GCP steps:

Steps 0: Create a Github repo

https://github.com/cs-cloud-store/assessment

Step 1: Create a Cloud account

Google Cloud account with $300 free trial band set up billing.

Step 2: Upload file to Cloud Storage and create service accounts

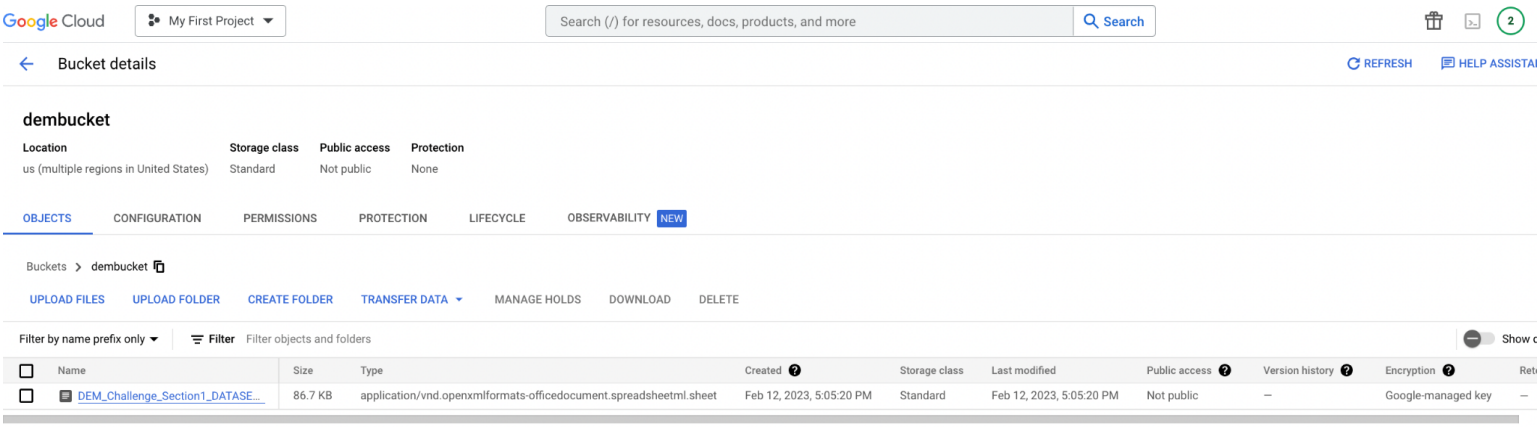
Service accounts code:

```
gcloud projects add-iam-policy-binding $GOOGLE_CLOUD_PROJECT \
--member
serviceAccount:https-console-cloud-google-com@hale-brook-377621.iam.gserviceaccount.com
\
--role roles/dlp.admin

gcloud projects add-iam-policy-binding $GOOGLE_CLOUD_PROJECT \
--member
serviceAccount:https-console-cloud-google-com@hale-brook-377621.iam.gserviceaccount.com
\
--role roles/dlp.serviceAgent

gcloud projects add-iam-policy-binding $GOOGLE_CLOUD_PROJECT \
--member serviceAccount:service-`gcloud projects list
--filter="PROJECT_ID:$GOOGLE_CLOUD_PROJECT"
--format="value(PROJECT_NUMBER)"`@dlp-api.iam.gserviceaccount.com \
--role roles/viewer
```

Cloud Storage bucket

GSUTIL to have the data uploaded into Storage for project data.

gsutil mb gs://dembucket



We'll need to create 3 Cloud Storage buckets to store our data:

- Quarantine bucket: our data will initially be uploaded here.
- Sensitive data bucket: the data determined by the DLP API to be sensitive will be moved here.
- Non-sensitive data bucket: the data determined by the DLP API not to be sensitive will be moved here

We can use the gsutil command to create all three of our buckets in one swoop:

#Actual commands to run:

gsutil mb gs://dem_quarantine_bucket \
gs://sensitive_dem_bucket \
gs://non_sensitive_dem_bucket

```
chandnis3010@cloudshell:~ (hale-brook-377621)$ gsutil mb gs://dem_quarantine_bucket \
gs://sensitive_dem_bucket \
gs://non_sensitive_dem_bucket
Creating gs://dem_quarantine_bucket/...
Creating gs://sensitive_dem_bucket/...
Creating gs://non_sensitive_dem_bucket/...
chandnis3010@cloudshell:~ (hale-brook-377621)$
```

We will use Cloud DLP on the Storage bucket. For future, we can keep 2 copies.

- one sensitive data and
- one non-sensitive data.

Cloud Pub/Sub provides many to many asynchronous messaging between applications. A publisher will create a message and publish it to a feed of messages called a topic. A subscriber will receive these messages by way of a subscription. Based on that subscription, in our case, we'll have a Cloud Function move files to their respective buckets after a DLP job runs.

First, let's create a topic. A message will be published here each time a file is added to our quarantine storage bucket. We'll name it ‘classify-topic'

gcloud pubsub topics create classify-topic

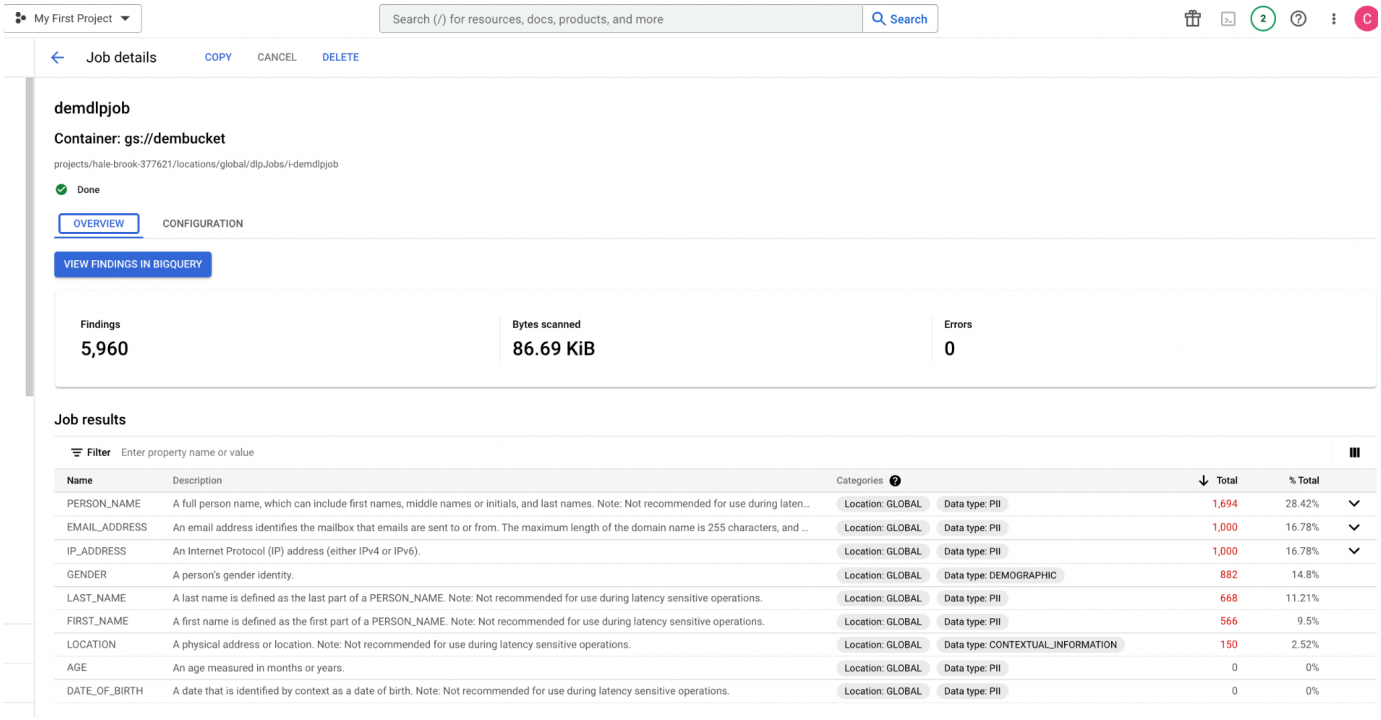
A subscription will be notified when the topic publishes a message. Let's create a pubsub subscription named ‘classify-sub':

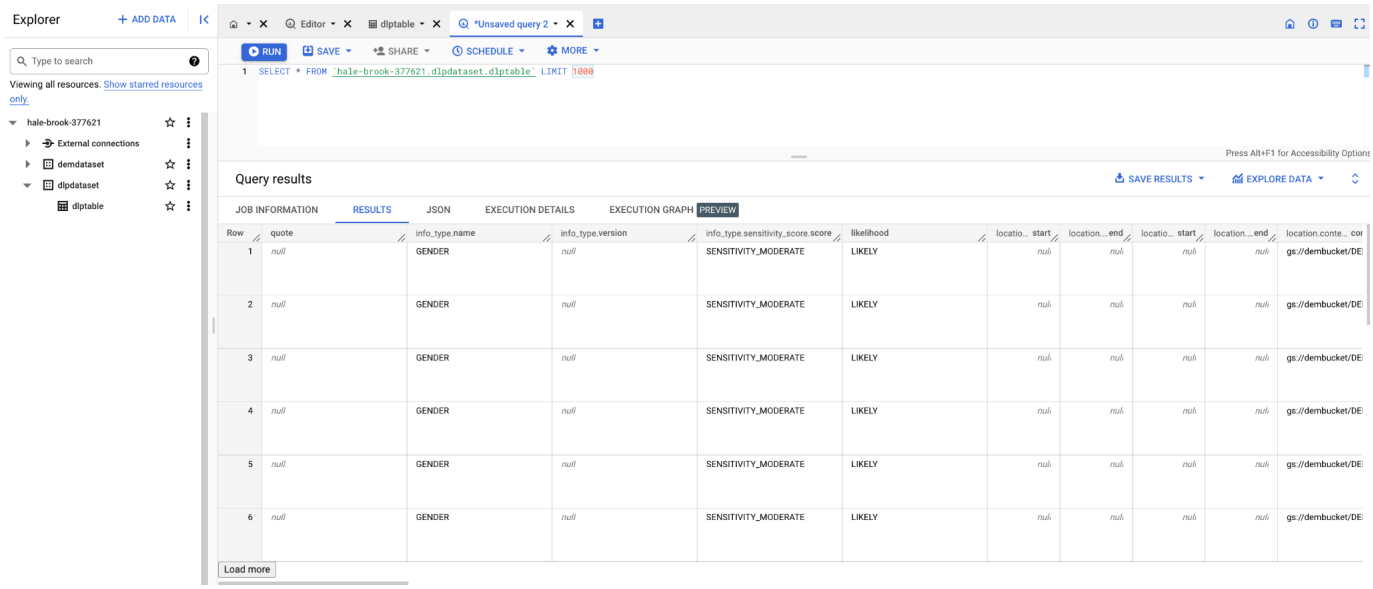
gcloud pubsub subscriptions create classify-sub --topic classify-topic

Cloud DLP for Cloud Storage

For now we will be scanning your Cloud Storage buckets with Cloud DLP, and you can set up this new scan job to be run regularly.

- Browse to Cloud Storage in the GCP console, then click on the three-dot menu icon to the right of a relevant bucket. Click on the “Scan with Data Loss Prevention” menu item
- Complete the Cloud DLP scan creation by clicking the “Create” button or, optionally, specify custom configurations such as what info types to inspect for, what sampling options to use, what actions to take, and more.
- Once Cloud DLP scans are completed, you’ll get emails with links to the “Scan details” page, where you can analyze findings and take further actions. From there, click on “**View Findings in BigQuery**” to analyze the results.





