Configuration Manual

MSc Research Project
Cloud Computing

Ghiridhar Iyer
Student ID: X18183468

School of Computing
National College of Ireland

Supervisor: Dr. Manuel Tova-Izquierdo

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Ghiridhar Iyer
Student ID:	X18183468
Programme:	Cloud Computing
Year:	2020
Module:	MSc Research Project
Supervisor:	Dr. Manuel Tova-Izquierdo
Submission Due Date:	17/8/2020
Project Title:	Configuration Manual
Word Count:	4283
Page Count:	61

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

I agree to an electronic copy of my thesis being made publicly available on TRAP the National College of Ireland's Institutional Repository for consultation.

Signature:	
Date:	28th September 2020

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	☐
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	☐
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	☐

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Configuration Manual

Ghiridhar Iyer
X18183468

1 Introduction

The implementation of the artifact was done using the Amazon Web Services (AWS) platform. This configuration manual guides in executing the artifact. The following services were utilised during the implementation:

- Amazon S3
- AWS Athena
- AWS IAM
- AWS Sagemaker
- AWS QuickSight

The following sections are divided based on the implementation process.

2 Data Acquisition

Log in to the AWS Console at https://console.aws.amazon.com/

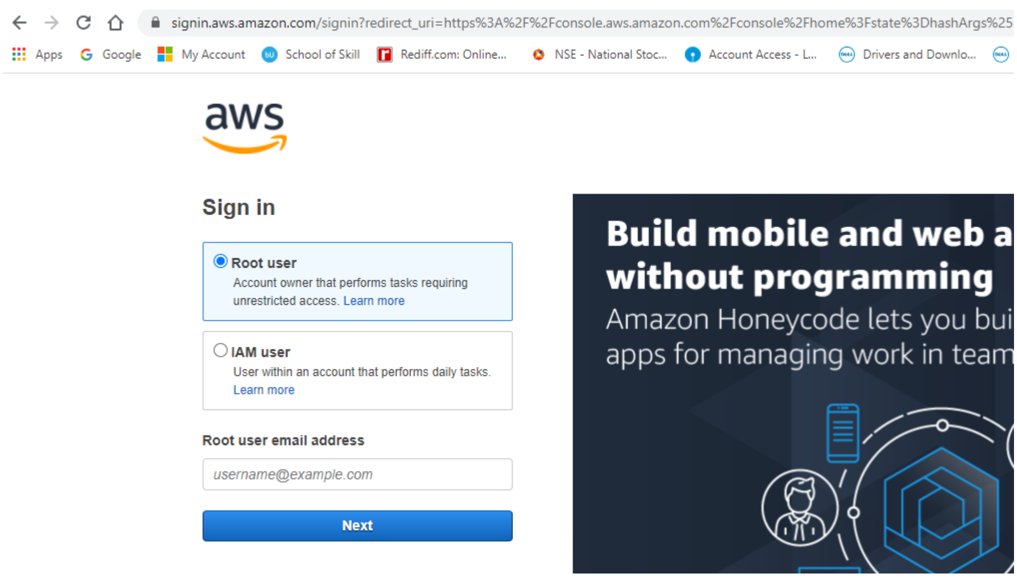


Figure 1: AWS Console Login

In the list of services, navigate to Storage section and click on S3.

