objects then if changes were required the pipeline would have to be shut down, and any data/tweets that were generated by the API during that time would be lost. [16]” [38]

There are three key concepts of Apache Kafka which will be utilised to collect and store data: topics, producers and consumers. Topics are named record streams where data can be collected and stored, on disk. Producers are code designed to publish or ‘push’ data into the topic record streams. Consumers are code designed to collect the data, usually in combination with data processing. [17]

Apache Kafka topics will be used to pick up and store both data/messages from the Twitter API stream and from the data processing stage to create a new ‘stream’ of tweet data with attached sentiments and coordinates. Data storage will accomplished by this use of Kafka topics rather than use of a relational or non-relational database system. Consumers will be utilised to collect and use the data as appropriate. This will be described in more detail in the next chapter.

A Kafka ‘instance’ or ‘broker’ can run either on one machine, usually for development purposes, or as multiple brokers distributed across a cluster when deploying to scale. The Docker container for the project will be built with a single node Kafka broker setup for development and demoing of the front end.



A typical single node Kafka cluster used in development. Three producers collect data into two topics with the data collected ‘consumed’ by two consumers for data processing. [17]

Kafka-Python Python Library

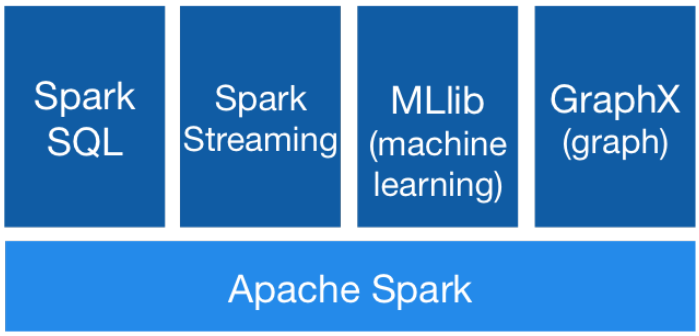
Kafka-Python is a Python client for interacting with with Apache Kafka. The library offers both a python message producer and message consumer which is, as stated in the documentation, designed to ‘operate as similarly as possible to the official Java client’.

The project will use this library for creating the Apache Kafka producer and consumer code sections.

2.2.3 Data Processing and Storage

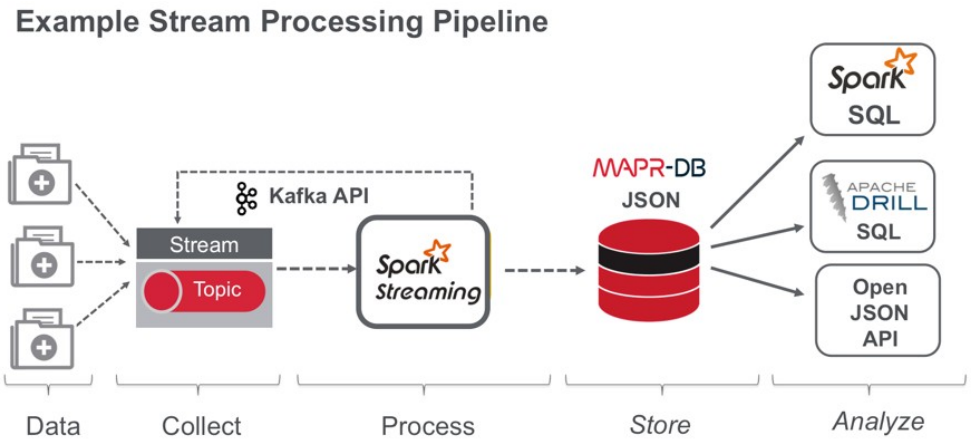
Apache Spark

"Apache Spark is a technology and set of programming libraries designed for parallel data processing on multiple computers and is becoming a standard tool for big data processing.



Apache Spark Ecosystem

Spark is incredibly fast due to its focus on in-memory processing of the data, rather than relying on data stored in stable storage as with comparable platforms such as MapReduce. Therefore Spark is particularly suited, due to its speed, for real time processing/streaming where data must be processed quickly to prevent bottlenecks as new data becomes available. [19]" [38]



Apache Spark filling the data processing role in a typical streaming process [20]

Spark will be used in the project' data processing stage to analyse tweet data as it is received from the Kafka topics, producing the running total analytics and sentiment analysis.

PySpark Python Library

Pyspark is a Python library for creating Spark dataframes and analysis. The project will make use of this library to complete the Spark analysis work in Python. [21]

TextBlob Python Library

TextBlob is a Python library for text processing and natural language processing and sentiment analysis functions. TextBlob can be used to extract polarity/sentiment level from English text simply.

The project will use the TextBlob library, as above, in conjunction with Apache Spark to analyse the tweet text data for sentiment in real time. The TextBlob library was chosen for its simplicity and accuracy. [22]

```
from textblob import TextBlob

text = '''
The titular threat of The Blob has always struck me as the ultimate movie
monster: an insatiably hungry, amoeba-like mass able to penetrate
virtually any safeguard, capable of--as a doomed doctor chillingly
describes it--"assimilating flesh on contact.
Snide comparisons to gelatin be damned, it's a concept with the most
devastating of potential consequences, not unlike the grey goo scenario
proposed by technological theorists fearful of
artificial intelligence run rampant.
'''

blob = TextBlob(text)
blob.tags           # [('The', 'DT'), ('titular', 'JJ'),
                    #  ('threat', 'NN'), ('of', 'IN'), ...]

blob.noun_phrases   # WordList(['titular threat', 'blob',
                    #            'ultimate movie monster',
                    #            'amoeba-like mass', ...])

for sentence in blob.sentences:
    print(sentence.sentiment.polarity)
# 0.060
# -0.341

blob.translate(to="es")  # 'La amenaza titular de The Blob...'
```

The ‘polarity’ function in the TextBlob library will allow sentiment to be collected simply and accurately [22]

As detailed in the proposal for this project I had intended to pursue Spark’s ML features to manually train and develop a custom sentiment analysis model. However, I have chosen to use TextBlob to provide the data more easily and allow more time/focus on the data collection and processing pipelines and front-end development which were more time consuming than anticipated.

2.2.4 Front-end Web Interface, Charts and Sentiment Map

Flask

Flask is a lightweight and easy to implement web framework written for Python. Flask is more lightweight than larger web implementations such as Apache as Flask supports only the most common uses of a web server. It doesn't require complicated configuration files or setup which need to be customised as in larger deployments. [9]

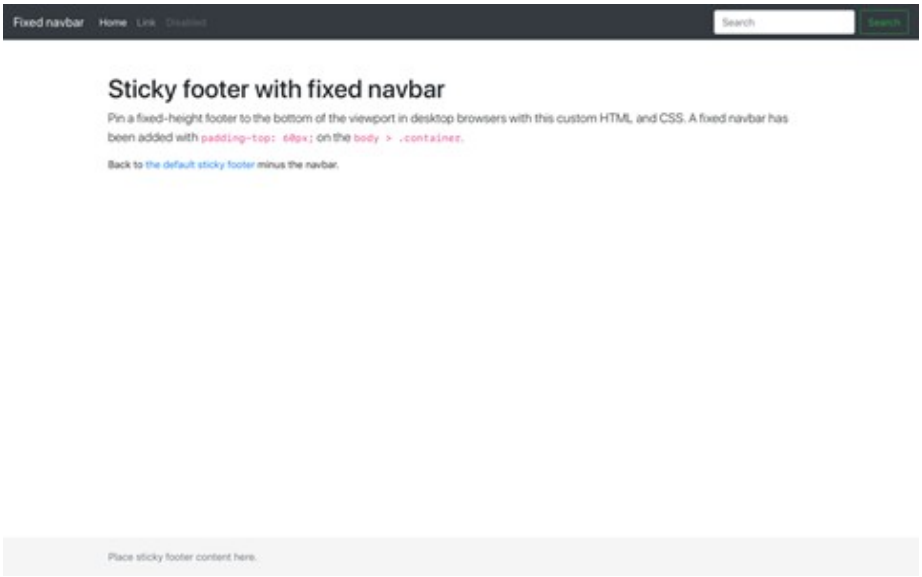
```
from flask import Flask, escape, request

app = Flask(__name__)

@app.route('/')
def hello():
    name = request.args.get("name", "World")
    return f'Hello, {escape(name)}!'
```

A simple flask app [9]

The web-front end will be developed in flask with supporting CSS and JavaScript libraries.

Bootstrap

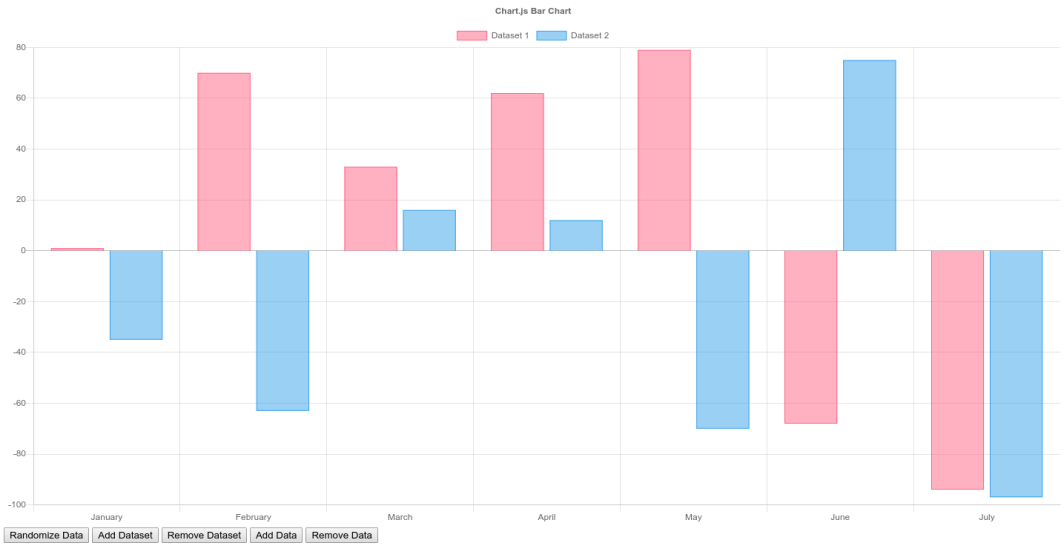
Bootstrap is an open source responsive HTML5, CSS and JavaScript framework which provides various themes, navigation options and customised elements for web projects. Bootstrap will be utilised in the project to provide a consistent and visually appealing style to the front-end. [23]



Bootstrap theming and navigation bar example [23]

Chart.js

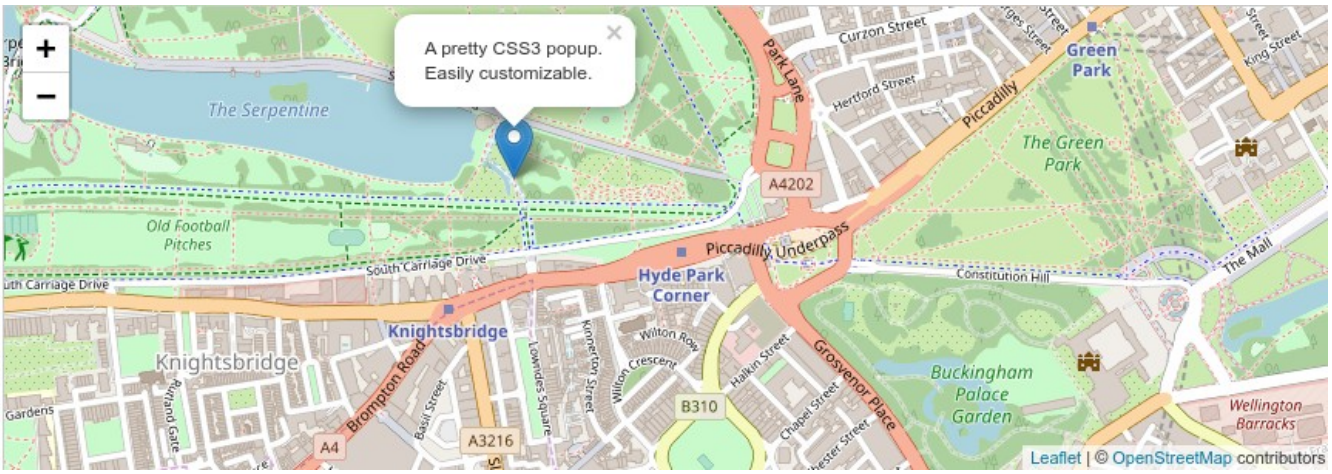
Chart.js is a JavaScript library for creating charts and visualisations. This will be used in the project to create the real-time bar charts of top trending and tagged users. [24]



Example of a Chart.js bar chart [24]

Leaflet.js

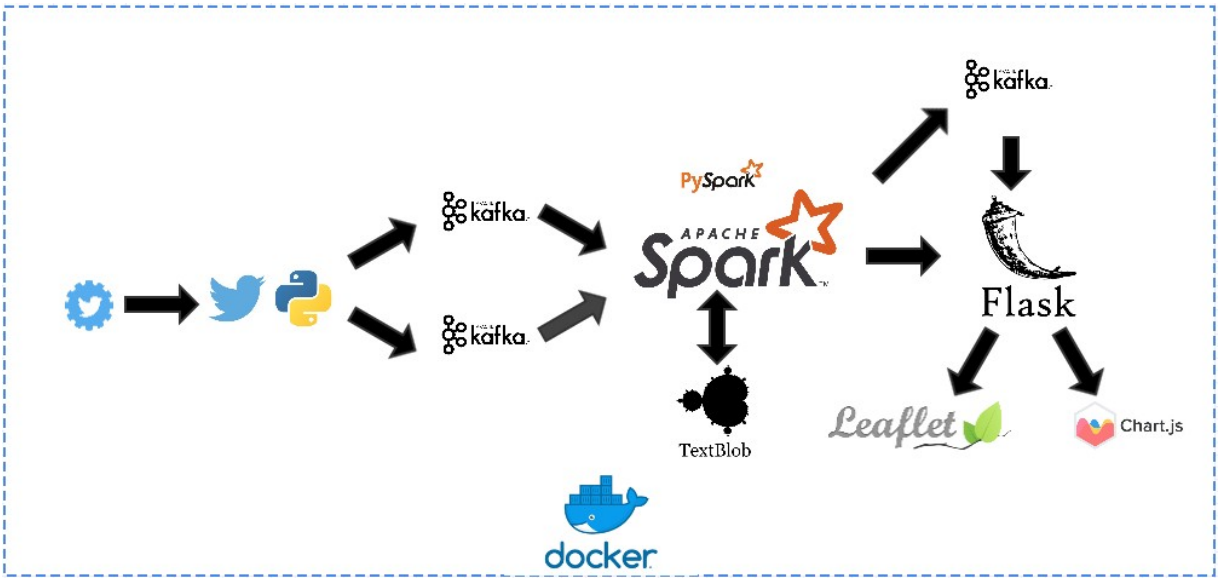
Leaflet.js is a JavaScript visualisation library for creating maps. It allows for mapping of points by coordinate and for customising markers. This will be used to create the live map of tweets as a map of London showing tweets by location and sentiment level. [25]



An example of a Leaflet.js map of London, showing a customised map marker at a specific coordinate [25]

2.2.5 Overall Design Diagram, Development Stages

The diagram below shows the final design of the project and the template for its development, outlining how the technologies detailed above will fit together to complete the project and its aims:



Stage 1: Data Ingestion

The Tweepy and Kafka Python libraries are used to pull the tweets as JSON objects and to create two ‘streams’ of tweets into two separate Kafka topics. The first topic will take all available tweet objects created across London and will be used to analyse the running trending topic and tagged users. The second topic will take all tweet object created across London where an exact coordinate is given (the user has both opted into location sharing and is using a GPS enabled device). By definition this stream will be slower/contain less tweets than the full stream but it will be necessary to use this stream with known coordinates to create the sentiment map.

