EVENT-DRIVEN SERVERLESS APPLICATION USING GCP ML

CSCI 5410 – Assignment 4 – Part B

Dhrumil Amish Shah (B00857606)
dh416386@dal.ca

Event-Driven Serverless Application Using GCP ML

Google Cloud Platform (GCP) is a cloud platform provided by Google for developers to build scalable, highly available, and secured applications. It allows developers to integrate a variety of services into their applications with ease. GCP offers compute services for computations and processing, serverless services for easy deployment, storage services, networking, CI/CD, Machine Learning, Artificial Intelligence and many more. For this assignment, I have developed an event-driven serverless application using various GCP services, namely Cloud Storage, Cloud Functions, and Vertex AI.

This application first creates a storage bucket called **sourcedatab00xxxxxx** in Google Cloud Storage [1] and uploads 299 training files starting from 001.txt to 299.txt in the created bucket. For this, I wrote the code in JAVA programming language using IntelliJ IDE. Once all the files are uploaded to Cloud Storage, a Cloud Function [2] called **generateVector** is executed. It reads all the files one at a time, filters out special characters and extra spaces. Then, it removes stop words from the file content and finally, it computes Levenshtein distance [3] for the file content. The code for this is also written in the JAVA programming language and is deployed on Google Cloud Function. After all the files are processed, the final output is stored in a CSV file called **trainVector.csv**. The **trainVector.csv** file consists of three columns - the current word, the next word, and the Levenshtein distance between these words. The CSV file is stored in a separate bucket called **traindatab00xxxxxx** in Cloud Storage. This CSV file is used by the K-means algorithm [4] for clustering. The algorithm is trained using the Levenshtein distance column. For this part of the assignment, the Vertex AI [5] service by GCP is used. It provides Jupyter Notebook and Python support for writing the K-means algorithm.

After the algorithm is trained, test files from 300.txt to 401.txt are uploaded to the same source bucket - **sourcedatab00xxxxxx** in Cloud Storage. Again, the **generateVector** Cloud Function is executed. Similar to the training steps, a **testVector.csv** file is generated with three columns containing the output computed for files 300.txt to 401.txt. This CSV file is stored in a new bucket called **testdatab00xxxxxx**. The trained K-means model gets the **testVector.csv** from the **testdatab00xxxxxx** bucket. The output generated is displayed in terms of end results which consists of centroids, cluster numbers, and outliers.

Prerequisite Steps

Figures 1 to 11 display the initial steps that I performed to create this serverless application. These steps are required for building this application and are performed on Google Cloud Platform (GCP).

Step 1: Initially, I created a new project on GCP and named it **CSCI5410-A4-PartB**. **Figure 1** displays the creation of project **CSCI5410-A4-PartB** on GCP and **Figure 2** displays the dashboard of the newly created project along with the notifications menu opened.

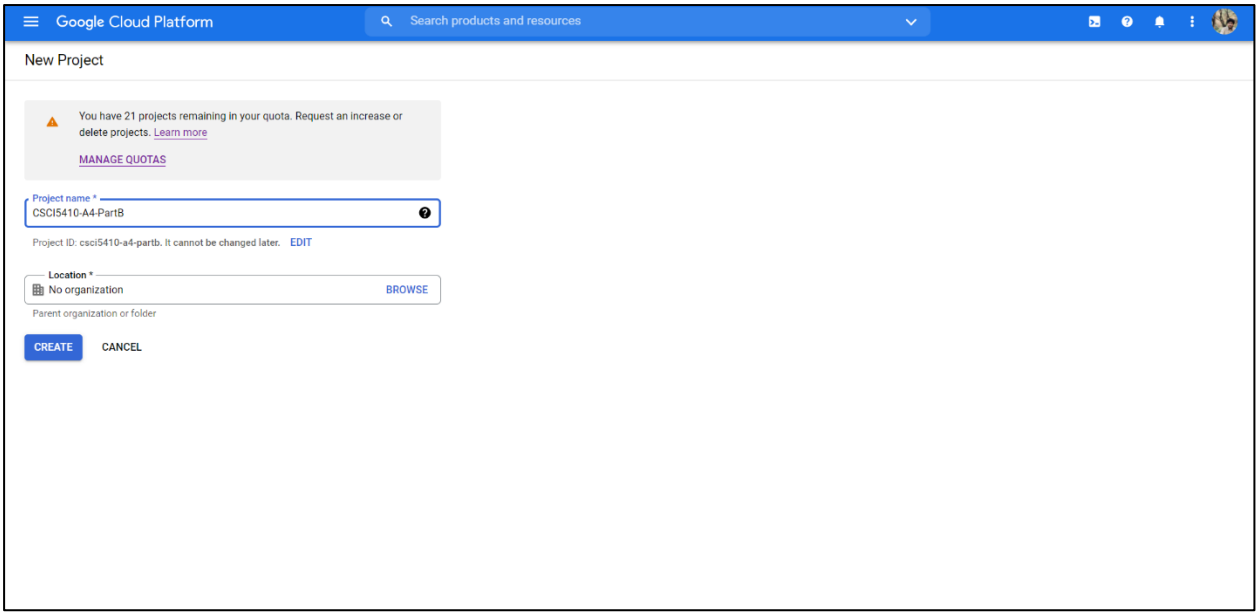


Figure 1 - Creation of CSCI5410-A4-PartB project in Google Cloud Platform (GCP) [6]

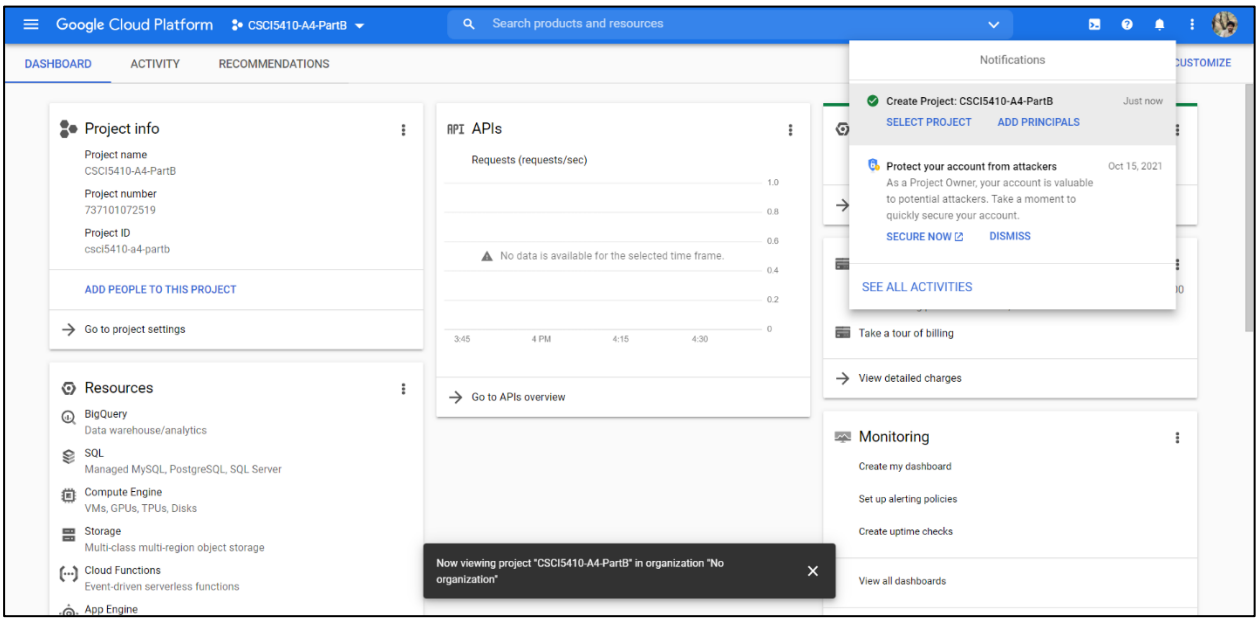


Figure 2 - Project CSCI5410-A4-PartB created on Google Cloud Platform (GCP) [6]

Step 2: Once the set-up of project **CSCI5410-A4-PartB** is completed, I created a service account inside the project and named it **CSCI5410-A4-PartB-SA**. This service account is required to connect our application with various Google cloud services. **Figures 3 to 5** display the steps performed to create the **CSCI5410-A4-PartB-SA** service account inside the project. **Figure 6** displays a list of all service accounts inside the project **CSCI5410-A4-PartB** with the newly created service account.

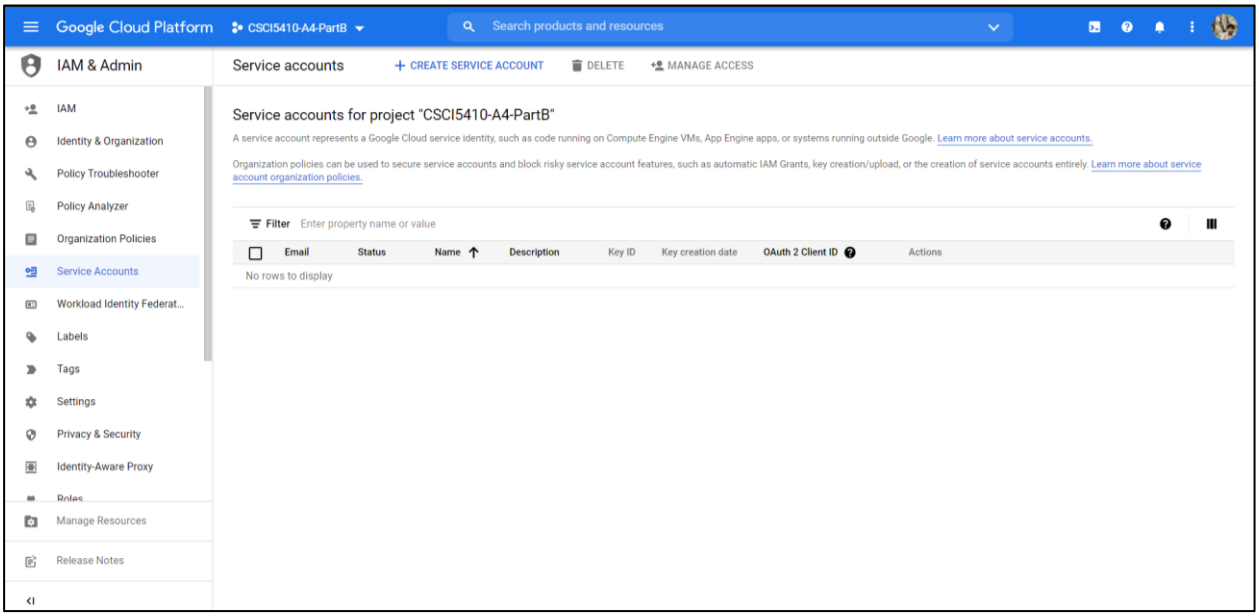


Figure 3 - Service account creation for project CSCI5410-A4-PartB [6]

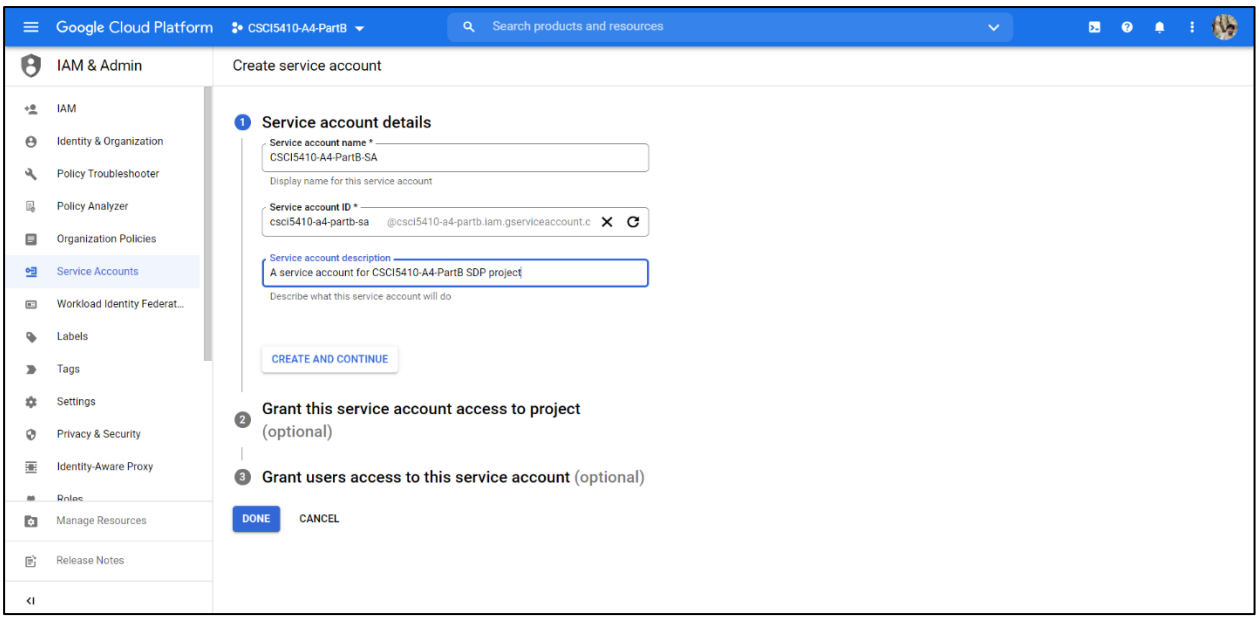


Figure 4 - Service account creation for project CSCI5410-A4-PartB (contd.) [6]

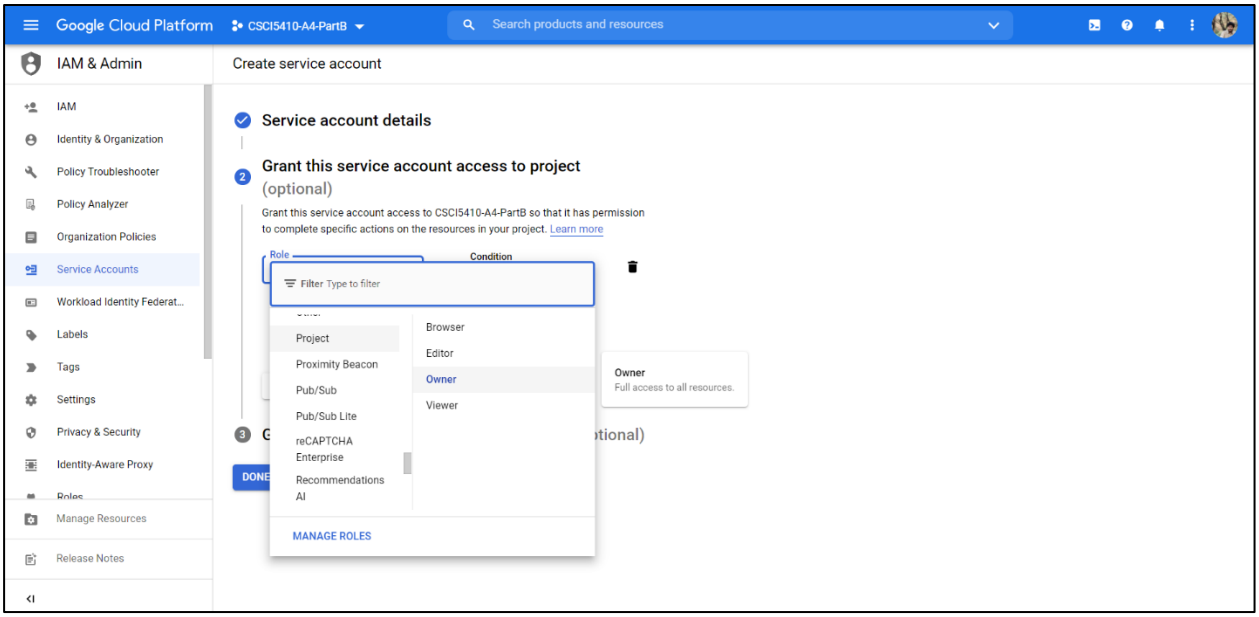


Figure 5 - Service account creation for project CSCI5410-A4-PartB (contd.) [6]

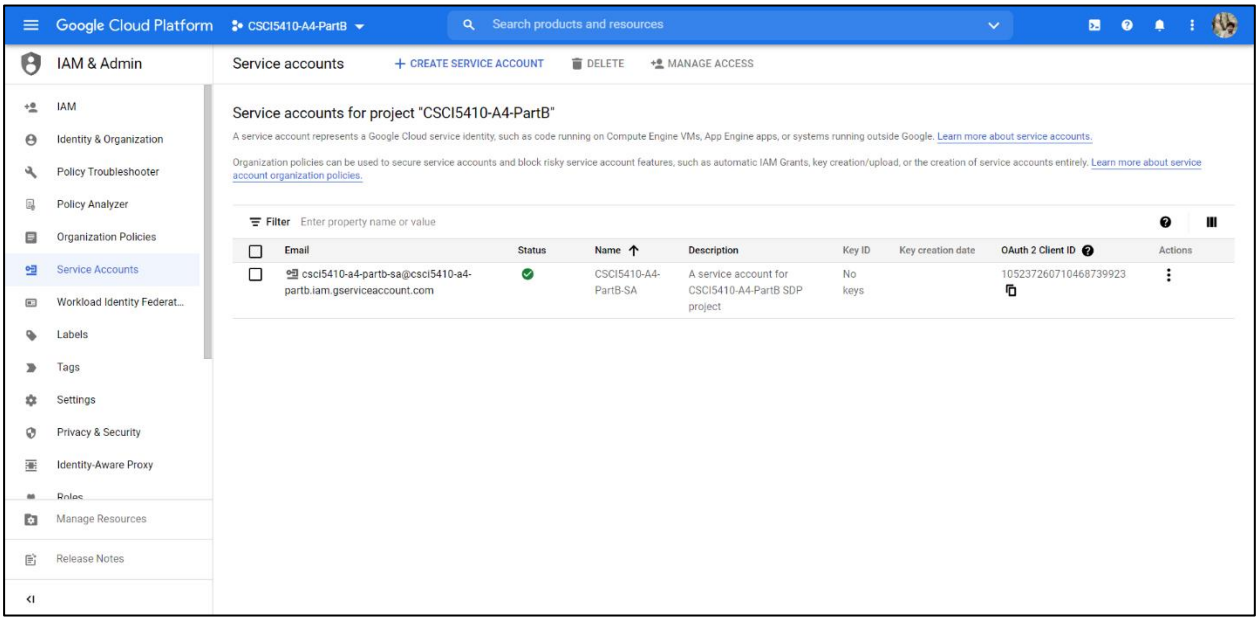


Figure 6 - Service account created for project CSCI5410-A4-PartB [6]

Step 3: Once the set-up of service account **CSCI5410-A4-PartB-SA** is completed, I created a key pair using the **ADD KEY** option. It generates a JSON file that contains below information:

```
{
  "type": "type",
  "project_id": "project_id",
  "private_key_id": "private_key_id",
  "private_key": "private_key"
  "client_email": "client_email",
  "client_id": "client_id",
  "auth_uri": "auth_uri",
```

```
  "token_uri": "token_uri",
  "auth_provider_x509_cert_url": "auth_provider_x509_cert_url",
  "client_x509_cert_url": "client_x509_cert_url"
}
```


Figure 7 displays the key pair creation inside the service account **CSCI5410-A4-PartB-SA** using the ADD KEY option and **figure 8** displays the key pair created in the same service account.