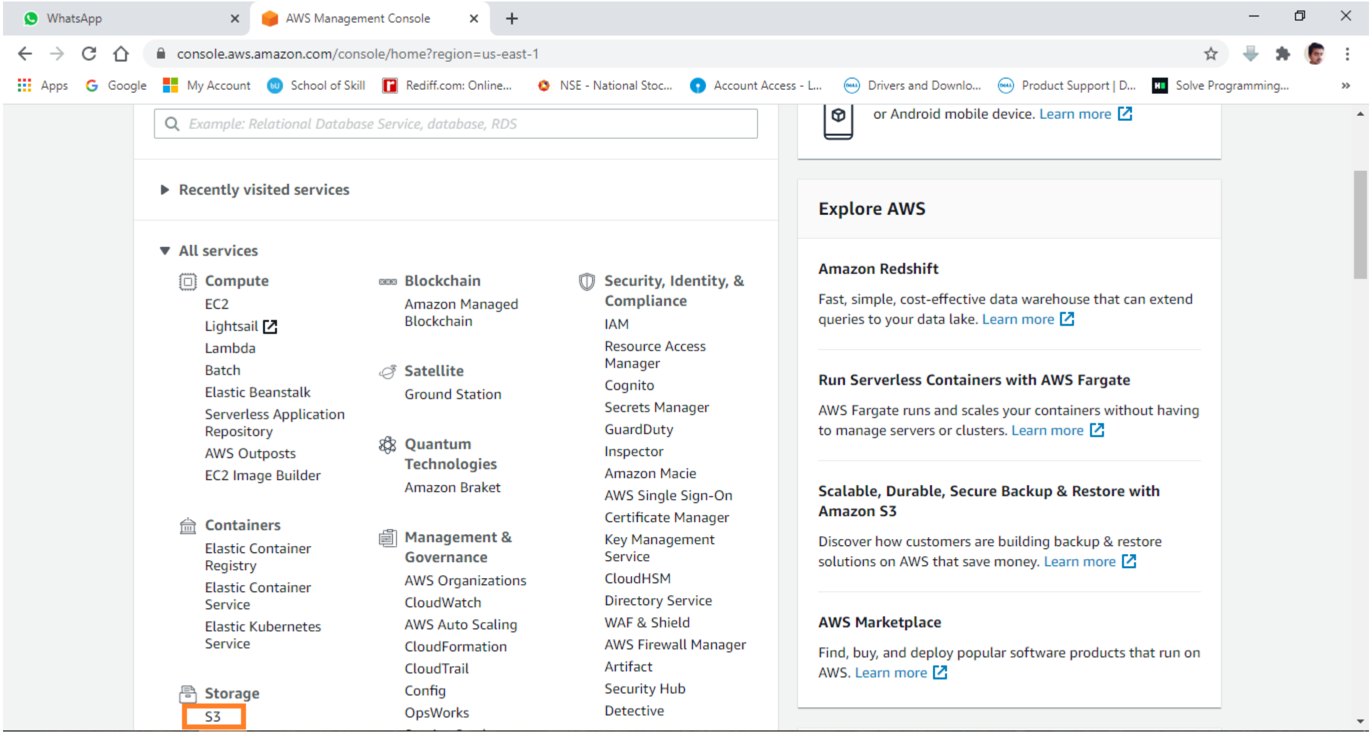


Figure 2: Click on S3

Click on 'Create Bucket' button in the page. Enter the bucket name as 'flood-prediction-master-dataset'. Since S3 is Global, Region needs to be explicitly mentioned while bucket creation. Ensure to have the S3 bucket in the same region where the rest of the services like Sagemaker are deployed. This manual has deployed all the services at the 'US East (N. Virginia)' region. **In the ML code, bucket name is explicitly mentioned. If this bucket name is unavailable and different Bucket name is being used, ensure to update the code with the new bucket name.**

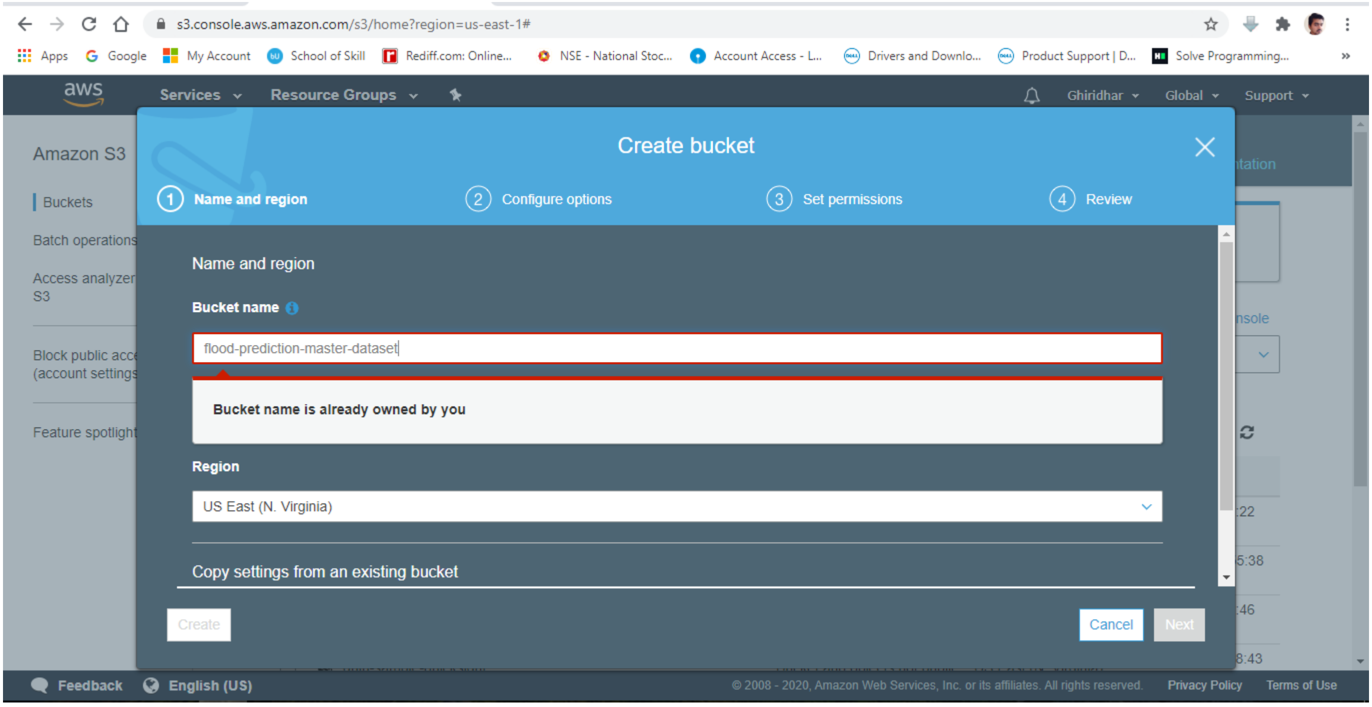


Figure 3: Create a bucket

Select the bucket and click on 'Edit Public access settings'.