Optional: Python code to use DLP (Data Loss prevention as this is sensitive data like gender, email, ip address)

```
import google.cloud.dlp
from google.cloud import dlp_v2
```

```
# Instantiates a DLP client
dlp = dlp_v2.DlpServiceClient()

# Define the info types to be de-identified
info_types = [{"name": "PERSON_NAME"}, {"name": "EMAIL_ADDRESS"}, {"name":
"IP_ADDRESS"}]

# Define the de-identification transformation
deidentify_config = {
    "record_transformations": {
        "field_transformations": [
            {
                "fields": [{"name": "first_name"}, {"name": "last_name"}, {"name": "email"}],
                "primitive_transformation": {
                    "character_mask_config": {
                        "masking_character": "*",
                        "number_to_mask": 0.6,
                        "characters_to_ignore": [{"characters_to_skip": ".,;:-'\"()"}],
                    }
                },
            },
            {
                "fields": [{"name": "ip_address"}],
                "primitive_transformation": {
                    "replace_with_info_type_config": {"info_type": {"name": "IP_ADDRESS"}},
                },
            },
        ]
    },
    "info_type_transformations": {"transformations": [{"info_types": info_types, "action":
{"redact": {}}}]},
}

# Define the re-identification transformation
reidentify_config = {
    "record_transformations": {
        "field_transformations": [
            {
                "fields": [{"name": "first_name"}, {"name": "last_name"}, {"name": "email"}],
```

```
            "primitive_transformation": {"replace_with_info_type_config": {"info_type":
{"name": "PERSON_NAME"}}},
          },
          {"fields": [{"name": "ip_address"}], "primitive_transformation":
{"replace_with_info_type_config": {"info_type": {"name": "IP_ADDRESS"}}}},
       ]
   },
   "info_type_transformations": {"transformations": [{"info_types": info_types, "action":
{"replaceWithInfoTypeConfig": {"infoType": {"name": "PII"}}}}]},
}

# Define the dataset
dataset = [
   {"id": 1, "first_name": "John", "last_name": "Doe", "email": "john.doe@example.com",
"gender": "male", "ip_address": "192.168.0.1"},
   {"id": 2, "first_name": "Jane", "last_name": "Doe", "email": "jane.doe@example.com",
"gender": "female", "ip_address": "192.168.0.2"},
]

# De-identify the dataset
deidentify_request = dlp_v2.DeidentifyContentRequest(
   parent=f"projects/{PROJECT_ID}/locations/{LOCATION}",
   deidentify_config=deidentify_config,
   item={"table": {"headers": [{"name": "id"}, {"name": "first_name"}, {"name": "last_name"},
{"name": "email"}, {"name": "gender"}, {"name": "ip_address"}], "rows": [{"values":
[str(d[key]) for key in ["id", "first_name", "last_name", "email", "gender", "ip_address"]]} for
d in dataset]}},
)
deidentified_response = dlp.deidentify_content(deidentify_request)
deidentified_dataset = [{header["name"]: row["values"][i] for i, header in
enumerate(deidentified_response.item.table.headers)} for row in
deidentified_response.item.table.rows]

# Re-identify the dataset
reidentify_request = dlp_v2.ReidentifyContentRequest(
parent=f"projects/{PROJECT_ID}/locations/{LOCATION}",
reidentify_config=reidentify_config,
inspect_config={"info_types": info_types},
```

```
item={"table": {"headers": [{"name": "id"}, {"name": "first_name"}, {"name": "last_name"}, {"name": "email"}, {"name": "gender"}, {"name": "ip_address"}], "rows": [{"values": [str(d[key]) for key in ["id", "first_name", "last_name", "email", "gender", "ip_address"]]} for d in deidentified_dataset]}},
)
reidentified_response = dlp.reidentify_content(reidentify_request)
reidentified_dataset = [{header["name"]: row["values"][i] for i, header in enumerate(reidentified_response.item.table.headers)} for row in reidentified_response.item.table.rows]

#Print the original and re-identified datasets to test the code
print("Original dataset:")
print(dataset)
print("Re-identified dataset:")
print(reidentified_dataset)
```
css

Step 3: Python ETL to load data from Cloud Storage to Cloud SQL

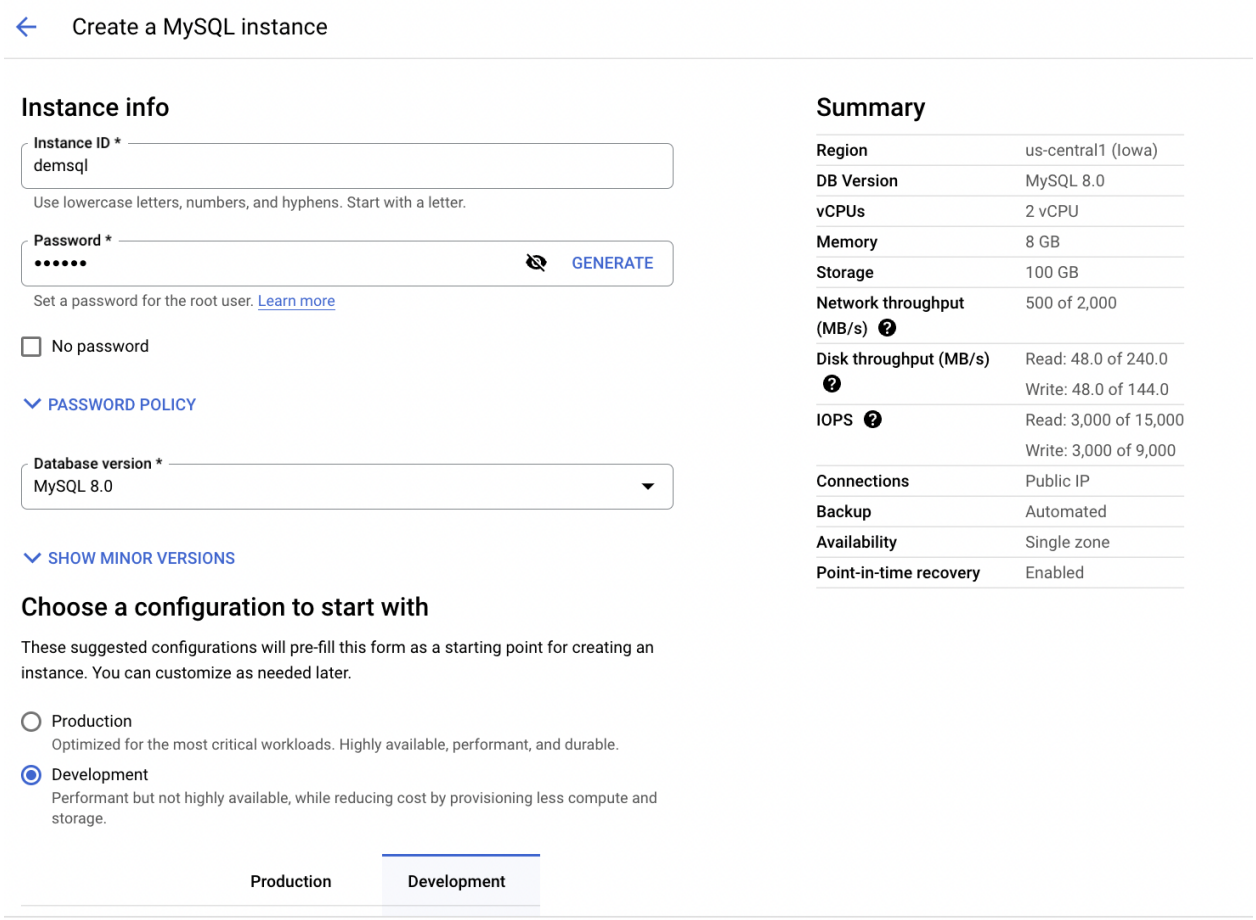
```
import pandas as pd
import mysql.connector
from google.cloud import storage

# Set up connection to Google Cloud Storage
client = storage.Client()
bucket = client.get_bucket('your-bucket-name')

# Load data from CSV file in GCS into a pandas dataframe
blob = bucket.blob('your-file.csv')
data = blob.download_as_string()
df = pd.read_csv(io.BytesIO(data))

# Transform data (if needed)
# For example, you could concatenate first_name and last_name into a new column called full_name:
df['full_name'] = df['first_name'] + ' ' + df['last_name']
```

```
# Set up connection to MySQL Cloud SQL
cnx = mysql.connector.connect(user='your-db-user', password='your-db-password',
                              host='your-db-host', database='your-db-name')

# Create cursor and table if not exists
cursor = cnx.cursor()
create_table_query = '''
    CREATE TABLE IF NOT EXISTS customers (
        id INT PRIMARY KEY,
        first_name VARCHAR(50),
        last_name VARCHAR(50),
        email VARCHAR(255),
        gender VARCHAR(10),
        ip_address VARCHAR(50),
        full_name VARCHAR(100)
    )
'''
cursor.execute(create_table_query)
cnx.commit()

# Load data into MySQL Cloud SQL
for index, row in df.iterrows():
    insert_query = '''
        INSERT INTO customers (id, first_name, last_name, email, gender, ip_address, full_name)
        VALUES (%s, %s, %s, %s, %s, %s, %s)
    '''
    values = (row['id'], row['first_name'], row['last_name'], row['email'], row['gender'],
row['ip_address'], row['full_name'])
    cursor.execute(insert_query, values)
    cnx.commit()

# Close cursor and connection
cursor.close()
cnx.close()
```

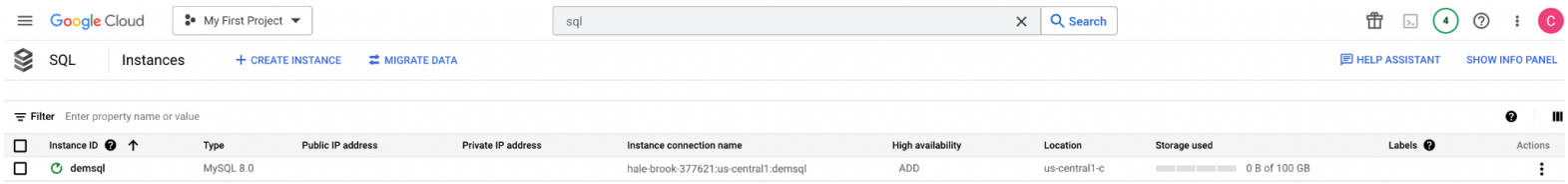
Step 4: Cloud SQL (MySQL) database connection for the DEM file

To import data to a Cloud SQL instance using a CSV file

1. Download the file shared by Deloitte as “CSV”
2. Enable “Compute Engine API”
3. Create a development MySQL 8.0 Instance



Id: demsql
Password: demsql



Python code to load data from Cloud SQL to BigQuery (as data will grow, they will need to be in Data Warehouse)

```
import pandas as pd
import mysql.connector
from google.cloud import bigquery
from google.cloud import storage

# Set up connection to Google Cloud Storage
client = storage.Client()
bucket = client.get_bucket('your-bucket-name')

# Load data from CSV file in GCS into a pandas dataframe
blob = bucket.blob('your-file.csv')
data = blob.download_as_string()
df = pd.read_csv(io.BytesIO(data))

# Transform data (if needed)
# For example, you could concatenate first_name and last_name into a new column called full_name:
df['full_name'] = df['first_name'] + ' ' + df['last_name']

# Set up connection to GCP Cloud SQL
cnx = mysql.connector.connect(user='your-db-user', password='your-db-password',
                              host='your-db-host', database='your-db-name')

# Load data into a pandas dataframe
df = pd.read_sql('SELECT * FROM customers', con=cnx)

# Set up connection to BigQuery
client = bigquery.Client()
table_ref = client.dataset('your_dataset_name').table('your_table_name')

# Create job configuration and load data into BigQuery
```

```
job_config = bigquery.LoadJobConfig()
job_config.write_disposition = bigquery.WriteDisposition.WRITE_TRUNCATE
job_config.source_format = bigquery.SourceFormat.CSV
job_config.schema = [
    bigquery.SchemaField('id', 'INTEGER'),
    bigquery.SchemaField('first_name', 'STRING'),
    bigquery.SchemaField('last_name', 'STRING'),
    bigquery.SchemaField('email', 'STRING'),
    bigquery.SchemaField('gender', 'STRING'),
    bigquery.SchemaField('ip_address', 'STRING'),
    bigquery.SchemaField('full_name', 'STRING')
]
job = client.load_table_from_dataframe(df, table_ref, job_config=job_config)
job.result()  # Wait for job to complete

# Close connection to GCP Cloud SQL
cnx.close()
```

Step 5: Terraform code to automate process from Storage to Cloud SQL uploading

```
# Configure provider
provider "google" {
  project = "hale-brook-377621"
  region  = "your-region"
  zone    = "your-zone"
}

# Configure storage bucket
resource "google_storage_bucket" "data_bucket" {
  name = "your-bucket-name"
}

# Configure Cloud SQL instance
resource "google_sql_database_instance" "db_instance" {
  name             = "your-instance-name"
  region           = "your-region"
```

```
  database_version = "MYSQL_5_7"
  settings {
    tier = "db-f1-micro"
  }
}

# Configure Cloud SQL database and user
resource "google_sql_database" "db" {
  name     = "your-db-name"
  instance = google_sql_database_instance.db_instance.name
}

resource "google_sql_user" "db_user" {
  name     = "your-db-user"
  password = "your-db-password"
  instance = google_sql_database_instance.db_instance.name
  host     = "%"
}

# Configure Cloud Function to run ETL code
resource "google_cloudfunctions_function" "etl_function" {
  name        = "your-function-name"
  description = "ETL function that ingests, transforms, and loads data from GCS to Cloud
SQL"

  source_archive_bucket = google_storage_bucket.data_bucket.name
  source_archive_object = "etl_function.zip"

  entry_point = "main"

  runtime = "python37"

  environment_variables = {
    SQL_USER     = google_sql_user.db_user.name
    SQL_PASSWORD = google_sql_user.db_user.password
    SQL_HOST     = google_sql_database_instance.db_instance.ip_address
    SQL_DATABASE = google_sql_database.db.name
    GCS_BUCKET   = google_storage_bucket.data_bucket.name
  }
```

```
  timeout    = "180s"
  available_memory_mb = 256

  trigger_http = true
}

# Configure Cloud Scheduler job to run Cloud Function daily
resource "google_cloud_scheduler_job" "etl_job" {
  name        = "your-job-name"
  description = "Schedule ETL function to run daily at 7AM"

  schedule = "0 7 * * *"

  target_type = "google_cloudfunctions_function"
  target_http_method = "POST"
  target_uri = google_cloudfunctions_function.etl_function.https_trigger_url
}
```

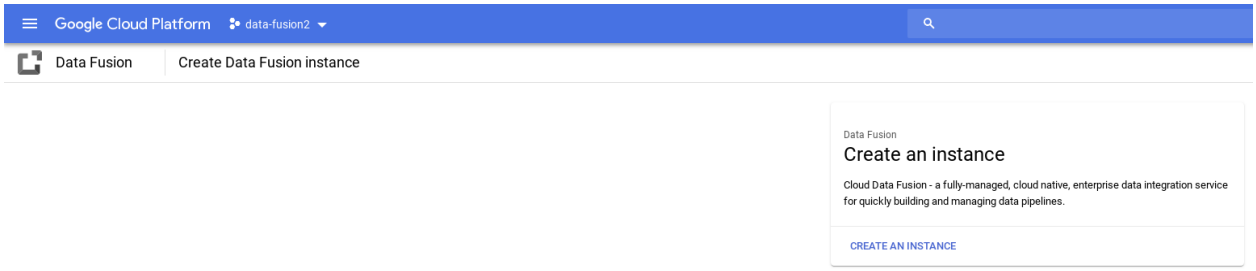
Step 6: ETL to load data from Cloud Storage to BigQuery

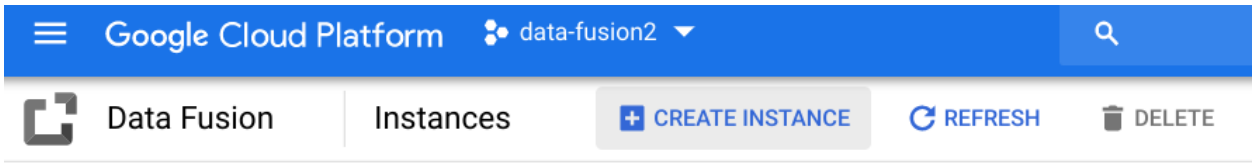
We will build a data integration pipeline in Cloud Data Fusion for loading, transforming and masking healthcare data in bulk.

