Data Engineering Project

Data Pipeline Prototype

Submitted by

Yashawant Parab (11012130)

Shidharth Bammani (11011885)

Sreenath Gopi (11012060)

Sandeep RK (11011854)

Introduction:

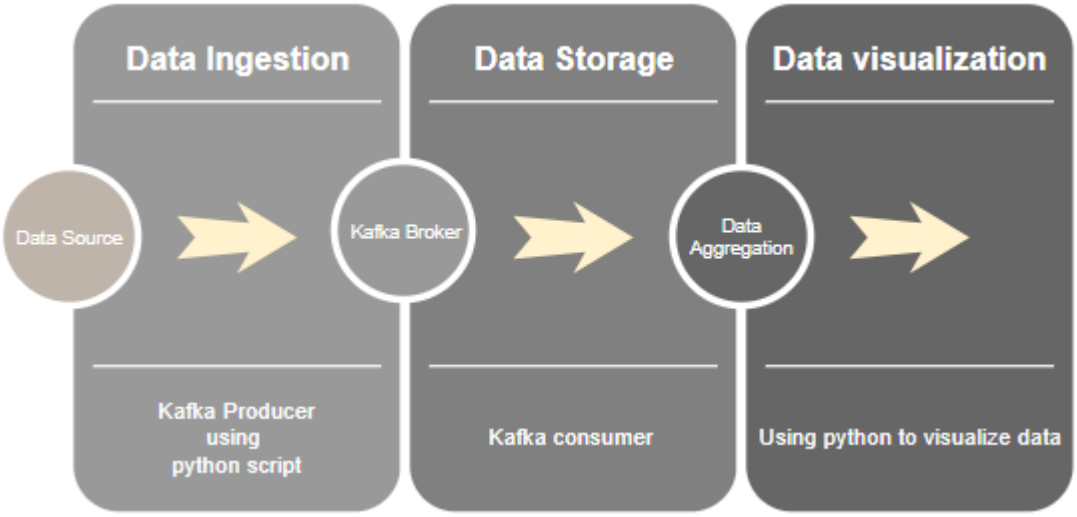
This report is intended to describe a prototype of a data pipeline using Apache Kafka and addresses the three main layers of data pipeline that are data ingestion, data storage, and data processing/visualization. We are using Apache Kafka as a platform for streaming of data across distributed systems. Furthermore, the steps involved in creating the layers are explained in detail along with the analysis of the data.

We have created two data pipeline application for two data sources which are Twitter and New York Cabs, both the data sources are different. So we are covering two classes of application i.e. real-time streaming data pipelines and the other one is streaming of batch data processing that transform to the streams of data.

Twitter as a data source, we intend to build a prototype which takes the real time tweets as streaming data. The stream data is accessed using Tweepy library which enables python to interact with the Twitter platform and use its API. The streamed data is directed to Kafka and then stored in mongodb from which analysis and visualisation is tried out.

New York Cabs is the data source which provides data on taxi trips in New York. This data source contains the data files in csv format which we download and stream the data in data pipeline, store it in MongoDB and further visualize the data by doing analysis.

Basic flow diagram of Data pipeline process.



I. Data pipeline for batch processing:

Below are the layers of data pipeline for batch processing of New York Cabs data source.

Data Source:

The New York Cabs has trip record data of the taxis in New York, the records from 2009 till now are present in the csv format. We are using data record of the Green taxi for the month of January ‘18 and December ’17. This is link for downloading the data is provided in the External source section. The trip data records include fields capturing pick-up and drop-off dates/times, pick-up and drop-off locations, trip distances, itemized fares, rate types, payment types, and driver-reported passenger counts.

Data Ingestion:

After the data is collected from the data source, the record files have to streamed in to the data pipeline. We are using Apache Kafka as tool to store the stream of records in a fault tolerant durable way and process the stream of data as they occur.

As we know Kafka has four APIs i.e. Producer API, Consumer API, Stream API, Connector API. We will be using the producer API to load the data in to the Kafka server. Kafka cluster stores streams of records in categories called topics.

Below mentioned are the basic steps to initialize Apache Kafka server.

Step 1: Start the zookeeper and the Kafka server.

Step 2: Create the topic and provide the localhost, partition details and run the producer.

