



Twitter Visualizer

Data Engineering Bootcamp Project Report

Done by: Fatima W Khadra

Instructor: Kassem Shehady

Advisor: Dr Imad Moukadem

2022-2023

Table of Contents

Project Description:	3
Project Stages:.....	3
Project Folder:.....	3
Data Collection:	4
Data Saving:.....	5
Data Analysis:	7
Understanding the data:	8
Query 1: Total Count	8
Query 2: Number of Users	8
Query 3: Latest Date	10
Query 4: Languages	11
Query 5: Countries	12
Query 6: Top Hashtags	13
Query 7: Sentimental Analysis	14
Query 8: Verified Users	16
Query 9: Top Liked.....	17
Query 10: Top Shared.....	17
Data Visualizer:	18
Building charts:.....	19
Final Result:	20
Recommendations:.....	20

Project Description:

This project aims building a web application that visualizes a set of data fetched from a social media platform -Twitter in this case-.

The project is fully committed on a Github repository: <https://github.com/12-fwkhadra/Visualizer.git>

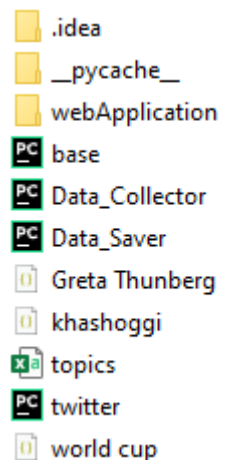
Project Stages:

1. Data Collection: in this stage, data is collected from the Twitter Platform. Python is used.
2. Data Saving: data collected is uploaded and saved in a database. Python and MongoDB are used.
3. Data Analysis: business questions are answered by querying the data collected. Python and MongoDB are used.
4. Data Visualization: results are visualized on a web application using Flask framework.

Project Folder:

The project consists of:

1. Topics csv: has the topics to be fetched with their number
2. Khashoggi, Greta Thunberg, world cup json: has the fetched tweets
3. Data_collector python: has the code that collects data
4. Data_saver python: has the code that saves the data
5. Base python: has the necessary code for running the scraper
6. Twitter python: has the twitter module responsible for scraping
7. webApplication: has all web app content
 - a. App python: includes all functions flask must run
 - b. Templates folder: includes all html pages
 - c. Static folder: includes all css files
 - d. Positive csv: has the percentages of the sentiments
 - e. Data csv: has the sentimental results for each tweet



Data Collection:

In order to collect the tweets from Twitter, Snscape library is used. This library is downloaded from the Github repository: <https://github.com/JustAnotherArchivist/snscape> and saved in the project's folder. All modules are deleted except the Twitter module and the base for its proper functioning. Python is used to scrape tweets because the Twitter Snscape module is python based.

To collect data 2 functions are created in a Data_Collector python file:

1. The function path_finder() that returns the path of the csv file- using the imported os library- that has the list of topics aiming to collect tweets about. The user shouldn't access the code directly to add his topic. The user must only access the csv file to add his hashtags to be scraped.

```
import twitter as sntwitter
import pandas as pd
import time
import json
import os

def path_finder():
    """
    This function returns the full path of a csv file
    The csv file stores the topics to be fetched later
    add the topics to the csv file
    don't apply changes to this function
    """
    csv_path = os.path.abspath(r'topics.csv')
    return csv_path
```

2. By importing Twitter.py and by using its methods, a function called fetcher is built. This function is responsible for scraping tweets from Twitter. For this project, tweets are fetched by hashtag thus used the TwitterHashtagScraper method. Also, for analysis purposes, tweets are fetched more concisely and specifically in a certain period of time identified by since and until abiding by the module's functions. The function fetcher() reads this csv file