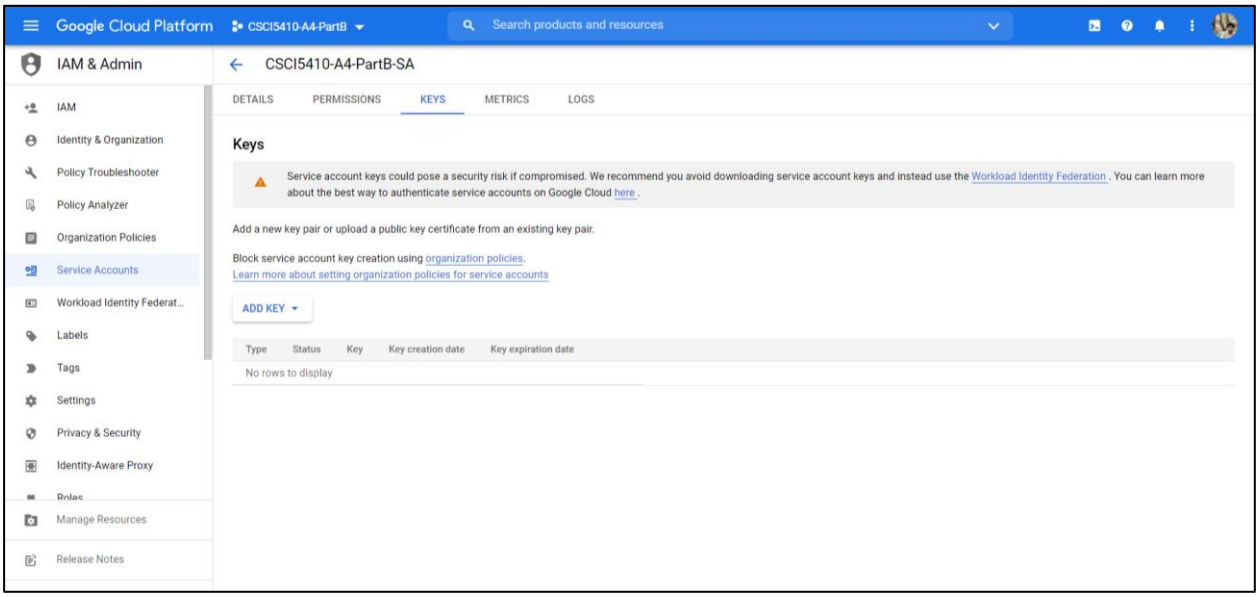


Figure 7 – Key pair creation for service account CSCI5410-A4-PartB-SA using ADD KEY [6]

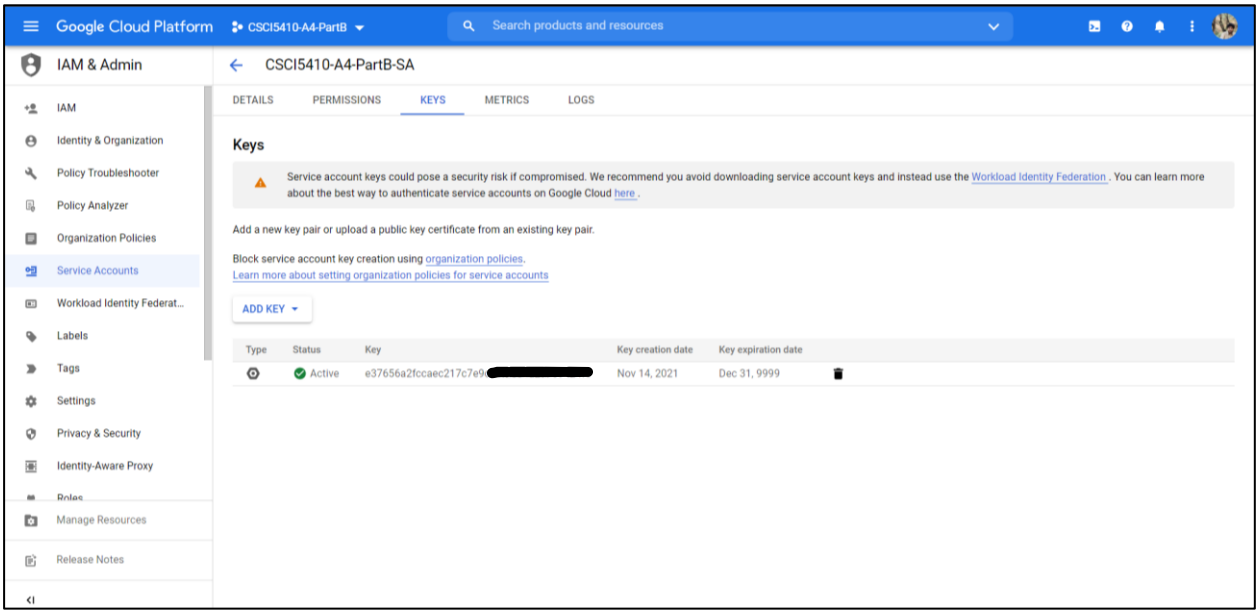


Figure 8 - Key pair created for service account CSCI5410-A4-PartB-SA [6]

Step 4: I enabled two APIs in account **CSCI5410-A4-PartB** namely Cloud Functions API - cloudfunctions.googleapis.com and Google Cloud Build API - cloudbuild.googleapis.com. **Figure 9** displays enabled Cloud Functions API and **figure 10** displays enabled Cloud Build API.

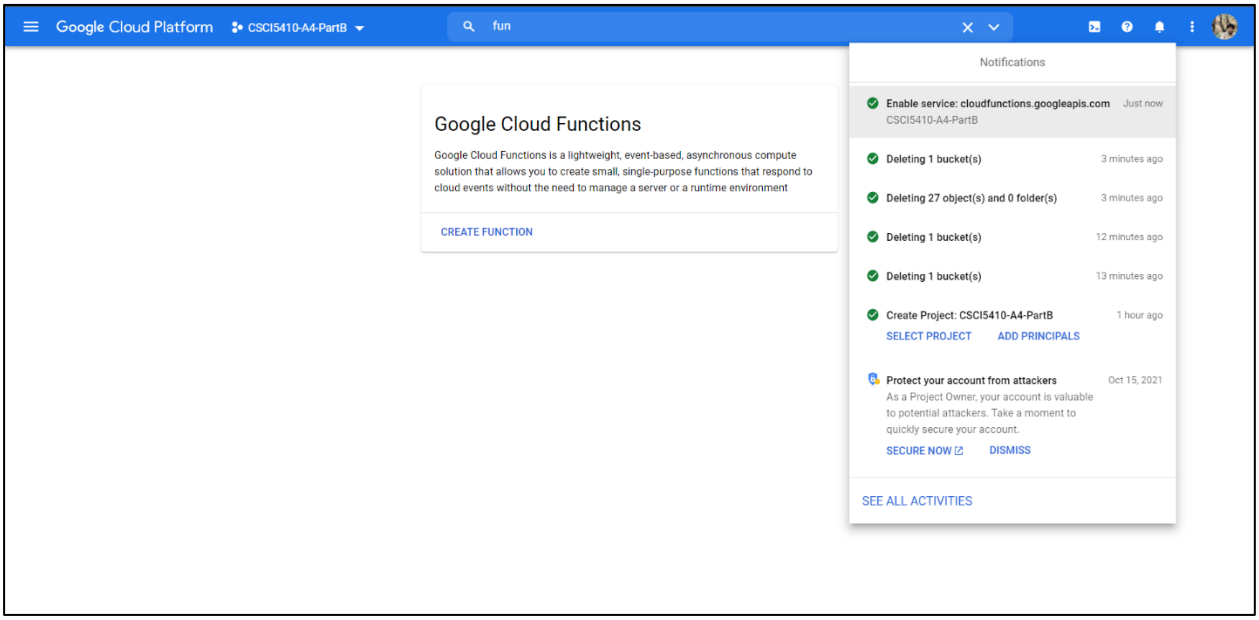


Figure 9 - Cloud function API enabled in account CSCI5410-A4-PartB [2]

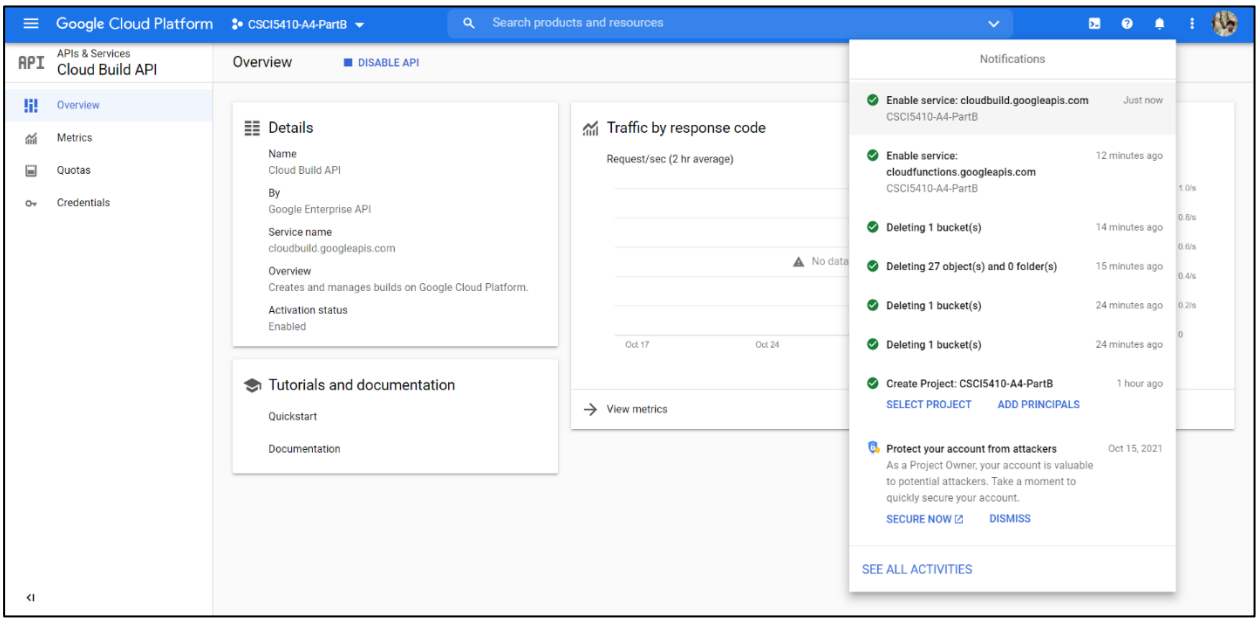


Figure 10 - Cloud Build API enabled in account CSCI5410-A4-PartB [6]

a) Create your 1st storage bucket SourceDataB00xxxxxx and upload the files (from 001 to 299) given in the Train folder. You need to write a script or use the SDK to upload the files on the bucket.

Figure 11 displays the initial state of Cloud Storage with no buckets in it.

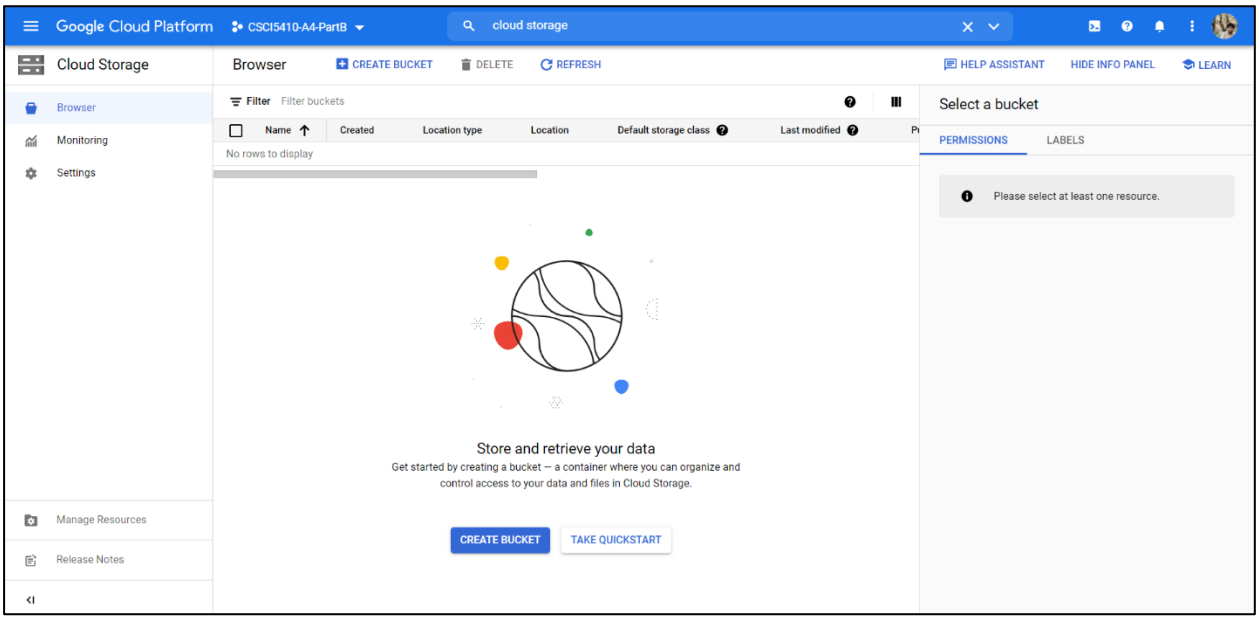


Figure 11 - No buckets initially in Cloud Storage [1]

Figures 12 to 15 display the program execution log messages. Since there is no bucket named **sourcedatab00xxxxxx** in Cloud Storage, the program creates a bucket **sourcedatab00xxxxxx**. Successful creation of bucket displays "**Bucket sourcedatab00xxxxxx created successfully**" message. Then, all the files inside the **Dataset\Train** folder are uploaded inside the **sourcedatab00xxxxxx** bucket one at a time.

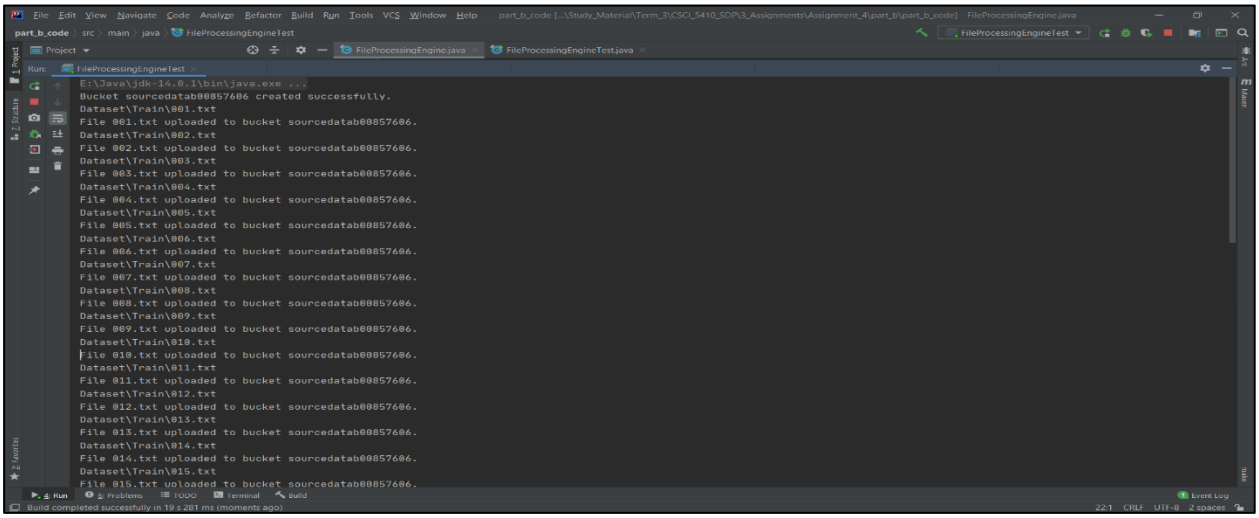


Figure 12 - Bucket sourcedatab00xxxxxx created successfully and training files uploading started

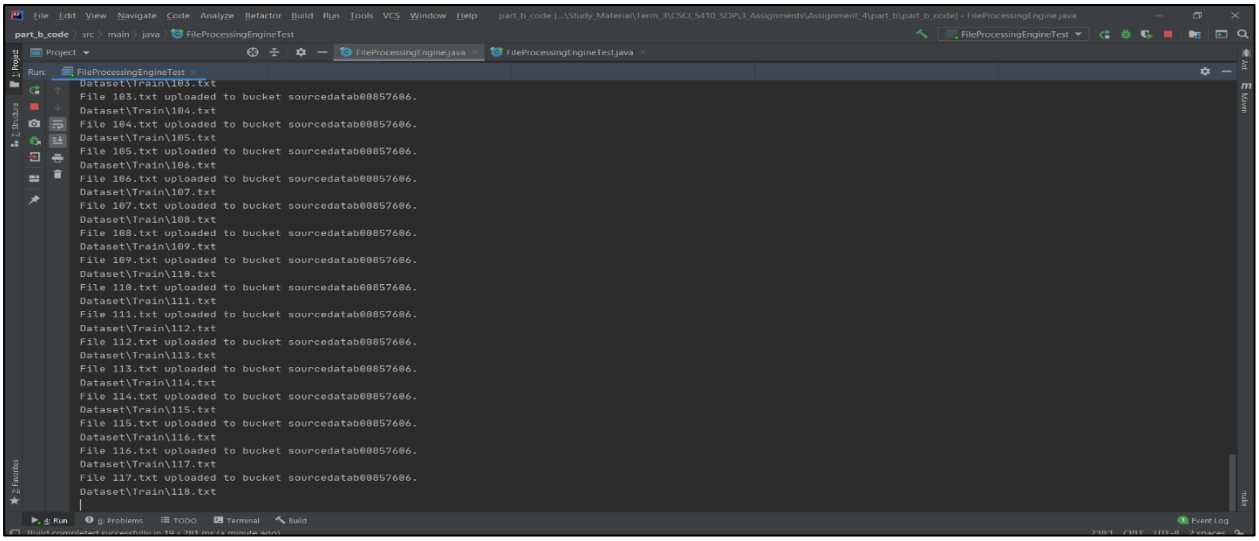


Figure 13 – Training files uploading in progress (contd.)