Data Fusion

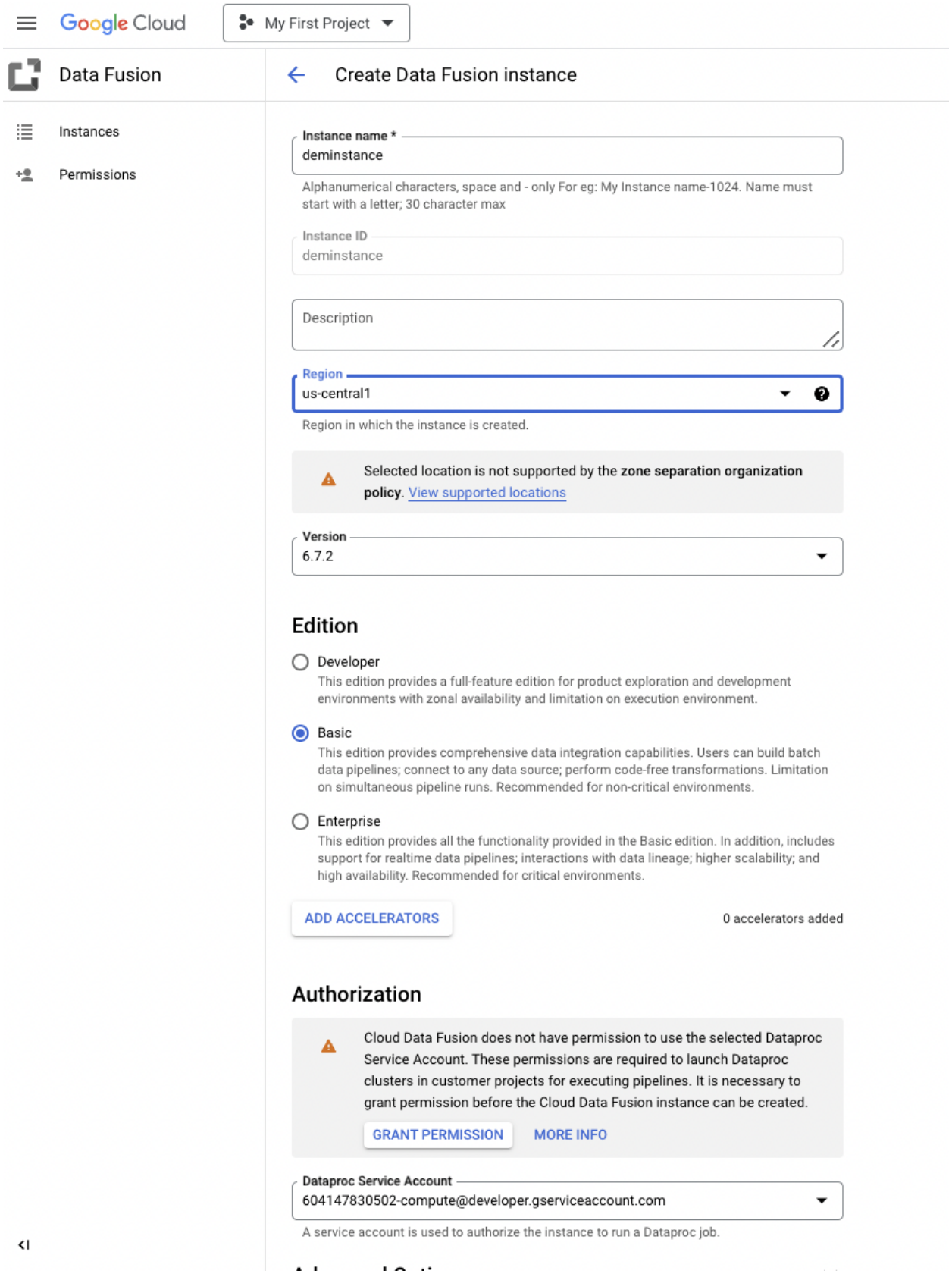
```
export PROJECT_ID=hale-brook-377621
export BUCKET_NAME=dembucket
export DATASET_ID=demdataset
```

1. Select Data Fusion from the left menu, then click the CREATE AN INSTANCE button in the middle of the page (1st creation), or click the CREATE INSTANCE button at the top menu (additional creation).



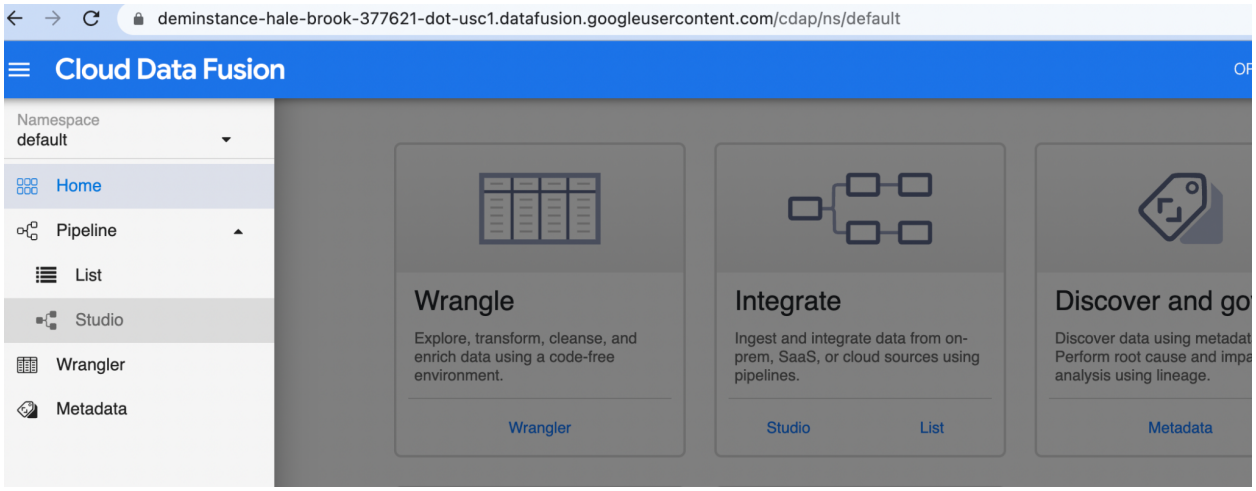


3. Provide the instance name. Select Enterprise or Basic.

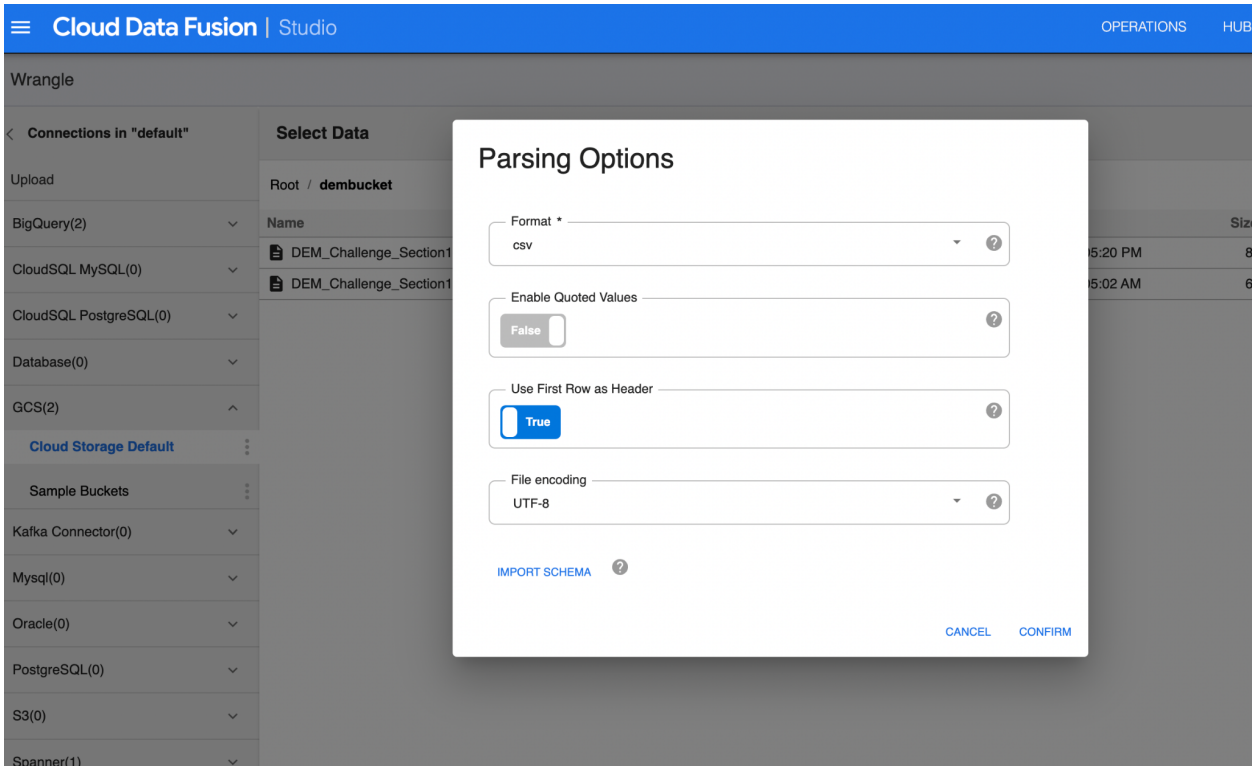


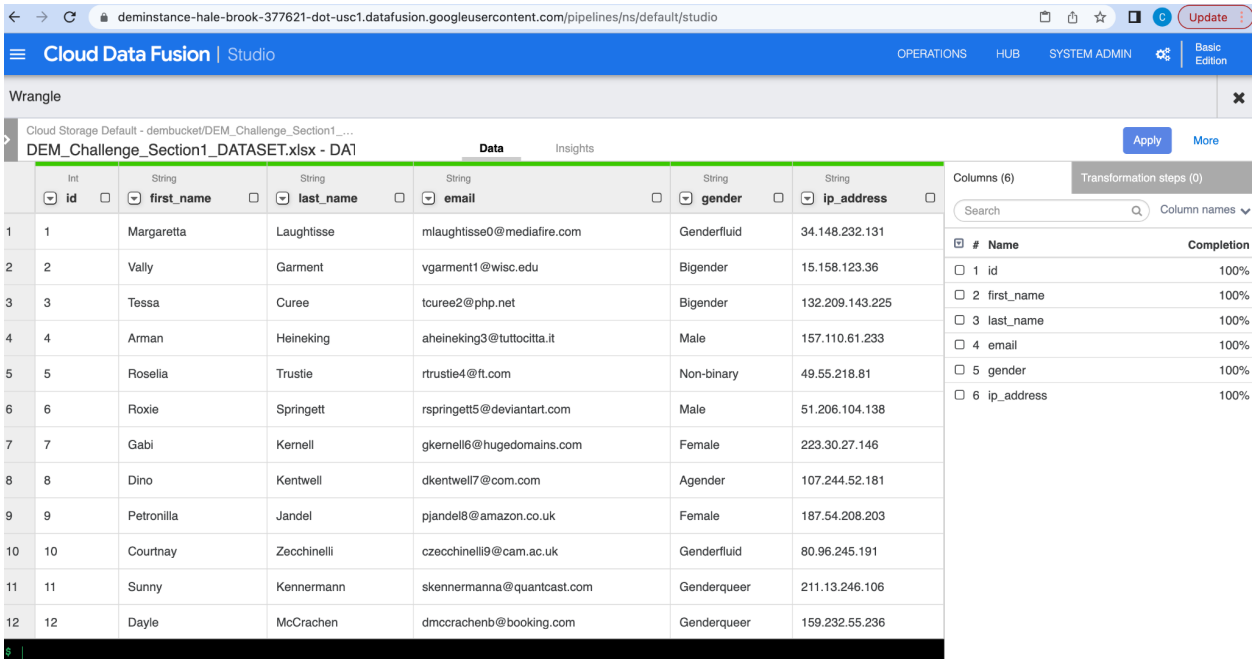
4.
5. Copy the service account.