Stage 2: Data Processing and Analysis

The two Kafka topics ‘produced’ in the previous stage are ‘consumed’ by the data processing stage. This stage utilised Apache Spark to create the running total dataframes and, in combination with the TextBlob library will ‘produce’ a new Kafka topic stream of tweet coordinates and sentiment levels. The two dataframes will be pushed to the front-end at regular 2 second intervals. The separate tweet sentiment stream will be ‘consumed’ during the visualisation stage.

Stage 3: Front-End Visualisation

The final stage is the creation of the web front end and visualisations. A flask app will be created to route and collect data directly from Spark and the produced Kafka sentiment stream. Bootstrap and CSS will be implemented to create a consistent interface to ‘store’ the three visualisations. The two running total charts will be produced in Chart.js with the sentiment map created in Leaflet.js.

3. Implementation

This chapter details the development and implementation of the project through each of its stages: setup, data ingestion, data processing and front-end visualisations.

3.1 Initial Setup, Development Environment and Docker

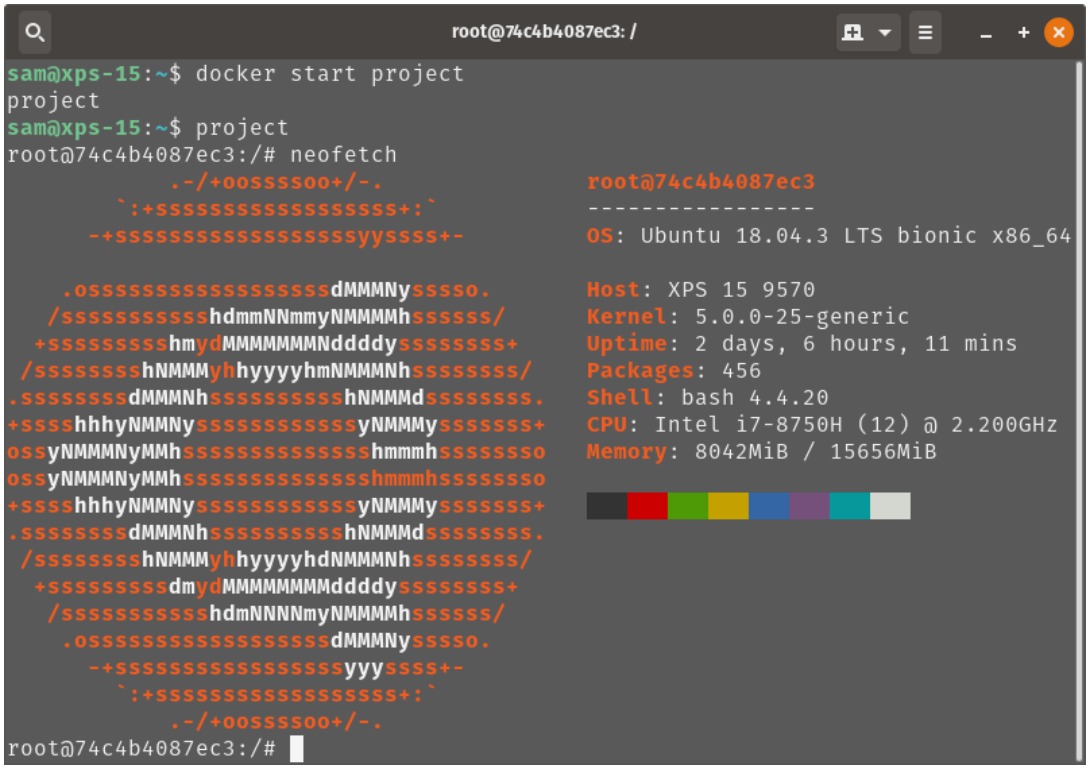
This section describes the process taken to setup the development environment for the project.

3.1.1 Docker Image Creation

In order to develop the applications using a consistent environment, and to be able to save and deploy the application and its dependencies as a single unit the implementation of the project was set up and completed in a docker image/container and hosted in a private DockerHub repository.

The docker image was started from a copy of the official Ubuntu 18.04 LTS image. [26] This provided a full copy of Ubuntu's Long Term Support OS on which the project could be build upon.

Ubuntu 18.04 LTS was chosen as the docker image OS due to Ubuntu's ubiquity both in web and server development and in data science and engineering technology documentation examples and tutorials.



Project docker image running Ubuntu 18.04 LTS showing bash prompt

3.1.2 Python, Java and Scala Versions

Python3, Java 8 and Scala 2.11 programming environments are requirements for the project and Apache Kafka/Spark. These were installed on the container and added to the relevant path variables. The python environment was updated to include the required python libraries as outlined in the previous chapter.

3.1.3 Apache Kafka and Apache Zookeeper Installation

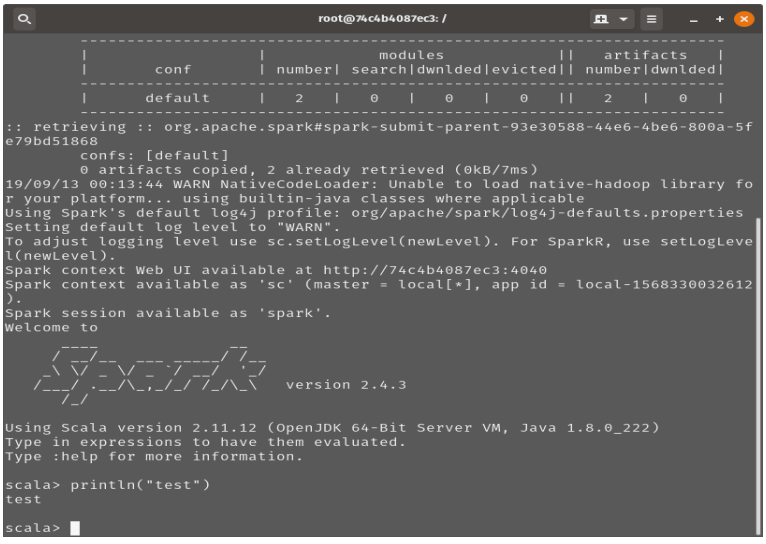
An Apache Kafka installation is required. Alongside this an installation of Apache Zookeeper, a centralised service for managing distributed hosts, is required and included in the Apache Kafka service.

Apache Kafka version 2.11-2.3.0 was installed in the root directory. [27] The command to start the Kafka service was added to the Linux OS' .bashrc file so that a new Kafka service is started on port 9092 at startup. The command to start the required Zoopkeeper instance on port 2181 at startup was also added. Every time the project docker container is started the Kafka and Zookeeper services will start automatically on these ports.

```
# start zookeeper
./kafka_2.11-2.3.0/bin/zookeeper-server-start.sh-daemon/kafka_2.11-
2.3.0/config/zookeeper.properties
# start kafka
./kafka_2.11-2.3.0/bin/kafka-server-start.sh-daemon/kafka_2.11-
2.3.0/config/server.properties
```

3.1.4 Apache Spark Installation

An Apache Spark installation is required. Apache Spark version 2.4.3 was installed in the root directory. [28]



3.1.5 Flask, Chart.js and Leaflet.js Installation

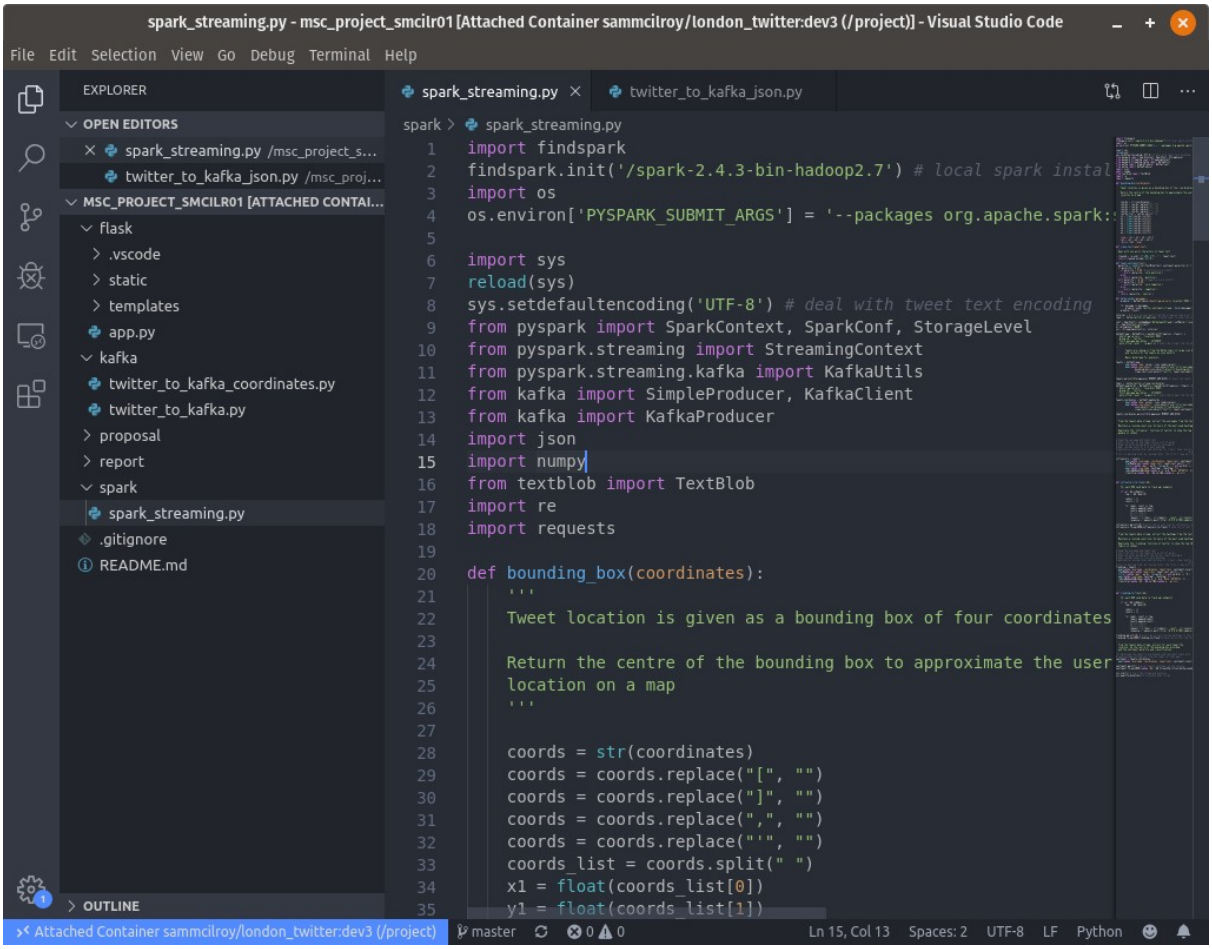
Flask was installed as a standard python library. [29] The Chart.js and Leaflet.js libraries are included as JavaScript files which can be referenced from code developed through Flask.

3.1.6 DockerHub

Following the steps outlined above I have created a docker image which contains all of the requirements required to develop, run and deploy the project as intended. The image can be pulled from the repository and run on any Linux, OSX, or Windows machine with Docker installed. Access to this private repository to pull the image can be granted on request to markers.

3.1.7 Visual Studio Code

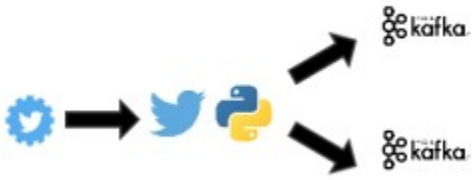
The Visual Studio Code can be installed with Docker and Remote Development Tools which allow for connections to Docker container folders. This allowed me to develop code directly onto the container from the host OS using a well featured code editor and its GUI.



3.2 Data Ingestion

This section describes the process taken in the data ingestion phase, describing the data and how it is collected.

3.2.1 Summary and Design Diagram



For the data ingestion stage, the goal is to collect the Twitter Stream API tweets as JSON objects and 'produce' them as messages to two Kafka topics, using the Tweepy and Kafka-Python libraries, to be later 'consumed' at the data processing stage for analytics using Apache Spark.

3.2.2 Twitter Streaming API and Kafka Topics Creation

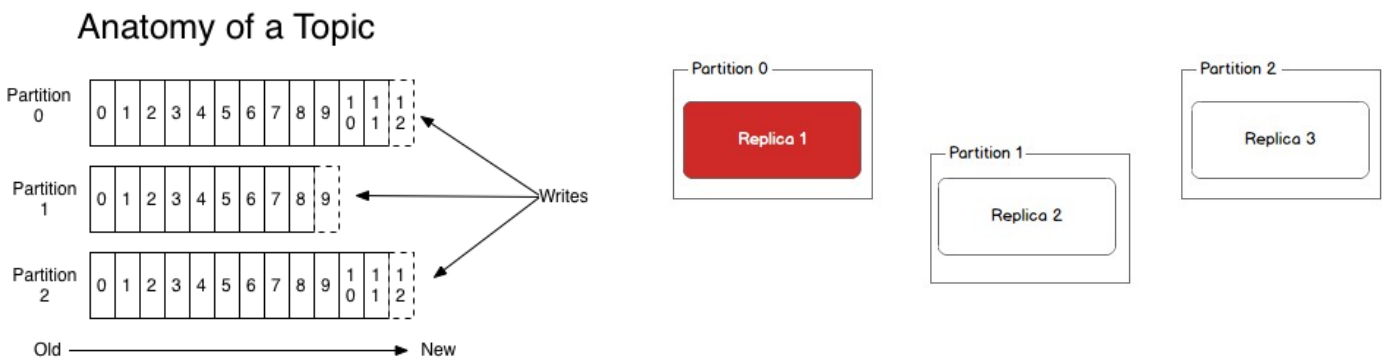
Two Kafka topic were created which will be used to collect and store streams from the Twitter API. The first topic, *kafka_twitter_stream_json*, collects all available tweets from the London Area as they occur in real time. This will give the maximum amount of data possible for analysis. The second topic, *kafka_twitter_stream_coordinates*, collects all tweets from the London Area where the **exact** location coordinates of the user are known (given as a single latitude and longitude), rather than just a general area (given as a four coordinate bounding box) which can be quite large and imprecise, often covering large areas. This information is essential for the the sentiment map to be developed for the front-end. However, as this information is only provided if the user has opted in to sharing their location and is also using a GPS connected device the number of tweets with this information is far less. This issue is further compounded by the Twitter Stream API's restriction, as discussed previously, to a 1% stream.

