How Similar Are We to Our President? A Use Case for Working with Many Kafka Streams and Deep Learning Technologies in Python

Nathan Cooper Jones
CSP 554
December 4th, 2019
Computer Science Department
Illinois Institute of Technology
Chicago, Illinois 60616
njones10@hawk.iit.edu

***Abstract* – With the demand for data science modeling exceeding the limitations of batch processing, many organizations have turned towards real-time processing of data through deep learning models by consuming rows of data through a messaging stream. This project seeks to develop a framework for this pattern, relying on a Python application that allows a user to input a variable-length sentence, which is published to a Kafka stream and consumed by another node that transforms the sentence to a fixed length embedding via a BERT model. This node sends the embedding to a separate Kafka stream which is consumed by a final node that publishes the data to a Google BigQuery table, and using a SQL command, returns the most similar embedding in a source table. This paper goes over implementation details, optimizations and decisions made for this specific use-case, and a plethora of future improvements. The code used for this project is made publicly available on GitHub.**

Index Terms – BERT, data science, Kafka, Python, Trump, Twitter.

INTRODUCTION

With massive amounts of computational power available at an instant for low costs, and more data being produced than machines to analyze it, we live in a booming age of computing, specifically for big data technologies. Advanced GPU capabilities have allowed data scientists to train massive deep neural networks that are able to analyze images, text, and more with higher accuracy than many traditional statistical learning approaches. Naturally, many companies have business use-cases to process a large volume and velocity of data through these deep learning models where using a traditional batch processing method may be too slow for some real-time purposes. For data being published on a messaging stream to be consumed by many different applications, it is useful for a data scientist to be able to adapt to this data transport style and work with these streams to process data through a complicated model, and even publishing intermediary results to *another* stream for *another* model, and so on. This framework for consuming data streams for data science modeling and sending results to another data stream is not yet popular, but has great potential for real-time analysis.

This project seeks to develop a small use-case exemplifying how this workflow would look, specifically using Python, Apache Kafka messaging, a BERT deep learning model, and Google BigQuery database tables.

Specifically, this project sets out to build an application that uses two Apache Kafka streams: one for taking raw data to a transformer, and one that takes transformed data to an external database table, to generate sentence similarity between an inputted sentence of variable length and a database composed of pre-computed sentence embeddings.

This project is composed of two phases that will be discussed in the *Building the Project* section of this paper: 1) the initial setup and 2) the client-side

application. Initially, I will begin by downloading all tweets ever made by Donald Trump (as this serves as a large enough dataset to use as our data to compare against), process each of them through a Siamese BERT / RoBERTa model that transforms a variable-length sentence into a fixed numeric encoding, and store them into a Google BigQuery NewSQL database table.

For the client-side, I plan to build a simple user-interface that accepts an English sentence, sends the data blob containing the sentence through an initial Kafka stream to a node that transforms the sentence using the modified BERT model. Once done, the sentence and its embedding transformation will be sent through a second Kafka stream to a node that will run cosine similarity through the Trump BigQuery table and upload the sentence and its most similar Tweet in the Trump table in a separate, second table. The front-end application will query this table using the inputted sentence as a key until it has a result, showing this most similar Trump Tweet to the inputted sentence to the user.

We begin with a brief literature review of the big data technologies used for this project.

LITERATURE REVIEW

While there are not many projects similar in intention to this, there are plenty of big data technologies and literature to review before diving into the technicalities of this project's implementation. This includes choosing a cloud computing environment, messaging service, transformer, and database. These are each discussed in detail below.

I. Cloud Computing Virtual Environment

More than any other technology, an elastic cloud computing platform is essential to big data applications, providing a pay-as-you go, flexible system to run massive virtual machine instances on for a wide variety of uses. Amazon's **Amazon Elastic Compute Cloud** (EC2), Microsoft's **Azure**, and Google's **Google Compute Engine** are three major offerings for users to quickly launch these machines, and each have their own unique qualities. While Azure offers greater flexibility to machine size and customizations and Google Compute Engine allows for greater options regarding both local and ephemeral storage options [2], Amazon's EC2 is one of the most widely used cloud services, offering simple network and storage configurability. Paired with its direct integration with Amazon **S3**, which acts as external storage for larger files, Amazon's services will be used to host a virtual machine for the project, though any virtual machine should be able to replicate the bulk of this project's code with no issue.