and for each topic and number of tweets to be fetched, fetches the identified number of tweets using the `TwitterHashtagScraper` of the `Twitter sncrape` module imported initially. The `.get_items()` allows getting the tweet items. Each tweet -with all its fields- is then converted to a json string and appended to a predefined list resulting with a list holding all the scraped tweets. Then, a new file is created and opened with the name of the topic being fetched about and the tweets are written to it respectively. This step is essential to ensure saving the tweets in json file not jsonl to avoid later errors. Notice the `time.sleep()` method at the end of the function whose value is set to 1 second. This ensures no huge successive scrapings which may cause traffic on Twitter's API thus authentication problems.

```
def fetcher():
    """
    this function gets the data for the targeted hashtags using the TwitterHashtagScraper method of sncrape library
    add your hashtag title to the csv file
    only change the since and until dates which can be omitted too and number of tweets to be collected
    the results will be saved in json files in the same directory
    """
    csv_path = path_finder()
    topics = pd.read_csv(fr'%s' % csv_path, names=['Topic', 'number'])
    for topic, nb in zip(topics['Topic'], topics['number']):
        tweets = []
        for i, tweet in enumerate(
            sntwitter.TwitterHashtagScraper(f'{topic} since:2018-06-01 until:2019-11-30').get_items()):
            if i > nb: # nb of tweets to be collected
                break
            tweets.append(tweet.json())

        with open(f'{topic}.json', 'w') as file:
            for item in tweets:
                file.write(item + '\n')

        time.sleep(1)
```

To make sure that the fetching of the data worked properly, check in the project's folder for a json file holding the same topic's name.

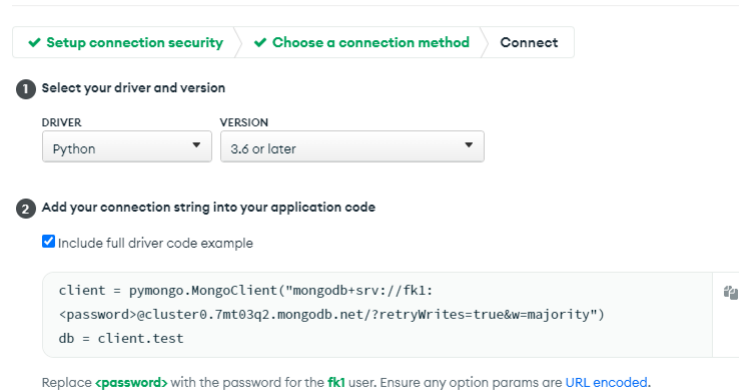
```
khashoggijson X
F: > Al Maaref Uni > Second Year 2021-2022 > Summer 21-22 > Practical Training > Final Project > flaskProject1 > khashoggijson
1 [{"_type": "twitter.Tweet", "url": "https://twitter.com/OldSchoolSciFi/status/1200564603136225281", "date": "2019-11-29T12:00:00.000Z", "text": "The new sci-fi movie is out now! #OldSchoolSciFi"}]
2 [{"_type": "twitter.Tweet", "url": "https://twitter.com/MCOSDU/status/1200544989056557056", "date": "2019-11-29T12:00:00.000Z", "text": "The new sci-fi movie is out now! #OldSchoolSciFi"}]
3 [{"_type": "twitter.Tweet", "url": "https://twitter.com/PaulDereume/status/1200543078299209729", "date": "2019-11-29T12:00:00.000Z", "text": "The new sci-fi movie is out now! #OldSchoolSciFi"}]
4 [{"_type": "twitter.Tweet", "url": "https://twitter.com/billyshakes1561/status/1200526787651891200", "date": "2019-11-29T12:00:00.000Z", "text": "The new sci-fi movie is out now! #OldSchoolSciFi"}]
5 [{"_type": "twitter.Tweet", "url": "https://twitter.com/wash_gton/status/1200511084999757824", "date": "2019-11-29T12:00:00.000Z", "text": "The new sci-fi movie is out now! #OldSchoolSciFi"}]
6 [{"_type": "twitter.Tweet", "url": "https://twitter.com/PolliMo4/status/1200493948051042304", "date": "2019-11-29T12:00:00.000Z", "text": "The new sci-fi movie is out now! #OldSchoolSciFi"}]
```

Data Saving:

A new python file is created called `data_saver` responsible for saving the fetched data in the database. This project chose the no-sql mongodb to interact with the data.

To save the data, a cluster holding the name cluster0 is created on mongoDB Atlas. A connection link to this cluster is generated and copied.

Connect to Cluster0



1 Select your driver and version

DRIVER: Python | VERSION: 3.6 or later

2 Add your connection string into your application code

☒ Include full driver code example

```
client = pymongo.MongoClient("mongodb+srv://fk1:
<password>@cluster0.7mt03q2.mongodb.net/?retryWrites=true&w=majority")
db = client.test
```

Replace **<password>** with the password for the **fk1** user. Ensure any option params are [URL encoded](#).