Initial testing showed that the full stream could expect 1 to 2 tweets per second, whereas the stream of coordinate tagged data was showing 2-3 per minute. For this reason I decided to use both streams, the full stream to focus on the running totals of trending topics and tagged 'influencer' users and the coordinate tagged stream to be used for the sentiment map. This gives the most amount of data, and hence the most accuracy possible, to each of the visualisations. However, if I did have access to a larger volume of streaming data from a paid for Twitter API, which is cost prohibitive for the scope of this project, I would reconsider using a single stream of data only.

The topics were created using the Kafka script/command for topic creation, e.g.

```
> bin/kafka-topics.sh --create --bootstrap-server localhost:9092 --
replication-factor 1 --partitions 1 --topic kafka_twitter_stream_json
```

Two important arguments in the topic creation process to note are the replication-factor and the number of partitions. Basically the number of partitions is how many nodes the topic is set to be distributed across the replication-factor is the number of times the data is to be replicated during distribution [30]:



For development purposes in this project I am working on a Docker container image with a single Kafka/Spark node setup so the partitions and replication factor are set to 1. However, if/when the image is deployed to a distributed server or cloud environment these arguments could be changed to suit. This is in line with my overarching goal of the project being able to be deployed to run larger/faster streams of data with minimal changes to the code and environment.

3.2.3 Tweepy Connection, Pulling JSON Tweets and Producing as Kafka Messages

The Tweepy python library can be used to create a connection to the Twitter Streaming API using the required OAuth authentication and the API tokens provided by the Twitter Developer account and 'listen' to tweets to consume as they come in real time, filtered by any necessary criteria.

The Tweepy Class `StdOutListener()` is used as a listener to gather new tweets. It contains three key methods which can be overridden to customize the data collected: `on_data, on_error` and `on_limit`:

```
16  class StdOutListener(StreamListener):
17      def on_data(self, data):
18          producer.send_messages("kafka_twitter_stream_json", data.encode('utf-8'))
19          #print (data)
20          return True
21      def on_error(self, status):
22          print (status)
23      def on_limit(self,status):
24          print ("Twitter API Rate Limit")
25          print("Waiting...")
26          time.sleep(15 * 60) # wait 15 minutes and try again
27          print("Resuming")
28          return True
```

On each data event ‘on_data’ I am using the Kafka-Python library to create a producer and send the tweet data as a message to the required Kafka topic:

```
                kafka = KafkaClient("localhost:9092")
                producer = SimpleProducer(kafka)
producer.send_messages("kafka_twitter_stream_json", data.encode('utf-8'))
```

Data is encoded as UTF-8 to ensure compatibility with non-standard tweet characters such as emojis and tags.

Errors can be handled using the on_error method, I have chosen to have errors printed to the console for testing and evaluation purposed but allow the production to continue. Non-critical errors which do not disrupt the flow of data can be overlooked to some extent during development and proof of concept stages.

The on_limit method allows for working around limits set by Twitter on their Streaming API, especially the non-paid for connections. There are arbitrary limits set by Twitter based on current usage and capacity where a limit message is sent to a listener such as Tweepy to cut off the stream. Using the on_limit method I have set a waiting time of 15 minutes if this should occur, at which point the stream can resume. This is a generally accepted wait time among other developers working with Twitter data. Using this method I have not had any issues with the stream being terminated due to a Twitter limit message.

As detailed in the previous chapter the Twitter Streaming API provides a stream of tweets in real time with each tweet represented as a JSON object. Aspects of the JSON can be used to ‘filter’ the stream based on selection criteria. Key to this project, a filter on the location information in the JSON can be provided in the form of a ‘bounding box’ a set of four coordinates, forming a polygon, which define an area. I have used four coordinates producing a large square over Central London and surrounding areas of the city. This was fed into the filter argument, thus listening only for tweets from that area:

```
# Central London Boundary Box, covers main areas of the city
london = [-0.2287, 51.4110, -0.0294, 51.5755]

stream = Stream(auth, l, tweet_mode='extended')
stream.filter(locations=london, languages=["en"])
```

To simplify the sentiment analysis process, I am also using the filter arguments to listen only for tweets marked as English language. The tweet_mode argument for the string also allows me to pull in tweet data with the full text of the tweet without truncation which is useful for a more accurate sentiment analysis.

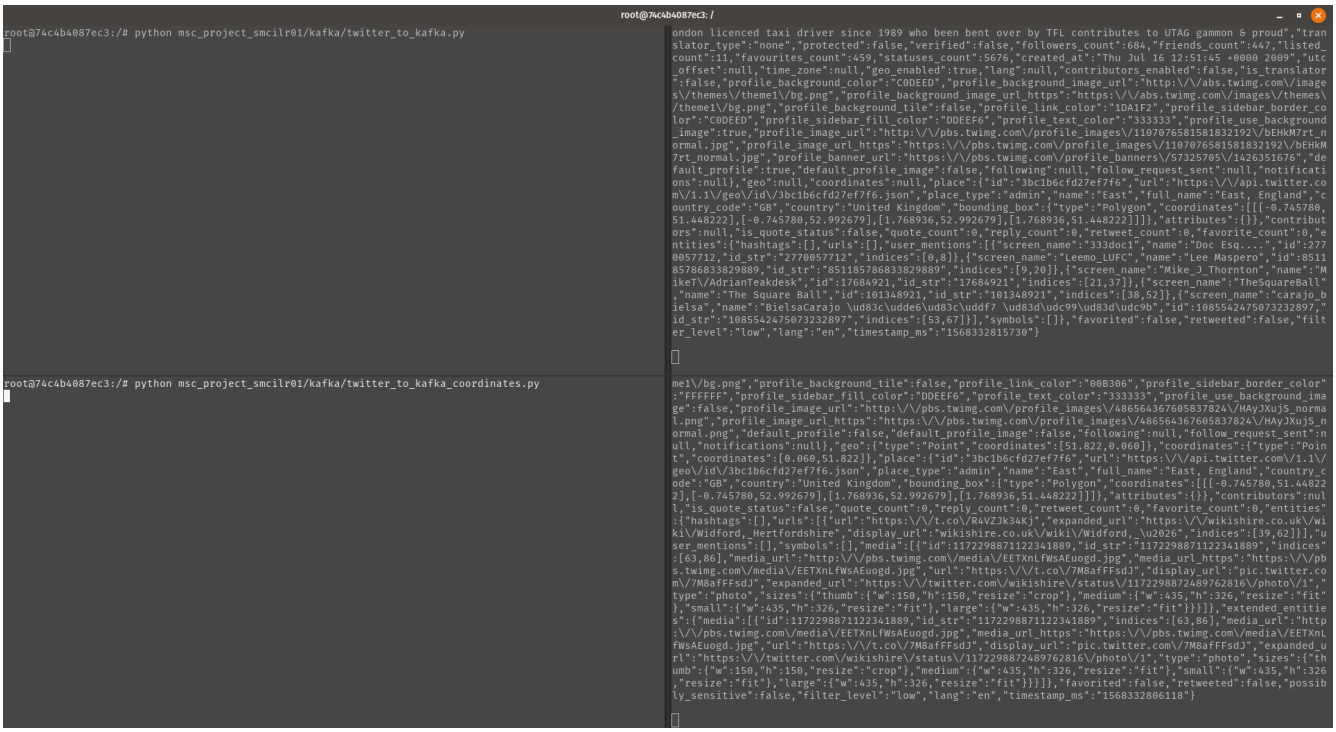
The code below summarises how the Tweepy and Kafka-Python libraries can be used in collaboration to produce a stream of all English language tweets within a chosen area, London in this case, and produce those tweets as JSON formatted messages to an Apache Kafka topic:

```
1  from tweepy.streaming import StreamListener
2  from tweepy import OAuthHandler
3  from tweepy import Stream
4  from kafka import SimpleProducer, KafkaClient
5  import json
6  import time
7
8  access_token = "1330365234-xDjSixFZfSeboDSkHS0WgNvOu5zZw4HeUL8ijVq"
9  access_token_secret =  "QuhhHxIMSxVC2QhVqaxtdgZtc4pyJBWVg2C6D5IHCH9ph"
10 consumer_key =  "JES0pDVJW2WCscy1LhFFMxz4A"
11 consumer_secret =  "uOoW3PCx8nI0kIfsifXfCibYwaeMrHh73TrV2TyuILL9vR9Bdx"
12
13 # Central London Boundary Box, covers main areas of the city
14 london = [-0.2287, 51.4110, -0.0294, 51.5755]
15
16 class StdOutListener(StreamListener):
17     def on_data(self, data):
18         producer.send_messages("kafka_twitter_stream_json", data.encode('utf-8'))
19         #print (data)
20         return True
21     def on_error(self, status):
22         print (status)
23     def on_limit(self,status):
24         print ("Twitter API Rate Limit")
25         print("Waiting...")
26         time.sleep(15 * 60) # wait 15 minutes and try again
27         print("Resuming")
28         return True
29
30 kafka = KafkaClient("localhost:9092")
31 producer = SimpleProducer(kafka)
32 l = StdOutListener()
33 auth = OAuthHandler(consumer_key, consumer_secret)
34 auth.set_access_token(access_token, access_token_secret)
35 stream = Stream(auth, l, tweet_mode='extended')
36 stream.filter(locations=london, languages=["en"])
37
```

In order to customise this collection and production further to collect only the tweets with exact location coordinates attached, the JSON structure of the Tweet Objects can be used to further filter/select the data within the on_data method. Below, the data is encoded as JSON as it is collected to exploit the structure to filter only those where the coordinates field is populated (***decoded.get("coordinates") is not None***) and produce only these matches to the Kafka topic:

```
17     def on_data(self, data):
18         decoded = json.loads(data)
19         if decoded.get("coordinates") is not None:
20             producer.send_messages("kafka_twitter_stream_coordinates", data.encode('utf-8'))
21         #print (data)
22         return True
```

At this point, the two Apache Kafka topics have been set up to collect and store the JSON formatted tweets. The Kafka ‘Console Consumer’ script can be used to consume and display the contents of topics in the terminal, this is useful to show the data is being collected and looks correct:



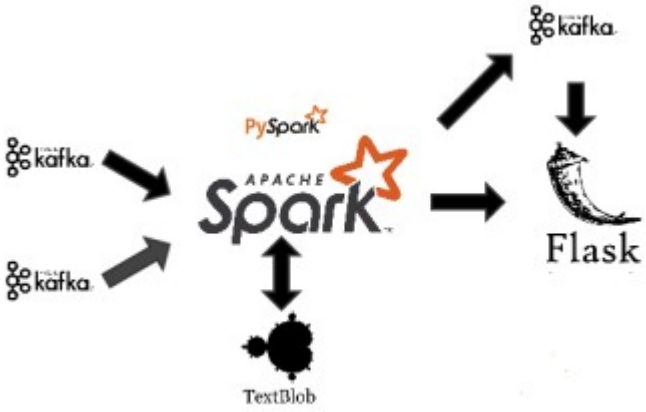
JSON Tweet Objects being successfully produced to the two Kafka topics

The two Apache Kafka topics are now being produced to successfully, containing JSON tweet data from the London area. These topics represent the data in real time and are the basis for the next stage of data processing and analysis.

3.3 Data Processing and Analysis

This section describes the process taken in the data processing and analysis phases.

3.3.1 Summary and Design Diagram



For the data processing and analysis stage, the tweets being produced to the two Apache Kafka topics, as detailed in the previous section, are consumed using the Kafka-Python library and analysed in Apache Spark and TextBlob to create three outputs:

- an Apache Spark DataFrame (RDD), created every 2 seconds, showing the top ten trending topics in London, indicated by tweet text beginning with a hashtag #. Each RDD is then sent to Flask.

- an Apache Spark DataFrame (RDD), created every 2 seconds, showing the top ten influential users in London, indicated by tweet text beginning with a user tag @, indicating that the tweet is referencing that user. Each RDD is then sent to Flask.

- an Apache Spark DataFrame (RDD), created every 2 seconds, showing tweet coordinates and analysed polarity and sentiment levels for every tweet processed. This is then 'produced', using Kafka-Python, to a new Apache Kafka topic which will form a real time stream of tweet locations and corresponding polarity and sentiment levels to be pulled and visualised by the Flask/Leaflet.js London sentiment map.

3.3.2 Consuming Kafka Topics, Data Collection and Preparation

The PySpark library was used to create a direct stream from the Apache Kafka topics collecting and storing the tweet data.

A Spark StreamingContext, a connection to a Spark cluster, was created and named. The name chosen is arbitrary and was chosen as “KafkaTweetStream”. The URL for the cluster (master) was set to the localhost, the single node Spark cluster set up previously on the Docker container. The log level (what is produced to the log files) was set to “WARN” to show any warnings during runtime. PySpark also contains Kafka Utilities to communicate with the Kafka topics, a connection was set up to stream from the Kafka topics stored on the localhost, again the single node Kafka setup on the Docker container. The interval, how often to consume from the topic, and the topic name were set as variables to be reused.

```
 9  from pyspark import SparkContext, SparkConf, StorageLevel
10  from pyspark.streaming import StreamingContext
11  from pyspark.streaming.kafka import KafkaUtils
```

```
76  interval = 2 # pick up new tweets from the stream as they occur (every 2 seconds)
77  topic = 'kafka_twitter_stream_json' # kafka topic to subscribe to, contains stream of tweets in json
78
79  conf = SparkConf().setAppName("KafkaTweetStream").setMaster("local[*]")
80  sc = SparkContext(conf=conf)
81  sc.setLogLevel("WARN")
82  ssc = StreamingContext(sc, interval)
83
84  kafkaStream = KafkaUtils.createDirectStream(ssc, [topic], {
85    'bootstrap.servers': 'localhost:9092',
86    'group.id': 'twitter',
87    'fetch.message.max.bytes': '15728640',
88    'auto.offset.reset': 'largest'}) # subscribe/stream from the kafka topic
```

The required data fields from the JSON structure were pulled into Spark Resilient Distributed Datasets (RDDs) for processing. The user, location, and text data were collected. Actions were performed on the text using python function, to clean the text formatting and ensure consistent UTF-8 formatting, determine coordinates if no exact were given and to determine the sentiment and polarity:

```
90  '''
91      Tweets are coming in from the Kafka topic in large json blocks, extract the username, location
92      and contents of the tweets as json objects.
93
94      Basic dataframe for analysis.
95  '''
96  tweets = kafkaStream. \
97      map(lambda (key, value): json.loads(value)). \
98      map(lambda json_object: (json_object["user"]["screen_name"],
99              bounding_box(json_object["place"]["bounding_box"]["coordinates"]),
100             clean_text(json_object["text"]), tweet_sentiment(clean_text(json_object["text"]))))
```

```
47  def clean_text(tweet_text):
48    '''
49    Deal with non-ascii characters in tweet text
50    '''
51    cleaned = re.sub(r'[^\x00-\x7f]',r'', tweet_text)
52    return cleaned.encode('utf-8')
```

More detail on the sentiment scoring function will be given in subsequent sections of this chapter.

