Serverless Tweet Analysis Implementing Cloud Patterns

El-Asmar, Jad (1853357)

Supervisor: Sayed AmirHossein Abtahizadeh

Department of Computer and Software Engineering

Polytechnique Montreal

December 15, 2020

1 Introduction

In this project, I had to build a sentiment analysis system that uses some AWS services such as S3, Lambda, API Gateway, DynamoDB and EC2. In this paper, I will start by doing a performance evaluation. I will then talk about the issues I faced during this project. After that, I will be presenting the mechanism used to load the dataset into the DynamoDB table and the library used for the sentiment analysis. To end this paper, I added some future directions and a conclusion.

2 Performance evaluation

For all the performance evaluation, I decided to run each test five times and calculate an average on them. Here is the response time of the full sentiment analysis system on the first input file input1. We can see that the first call takes longer which is normal. Also, the mean response time for the full sentiment analysis system is around 101 000 ms. The first call is the one raising that average otherwise, by ignoring it I would obtain an average of around 98 000 ms.

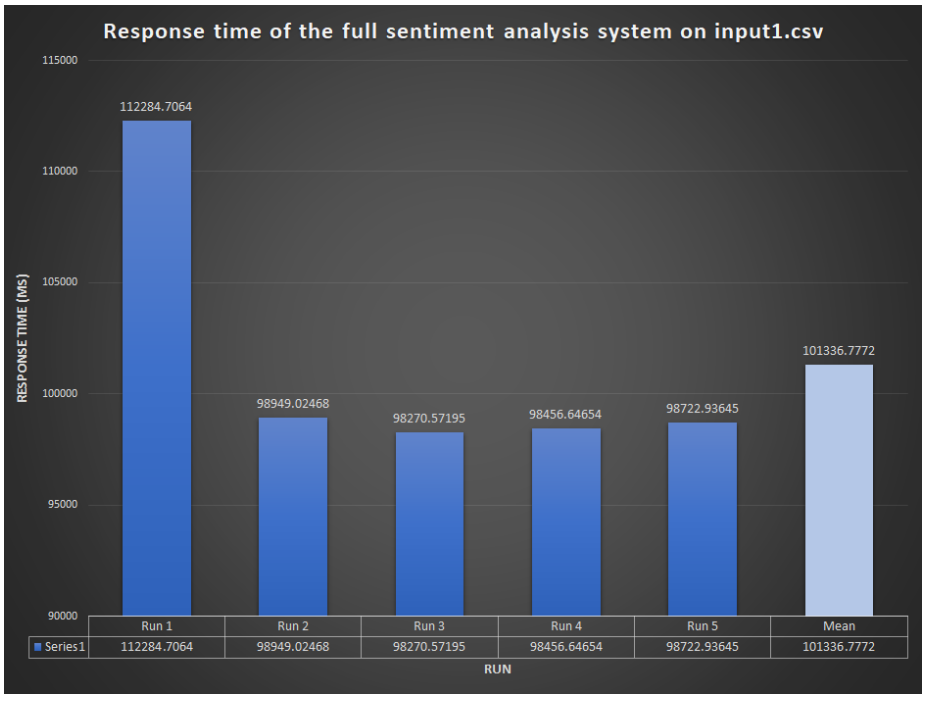

Figure 1: Response time in milliseconds for the full sentiment analysis system on input1.csv

Now that we know the full time of the sentiment analysis, I thought that it would be interesting to present the response time for each lambda function. By looking at figure 2, we can see that the second lambda function takes around 10 seconds more than the first lambda function. The mean response time for the

first lambda function is 45938.92648 ms while for the second lambda function it is 55839.92648 ms. The first lambda function executes the sentiment analysis on all the tweets in the input CSV and stores them in files in S3. It also saves a copy of the rawdata file in a JSON format. The second lambda function takes care of storing the data in DynamoDB.

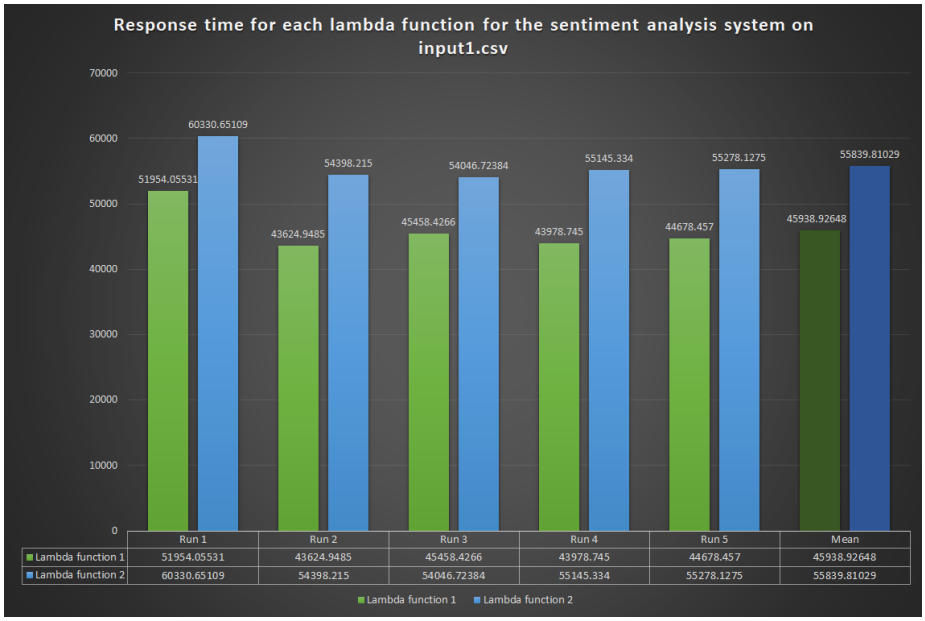

Figure 2: Response time in milliseconds each lambda function on input1.csv

Here is the response time of the full sentiment analysis system on the second input file input2. We can see that the first call takes longer like for input1. Since the input2 file is a lot smaller than the input1 file. Indeed the input1 is 13.3 MB and input2 is 3.0 MB. The mean response time for the full sentiment analysis system is around 14 000 ms.