A function is created called `connecting_to_mongo()`. This function ensures connecting to the cluster created. In this function, the connection string variable is set equal to the connection link copied. Then a new client is declared for this connection using the `MongoClient` method. To use this method, the `pymongo` library must be imported initially. This library contains tools for working with MongoDB. Then, via this client a database is created. This returned database is accessed via `dbname` variable.

```
from pymongo import MongoClient
import pymongo
import json
import pandas as pd
from data_collector import path_finder

def connecting_to_mongo():
    # connecting python to mongodb
    CONNECTION_STRING = "mongodb+srv://fk1:mongodb22@cluster0.7mt03q2.mongodb.net/?retryWrites=true&w=majority"

    # set a client for the connection
    client = MongoClient(CONNECTION_STRING)

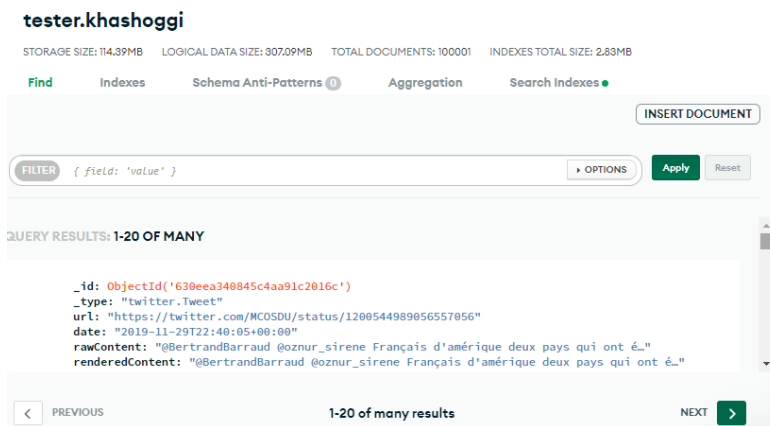
    # Create the database, tester is a sample name
    return client['tester']

# access the database
dbname = connecting_to_mongo()
```

In the function `save_to_mongo()`, the topics csv is read again to name the collections to be created in the database `dbname` by the same topic name. The json file saving the fetched data is opened, and each tweet is converted to a python dictionary using `json.loads()`. The resulted dictionary is then appended to a declared list called `requesting` as a pymongo request `InsertOne()`. This request is executed in the upcoming step. The `requesting` list is finally pushed into the collection by executing the previous set of requests using `bulk_write()`.

```
def save_to_mongo():
    csv_path = path_finder()
    topics = pd.read_csv(fr'%s' % csv_path, names=['Topic'])
    for topic in topics['Topic']:
        collection = dbname[topic]
        requesting = []
        with open(fr'%s.json' % topic) as file:
            for jsonobj in file:
                myDict = json.loads(jsonobj)
                requesting.append(pymongo.InsertOne(myDict))
        result = collection.bulk_write(requesting)
```

By this, the fetched data is saved to the database for later usage.



Data Analysis:

Mongo is chosen to query the data for further and in-depth analysis. 9 queries are built using mongo and another 1 query is hard coded in Python. In this project, queries are integrated with a python code. That's why queries are modified to pymongo. Each query is put in a separate function that returns the result as a json for further purposes -to be discussed in the next section-.

Understanding the data:

Before querying the data it is good to study it out and have a general overview on what we have. Each fetched tweet has the following fields that can be accessed and used.

```
fields 183
> { _id ObjectId
> { _type String
> { card String
> { cashtags String
> { conversationId Int64
> { coordinates String
> { date String
> { hashtags Array
> { id Int64
> { inReplyToTweetId Int64
> { inReplyToUser Object
> { lang String
```

```
> { likeCount Int32
> { links String
> { media list
> { mentionedUsers list
> { place String
> { quoteCount Int32
> { quotedTweet Object
> { rawContent String
> { renderedContent String
> { replyCount Int32
> { retweetCount Int32
> { retweetedTweet String
> { source String
> { sourceLabel String
> { sourceUrl String
> { url String
```

```
{ user Object
> { _type String
> { created String
> { descriptionLinks String
> { displayName String
> { favouritesCount Int32
> { followersCount Int32
> { friendsCount Int32
> { id Int32
> { label String
> { link Object
> { listedCount Int32
> { location String
> { mediaCount Int32
> { profileBannerUrl String
> { profileImageUrl String
> { protected Boolean
> { rawDescription String
> { renderedDescription String
> { statusesCount Int32
> { url String
> { username String
> { verified Boolean
```