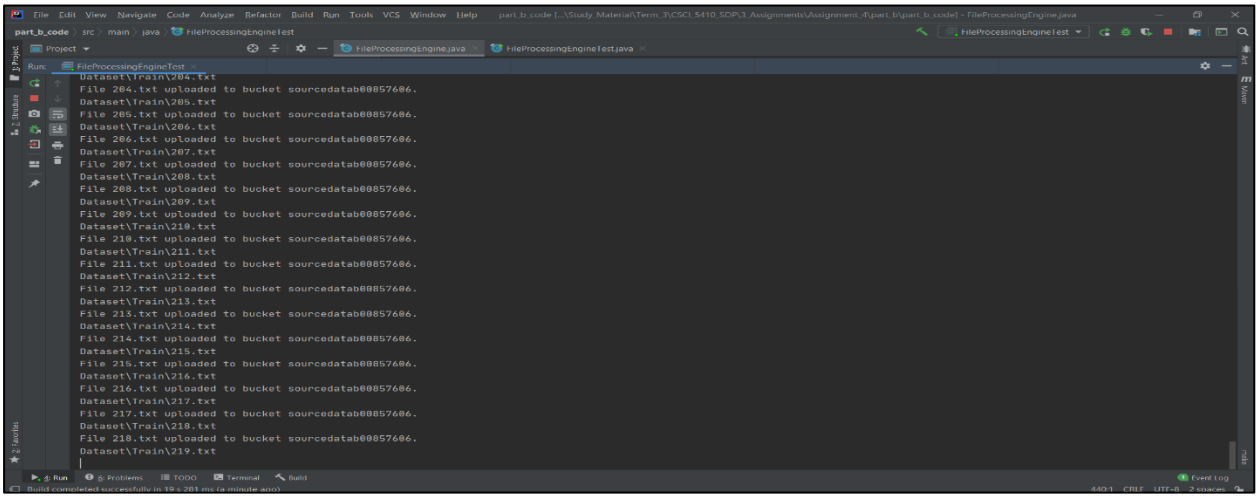


Figure 14 – Training files uploading in progress (contd.)

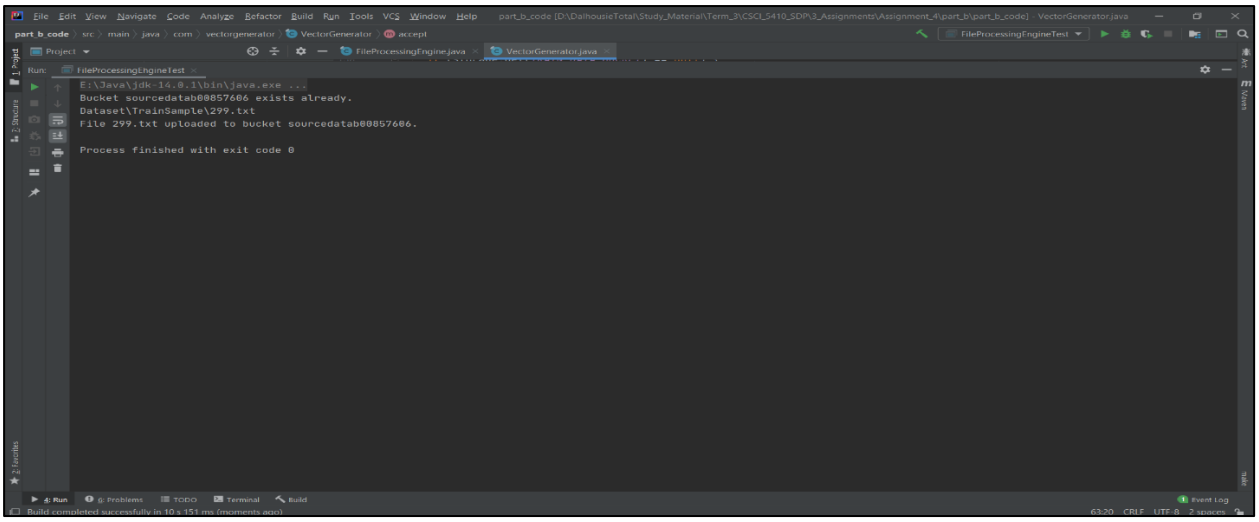


Figure 15 - Training files uploading completed (Last file uploaded - 299.txt)

Figure 16 displays the created **sourcedatab00xxxxxx** bucket in Cloud Storage.

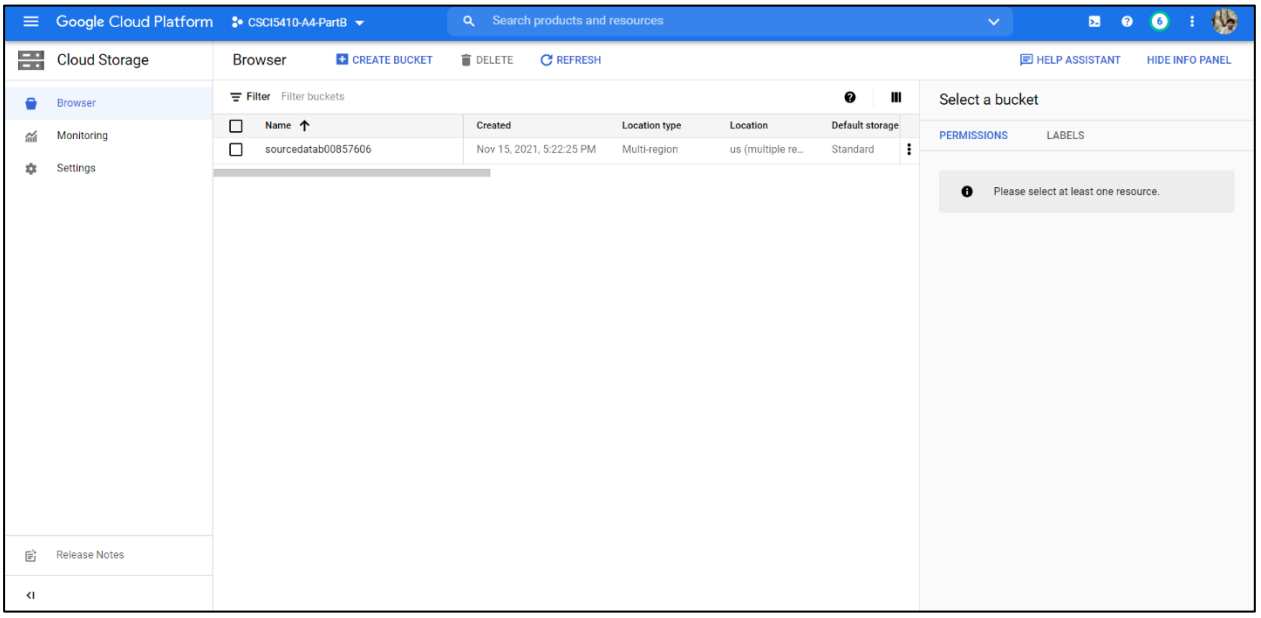


Figure 16 - Bucket sourcedatab00xxxxxx created in Cloud Storage under CSCI5410-A4-PartB project [1]

Figures 17 to 19 display the training files uploaded inside the **sourcedatab00xxxxxx** bucket. Total 299 files (001.txt to 299.txt) are uploaded inside the **sourcedatab00xxxxxx** bucket in Cloud Storage.

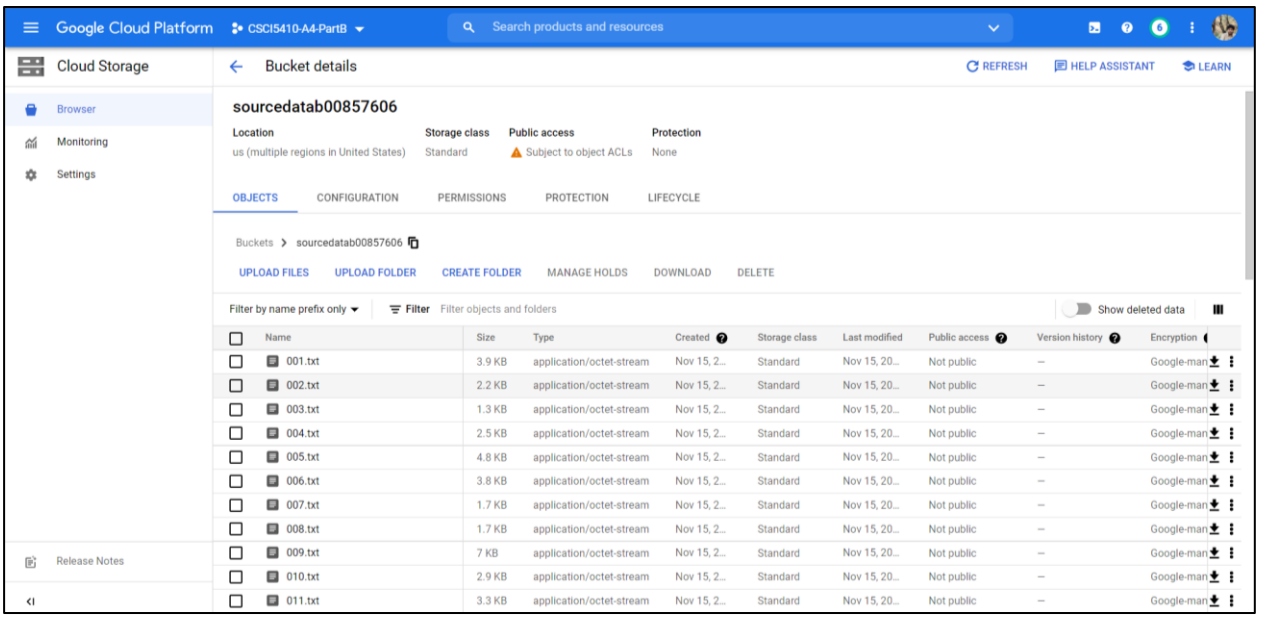


Figure 17 – Training files uploaded to bucket sourcedatab00xxxxxx [1]

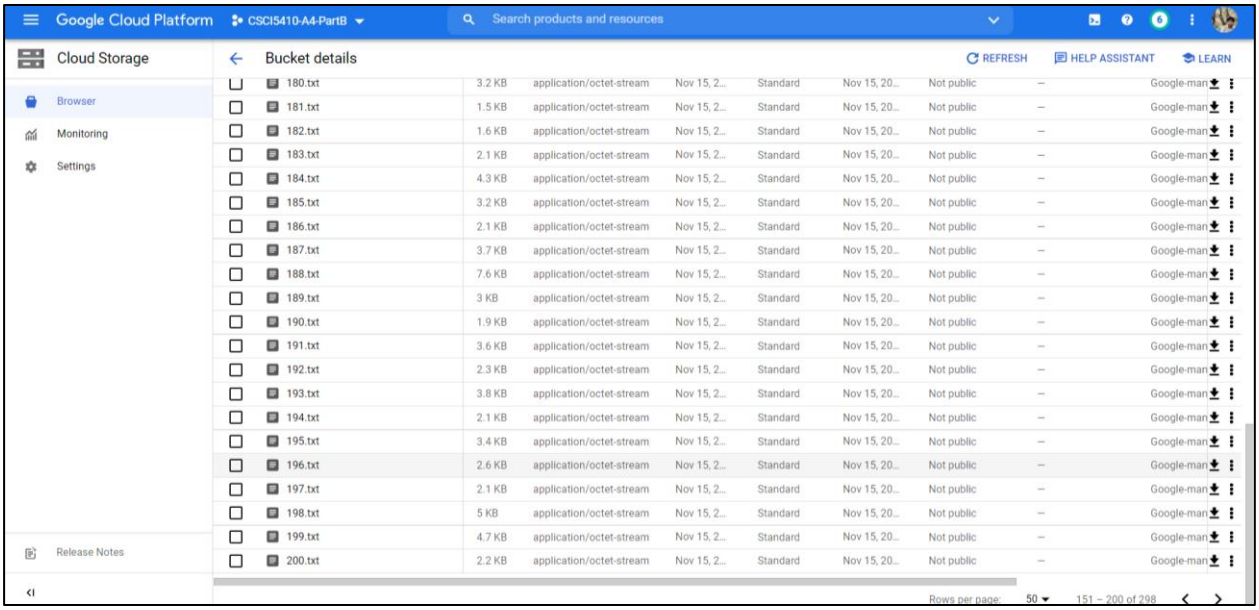


Figure 18 - Training files uploaded to bucket sourcedatab00xxxxxx (contd.) [1]

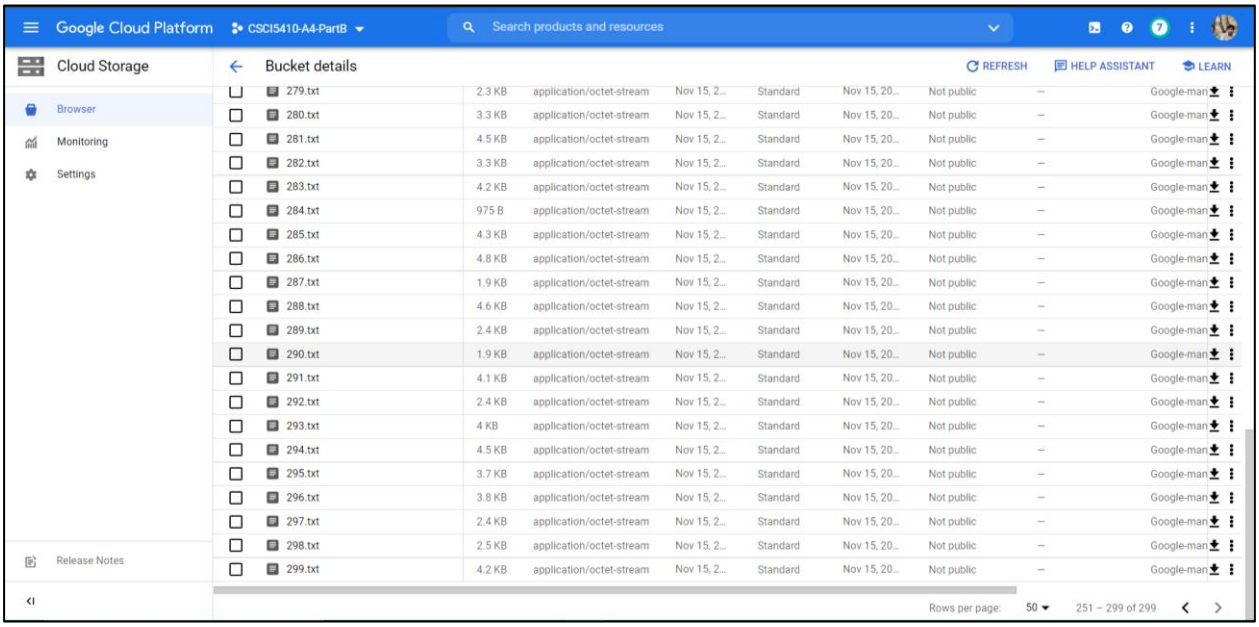


Figure 19 - Training files uploaded to bucket sourcedatab00xxxxxx (contd.) [1]

b) Once a file is uploaded, a cloud function - “generateVector” should extract words from all the files (remove the stop words). Then compute Levenshtein distance between the Current, and Next word.

Figure 20 displays the Cloud Function creation page. The name of the Cloud Function is **generateVector**. A trigger is set to Cloud Storage bucket **sourcedatab00xxxxxx** with event type Finalize/Create. Thus, whenever a new file is uploaded to the **sourcedatab00xxxxxx** bucket, this **generateVector** Cloud Function is executed. The memory allocated to the code is 512 MB and the timeout is set to 540 seconds (9 minutes).