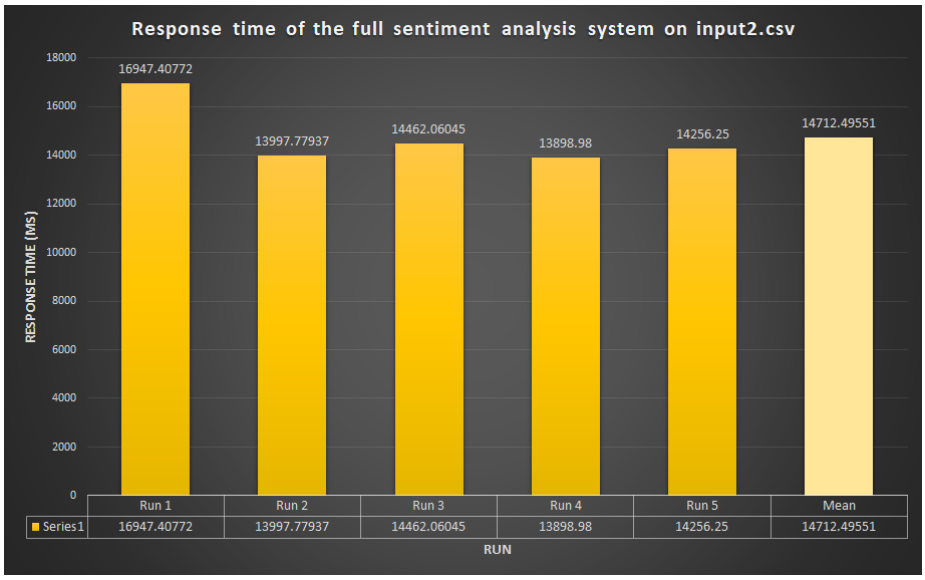

Figure 3: Response time in milliseconds for the full sentiment analysis system on input2.csv

As I did for the input1, I decided to measure the response time of each lambda function. In figure 4, it's interesting to see that for a smaller input file, the response time of my second lambda function is bigger. For the second input CSV, the mean response time for my first lambda function was 7377.934197 ms and for the second lambda function it's 6455.044455 ms.

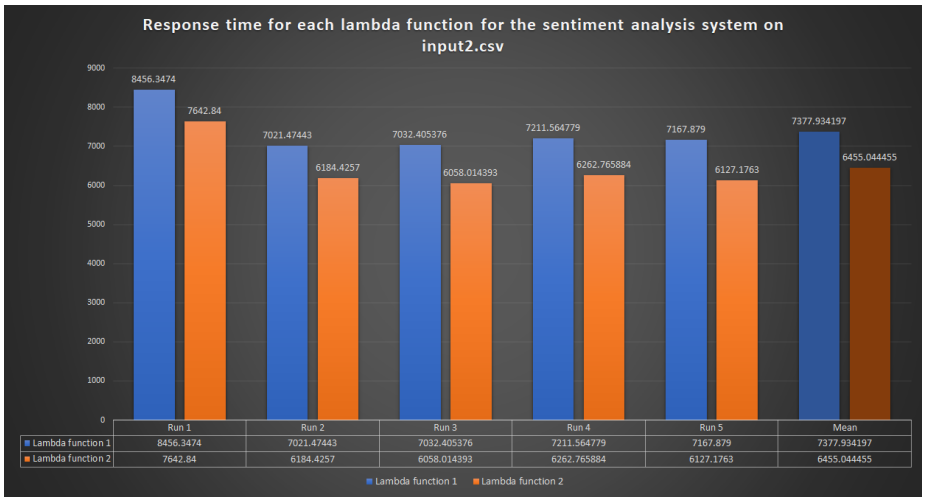

Figure 4: Response time in milliseconds each lambda function on input2.csv

For fetching results, for one single tweet, it takes around 617.0012474 ms. Again, the first call has a much bigger response time. If I ignore it, the mean response time would be around 400 ms.

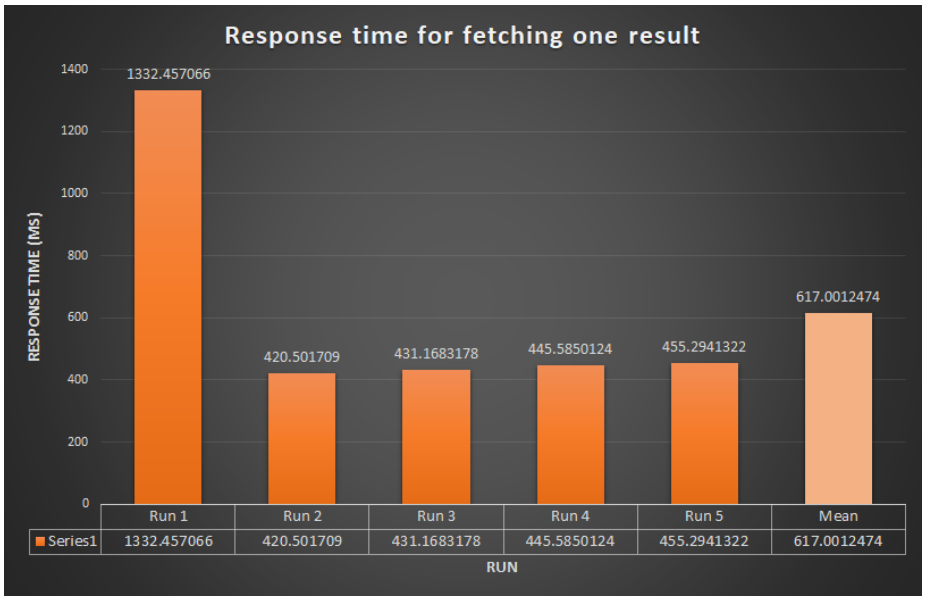

Figure 5: Response time in milliseconds for the fetching one individual result

For all the lambda functions, the first run always takes longer to initialize everything.