Before querying, you must ensure that a connection is set at the beginning.

```
client = MongoClient("mongodb+srv://fk1:mongodb22@cluster0.7mt03q2.mongodb.net/?retryWrites=true&w=majority")
database = client["tester"]
col = database["khashoggi"]
```

Below is a detailed description of each query and how it is built.

Query 1: Total Count

This query returns the total number of tweets collected. It simply counts the number of documents-items- in the collection.

```
def total_count():
    total = col.count_documents({})
    return jsonify(total)
```

Query 2: Number of Users

First, this function returns the number of users by collecting the user ids without repetition using distinct. Then, counting them using len(). Second, the function also returns the link of the user who has the highest number of tweets. So, the number of tweets for each user must be counted in addition to retrieving the profile link of the top user. For this, a pipeline is created. This pipeline is aggregated from the corresponding collection and saved in the cursor variable. The pipeline

searches for the records that have the `inReplyToTweetId` and `quotedTweet.id` fields null to avoid duplicates. Then, it groups the values of the `url` field that are the same (the `url` field points to the user's profile). This means that, if the same `url` is repeated, the same user tweeted more than once. While grouping the query is also counting the number of records grouped together and saving the values in a new field `count`. This `count` field is projected -outputted- in the final result in addition to the user's name and `url`. Moreover, this `count` is also sorted in descending order. The highest record is then selected using the `limit` aggregate. For a more feasible result and because the cursor is not serializable -result of mongo queries are not serializable-, a loop iterates over it.

```
def nb_of_users():
    nb_of_users = len(col.distinct('user.id'))
    pipeline = [
        {
            u"$match": {
                u"$and": [
                    {
                        u"inReplyToTweetId": None
                    },
                    {
                        u"quotedTweet.id": None
                    }
                ]
            }
        },
        {
            u"$group": {
                u"_id": u"$user.url",
                u"count": {
                    u"$sum": 1.0
                }
            }
        },
        {
            u"$project": {
                u"user.name": 1.0,
                u"user.url": 1.0,
                u"count": 1.0
            }
        }
    ]
```

```
        {
            u"$sort": SON([(u"count", -1)])
        },
        {
            u"$limit": 1.0
        }
    ]
    cursor = col.aggregate(pipeline)
    for c in cursor:
        link = c['_id']
    return jsonify(nb_of_users, link)
```

Query 3: Latest Date

This function returns the latest date tweets were posted in in the fetched data. First, a new collection is created called `changeformatofdate`. This collection has the date field split. This change in format helps access the fields of the date (DD, MM, YY) separately.

tester.khashoggi

STORAGE SIZE: 114.39MB LOGICAL DATA SIZE: 307.09MB

Find Indexes Schema Anti-Pattern

FILTER { field: 'value' }

```
_id: ObjectId('630eea340845c4aa91c26')
_type: "twitter.Tweet"
url: "https://twitter.com/billyshake"
date: "2019-11-29T21:27:45+00:00"
rawContent: "@CNNPolitics @gtconway3"
renderedContent: "@CNNPolitics @gtcc"
id: 1200526787651891200
> user: Object
  replyCount: 0
```

tester.changeformatofdate

STORAGE SIZE: 4.69MB LOGICAL DATA SIZE: 10.43MB TOTAL DOCUMENTS

Find Indexes Schema Anti-Patterns 1 Aggr

FILTER { field: 'value' }

QUERY RESULTS: 1-20 OF MANY

```
_id: ObjectId('633fecc8eac7af2d17994129')
> hashtags: Array
  updatedate: 2019-11-29T22:40:05.000+00:00
```