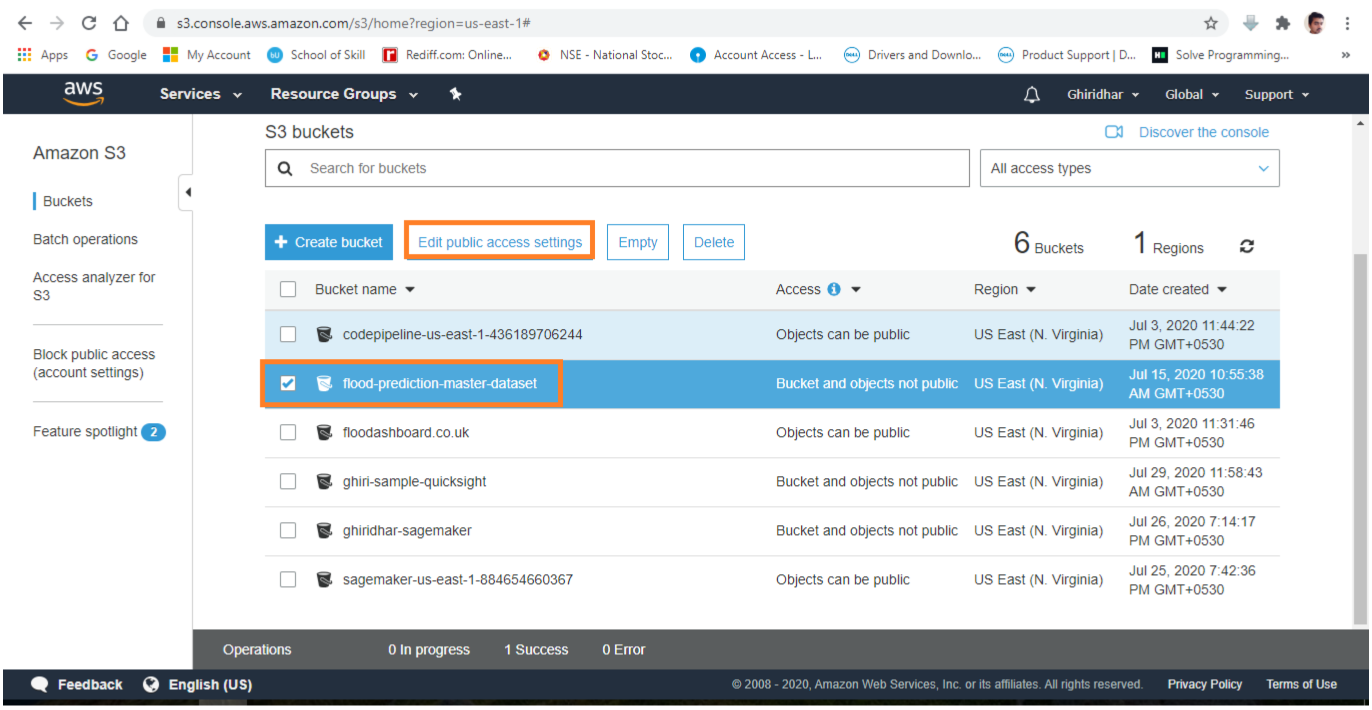


Figure 4: Change Public Access Settings

Deselect 'Block all public access' option. The reason for making bucket public will be explained in the upcoming steps. Click on Save button

