Sam Colelli

# Big Data Wrangling with Google Books Ngrams

The following document is a summary and outline of the findings and steps taken to fetch, store, analyze, and visualize the large Google Ngrams dataset.

## Summary

Google Ngrams is an open-source dataset that represents content found in all books that span Google Books – roughly 4% of all books ever published. The data is summarized via 5 columns:

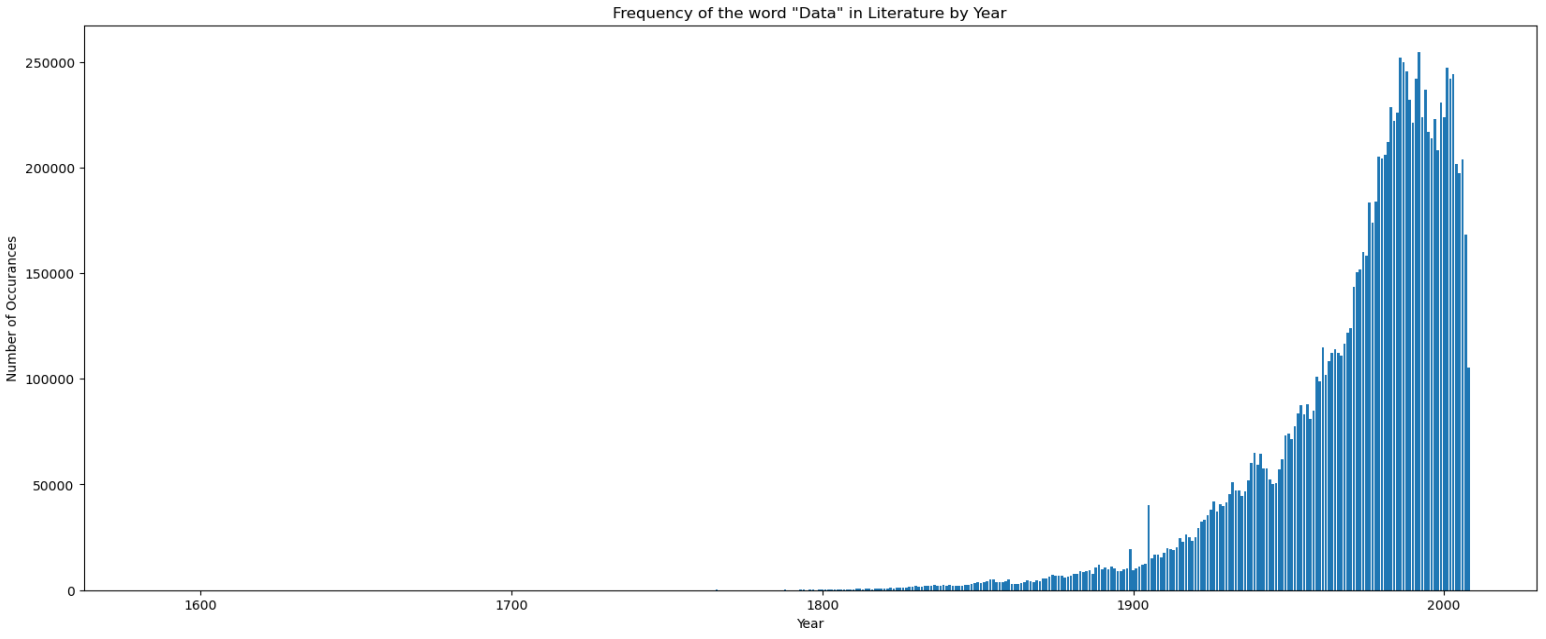
1. **Tokens**: Keywords or phrases
2. **Year**: The year that each keyword or phrase first appeared in each published book
3. **Frequency**: The number of times a token appeared each year
4. **Pages**: The number of pages the token appeared in each book in that year
5. **Books**: The total number of books that each token and year combination represent

The entire dataset is comprised of these 5 columns and 261,823,225 rows. To better visualize this data, it was scaled down by identifying only one token of interest: the word “data”.

When filtering the dataset for only “data”, a word that is seemingly only fit for modern times, was seen in published books in 316 different years (316 rows, each representing a different year). The earliest year it was written in a publication was in 1584, where in a single book, it was referred to 16 times.

However, the early years still pale in comparison to today’s published works. The visualization to the right represents the frequency of the occurrence “data” in all publications that can be found on Google Books.