From this new collection, the pipeline is aggregated. The pipeline groups records having same day, month and year and counts as it groups. In other words, the pipeline is grouping records posted in the same day. Then, results are sorted with respect to the count in ascending order using the aggregate sort equal to 1. The first record is selected and saved in cursor representing latest date. For proper display formatting, the split date fields are concatenated together.

```
def latest_date():
    collection = database["changeformatofdate"]
    pipeline = [
        {
            "$group": {
                "_id": {
                    "year": {
                        "$year": "$updatedate"
                    },
                    "month": {
                        "$month": "$updatedate"
                    },
                    "day": {
                        "$dayOfMonth": "$updatedate"
                    }
                },
                "count": {
                    "$sum": 1
                }
            }
        },
        {
            "$sort": SON([("$count", 1)])
        },
        {
            "$limit": 1.0
        }
    ]
```

```
cursor = collection.aggregate(pipeline)
for _ in cursor:
    date = str(l['_id']['year']) + '-' + str(l['_id']['month']) + '-' + str(l['_id']['day'])
return jsonify(date)
```

Query 4: Languages

The function below returns the top 10 languages used in all fetched tweets with their count; how many tweets used each language. From the main collection, a pipeline is aggregated. The pipeline groups records based on the tweet's lang field and counts them. The count is sorted in descending order and the first 10 languages are selected. In the result, the language and its count are chosen to be projected.

```
def languages():
    pipeline = [
        {
            "$group": {
                "_id": "$lang",
                "count": {
                    "$sum": 1.0
                }
            }
        },
        {
            "$project": {
                "lang": 1.0,
                "count": 1.0
            }
        },
        {
            "$sort": SON([("$count", -1)])
        },
        {
            "$limit": 10
        }
    ]
    cursor = col.aggregate(pipeline)
```

Twitter assigns special lang values for a specific tweet content:

lang:und for undefined language

lang:qam for tweets with mentions only (works for tweets since 2022-06-14)

lang:qct for tweets with cashtags only (works for tweets since 2022-06-14)

lang:qht for tweets with hashtags only (works for tweets since 2022-06-14)

lang:qme for tweets with media links (works for tweets since 2022-06-14)

lang:qst for tweets with a very short text (works for tweets since 2022-06-14)

lang:zxx for tweets with either media or Twitter Card only, without any additional text (works for tweets since 2022-06-14)

To filter the results, the above special langs are defined in the list terms. Then, a loop iterates over the cursor that holds the top 10 languages. At each iteration, the language is checked whether it is any of the above special terms. If not, it is added to a dictionary with its count. This dictionary is then appended to a predefined list that is finally jsonified. The other fields added to the dictionary like value, full and column settings as well as calculating the percentage of each count is due to graph formatting purposes to be discussed later.

```
list = []
total = 0
terms = ["und", "qct", "qme", "qam", "qht", "qst", "zxx"]
for l in cursor:
    dict = {}
    if l['_id'] not in terms:
        dict['category'] = l['_id']
        total += l['count']
        dict['value'] = l['count']
        dict['full'] = 100
        dict['columnSettings'] = {'fill': 'chart.get("colors").getIndex(2)'}
        list.append(dict)
for j in list:
    j['value'] = (j['value']/total)*100
return jsonify(list)
```

Query 5: Countries

The query collects the top 10 countries of the fetched tweets. In other words, the query is collecting the most countries that interacted with the topic thus having the highest number of tweets written from their region. Same as in languages, the pipeline groups the tweets based on the location posted from.

```
def countries():
    pipeline = [
        {
            u"$group": {
                u"_id": u"$user.location",
                u"count": {
                    u"$sum": 1.0
                }
            }
        },
        {
            u"$project": {
                u"user.location": 1.0,
                u"count": 1.0
            }
        },
        {
            u"$sort": SON([(u"count", -1)])
        },
        {
            "$limit": 10
        }
    ]
    cursor = col.aggregate(pipeline)
```

The pycountry library is used here after being imported initially to help identify the alpha code of each country to be used later. The function returns a list in a json format holding the country name, code and count in addition to circle template.

```
list = []
results = []
for country in pycountry.countries:
    results.append({
        "key": country.alpha_2,
        "text": "{}".format(country.name),
    })
for l in cursor:
    for k in results:
        for country in pycountry.countries:
            if country.name in l['_id']:
                list.append({"id": country.alpha_2, "name": country.name,
                    "value": l["count"], "circleTemplate": {'fill': 'chart.get("colors").getIndex(2)'}})
return jsonify(list)
```