In order to reuse the same data to continually refresh the analysis, the dataframes were retained in persistent memory and disk using the following command:

```
103   tweets.persist(StorageLevel.MEMORY_AND_DISK) # retain the tweets dataframe in persistent memory and disk to allow continued analysis of the stream
```

A similar process was used to collect and persistently store the exact coordinate data as a dataframe:

```
112   tweets_coordiates = kafkaStreamCoords. \
113           map(lambda (key, value): json.loads(value)). \
114           map(lambda json_object: (json_object["user"]["screen_name"],
115                   json_object["coordinates"]["coordinates"],
116                   clean_text(json_object["text"]), tweet_sentiment(clean_text(json_object["text"])))))
117
118   tweets_coordiates.persist(StorageLevel.MEMORY_AND_DISK)
```

3.3.3 Data Processing, Apache Spark Analytics

Running Totals Data

The first dataframe processed above, *tweets*, was used to generate a continued ordered count of hashtag topics and tagged users in the text field of the tweets.

```
168   '''
169     From the tweets data stream, extract the hashtags from the text.
170
171     Maintain a running count over 24 hours of the most used hashtags.
172
173     Replicate the 'trending' function of twitter to show the top 10 talked about
174     topics in London.
175   '''
176   # keep the username and tweet text
177   # split the text of the tweet into a list of words
178   # keep the words marked with the twitter topic hashtag #
179   # map - assign count of 1 to each hashtag
180   # maintain a running total and continue for 24 hours, show results at intervals of 2 seconds
181                                                   # ensure stream is running without issue
182   # sort in descing order by running total, the first x rows will now indicate the most popular #
183   trending = tweets. \
184     map(lambda (username, coordinates, tweet_text, sentiment_level): (username, tweet_text)). \
185     flatMapValues(lambda tweet_text: tweet_text.split(" ")). \
186     filter(lambda (pair, word): len(word) > 1 and word[0] == '#'). \
187     map(lambda (username, hashtag): (hashtag, 1)). \
188     reduceByKeyAndWindow(lambda a, b: a+b, None, 60*60*24, 2). \
189     transform(lambda rdd: rdd.sortBy(lambda a: -a[-1]))
```

The *tweets* dataframe was transformed into continually updating micro RDDs. The text was split into its constituent words, words indicating @users or hashtags # (as in the above code example) were retained, and a running total maintained. A window of 24 hours is also given with an interval of two seconds, refreshing the data continuously for a one day period. [33] I then used PySpark pprint() functions to view the running totals being successfully populated when the code is run through Spark:

```
212   trending.pprint(10) # print the current top ten hashtags to the console
```

The top ten from each of the two running totals could then be viewed through the terminal to monitor the data being produced as expected:

```
-------------------------------------------
Time: 2019-09-14 12:45:12
-------------------------------------------
('@prydwen3', 3)
('@JCWI_UK', 2)
('@yezzer', 2)
('@SpursOfficial', 2)
('@JamesFenn90', 2)
('@MattTooze', 2)
('@lederroux', 2)
('@David_Cameron', 2)
('@JarroldKen,', 2)
('@thetimes', 2)
...

-------------------------------------------
Time: 2019-09-14 12:45:12
-------------------------------------------
('#LFW', 3)
('#Brexit', 2)
('#LIVNEW', 2)
('#fashionshow', 2)
('#ExtinctionRebellion', 2)
('#fashion', 2)
('#onstreet', 1)
('#SNP', 1)
('#brexit', 1)
('#britsfa', 1)
...
```

Running totals, top ten hashtags and tagged users, being collected, totaled and sorted

Once the running totals were confirmed to be collating successfully as expected, functionality was added to push the data to Flask, through API endpoints set up and detailed in the next section. For each RDD generated, i.e. the current sorted hashtags/tagged users every 2 seconds, the RDD was passed into the below function, taking the top 10 entries, splitting the data into 2 arrays representing the labels and counts and posted using the Python requests library to the Flask API [32]:

```
194  def trending_to_flask(rdd):
195      '''
196      For each RDD send top ten data to flask api endpoint
197      '''
198      if not rdd.isEmpty():
199          top = rdd.take(10)
200
201          labels = []
202          counts = []
203
204          for label, count in top:
205              labels.append(label)
206              counts.append(count)
207              #print(labels)
208              #print(counts)
209              request = {'label': str(labels), 'count': str(counts)}
210              response = requests.post('http://0.0.0.0:5001/update_trending', data=request)
211
212  trending.pprint(10) # print the current top ten hashtags to the console
213  trending.foreachRDD(trending_to_flask) # send the current top 10 hashtags rdd to flask for visualisation
```

As the script, and the Flask application, are running, the data can be viewed as the expected two arrays into the Flask API endpoint using a browser window:

Sentiment Analysis

For the sentiment analysis map, which is outlined in more detail in the next chapter, I intend to plot points based on the known coordinates of the user/tweet and to color code the markers based on how positive or negative the sentiment of the text is.

The TextBlob Python library `sentiment` method returns a polarity score as a continuous value between -1 (very negative) and 1 (very positive). The values are derived from a NaiveBayes text classification algorithm rained on labeled movie reviews texts. [31] As discussed in the previous chapter, I have decided to use the TextBlob library rather than training a custom model to simplify the design process. I can be confident in producing reasonably accurate and trustworthy sentiment analysis scores while being able to focus more development time on the data ingestion/data pipelines and front-end visualisation stages which proved to be more time consuming than originally planned. However, I am keen to investigate implementing a custom sentiment analysis as a future improvement and learning opportunity.

The TextBlob sentiment method was used create my own custom method to sort tweet text into five distinct categories: very positive, positive, neutral, negative and very negative. These values will correspond to the colouring chosen for the sentiment map. These colours will show at a glance the general sentiment across different areas of the city as the tweets analysed builds up over time during the course of 24 hours.

```
54  def tweet_sentiment(text):
55    polarity = round(float(TextBlob(text).sentiment.polarity),2) # tweet text polarity score between -1, negative, and 1 positive
56    if polarity > 0.10:
57      if polarity > 0.60: # positive score cutoffs
58        return (polarity, 'very positive')
59      else:
60        return (polarity, 'positive')
61    elif polarity < -0.10: # negative score cutoffs
62      if polarity < -0.60:
63        return (polarity, 'very negative')
64      else:
65        return (polarity, 'negative')
66    else:
67      return (polarity, 'neutral')
```

As tweets are consumed from the Apache Kafka topic this method is immediately applied to the text as the Spark RDDs are created:

```
105 topic = 'kafka_twitter_stream_coordinates'
106 kafkaStreamCoords = KafkaUtils.createDirectStream(ssc, [topic], {
107   'bootstrap.servers': 'localhost:9092',
108   'group.id': 'twitter',
109   'fetch.message.max.bytes': '15728640',
110   'auto.offset.reset': 'largest'}) # subscribe/stream from the kafka topic
111
112 tweets_coordiates = kafkaStreamCoords. \
113         map(lambda (key, value): json.loads(value)). \
114         map(lambda json_object: (json_object["user"]["screen_name"],
115                 json_object["coordinates"]["coordinates"],
116                 clean_text(json_object["text"]), tweet_sentiment(clean_text(json_object["text"]))))
117
118 tweets_coordiates.persist(StorageLevel.MEMORY_AND_DISK)
```

The coordinates and sentiment are then fed into a new Apache Kafka topic which will form the real-time feed on the sentiment analysis London map:

```
-------------------------------------------
Time: 2019-09-13 21:41:16
-------------------------------------------
([-0.08651547, 51.50436548], (0.5, 'positive'))
([-0.12496349, 51.51314651], (1.0, 'very positive'))
```

Coordinates and custom sentiment tuple being produced

As previously, the Kafka-Python library is used to create the Kafka producer. A custom function was created to create the producer and send messages to the topic:

```
69  def kafka_sender(messages):
70      producer = KafkaProducer(bootstrap_servers='localhost:9092')
71
72      for message in messages:
73          producer.send('twitter_sentiment_stream', bytes(message))
74      producer.flush()
```

The tweets RDD is mapped to coordinates and sentiment level and for each newly mapped RDD the custom Kakfa producer function is called:

```
215 '''
216   From the tweets data stream, extract for each tweet the
217   location, by the centre of the bounding box provided,
218   and the sentiment polarity and classification
219 '''
220  # reduce/map the tweet to coordinates and sentiment tuple only
221 # send minimum information from the tweet to flask app
222 sentiment = tweets_coordiates. \
223   map(lambda (username, coordinates, tweet_text, sentiment_level): (coordinates, sentiment_level))#.window(60*60, 2)
224
225 sentiment.pprint() # print tweets sentiment to the console
226 sentiment.foreachRDD(lambda rdd: rdd.foreachPartition(kafka_sender))
```

The Kafka console consumer script can be used to view the topic being successfully populated:

```
sam@xps-15:~$ project
root@74c4b4087ec3:/# ./kafka_2.11-2.3.0/bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic twitter_sentiment_stream
([-0.1952233, 51.58762167], (0.5, 'positive'))
([-0.12731805, 51.50711486], (0.07, 'neutral'))
([-0.04101183, 51.51183971], (0.44, 'positive'))
([-0.17777778, 51.5025], (-0.07, 'neutral'))
([-0.0601892, 51.4642345], (0.0, 'neutral'))
([-0.2024064, 51.4454282], (0.7, 'very positive'))
([-0.1499052, 51.5107011], (0.0, 'neutral'))
([-0.12731805, 51.50711486], (0.0, 'neutral'))
([-0.12731805, 51.50711486], (0.38, 'positive'))
([-0.09140041, 51.427282], (0.0, 'neutral'))
^CProcessed a total of 10 messages
root@74c4b4087ec3:/#
```

Summary

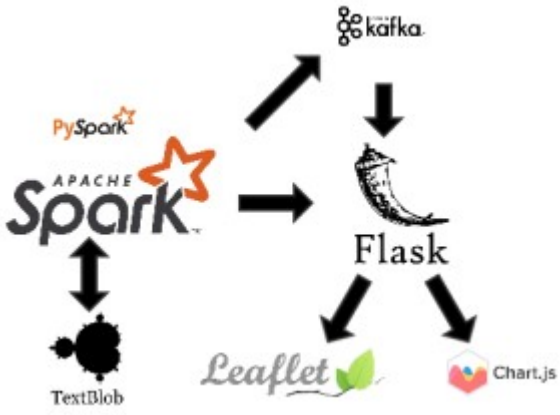
At this stage in the implementation process the data has been shown to be successfully produced and consumed. The running top ten trending hashtags and tagged influential users are being produced every 2 seconds and being successfully passed as arrays to the relevant Flask API endpoints. The sentiment analysis classifications and locations of each tweet are being successfully produced to an Apache Kafka topic.

All of the required data is being transformed and analysed as expected and is available in order to produce the required Flask visualisations as detailed in the design stage.

3.4 Front-end Visualisations

This section details the front-end flask application implementation and JavaScript based visualisations.

3.4.1 Summary and Design Diagram



For the front-end visualisation production stage, data is pushed from the the Apache Spark processes detailed in the previous section both directly to Flask, for the running total dataframes, and into an Apache Kafka topic for the real-time sentiment analysis mapping.

Flask forms the basis of the web front end and data collection routes and the visualisations are produced using JavaScript libraries, Chart.js and Leaflet.js.

This stage has the following general outputs:

- A web based front end, built on Python's Flask, capable of collecting and storing data and hosting visualisations, these visualisations are hosted on HTML pages which are mapped using Flask routes
- Navigation and user interface features, CSS and Bootstrap theming using a custom Bootstrap navigation bar which is fully responsive for viewing on various platforms and screen resolutions/orientations.
- A bar-chart visualisation, updating every 2 seconds, showing the current top ten trending topics from London based tweets, based on hashtags in tweet text
- A bar-chart visualisation, updating every 2 seconds, showing the current top ten influential twitter users in London, based on users being tagged in London based tweet text
- An interactive map of London, showing in real time (as tweets are produced) the location of the tweet marked on the map, with markers colour coded to sentiment classification. This is designed to build up over time and show the general sentiment levels across London areas.

3.4.2 Flask Implementation

The Flask web application framework was implemented to handle the server side processing of data, to map the request routes and to define the HTTP routes of the web front end interface and visualisations.