Step 3: Run the command line for consumer to dump the messages sent by producer.

After starting up the Kafka server we will create the producer in python. The python script reads the csv file send them to Kafka broker. Then a consumer will read the data from the broker and store them in a MongoDb collection. Below are the snippets of code along with its short explanation for creating a Producer.

```
import csv
from json import dumps
from time import sleep
from kafka import KafkaProduce
```

- We will create a python script and import the above mentioned libraries.

```
csv.register_dialect('taxijan',delimiter = ',',skipinitialspace=True)
```

- csv.register - Associate dialect with name. name must be a string. The dialect can be specified either by passing a sub-class of Dialect, or, with keyword arguments overriding parameters of the dialect.
- The Dialect class is a container class relied on primarily for its attributes, which are used to define the parameters for a specific reader or writer instance.
- skipinitialspace - When True, whitespace immediately following the delimiter is ignored. The default is False.
- delimiter- A one-character string used to separate fields. It defaults to ','.

```
producer = KafkaProducer(bootstrap_servers=['localhost:9092'],value_serializer=lambda x: dumps(x).encode('utf-8'))
```

- bootstrap_servers=[‘localhost:9092’]: sets the host and port the producer should contact to bootstrap initial cluster metadata. It is not necessary to set this here, since the default is localhost:9092.
- value_serializer=lambda x: dumps(x).encode(‘utf-8’): function of how the data should be serialized before sending to the broker. Here, we convert the data to a json file and encode it to utf-8.

```
with open("Y:\SRH Heidelberg\Modules\DataEngg\Taxi Data\Jan\green_Jan.csv", 'r') as csvfile:
    reader = csv.DictReader(csvfile, dialect = 'taxijan')
    for row in reader:
        data = {'NYCData' : row}
        producer.send('NYCDatabase7', value=data)
```

- with open() – opens the csv file and reads it, so we provide the path of the directory where the csv file is stored.
- we create a variable as 'reader' to store the records of the csv file.
- now we run 'for' loop to read each rows of the csv file and send the values of the row to the producer.
- calling the send method on the producer and specifying the topic and the data.

```
csvfile.close()
```

- calling the close() method to close the file.

Difficulties faced for data ingestion:

- The data file we had was in CSV format and the Kafka producer sends the stream of data in json format, so the records present in the csv file had to serialized into json format, but we were receiving the error while running the producer python script. The error being displayed was that 'json is not serializable '. So we tried to first convert the csv file into json by writing code in python and then feeding the json data file into producer, but even that didn't work since the data was too large. We made few modifications in the code and the producer script worked just fine, the data records were sent to the producer API.

Data Storage:

We have created the producer that reads the csv data and send them to our Kafka broker. Now we will create a consumer by writing python script and the consumer will read the data from the broker and store them in a MongoDb collection. We are storing the data in NOSQL database

Below are the snippets of code along with its short explanation for creating a Consumer.

```
from kafka import KafkaConsumer
from pymongo import MongoClient
from json import loads
```

```
consumer = KafkaConsumer(
    'TaxiJan',
     bootstrap_servers=['localhost:9092'],
     auto_offset_reset='earliest',
     enable_auto_commit=True,
     group_id='my-group',
     value_deserializer=lambda x: loads(x.decode('utf-8')))
```

- bootstrap_servers=[‘localhost:9092’]: same as our producer
- auto_offset_reset=’earliest’: It handles where the consumer restarts reading after breaking down or being turned off. We have set to earliest, so the consumer starts reading at the latest committed offset.
- enable_auto_commit=True: this makes sure the consumer commits its read offset every interval.
- group_id=’my-groups’: this is the consumer group to which the consumer belongs.
- The value deserializer deserializes the data into a common json format, the inverse of what our value serializer was doing in producer script.

```
client = MongoClient('localhost:27017')
collection = client.Taxi.TaxiJan
```

- The code above connects to the TaxiJan collection of our MongoDb database.

```
for message in consumer:
    message = message.value
    collection.insert_one(message)
    print('{} added to {}'.format(message, collection))
```

- We will extract the data from our consumer by looping through it. The consumer will keep listening until the broker doesn’t respond anymore. A value of a message can be accessed with the value attribute. Here, we overwrite the message with the message value. The next line inserts the data into our database collection. The last line prints a confirmation that the message was added to our collection.