Query 6: Top Hashtags

This query is written in python where it returns the top ten hashtags of the fetched tweets. It starts by opening the json file the scraped data is saved in initially. Each line representing a tweet is turned into a python dictionary and then appended to a list named tweets. For each tweet, its hashtags field of type array is accessed. Each hashtag in this array is added to a dictionary with its

count in case it is encountered for the first time. If not, this counter by 1 means that this hashtag is repeated. Then, we increment the hashtag's counter. Note that the if condition is set to avoid abortion in case there's a null hashtags field – a tweet with no hashtags-. After counting how many times each hashtag is mentioned, it is time to sort them out to select the top 20. They are sorted using the sorted method in a descending order. From the sorted list, the first 20 are saved into top_20 variable. Another for loop iterates over the top_20 hashtags to fix their formatting for proper usages later.

```
def top_hashtags():  
    tweets = []  
    data = open(  
        r'F:\Al Maaref Uni\Second Year 2021-2022\Summer 21-22\Practical Training\Fatima Khadra Project\khashoggi.json',  
        "r", encoding="utf8")  
    for line in data:  
        tweets.append(json.loads(line))  
    dict = {}  
    for tweet in tweets:  
        if tweet["hashtags"] is None or tweet["hashtags"] == "null":  
            continue  
        for hashtags in tweet["hashtags"]:  
            if hashtags in dict:  
                dict[hashtags] += 1  
            else:  
                dict[hashtags] = 1  
    sort_orders = sorted(dict.items(), key=lambda x: x[1], reverse=True)  
    top_20 = sort_orders[0:20]  
    list = []  
    for l in top_20:  
        dict3 = {}  
        dict3['name'] = l[0]  
        dict3['weight'] = l[1]  
        list.append(dict3)  
    return jsonify(list)
```

Query 7: Sentimental Analysis

For the sentimental analysis, a set of functions are declared:

1. tweets_content(): this function analyzes the tweets and saves the results in a csv file. To do so, the SentimentIntensityAnalyzer library is imported initially. This library gives the sentiment intensity of a sentence. Start by writing a mongo query that projects the tweet's content represented by the rawContent field. Define tweets text list and append all tweets' content in. Turn the list into a dataframe respectively using the pandas library. Duplicate the tweets text column and name it tw_list. This new column will have the tweets' raw text with out any characters. Characters are removed using the re library to find out matches. Then, the text is all converted to lower characters.

Then using the textblob library, which is also a sentimental analysis tool, read the text. The results returned by the library will be saved in the added columns: polarity and subjectivity respectively. For further analysis, for each text again, using the SentimentIntensityAnalyzer, a score dictionary is resulted. This dictionary holds the values showing how much the text is negative, positive and neutral. These values are then split into columns. Then, another column is added named Sentiment where it states the sentiment of the text based on its polarity values. The sentiment column is passed to another function count_values_in_column for the percentages of each sentiment value to be calculated. For faster access, these results are then saved in a csv file.

```
def tweets_content():|
    cursor = col.find({}, {"_id": 0, "rawContent": 1})
    tweets_text, positive_tweets, negative_tweets, neutral_tweets = [], [], [], []
    for tweet in cursor:
        text = tweet["rawContent"]
        tweets_text.append(text)
    tweets_text = pd.DataFrame(tweets_text)
    neutral_tweets = pd.DataFrame(neutral_tweets)
    negative_tweets = pd.DataFrame(negative_tweets)
    positive_tweets = pd.DataFrame(positive_tweets)
    #save in csv
    tw_list = pd.DataFrame(tweets_text)
    tw_list["text"] = tw_list[0]
    remove_rt = lambda x: re.sub('RT @\w+: ', " ", x)
    rt = lambda x: re.sub("(@[A-Za-z0-9]+)|(^@-@A-Za-z \t))|(\w+:\/\/\S+)", " ", x)
    tw_list["text"] = tw_list.text.map(remove_rt).map(rt)
    tw_list["text"] = tw_list.text.str.lower()
    tw_list[['polarity', 'subjectivity']] = tw_list['text'].apply(lambda Text: pd.Series(TextBlob(Text).sentiment))
    for index, row in tw_list['text'].iteritems():
        score = SentimentIntensityAnalyzer().polarity_scores(row)
        neg = score['neg']
        neu = score['neu']
        pos = score['pos']
```

```
    if neg > pos:
        tw_list.loc[index, 'sentiment'] = "negative"
    elif pos > neg:
        tw_list.loc[index, 'sentiment'] = "positive"
    else:
        tw_list.loc[index, 'sentiment'] = "neutral"
    tw_list.loc[index, 'neg'] = neg
    tw_list.loc[index, 'neu'] = neu
    tw_list.loc[index, 'pos'] = pos
    tw_list_negative = tw_list[tw_list["sentiment"] == "negative"]
    tw_list_positive = tw_list[tw_list["sentiment"] == "positive"]
    tw_list_neutral = tw_list[tw_list["sentiment"] == "neutral"]
    per = count_values_in_column(tw_list, "sentiment")
    per.to_csv('F:\Al Maaref Uni\Second Year 2021-2022\Summer 21-22\Practical Training\data.csv')
```