The Flask app was initialised and global variables were created to hold the data pushed from Spark to populate the running total bar charts:

```
1   from flask import Flask, jsonify, request, Response, render_template
2   from pykafka import KafkaClient
3   import ast
4
5   def get_kafka_client():
6       return KafkaClient(hosts='localhost:9092')
7
8   app = Flask(__name__)
9
10  trending = {'labels': [], 'counts': []}
11  influencers = {'labels': [], 'counts': []}
```

Flask API endpoints were defined to update data pushed from Spark in the data analytics phase. The global variables are populated with this data as it is pushed, using the POST method, with the Python ast library handling native Python abstract syntax grammar in order to interact with the request data. [32][34]

```
61  @app.route('/update_trending', methods=['POST'])
62  def update_trending():
63      global trending
64      if not request.form not in request.form:
65          return "error", 400
66
67      trending['labels'] = ast.literal_eval(request.form['label'])
68      trending['counts'] = ast.literal_eval(request.form['count'])
69
70      return "success", 201
```

The data could then be viewed populating in the browser against that route:

Flask routes were then set up to give access to the data for the charts by providing access to the global variables which can be then used in the JavaScript chart definitions:

```
45  @app.route('/refresh_trending')
46  def refresh_trending():
47      global trending
48
49      return jsonify(
50          Label=trending['labels'],
51          Count=trending['counts'])
```

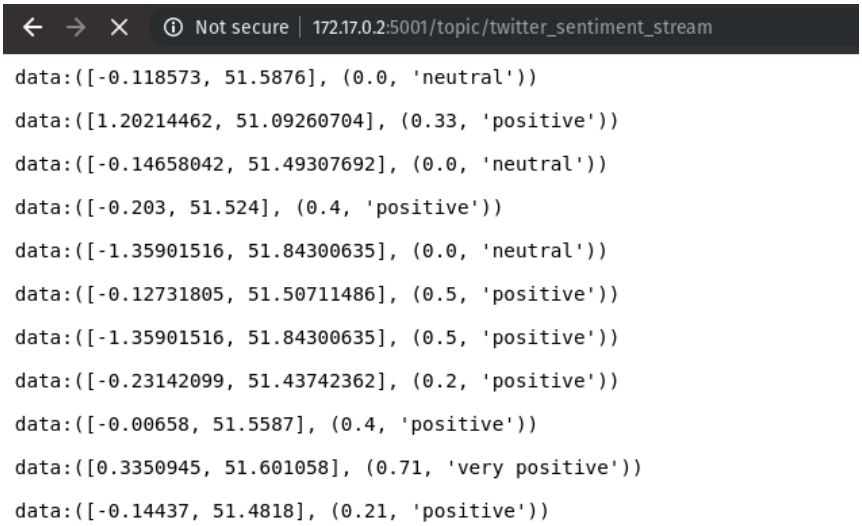
as well as Flask routes for the HTML pages/site navigation:

```
38  @app.route("/hashtags.html")
39  def get_hashtags_page():
40      global trending
41
42      return render_template('hashtags.html',
43      trending=trending)
44
```

To produce the real-time sentiment map I also required a Flask route set up to collect data from Apache Kafka topics. The below custom Flask route uses server-sent events to collect data from any running Kafka topic into the specified route. [35]

```
21  @app.route('/topic/<topicname>')
22  def get_messages(topicname):
23      client = get_kafka_client()
24      def events():
25          for i in client.topics[topicname].get_simple_consumer():
26              yield 'data:{0}\n\n'.format(i.value.decode())
27      return (Response(events(), mimetype="text/event-stream"))
```

Data from running Kafka topics can then be seen populating in the browser. Below is the Kafka stream used to populate the sentiment map:



3.4.3 Chart.js Visualisations

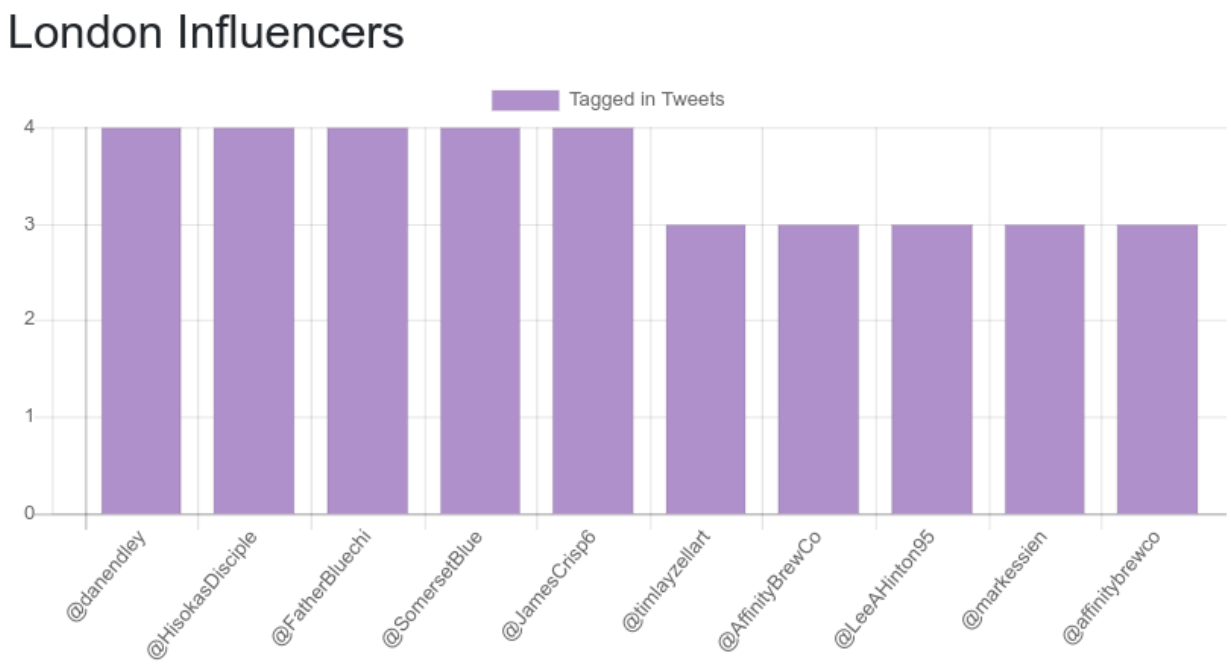
Chart.js was implemented to use JavaScript to create the running total bar charts connected to the data coming into the API endpoints set up in the previous section:

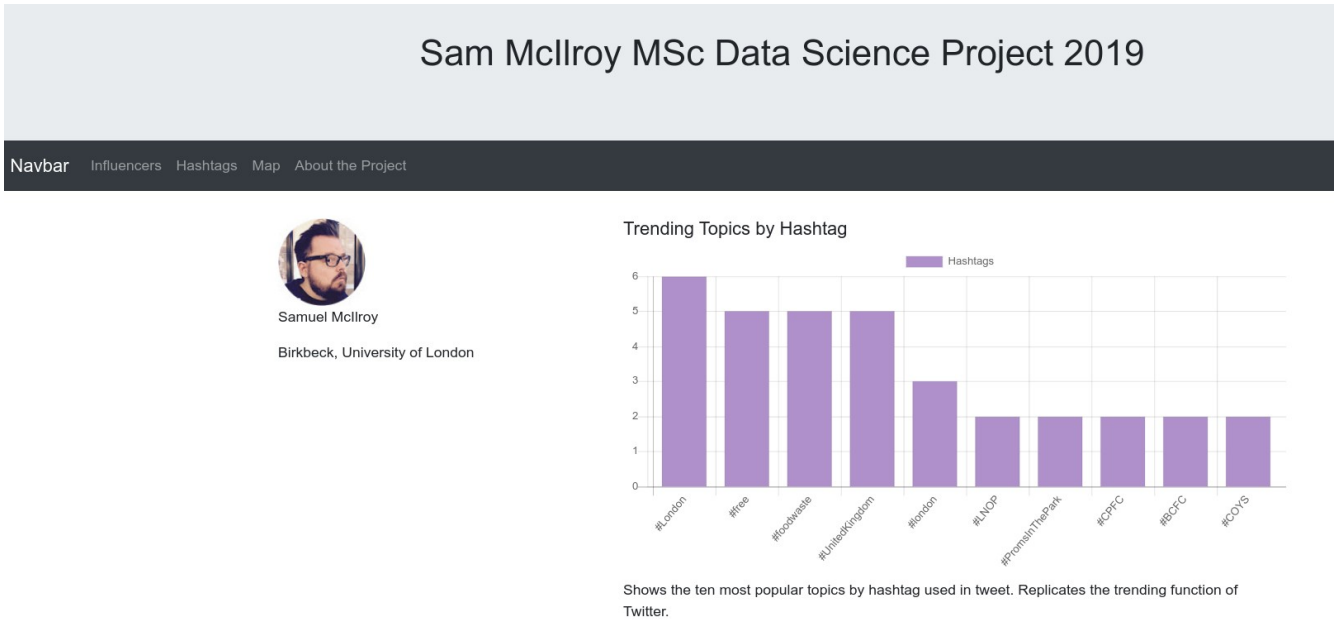
```
var ctx = document.getElementById('trending_chart').getContext('2d');
var trending_chart = new Chart(ctx, {
    type: 'bar',
    data: {
        labels: [],
        datasets: [{
            label: 'Hashtags',
            data: [],
            backgroundColor:' #af90ca',
            borderWidth: 1
        }]
    },
    options: {
        scales: {
            yAxes: [{
                ticks: {
                    beginAtZero: true,
                    callback: function(value) {if (value % 1 === 0) {return value;}}
                }
            }]
        }
    }
});
```

JavaScript was used to populate arrays with the array data stored in the API endpoints and to use these to populate the labels and data arrays attached to the chart. This was set to repeat every 2 seconds. In this way the chart's data arrays would be populated with new top 10 data each time: [32]

```
var src_data_trending= {
    labels: [],
    counts: []
}

setInterval(function(){
    $.getJSON('/refresh_trending', {
    }, function(data) {
        src_data_trending.labels = data.Label;
        src_data_trending.counts = data.Count;
    });
    trending_chart.data.labels = src_data_trending.labels;
    trending_chart.data.datasets[0].data = src_data_trending.counts;
    trending_chart.update();
},2000);
```

Links to the chart JavaScript files were embedded into the relevant HTML pages for display in the browser:



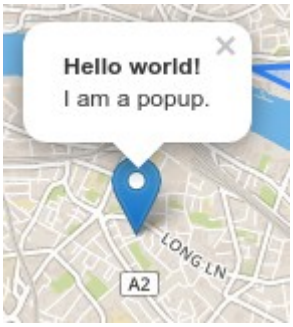


3.4.4 London Sentiment Map, Leaflet.js

JavaScript employing the Leaflet.js library was applied to the production of the London Sentiment Map. Leaflet.js works in conjunction with the mapbox API, based on open source OpenStreetMap data, to display and append to map visualisations in the browser. I set up a JavaScript/Leaflet.js script to connect to the mapbox API and display a map of London:

```
var mymap = L.map('mapid').setView([51.505, -0.09], 11);

L.tileLayer('https://api.tiles.mapbox.com/v4/{id}/{z}/{x}/{y}.png?access_token={accessToken}', {
  attribution: 'Map data &copy; <a href="https://www.openstreetmap.org/">OpenStreetMap</a> contributors, <a href="https://creativecommons.org/licenses/by-sa/2.0/">CC
  maxZoom: 18,
  id: 'mapbox.streets',
  accessToken: 'pk.eyJ1Ijoic2FtbWNpbHJveSIsImEiOiJjazA5aTdkaTMwOGZ3M3BtcTIzbWVvd2VmIn0.eOXHT1jSc7Ut92BvnhJGJQ'
}).addTo(mymap);

```

Leaflet.js allows for custom markers to be added to coordinates on the map with user defined colours and messages:



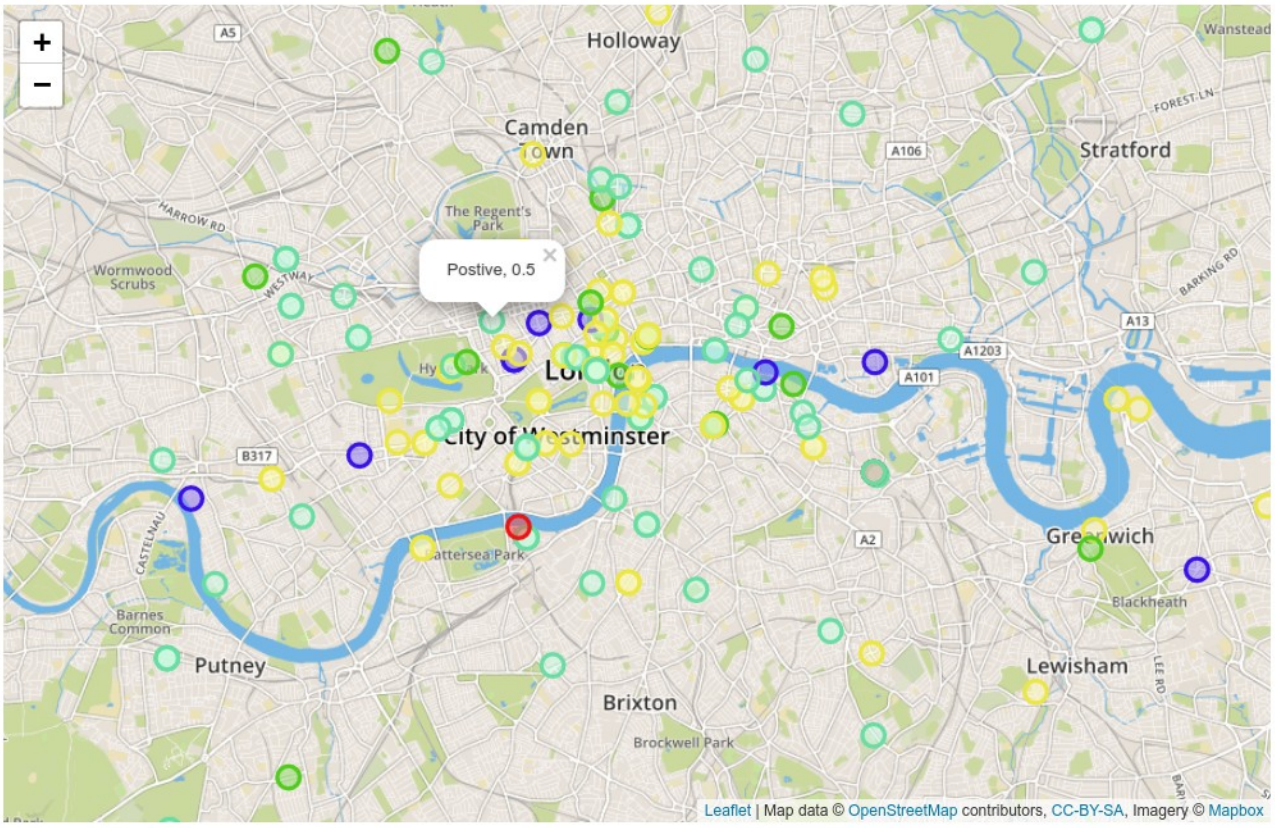
I added to the map's script a connection to the Kafka topic API endpoint data source, as detailed previously, and to extract latitude, longitude, polarity and sentiment as separate variables by cleaning and splitting the data into arrays and indexing:

```
var source = new EventSource('/topic/twitter_sentiment_stream');
source.onmessage = function(event) {
  obj = String(event.data);
  var strnew = obj.replace(/[{()}]/g, '').replace(/[\[\]']+/g,'');
  var elems = strnew.split(',');
  var lat = elems[1];
  var long = elems[0];
  var pol = elems[2];
  var sent = elems[3].trim();
  console.log(lat + ' ' + long + ' ' + pol+ ' ' + sent);
```

I then added if else statements to populate the map with a marker at each coordinate with the colour and pop up text changing based on sentiment:

```
  if (sent.valueOf() == "very positive") {
    var circle = L.circle([lat, long], {
      color: '#50cc1b',
      fillColor: '#8bde68',
      fillOpacity: 0.5,
      radius: 200,
      title: "very positive"
    }).addTo(mymap);
```

A link to the sentiment map JavaScript file was then embedded into the an HTML page for display in the browser:



Viewing the map in the browser we can see the tweets being populated as markers based on their position. Their colour is set based on their sentiment, ranging from dark green, light green, yellow, blue, to red representing very positive, positive, neutral, negative and very negative. Clicking on a marker displays a popup showing the sentiment and associated polarity.