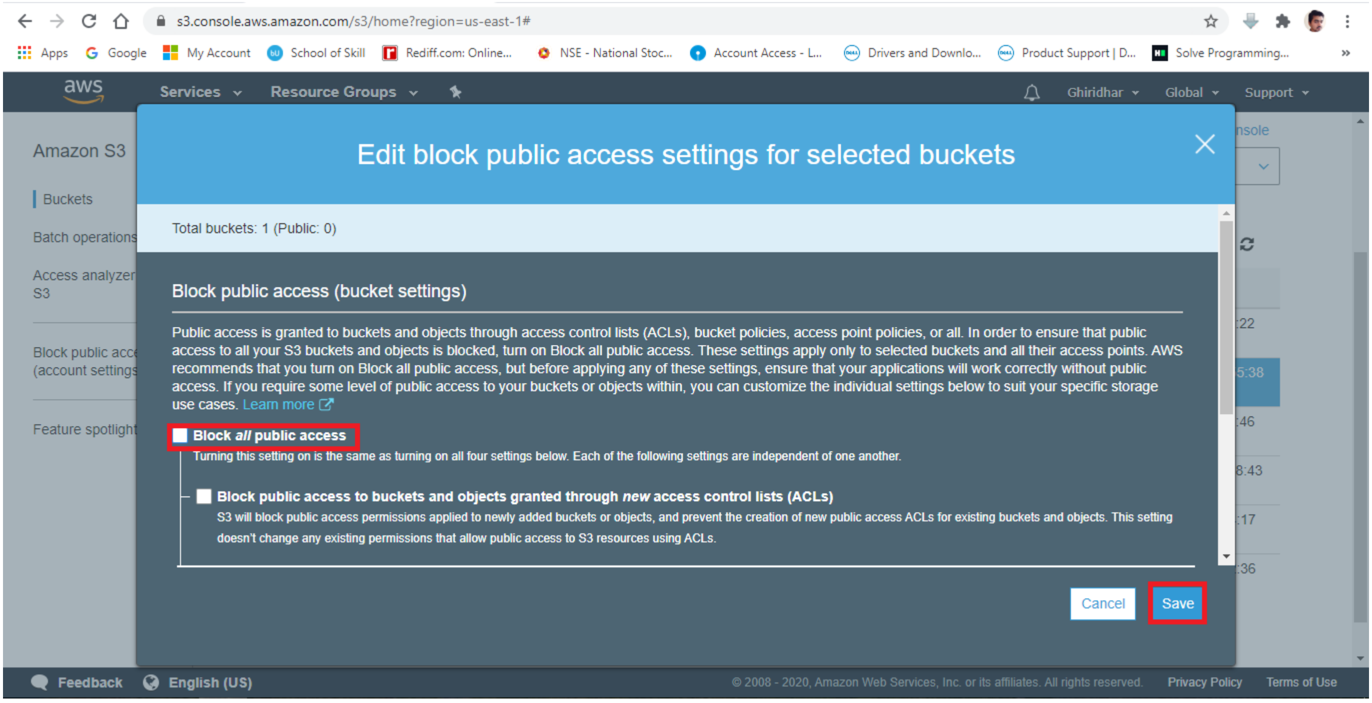


Figure 5: Unblock Public Access

Type 'confirm' in the textbox and click on confirm.

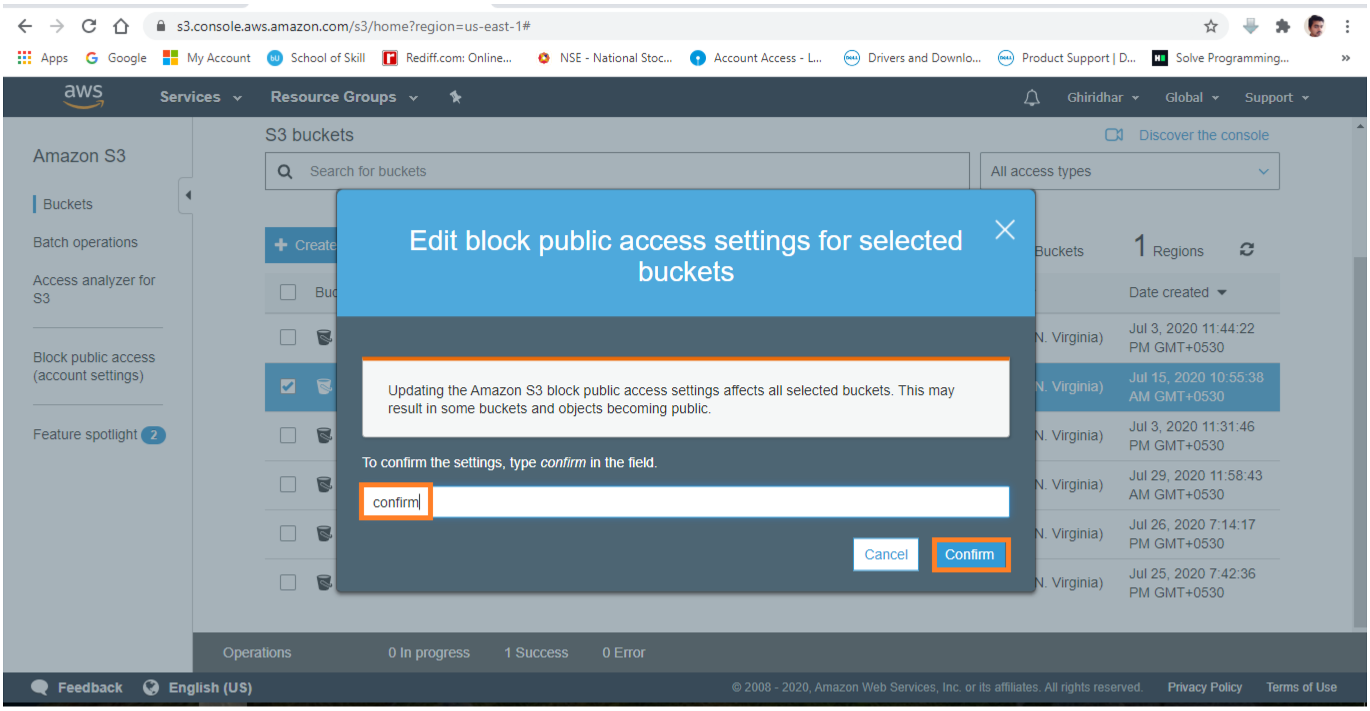


Figure 6: Type confirm

Click on the bucket. Click on 'Create Folder'. Enter the folder name as 'raw-datasets' and click on Save button.