Due to the sheer size of the Google Ngrams dataset, traditional means of data wrangling and EDA could not be attained through computing resources on local machines.



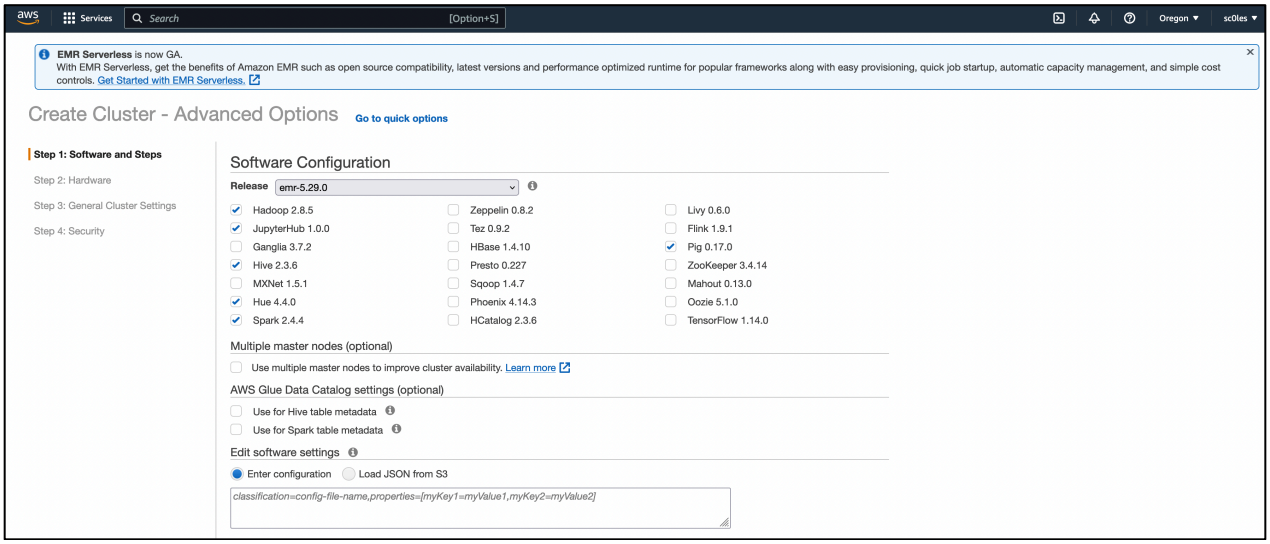

The outline below is a step-by-step representation of how to process and analyze this dataset via AWS’s EMR with Hadoop & Spark.