As points begin to build and cluster on the map it begins to create a visualisation indicating general positive or negative sentiment levels across London. The screenshot above shows data collecting for about an hour.

3.4.5 Page Design: Bootstrap and CSS

Bootstrap 4 was chosen to give a consistent CSS framework meaning all design elements, fonts etc., are all pulled in with the link to Bootstrap. Links were used instead of pulling down the Bootstrap files into the project so that any changes to Bootstrap do not effect the project, keeping Bootstrap always up to date [23]:

```
<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.min.css">
<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.4.1/jquery.min.js"></script>
<script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.3.1/js/bootstrap.min.js"></script>
<script src="/static/css/dashboard.css"></script>
```

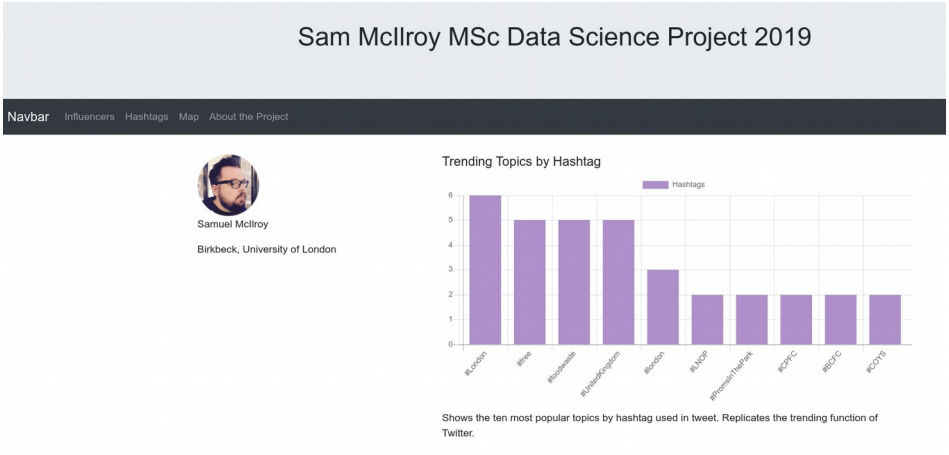
Bootstrap also uses the JQUERY library which implements custom JavaScript to accurately map the page making it simple to include design and theming elements such as navigation bars, headers and footers with minimal coding and web-design knowledge [23]:

```
<nav class="navbar navbar-expand-sm bg-dark navbar-dark">
  <a class="navbar-brand" href="#">Navbar</a>
  <button class="navbar-toggler" type="button" data-toggle="collapse" data-target="#collapsibleNavbar">
    <span class="navbar-toggler-icon"></span>
  </button>
```

Custom CSS was also used to constrain the charts within the chart container as a <canvas> tag on the HTML page:

```
.chart_container{
    width: 45%;
    float: left;
    margin: 1%;
    text-align: center;
}

#trending_area {
  width: 45%;
    float: left;
    margin: 1%;
    text-align: center;

}
```

```
<div class="col-sm-8">
  <h5>Trending Topics by Hashtag</h5>
  <div id="trending_area" class="chart_container">
    <div>
        <canvas id="trending_chart"></canvas>
        <script src="static/js/trending_chart.js"></script>
    </div>
</div>
```



The CSS selector # was used to target the specific chart area on the page, along with the class ***chart_container*** to ensure that the chart stays within the canvas in a fixed location, relative to the page dimensions.

4. Testing

This chapter details the tests and checks performed at each stage to ensure as far as possible that the project is performing as required and expected.

4.1 Testing: Data Ingestion

Kafka Topic Producer: kafka_twitter_stream_json

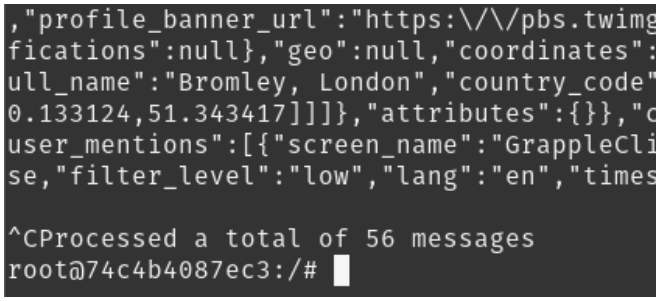
The first Kafka topic to be produced messages to is the topic, ***kafka_twitter_stream_json***. This topic is designed to collect and store the London based tweets, as defined by the provided bounding box of four coordinates.

We are expecting the text of the tweet to be included and for the text to be formatted as UTF-8 to prevent errors caused by unrecognized/incompatible non-alphanumeric characters which unable to be processed.

Initially, technical testing was performed. I used the overridden on_error method from the Tweepy library to print all caught errors as text to the terminal:

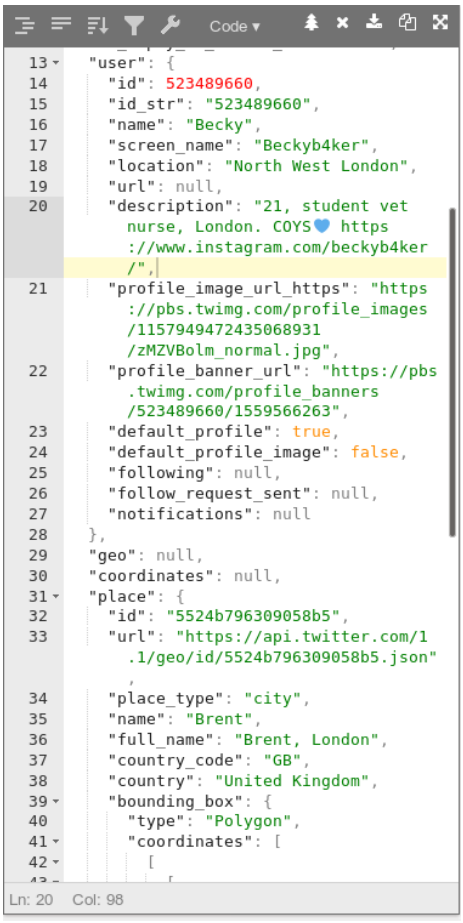
```
21      def on_error(self, status):
22          print (status)
```

The script was allowed to run for a period of 60 minutes during which the terminal was monitored for errors and an additional window was monitored running a Kafka console consumer to observer the data being produced the script and consumed by the topic as expected:



Following technical testing, once the tweets had been observed populating as expected in JSON format I began sense checking/acceptance testing the data, checking the produced data matches expectations.

I used the online tool Code Beautify JSON Viewer [36] to inspect the contents of the produced tweet objects. Showing one example:



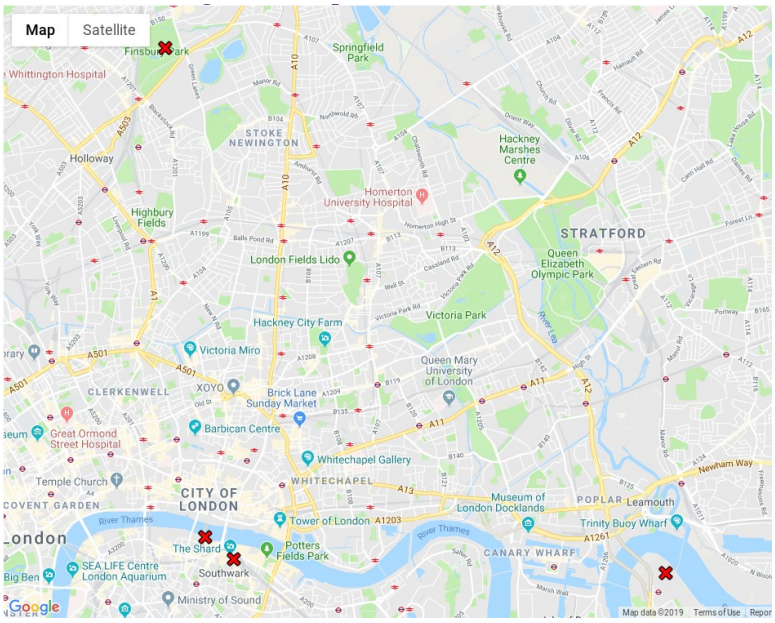
It can be seen that the tweet is properly collecting UTF-8 characters, from the heart emoji present,and that the location is as expected within London. I repeated these checks for several other tweets and all were processed as expected and fell within London coordinates.

Kafka Topic Producer: kafka_twitter_stream_coordinates

The second Kafka topic to be produced messages to is the topic, ***kafka_twitter_stream_coordinates***. This topic is designed to collect and store London based tweets as with the previous topic but in this case filtered further to only tweets where a user has provided their exact location (latitude and longitude) by opting in through a GPS enabled device such as a smartphone, rather than their general location being determined by their user account etc. in which only a general area (four coordinate bounding box) is given which is not accurate enough for mapping due to the fact that this area can be quite wide and many users often share the same general area. The code to produce this stream is identical to that of the previous topic except for the addition of an additional filter against the JSON to show only tweets with the location populated:

```
17      def on_data(self, data):
18          decoded = json.loads(data)
19          if decoded.get("coordinates") is not None:
20              producer.send_messages("kafka_twitter_stream_coordinates", data.encode('utf-8'))
21          #print (data)
22          return True
```

As such the testing was identical to the previous topic so will be omitted here. An additional sense check was added however to ensure that the coordinates were populated, did fall within London, and were sufficiently distinct and spaced from each other (to ensure the coordinates were not general estimates shared among several users). All of the tweets sampled has exact coordinates populated and I used an online tool, 'Copy Paste Map' to plot the coordinates of a random selection of the coordinates for checking and the following was produced:



showing expected unique and spaced tweet location coordinates within London. Following this testing I can say with sufficient confidence that the data is being produced as expected and the deliverables of this stage, namely the two Kafka topics of twitter stream data, have been produced successfully and are working to specification.

4.2 Testing: Data Processing and Analysis

Kafka Topic Consumer: kafka_twitter_stream_json

The analysis/Spark processing at this stage is designed to consume data from the ***kafka_twitter_stream_json*** topic and reduce the JSON data collected to Spark RDDs containing: the user, the coordinates (bounding box), the text (further checked for UTF-8 consistency) and the sentiment/polarity extracted from the text. This filtering of the data is detailed previously, at the testing stage PySpark pprint() statements were used to observe the data being collected, to allow the spark process to run, and to monitor for errors:

```
tweets = kafkaStream. \
        map(lambda (key, value): json.loads(value)). \
        map(lambda json_object: (json_object["user"]["screen_name"],
                bounding_box(json_object["place"]["bounding_box"]["coordinates"]),
                clean_text(json_object["text"]), tweet_sentiment(clean_text(json_object["text"]))))

tweets.pprint(10)
```

The collected tweet data was successfully monitored being collected through the terminal:

```
-------------------------------------------
(u'johngreenn', (51.4603395, -0.11540600000000001), '@AVFCfan1982 @LondonOrat Highly recommended, but I may be biased', (0.16, 'positive'))
```

While monitoring the data in terminal the data was also ‘sense’ checked for text being readable, hashtags and @ symbols being included occasionally, as expected and required for subsequent analysis. The polarity and sentiment being returned as expected was also checked, for example in the above example the value of 0.16 matching to ‘positive’ and the text itself containing positive phrases such as ‘highly recommended’.

These checks were continued for several examples for consistency.

Kafka Topic Consumer: kafka_twitter_stream_coordinates

The consumption of data from the topic ***kafka_twitter_stream_coordinates*** was also monitored as above with similar positive results.

Spark Data Analysis, RDD Creation

During the analytics phase it is expected that Spark will produce, every 2 seconds, an updated list of used hashtags and tagged users in the tweet text sorted by frequency in descending order. Again, PySpark pprint() statements were used to monitor the creation and updating of these dataframes in the terminal and an example is shown below:

```
-------------------------------------------
Time: 2019-09-14 12:45:12
-------------------------------------------
('@prydwen3', 3)
('@JCWI_UK', 2)
('@yezzer', 2)
('@SpursOfficial', 2)
('@JamesFenn90', 2)
('@MattTooze', 2)
('@lederroux', 2)
('@David_Cameron', 2)
('@JarroldKen,', 2)
('@thetimes', 2)
...

-------------------------------------------
Time: 2019-09-14 12:45:12
-------------------------------------------
('#LFW', 3)
('#Brexit', 2)
('#LIVNEW', 2)
('#fashionshow', 2)
('#ExtinctionRebellion', 2)
('#fashion', 2)
('#onstreet', 1)
('#SNP', 1)
('#brexit', 1)
('#britsfa', 1)
...
```

Kafka Topic Producer: twitter_sentiment_stream

During the Spark analysis stage a new Kafka topic is sent messages to/populated forming the stream of coordinates and sentiment to support the London sentiment map on the front-end. As also performed during the Implementation stage, the Kafka console consumer script was used to view the topic being successfully populated and the stream monitored for accuracy/faults:

```
sam@xps-15:~$ project
root@74c4b4087ec3:/# ./kafka_2.11-2.3.0/bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic twitter_sentiment_stream
([-0.1952233, 51.58762167], (0.5, 'positive'))
([-0.12731805, 51.50711486], (0.07, 'neutral'))
([-0.04101183, 51.51183971], (0.44, 'positive'))
([-0.17777778, 51.5025], (-0.07, 'neutral'))
([-0.0601892, 51.4642345], (0.0, 'neutral'))
([-0.2024064, 51.4454282], (0.7, 'very positive'))
([-0.1499052, 51.5107011], (0.0, 'neutral'))
([-0.12731805, 51.50711486], (0.0, 'neutral'))
([-0.12731805, 51.50711486], (0.38, 'positive'))
([-0.09140041, 51.427282], (0.0, 'neutral'))
^CProcessed a total of 10 messages
root@74c4b4087ec3:/#
```

Spark RDDs to Flask API Endpoints

Following the creation of the running total dataframes for the trending topics and tagged users, updated every two seconds (detailed above), these are designed to be pushed as arrays to the designated Flask API endpoints, e.g.:

```
148  def influencers_to_flask(rdd):
149      '''
150      For each RDD send data to flask api endpoint
151      '''
152      if not rdd.isEmpty():
153          top = rdd.take(10)
154
155          labels = []
156          counts = []
157
158          for label, count in top:
159              labels.append(label)
160              counts.append(count)
161              #print(labels)
162              #print(counts)
163              request = {'label': str(labels), 'count': str(counts)}
164              response = requests.post('http://0.0.0.0:5001/update_influencers', data=request)
165
166  influencers.pprint(10) # print the current top ten influencers to the console
167  influencers.foreachRDD(influencers_to_flask) # send the current top 10 influencers rdd to flask for visualisation
```