  Difficulties faced for data ingestion: We faced many challenges while implementing the consumer API, the code had to modified many times. One such instance is we were receiving mongo client error and sometimes the code run but the collection was not getting created in the database while using Pycharm runtime environment for python. We run the same code in Jupyter and the consumer code worked and collection was created in the database.

Data Visualization:

The data records are being collected in the mongodb database. Now we can further proceed with the analysis of the data. For further processing the data we need to aggregate the data records. For data aggregation we used aggregation queries in the python script to collect the data from database and group values from documents together, and performed a variety of operations on the grouped data to analyse and form simple visualization by plotting graph of the obtained data.

Below are the snippets of code along with its short explanation for data aggregation and plotting the graph.

```
client = MongoClient('localhost:27017')
collection = client.NYC4.NYCTaxiJan4
```

- The code above connects to the NYC4 collection of our MongoDb database.

```
TaxiData = collection.aggregate(
    [
        {'$unwind': '$NYCData.fare_amount'},
        {'$group': {'_id': '$NYCData.fare_amount.text', 'FareAmountCount' : {'$sum' : 1}}},
        {'$sort': {'FareAmountCount' : -1}}, {'$limit' : 10},
        {'$project' : {'NYCData' : '$_id', 'FareAmountCount' : 1, '_id' : 0 }}
    ])

df = pd.DataFrame.from_records(TaxiData)
print(df)
```

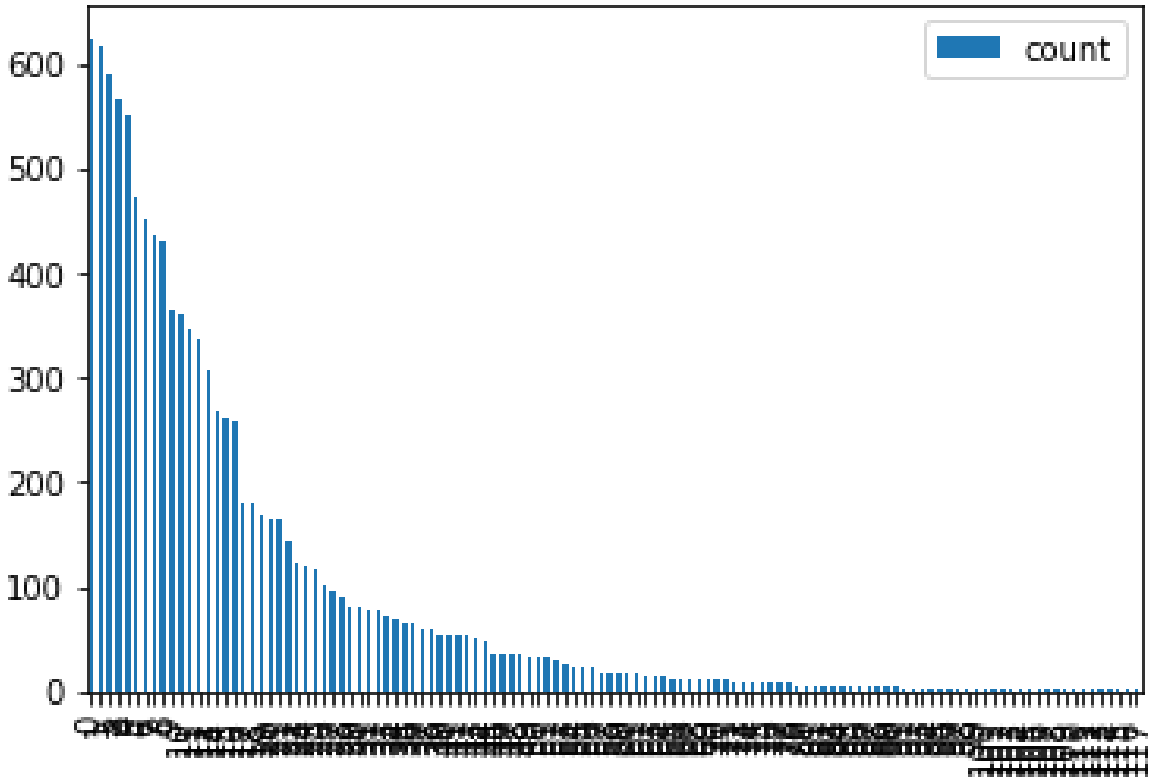
- We have passed the data records in the producer in an array named as “NYCData”, here we will use unwind that will deconstruct an array field from the input documents to output a document for each element.

```
fare_count = collection.aggregate([{'$group' : {'_id' : '$NYCData.fare_amount', 'count' : {'$sum' : 1}}},
                                   {'$match' : {'count' : {'$gt' : 1}}},
                                   {'$sort' : {'count' : -1}},
                                   {'$project' : {'fare_amount' : '$_id', 'count' : 1, '_id' : 0}}])
FareCountFrame = pd.DataFrame.from_records(fare_count)
print(FareCountFrame)
```

```
var = FareCountFrame.plot.bar()
print(var)
x = ('count'),
y = ('fare_amount'),
rot = 0  #rotation
```