6. On the IAM permissions page, we will now add the service account as a new member and grant it the Cloud Data Fusion API Service Agent role. Click the Add button, then paste the "service account" in the New members field and select Service Management -> Cloud Data Fusion API Server Agent role.
7. Check for default SSH in VPC. If not there, then create: **gcloud beta compute --project=hale-brook-377621 firewall-rules create default-allow-ssh --direction=INGRESS --priority=1000 --network=default --action=ALLOW --rules=tcp:22 --source-ranges=0.0.0.0/0 --enable-logging**
8. Now that we have the Cloud Fusion environment in GCP let's build a schema. We need this schema for transformation of the CSV data. In the Cloud Data Fusion window, click the View Instance link in the Action column. You will be redirected to another page. Click the provided url to open Cloud Data Fusion instance. Your choice to click "Start Tour" or "No, Thanks" button at the Welcome popup. Expand the "hamburger" menu, select Pipeline -> Studio

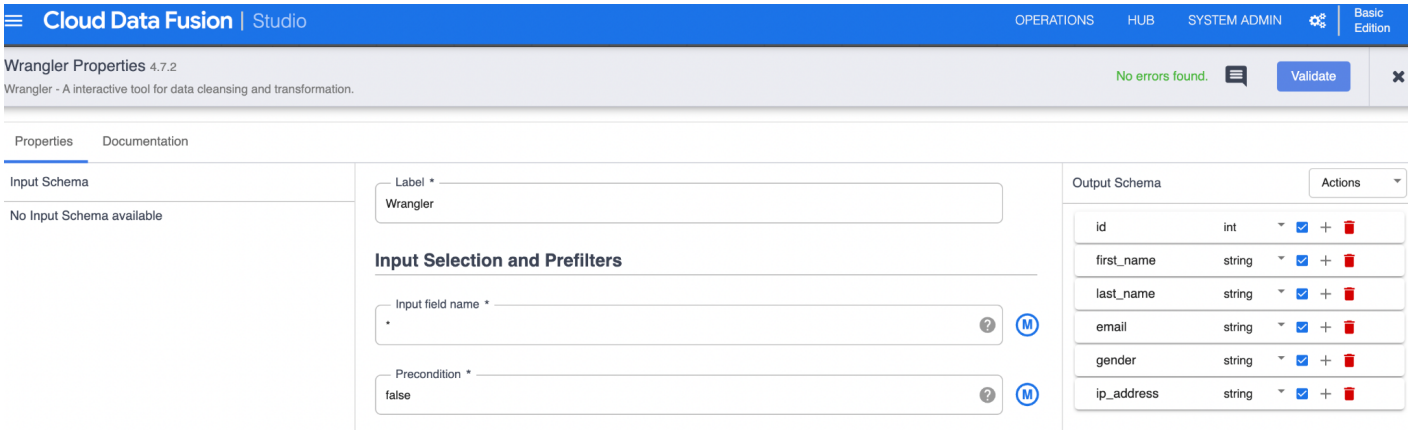


9. By default, the initial import will assume there is only one column in your data file. To parse it as a CSV, choose Parse → CSV, then select the delimiter and check the "Set first row as header" box as appropriate. Click the Apply button.





10. The "Columns" and "Transformation steps" tabs show output schema and the Wrangler's recipe. Click Apply at the upper right corner. Click the Validate button. The green "No errors found" indicates success.



11. In Wrangler Properties, click the Actions dropdown to Export the desired schema into your local storage for future Import if needed.
12. Save the Wrangler Recipe for future usage.
13. In the Data Pipelines UI, in the upper left, you should see that Data Pipeline - Batch is selected as the pipeline type.
14. Select the Source node.
15. Under the Source section in the Plugin palette on the left, double-click on the Google Cloud Storage node, which appears in the Data Pipelines UI.
16. Point to the GCS source node and click Properties.
17. Source Node: Fill in the required fields. Set following fields:

- Label = {any text}
- Reference name = {any text}
- Project ID = auto detect
- Path = GCS URL to bucket in your current project. For example, gs://$BUCKET_NAME/csv/
- Format = text
- Path Field = filename
- Path Filename Only = true
- Read Files Recursively = true
- Add field 'filename' to the GCS Output Schema by clicking the + button.
- Click Get Schema to generate schema.

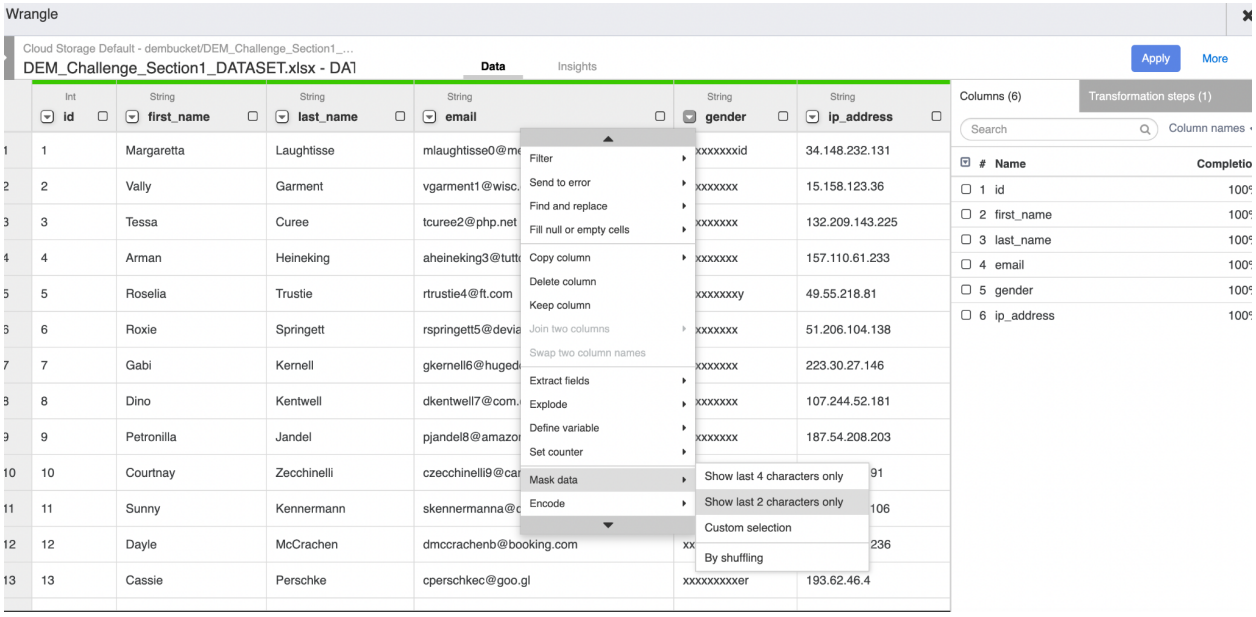
16. Select the Transform node.

- Under the Transform section in the Plugin palette on the left, double-click the Wrangler node, which appears in the Data Pipelines UI. Connect GCS source node to Wrangler transform node.
- Point to the Wrangler node and click Properties.
- Click Actions drop down and select Import to import a saved schema

17. Data masking and de-identification

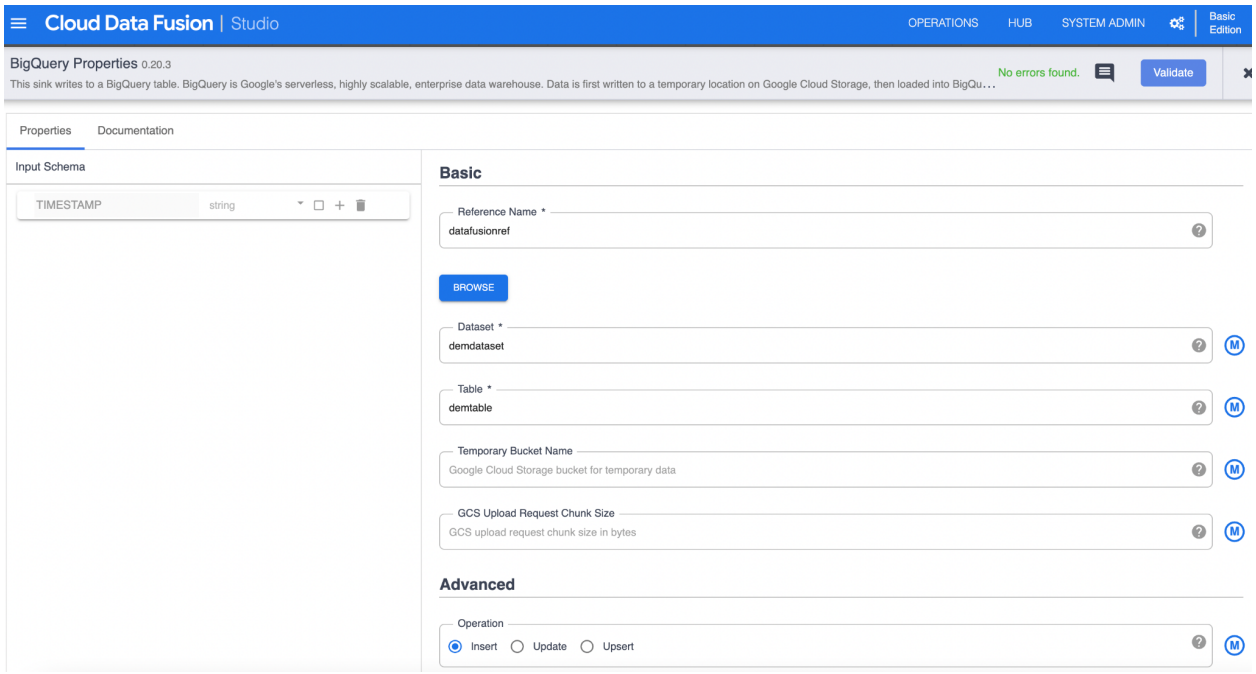
- You can select individual data columns by clicking the down arrow in the column and applying masking rules under the Mask data selection as per your requirements (for example, Gender and email address column).

- I masked the gender as a test to show that the function works. I selected Show last 2 characters only.

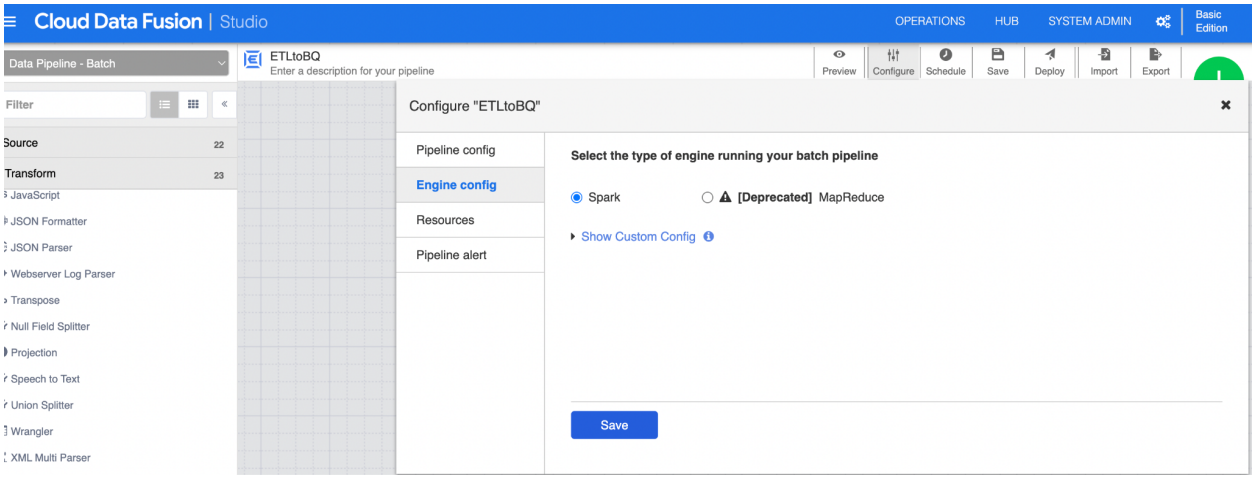


18. Sink node: Select the sink node.

- Under the Sink section in the Plugin palette on the left, double click on BigQuery node, which will appear in the Data Pipeline UI.
- Point to the BigQuery sink node and click Properties.
- Drag a connection arrow > on the right edge of the source node and drop on the left edge of the destination node.