The printed dataframes from the Spark process and the arrays populated on the API endpoints, visible through a web browser, were monitored simultaneously for consistency to ensure that the data is successfully being pushed to Flask:

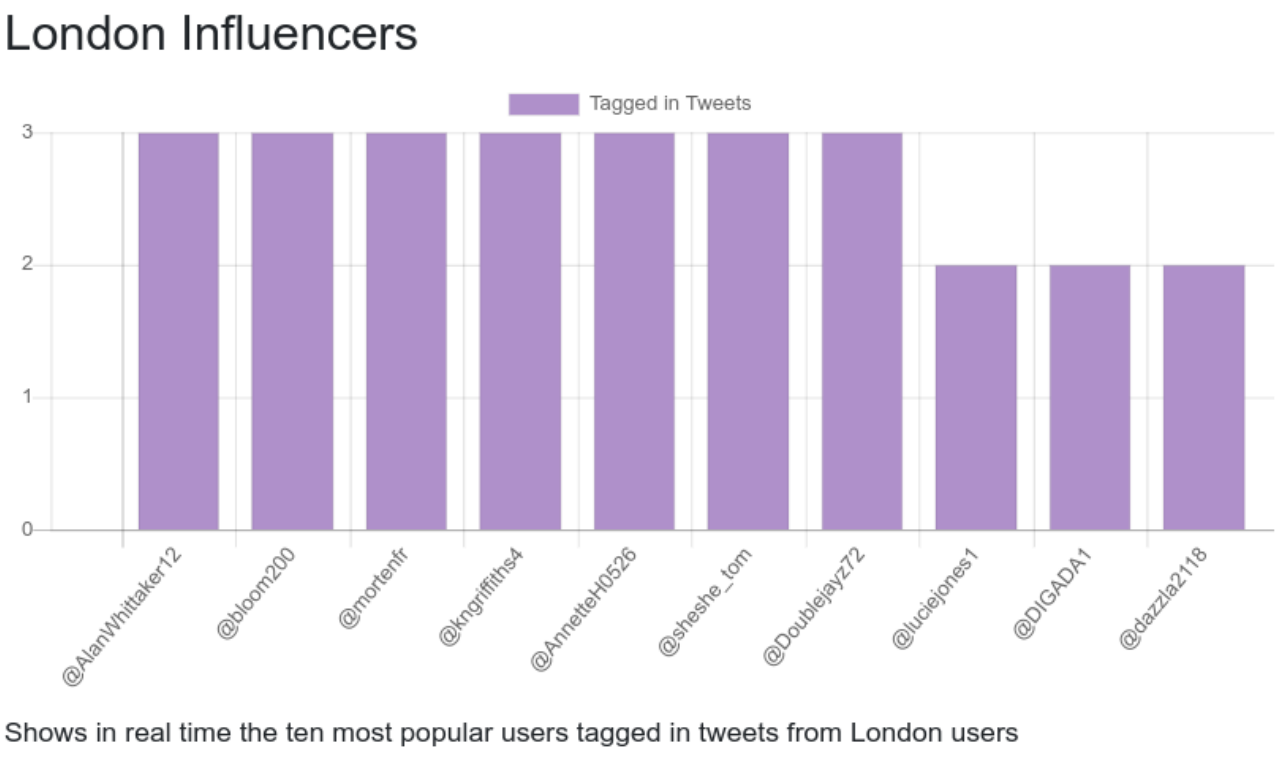
```
-------------------------------------------
Time: 2019-09-14 23:52:08
-------------------------------------------
('@AlanWhittaker12', 3)
('@bloom200', 3)
('@mortenfr', 3)
('@kngriffiths4', 3)
('@AnnetteH0526', 3)
('@sheshe_tom', 3)
('@Doublejayz72', 3)
('@luciejones1', 2)
('@DIGADA1', 2)
('@NickyStix55', 2)
```

Following all of the testing above, I can conclude the data processing and analysis is being performed correctly and data is being populated as specified and required.

4.3 Testing: Front End Visualisations

Visualisation: Influencers Chart

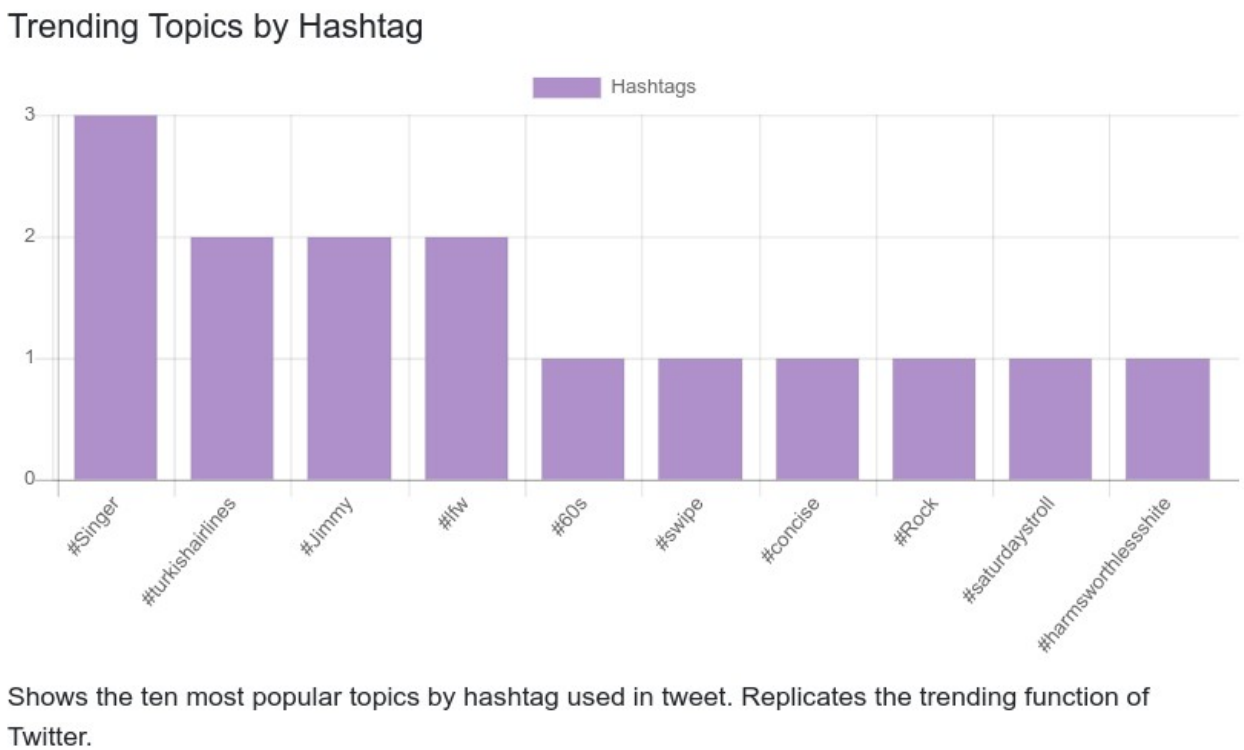
The bar-chart showing the top ten tagged ‘influential’ users should update every 2 seconds and contain data in line with the arrays stored in the ***refresh_influencers*** API endpoint:



Visualisation: Trending Chart

The bar-chart showing the top ten trending ‘hashtag’ topics should update every 2 seconds and contain data in line with the arrays stored in the ***refresh_trending*** API endpoint:


```
{
  "Count": [
    3,
    2,
    2,
    2,
    1,
    1,
    1,
    1,
    1,
    1
  ],
  "Label": [
    "#Singer",
    "#turkishairlines",
    "#Jimmy",
    "#lfw",
    "#60s",
    "#swipe",
    "#concise",
    "#Rock",
    "#saturdaystroll",
    "#harmsworthlessshite"
  ]
}
```

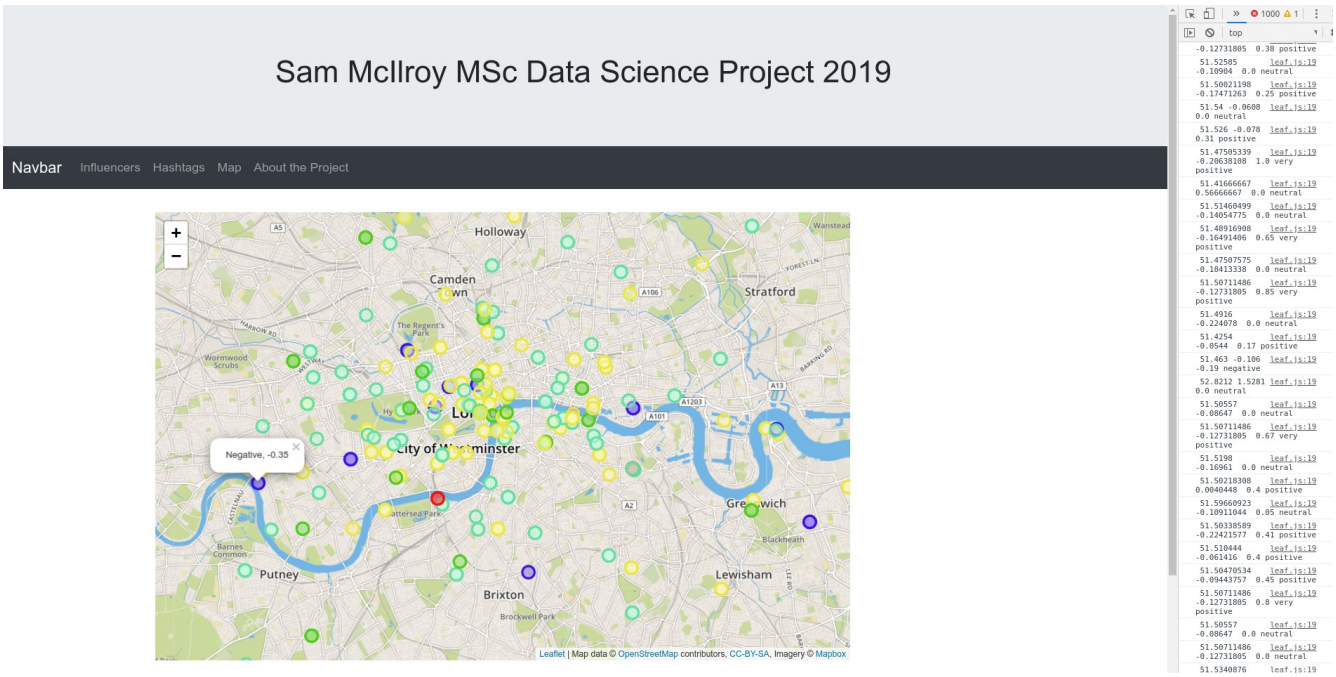



Shows the ten most popular topics by hashtag used in tweet. Replicates the trending function of Twitter.

Visualisation: London Tweet Sentiment Map

The London Sentiment Map should be populated in real time with location based tweets. Tweets should be correctly colour coded and tagged/titled with the correct sentiment/polarity.

I added a console log statement to the chart' JavaScript to print to the browser developer console the extracted elements (latitude, longitude, polarity, sentiment) so that tweets could be monitored as they are populated for correctness:



Time was spend monitoring the log and inspecting the markers until I was confident that the tweets were being populated and marked correctly.

4.4 Testing: Summary

Concluding from the testing steps performed at each step as detailed in this chapter I feel that, especially at the project/proof of concept stage, that the project is producing what is expected against the design specification and that results being displayed are accurate.

5. Results

This chapter analyses to what extent the results/outcomes of the project match the outcomes in the design specification.

5.1 Results: Real time data ingestion and storage

Specification	**Result**
Tweet data is collected as JSON representation of tweet object	Success – verified through testing and analysis of the JSON structure
Tweet data is restricted to London coordinates by use of a geographical boundary coordinate box	Success – verified through testing and manual mapping of coordinates
Location/coordinates data must be included for any map based visualisations	Success – verified through testing, map visualisations are populating as expected
Text data mutt be included, and formatted correctly, for sentiment analysis and hashtag/user analysis	Success – verified through testing, sentiment analysis successful, no errors in stream(s) due to incorrectly formatted/encoded data
Data should be held in short or long term storage as ‘streams’ to be processed by subsequent stages	Success – streams populated as Apache Kafka topics, data stored in topic/on disk
Data ‘streams’ should be held either in long term storage or in short term storage,for example stored in a message broker system, i.e. an appropriate Apache Kafka topic	Success – streams populated as Apache Kafka topics, data stored in topic/on disk

5.2 Results: Real time data processing and storage

Specification	**Result**
Data should be made available as ‘streams’ to the processing stage	Success – streams populated as Apache Kafka topics, data stored in topic/on disk
Output: dataframe(s): running total of the top ten hashtags used in all tweets processed so far	Success – verified through testing, dataframes populated, refreshed and pushed to Flask as required
Output: dataframe(s): running total of the top ten users tagged in all tweets produced so far	Success – verified through testing, dataframes populated, refreshed and pushed to Flask as required
Output: transformed data: each tweet should be processed to extract polarity and sentiment from the text, re-stored as new ‘stream’	Success – verified through testing, transformed data populating new Kafka topic as required

5.3 Results: Web based front-end

Specification	Result
Output: real-time bar-chart: top ten trending topics	Success – verified through testing, chart is populating and updating in line with pushed data
Output: real-time bar-chart: top ten tagged ‘influential’ users	Success – verified through testing, chart is populating and updating in line with pushed data
Output: sentiment map of London, should show coordinates/user location and attached sentiment/polarity value for each tweet as processed, as close to real-time as possible (allowing for processing time), should give an indication of sentiment levels and tweet volume across London	Success -verified through testing. The sentiment map is populating with tweets close to real time, immediately as the data is consumed from the Kafka topic. The overlapping markers over time give a good indication of sentiment (by colour) and volume (by density)
Appropriate organisation, navigation and visual elements to create an efficient user interface/access to the visualised data	Success – CSS and Bootstrap elements provide theming and navigation elements in line with requirements

5.4 Results: Summary

Through testing and analysis I have shown that the project meets the original design requirements and constitutes a successful proof of concept application of the analysis and visualisation of Twitter data trends and sentiment mapping.

6. Evaluation and Future Work

In this chapter I will self evaluate the work and lessons learned. As the project is a proof of concept approach, I will also speculate on some future work and expansion I would like to perform on the project, especially in terms of scalability, to use it as a tool to further my knowledge of data application development, software engineering and distributed technologies.