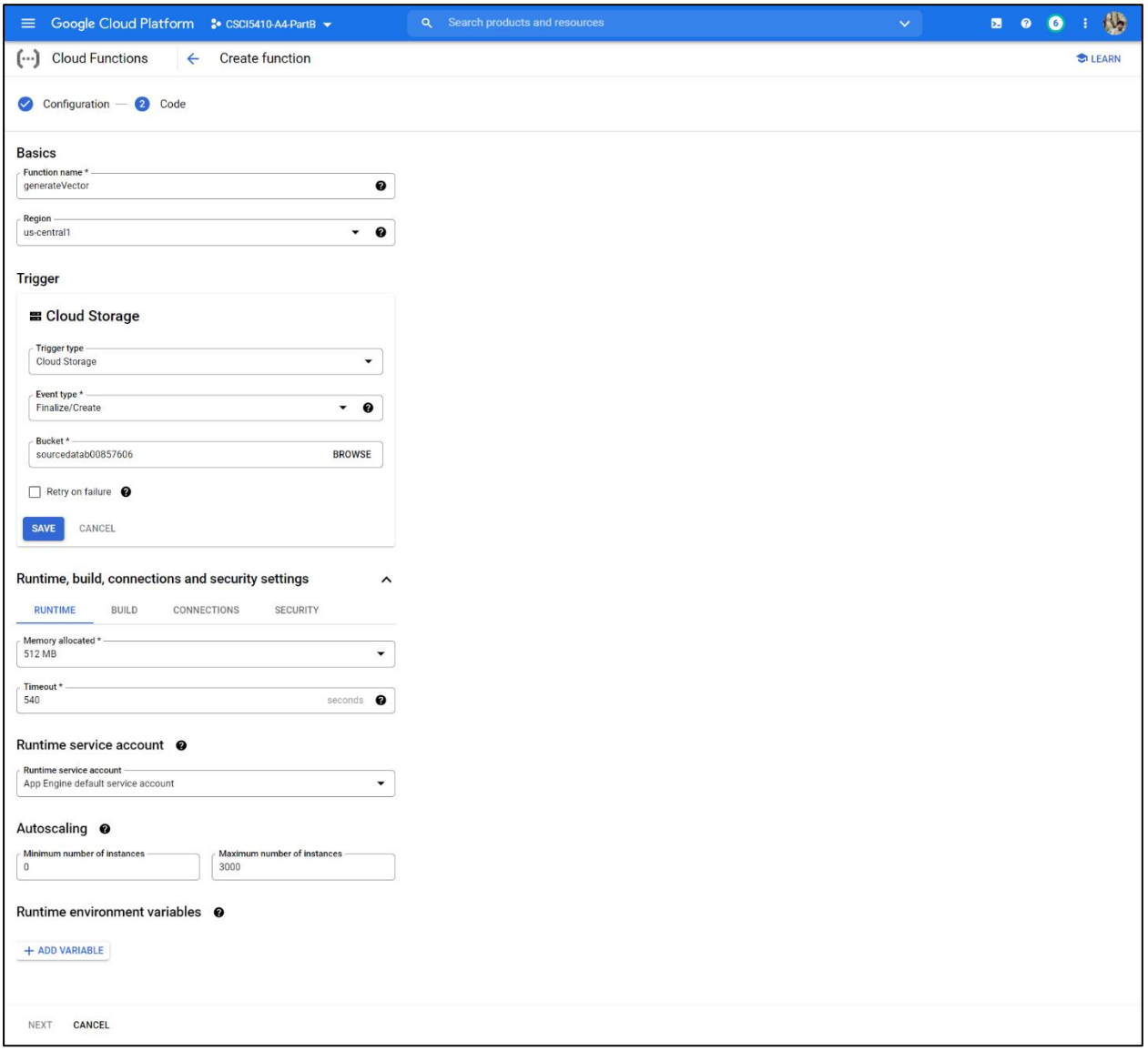


Figure 20 - Creation page of generateVector in Cloud Function. [2]

Figure 21 displays the successful deployment of Cloud Function – generateVector and **figure 22** displays the code for the Cloud Function.

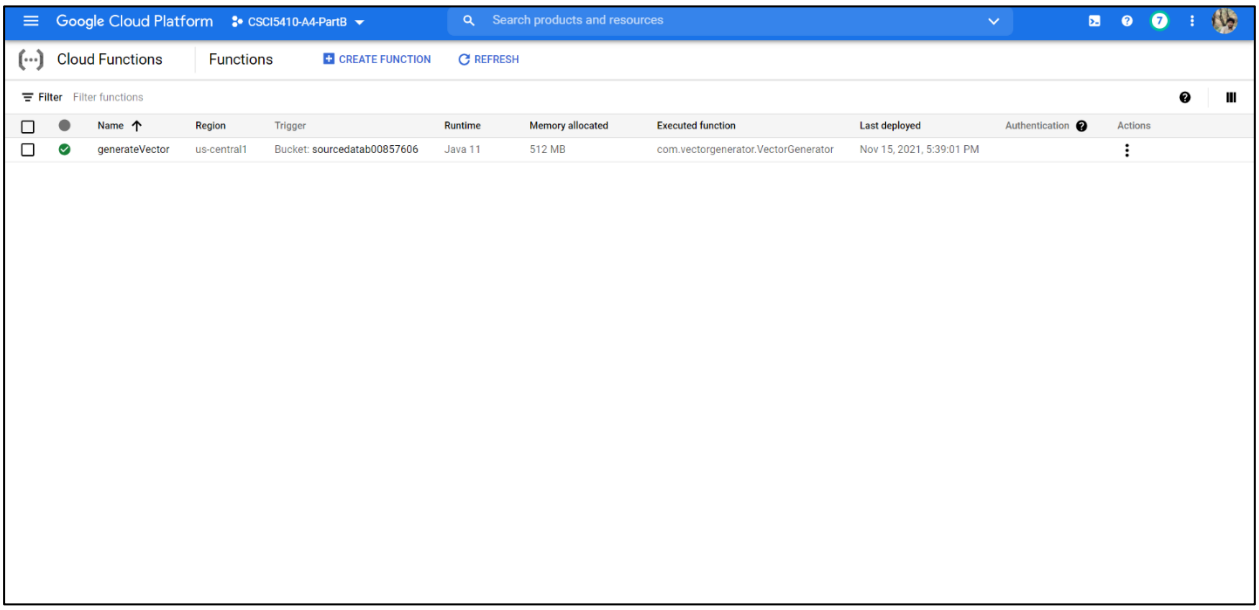


Figure 21 - generateVector Cloud Function deployed successfully [2]

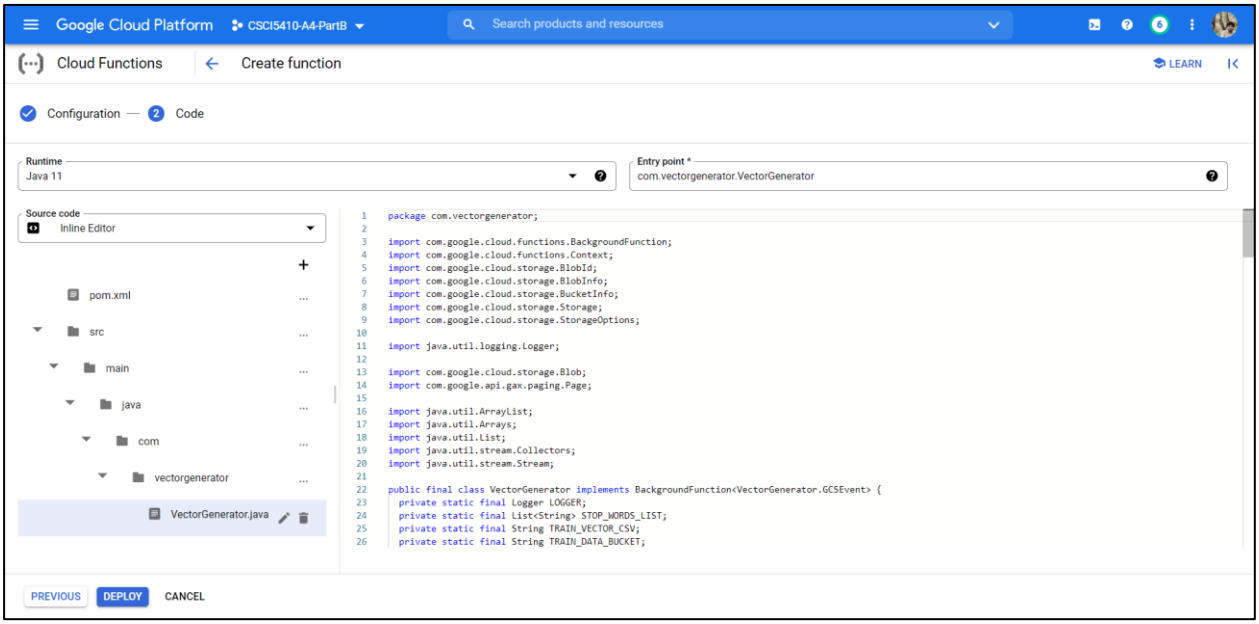


Figure 22 - Code added for Cloud Function - generateVector [2]

Figures 23 to 26 display successful log messages generation when the last training file (i.e., File 299.txt) is uploaded to the **sourcedatab00xxxxxx** bucket. This training file triggered the Cloud Function. The log messages display the file name under process and the list of words after removing stop words. The last figure (i.e., Figure 26) displays log messages that say a new bucket **traindatab00xxxxxx** is created successfully and the **trainVectors.csv** file is uploaded to the bucket **traindatab00xxxxxx**.

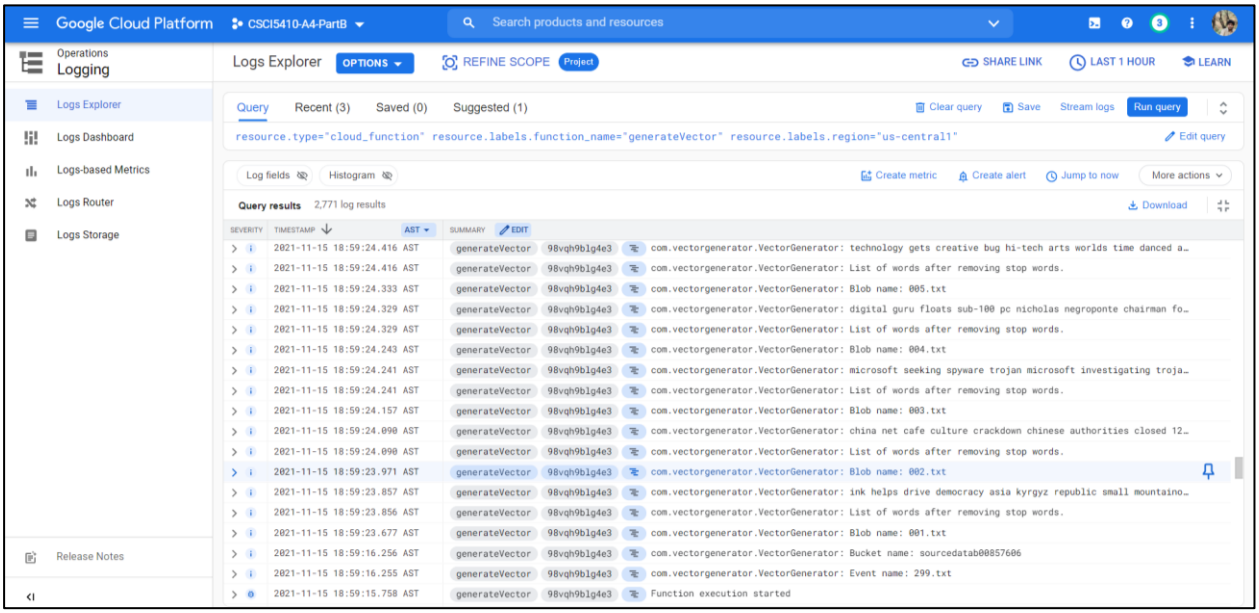


Figure 23 – Log messages generated on successful execution of Cloud Function – generateVector [7]

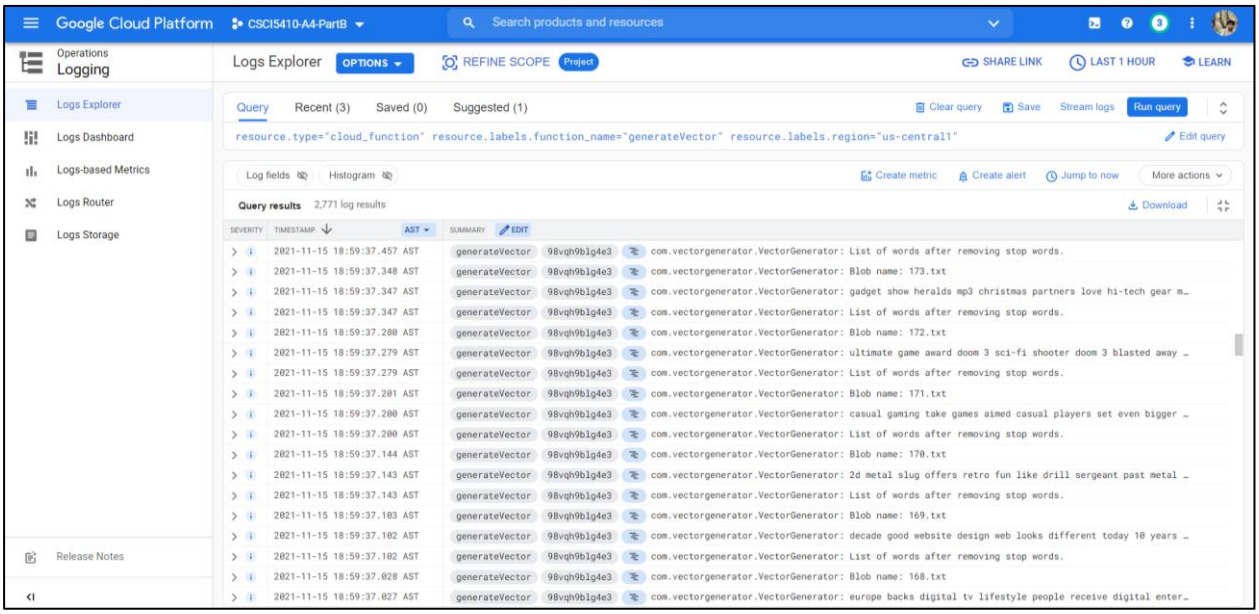


Figure 24 – Log messages generated on successful execution of Cloud Function - generateVector (contd.) [7]

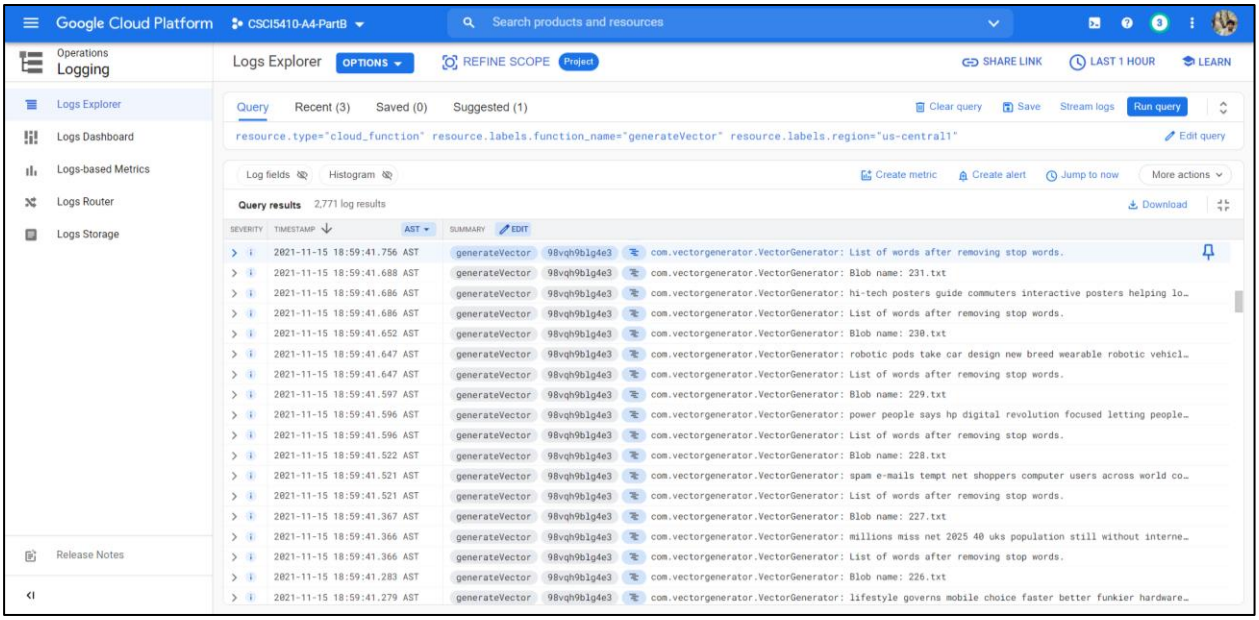


Figure 25 – Log messages generated on successful execution of Cloud Function - generateVector (contd.) [7]

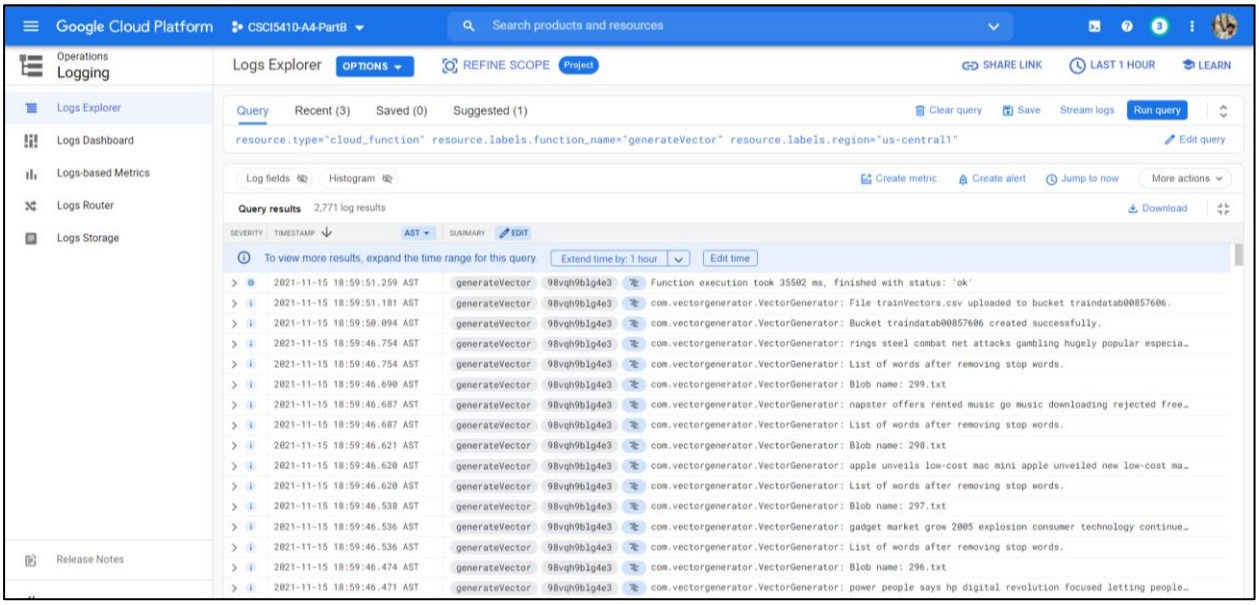


Figure 26 – Log messages generated on successful execution of Cloud Function - generateVector (contd.) [7]

Figure 27 displays the newly created bucket – traindatab00xxxxxx.