3 Issues

3.1 High cost of Amazon Comprehend

At the beginning of this project, I decided to go for Amazon Comprehend for my sentiment analysis. I wanted to try something new and this was my opportunity. After, doing many tests, everything seemed to work well. To use Amazon Comprehend, I had to specify the language of the batch of tweets or the single tweet I was passing. I first used langdetect to find the language of each tweet individually. Then, I found out that Amazon Comprehend has it's own language detection service that could return a list of languages for up to 25 tweets at a time. For my small tests, I used a JSON object of 10 tweets which seemed to work very well. When my implementation was done, I decided to run this sentiment analysis on all the input 1. The day after, I saw that my amount of credit went down a lot. I then did an investigation in my AWS console to see which service was the one to blame. I realized that Amazon Comprehend required a lot of money.

Price Per Unit

Feature	Up to 10M units	From 10M-50M units	Over 50M units
Key Phrase Extraction	$0.0001	$0.00005	$0.000025
Sentiment Analysis	$0.0001	$0.00005	$0.000025
Entity Recognition	$0.0001	$0.00005	$0.000025
Language Detection	$0.0001	$0.00005	$0.000025
PII Detection	$0.0001	$0.00005	$0.000025
Event Detection per event type	$0.003	$0.0015	$0.00075
Syntax Analysis	$0.00005	$0.000025	$0.0000125

Figure 6: Price per unit for Amazon Comprehend's services

For my case, the first input file contained around 100 000 tweets, but that does not mean each tweet is one unit. For Amazon comprehend, every 100 characters represent 1 unit. But when I read the official documentation, I saw that for the services related to natural language processing, I will be charged a minimum of 3 units per request. This was an issue, since almost all my tweets were below three units. Also by looking at figure 6, I see that I will be charged 0.0001$ for using sentiment analysis and the same amount for language detection, therefore a total of 0.0002$ per unit. For 100 000 tweets, and three units per tweet this would cost me around 60$. This was way too expensive for the number of credits I had in my AWS educate. I decided to use another sentiment analyzer, NLTK's sentiment analyzer Vader. [1]

3.2 Unsupported languages

For the sentiment analysis, I first decided to use Amazon Comprehend that will briefly be explained in one of the following sections. The methods used to detect the sentiment takes the text and a language code. Only twelve languages

are supported by the Amazon comprehend service for the sentiment analysis. Therefore, since I had to work with tweets from many languages. I decided to use the method detect from the library langdetect which takes a string as a parameter and returns the detected language. In my implementation, I first decided that if the detected language is not one of the supported ones I would set the language to English. After looking at Amazon Comprehend's documentation, I realized it had a function to detect the language for a batch of tweets. I choose to change my implementation to use Amazon Comprehend's language detection instead of langdetect to be able to retrieve the languages for a batch of tweets instead of one tweet at a time. Unfortunately, like described in the issue before, Amazon Comprehend was too expensive for the number of credits we had. Then, I decided to change my approach for the sentiment analyzer and went for NLTK's sentiment analyzer known as VADER.

3.3 Determining the right policies

For each lambda function, I had to do some researches to determine the right policies to assign to each lambda function. For example, at first, to be able to use Amazon Comprehend, I had to set the policy ComprehendFullAccess which provided full access to that service. Also, to be able to store items in DynamoDB, I had to add the policy AmazonDynamoDBFullAccess to the lambda function taking care of storing the tweets with their sentiment and score in DynamoDB.

3.4 Dynamo DB throttling

When writing inside my Dynamo DB table, I noticed that there was a lot of throttling. As we can see in figure 7 even when setting a higher write capacity we can see that the consumed amount is higher. Therefore, I decided to set my table as on demand. This would be the best option for me since I didn't know in advance how much write capacity was required.

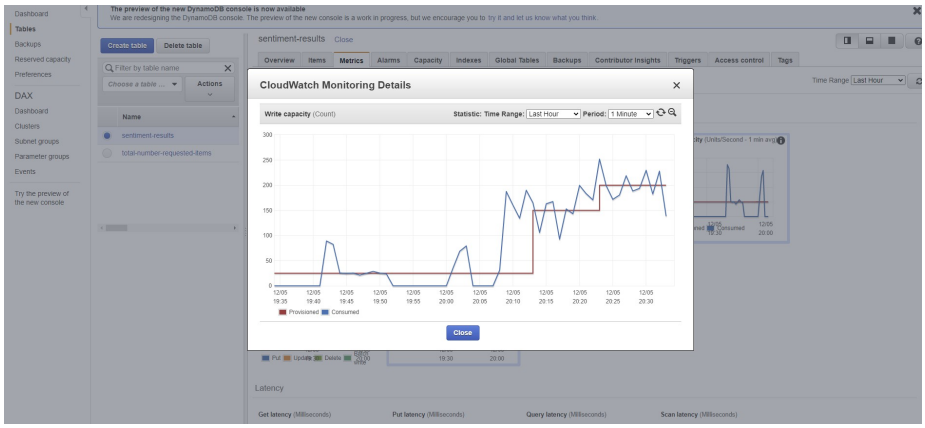

Figure 7: Throttling

3.5 Read timeout

Since some of my lambda functions take more than 60 seconds to finish depending on the size of the input data, my flask app would output an error indicating a read timeout. Therefore I decided to change my invocation type to Event instead RequestResponse. Event is asynchronous and my flask app would not need to wait for a response anymore. RequestResponse is synchronous which caused the error since the waiting time is more than 60 seconds. Now that I do not have the error anymore, I had to find a way to notify my Flask app that the last lambda function was done storing the data in DynamoDB. I decided to use a second DynamoDB table where an Item would be stored with a boolean field indicating if my lambda function was done storing the data. This method is explained more in the following section. My flask app would now loop until the item is updated with the boolean field set to True. When this was done, my Flask app updated the item with the boolean field back to false for the next sentiment analysis.