II. Messaging Service

The core of the project will be implemented with a messaging service that can transport different blobs of data to different services and streams. This project will have two separate topics for data to be transported: one from the client to the transformer node, and one from the transformer node to the storage node (see *Building the Project* section for more details). Below I will briefly review some of the more popular big data messaging technologies to determine the one that best suits this project.

Amazon Kinesis Streams is a stream processing framework offered by Amazon Web Services, making its usage simpler and more popular for technologies already built using AWS resources. Kinesis Streams applications can put data into a stream for consumers to read as a data record, with the white paper *Streaming Data Solutions on AWS with Amazon Kinesis* citing times less than a second from writing to consuming data [6]. Data can be sent to Kinesis through a variety of languages, and Kinesis guarantees durability with automatic and configurable retry mechanisms in case of failure, optimized payload sizes to improve throughput, and synchronous and asynchronous write support. Since this is a managed platform within AWS, Kinesis does not allow for as much flexibility as other services might.

Azure Service Bus is a message broker set to decouple applications and services with asynchronous data messaging. Service Bus uses topics to send and receive data stored as JSON, XML, or text in binary format, with advanced features for triggers based on certain data conditions as well as delays and subscriptions expiring from a topic, giving it more flexibility and advanced features for a wide variety of projects compared to other messaging services [1]. However, Service Bus

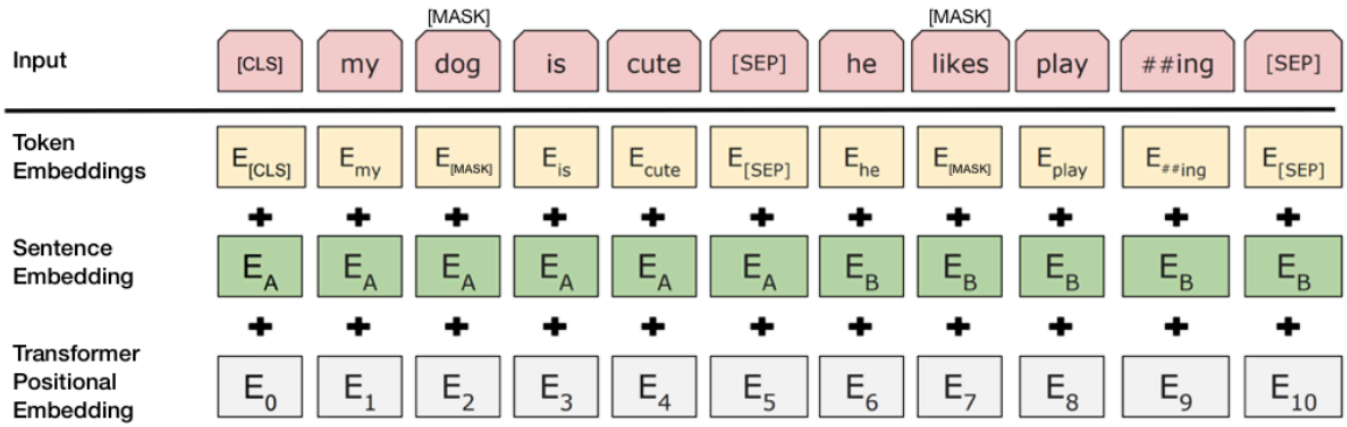

***Figure 1**: BERT architecture detailing an inputted sentence in the training pipeline [3].*

only supports client libraries for .NET, Java, and JMS, making it difficult to integrate with Python, the desired language for the project.

Originally developed by LinkedIn, **Apache Kafka** is another distributed messaging system with similar features to Kinesis and Service Bus in terms of scalability, perseverance of messages, and guaranteed ordering. Kafka is open source, making it incredibly flexible to use in a variety of different platforms, and guarantees at-least-once delivery in order to combat potential failures from system downtime or crashes, a safeguard that does not negatively affect the intent for this project, since data getting written to a database a number of times will not impact results [4].