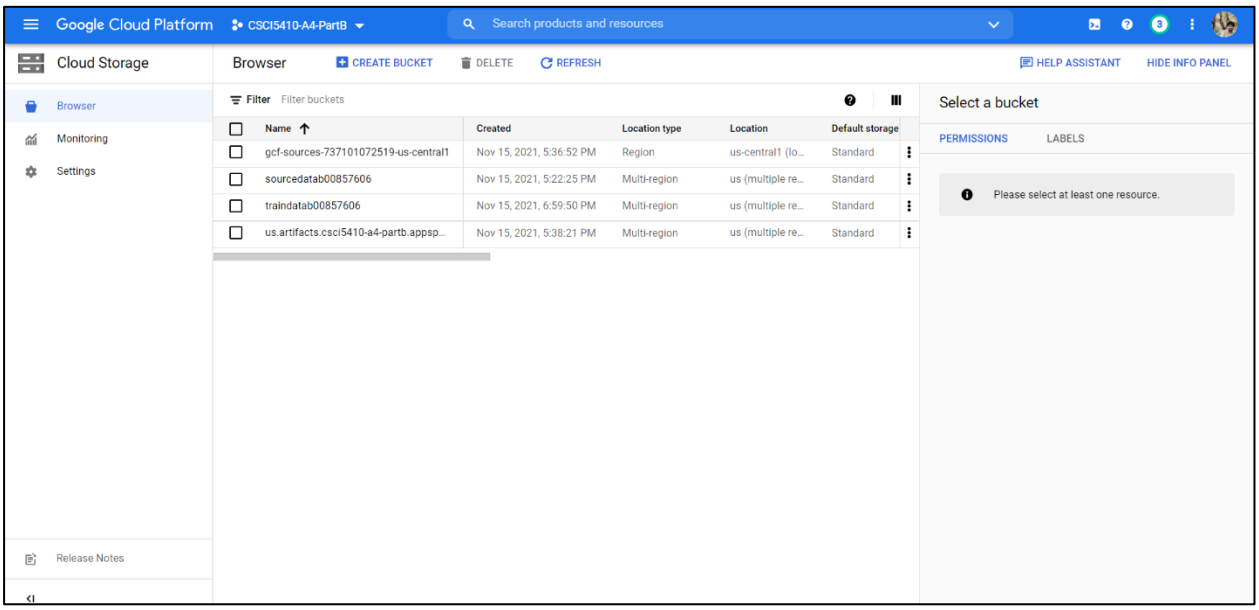


Figure 27 - traindatab00xxxxxx bucket created [1]

Figure 28 displays the **trainVectors.csv** file uploaded to the traindatab00xxxxxx bucket.

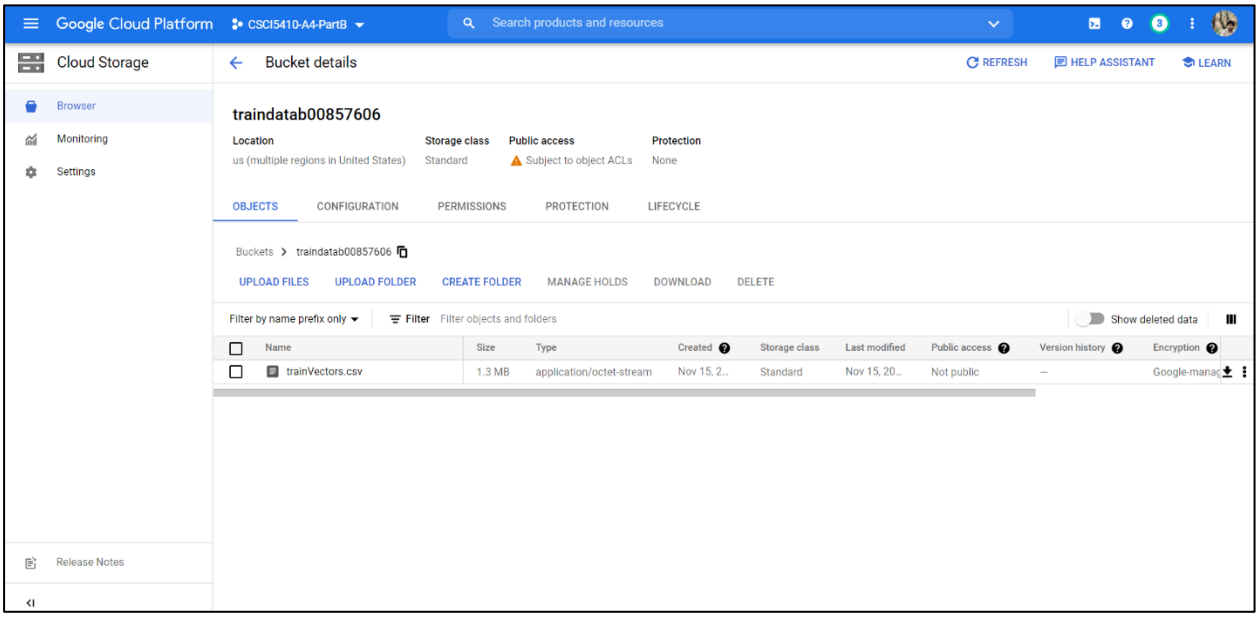
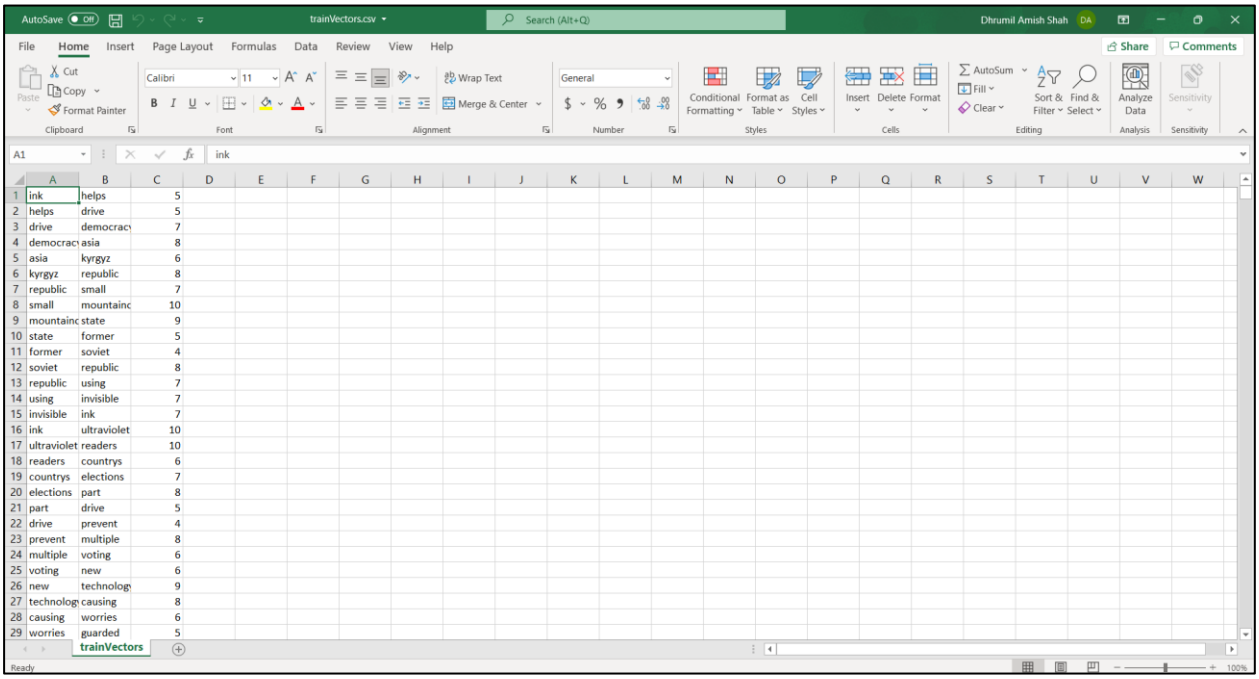


Figure 28 - trainVectors.csv file created inside traindatab00xxxxxx bucket [1]

c) This file "trainVector.csv" is saved in a new bucket TrainDataB00xxxxxx.

Figure 29 displays the file **trainVector.csv** downloaded from the bucket **traindatab00xxxxxx**. The **trainVector.csv** file contains three columns namely the current word, the next word, and the Levenshtein distance.



	A	B	C
1	ink	helps	5
2	helps	drive	5
3	drive	democracy	7
4	democracy	asia	8
5	asia	kyrgyz	6
6	kyrgyz	republic	8
7	republic	small	7
8	small	mountainc	10
9	mountainc	state	9
10	state	former	5
11	former	soviet	4
12	soviet	republic	8
13	republic	using	7
14	using	invisible	7
15	invisible	ink	7
16	ink	ultraviolet	10
17	ultraviolet	readers	10
18	readers	countrys	6
19	countrys	elections	7
20	elections	part	8
21	part	drive	5
22	drive	prevent	4
23	prevent	multiple	8
24	multiple	voting	6
25	voting	new	6
26	new	technology	9
27	technology	causing	8
28	causing	worries	6
29	worries	guarded	5

Figure 29 - trainVector.csv file with three columns

d) GCP ML should get the training data for a clustering algorithm (KMenas) from the TrainDataB00xxxxxx bucket.

The service used for training the K-Means algorithm with the training dataset is VertexAI. The Vertex AI service in GCP allows developers to enable a service called notebooks.googleapis.com. With the help of this cloud-based API service, I created a Notebook instance using Workbench from the left navigation panel in Vertex AI. The Notebook instance, on creation and configuration, offers an option of hosting JupyterLab on the browser where the Python code can be written and executed.

Figure 30 displays the Vertex AI console in GCP offering an option to create a Notebook instance.
.

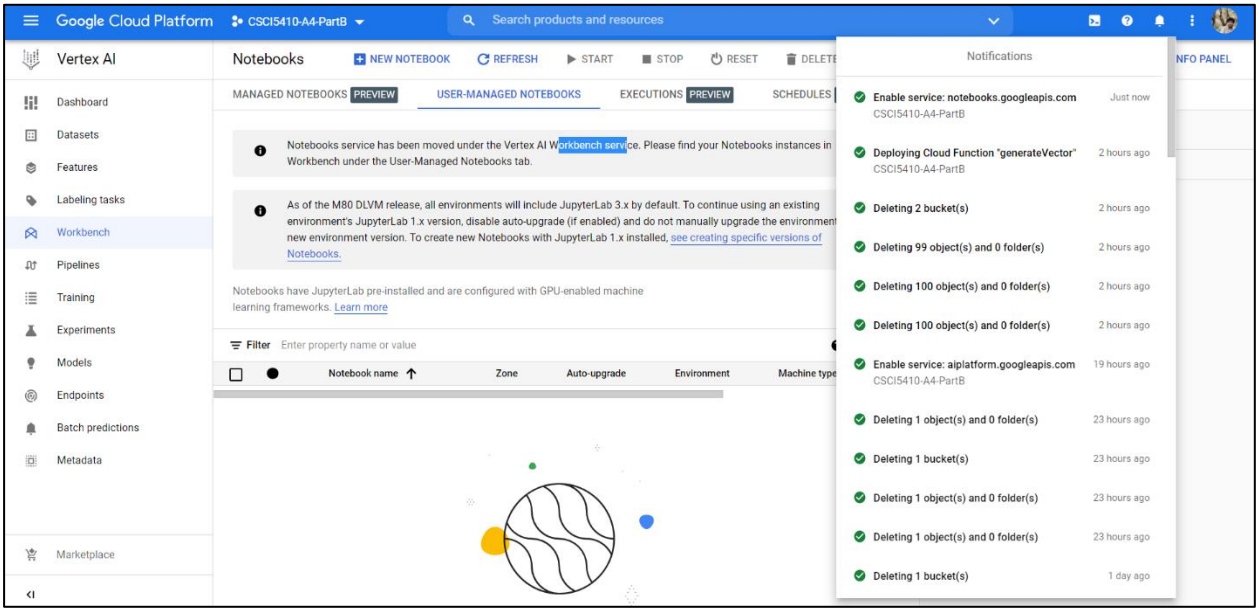


Figure 30 - Vertex AI console with Notebooks API enabled [5]

The Elbow method is used to find the optimal value of K (i.e., number of clusters) for the K-Means algorithm. The sharp bend in the graph determines the optimal value of K. **Figure 31** displays the graph plotted by the Python code. From the graph, it can be determined that the value of K is 4 as there is a sharp bend for K = 4.

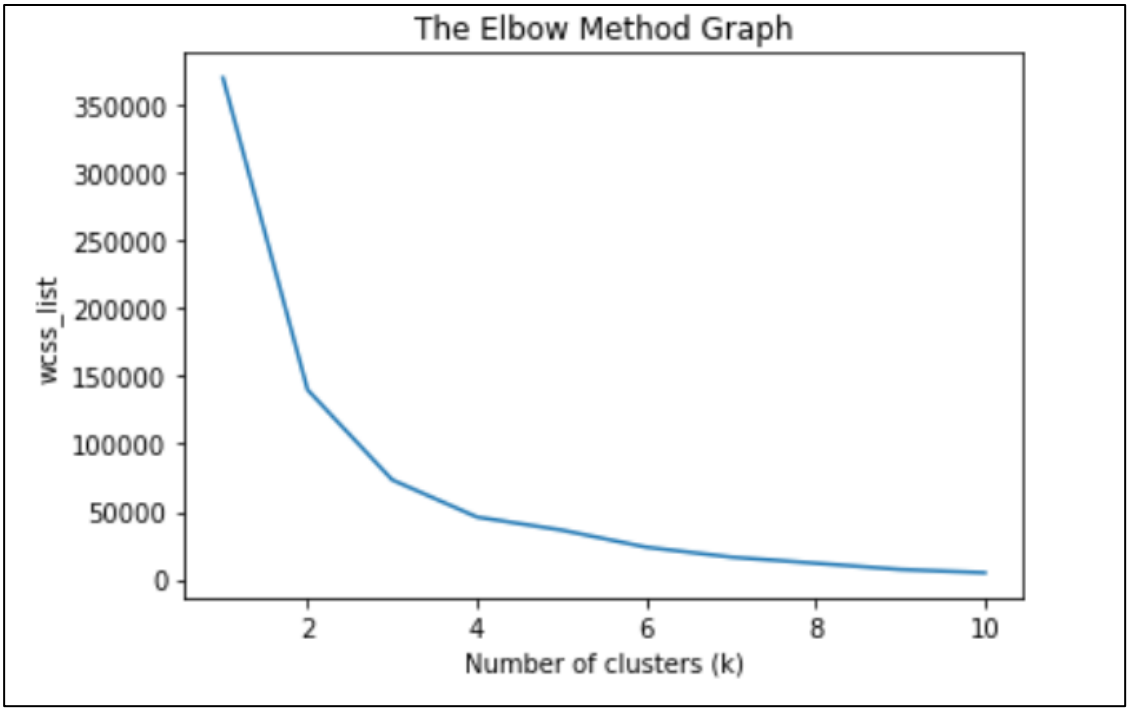


Figure 31 - Elbow method graph [8]

Figure 32 displays the final graph plotted by the Python code for the training dataset by keeping the value of K as 4. Thus, the training dataset is grouped into 4 clusters. Also, the figure displays the centroids for all 4 clusters.

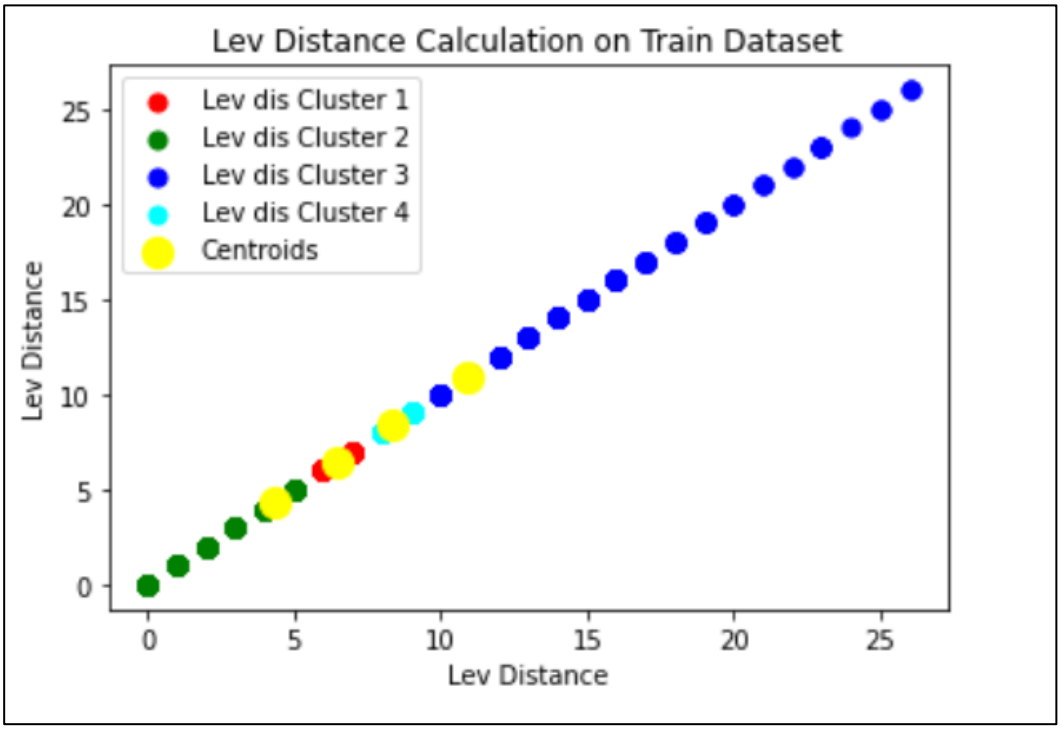


Figure 32 - Levenshtein distance calculated for the Training dataset

e) Once the training is done, like point (a), upload the test files given in the Test folder (300 to 401) to SourceDataB00xxxxxx.