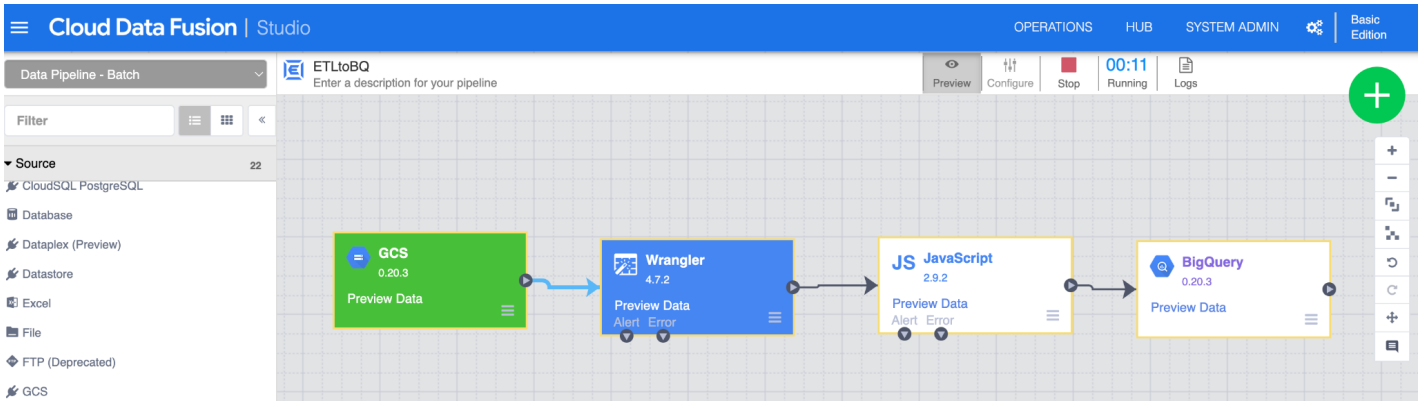


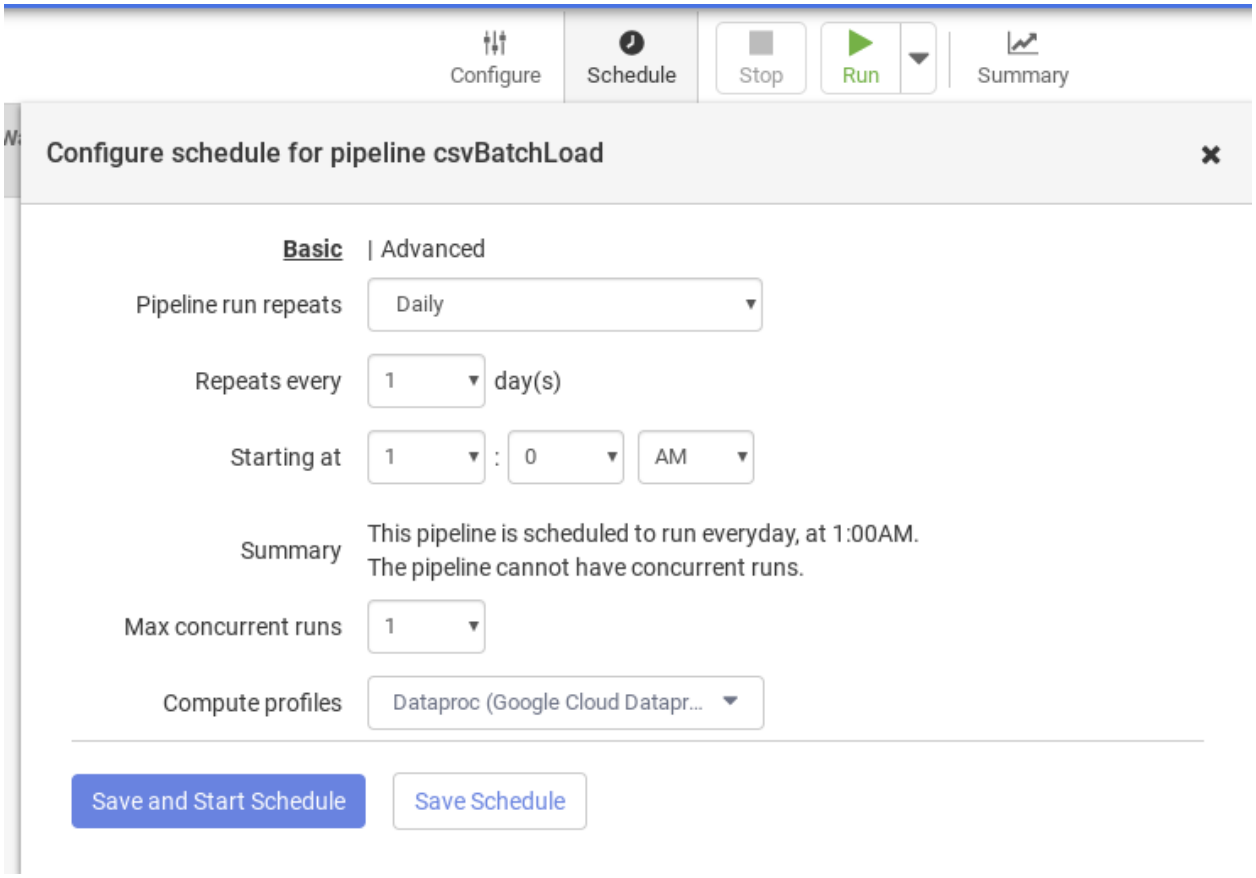
19. Automate: In the upper-right corner of Data Fusion studio, click Configure. Select Spark for Engine Config. Click Save in Configure window.



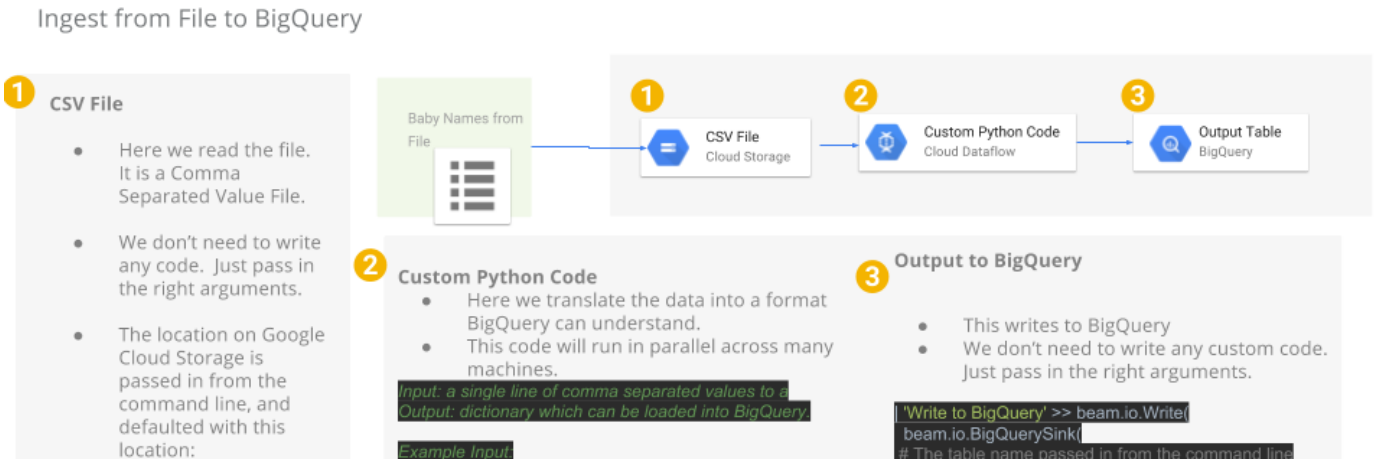
Note: When you run a pipeline, Cloud Data Fusion provides an ephemeral Cloud Dataproc cluster, runs the pipeline, and then tears down the cluster. This could take a few minutes. You can observe the status of the pipeline transition from Provisioning to Starting to Running to Deprovisioning to Succeeded during this time.

Running the job without errors and successfully !!!




Configure
Schedule
Stop
Run
Summary
Configure schedule for pipeline csvBatchLoad
Basic | Advanced
Pipeline run repeats
Daily
Repeats every
1
day(s)
Starting at
1
0
AM
Summary
This pipeline is scheduled to run everyday, at 1:00AM.
The pipeline cannot have concurrent runs.
Max concurrent runs
1
Compute profiles
Dataproc (Google Cloud Datapr...
Save and Start Schedule
Save Schedule

DataFlow


Ingest from File to BigQuery
1
CSV File
Here we read the file. It is a Comma Separated Value File.
We don't need to write any code. Just pass in the right arguments.
The location on Google Cloud Storage is passed in from the command line, and defaulted with this location:
Baby Names from File
1
CSV File
Cloud Storage
2
Custom Python Code
Cloud Dataflow
3
Output Table
BigQuery
2
Custom Python Code
Here we translate the data into a format BigQuery can understand.
This code will run in parallel across many machines.
Input: a single line of comma separated values to a
Output: dictionary which can be loaded into BigQuery.
Example Input
3
Output to BigQuery
This writes to BigQuery
We don't need to write any custom code. Just pass in the right arguments.
| 'Write to BigQuery' >> beam.io.Write(
beam.io.BigQuerySink(
The table name passed in from the command line

Dataingestion.py

```
from __future__ import absolute_import
import argparse
import logging

import re

import apache_beam as beam

from apache_beam.options.pipeline_options import
PipelineOptions




class DataIngestion:

   """A helper class which contains the logic to
translate the file into
   a format BigQuery will accept."""

   def parse_method(self, string_input):

       """This method translates a single line of comma
separated values to a
       dictionary which can be loaded into BigQuery.

       Args:

           string_input: A comma separated list of
values in the form of


state_abbreviation,gender,year,name,count_of_babies,data
set_created_date
               Example string_input:
KS,F,1923,Dorothy,654,11/28/2016
       Returns:

           A dict mapping BigQuery column names as keys
to the corresponding value
           parsed from string_input. In this example,
the data is not transformed, and
           remains in the same format as the CSV.

           example output:

           {

               'state': 'KS',
```

```
                'gender': 'F',
                'year': '1923',
                'name': 'Dorothy',
                'number': '654',
                'created_date': '11/28/2016'
            }
        """
        # Strip out carriage return, newline and quote
characters.
        values = re.split(",",
                          re.sub('\r\n', '', re.sub(u'"',
'', string_input)))
        row = dict(
            zip(('state', 'gender', 'year', 'name',
'number', 'created_date'),
                values))
        return row




def run(argv=None):
    """The main function which creates the pipeline and
runs it."""


    parser = argparse.ArgumentParser()



    # Here we add some specific command line arguments we
expect.
    # Specifically we have the input file to read and the
output table to write.
    # This is the final stage of the pipeline, where we
define the destination
    # of the data. In this case we are writing to
BigQuery.
    parser.add_argument(
```

```
        '--input',
        dest='input',
        required=False,
        help='Input file to read. This can be a local
file or '
        'a file in a Google Storage Bucket.',
        # This example file contains a total of only 10
lines.
        # Useful for developing on a small set of data.

default='gs://python-dataflow-example/data_files/head_us
a_names.csv')


    # This defaults to the lake dataset in your BigQuery
project. You'll have
    # to create the lake dataset yourself using this
command:
    # bq mk lake
    parser.add_argument('--output',
                        dest='output',
                        required=False,
                        help='Output BQ table to write
results to.',
                        default='lake.usa_names')


    # Parse arguments from the command line.
    known_args, pipeline_args =
parser.parse_known_args(argv)


    # DataIngestion is a class we built in this script to
hold the logic for
    # transforming the file into a BigQuery table.
    data_ingestion = DataIngestion()
```

```
   # Initiate the pipeline using the pipeline arguments
passed in from the
   # command line. This includes information such as the
project ID and
   # where Dataflow should store temp files.
   p =
beam.Pipeline(options=PipelineOptions(pipeline_args))


   (p
    # Read the file. This is the source of the pipeline.
All further
    # processing starts with lines read from the file.
We use the input
    # argument from the command line. We also skip the
first line which is a
    # header row.
    | 'Read from a File' >>
beam.io.ReadFromText(known_args.input,

skip_header_lines=1)
    # This stage of the pipeline translates from a CSV
file single row
    # input as a string, to a dictionary object
consumable by BigQuery.
    # It refers to a function we have written. This
function will
    # be run in parallel on different workers using
input from the
    # previous stage of the pipeline.
    | 'String To BigQuery Row' >>
    beam.Map(lambda s: data_ingestion.parse_method(s))
    | 'Write to BigQuery' >> beam.io.Write(
        beam.io.BigQuerySink(
            # The table name is a required argument for
the BigQuery sink.
            # In this case we use the value passed in
from the command line.
            known_args.output,
            # Here we use the simplest way of defining a
schema:
```

```
            # fieldName:fieldType

schema='state:STRING,gender:STRING,year:STRING,name:STRI
NG,'
            'number:STRING,created_date:STRING',
            # Creates the table in BigQuery if it does
not yet exist.
create_disposition=beam.io.BigQueryDisposition.CREATE_IF
_NEEDED,
            # Deletes all data in the BigQuery table
before writing.
write_disposition=beam.io.BigQueryDisposition.WRITE_TRUN
CATE)))
   p.run().wait_until_finish()




if __name__ == '__main__':
   logging.getLogger().setLevel(logging.INFO)
   run()
```

Run the Apache Beam pipeline

Run the following in Cloud Shell:

docker run -it -e PROJECT=$PROJECT -v $(pwd)/dataflow-python-examples:/dataflow python:3.7 /bin/bash

This command will pull a Docker container with the latest stable version of Python 3.7 and execute a command shell to run the next commands within the container. The `-v` flag provides the source code as a `volume` for the container so that we can edit in Cloud Shell editor and still access it within the container.

3. Once the container finishes pulling, run the following to install `apache-beam`:

pip install apache-beam[gcp]==2.24.0

4. Next, change directories into where you linked the source code:

cd dataflow/

You will run the Dataflow pipeline in the cloud.