Sam Colelli

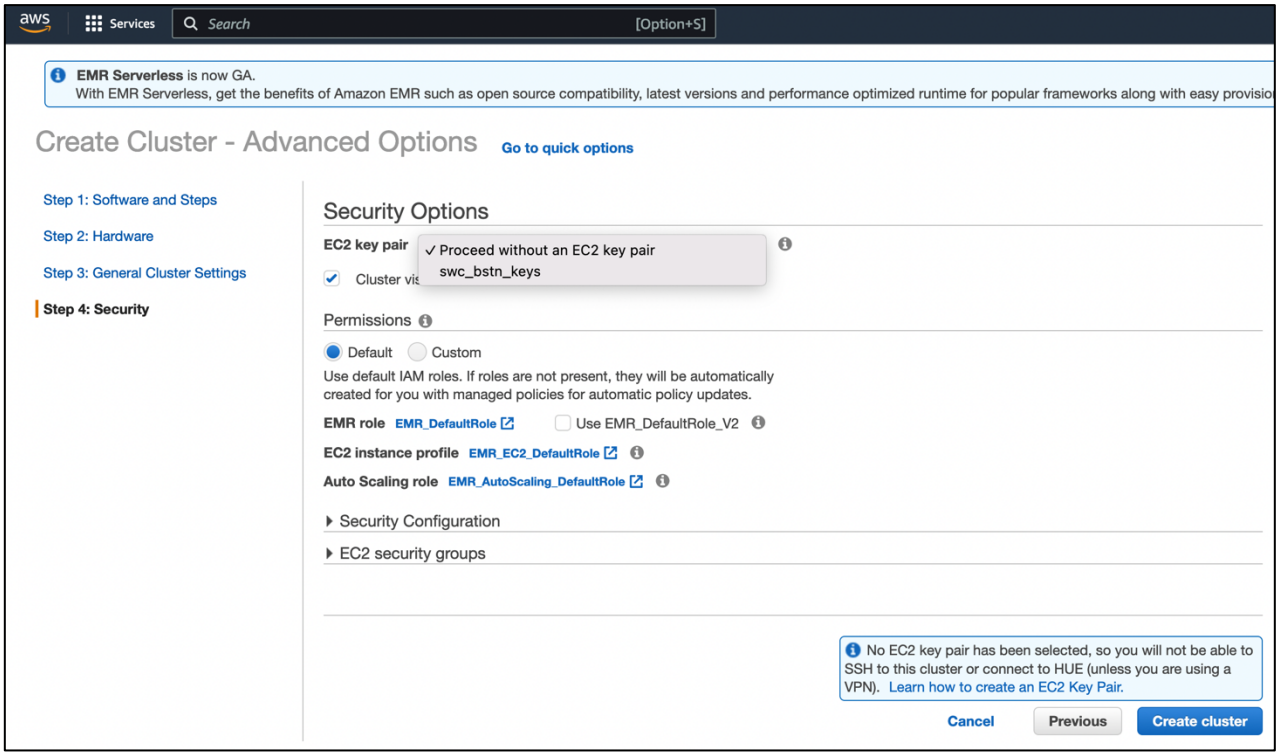
# Spinning Up an EMR Cluster

The first step to fetching and storing the Google Ngrams dataset is to create a storage cluster in AWS. For this exercise, we'll leverage EMR since it automatically spins up the needed EC2 instances needed we'll need.

Since we'll be leveraging Hadoop, Spark, Hive, and Jupyterhub for proceeding steps, we'll need to be sure to include these as well. The screenshot below represents the needed packages to include within your cluster.

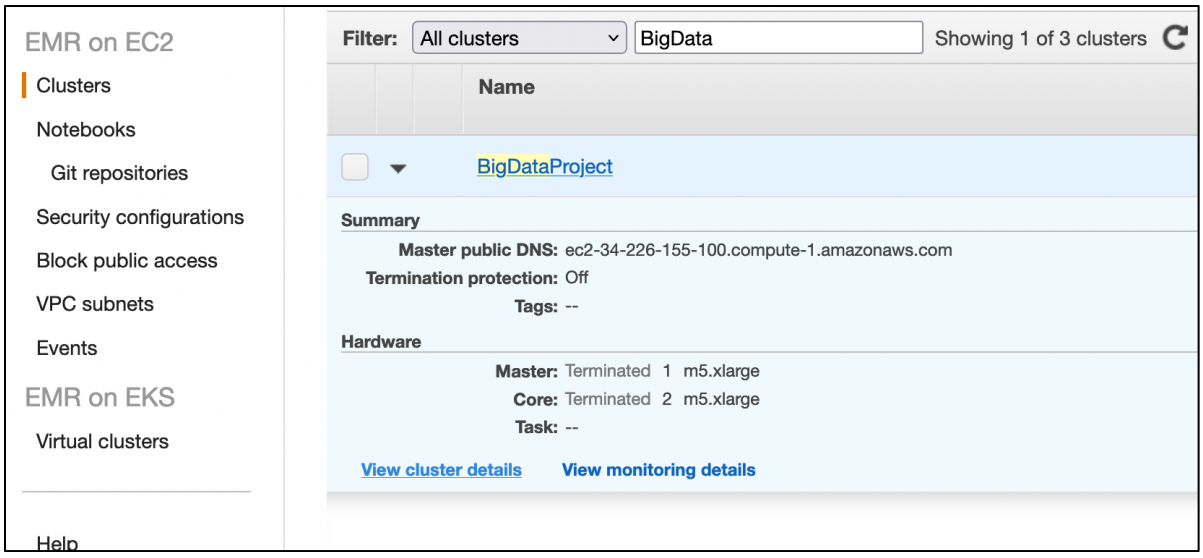



Outside of Software Configuration, the other thing to ensure is the use of an EC2 key pair, which will be needed to SSH into the cluster. If you do not have a .pem file containing a key pair, you'll need to create one.