Figures 33 to 36 display the program execution log messages. Since the bucket named **sourcedatab00xxxxxx** exists already in Cloud Storage, the program does not create a new bucket. Instead, the program displays **"Bucket sourcedatab00xxxxxx exists already."**. Then, all the files inside the **Dataset\Test** folder are uploaded to the **sourcedatab00xxxxxx** bucket one at a time.

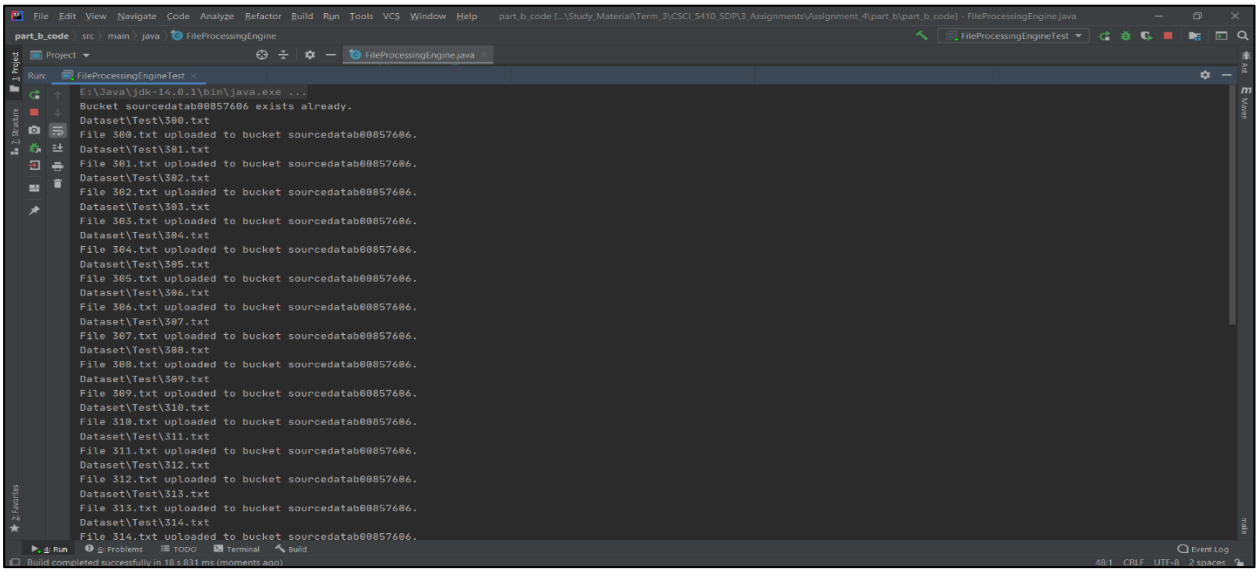


Figure 33 - Test files uploading in progress

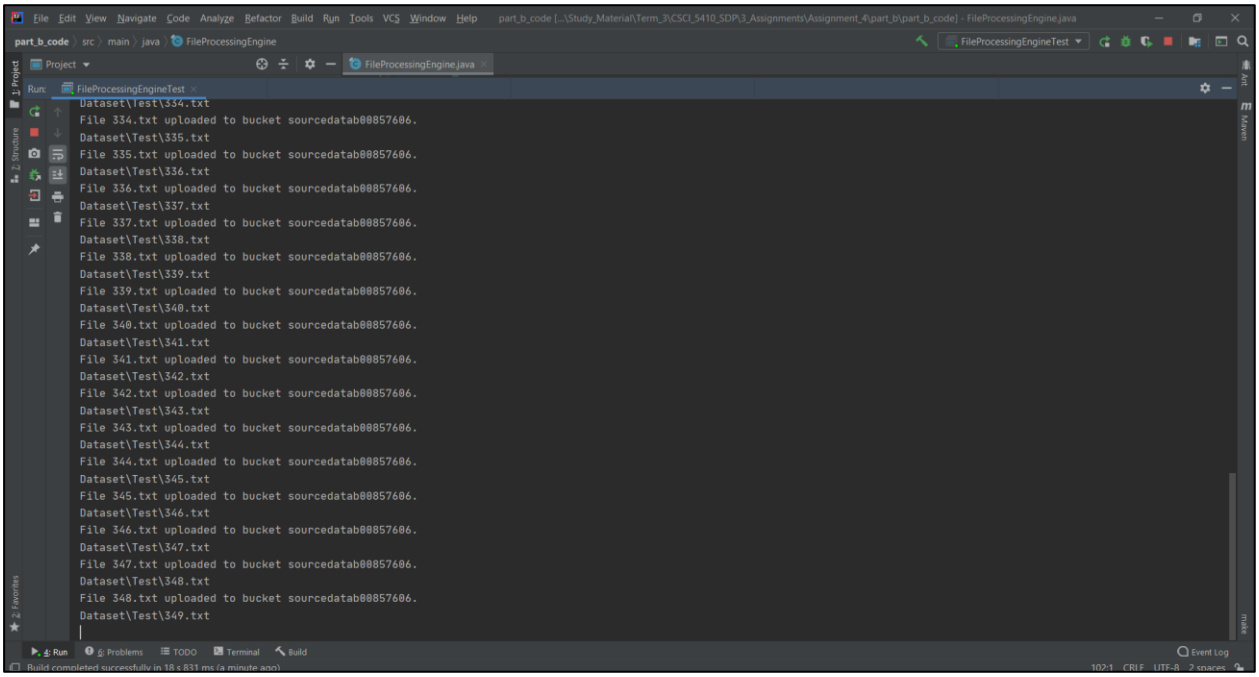


Figure 34 - Test files uploading in progress (contd.)

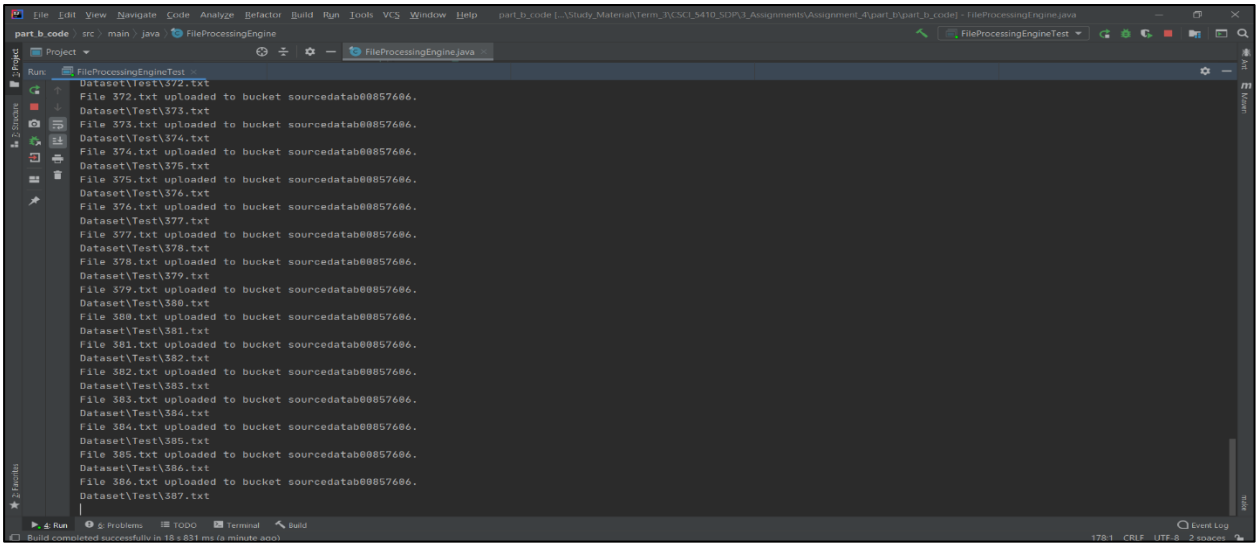


Figure 35 - Test files uploading in progress (contd.)

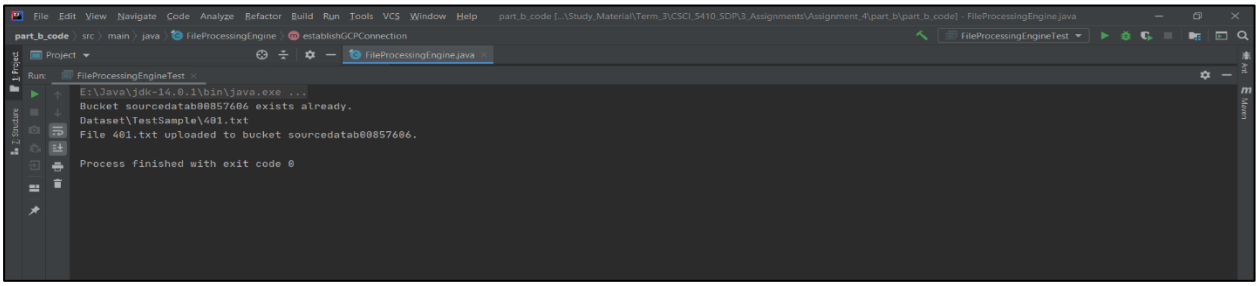


Figure 36 - Test files uploading completed (Last file uploaded – 401.txt)

Figures 37 to 39 display the test files uploaded to the **sourcedatab00xxxxxx** bucket. Total 102 files (300.txt to 401.txt) are uploaded to the **sourcedatab00xxxxxx** bucket in Cloud Storage.

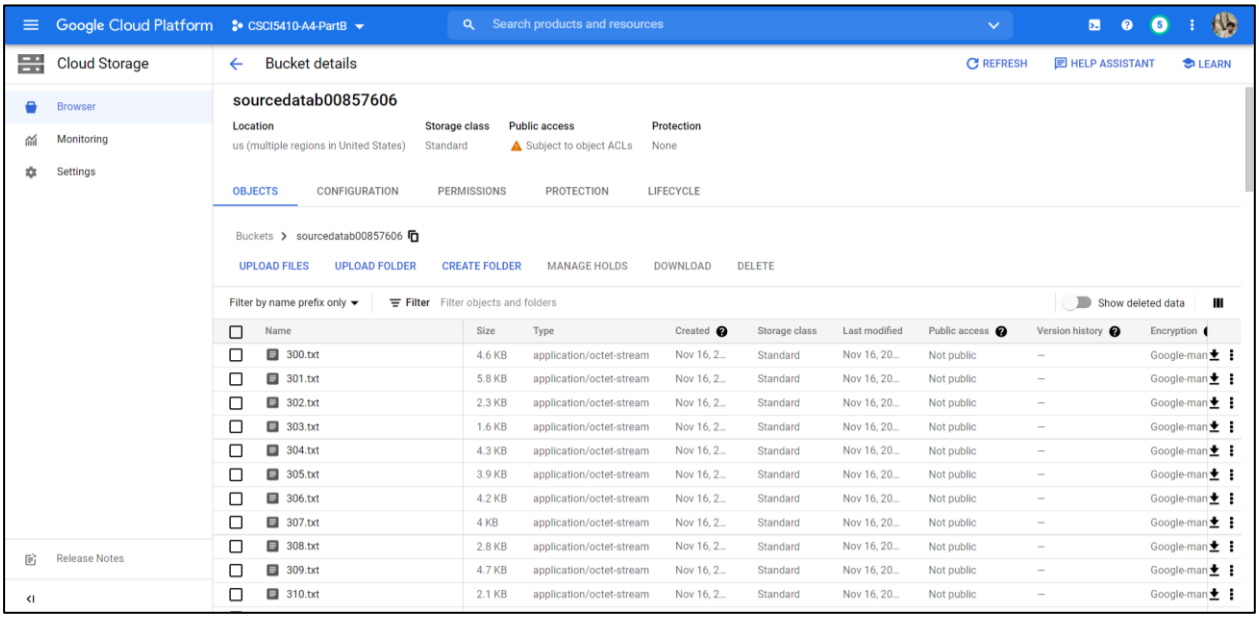


Figure 37 - Test files uploaded to the sourcedatab00xxxxxx bucket [1]

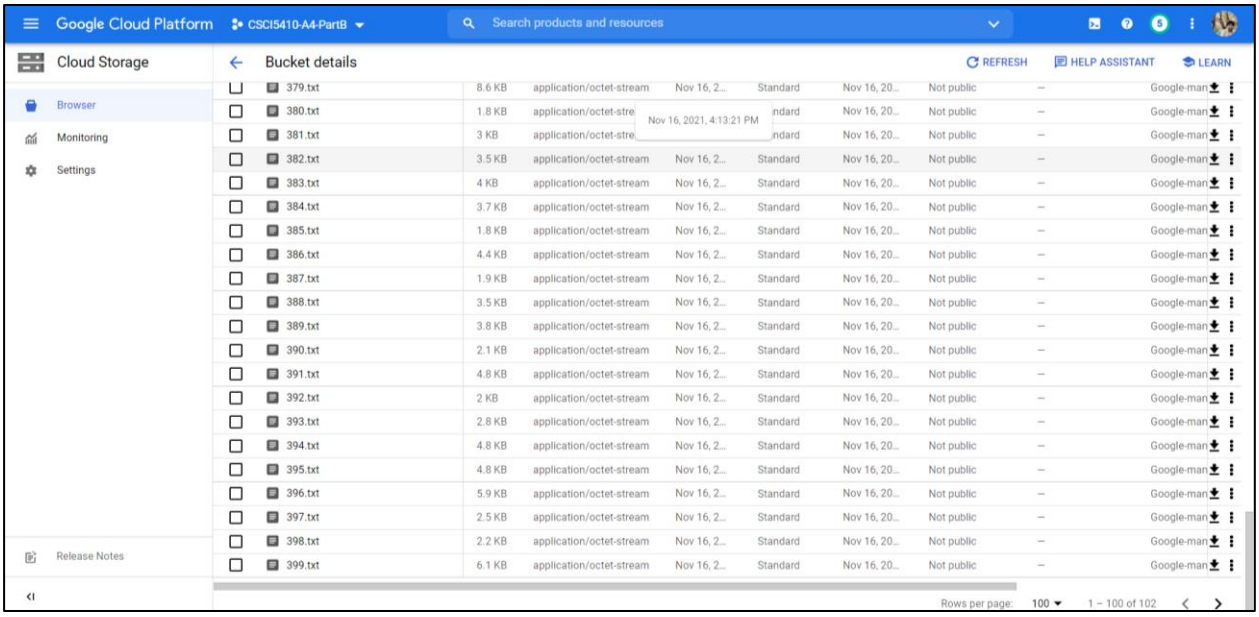


Figure 38 - Test files uploaded to the sourcedatab00xxxxxx bucket (contd.) [1]

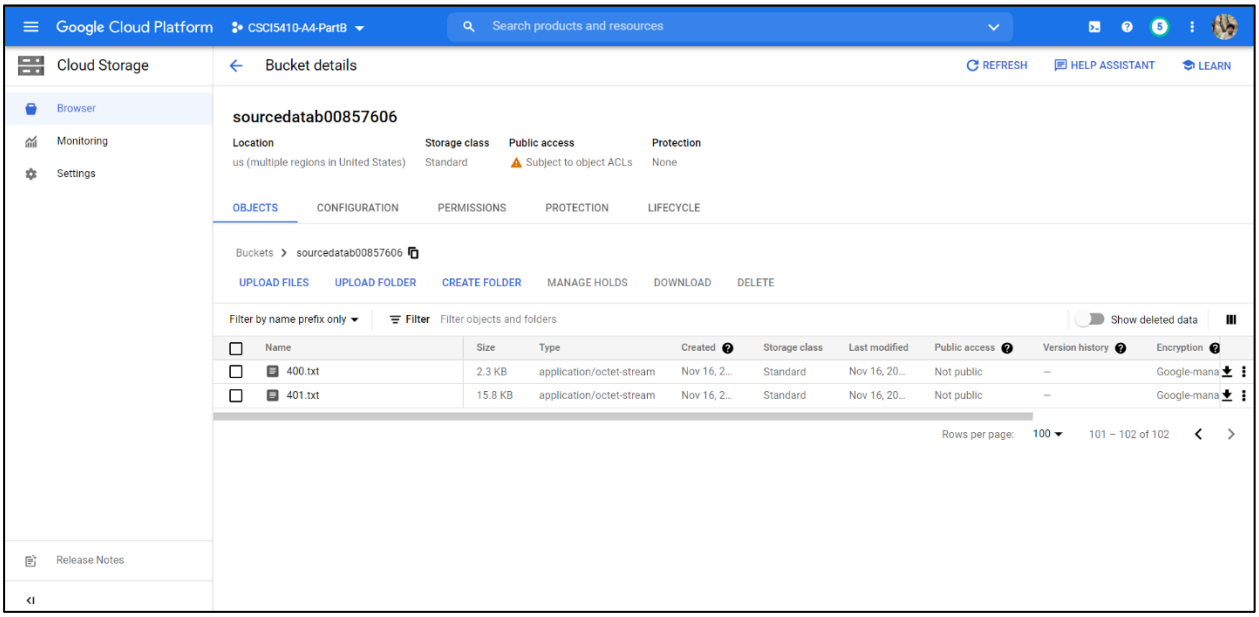


Figure 39 - Test files uploaded to the sourcedatab00xxxxxx bucket (contd.) [1]

f) The cloud function generateVector computes the distance vector same as point (b) and store it in "testVector.csv".

Figure 40 displays the newly created bucket named **testdatab00xxxxxx** and **figure 41** displays the testVectors.csv file uploaded to the newly created bucket **testdatab00xxxxxx**.