5. The following will spin up the workers required, and shut them down when complete:

```
python dataflow_python_examples/data_ingestion.py \
  --project=$PROJECT --region= \
  --runner=DataflowRunner \
  --staging_location=gs://$PROJECT/test \
  --temp_location gs://$PROJECT/test \
  --input gs://$PROJECT/data_files/head_usa_names.csv \
  --save_main_session
```

Datatransformation.py

```
from __future__
import
absolute_import
                    import argparse
                    import csv
                    import logging
                    import os


                    import apache_beam as beam
                    from apache_beam.options.pipeline_options import
                    PipelineOptions
                    from apache_beam.io.gcp.bigquery import
                    parse_table_schema_from_json



                    class DataTransformation:
                      """A helper class which contains the logic to translate the file
                    into a
                     format BigQuery will accept."""


                      def __init__(self):
                        dir_path = os.path.dirname(os.path.realpath(__file__))
                        self.schema_str = ''
                        # Here we read the output schema from a json file.  This is
                    used to specify the types
                        # of data we are writing to BigQuery.
                        schema_file = os.path.join(dir_path, 'resources',
                    'usa_names_year_as_date.json')
                        with open(schema_file) \
```

```
            as f:
        data = f.read()
        # Wrapping the schema in fields is required for the
BigQuery API.
        self.schema_str = '{"fields": ' + data + '}'


  def parse_method(self, string_input):
    """This method translates a single line of comma separated
values to a
  dictionary which can be loaded into BigQuery.
    Args:
      string_input: A comma separated list of values in the form
of

state_abbreviation,gender,year,name,count_of_babies,dataset_cr
eated_date
          example string_input: KS,F,1923,Dorothy,654,11/28/2016
    Returns:
      A dict mapping BigQuery column names as keys to the
corresponding value
      parsed from string_input.  In this example, the data is not
transformed, and
      remains in the same format as the CSV.  There are no date
format transformations.
          example output:
              {'state': 'KS',
               'gender': 'F',
               'year': '1923-01-01', <- This is the BigQuery date
format.
               'name': 'Dorothy',
               'number': '654',
```

```
                'created_date': '11/28/2016'
                }
        """
        # Strip out return characters and quote characters.
        schema = parse_table_schema_from_json(self.schema_str)


        field_map = [f for f in schema.fields]


        # Use a CSV Reader which can handle quoted strings etc.
        reader = csv.reader(string_input.split('\n'))
        for csv_row in reader:
          values = [x.decode('utf8') for x in csv_row]
          # Our source data only contains year, so default January
1st as the
          # month and day.
          month = u'01'
          day = u'01'
          # The year comes from our source data.
          year = values[2]


          row = {}
          i = 0
          # Iterate over the values from our csv file, applying any
transformation logic.
          for value in values:
            # If the schema indicates this field is a date format, we
must
```

```
            # transform the date from the source data into a format
that
            # BigQuery can understand.
            if field_map[i].type == 'DATE':
               # Format the date to YYYY-MM-DD format which
BigQuery
               # accepts.
               value = u'-'.join((year, month, day))


            row[field_map[i].name] = value
            i += 1


         return row



def run(argv=None):
  """The main function which creates the pipeline and runs it."""
  parser = argparse.ArgumentParser()
  # Here we add some specific command line arguments we
expect.   Specifically
  # we have the input file to load and the output table to write to.
  parser.add_argument(
     '--input', dest='input', required=False,
     help='Input file to read.  This can be a local file or '
         'a file in a Google Storage Bucket.',
     # This example file contains a total of only 10 lines.
     # It is useful for developing on a small set of data
```

```
default='gs://python-dataflow-example/data_files/head_usa_names.csv')
  # This defaults to the temp dataset in your BigQuery project. You'll have
  # to create the temp dataset yourself using bq mk temp
  parser.add_argument('--output', dest='output', required=False,
                      help='Output BQ table to write results to.',
                      default='lake.usa_names_transformed')

  # Parse arguments from the command line.
  known_args, pipeline_args = parser.parse_known_args(argv)
  # DataTransformation is a class we built in this script to hold the logic for
  # transforming the file into a BigQuery table.
  data_ingestion = DataTransformation()

  # Initiate the pipeline using the pipeline arguments passed in from the
  # command line.  This includes information like where Dataflow should
  # store temp files, and what the project id is.
  p = beam.Pipeline(options=PipelineOptions(pipeline_args))
  schema = parse_table_schema_from_json(data_ingestion.schema_str)

  (p
   # Read the file.  This is the source of the pipeline.  All further
```

```
   # processing starts with lines read from the file.  We use the
input
   # argument from the command line.  We also skip the first line
which is a
   # header row.
   | 'Read From Text' >> beam.io.ReadFromText(known_args.input,
                                             skip_header_lines=1)
   # This stage of the pipeline translates from a CSV file single
row
   # input as a string, to a dictionary object consumable by
BigQuery.
   # It refers to a function we have written.  This function will
   # be run in parallel on different workers using input from the
   # previous stage of the pipeline.
   | 'String to BigQuery Row' >> beam.Map(lambda s:
                                          data_ingestion.parse_method(s))
   | 'Write to BigQuery' >> beam.io.Write(
     beam.io.BigQuerySink(
         # The table name is a required argument for the BigQuery
sink.
         # In this case we use the value passed in from the
command line.
         known_args.output,
         # Here we use the JSON schema read in from a JSON file.
         # Specifying the schema allows the API to create the table
correctly if it does not yet exist.
         schema=schema,
         # Creates the table in BigQuery if it does not yet exist.

create_disposition=beam.io.BigQueryDisposition.CREATE_IF_NEE
DED,
```

```
                # Deletes all data in the BigQuery table before writing.

write_disposition=beam.io.BigQueryDisposition.WRITE_TRUNCATE)))
  p.run().wait_until_finish()



if __name__ == '__main__':
  logging.getLogger().setLevel(logging.INFO)
  run()
```

1. Run the following commands:

```
python dataflow_python_examples/data_transformation.py \
  --project=$PROJECT \
  --region= \
  --runner=DataflowRunner \
  --staging_location=gs://$PROJECT/test \
  --temp_location gs://$PROJECT/test \
  --input gs://$PROJECT/data_files/head_usa_names.csv \
  --save_main_session
```

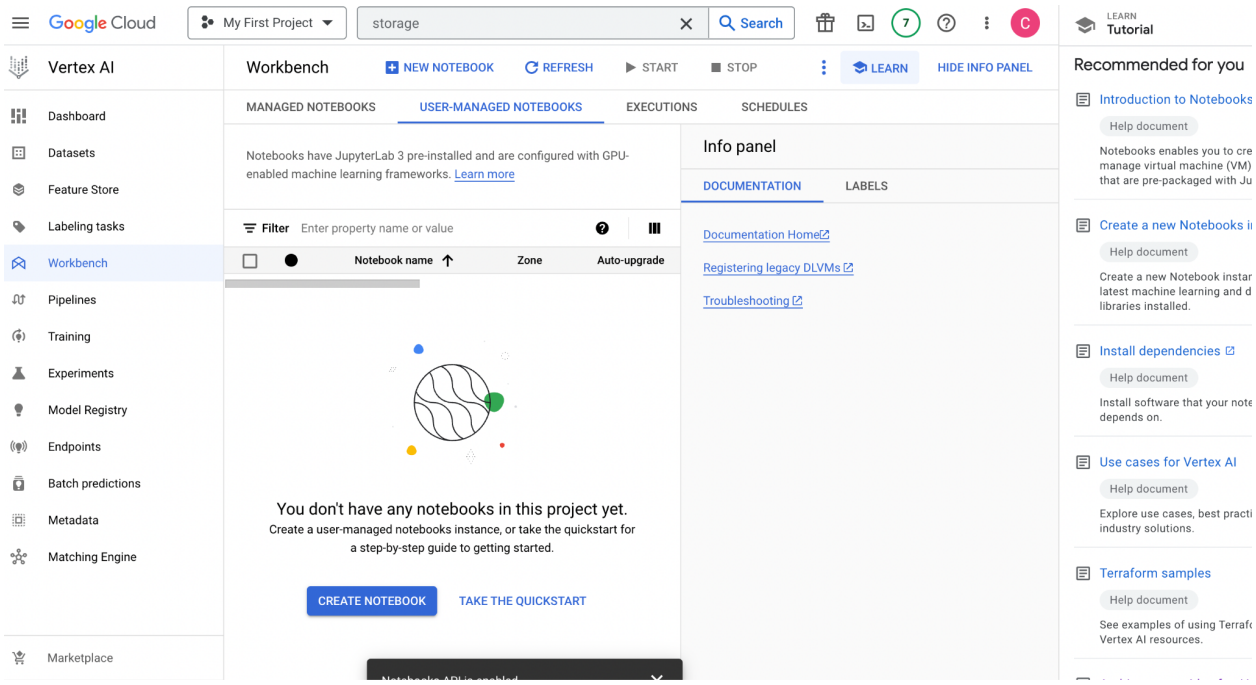
2. When your Job Status is Succeeded in the Dataflow Job Status screen, navigate to BigQuery to check to see that your data has been populated.

Step 7: ML model - Built on Vertex AI (Google Cloud)

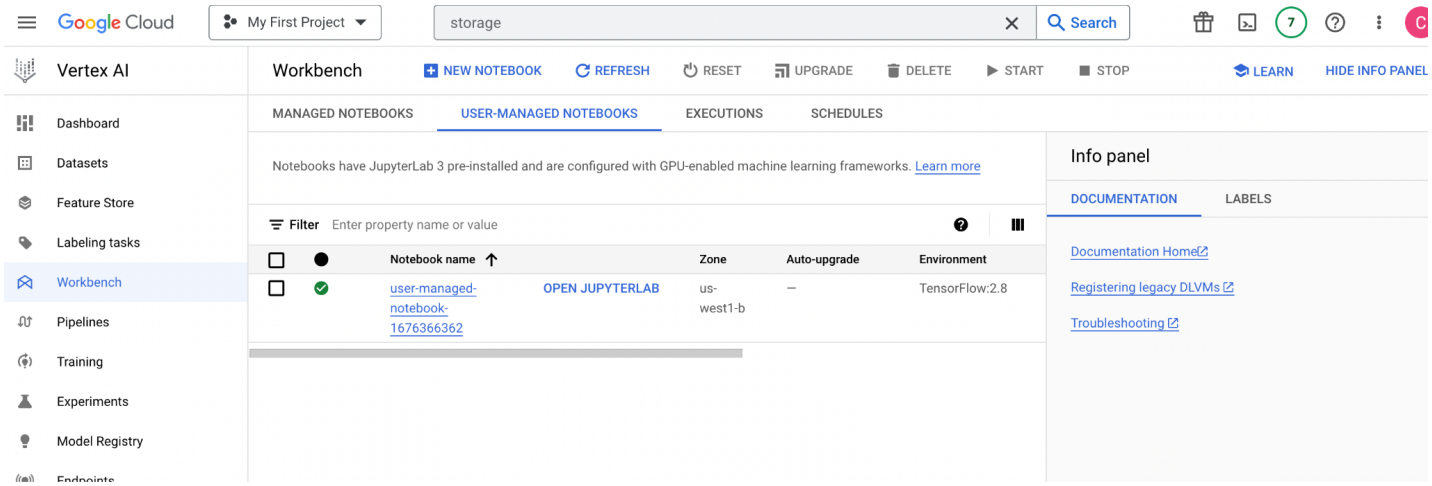
Setting up Vertex AI Workbench

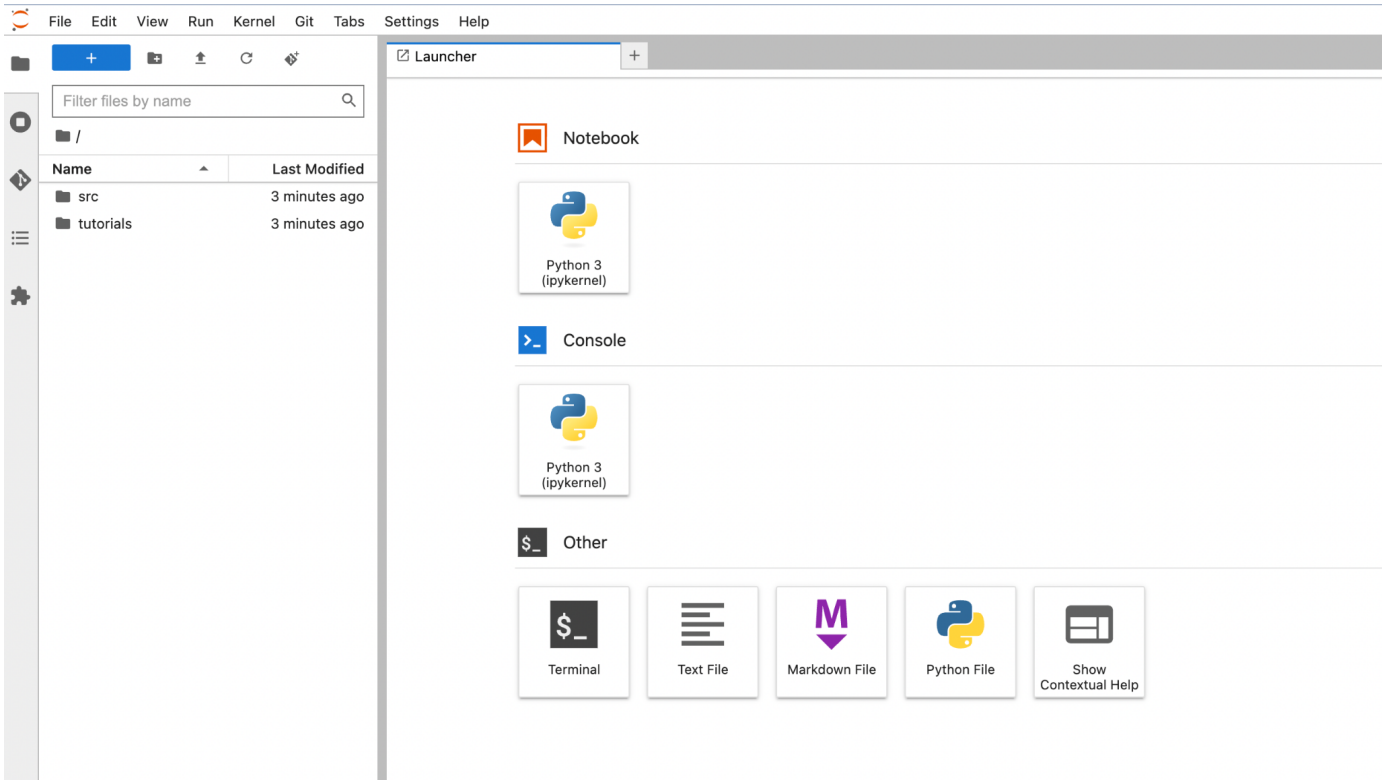
First enable Vertex AI API.