6.1 Evaluation: What went well?

Overall I am very happy with the results of the project. I learned a great deal about advancements in modern data software and distributed technologies. The project was an ideal opportunity to gain experience with Docker, Apache Kafka and Spark and working with streaming data. I also greatly increased my experience and skills in software development and Python programming, building upon the skills learned in the Principles of Programming I and Principles of Programming II modules and having the opportunity to put into practice skills learned on other modules such as Cloud Computing.

I was also pleased with the design and functionality of the front end web application and visualisations. The sentiment map in particular is an interesting application and will be useful to develop further to improve my skills. Web development and visualisation in general are skills I am weaker at so it was beneficial to have the opportunity to develop further in these areas.

Through the completion of this project I feel I have demonstrated many of the skills learned on the MSc program and have gained experience in end to end data software development. It has given me the chance to expand on my experience in data analytics and relational database development in my current work role and demonstrate ability to move into big data and software developer roles in the future.

6.2 Evaluation: Lessons Learned

- **Time Management and Planning:** the original scope of the project as outlined in the project proposal was very ambitious as I did not have experience in many of the technologies to be implemented. Although the scope was modified somewhat when beginning the project, for example by scaling back the sentiment analysis to use a Python library rather than a custom model, there was still a great deal of content to learn and implement in order to meet the aims of the project. I would often hit roadblocks or spend undue time on tutorials or documentation that was not productive.

- **Reacting to Issues:** due to the nature of the project being end to end, i.e. each stage depends somewhat on outcomes of previous stages, there were several times during development that I would be stuck on some aspect and felt that I could not continue until an issue was solved. This led to several days of no progress being made at all. I could in future look at investigating other stages of a project, perhaps by using synthesised data etc., and spend time more productively on simpler elements

- **Seeking Support:** at several stages during the project, especially during the beginning, being stuck on a certain aspect meant that progress was slow. I was reluctant to reach out for support to my supervisor or others feeling that it would not be appropriate to do so until I could demonstrate that sufficient progress had been made to warrant a conversation. This was very detrimental to my progress.

- **Write-up:** I had planned a strategy of completing the work in full before beginning the project write up. However, as I was not able to complete the project until near the deadline I had left the entire write up to complete in a short space of time. In future I will complete all write ups of a project at the end of each stage or sub-stage of development. This will both reduce last-minute workload and foster a feeling of progression.

6.3 Future Work

I found the workload and scope of this project to be difficult and as such I was focused on delivering a minimum viable product to support my idea of a proof of concept approach and to aid completion of the project on time. Therefore, there are still features and developments that could be made to the project to improve its functionality and usefulness as a data product, and to facilitate learning and demonstration of further big data software skills and experience.

- **Cloud Deployment:** I had hoped to test the project on larger volumes of data by deploying the Docker container to a cloud environment, distributed over a cluster through technology such as Kubernetes. I would then be able to demonstrate that the project is scalable to the full stream of Twitter data, if available, with the intended minimal changes to the codebase and structure. There was not sufficient time to implement this work in detail.

- **NoSQL Data Storage and SparkSQL:** To keep the project in scope, I focused data storage on disk storage in Kafka topics. This is suitable for the data I am working with but meant that the project only ‘answers’ the queries which I have designed it to. I could increase the usefulness of the project as a general API by implementing storage of the tweet data and analysed dataframes in a NoSQL database storage system over a period of time. Technologies such as SparkSQL or other distributed query tools could then be connected to the data to make ad-hoc and customised queries and requests on the data, for example to examine trends over time.

- **Hosting:** It would be interesting to investigate hosting the project as a public website where it would be more easily demonstrable to others. However, with the volume of data involved the bandwidth costs may be cost prohibitive/unfeasible.

Appendix A: Project Running Instructions for Examiners

- Visit https://docs.docker.com/install/ to install Docker on a Windows, Linux or OSX based machine
- Visit DockerHub at https://hub.docker.com/ to create a free account
- Open a **new** terminal window
- Login to DockerHub using the command: `docker login --username=your_username`

```
sam@pop-os:~$ docker login --username=sammcilroy
Password:
WARNING! Your password will be stored unencrypted in /home/sam/.docker/config.json.
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/#credentials-store

Login Succeeded
sam@pop-os:~$
```

- Pull the latest Docker image of the project using the command: `docker pull sammcilroy/london_twitter:latest`

```
sam@pop-os:~$ docker pull sammcilroy/london_twitter:latest
latest: Pulling from sammcilroy/london_twitter
7413c47ba209: Pull complete
0fe7e7cbb2e8: Pull complete
1d425c982345: Pull complete
344da5c95cec: Pull complete
1364639844f5: Pull complete
88e11624e8d9: Pull complete
71b8838a47ed: Pull complete
54b12eb8e055: Pull complete
ff6f0c9bf6af: Pull complete
57b0cbfec10a: Pull complete
c8bdea22689b: Pull complete
Digest: sha256:f57ea6023f0c333a11df23942666b3e50fcb1ee50edf0a913ff5c50fcab545a7
Status: Downloaded newer image for sammcilroy/london_twitter:latest
docker.io/sammcilroy/london_twitter:latest
sam@pop-os:~$
```

- Create a new container using the command: `docker run -it sammcilroy/london_twitter:latest`

```
sam@pop-os:~$ docker run -it sammcilroy/london_twitter:latest
root@271f6b7c889a:/#
```

- The container will automatically start, exit the container for now using the command : `exit`

```
root@271f6b7c889a:/# exit
exit
sam@pop-os:~$
```

- Docker will assign the container a random name, find this name by showing all containers using the command and referencing the NAMES column for the most recently CREATED container : `docker ps -a`

```
sam@pop-os:~$ docker ps -a
CONTAINER ID        IMAGE                              COMMAND             CREATED             STATUS                          PORTS               NAMES
271f6b7c889a        sammcilroy/london_twitter:latest   "/bin/bash"         2 minutes ago       Exited (0) About a minute ago                       pensive_kalam
sam@pop-os:~$
```

- Rename the container 'project' using the command : `docker rename NAME project`

```
sam@pop-os:~$ docker rename pensive_kalam project
sam@pop-os:~$
```

- Star the project container using the command : `docker start project`

```
sam@pop-os:~$ docker start project
project
```

- Open a **new** terminal window
- Open a bash prompt on the project container using the command: `docker exec -it project /bin/bash`

```
sam@pop-os:~$ docker exec -it project /bin/bash
root@271f6b7c889a:/#
```

- Start the Kafka consumer ***kafka_twitter_stream_json*** by running the command: `python msc_project_smcilr01/kafka/twitter_to_kafka.py`

```
root@271f6b7c889a:/# python msc_project_smcilr01/kafka/twitter_to_kafka.py
```

- Open a **new** terminal window
- Open a bash prompt on the project container using the command: `docker exec -it project /bin/bash`

```
sam@pop-os:~$ docker exec -it project /bin/bash
root@271f6b7c889a:/#
```

- Start the Kafka consumer ***kafka_twitter_stream_coordinates*** by running the command: `python msc_project_smcilr01/kafka/twitter_to_kafka_coordinates.py`

```
root@271f6b7c889a:/# python msc_project_smcilr01/kafka/twitter_to_kafka_coordinates.py
```

- Open a **new** terminal window
- Open a bash prompt on the project container using the command: `docker exec -it project /bin/bash`

```
sam@pop-os:~$ docker exec -it project /bin/bash
root@271f6b7c889a:/#
```

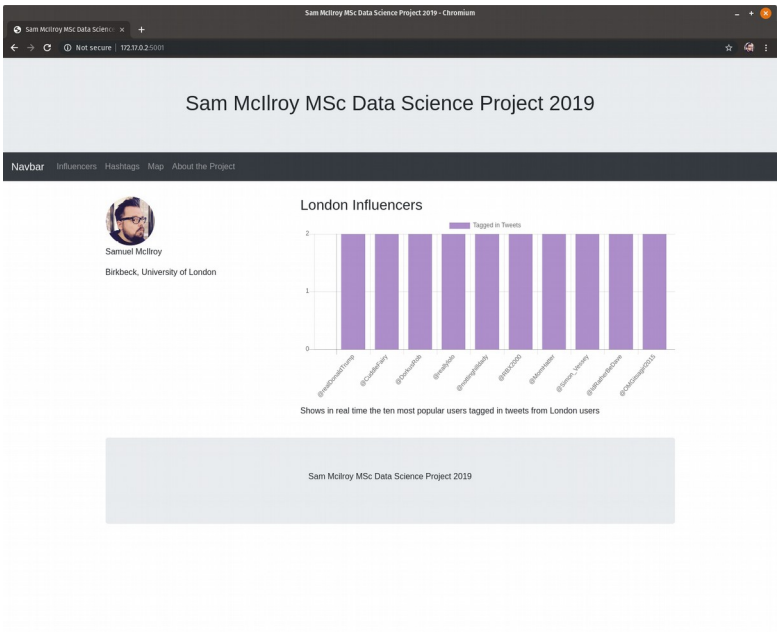
- Start the Flask application by running the command: `python msc_project_smcilr01/flask/app.py`

```
root@271f6b7c889a:/# python msc_project_smcilr01/flask/app.py
 * Serving Flask app "app" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: on
 * Running on http://0.0.0.0:5001/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 242-272-544
```

- Open a **new** terminal window
- Open a bash prompt on the project container using the command: `docker exec -it project /bin/bash`

```
sam@pop-os:~$ docker exec -it project /bin/bash
root@271f6b7c889a:/#
```

- Start the Spark application by running the command: `./spark-2.4.3-bin-hadoop2.7/bin/spark-submit msc_project_smcilr01/spark/spark_streaming.py`
    - You should see the data collecting in the terminal window
- Open a new web browser window, visit http://172.17.0.2:5001, you can now view the project front-end running against the Twitter data



References

[1] Fassbender, Melissa, More data will be generated in 2019 than in the last 5,000 years, https://www.outsourcing-pharma.com/Article/2018/12/11/More-data-will-be-generated-in-2019-than-in-the-last-5-000-years, 2018

[2] Marr, Bernard, How Much Data Do We Create Every Day?, https://www.forbes.com/sites/bernardmarr/2018/05/21/how-much-data-do-we-create-every-day-the-mind-blowing-stats-everyone-should-read 2018

[3] Kleppman, Martin, Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable and Maintainable Systems, O'Reilly Media, 2017

[4] Ballatore, Andrea, Birkbeck Bsc/MSc Research Themes, PS Mapping Place Sentiment in Social Media, https://docs.google.com/document/d/1YqGhZKp_8Fl_j2-PtYrVlaK-tiNelbgOj90Qt4huGmM/edit

[5] Docker, What is a Container?, https://www.docker.com/resources/what-container

[6] HackerNoon, What is Containerizaion?, https://hackernoon.com/what-is-containerization-83ae53a709a6

[7] Pulralsight, Developing Apache Spark Applications: Scala vs Python, https://www.pluralsight.com/blog/software-development/scala-vs-python

[8] Kafka-Python Python Library, https://pypi.org/project/kafka-python/

[9] The Pallets Projects, Flask, https://palletsprojects.com/p/flask/

[10] GitHub, Samuel McIlroy, https://github.com/sammcilroy

[11] DockerHub, Samuel McIlroy, https://hub.docker.com/u/sammcilroy

[12] Twitter Developers, Consuming Streaming Data, https://developer.twitter.com/en/docs/tutorials/consuming-streaming-data.html

[13] Twitter Developers, Tweet Objects, https://developer.twitter.com/en/docs/tweets/data-dictionary/overview/tweet-object.html

[14] Tweepy, https://www.tweepy.org/

[15] Apache Kafka Integration with Spark, https://www.tutorialspoint.com/apache_kafka/apache_kafka_integration_spark.htm

[16] Kreps, Jay, It's Okay To Store Data In Apache Kafka, https://www.confluent.io/blog/okay-store-data-apache-kafka, 2017

[17] Adnan, Siddiqi, Getting Started with Apache Kafka in Python, Towards Data Science, https://towardsdatascience.com/getting-started-with-apache-kafka-in-python-604b3250aa05, 2018

[18] Kafka-Python, https://github.com/dpkp/kafka-python

[19] Zhang, Dell, Cloud Computing Week 11 Machine Learning in the Cloud, Birkbeck University of London, http://www.dcs.bbk.ac.uk/~dell/teaching/cc, 2019

[20] Streaming Data Pipeline to Transform Store and Explore with Kafka Spark and Drill, https://dzone.com/articles/streaming-data-pipeline-to-transform-store-and-exp

[21] Apache Spark, Spark Python API Docs, https://spark.apache.org/docs/latest/api/python/index.html

[22] TextBlob Documentation, https://textblob.readthedocs.io/en/dev/

[23] Bootstrap, https://getbootstrap.com

[24] Chart.js, https://www.chartjs.org

[25] Leaflet.js, https://leafletjs.com

[26] Ubuntu Docker Official Images, https://hub.docker.com/_/ubuntu/

[27] Digital Ocean, How to Install Apache Kafka on Ubuntu 18.04, https://www.digitalocean.com/community/tutorials/how-to-install-apache-kafka-on-ubuntu-18-04

[28] Portilla, Jose, Installing Scala and Spark on Ubuntu, https://medium.com/@josemarcialportilla/installing-scala-and-spark-on-ubuntu-5665ee4b62b1

[29] Linuxize, How to Install Flask on Ubuntu 18.04, https://linuxize.com/post/how-to-install-flask-on-ubuntu-18-04/

[30] Sookochef, Kevin, Kafka in a Nutshell, https://sookocheff.com/post/kafka/kafka-in-a-nutshell/

[31] TextBlob, Sentiment Analysis Method, https://github.com/sloria/TextBlob/blob/90cc87ab0f9e25f37379079840ec43aba59af440/textblob/en/sentiments.py

[32]Suarez, David, Real Time Twitter Analysis Displaying Results, http://davidiscoding.com/real-time-twitter-analysis-4-displaying-the-data

[33] Suarez, David, Real Time Twitter Analysis on Spark, http://davidiscoding.com/real-time-twitter-analysis-3-tweet-analysis-on-spark

[34] FreeCodeCamp, How to Build a Web Application Using Flask, https://www.freecodecamp.org/news/how-to-build-a-web-application-using-flask-and-deploy-it-to-the-cloud-3551c985e492/

[35] Data Adventures, Kafka, Flask and Server-Sent Events, https://www.data-adventures.com/2016/06/22/iot-101-part-2-kafka-flask-and-server-sent-events.html

[36] Code Beautify JSON Viewer, https://codebeautify.org/jsonviewer

[37] CopyPasteMap, http://www.copypastemap.com/

[38] McIlroy, Samuel, Project Proposal: Twitter Sentiment Analysis Big Data Software Development, 2019 (Proposal Submitted and Available on Request)