2. `count_values_in_column()`: this function returns the number of each sentiment value calculates its percentages.

```
def count_values_in_column(data, feature):  
    total=data.loc[:,feature].value_counts(dropna=False)  
    percentage=round(data.loc[:,feature].value_counts(dropna=False,normalize=True)*100,2)  
    return pd.concat([total,percentage],axis=1,keys=['Total','Percentage'])
```

3. `Reader()`: this function returns the percentage of each sentiment and the text of the most positive, neutral and negative tweets. The percentage of each sentiment is saved in a variable and then appended in a list. The positive csv is read. This csv file has all the sentimental analysis calculated in the first function. To avoid repetition, a list of terms is declared. According to the term at each iteration, the text with the max value is detected and then appended to its corresponding term.

```
def reader():  
    df = pd.read_csv('F:\Al Maaref Uni\Second Year 2021-2022\Summer 21-22\Practical Training\data.csv')  
    pos_percentage = df.iat[2, 2]  
    neg_percentage = df.iat[0, 2]  
    neu_percentage = df.iat[1, 2]  
    df2 = pd.read_csv('F:\Al Maaref Uni\Second Year 2021-2022\Summer 21-22\Practical Training\positive.csv')  
    terms = ['neg', 'pos', 'neu']  
    g = globals()  
    list = []  
    for x in terms:  
        dict = {}  
        if x == 'neu':  
            s = df2[df2[f'{x}'] == df2[f'{x}'].max()][0]  
            g[f'{x}'] = s[0]  
            dict[f'{x}'] = g[f'{x}']  
            list.append(dict)  
        else:  
            g[f'{x}'] = (df2[df2[f'{x}'] == df2[f'{x}'].max()][0]).values.tolist()  
            dict[f'{x}'] = g[f'{x}']  
            list.append(dict)  
    return jsonify(pos_percentage, neg_percentage, neu_percentage, list)
```

Query 8: Verified Users

This function returns the percentage of the verified accounts. Using mongo, display all ids whose account is verified. Then, count the number of verified accounts by counting how many records are returned. Then calculate its percentage.


```
def verified_users():
    query = {}
    query["user.verified"] = True
    projection = {}
    projection["_id"] = 0.0
    projection["user.id"] = 1.0
    cursor = col.find(query, projection=projection)
    count = 0
    for l in cursor:
        count += 1
    percentage_of_verified = (count / 100000) * 100 #make this dynamic
    return jsonify(percentage_of_verified)
```

Query 9: Top Liked

This function returns the top liked tweet with its url, author and content.

```
def top_liked():
    pipeline = [
        {
            u"$project": {u"rawContent": 1, u"likeCount": 1, u"user.id": 1, u"_id": 0, u"url": 1}
        },
        {
            u"$sort": {u"likeCount": -1}
        },
        {
            "$limit": 1
        }
    ]
    cursor = col.aggregate(pipeline)
    list = []
    for l in cursor:
        list.append(l)
    return jsonify(list)
```

Query 10: Top Shared

This function returns the top shared tweet with its url, author and content.

```
def top_shared():
    pipeline = [
        {
            u"$project": {u"rawContent": 1, u"retweetCount": 1, u"user.id": 1, u"_id": 0, u"url": 1}
        },
        {
            u"$sort": {u"retweetCount": -1}
        },
        {
            "$limit": 1
        }
    ]
    cursor = col.aggregate(pipeline)
    list = []
    for l in cursor:
        list.append(l)
    return jsonify(list)
```

Therefore, the queries answered the following business questions:

1. What is the top hashtag?
2. What are the most languages used in the fetched tweets?
3. Which country scored more in speaking about this topic?
4. What is the top liked/shared tweet?
5. What emotions did the users reflect in their tweets?
6. How many verified users are there with respect to the whole dataset?