Created a Vertex AI Workbench. It is Jupyter Notebooks as a service on GCP where we can run Python or other languages code to do ML on a managed or user managed notebooks.

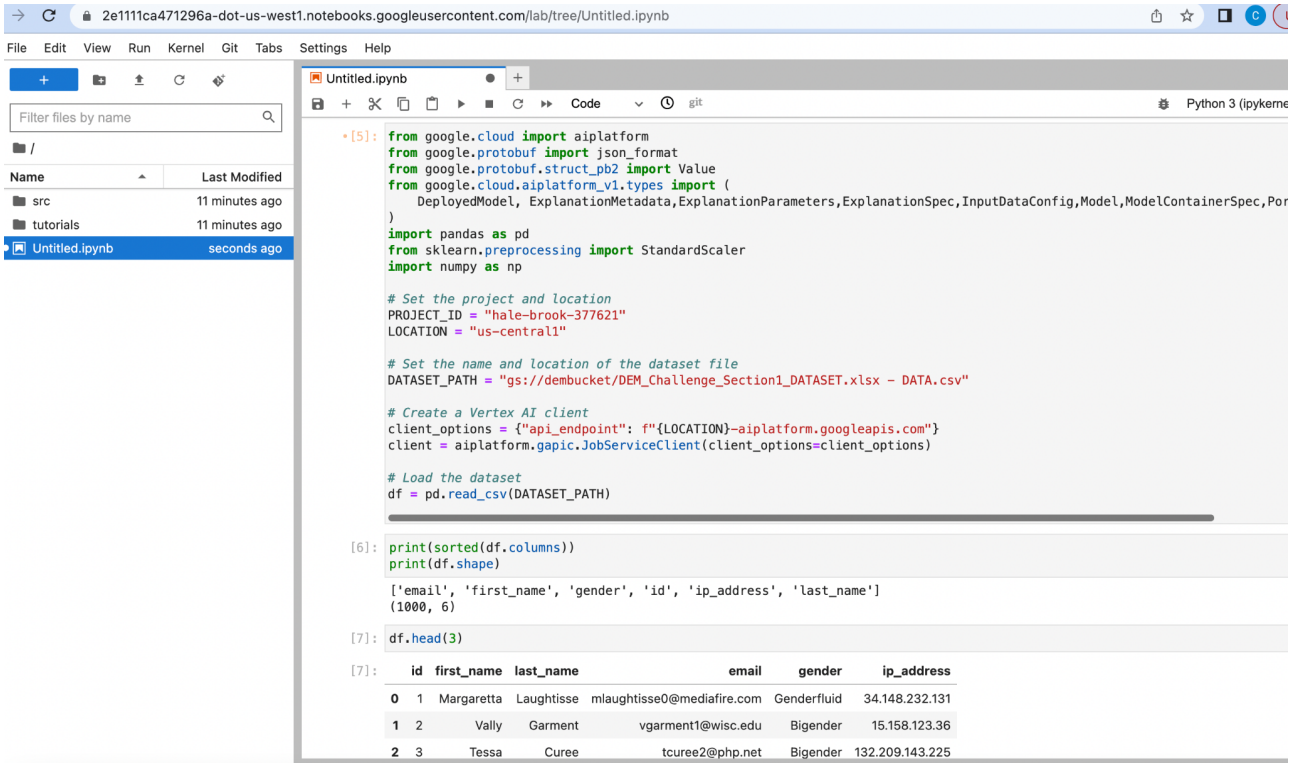


I chose a small config machine as this is for development only.

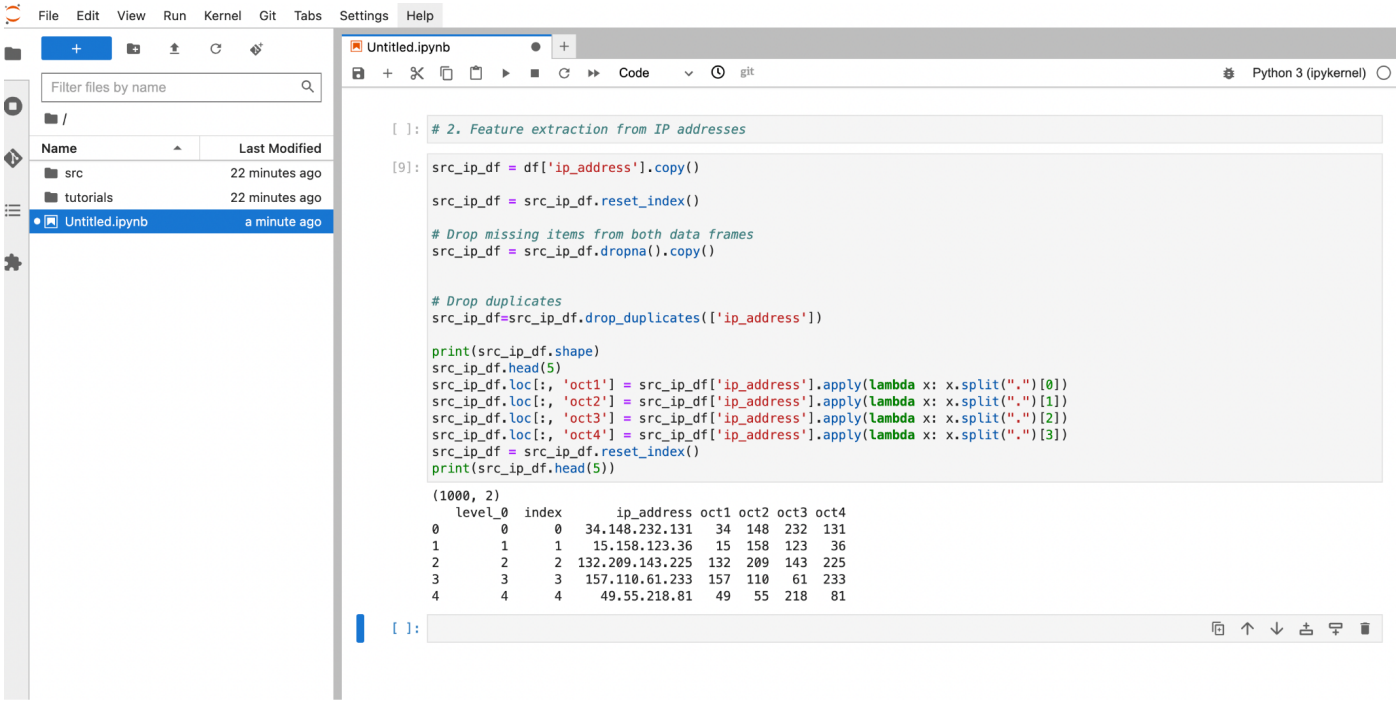


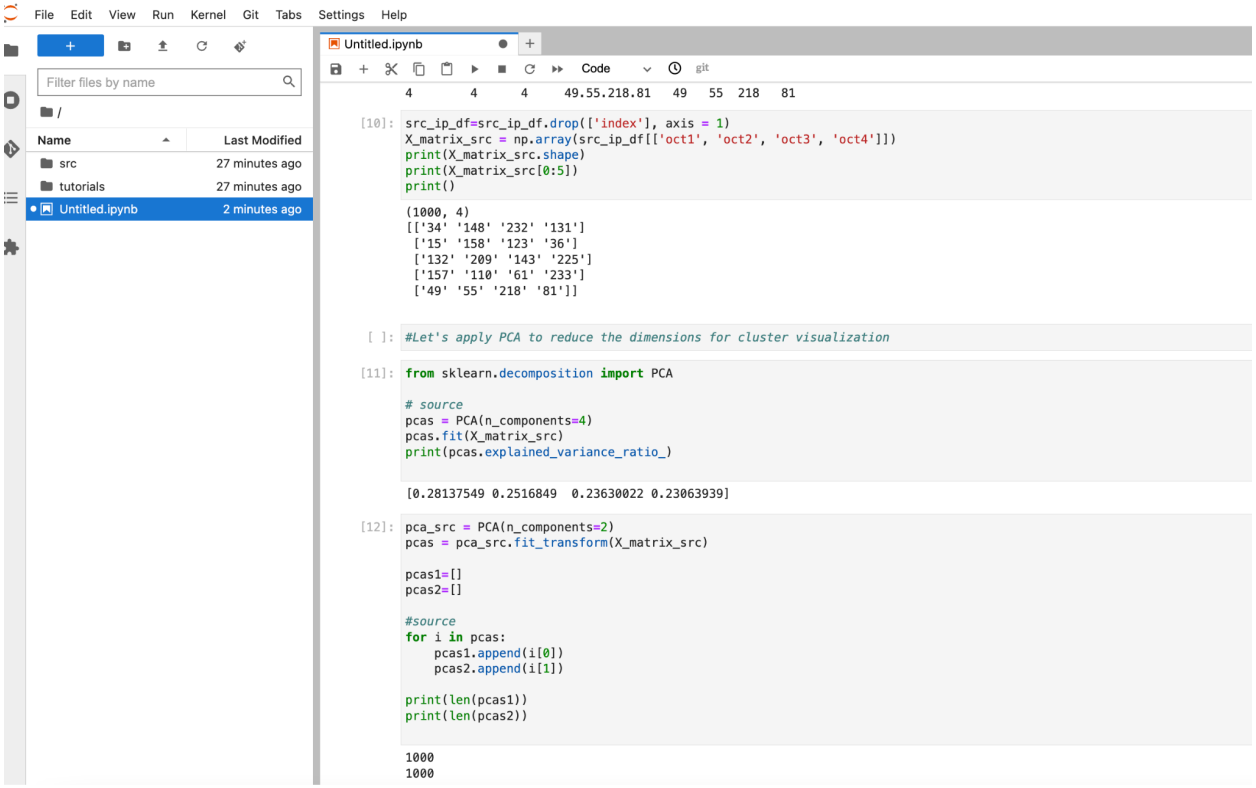


Tested connection successfully !!



Preprocessing and Feature Extraction



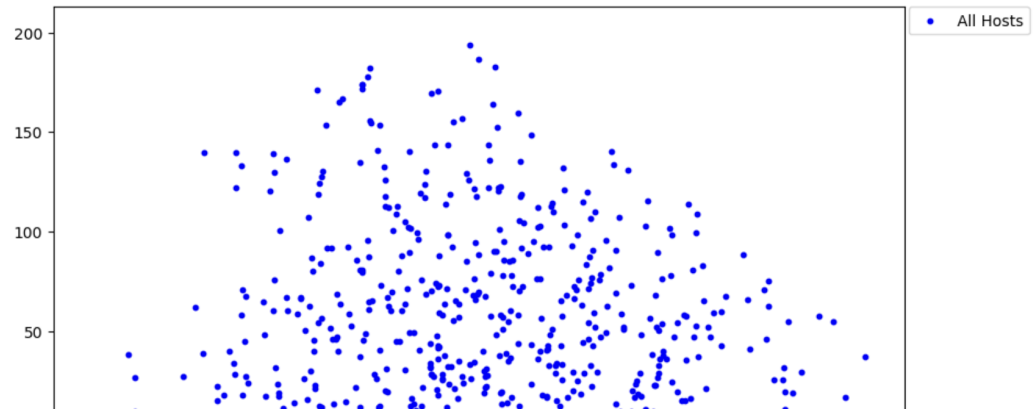


MLmodel_VertexAI.ipynb

Code

[13]:
```
#Store the two principal components as new features in the source and destination data frames.
src_ip_df['pca1'] = pcas1
src_ip_df['pca2'] = pcas2
print(src_ip_df[:2])
print()

# Maintain a copy for using for K-Means later.
src_ip_df_copy = src_ip_df.copy()

#Scatter plot of Source IP address data
import matplotlib.pyplot as plt
%matplotlib inline
plt.rcParams["figure.figsize"] = (10,10)
plt.scatter(src_ip_df['pca1'], src_ip_df['pca2'], s=10, color='blue',label="All Hosts")
plt.legend(bbox_to_anchor=(1.005, 1), loc=2, borderaxespad=0.)
plt.show()
```