4 Mechanism used to load the dataset into the DynamoDB table

4.1 Flask app

Flask is a web framework written in Python. In our case, we had to develop an application that would communicate with AWS to perform a sentiment analysis on a set of tweets. At first, the user passes the input path to the CSV file containing the tweets in AWS S3. The complete path needs to be passed, for example in my case the path for input1.csv is s3://project-8415/rawdata/csv/input2.csv. My bucket's name is project-8415 and the file is inside the CSV folder that is inside the rawdata file. To run the post request, I used curl and executed the next call curl -X POST http://localhost:5000/sentiment/?data=s3://project-8415/rawdata/csv/input2.csv.

To be able to use Flask, I had to install it using pip and create an application.

```
from flask import Flask
from flask import request

app = Flask(__name__)
```

Listing 1: Flask usage

To retrieve the user's input, I extracted the argument using the request object by doing request.args.get('data'). I then had to pass that path to my lambda function. I created a dictionary and set the key `csv_path` to the value indicated by the user in the post request. Now that I have the path accessible to my code, I can invoke the lambda function that takes care of the sentiment analysis. I used boto3 to create a low-level client representing AWS Lambda. Then, to call my lambda function, I used the `invoke()` method. At first, I set the InvocationType to RequestResponse. This invokes the function synchronously. It keeps the connection open waiting for the function to return a response or to time out. This caused an issue because my lambda function would take more than 60 seconds to finish which caused a ReadTimeout. This issue was explained in the previous section. Therefore, I decided to invoke my function asynchronously by setting the InvocationType to Event.

```
csv_path = request.args.get('data')
data_to_send = {}
data_to_send["csv_path"] = csv_path
lambda_client = boto3.client("lambda")
response = lambda_client.invoke(FunctionName=
                                "sentimental-lambda",
                                InvocationType="Event",
                                Payload=
                                json.dumps(
                                data_to_send))
```

Listing 2: Data to send

The disadvantage of having an asynchronous approach is that I never receive a response from my lambda function call to confirm its success. This is why I decided to use another table when an item with the original table's name as key with a Boolean value `is_done` set to False by default. Once I finished storing all the data in my DynamoDB table, my second lambda function would update the item in the second table and set the value of `is_done` to True. In my Flask app, I implemented a while loop that loops until the value of the item is True for the `is_done` field.

As we can see in the following code, I used boto3 to create a DynamoDB resource. The table containing the field to set to false was named total-number-requested-items. I used the method `get_item()` by providing the corresponding key which in my case was the original table's name. This method returns a dictionary with the Item field containing the actual information of the requested item with the given id.

When the concerned field becomes True, the flask app updates the item back to set the `is_done` field to false for the next calls. To do so, I used the `update_item()` method passing the key of the item I wish to update, the update expression and the expression attribute values which are the values of the field I want to update. In this situation, it is the `is_done` field and the value is False. Also, I set the ReturnValues to `UPDATED_NEW` to only return the updated attributes as they appear after being updated.

```
dynamo_client = boto3.resource("dynamodb")
table2 = dynamo_client
.Table("total-number-requested-items.")

while table2.get_item(Key={'table_name': "results"})
["Item"]["is_done"] != True:
    continue
else:
    updated_item = table2.update_item(
        Key={
            'table_name': "results"
        },
        UpdateExpression="set is_done=:d",
        ExpressionAttributeValues={
            ':d': False
        },
        ReturnValues="UPDATED_NEW"
    )
```

Listing 3: Wait for confirmation

Here is what the object sent from the flask app to the first lambda function looks like:

```
{
  "csv_path": "s3://project-8415/rawdata/csv/input1.csv"
}
```

Listing 4: Object sent from Flask app to lambda function 1

As we can see, there is only the key `csv_path` with the path to the S3 file containing the rawdata in the CSV format.

4.2 Lambda function 1

For my first lambda function, it is the one that would take care of the sentiment detection. To do so, I decided to use VADER, an NLTK sentiment intensity analyzer. I decided that this lambda function would take care of detecting the sentiments and the scores. Then it would store the results in a CSV file and a JSON file in the corresponding directories in S3. The path to the output CSV result will be passed to the next lambda function in the Payload by invoking it. To allow my lambda function to invoke another lambda function and store files in S3, I had to set up an IAM role. This role contained AWSLambdaFullAccess and AmazonS3FullAccess. These are policies that allowed my lambda function to have full access to lambda and S3. The flask app invokes this lambda function and passes a dictionary in its payload with the s3 path to the rawdata. Using boto3, I created an AWS S3 resource. To read the CSV, I used the `read_csv()` from pandas library. This required me to add a layer. My lambda function requires external modules such as pandas, NLTK and S3FS. S3FS is the Py-Filesystem interface to Amazon S3 cloud storage. It allows me to work with S3 in the same way as any other supported file system.