Data Visualizer:

This section will discuss how the query results are visualized in convenient charts. Highcharts and amcharts are used. Charts are distributed on several HTML pages according to what they discuss.

The templates are designed using HTML, CSS and bootstrap v5. To avoid duplications, fixed page template was inherited from a base html page via the jinja feature of flask.

```
</div>
{% block content3 %}
{% endblock %}

{% block content %}
{% endblock %}

{% block content2 %}
{% endblock %}
</main>
```

Flask web framework is used to build this web application in addition to an API to send and receive data between the web page and python.

For each HTML page, a python function is added in the app.py file. This function has a unique route to connect to and render its corresponding html page.

```
@app.route('/')
@app.route('/home')
def home():
    return render_template('main.html')
```

To build this web application, a flask project is created and the functions rendering the queries are added in the app.py. Note that, a connection to the database must be added at the beginning for the queries to run properly. Also, in app.py all routes and render_template are added in. In the flask project, create a templates folder where all html pages are added in and a static folder for the css stylesheets.

Dashboard Page:

This page reads from the user the topic he would like to visualize from a list of topics. To do, two functions that render the dashboard page are defined. One for get method and another for post method. Both methods render the same page and pass a names variable to the html page. This variable is a list that holds the collection names saved in the database. This list is looped over in the html page to be displayed. The result is saved in a variable and passed to the class.

```
@app.route('/Dashboard')
def dashboard():
    return render_template('Dashboard.html', names)

@app.route('/Dashboard', methods=['POST'])
def dashboard():
    n=request.form.get('topic_to_select')
    nn=myClass(n)
    return render_template('Dashboard.html', names)
```

```
<div class="offcanvas-body">
  <form class="d-flex" method="POST">
    <select name="topic_to_select">
      {% for o in data %}
        <option value="{ o }">{{ o }}</option>
      {% endfor %}
    </select>
    <button class="btn btn-outline-secondary" type="submit" href="#">Visualize</button>
  </form>
</div>
```

Building charts:

Charts are built using JS. Each function returns its result as a json -return jsonify()- for proper communication. Each function has a route sending its results via it. In the corresponding HTML page, in the script tag, the result across the API is read via JQuery or a JS fetch function. These functions have the python function's results passed to and rendered in the chart's function.

Each chart reads a specific data format. In other words, each chart has its data set in a specific order and keys. This explains the need for formatting the results in some functions. For example, notice below the function of route /nbofusers. This function returns a jsonified result. This result is read by a JS function in the corresponding HTML page. the fetch JS function takes as an argument the same route in order to read the returned data which is saved in the variable result. To display the data, a div tag of a unique id is created. This id is passed to the JS function. As a result, the data will be displayed in this div.

```
@app.route('/nbofusers')
def nb_of_users():
    nb_of_users = len(col.distinct('user.id'))
    pipeline = [...]
    cursor = col.aggregate(pipeline)
    for c in cursor:
        link = c['_id']
    return jsonify(nb_of_users, link)
```

```
<script>
fetch('/totalcount', {method: 'GET'})
  .then(res => res.json())
  .then(result => {document.getElementById("total_count").innerHTML = result;})

fetch('/nbofusers' {method: 'GET'})
  .then(res => res.json())
  .then(result => {
    let nb = result[0]
    let link = result[1]
    document.getElementById("nb_of_users").innerHTML = nb;
    document.getElementById("top_talker").setAttribute("href", link);
  })
</script>
```

```
<div class="text-xs font-weight-bold text-success text-uppercase mb-1">
  Number of Users</div>
<div class="h5 mb-0 font-weight-bold text-gray-800" id="nb_of_users"></div>
```

Final Result:

[bootcamp.mp4](#)

Recommendations:

1. Make the web pages responsive
2. Continue making it dynamic: user can choose a topic and visualize it
3. Add the data as a csv file so the user can download
4. Add more queries. For instance, the top hashtag in each country and the change of top hashtag throughout time.

5. Try fetching from Instagram as well.