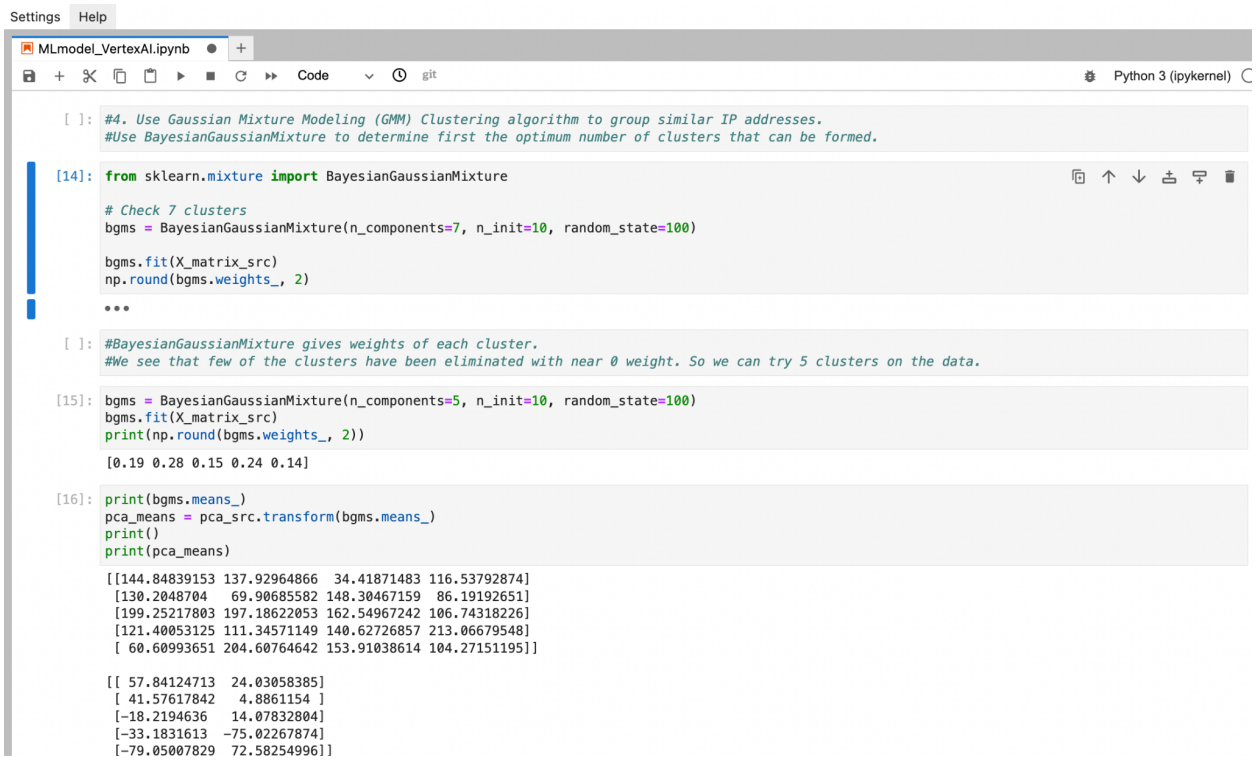
	level_0	ip_address	oct1	oct2	oct3	oct4	pca1	pca2
0	0	34.148.232.131	34	148	232	131	-110.737651	23.137985
1	1	15.158.123.36	15	158	123	36	-28.086457	130.277664



Machine Learning model

Perform K-Means clustering and GMM (Gausian) clustering as the data looks like demographics data of people and contains network information (IP address).

My thought process is that we can find patterns and outliers from the people using clustering algorithms. These are unsupervised types of machine learning.



GMM

```
from google.cloud import aiplatform
from google.cloud.aiplatform.gapic.schema import predict as predict_pb2
import numpy as np
import pandas as pd
from sklearn.preprocessing import StandardScaler
from sklearn.mixture import GaussianMixture

# Load the dataset
df = pd.read_csv("dataset.csv")

# Select the features for clustering
X = df[["first_name", "last_name", "email", "gender", "ip_address"]]

# Preprocess the features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
```

```
# Set the number of clusters
num_clusters = 3

# Set the container image URI for the GMM model
container_image_uri =
"us-docker.pkg.dev/vertex-ai/prediction/gaussian-mixture-model:latest"

# Create the model container specification
model_container_spec = {
    "image_uri": container_image_uri,
}

# Create the GMM model
model = aiplatform.Model(
    display_name="gmm-clustering-model",
    container_spec=model_container_spec,
    predict_schemata=predict_pb2.ValueSpec(
        encoded_value=predict_pb2.EncodedValueSpec(
            tensor_shape=predict_pb2.TensorShape(dim=[predict_pb2.FixedDim(size=5)])
        )
    ),
)

# Create the input data configuration
input_data_config = {
    "instances": [
        {
            "first_name": x[0],
            "last_name": x[1],
            "email": x[2],
            "gender": x[3],
            "ip_address": x[4],
        }
        for x in X_scaled
    ]
}

# Set the number of clusters in the explanation metadata
```

```
explanation_metadata = {
    "inputs": {
        "first_name": {"type": "CATEGORY"},
        "last_name": {"type": "CATEGORY"},
        "email": {"type": "CATEGORY"},
        "gender": {"type": "CATEGORY"},
        "ip_address": {"type": "CATEGORY"},
    },
    "outputs": {"cluster": {"type": "CATEGORY", "number_of_categories": num_clusters}},
}

# Set the explanation spec
explanation_spec = {"metadata": explanation_metadata}

# Train the GMM model and deploy it to an endpoint
endpoint = model.deploy(
    machine_type="n1-standard-4",
    min_replica_count=1,
    max_replica_count=1,
)

# Create a prediction request for a sample dataset
sample_dataset = [
    {"first_name": "Alice", "last_name": "Smith", "email": "alice@example.com", "gender":
"Female", "ip_address": "123.45.67.89"},
    {"first_name": "Bob", "last_name": "Jones", "email": "bob@example.com", "gender":
"Male", "ip_address": "98.76.54.32"},
    {"first_name": "Charlie", "last_name": "Lee", "email": "charlie@example.com", "gender":
"Male", "ip_address": "12.34.56.78"},
]
sample_dataset_scaled =
scaler.transform(pd.DataFrame(sample_dataset)[["first_name", "last_name", "email",
"gender", "ip_address"]])
prediction_request = predict_pb2.PredictRequest(
    instances=[{"values": [list(row)]} for row in sample_dataset_scaled],
    parameters={"explanation_spec": explanation_spec},
)

# Make the prediction request using the Vertex AI Python SDK
```

response = endpoint.predict(prediction_request)

Get the cluster assignments from

# Extract the predicted clusters
predicted_clusters = np.array(response.predictions[0].tables.values).flatten()

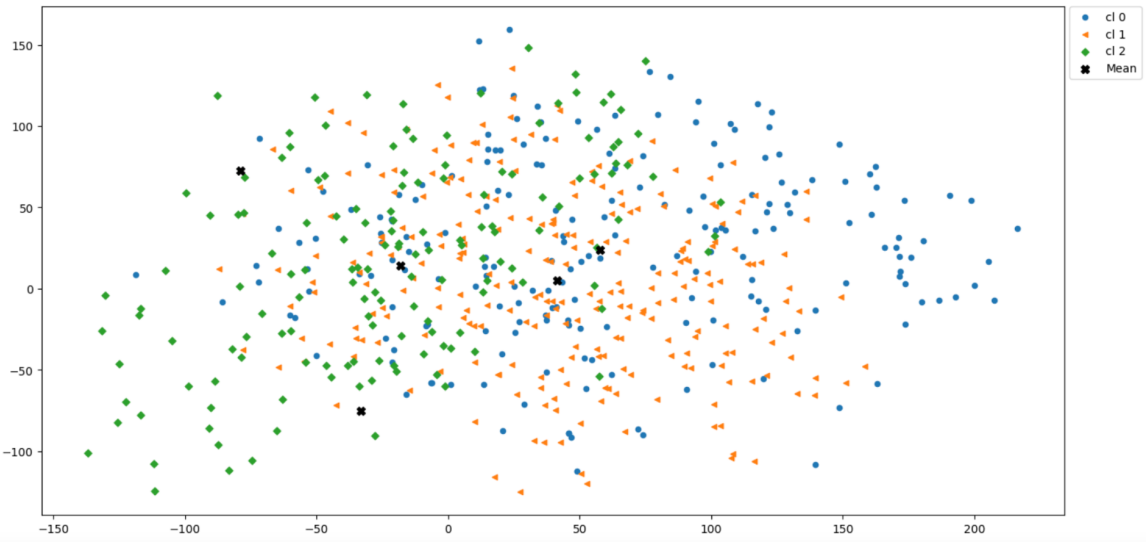
# Print the predicted clusters for each sample dataset
for i, cluster in enumerate(predicted_clusters):
    print(f"Sample dataset {i+1} is assigned to cluster {cluster}")

MLmodel_VertexAI.ipynb

Code

Python 3 (ipykernel)

[23]:
```
#Plot the clusters identified by GMM
dfs0=src_ip_df[src_ip_df.kcluster==0]
dfs1=src_ip_df[src_ip_df.kcluster==1]
dfs2=src_ip_df[src_ip_df.kcluster==2]
plt.rcParams["figure.figsize"] = (16,8)
pyplot.scatter(dfs0['pca1'],dfs0['pca2'],s=20,label="cl 0")
pyplot.scatter(dfs1['pca1'],dfs1['pca2'],s=20,marker="<",label="cl 1")
pyplot.scatter(dfs2['pca1'],dfs2['pca2'],s=20,marker="D",label="cl 2")
pyplot.scatter(means_df_s['pca1'],means_df_s['pca2'],s=50,marker="X",label="Mean", color='black')

pyplot.legend(bbox_to_anchor=(1.005, 1), loc=2, borderaxespad=0.)
pyplot.show()
```



K-Means clustering

from google.cloud import aiplatform
from google.protobuf import json_format
from google.protobuf.struct_pb2 import Value
from google.cloud.aiplatform_v1.types import (
    DeployedModel,
    ExplanationMetadata,

```
    ExplanationParameters,
    ExplanationSpec,
    InputDataConfig,
    Model,
    ModelContainerSpec,
    Port,
    PredictRequest
)
import pandas as pd
from sklearn.preprocessing import StandardScaler
import numpy as np

# Set the project and location
PROJECT_ID = "hale-brook-377621"
LOCATION = "us-central1"

# Set the name and location of the dataset file
DATASET_PATH = "gs://your-bucket/dataset.csv"

# Create a Vertex AI client
client_options = {"api_endpoint": f"{LOCATION}-aiplatform.googleapis.com"}
client = aiplatform.gapic.JobServiceClient(client_options=client_options)

# Load the dataset
df = pd.read_csv(DATASET_PATH)

# Select features for clustering
X = df[["first_name", "last_name", "email", "gender", "ip_address"]]

# Preprocess the features using standard scaler
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Set the number of clusters
num_clusters = 3

# Set the container image URI
image_uri = "us-docker.pkg.dev/vertex-ai/prediction/kmeans-cpu.1-0:latest"
```

```
# Create the model container specification
model_container_spec = ModelContainerSpec(image_uri=image_uri)

# Create the K-Means model
model = Model(
    display_name="kmeans-model",
    container_spec=model_container_spec,

metadata_schema_uri="gs://google-cloud-aiplatform/schema/prediction/classification_
1.0.0.yaml",
)

# Create the input data configuration
input_data_config = InputDataConfig(
    instances_format="json",

json_instance_schema_uri="gs://google-cloud-aiplatform/schema/prediction/input/text
_classification_example.json",
)

# Create the explanation spec
explanation_metadata = ExplanationMetadata(
    inputs={"features": {"input_tensor_name": "input"}}
)
explanation_parameters = ExplanationParameters(
    {"sampled_shapley_attribution": {"path_count": 20}}
)
explanation_spec = ExplanationSpec(
    metadata=explanation_metadata, parameters=explanation_parameters
)

# Create the model deployment
deployed_model = DeployedModel(model=model)
endpoint = aiplatform.Endpoint.create(
    display_name="kmeans-endpoint", explanation_spec=explanation_spec
)
endpoint.deploy(
    deployed_model=deployed_model,
    traffic_percentage=100,
```

```
    machine_type="n1-standard-2",
)

# Create a prediction request for a sample dataset
input_data = [{"first_name": row[0], "last_name": row[1], "email": row[2], "gender": row[3],
"ip_address": row[4]} for row in X.values]
predict_request = PredictRequest(
    endpoint=endpoint.resource_name,

instances=[json_format.Parse(json_format.MessageToJson(PredictionInput(data=Input
DataConfig.EncodedData(json_format.Parse(json_format.MessageToJson(Value(json_v
alue=bytes(json.dumps(instance), "utf-8")))))), PredictRequest) for instance in
input_data],
)

# Make a prediction request
response = client.predict(predict_request)

# Get the cluster assignments
cluster_assignments = np.array([np.argmax(prediction.outputs["scores"].value) for
prediction in response.predictions])

#add a new column in the dataset
df["cluster"] = cluster_assignments
#Print the dataset with the predicted clusters
print(df.head())
```

Step 8: Deploy the ML model as API endpoint - I used Vertex AI Platform prediction Service

```
from googleapiclient import discovery
from google.oauth2 import service_account
import numpy as np

# Specify the project ID and the name of the model to be deployed
PROJECT_ID = 'hale-brook-377621'
```

```
MODEL_NAME = 'your-model-name'

# Load the service account credentials
credentials =
service_account.Credentials.from_service_account_file('path/to/your/credentials.json')

# Create a client object for the AI Platform Prediction service
ml = discovery.build('ml', 'v1', credentials=credentials)

# Define a function to preprocess the input data before making predictions
def preprocess(input_data):
    # Apply any necessary preprocessing steps here, such as scaling or one-hot encoding
    return input_data

# Define a function to make predictions using the deployed model
def predict(input_data):
    # Preprocess the input data
    preprocessed_data = preprocess(input_data)

    # Create a request body for the prediction request
    request_body = {"instances": preprocessed_data.tolist()}

    # Make the prediction request using the AI Platform Prediction API
    parent = f'projects/{PROJECT_ID}/models/{MODEL_NAME}'
    response = ml.projects().predict(name=parent, body=request_body).execute()

    # Extract the predicted labels from the response
    predicted_labels = np.array(response['predictions']).flatten()

    # Return the predicted labels
    return predicted_labels

gcloud ai-platform models create your-model-name --regions=us-central1
gcloud ai-platform versions create your-version-name --model=your-model-name
--runtime-version=2.5 --python-version=3.7 --framework=scikit-learn
--origin=gs://your-bucket/path/to/model --project=hale-brook-377621
```