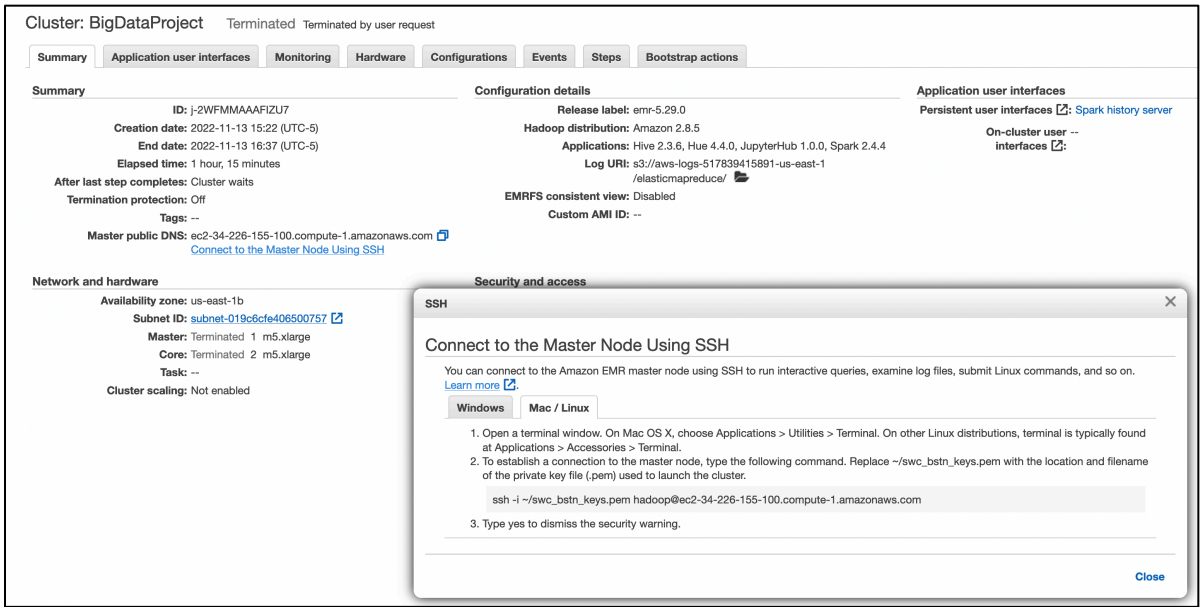


# Connecting to Head Node via SSH

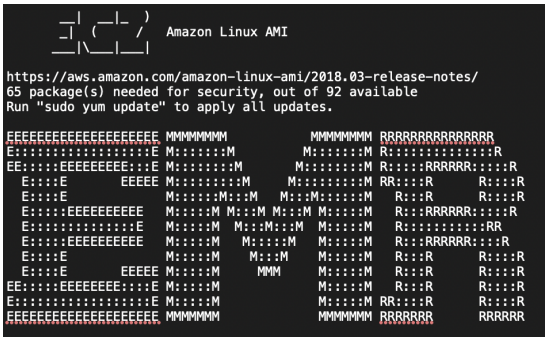
Once the cluster is created, the next step is to access the cluster via SSH. To do so, navigate to *View Cluster Details* within the *EMR on EC2* navigation pane.




Within the Summary tab, you'll find a pathway called Connect to the Master Node Using SSH. When you click on this pathway, it will provide you with a bash script to connect to the node via the terminal. This is where the EC2 key pair will come into play and allow you to connect to the head node via your local machine.




If connected successfully, you should see the following EMR logo in your terminal window:



## Fetch Google Ngrams via S3 and into HDFS

The next step is to fetch the dataset and store it into the newly created cluster. There are a number of ways to fetch data within S3 buckets, but this outline will cover the most straight-forward approach.

`hadoop distcp` is a function that copies files from one data source and directly stores it within the specified file system. The bash script needed is below.

```
hadoop distcp s3://brainstation-dsft/eng_1M_1gram.csv /user/hadoop/eng_1M_1gram
```

To expand on the above script further:

- `hadoop distcp` copies files from one source and stores it within the specified file system
- `s3://brainstation-dsft…` is the S3 link to the Google Ngrams file
- `/user/hadoop/eng_1M_1gram` is the filepath we want to copy this data to

To ensure that the file has been properly copied to our file system, we can run `hadoop fs -ls` to return the files that are currently stored within our hadoop cluster.

```
[hadoop@ip-172-31-85-183 ~]$ hadoop fs -ls
Found 1 items
-rw-r--r--   1 hadoop hadoop 5292105197 2022-11-13 20:33 eng_1M_1gram
```

## Data Exploration with PySpark

The next step is to do some light data exploration using PySpark on the newly acquired dataset. While you can spin up a Jupyterhub Notebook in the EMR GUI, performing said analysis directly in the terminal proved to be frictionless.