To use VADER, I downloaded it from NLTK like presented in the code snippet below.

```
from nltk.sentiment.vader import
SentimentIntensityAnalyzer

# Download vader from nltk
nltk.data.path.append("/tmp")
nltk.download('vader_lexicon', download_dir="/tmp")
```

Listing 5: Download NLTK VADER

At the beginning of my lambda function, I started by reading the CSV file that is at the path passed by my Flask app's POST request. I used pandas to read the S3 rawdata file and saved its content inside a JSON file in S3. I did so by converting the CSV file to JSON by using pandas' method `to_json()` with an index orientation. The code used is visible in the next code snippet.

```
s3_resource = boto3.resource('s3')
all_infos = {}
bucket_name = "project-8415"
csv_path = event["csv_path"]
csv_file_name = str(os.path.basename(csv_path))
json_file_name = csv_file_name.replace("csv", "json")

# Read rawdata CSV
raw_data_csv = pd.read_csv(csv_path)
```

```
# Save JSON rawdata
json_rawdata = f"rawdata/json/{json_file_name}"

# Save JSON file with rawdata
s3_resource.Object(bucket_name, json_rawdata).put(
Body=raw_data_csv.to_json(orient='index'))
```

Listing 6: Lambda function 1 saving raw data

Now that I have my rawdata saved in my S3 bucket in a CSV and a JSON format. I can start my sentiment analysis. I then create the sentiment analyzer object to call the method `polarity_scores()` on every tweet in the rawdata file. While implementing the function, I realized that some of the values in the input CSV where NAN. Therefore, I decided to replace these values with a string "None" and set the sentiment to neutral with a score of 0 by default. For the other tweets, I used their compound score like suggested in the documentation of NLTK. If the compound score was greater or equal to 0.05 then the tweet is positive, if the tweet's compound score was between -0.05 and 0.05, the tweet would be neutral otherwise the tweet is considered negative. NLTK VADER will be explained with more details in one of the following sections.

```
# Replace nan to None
raw_data_csv.fillna("None", inplace = True)
all_infos["raw_csv"] = event["csv_path"]
texts_list = raw_data_csv['text'].tolist()


# Create sentiment analyzer
sentiment_analyzer = SentimentIntensityAnalyzer()
scores_list = []
sentiments_list = []
for sentence in texts_list:

    if sentence == "None":
        sentiments_list.append("Neutral")
        scores_list.append(0.0)
    else:
        sentiment_results =
        sentiment_analyzer.polarity_scores(sentence)
        scores_list.append(sentiment_results["compound"])
        # If the detected sentiment is positive
        if sentiment_results["compound"] >= 0.05:
            sentiments_list.append("Positive")
         # If the detected sentiment is neutral
        elif sentiment_results["compound"] > -0.05 and
        sentiment_results["compound"] < 0.05:
            sentiments_list.append("Neutral")
        # If the detected sentiment is negative
        else:
            sentiments_list.append("Negative")
```

```
raw_data_csv["score"] = scores_list
raw_data_csv["sentiment"] = sentiments_list
```

Listing 7: Lambda function 1 sentiment analysis part

Now my dataframe contains all the information that I need. I added a column score with the compound scores and a sentiment column to indicate if the tweet is positive, neutral, or negative. To save the data, I used the S3 client once again. The path to the result CSV file is added in a dictionary that will be passed to the payload when invoking the second lambda function.

```
# Save results
csv_results_name = "sentiments/csv/" + csv_file_name
json_results_name =
csv_results_name.replace("csv", "json", 2)
all_infos["results_json"] = json_results_name
all_infos["results_csv"] = csv_results_name
final_dict = {}
for key, value
in raw_data_csv.to_dict(orient="index").items():
    final_dict[value["id"]] = value

csv_buffer = StringIO()
raw_data_csv.to_csv(csv_buffer, index=False)
s3_resource.Object(bucket_name, csv_results_name).put(
Body=csv_buffer.getvalue())
s3_resource.Object(bucket_name, json_results_name).put(
Body=json.dumps(final_dict))
lambda_client = boto3.client("lambda")
print(all_infos)

s = time.time()
response = lambda_client.invoke(
                        FunctionName="store-results",
                        InvocationType="RequestResponse",
                        Payload=json.dumps(all_infos))
```

Listing 8: Lambda function 1 saving results as CSV and JSON in S3

The results stored in S3 have the following syntax. I choose to use the id as the primary key for each tweet, in case I would like to access the data related to a specific tweet by using its id. The full syntax of each tweet stored in an S3 file is represented in the following code snippet. Note that we have the text, date, id, score and sentiment of a tweet. I choose to put all the information that we have for a tweet.

```
{
    1113125043213660000":
        {
            "date": "2019-04-02 17:04",
            "text": " @jobillico presents 7key #hr #trends
```

```
            that have emerged in #2018
            https://bit.ly/2ANQ6vc #job #hr #trends #2018
            #mtl #qc",
            "id": 1113125043213660000,
            "score": 0.0,
            "sentiment": "Neutral"
        }
}
```

Listing 9: Result for one tweet in JSON format

Here is the full code for my first lambda function.

```
import json
import pandas as pd
import nltk
import boto3
import os
import time
from io import StringIO

def lambda_handler(event, context):
    s3_resource = boto3.resource('s3')
    all_infos = {}
    bucket_name = "project-8415"
    csv_path = event["csv_path"]
    csv_file_name = str(os.path.basename(csv_path))
    json_file_name =
    csv_file_name.replace("csv", "json")

    # Read rawdata CSV
    raw_data_csv = pd.read_csv(csv_path)

    # Save JSON rawdata
    json_rawdata = f"rawdata/json/{json_file_name}"

    # Save JSON file with rawdata
    s3_resource.Object(bucket_name, json_rawdata)
    .put(Body=raw_data_csv.to_json(orient='index'))

    # Replace nan to None
    raw_data_csv.fillna("None", inplace = True)
    all_infos["raw_csv"] = event["csv_path"]
    texts_list = raw_data_csv['text'].tolist()


    # Create sentiment analyzer
    sentiment_analyzer = SentimentIntensityAnalyzer()
    scores_list = []
    sentiments_list = []
    for sentence in texts_list:
```

```
        if sentence == "None":
            sentiments_list.append("Neutral")
            scores_list.append(0.0)
        else:
            sentiment_results =
            sentiment_analyzer.polarity_scores(sentence)
            scores_list
            .append(sentiment_results["compound"])

            # If the detected sentiment is positive
            if sentiment_results["compound"] >= 0.05:
                sentiments_list.append("Positive")

             # If the detected sentiment is neutral
            elif sentiment_results["compound"] > -0.05 and
            sentiment_results["compound"] < 0.05:
                sentiments_list.append("Neutral")

            # If the detected sentiment is negative
            else:
                sentiments_list.append("Negative")

    raw_data_csv["score"] = scores_list
    raw_data_csv["sentiment"] = sentiments_list

    # Save results
    csv_results_name = "sentiments/csv/" + csv_file_name
    json_results_name =
    csv_results_name.replace("csv", "json", 2)

    all_infos["results_json"] = json_results_name
    all_infos["results_csv"] = csv_results_name
    final_dict = {}

    for key, value
    in raw_data_csv.to_dict(orient="index").items():
        final_dict[value["id"]] = value

    csv_buffer = StringIO()
    raw_data_csv.to_csv(csv_buffer, index=False)
    s3_resource.Object(bucket_name, csv_results_name)
    .put(Body=csv_buffer.getvalue())

    s3_resource.Object(bucket_name,json_results_name)
    .put(Body=json.dumps(final_dict))
    lambda_client = boto3.client("lambda")
    print(all_infos)

    s = time.time()
```

```
    response =
    lambda_client.invoke(FunctionName="store-results",
                         InvocationType="RequestResponse",
                         Payload=json.dumps(all_infos))

    print(response["Payload"].read())
    e = time.time()
    print(e-s)

    return {
        'statusCode': 200,
        'body': json.dumps(
        'Sentiment Analysis is done
        and results are stored in S3!')
    }
```