With how lightweight Kafka is compared to other services such as Kinesis and Service Bus (as well as ones not discussed in this literature review), offering us full customizability for development, allowing for cheaper costs to run the project, Kafka will be the service used for publishing and consuming data in this project.

III. Transformer

The goal of the project is to take a sentence generated from the client application and find the most similar sentence in a database. While simple in theory, there are many ways we could accomplish this goal, such as with TF-IDF scores, a bag-of-words approach, etc. While these have been done in countless other projects, I decided to take a more experimental approach to representing sentences numerically, specifically using embeddings outputted from a **BERT** deep learning text model, the current state-of-the-art approach for natural language processing developed at Google. BERT distinguishes itself from other language models by coming pre-trained to predict tokens in the English language, making it able to generate deeply detailed sentence embeddings out of the box [2]. With a rich understanding of sentences, tokens, and vocabulary positioning, the BERT model is able to deeply understand language structure more than a traditional method such as TF-IDF is able to approximate.

For this project, we will be using a variant of a finely-tuned BERT model developed by the Ubiquitous Knowledge Processing Lab (UKP), which uses a Siamese network structure composed of a standard BERT model (with architecture shown in Figure 1) and a variant of BERT developed at the University of Washington entitled **RoBERTa** (which increases the data size and tunes hyperparameters of the original BERT model to develop a higher performance version of the model [5]) to generate Sentence Embeddings [7]. This allows a user to download the pre-trained model and input a sentence of variable length to generate a fixed numerical sentence embedding of length 768. We will use this output for all sentences, then use Euclidean distance on the embeddings to determine sentence similarity.

IV. Database

The final step of the project will rely on a NoSQL database that is able to handle large volume and velocity of data with low latency and high availability so our client application will always be able to get a response when querying the table. In this

project, our client-side application will be querying the database until the sentence sent in the initial Kafka stream has sent the data to the transformer node and then been uploaded to the database table with the most similar sentence. Since low latency is of the utmost importance, I have opted for a NewSQL database in order to get fast, in-memory SQL while still being able to handle large volumes of data. In addition, the SQL language will allow us to perform cosine similarity in the database, allowing us to determine sentence structure with ease.

While there are many different variants of the NewSQL database, this project opts to use **Google BigQuery**, an extension over Google's internal tool Dremel, that is built for "Google Speed" and used in a variety of Google tasks requiring incredibly low latency with the ease of the SQL language [8]. While fixed to the Google Cloud Platform system, reducing its overall flexibility and customizability, its simple web UI and variety of APIs in many languages makes this a safe choice of database.

BUILDING THE PROJECT

With our technologies selected, this section of the paper focuses on building the actual application, overcoming difficulties presented in its implementation, and discussing future areas for improvement.

I. Phase I: Data Extraction and Processing

To begin, I spun up a `t3a.xlarge` Ubuntu 18.04 Amazon Machine Image on EC2 with 30 gigabytes of storage. I installed Java, Kafka, and Python 3 on to the Linux environment and set up a GitHub repository to keep track of the code changes or additions made.

With this, I began the process of data extraction. I used the Trump Twitter Archive, an online tool tracking every tweet made by Donald Trump's Twitter handle (@realDonaldTrump) as well as other metadata associated with the tweet included favorite count, retweet count, date of creation, etc. This source was used over the standard Twitter API due to the Twitter API's limit of only allowing developers to pull the latest 3,000 tweets from any user. Since Donald Trump has tweeted nearly tenfold that number, we must seek an alternative data source.

I exported all tweets in the database to a JSON file and wrote a Python script `01_clean_tweets.py` that processes the data, including formatting the date of creation, removing retweets from the data (to only consider tweets originating from Donald Trump himself), removing hyperlinks and images in tweets, dropping duplicates, etc. With the data clean, we are able to run each Tweet through the UPK `SentenceTransformer` model to output a length-768 embedding for each Tweet. This is appended to a Pandas DataFrame and written back into a JSON file with a record orient and delimited by newlines so it can be consumed through the BigQuery API.