To access PySpark, simply write pyspark in the terminal. If the configuration was correctly installed when first creating the cluster, you should see the following Spark logo:


While fairly similar to Python, PySpark does have some notable differences in its syntax.
**To see the full code, please see the .txt files that were attached with this report.**

### Create PySpark DataFrame

Some functions that were leveraged for this step:

- `spark.read.csv` is how PySpark reads in CSV files, similar to `pd.read_csv`. For this step, it is important to specify within the function `header = True`

### Describe DataFrame using PySpark

- `df.show()` is used to display the dataframe in a table-legible format
- `df.printSchema` is used to see column names and dtypes, similar to `pd.info()`
- `df.withColumn` is used to change the datatype of a column. As seen in the code, we do this for year, frequency, pages, and books

### Create Filtered DataFrame Using Spark SQL

An interesting aspect of Spark is the ability to easily leverage SQL for RDBMS-related data exploration. To do this, you'll first need to create a temporary view to reference, similar to WITH in standard MySQL.

- `df.createOrReplaceTempView("data_ngram")`

Once the view is created, Spark can process generic SQL syntax to filter a DataFrame. For this exercise, we selected all columns where the token was equal to "data"

- `spark.sql("SELECT * FROM data_ngram WHERE token = 'data'")`

### Write Filtered DataFrame Back to HDFS

Now that data exploration is complete, the next step is to write the filtered DataFrame back to the hadoop cluster. Very similar to pd.write_csv, the PySpark syntax is:

- `df.write.csv("/user/hadoop/data_ngram", header = True)`

To exit PySpark, simply type `quit()` into the terminal after you've written the CSV to the file system.

## Merge Contents of Directory & Send to S3

The output of the CSV file from PySpark is saved as a directory within the file path that was specified in the function above. We'll need to merge all of the files within this directory into a single file, and then send this file into a storage bucket within S3.

The first step to this is to leverage the `hadoop fs –getmerge` function, which will merge 1 or more source files into a specified destination file. Since we are merging all files within a specified directory, we only need to specify the file path of the directory, as opposed to each individual file within the directory.

```
hadoop fs –getmerge /user/hadoop/data_ngram /home/hadoop/final_ngram.csv
```

To send this file to an S3 bucket, we'll simply need to denote the S3 bucket file link and the new filepath we wish to add to the bucket. The script needed is the following:

```
sudo aws s3 cp /home/hadoop/final_ngram.csv s3://scoles-bstn-bucket/final_ngram.csv
```

- `sudo` may not be needed for this but was used just in case. sudo allows you to run a program of another user, separate from the application or instance you are currently operating in. I was unsure if aws s3 was available in the hadoop instance, so I used sudo just in case.
- `aws s3 cp` is the function needed to copy file to an S3 bucket. You'll need to specify the file you wish to copy and the location you wish to copy the file to.

## Fetch & Visualize the DataFrame Using Pandas

**Please refer to the attached Jupyter Notebook for the remaining part of the code exercise.**

**But continue below for Hadoop vs Spark comparisons!**

Sam Colelli

# Hadoop vs Spark

Hadoop is an open-source filesystem for large datasets, and was one of the first utility softwares that addressed the need for distributed file systems. The core components of Hadoop are HDFS, which splits data processing tasks across master and slave nodes (as seen in our EC2 instances spun up from our EMR Hadoop cluster creation), MapReduce, which is responsible for processing the data via Map and Reduce phases, and YARN, which acts as a resource orchestrator for computing allocation.

Spark, also open source, was created as an enhancement of Hadoop (specifically MapReduce) for a number of specific workflows. Spark is advantageous to use with relatively smaller datasets since it processes and stores data with machine memory, as opposed to hard disk, which can drastically increase speed and reduce latency. Spark has been widely adopted within ML development and applications due to its MLlib, which is a set of ML algorithms with scaling tools.

Nevertheless, Hadoop still has its advantages within certain use cases. Since it relies on disk, as opposed to memory, data processing and storage costs can be significantly lower. That cost differentiation makes Hadoop better suited for batch processing that do not require high frequency SLAs. However, for real-time data processing, Spark's in-memory processing is often worth the increased cost.