Listing 10: Full code for lambda function 1

Here is what the object sent from lambda function 1 to lambda function 2 looks like. As we can see there is a key `results_csv` with the path to the s3 results in the CSV format.

```
{
  "results_csv": "sentiments/csv/input1.csv"
}
```

Listing 11: Object sent to the second lambda function

4.3 Lambda function 2

Now that I saved my sentiment analysis results in a CSV and a JSON file in S3, I can store them in DynamoDB. The first lambda function passed a dictionary with the S3 path to the CSV file with the results. To be able to access the S3 file I had to create another IAM role. This new role contained two policies AmazonS3FullAccess and AmazonDynamoDBFullAccess. Indeed, I had to access S3 and DynamoDB. I created an AWS S3 resource and one representing DynamoDB. Here is the code snippet to load the results from the CSV file stored in S3.

```
s3 = boto3.resource('s3')
bucket_name = "project-8415"
path_to_csv_result = "s3://project-8415/"
                     + event["results_csv"]
res_csv = pd.read_csv(path_to_csv_result)
```

Listing 12: Code to load the results from S3 CSV file

At first, to add an item in my DynamoDB table, I was adding them one tweet at an item by using the boto3 method `put_item()`. By looking at the documentation of boto3 I discovered that I could put items by batch. Therefore by calling `table.batch_writer()`, I was able to write 25 tweets at a time. When running

my code I had an error telling me that I was trying to insert a tweet with the same keys. By looking at the CSV, I realized that there were two tweets with the same id and the same date but with a different text. I could have removed one of the duplicates in the CSV file, but I finally decided to pass the parameter `overwrite_by_pkeys` to the batch writer. I also had to specify the keys like so: `table.batch_writer(overwrite_by_pkeys=['id', 'date'])`. Another error occurred when storing the tweets in DynamoDB. It did not support float types. Since my scores were all floats, I converted them into Decimals which is a supported type in DynamoDB. Here is the code to store all the tweets in my DynamoDB table.

```
# DynamoDB part
dynamo_start = time.time()
dynamo_client = boto3.resource("dynamodb")
# table = dynamo_client.Table("sentiment-results")
table = dynamo_client.Table("results")
print(res_csv.to_dict(orient="index"))
with table.batch_writer(overwrite_by_pkeys=['id', 'date'])
                                                    as batch:
    for tweet in res_csv.to_dict(orient="index").values():
        tweet["score"] = Decimal(str(tweet["score"]))
        batch.put_item(
            Item=tweet)
dynamo_end = time.time()
print("Dynamo_time")
print(dynamo_end - dynamo_start)
```

Listing 13: Code to write results in DynamoDB

My flask app is waiting to see the Boolean value `is_done` in the second table set to True. Since I was done storing the tweets in DynamoDB, I had to notify my flask app, that I was done. Therefore, by using boto3 and the DynamoDB resource, I accessed that table and updated the item to have the value True to its `is_done` field by using the `update_item()` method. This would notify my flask app that my second lambda function was done storing all the results of the tweets that were in the input CSV file that was passed in the beginning. Here is the part of the code that updated the corresponding field.

```
table2 = dynamo_client.Table("total-number-requested-items")
response = table2.update_item(
    Key={
        'table_name': "results"
    },
    UpdateExpression="set is_done=:d",
    ExpressionAttributeValues={
        ':d': True
    },
    ReturnValues="UPDATED_NEW"
)
```

Listing 14: Code to update field in second table

Here is the full code of my second lambda function.

```
import json
import boto3
import time
import pandas as pd
from decimal import Decimal


def lambda_handler(event, context):
    path_to_csv_result = "s3://project-8415/"
                         + event["results_csv"]
    res_csv = pd.read_csv(path_to_csv_result)

    # DynamoDB part
    dynamo_start = time.time()
    dynamo_client = boto3.resource("dynamodb")
    table = dynamo_client.Table("results")
    print(res_csv.to_dict(orient="index"))
    with table.batch_writer(
                overwrite_by_pkeys=['id', 'date']) as batch:
        for tweet
        in res_csv.to_dict(orient="index").values():
            tweet["score"] = Decimal(str(tweet["score"]))
            batch.put_item(
                Item=tweet)
    dynamo_end = time.time()
    print("Dynamo_time")
    print(dynamo_end - dynamo_start)

    table2 =
    dynamo_client.Table("total-number-requested-items")
    response = table2.update_item(
        Key={
            'table_name': "results"
        },
        UpdateExpression="set is_done=:d",
        ExpressionAttributeValues={
            ':d': True
        },
        ReturnValues="UPDATED_NEW"
    )
    return {
        'statusCode': 200,
        'body': json.dumps(
            'Done storing data in DynamoDB!')
    }
```

Listing 15: Lambda function 2 full code

Note that for my first and second lambda functions, I used a memory of

1024 MB instead of the original 128 MB and set the timeout to 15 minutes to avoid being interrupted since the original timeout is only 3 seconds.