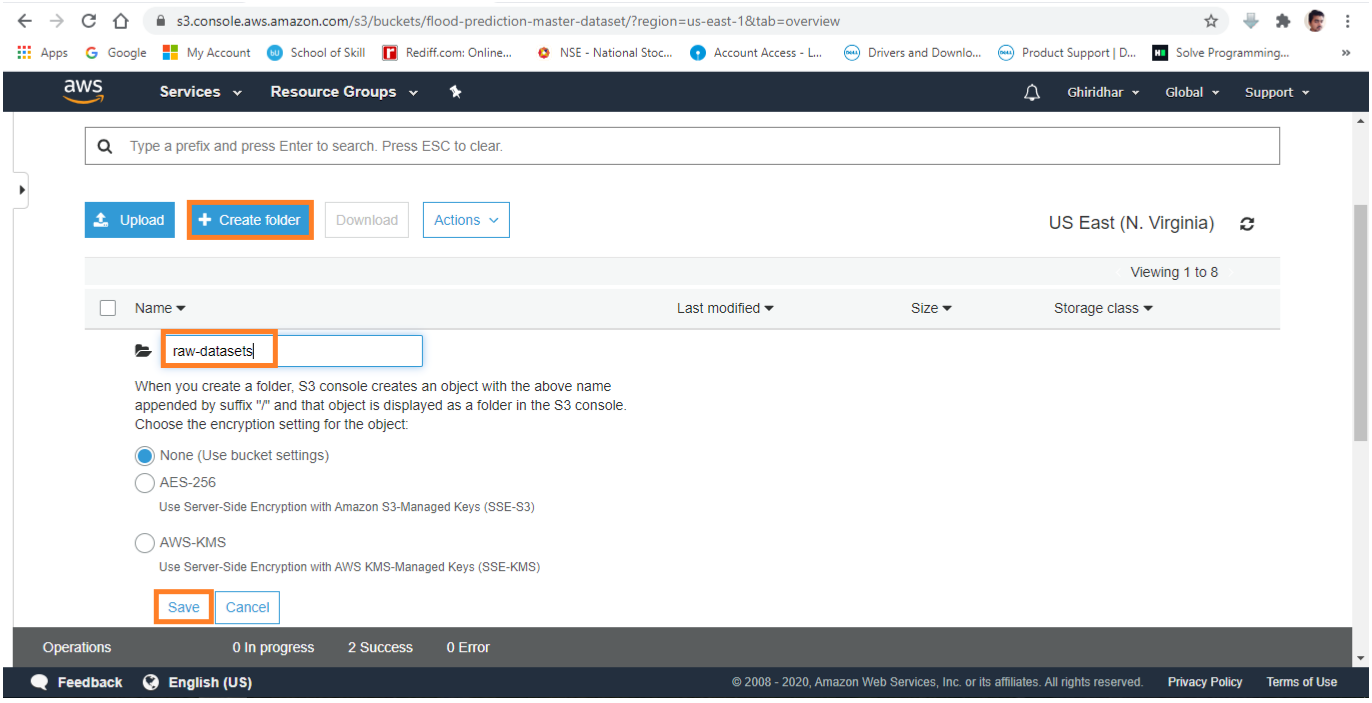


Figure 7: Create Folder in Bucket

Create all the folders as shown in the below Figure. 8.