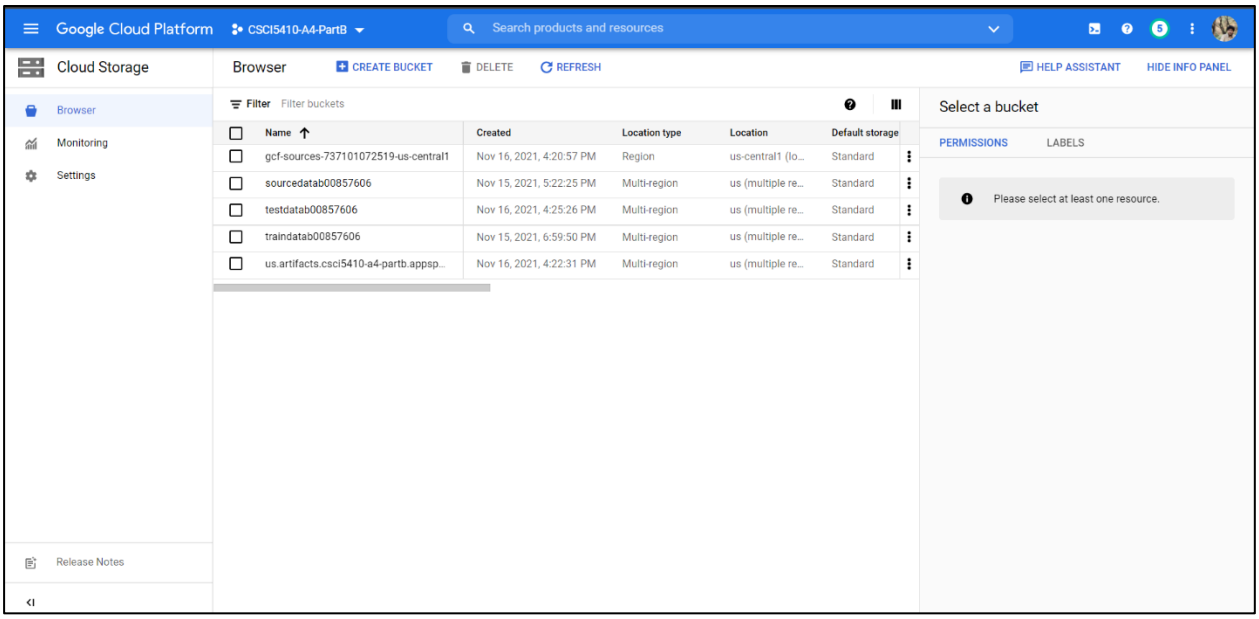


Figure 40 - Bucket testdatab00xxxxxx created in Cloud Storage [1]

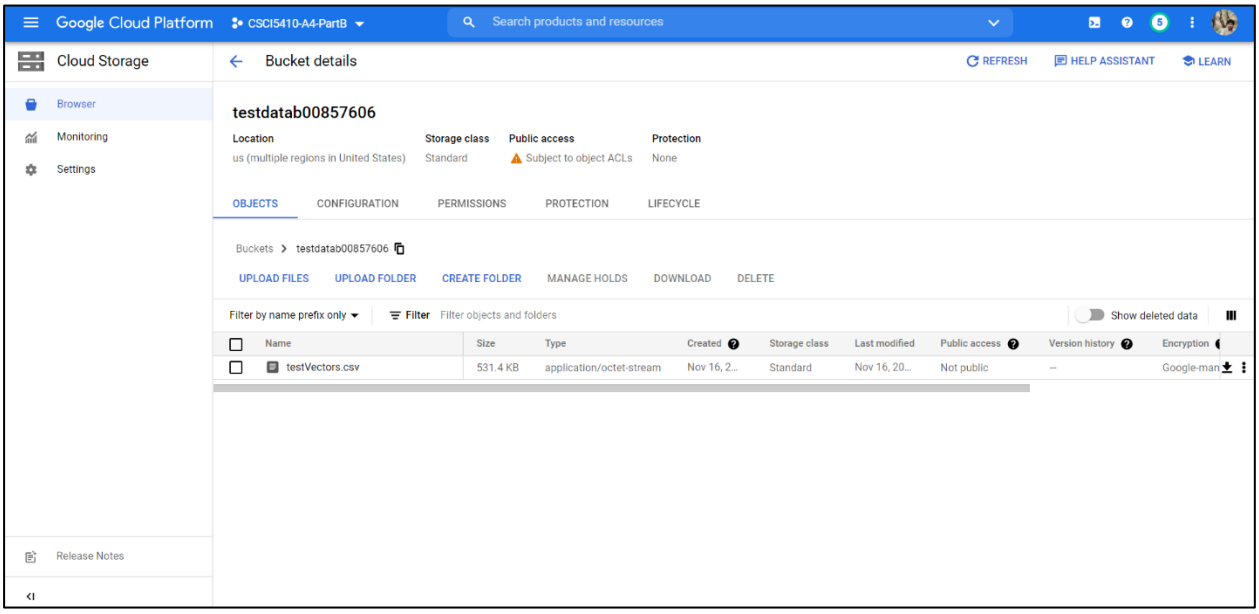
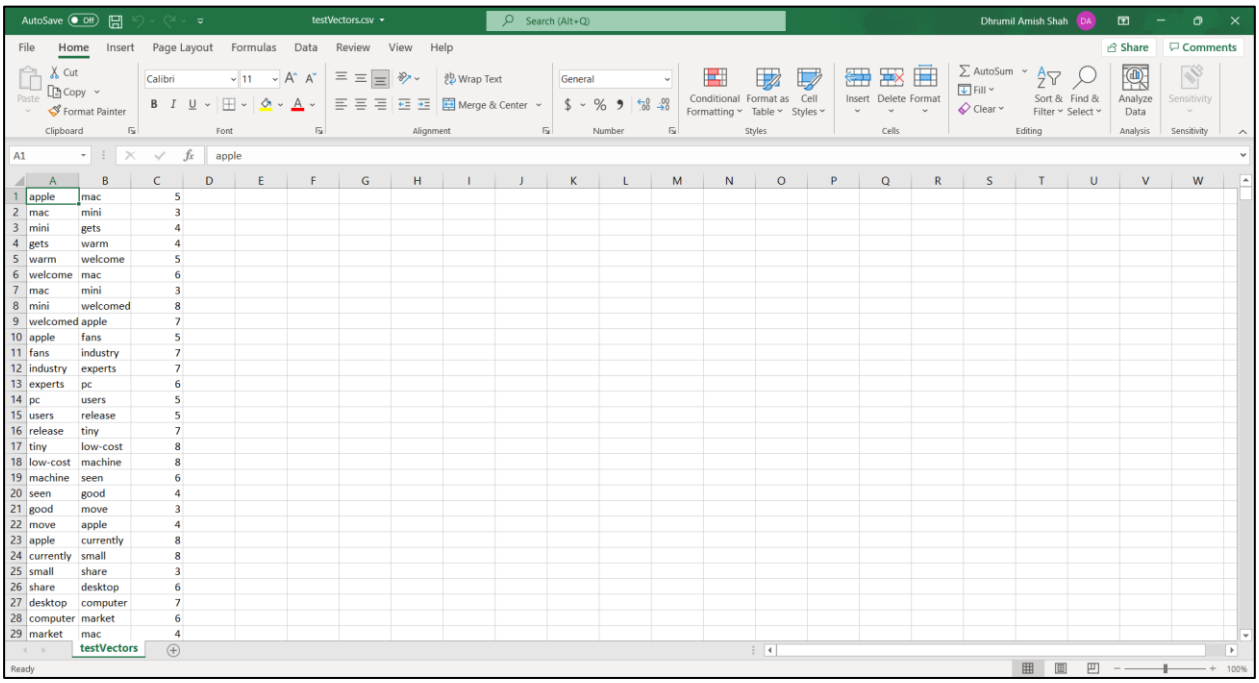


Figure 41 - testVectors.csv file uploaded to testdatab00xxxxxx bucket in Cloud Storage [1]

g) This file (testVector.csv) is saved in another new bucket, TestDataB00xxxxxx.

Figure 42 displays the content of the file **testVector.csv** downloaded from the bucket **testdatab00xxxxxx**. The **testVector.csv** file contains three columns namely the current word, the next word, and the Levenshtein distance.



	A	B	C
1	apple	mac	5
2	mac	mini	3
3	mini	gets	4
4	gets	warm	4
5	warm	welcome	5
6	welcome	mac	6
7	mac	mini	3
8	mini	welcomed	8
9	welcomed	apple	7
10	apple	fans	5
11	fans	industry	7
12	industry	experts	7
13	experts	pc	6
14	pc	users	5
15	users	release	5
16	release	tiny	7
17	tiny	low-cost	8
18	low-cost	machine	8
19	machine	seen	6
20	seen	good	4
21	good	move	3
22	move	apple	4
23	apple	currently	8
24	currently	small	8
25	small	share	3
26	share	desktop	6
27	desktop	computer	7
28	computer	market	6
29	market	mac	4

Figure 42 - testVector.csv file content

h) GCP ML should get the test data for the KMeans algorithm from the TestDataB00xxxxxx bucket.

Figure 43 displays the final graph plotted by the Python code for the testing dataset by keeping the value of K as 4. The testing dataset is predicted. Also, the figure displays the centroids for all 4 clusters.

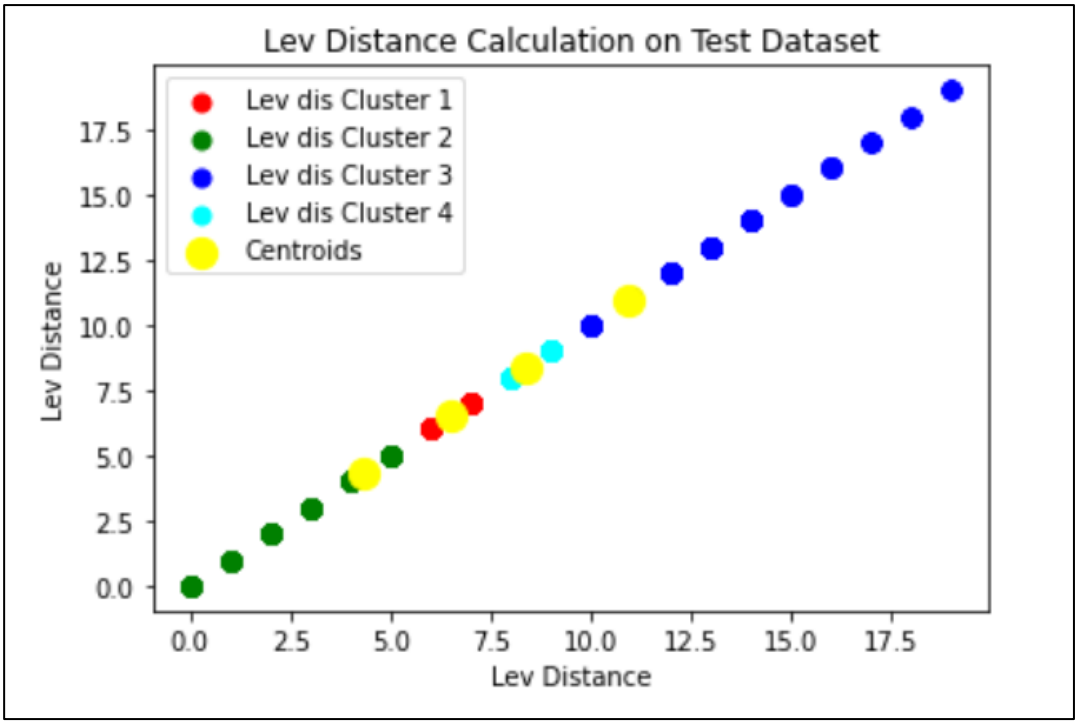


Figure 43 - Levenshtein distance calculated for the Testing dataset

i) Finally, write a code or configure a service to obtain information about clusters (e.g. centroids, cluster numbers, outliers etc. which are generated by GCP ML), and display the clusters.

The code below is responsible for generating images in **step d** and **step h**. I have uploaded the same code on GitLab. Here is the link to that code - https://git.cs.dal.ca/dashah/csci-5410-f2021-b00857606-dhrumil-amish-shah/-/blob/main/assignment_4_code/part_b/k-means-algorithm.py.

I have explained the working of the below code using the inline comments. Also, the major steps performed by the program are separated by a space. I followed a tutorial on the javaTpoint website [9] to write the code.

```
# Author: Dhrumil Amish Shah (B00857606|dh416386@dal.ca)
# Import required libraries
import numpy as nm # numpy for performing mathematical calculations
import matplotlib.pyplot as mtp # matplotlib.pyplot for plotting graphs
import pandas as pd # pandas for managing the dataset
from google.cloud import storage # google.cloud storage for getting training and testing dataset
from io import BytesIO # BytesIO for byte manipulation

# Import train CSV file from Google Cloud Storage into dataframe - train_csv_df
train_bucket_name = "traindatab00857606" # Train bucket name
train_csv_file_name = "trainVectors.csv" # Train CSV file name
storage_client = storage.Client() # Google Cloud Storage client instance
train_bucket = storage_client.get_bucket(train_bucket_name) # Train bucket instance
train_csv_file_blob = train_bucket.get_blob(train_csv_file_name) # Train CSV file blob instance
train_csv_file_content = train_csv_file_blob.download_as_bytes() # Train CSV file content
instance
train_csv_df = pd.read_csv(BytesIO(train_csv_file_content)) # Train CSV dataframe instance
print(train_csv_df) # Print the content of train_csv_df instance

# Train CSV dataset of only last column (Levenshtein distance last column)
lev_dis_train_ds = train_csv_df.iloc[:, [2]].values
print(lev_dis_train_ds)

# Finding optimal number of clusters using the Elbow method
from sklearn.cluster import KMeans # Import KMeans algorithm (Clustering algorithm)
wcss_list= []  # Initializing the list for the values of WCSS

# Using for loop for iterations from [1, 11)
for cluster_num in range(1, 11):
kmeans = KMeans(n_clusters=cluster_num, init='k-means++', random_state= 42)
kmeans.fit(lev_dis_train_ds)
wcss_list.append(kmeans.inertia_)
```

```
# Plot the Elbow graph to decide the value of 'k' in K-means algorithm
mtp.plot(range(1, 11), wcss_list)
mtp.title('The Elbow Method Graph')
mtp.xlabel('Number of clusters (k)')
mtp.ylabel('wcss_list')
mtp.show()

# Training the K-means model on a dataset
kmeans = KMeans(n_clusters=4, init='k-means++', random_state= 42)
y_train_pred = kmeans.fit_predict(lev_dis_train_ds)
print(y_train_pred)

# Visulaizing the clusters
mtp.scatter(lev_dis_train_ds[y_train_pred == 0], lev_dis_train_ds[y_train_pred == 0], s = 50, c =
'red', label = 'Lev dis Cluster 1') # For first cluster
mtp.scatter(lev_dis_train_ds[y_train_pred == 1], lev_dis_train_ds[y_train_pred == 1], s = 50, c =
'green', label = 'Lev dis Cluster 2') # For second cluster
mtp.scatter(lev_dis_train_ds[y_train_pred== 2], lev_dis_train_ds[y_train_pred== 2], s = 50, c =
'blue', label = 'Lev dis Cluster 3') # For third cluster
mtp.scatter(lev_dis_train_ds[y_train_pred == 3], lev_dis_train_ds[y_train_pred == 3], s = 50, c =
'cyan', label = 'Lev dis Cluster 4') # For fourth cluster
mtp.scatter(kmeans.cluster_centers_[:], kmeans.cluster_centers_[:], s = 150, c = 'yellow', label =
'Centroids')
mtp.title('Lev Distance Calculation on Train Dataset')
mtp.xlabel('Lev Distance')
mtp.ylabel('Lev Distance')
mtp.legend()
mtp.show()

# Import test CSV file from Google Cloud Storage into dataframe - test_csv_df
test_bucket_name = "testdatab00857606" # Test bucket name
test_csv_file_name = "testVectors.csv" # Test CSV file name
test_bucket = storage_client.get_bucket(test_bucket_name) # Test bucket instance
test_csv_file_blob = test_bucket.get_blob(test_csv_file_name) # Test CSV file blob instance
test_csv_file_content = test_csv_file_blob.download_as_bytes() # Test CSV file content instance
test_csv_df = pd.read_csv(BytesIO(test_csv_file_content)) # Test CSV dataframe instance
print(test_csv_df) # Print the content of test_csv_df instance

# Test CSV Dataset of only last column (Levenshtein distance last column)
lev_dis_test_ds = test_csv_df.iloc[:, [2]].values
print(lev_dis_test_ds)

# Predict for test dataset
y_test_predict= kmeans.predict(lev_dis_test_ds)
print(y_test_predict)
```

```
# Visulaizing the clusters
mtp.scatter(lev_dis_test_ds[y_test_predict == 0], lev_dis_test_ds[y_test_predict == 0], s = 50, c = 'red', label = 'Lev dis Cluster 1') # For first cluster
mtp.scatter(lev_dis_test_ds[y_test_predict == 1], lev_dis_test_ds[y_test_predict == 1], s = 50, c = 'green', label = 'Lev dis Cluster 2') # For second cluster
mtp.scatter(lev_dis_test_ds[y_test_predict== 2], lev_dis_test_ds[y_test_predict== 2], s = 50, c = 'blue', label = 'Lev dis Cluster 3') # For third cluster
mtp.scatter(lev_dis_test_ds[y_test_predict == 3], lev_dis_test_ds[y_test_predict == 3], s = 50, c = 'cyan', label = 'Lev dis Cluster 4') # For fourth cluster
mtp.scatter(kmeans.cluster_centers_[:], kmeans.cluster_centers_[:],s = 150, c = 'yellow', label = 'Centroids')
mtp.title('Lev Distance Calculation on Test Dataset')
mtp.xlabel('Lev Distance')
mtp.ylabel('Lev Distance')
mtp.legend()
mtp.show()
```