4.4 Lambda function 3

To get the results from my DynamoDB table and get an atomic number that is incremented every call, I used the same second table that is used to notify my Flask app that the storing in my DynamoDB is done. I used the same item with the `table_name` as key and added a field `count` that would be incremented on every API call. That means the index should be increasing every time we call my API end-point.

I created my API using API Gateway. I added a resource `/sentiment`, then attached a GET method to it by linking it to my third lambda function. My lambda function starts by extracting the two necessary query parameters id and date. Then by using a DynamoDB resource using boto3, I was able to access the table and update the count of the item with my table's name as id.

```
id = int(event["queryStringParameters"]["id"])
date = event["queryStringParameters"]["date"]
dynamo_client = boto3.resource("dynamodb")
table = dynamo_client.Table("sentiment-results")
response = table.get_item(Key={"id": id, "date": date})

# Update counter item in other dynamoDB table
dynamo_client = boto3.resource("dynamodb")
number_items_table =
dynamo_client.Table("total-number-requested-items")
number_items_response = number_items_table.update_item(
    Key={"table_name": "sentiment-results"},
    UpdateExpression="SET #attr = #attr + :val",
    ExpressionAttributeNames={
        "#attr": "count"
    },
    ExpressionAttributeValues={
        ":val": 1
    },
    ReturnValues="UPDATED_NEW"
)
```

Listing 16: Lambda function 3

For the response of this lambda function, I created a dictionary that would have the count as main key and a dictionary as value. This dictionary contained the id, the sentiment and the score of the requested tweet.

```
formatted_response = {}
formatted_response[
str(number_items_response["Attributes"]["count"])] = {
    "id": int(response["Item"]["id"]),
```

```
    "sentiment": response["Item"]["sentiment"],
    "score": float(response["Item"]["score"])
}
```

Listing 17: Lambda function 3 response

Here is the full code of my third lambda function. We can see that the response is returned at the end with the status code.

```
import json
import boto3
from decimal import Decimal


def lambda_handler(event, context):
    id = int(event["queryStringParameters"]["id"])
    date = event["queryStringParameters"]["date"]
    dynamo_client = boto3.resource("dynamodb")
    table = dynamo_client.Table("sentiment-results")
    response = table.get_item(Key={"id": id, "date": date})

    # Update counter item in other dynamoDB table
    dynamo_client = boto3.resource("dynamodb")
    number_items_table =
    dynamo_client.Table("total-number-requested-items")
    number_items_response = number_items_table.update_item(
        Key={"table_name": "sentiment-results"},
        UpdateExpression="SET #attr = #attr + :val",
        ExpressionAttributeNames={
            "#attr": "count"
        },
        ExpressionAttributeValues={
            ":val": 1
        },
        ReturnValues="UPDATED_NEW"
    )
    formatted_response = {}
    formatted_response[
    str(number_items_response["Attributes"]["count"])] = {
        "id": int(response["Item"]["id"]),
        "sentiment": response["Item"]["sentiment"],
        "score": float(response["Item"]["score"])
    }
    print(formatted_response)

    return {
        'statusCode':
        response["ResponseMetadata"]["HTTPStatusCode"],
        'body': json.dumps(formatted_response)
    }
```

Listing 18: Lambda function 3

When fetching the result, this is what the response looks like:

```
{"20":
    {
        "id": 1123373654245500000,
        "sentiment": "Negative",
        "score": -0.29600000000000004,
        "time": 0.4260902404785156
    }
}
```

Listing 19: Response of get single tweet with API Gateway call

5 Library used for sentiment analysis

5.1 Amazon Comprehend

To perform sentiment analysis, I decided to use one of AWS's services to learn new technologies and expand my knowledge in AWS. Amazon Comprehend is a natural language processing service using machine learning to do many tasks. It offers Keyphrase Extraction, Sentiment Analysis, Entity Recognition, Topic Modeling, and Language Detection APIs.[2]

In my case, I only had to use the sentiment analysis to return the overall sentiment of a given text. To do so I used boto3 to create a low-level client representing Comprehend.

```
import boto3

client = boto3.client('comprehend')
```

Listing 20: The low-level client representing Amazon Comprehend

I then called the method `detect_sentiment()` to inspect a text and return an inference on the overall sentiment. The sentiment can be positive, negative, neutral or mixed.

```
response = client.detect_sentiment(
    Text='The actual text',
    LanguageCode='en'|'es'|'fr'|'de'|'it'|'pt'|'ar'|'hi'
    |'ja'|'ko'|'zh'|'zh-TW'
)
```

Listing 21: The request syntax for $detect_sentimentmethod$

As we can see, the first parameter is a string that represents the text we which to detect the sentiment. It is a UTF-8 text string and it must contain fewer than 5,000 bytes of UTF-8 encoded characters. The second parameter is the language code represented by a string. As we can see, only twelve languages are supported. This was one of the issues I faced and that was explained in one of the previous sections.

```
{
   "Sentiment": "string",
   "SentimentScore": {
      "Mixed": number,
      "Negative": number,
      "Neutral": number,
      "Positive": number
   }
}
```

Listing 22: The response syntax for $detect_sentimentmethod$

The data returned is in JSON format. The first element is the inferred sentiment that Amazon Comprehend has the highest level of confidence in. The SentimentScore is an object that lists the sentiments and their corresponding confidence levels.