With a completely processed dataset, I wrote a second script, `02_data_to_bigquery.py` that connects to Google BigQuery and sends the JSON file of cleaned Trump tweets to an external table. A third script, `03_create_results_tables.py`, creates two additional BigQuery tables that will be empty until we build out the second phase of the project. One will be an intermediary table holding a user-inputted text blob and its associated BERT encoding, and a second table will store the results of the most similar Trump tweet and its metadata that our user application will be able to query.

In total, my most-recent data pull on December 3rd, 2019 generated 30,445 cleaned tweets with a BigQuery table size of 277.64 MB. While this is not nearly enough data to warrant a big-data database, all scripts are flexible and customizable to instead select Tweets from another user, topics, etc. as long as they can be pulled into a JSON file. Paired with the ability to stream Tweets into the database in real-time, the potential for the dataset to grow beyond the realms of a standard relational database management system is likely, and if so, will still be supported with the technologies used in this project.

II. Phase II: Building the application

With our data cleaned and imported into BigQuery, we are finally able to begin building out the application that a user can interact with. For the purposes of a clearer explanation, we will assume four worker 'nodes' composing this project (even though each node lives on the single Ubuntu AMI). I

will delve into greater detail into each node's responsibilities below.

Node 1 is in charge of running Kafka. It starts the Zookeeper server (to keep track of Kafka partitions, Kafka node status, and ensure high availability, among other responsibilities) as well as the Kafka service, ensuring they are both running and accessible for the other nodes to publish and consume.

Node 2 displays the prompt for a user, allowing them to input a sentence for comparison. Node 2 publishes this sentence to a Kafka stream, waits until it is sent successfully, and then begins to query the BigQuery results table until it is populated with a row with text equal to the user-inputted blob.

Node 3 consumes the Kafka stream which Node 2 publishes to, and runs the user-generated sentence through the UPK `SentenceTransformer` model to generate a fixed-length embedding. Node 3 sends the original sentence and its embedding to an intermediary Google BigQuery table holding the text and embedding, and then publishes this data to a second Kafka stream.

Node 4 finishes the process by consuming the second Kafka stream which Node 3 publishes to. Node 4 organizes a BigQuery query to run Euclidean distance with the user-generated sentence embeddings and all embeddings in the Trump Tweet BigQuery table, selects the least different entry (`min(distance)`), and publishes this entry to and its metadata to a final BigQuery results table. Once published, Node 2 will pull this data down and display it to the user.

A diagram of this architecture is seen in Figure 2.

While this approach may seem like a complex solution to a simple problem, the goal of this project is to develop a framework for a business utilizing a pipeline of transformations of entries published to a variety of Kafka streams. Some data science teams build models that ingest embeddings resulting from different stages of a model (for example, one application may require a final prediction of a PyTorch model as input; another may take the final *layer* embedding of a PyTorch model as input). For these use-cases, it may be useful to have multiple Kafka streams with many transformation layers active at once. With its scalable

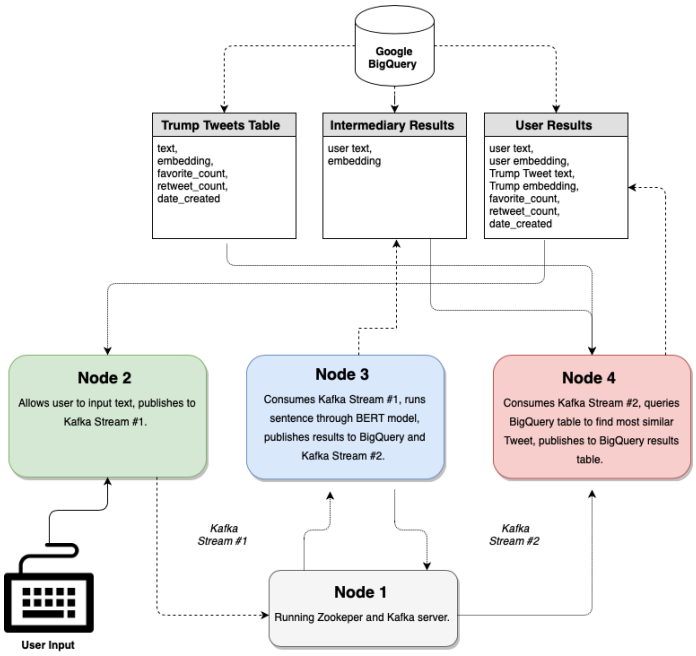

***Figure 2**: Architecture between the four nodes, Kafka streams, and BigQuery tables.*

behaviors, this project provides a general solution to this use-case.