Program files

1. **DatasetProcessingEngine.java** – The **DatasetProcessingEngine** class is responsible for uploading the dataset to Google Cloud Storage. The program iterates through all the files in the **Dataset/Train** and **Dataset/Test** folder and uploads the files to the bucket provided on Google Cloud Storage. I have uploaded the same code on GitLab. Here is the link to that code - https://git.cs.dal.ca/dashah/csci-5410-f2021-b00857606-dhrumil-amish-shah/-/blob/main/assignment_4_code/part_b/part_b_code/src/main/java/DatasetProcessingEngine.java. The working of the code is explained within the code itself.

```
import com.google.auth.Credentials;
import com.google.auth.oauth2.GoogleCredentials;
import com.google.cloud.storage.BlobId;
import com.google.cloud.storage.BlobInfo;
import com.google.cloud.storage.BucketInfo;
import com.google.cloud.storage.Storage;
import com.google.cloud.storage.StorageOptions;

import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;

/**
 * {@code DatasetProcessingEngine} class uploads the training dataset from location Dataset\Train and
 * testing dataset from location Dataset\Test to {@code SOURCE_BUCKET} bucket folder.
 *
 * @author Dhrumil Amish Shah (B00857606|dh416386@dal.ca)
 */
public final class DatasetProcessingEngine {
  private static final String SOURCE_BUCKET;
  private static final String CREDENTIALS_JSON_FILE;
  private static final String GOOGLE_PROJECT_ID;
  private static final String DATASET_FILES_PATH;

  static {
    SOURCE_BUCKET = "sourcedatab00857606";
    CREDENTIALS_JSON_FILE = "csci5410-a4-partb-e37656a2fcca.json";
    GOOGLE_PROJECT_ID = "csci5410-a4-partb";
    DATASET_FILES_PATH = "Dataset" + File.separator + "Test" + File.separator;
  }

  /**
   * Uploads the dataset (training and testing dataset) on the Google Cloud Storage.
```

```
   *
   * @param storage storage on which the dataset is to be uploaded.
   */
  private void uploadDatasetToGCSBucket(final Storage storage) {
    try {
      final File[] datasetFiles = new File(DATASET_FILES_PATH).listFiles();
      if (datasetFiles == null) {
        return;
      }
      for (File datasetCurrentFile : datasetFiles) {
        final String datasetFileName = datasetCurrentFile.getName();
        final BlobId blobId = BlobId.of(SOURCE_BUCKET, datasetFileName);
        final BlobInfo blobInfo = BlobInfo.newBuilder(blobId).build();
        System.out.println(DATASET_FILES_PATH + datasetFileName);
        storage.create(blobInfo, Files.readAllBytes(Paths.get(DATASET_FILES_PATH +
datasetFileName)));
        System.out.println("File " + datasetFileName + " uploaded to bucket " +
SOURCE_BUCKET + ".");
      }
    } catch (final IOException e) {
      System.out.println("Method uploadDatasetToGCSBucket: " + e.getMessage());
      e.printStackTrace();
    }
  }

  /**
   * Establishes connection to the Google Cloud Storage.
   *
   * @return storage object to which the connection is established.
   */
  private Storage establishGCSConnection() {
    try {
      final Credentials credentials = GoogleCredentials.fromStream(new
FileInputStream(CREDENTIALS_JSON_FILE));
      return
StorageOptions.newBuilder().setCredentials(credentials).setProjectId(GOOGLE_PROJECT_ID)
.build().getService();
    } catch (final IOException e) {
      System.out.println("Method establishGCPConnection: " + e.getMessage());
      e.printStackTrace();
    }
    return null;
  }

  /**
   * Processes the dataset.
```

```
   * Also, it creates the bucket if does not exist already.
   */
  public void processDataset() {
    final Storage storage = establishGCSConnection();
    if (storage != null) {
      if (storage.get(SOURCE_BUCKET) == null) {
        storage.create(BucketInfo.of(SOURCE_BUCKET));
        System.out.println("Bucket " + SOURCE_BUCKET + " created successfully.");
      } else if (!storage.get(SOURCE_BUCKET).exists()) {
        storage.create(BucketInfo.of(SOURCE_BUCKET));
        System.out.println("Bucket " + SOURCE_BUCKET + " created successfully.");
      } else {
        System.out.println("Bucket " + SOURCE_BUCKET + " exists already.");
      }
      uploadDatasetToGCSBucket(storage);
    }
  }
}
```

2. **DatasetProcessingEngineTest.java** – The **DatasetProcessingEngineTest** class is responsible for testing the **DatasetProcessingEngine** class. It executes the public method of **DatasetProcessingEngine** class. I have uploaded the code on GitLab. The link to the code is https://git.cs.dal.ca/dashah/csci-5410-f2021-b00857606-dhrumil-amish-shah/-/blob/main/assignment_4_code/part_b/part_b_code/src/main/java/DatasetProcessingEngineTest.java.

```
/**
 * {@code DatasetProcessingEngineTest} class tests the {@code DatasetProcessingEngine}.
 *
 * @author Dhrumil Amish Shah (B00857606|dh416386@dal.ca)
 */
public final class DatasetProcessingEngineTest {
  public static void main(String[] args) {
    new DatasetProcessingEngine().processDataset();
  }
}
```

3. **VectorGenerator.java** – The **VectorGenerator** class is responsible for generating the **trainVectors.csv** and **testVectors.csv**. The link to the code is https://git.cs.dal.ca/dashah/csci-5410-f2021-b00857606-dhrumil-amish-shah/-/blob/main/assignment_4_code/part_b/part_b_code/src/main/java/com/vectorgenerator/VectorGenerator.java. I have explained the working of the below code using the inline comments.

   The code reference for Levenshtein distance can be found on the Baeldung website [10]. The code reference for converting to CSV can be found on the Baeldung website [11]. A list of stop words can be found on the GitHub website [12].

```
package com.vectorgenerator;

import com.google.cloud.functions.BackgroundFunction;
import com.google.cloud.functions.Context;
import com.google.cloud.storage.BlobId;
import com.google.cloud.storage.BlobInfo;
import com.google.cloud.storage.BucketInfo;
import com.google.cloud.storage.Storage;
import com.google.cloud.storage.StorageOptions;

import java.util.logging.Logger;

import com.google.cloud.storage.Blob;
import com.google.api.gax.paging.Page;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.Stream;

/**
 * {@code VectorGenerator} class contains the implementation for the generateVector Google Cloud Function.
 * {@code VectorGenerator} class implementation works for both training dataset and testing dataset.
 * <p>
 * While working with training dataset, comment variables {@code TEST_VECTOR_CSV} and {@code TEST_DATA_BUCKET}
 * and use variables {@code TRAIN_VECTOR_CSV} and {@code TRAIN_DATA_BUCKET}.
 * <p>
 * While working with test dataset, comment variables {@code TRAIN_VECTOR_CSV} and {@code TRAIN_DATA_BUCKET}
 * and use variables {@code TEST_VECTOR_CSV} and {@code TEST_DATA_BUCKET}.
 * <p>
 * The working of {@code VectorGenerator} class is as below:
 * <p>
 * Step 1: Stop
 * <p>
 * Step 2: Read all the files (either training dataset files or test dataset files) uploaded to the bucket one at a time.
 * Each file is read as a blob (i.e. {@code Blob blob})
 * <p>
 * Step 3: For each file, first read the file content as {@code String fileContent} and filter out the special characters,
```

```
 * extra spaces and line breaks using regex.
 * <p>
 * Step 4: Split the file content from {@code String fileContent} into list of words (i.e., {@code
List<String>} wordsList)
 * and remove all stop words from the list (i.e., remove all words matching in list {@code
List<String> STOP_WORDS_LIST}).
 * <p>
 * Step 5: Calculate Levenshtein distance for the list of words by iterating the {@code
List<String> wordsList}.
 * Levenshtein distance is calculated for the current word and next word and the calculated
output is stored inside
 * {@code List<String[]> dataLinesList} with each {@code String[]} containing current word at
0th index, next word
 * at 1st index and Levenshtein distance between the current word and the next word at 3rd
index.
 * <p>
 * Step 6: After all files are processed, convert {@code List<String> wordsList} into {@code
String} with content as
 * {@code
current_word_1,next_word_1,lev_distance_1\ncurrent_word_1,next_word_1,lev_distance_1}
and is store it in
 * {@code StringBuilder dataLinesStringBuilder}.
 * <p>
 * Step 7: Convert the {@code StringBuilder dataLinesStringBuilder} into {@code String} and
store it in a CSV file
 * and upload it to the bucket.
 * <p>
 * - For train dataset, store the {@code String} content in {@code TRAIN_VECTOR_CSV} and
upload it
 * to {@code TRAIN_DATA_BUCKET}.
 * - For test dataset, store the {@code String} content in {@code TEST_VECTOR_CSV} and
upload it
 * to {@code TEST_DATA_BUCKET}.
 * <p>
 * Levenshtein distance calculation code reference
 * https://www.baeldung.com/java-levenshtein-distance
 * <p>
 * Stop Words List Reference
 * https://gist.github.com/sebleier/554280
 * <p>
 * Convert to code CSV reference
 * https://www.baeldung.com/java-csv
 *
 * @author Dhrumil Amish Shah (B00857606|dh416386@dal.ca)
 */
public final class VectorGenerator implements
```

```
BackgroundFunction<VectorGenerator.GCSEvent> {
  private static final Logger LOGGER;
  private static final List<String> STOP_WORDS_LIST;
  // private static final String TRAIN_VECTOR_CSV;
  // private static final String TRAIN_DATA_BUCKET;
  private static final String TEST_VECTOR_CSV;
  private static final String TEST_DATA_BUCKET;

  static {
    LOGGER = Logger.getLogger(VectorGenerator.class.getName());
    // Reference: https://gist.github.com/sebleier/554280
    STOP_WORDS_LIST = Arrays.asList("i", "me", "my",
        "myself", "we", "our", "ours", "ourselves", "you", "your", "yours", "yourself",
        "yourselves", "he", "him", "his", "himself", "she", "her", "hers", "herself",
        "it", "its", "itself", "they", "them", "their", "theirs", "themselves", "what",
        "which", "who", "whom", "this", "that", "these", "those", "am", "is", "are",
        "was", "were", "be", "been", "being", "have", "has", "had", "having", "do",
        "does", "did", "doing", "a", "an", "the", "and", "but", "if", "or", "because",
        "as", "until", "while", "of", "at", "by", "for", "with", "about", "against",
        "between", "into", "through", "during", "before", "after", "above", "below",
        "to", "from", "up", "down", "in", "out", "on", "off", "over", "under", "again",
        "further", "then", "once", "here", "there", "when", "where", "why", "how", "all",
        "any", "both", "each", "few", "more", "most", "other", "some", "such", "no",
        "nor", "not", "only", "own", "same", "so", "than", "too", "very", "s", "t",
        "can", "will", "just", "don", "should", "now");
    // TRAIN_VECTOR_CSV = "trainVectors.csv";
    // TRAIN_DATA_BUCKET = "traindatab00857606";
    TEST_VECTOR_CSV = "testVectors.csv";
    TEST_DATA_BUCKET = "testdatab00857606";
  }

  @Override
  public void accept(final GCSEvent event, final Context context) {
    LOGGER.info("com.vectorgenerator.VectorGenerator:  Event name: " + event.name);
    LOGGER.info("com.vectorgenerator.VectorGenerator:  Bucket name: " + event.bucket);

    final Storage storage = StorageOptions.getDefaultInstance().getService();
    final Page<Blob> blobs = storage.get(event.bucket).list();

    final List<String[]> dataLinesList = new ArrayList<>();
    for (final Blob blob : blobs.iterateAll()) {
      LOGGER.info("com.vectorgenerator.VectorGenerator:  Blob name: " + blob.getName());
      final String generalRegex = "[,.\\-_'\"(){}?!@#$%^&*\\[\\]]+";
      final String spaceRegex = "[\\n\\r\\t]+";
      final String fileContent = new String(blob.getContent())
          .replaceAll(generalRegex, "")
```

```
        .replaceAll(spaceRegex, " ");
    final List<String> wordsList = new
ArrayList<>(Arrays.asList(fileContent.toLowerCase().split(" ")));
    wordsList.removeAll(STOP_WORDS_LIST);

    LOGGER.info("com.vectorgenerator.VectorGenerator:  List of words after removing stop
words.");
    final StringBuilder wordsListBuilder = new StringBuilder();
    for (final String word : wordsList) {
      wordsListBuilder.append(word).append(" ");
    }
    LOGGER.info("com.vectorgenerator.VectorGenerator: " + wordsListBuilder.toString());

    for (int i = 0; i < wordsList.size() - 1; i++) {
      final String currentWord = wordsList.get(i);
      final String nextWord = wordsList.get(i + 1);
      final int distance = calculate(currentWord, nextWord);
      dataLinesList.add(new String[]{currentWord, nextWord, String.valueOf(distance)});
    }
   }

   final StringBuilder dataLinesStringBuilder = new StringBuilder();
   for (final String[] dataLine : dataLinesList) {
    dataLinesStringBuilder.append(convertToCSV(dataLine)).append("\n");
   }

   uploadCSVFileToGCSBucket(dataLinesStringBuilder.toString(), storage);
  }

  /**
   * Reference: https://www.baeldung.com/java-levenshtein-distance
   */
  private int costOfSubstitution(final char a, final char b) {
   return a == b ? 0 : 1;
  }

  /**
   * Reference: https://www.baeldung.com/java-levenshtein-distance
   */
  private int min(final int... numbers) {
   return Arrays.stream(numbers).min().orElse(Integer.MAX_VALUE);
  }

  /**
   * Reference: https://www.baeldung.com/java-csv
   */
```

```
  private int calculate(final String x, final String y) {
    final int[][] dp = new int[x.length() + 1][y.length() + 1];
    for (int i = 0; i <= x.length(); i++) {
      for (int j = 0; j <= y.length(); j++) {
        if (i == 0) {
          dp[i][j] = j;
        } else if (j == 0) {
          dp[i][j] = i;
        } else {
          dp[i][j] = min(dp[i - 1][j - 1] +
                costOfSubstitution(x.charAt(i - 1), y.charAt(j - 1)),
              dp[i - 1][j] + 1,
              dp[i][j - 1] + 1);
        }
      }
    }
    return dp[x.length()][y.length()];
  }

  /**
   * Reference: https://www.baeldung.com/java-csv
   */
  private String escapeSpecialCharacters(String data) {
    String escapedData = data.replaceAll("\\R", " ");
    if (data.contains(",") || data.contains("\"") || data.contains("'")) {
      data = data.replace("\"", "\"\"");
      escapedData = "\"" + data + "\"";
    }
    return escapedData;
  }

  /**
   * Reference: https://www.baeldung.com/java-levenshtein-distance
   */
  private String convertToCSV(String[] dataLine) {
    return Stream.of(dataLine)
        .map(this::escapeSpecialCharacters)
        .collect(Collectors.joining(","));
  }

  private void uploadCSVFileToGCSBucket(final String csvFileContent, final Storage storage) {
    if (storage.get(TEST_DATA_BUCKET) == null) {
      storage.create(BucketInfo.of(TEST_DATA_BUCKET));
      LOGGER.info("com.vectorgenerator.VectorGenerator: " + "Bucket " +
TEST_DATA_BUCKET + " created successfully.");
    } else if (!storage.get(TEST_DATA_BUCKET).exists()) {
```

```
    storage.create(BucketInfo.of(TEST_DATA_BUCKET));
    LOGGER.info("com.vectorgenerator.VectorGenerator: " + "Bucket " +
TEST_DATA_BUCKET + " created successfully.");
   } else {
    LOGGER.info("com.vectorgenerator.VectorGenerator: " + "Bucket " +
TEST_DATA_BUCKET + " exists already.");
   }
   final BlobInfo blobInfo = BlobInfo.newBuilder(BlobId.of(TEST_DATA_BUCKET,
TEST_VECTOR_CSV)).build();
   storage.create(blobInfo, csvFileContent.getBytes());
   LOGGER.info("com.vectorgenerator.VectorGenerator: File " + TEST_VECTOR_CSV + "
uploaded to bucket " + TEST_DATA_BUCKET + ".");
  }

  public static class GCSEvent {
   final String bucket;
   final String name;
   final String metaGeneration;

   public GCSEvent(final String bucket,
                   final String name,
                   final String metaGeneration) {
    this.bucket = bucket;
    this.name = name;
    this.metaGeneration = metaGeneration;
   }
  }
}
```

GitLab Source Code

The source code for **Part B** is available on GitLab URL given below:
https://git.cs.dal.ca/dashah/csci-5410-f2021-b00857606-dhrumil-amish-shah/-/tree/main/assignment_4_code/part_b.

References

[1] Google, "Cloud Storage," Google, [Online]. Available: https://cloud.google.com/storage. [Accessed 15 November 2021].

[2] Google, "Cloud Functions," Google, [Online]. Available: https://cloud.google.com/functions. [Accessed 15 November 2021].

[3] Wikipedia, "Levenshtein Distance," Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Levenshtein_distance. [Accessed 15 November 2021].

[4] Wikipedia, "k-means clustering," Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Levenshtein_distance. [Accessed 15 November 2021].

[5] Google, "Vertex AI," Google, [Online]. Available: https://cloud.google.com/vertex-ai/docs/reference/rest. [Accessed 11 November 2021].

[6] Google, "Google Cloud," Google, [Online]. Available: https://cloud.google.com/gcp. [Accessed 15 November 2021].

[7] Google, "Operation Suits," Google, [Online]. Available: https://cloud.google.com/logging. [Accessed 15 November 2021].

[8] Wikipedia, "Elbow method (clustering)," Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Elbow_method_(clustering). [Accessed 15 November 2021].

[9] javaTpoint, "K-Means Clustering Algorithm," javaTpoint, [Online]. Available: https://www.javatpoint.com/k-means-clustering-algorithm-in-machine-learning. [Accessed 15 November 2021].

[10] Baeldung, "How to calculate Levenshtein Distance in Java?," Baeldung, [Online]. Available: https://www.baeldung.com/java-levenshtein-distance. [Accessed 13 November 2021].

[11] Baeldung, "How to Write to a CSV File in Java," Baeldung, [Online]. Available: https://www.baeldung.com/java-csv. [Accessed 13 November 2021].

[12] GitHub, "NLTK's list of english stopwords," GitHub, [Online]. Available: https://gist.github.com/sebleier/554280. [Accessed 13 November 2021].