5.2 NLTK VADER

Since Amazon Comprehend cost a lot of money, I decided to go for VADER. Its name stands for Valence Aware Dictionary and sEntiment Reasoner. It is considered a lexicon and rule-based sentiment analysis tool trained on social media data. I decided to use this tool because based on many articles it gives very good results on social media-type texts. This is a good advantage since our data are tweets from Twitter. Also, VADER does not require any training data since it was built from a standard sentiment lexicon. Compared to Amazon Comprehend, it was very fast which is good since I had a big amount of data to analyze. [3]

5.2.1 Installation

To be able to use the Sentiment Intensity Analyzer VADER, I had to download it from NLTK. I also could have installed it locally with the pip command and extract it as a zip file to upload it as a layer in my lambda function. As we can see in the following code snippet, I had to create a temporary directory to download VADER. Then I could import the SentimentIntensityAnalyzer object.

```
from nltk.sentiment.vader import SentimentIntensityAnalyzer

# Download Vader from nltk
nltk.data.path.append("/tmp")
nltk.download('vader_lexicon', download_dir="/tmp")
```

Listing 23: Installation commands used

5.2.2 Sentiment analysis

To start a sentiment analysis, I first started by reading the data from the CSV passed to the lambda function. The library pandas was used to do so. The list of texts to analyze was the extracted from the column text in the dataframe. To detect the sentiment and the scores the following code was called. As we can see, the `polarity_scores()` method is called on each tweet. At first, I had errors since some values were NAN. Using one of pandas methods `fillna()` , I replaced all the nan values with a string saying "None." Then, later on, I set all the sentences only containing "None" to a neutral sentiment with a score of 0.0. After, analyzing all the tweets, I assigned the list of sentiments and the list of scores to two dataframe columns.

```
# Read rawdata CSV
raw_data_csv = pd.read_csv(csv_path)

# Replace nan to None
raw_data_csv.fillna("None", inplace = True)
```

```
# Extract list of tweets
texts_list = raw_data_csv['text'].tolist()

# Create sentiment analyzer
sentiment_analyzer = SentimentIntensityAnalyzer()
scores_list = []
sentiments_list = []
for sentence in texts_list:

    if sentence == "None":
        sentiments_list.append("Neutral")
        scores_list.append(0.0)
    else:
        sentiment_results =
        sentiment_analyzer.polarity_scores(sentence)
        scores_list.append(sentiment_results["compound"])
        if sentiment_results["compound"] >= 0.05:
            sentiments_list.append("Positive")
        elif sentiment_results["compound"] > -0.05
        and sentiment_results["compound"] < 0.05:
            sentiments_list.append("Neutral")
        else:
            sentiments_list.append("Negative")

# Setting two columns with the sentiments and scores
raw_data_csv["score"] = scores_list
raw_data_csv["sentiment"] = sentiments_list
```

Listing 24: Sentiment Analysis with NLTK VADER

For each sentence, the obtained results followed the syntax represented in the following code snippet.

```
{'neg':0.0, 'neu':0.326, 'pos':0.674, 'compound':0.7351}
```

Listing 25: Result of a sentiment analysis by VADER

5.2.3 Scoring

For the scoring, this tool provides a compound score for each tweet. It is obtained by summing the valence scores for each word in the lexicon. Then the obtained score is normalized to be a value between -1 and 1. The negative value indicates the most extreme negative and the positive value the most extreme positive. For the threshold, I went for the ones recommended by the authors of this tool.

- **1. Positive sentiment** Compound score greater or equal than 0.05
- **2. Neutral sentiment** (Compound score greater than 0.05) and (Compound score lesser than 0.05)
- **3. Negative sentiment** Compound score lesser or equal to -0.05

6 Future Directions

In the case of this laboratory, I had a limited amount of credits. It would have been interesting to work with more AWS tools related to machine learning such as Amazon SageMaker, Amazon Fraud Detector, and many others. Also, the limit of credits of 200$ can't really allow us to experiment with many of those tools at their full capacity. As I experienced with Amazon Comprehend, it costs a lot. Also, even though it isn't really a scope for this course, it would have been interesting to preprocess the tweets to obtain better results. For example, apply tokenization, remove punctuation, hashtags and many other elements that could have improved my results. To make a the system faster, multi-threading could have been used in our lambda function or when invoking the lambda function to parallelize the whole process.

7 Conclusion

Thanks to this assignment, I was able to use many useful AWS tools. This will certainly add value to my resume and I realized that I really enjoyed working with AWS. I would like to expand my knowledge of cloud services and maybe learn to work with more AWS tools related to Machine learning. I managed to create a sentiment analysis system that works well by using a serverless architecture. I am sure that I will be using AWS in the future for my projects and hopefully in my career as a software engineer.

I presented the response times for my full sentiment analysis system and for each of my lambda function depending on the input file. I also presented some of the issues I met and how I faced them. In another section, I presented the mechanism used to load the dataset into the DynamoDB table by explaining my code and all the details related to my implementation. It consisted of explaining my Flask app, the lambda function that did the sentiment analysis and stored my results in S3, the lambda function that stored the results in DynamoDB and the lambda function that fetches the result by using an API Gateway. I presented the different libraries used for sentiment analysis by presenting how they work.

References

[1] Amazon comprehend pricing. Available at https://aws.amazon.com/comprehend/pricing/?nc1=h_ls. Accessed: 2020-12-08.

[2] A quick look at natural language processing with amazon comprehend. Available at https://julsimon.medium.com/a-quick-look-at-natural-language-processing-with-amazon-comprehend-238b8d9ec11d. Accessed: 2020-12-10.

[3] Simplifying sentiment analysis using vader in python (on social media text). Available at https://medium.com/analytics-vidhya/simplifying-social-media-sentiment-analysis-using-vader-in-python-f9e6ec6fc52f. Accessed: 2020-12-10.