III. Optimizations

While I was able to get the above project outline working, there were some problems that immediately arose that prohibited the usefulness of this specific application.

The first problem came from the execution time of the Euclidean distance BigQuery SQL query. For the original embedding length of 768, the average execution time was 48.9 seconds – meaning from the time the user inputted a sentence to the time they got the most similar Tweet back, nearly a minute would have passed. This latency was unacceptable, meaning I needed a form of dimensionality reduction applied to each of the embeddings in order to speed up the SQL calculation.

While many optimal solutions for this problem exist (see the *Future Improvements* section), I decided to use Principal Component Analysis (PCA), a statistical procedure using a series of orthogonal transformations to reduce the dimensionality of a set of vectors to its more important and highest variability components, to reduce the embedding size of the BERT model. I tried several different component sizes and

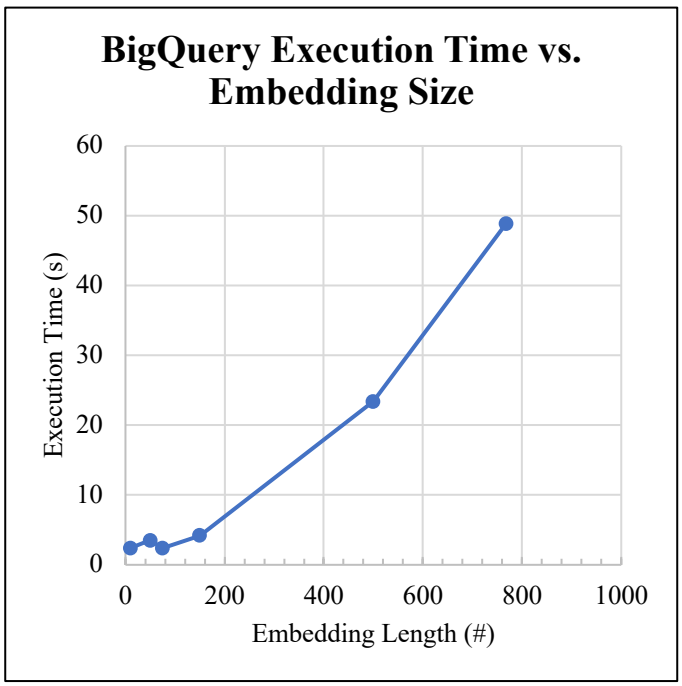

***Figure 3**: Results of the experiment taking the average times between the BigQuery Euclidean distance query execution time and embedding sizes outputted from the BERT model.*

compared execution times for BigQuery. I had to make a decision in a tradeoff between application execution time and information loss, as the bigger the embedding size, the longer the execution would take, but the more information we would retain, and vice versa. The results of this study are shown in Figure 3.

In the end, I settled on an embedding size of length 150, with an average Euclidean distance query time of 4.2 seconds. This seemed like an acceptable tradeoff between capturing a majority of vector variance and a "real-time" result from the BigQuery Euclidean distance command.

Additional code to run the PCA operation and save the transformer is appended to `01_clean_tweets.py`.

The second problem came with a flaw in the initial four-node setup, particularly with Nodes 3 and 4. While Node 1 launches Kafka, which itself is a big data technology, and Node 2 can be replicated across many different user machines without issue, Node 3 and 4 work with data in a one-at-a-time, synchronous manner. If ten sentences from ten users come into Nodes 3 and 4, they will all be processed individually, making this application *not* big-data compatible.