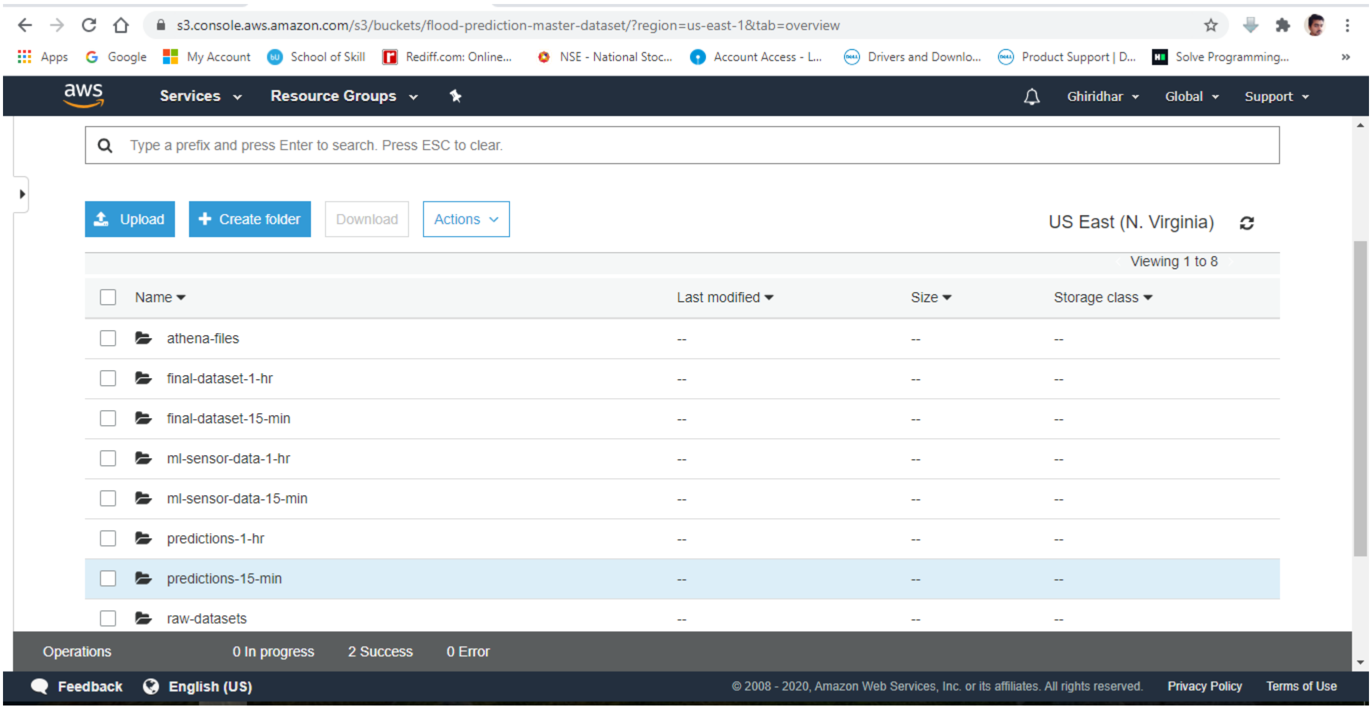


Figure 8: Create all these folders

Click on the Services drop down in the top left corner of the screen. Navigate to the Machine Learning section. Click on Amazon SageMaker.

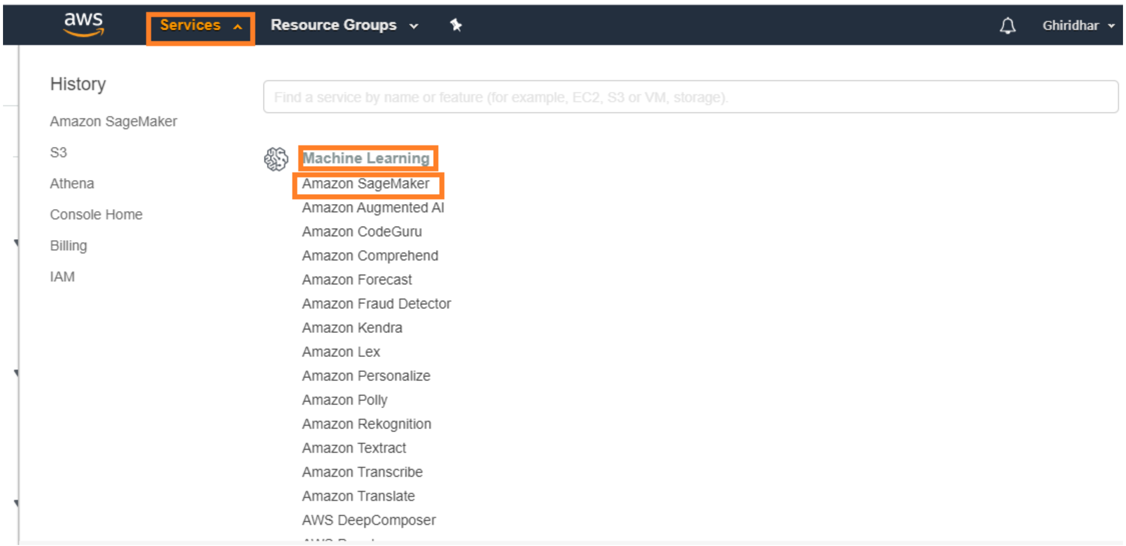


Figure 9: Click on Sagemaker

Click on Notebook instances from the menu on the left.