- The above code plots the graph of the selection of the coumn made in the aggregation query.

Below graph displays the result of aggregation and visualization of the data by answering the question as the highest earnings generated daily for the month of January based on entire.



II. Data Pipeline steps for Stream Processing:

Data Source:

Twitter API is being used as the data source. For accessing the data from API, python Library Tweepy is used.

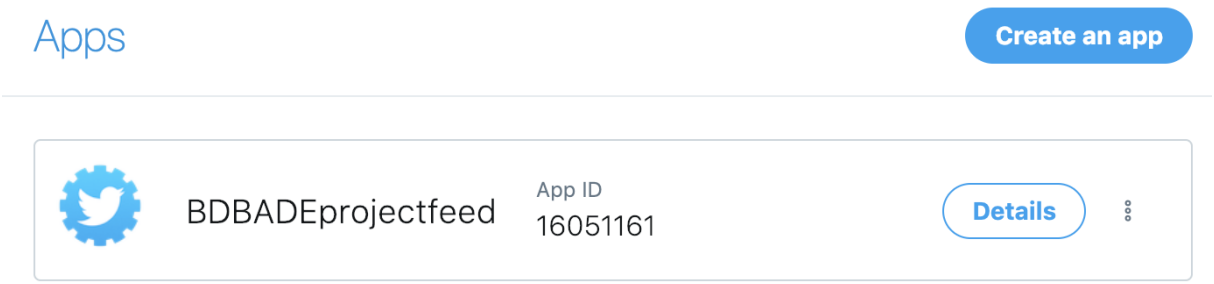
Tweepy API is a python library through which tweets are streamed directly from Twitter in real time.

Tweepy was installed via terminal:

pip install tweepy: # to install tweepy module

Firstly, a twitter app needs to be created to interface with python to stream the tweets using keywords or hashtag.

In order to create the app a twitter account is required. In this case my personal twitter account was used.



Once the app is created four essential keys were retrieved from the app details.

Keys required to authenticate and access Twitter API:

```
access_token = "102939965-x3DfNuPCjgUrvhF6S8mrZnIJRGtl"
access_token_secret = "vcxuGfhB7GwXqL3ETnO3xOmFpPrypEgK"
consumer_key = "06LOaskAI66Pmtpf09ad"
consumer_secret = "SEBg3rtAHwwNnFL54KZImfS9YNKhSxFxnT9"
```

These python strings can be either stored in a different file (same directory as other py files) and imported in the program or it can be directly defined in the other programs.

Data Ingestion:

Data ingestion i.e., process of obtaining and importing data for immediate use or storage in a database.

In this project Apache Kafka is used for data ingestion purpose.

Apache Kafka was installed using homebrew which also installs the dependency zookeeper.

In order to absorb the data from Twitter using the Twitter API both zookeeper and kafka services should be started.

To start zookeeper service:

$ zookeeper-server-start /usr/local/etc/kafka/zookeeper.properties

To start kafka:

$ kafka-server-start /usr/local/etc/kafka/server.properties

Both these services are required to be running in parallel in the local machine.