While there are many data streaming solutions that would be able to connect to the Kafka stream to run the data through the model, I decided to use the simplest and most flexible method of data parallelism possible – asynchronous functions. By rewriting the logic of Nodes 3 and 4 to do the work asynchronously using Python's `threading` library, Kafka messages and queries can be worked on many-at-a-time efficiently (see *Future Improvements* for limitations with this approach). This is the most extensible solution for allowing users to get the benefits of parallelism, efficient computation, and the entirety of the full Python language and libraries to use for processing each Kafka message.

These changes have been implemented in `02_kafka_1_to_kafka_2.py` as well as `03_kafka_2_to_bigquery.py`. Figure 4 shows an example output from Node 2 when running the application with these two optimizations in place.

IV. Future Improvements

While this project certainly works, it is far from perfect. Below, I list several opportunities for

```
(base) ubuntu@ip-172-31-0-244:~/python-kafka-bert-kafka-bigquery-trump/python_kafka_bert_kafka_bigquery_trump/02_run_the_app$ python 01_user_to_kafka_1.py
Type a sentence (or 'q' to quit): Sorry haters but you cannot beat my gramatical correct Tweets.
Working on it...

You entered "Sorry haters but you cannot beat my gramatical correct Tweets.".
We ran this through a BERT model with PCA applied to determine the most similar Trump tweet:

"The failing @HuffingtonPost and dopey @ariannahuff are writing so much false junk about me-they just can't get enough!  BE CAREFUL."

That wouldn't have been the most popular tweet!
The original tweet got 64 favorites and 52 retweets.
The original tweet was published on 08/13/2013 at 01:24:57 AM.
Your text scored a similarity score of 0.7209778821121845 (the lower, the more similar).

---------

Type a sentence (or 'q' to quit): q
I hope you learned how Trump you are!
(base) ubuntu@ip-172-31-0-244:~/python-kafka-bert-kafka-bigquery-trump/python_kafka_bert_kafka_bigquery_trump/02_run_the_app$
```

***Figure 3**: Output from running the user application in Node 2.*

future improvements to the core of the project I hope to continue after the course's duration as a side project.

- **Better method of dimensionality reduction**. While principal component analysis certainly gets the job done, it reduces much of the embedding's meaning that is generated from the modified BERT model. A *better* method of dimensionality reduction would be to add an additional dense layer to the BERT model with an output size being the desired embedding length. Not only would this effectively reduce the dimensionality of the embeddings, but by training a final layer of the model, the embeddings would hold more Trump-specific meaning, allowing for comparison between embeddings to be even more effective.
- **Better form of aychnchrnous parallelization**. While using Python's built in ability to make a function asychnronous offers an unparalleled level of flexibility, it is certainly not the best for a high volume of data. While using a technology like Spark Streaming would reduce the overall flexibility of the approach, resiliant distributed datasets that make up the core of Spark not only allow for great efficiency and parallelism, but a level of fault-tolerance vanilla Python will not be able to achieve. If this application were to be expanded in scale, this would be a necessary requirement.
- **Better selection of NewSQL database, or a new storage technology altogether**. Again, Google's BigQuery gets the job done for this use-case, but its latencies are very high for a relatively small dataset with a small embedding length (even length 10 embeddings took well over 2 seconds to execute the Euclidean distance SQL query). If we wanted to make this application faster, we might consider another database that offers the power of SQL and the ability to handle large volumes of data while being specifically designed for execution speed of queries. Another approach would be to load all of the text and embeddings into a tool like Elasticsearch that has native cosine similarity and Euclidean distance functions built in, with low latencies being one of its major features. While this will require some retooling of the application, its effects would be certainly welcome.
- **Expand the dataset!** While Donald Trump tweets are certainly entertaining, they are not the only source we can pull! Whether it be Tweets, movie reviews, Facebook posts, Instagram captions, etc., there exists an API to pull a massive dataset and run through the BERT model. Regardless of how short or long the sentence is, the BERT model will output a fixed length embedding, meaning that we can run the same Euclidean distance command to find similar source material. With *more* data, we can derive even *more* interesting results.
- **Use a different model**. There are plenty of natural language processing models meant to derive a fixed length embedding from a variable length sentence, and even BERT has several pre-trained variants with different hyperparameters and trained using different data, each built with a specific purpose in mind. It would be interesting to note differences in tweet similarity responses when using a different base model for analysis.