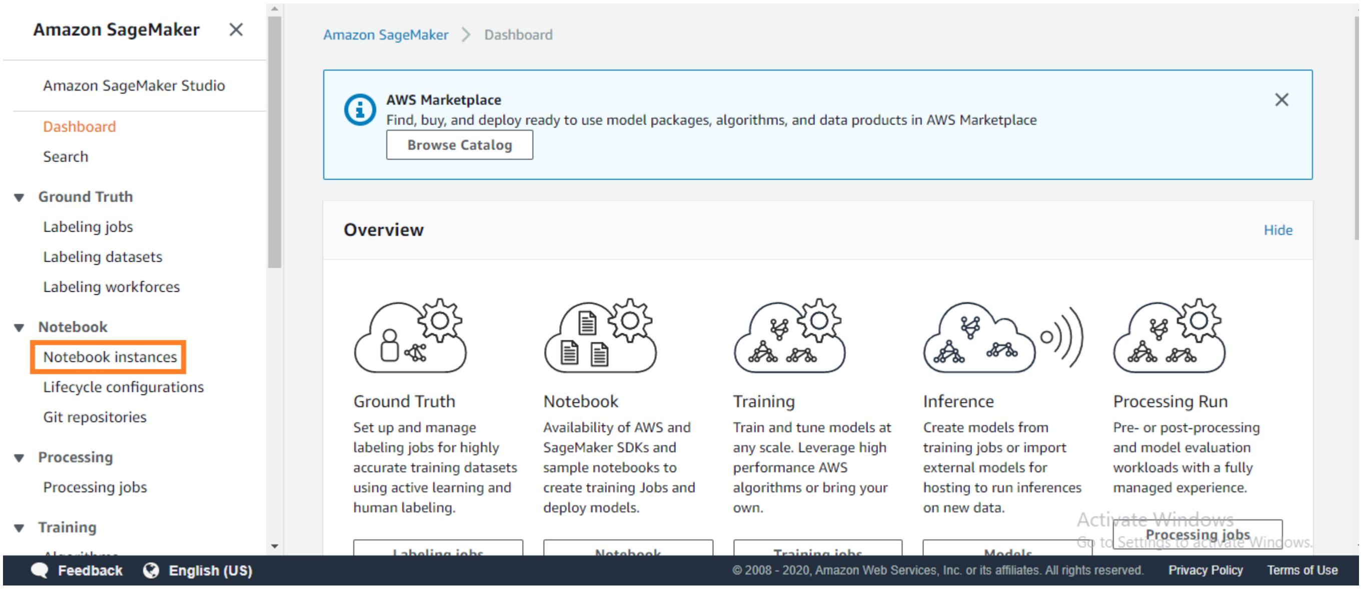


Figure 10: Click on Notebook Instances

Click on 'Create Notebook Instance'. Multiple Jupyter Notebooks can be created within an instance.

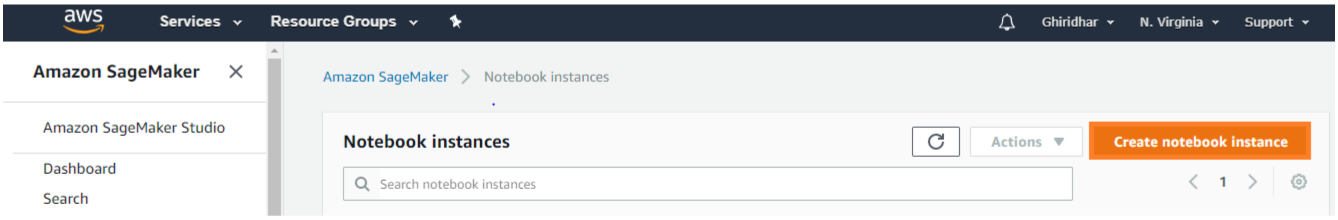


Figure 11: Click on Create Notebook Instance

Enter the Instance name and type.

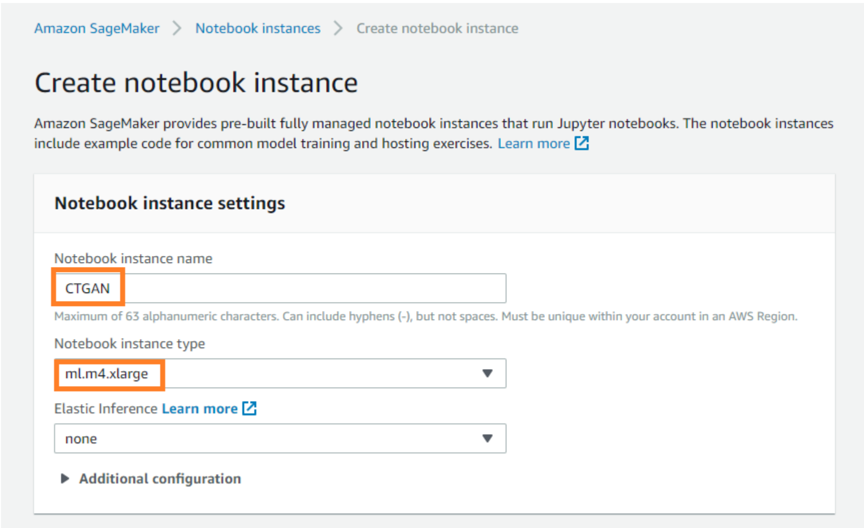


Figure 12: State the Name and Instance Type

Choose the IAM Role for the SageMaker instance. If not created choose create a new role.