Once started, a topic is created to send the data from Twitter API to Kafka.A command line producer in python is run which will take input from a file or streaming data and send it out to Kafka cluster.

```
from kafka import SimpleProducer, KafkaClient

# To send messages synchronously
kafka = KafkaClient('localhost:9092')
producer = SimpleProducer(kafka)
```

And the data is sent to the topic as a Unicode message using:

```
producer.send_messages("TwitterTopic1", data.encode('utf-8'))
```

Python Script1 also includes the Tweepy classes such as StreamListener, OAuthHandler, Stream which are imported from Tweepy Library.

```
from tweepy.streaming import StreamListener
from tweepy import OAuthHandler
from tweepy import Stream
```

StreamListener allow to listen to the tweets from the tweepy module

OAuthHandler is responsible for authentication based on the consumer and auth keys/secret.

Stream class allows us to access the streamed data.

An object is created which inherits from the StreamListener class

listener = StdOutListener()

Also, objects are created for authentication which will inherit from the OAuthHandler class in tweepy

auth = OAuthHandler(consumer_key, consumer_secret)

auth.set_access_token(access_token, access_token_secret)

A twitter stream is created which inherits from Tweepy Stream class.

filter() is a method provided by Stream class to filter the tweets based on keywords or hashtags.

stream = Stream(auth, listener)

stream.filter(track=["Manchester United", "Solskjaer"])

On running the script, the tweets are displayed on the screen in JSON format.

DATA STORAGE:

Data stream send using producer command line is absorbed using the consumer command line which directs the data into mongodb.

Consumer script includes importing of following libraries:

```
from kafka import KafkaConsumer
from pymongo import MongoClient
from json import loads
```

KafkaConsumer and MongoClient libraries re imported from kafka and pymongo packages respectively.

consumer variable is inheriting data from TwitterTopic1 which contains the stream data from Twitter .

```
consumer = KafkaConsumer(
    'TwitterTopic1',         # topic name in kafka
     bootstrap_servers=['localhost:9092'],
     auto_offset_reset='earliest',
     enable_auto_commit=True,
#    auto_commit_interval=50,
#    group_id='my-group',
     consumer_timeout_ms=5000,
     value_deserializer=lambda x: loads(x.decode('utf-8')))
```

Json messages in the topic is consumed and deserialized using the code :

value_deserializer=lambda x: loads(x.decode('utf-8')))

There can be more than one consumer running parallel which can be used to absorb data from a producer .

This consumed data is stored in mongodb as a collection:

```
client = MongoClient('localhost:27017')
collection = client.twitter1.tweet_data


for message in consumer:
    message = message.value
    collection.insert_one(message)
    print('{} added to {}'.format(message, collection))
```

collection= client.twitter1.tweet_data

This will store the data in collection tweet_data in twitter1 db.

Collection.insert_one(message)

Document is inserted into the collection using the above code and displayed in terminal using the print function.

The data will be displayed on the basis of *consumer_timeout_ms* variable value defined in the script.

Data Analysis and Visualization:

Sentiment analysis:

For sentiment analysis python library named TextBlob is made use of.

TextBlob is a python library used for processing textual data. It provides a simple API for diving into common natural language processing (NLP) tasks such as sentiment analysis.

```
import tweepy
from textblob import TextBlob
```

For sentiment analysis TexBlob class is imported from the TextBlob library,also tweepy library is imported for API authentication and streaming data in python.

```
api = tweepy.API(auth)

public_tweets = api.search('manchester united')
```

api variable is inheriting the tweepy.API(auth) class for authentication purpose.

public_tweets = api.search('Text/hashtags') inherits the streaming data based on the text or hashtag.

```
for tweet in public_tweets:
    print(tweet.text)
    analysis = TextBlob(tweet.text)
    print(analysis.sentiment)
```

Above code takes in the streamed twitter data/text and using the sentiment function provides the sentimental analysis output.

In this case the text ‘Manchester united’ is searched and tweets containing the same text is produced and using textblob each of the tweet’s sentiment analysis is generated.

Basically, two factors are taken into account which is polarity and subjectivity.

Polarity deals with the emotional aspect of the sentence or tweet whereas subjectivity deals with the personal feelings, beliefs or views which may not contain the emotional aspect.

On running the script, the tweet based on search is displayed and its polarity and subjectivity quotient.