V. Lessons learned

I set out on this project to learn how to use Kafka in a real-world setting, and get a bit more experience with PyTorch models, Python programming, and AWS EC2 along the way. Reflecting on this project, I feel successful in getting exposure in all of this and more.

The biggest lesson I learned from this project would be the importance of organization. Writing code that is decomposable, well-documented, and properly organized is incredibly important for working on a project iteratively. When I noticed speeds were slower than intended, or the model used was wrong, my code organization saved me hours of debugging and rewriting code, making changes very simple.

Another lesson learned is the importance of slowing down to read documentation, blog posts, and examples. I have never worked with Kafka or the Kafka Python client library before this project, and getting the exact environment set up to run properly on my virtual environment was incredibly difficult at first. There were many times I found myself

spending hours at a time reading online guides and documentation to understand the exact reproducible setup to get everything in place. In the beginning of my work on this project, I rushed through this documentation, only to hit a roadblock a bit later on that, sure enough, was covered in the literature I so quickly skipped. By the end, I learned to read documentation thoroughly and found I saved a great chunk of time as my work progressed.

Lastly, I learned how many ridiculous tweets Donald Trump has made! Reading through all of them has been an emotional roller coaster, and if anything, it is impressive a single human is able to generate over 30,000 somewhat-unique thoughts of 140 characters or less.

Conclusion

This project introduces a flexible and extensible framework to connect multiple Kafka streams together to asynchronously transform data in a variable number of intermediary steps. This pipeline has the potential to handle a large volume of data with a wide array of potential use cases.

All of the code and documentation for this project is available on my GitHub profile at https://github.com/nathancooperjones/python_kafka_bert_kafka_bigquery_trump. The code is broken down into three sections: `00_environment_prep`, `01_data_prep`, and `02_run_the_app`. Each directory contains its own `README.md` file with verbose instructions on how to prepare and run the code on your own virtual machine. With such a low storage and computing requirement to replicate this exact project, the costs are incredibly low to run, making this project even more friendly for those looking to use, improve upon, and extend a Python-Kafka-BERT-Kafka-BigQuery pipeline such as this.

References

[1] Axisc. “Azure Service Bus Messaging Overview.” Azure Service Bus Messaging Overview | Microsoft Docs, Microsoft Azure, 20 Nov. 201AD, docs.microsoft.com/en-us/azure/service-bus-messaging/service-bus-messaging-overview.

[2] Cloudyn. “AWS vs. Azure vs. GCP: Determining Your Optimal Mix of Clouds.” *Cloudyn White Papers* (2016).

[3] Devlin, Jacob, et al. "Bert: Pre-training of deep bidirectional transformers for language understanding." *arXiv preprint arXiv:1810.04805* (2018).

[4] Kreps, Jay, Neha Narkhede, and Jun Rao. "Kafka: A distributed messaging system for log processing." *Proceedings of the NetDB*. 2011.

[5] Liu, Yinhan, et al. "Roberta: A robustly optimized bert pretraining approach." *arXiv preprint arXiv:1907.11692* (2019).

[6] MacInnis, Allan & Matrubhutam, Chander. “Streaming Data Solutions on AWS with Amazon Kinesis.” *Amazon Web Services* (2017).

[7] Reimers, Nils, and Iryna Gurevych. "Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks." *arXiv preprint arXiv:1908.10084* (2019).

[8] Sato, Kazunori. “An Inside Look at Google BigQuery.” *Google White Papers* (2012).

[9] Wingerath, Wolfram, et al. "Real-time stream processing for Big Data." *it-Information Technology* 58.4 (2016): 186-194.

[10] Zaharia, Matei, et al. "Discretized streams: Fault-tolerant streaming computation at scale." *Proceedings of the twenty-fourth ACM symposium on operating systems principles*. ACM, 2013.