Polarity and subjectivity is measured on a scale from -1 to 1 in which any value less than 0 would be having a negative sentiment, value greater than zero would be a positive sentiment and 0 is considered as a neutral sentiment.

```
RT @utdxtra: How lucky were we to watch Zlatan Ibrahimovic play for Manchester United? Iconic moment. Swedish hero! 🇸🇪 #mufc
Sentiment(polarity=0.47916666666666663, subjectivity=0.6666666666666667)
```

Above tweets include words like lucky, iconic hero which clearly indicate that it’s a positive tweet.

As we can see the polarity and subjectivity quotient is way above zero.

Whereas in the below tweet the sentiment quotients for polarity and subjectivity is zero from which we can understand that the tweet is neutral.

```
RT @Devils_Latest: ON THIS DAY, In 2006: Manchester United signed Patrice Evra from Monaco. #MUFC https://t.co/Mfliz1PZC9
Sentiment(polarity=0.0, subjectivity=0.0)
```

Data Visualization using cursor pointer:

For returning data from mongodb in the console the function

Db.collection.find(query, projection) is used. The parameter *query* is optional, or it can be used to specify selection filter using query operators. To return all documents in the collection this parameter can be excluded or an empty document can be passed.

The projection parameter determines which fields to be returned in the documents that match the query filter. For returning all fields in the document this parameter can be omitted. It takes the following form

> { field1: <value>, field2: <value> ... }

The <value> can be any of the following:

1 or true to include the field in the return documents.

0 or false to exclude the field.

As used in the script below the output will be as follows:

cursor = collection.find({}, {"_id" :1, "user.screen_name" :1, "user.favourites_count" :1}).sort([("user.favourites_count", -1)]).limit(5)

for data in cursor:

print(data);

```
Sreenaths-MacBook-Pro:deproject sree$ python visualtest.py
{u'_id': ObjectId('5c3541188f546814b3e179b7'), u'user': {u'favourites_count': 518398, u'screen_name': u'onahunttoday'}}
{u'_id': ObjectId('5c374ae68f5468274978ff83'), u'user': {u'favourites_count': 518398, u'screen_name': u'onahunttoday'}}
{u'_id': ObjectId('5c3541188f546814b3e17976'), u'user': {u'favourites_count': 518397, u'screen_name': u'onahunttoday'}}
{u'_id': ObjectId('5c374ae68f5468274978ff42'), u'user': {u'favourites_count': 518397, u'screen_name': u'onahunttoday'}}
{u'_id': ObjectId('5c35411a8f546814b3e17a77'), u'user': {u'favourites_count': 489968, u'screen_name': u'SnowBiAuthor'}}
```

Issue 1 :matplotlib installation error

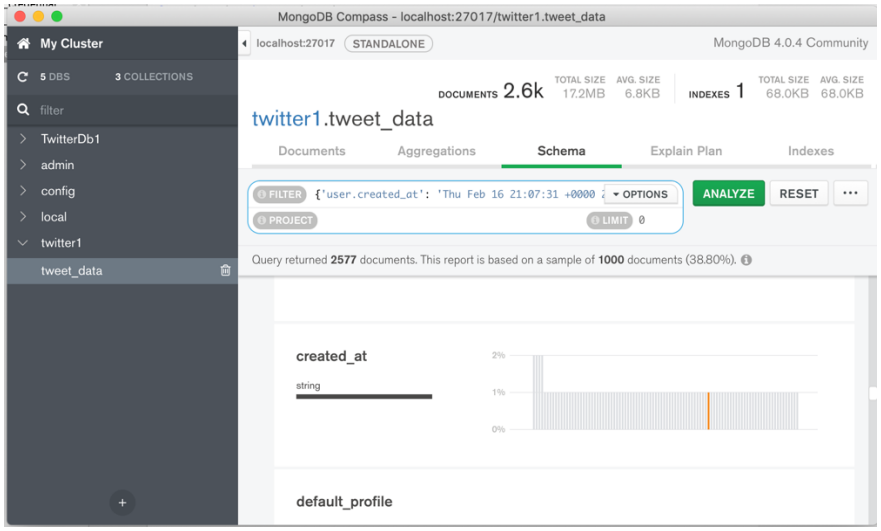
```
Sreenaths-MacBook-Pro:desktop sree$ brew install matplotlib
Error: No available formula with the name "matplotlib"
==> Searching for a previously deleted formula (in the last month)...
Warning: homebrew/core is shallow clone. To get complete history run:
  git -C "$(brew --repo homebrew/core)" fetch --unshallow

Error: No previously deleted formula found.
==> Searching for similarly named formulae...
Error: No similarly named formulae found.
==> Searching taps...
==> Searching taps on GitHub...
Error: No formulae found in taps.
Sreenaths-MacBook-Pro:desktop sree$ pip install matplotlib
Requirement already satisfied: matplotlib in /usr/local/lib/python2.7/site-packages (2.2.3)
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python2.7/site-packages (from matplotlib) (0.10.0)
Requirement already satisfied: subprocess32 in /usr/local/lib/python2.7/site-packages (from matplotlib) (3.5.3)
Requirement already satisfied: python-dateutil>=2.1 in /usr/local/lib/python2.7/site-packages (from matplotlib) (2.7.5)
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python2.7/site-packages (from matplotlib) (1.0.1)
Requirement already satisfied: numpy>=1.7.1 in /usr/local/lib/python2.7/site-packages (from matplotlib) (1.15.4)
Requirement already satisfied: pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=2.0.1 in /usr/local/lib/python2.7/site-packages (from matplotlib) (2.3.0)
Requirement already satisfied: pytz in /usr/local/lib/python2.7/site-packages (from matplotlib) (2018.7)
Requirement already satisfied: backports.functools-lru-cache in /usr/local/lib/python2.7/site-packages (from matplotlib) (1.5)
Requirement already satisfied: six>=1.10 in /usr/local/lib/python2.7/site-packages (from matplotlib) (1.12.0)
Requirement already satisfied: setuptools in /usr/local/lib/python2.7/site-packages (from kiwisolver>=1.0.1->matplotlib) (40.4.3)
Sreenaths-MacBook-Pro:desktop sree$ pip info matplotlib
ERROR: unknown command "info"
```

For mostly all package installations homebrew for mac is used.

For matplotlib pip installer had to be used.

MongoDB compass was made use for visualization, and aggregation. It analyzes the documents and displays rich structures within a collection through an intuitive GUI.

It allows you to quickly visualize and explore your schema to understand the frequency, types and ranges of fields in the data set.



Authorship of the Sections:

New York Cab (Data Ingestion, Data Storage, Data visualization)	Page number: 1 to 8	Shidharth Bammani (11011885), Yashawant Parab (11012130)
Twitter Data Source	Page number: 9 to 16	Sreenath Gopi (11012060)

External sites:

1. New York Cab batch processing data:

Data source:

http://www.nyc.gov/html/tlc/html/site_map/site_map.shtml

Data Ingestion and Storage:

https://kafka.apache.org/documentation/

https://dzone.com/articles/kafka-architecture

https://towardsdatascience.com/kafka-python-explained-in-10-lines-of-code-800e3e07dad1

Data Aggregation:

https://www.tutorialspoint.com/python/python_data_aggregation.html

http://api.mongodb.com/python/current/examples/aggregation.html

https://docs.mongodb.com/manual/reference/operator/aggregation/group/

https://www.youtube.com/watch?v=Kk6Er0c7srU

Data Visualization:

https://www.kaggle.com/hanriver0618/nyc-taxi-data-exploration-visualization

https://arxiv.org/pdf/1709.06176.pdf

https://docs.mongodb.com/manual/reference/operator/aggregation/unwind/

2. Twitter real time processing data:

Data Source

Created app via https://apps.twitter.com/

Data Ingestion:

https://kafka.apache.org/quickstart

https://medium.com/@Ankitthakur/apache-kafka-installation-on-mac-using-homebrew-a367cdefd273

http://docs.tweepy.org/en/v3.4.0/streaming_how_to.html

https://kafka-python.readthedocs.io/en/master/usage.html

Data Storage:

http://api.mongodb.com/python/current/tutorial.html

https://kafka-python.readthedocs.io/en/master/usage.html

https://towardsdatascience.com/kafka-python-explained-in-10-lines-of-code-800e3e07dad1

https://docs.mongodb.com/manual/reference/method/db.collection.insertOne/

Data Visualization:

https://textblob.readthedocs.io/en/dev/

https://docs.mongodb.com/manual/reference/method/db.collection.find/

https://matplotlib.org/faq/installing_faq.html#osx-notes

https://docs.python.org/2/library/sys.html