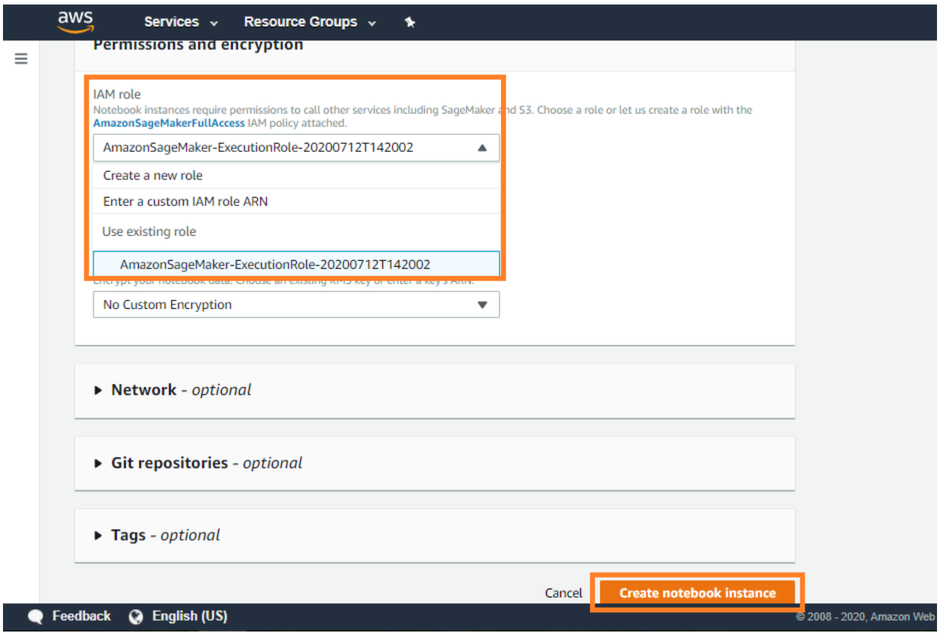


Figure 13: Select IAM Role

On clicking 'create a new role', the pop up asks to specify a particular bucket to provide access. Access can be provided to a single bucket, all buckets or no buckets. Select 'Specify S3 buckets' and enter 'flood-prediction-master-dataset'.

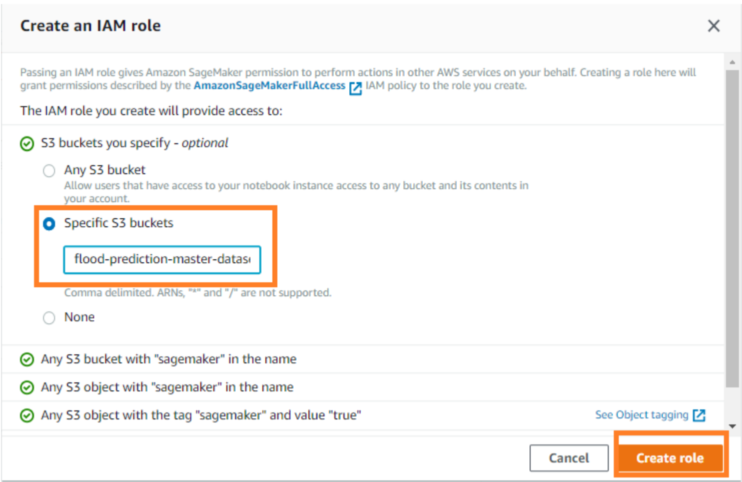


Figure 14: Specify S3 Bucket when creating IAM Role

Create Notebook instances as shown in the below figure 15. Ensure to provide 'ml.m4.xlarge' for any one instance. This instance is required for GAN generation. GAN generation is compute intensive process which is not possible in 'ml.t2.medium' or 'ml.t3.large' instances. Click on Open Jupyter link.

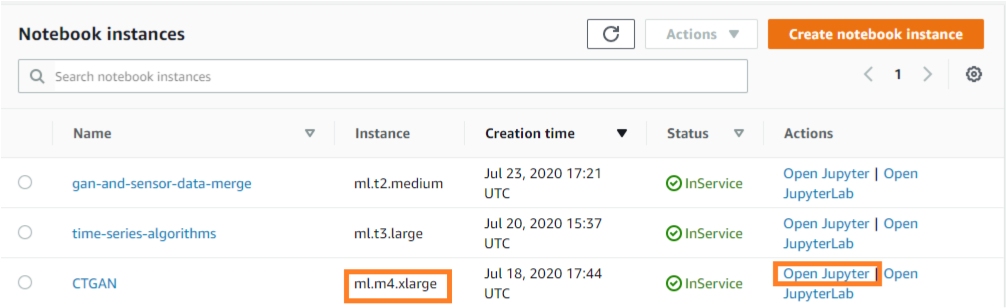


Figure 15: Click on Open Jupyter

Click on New and select 'conda_python3'. This creates a jupyter notebook instance with a Python 3 environment.

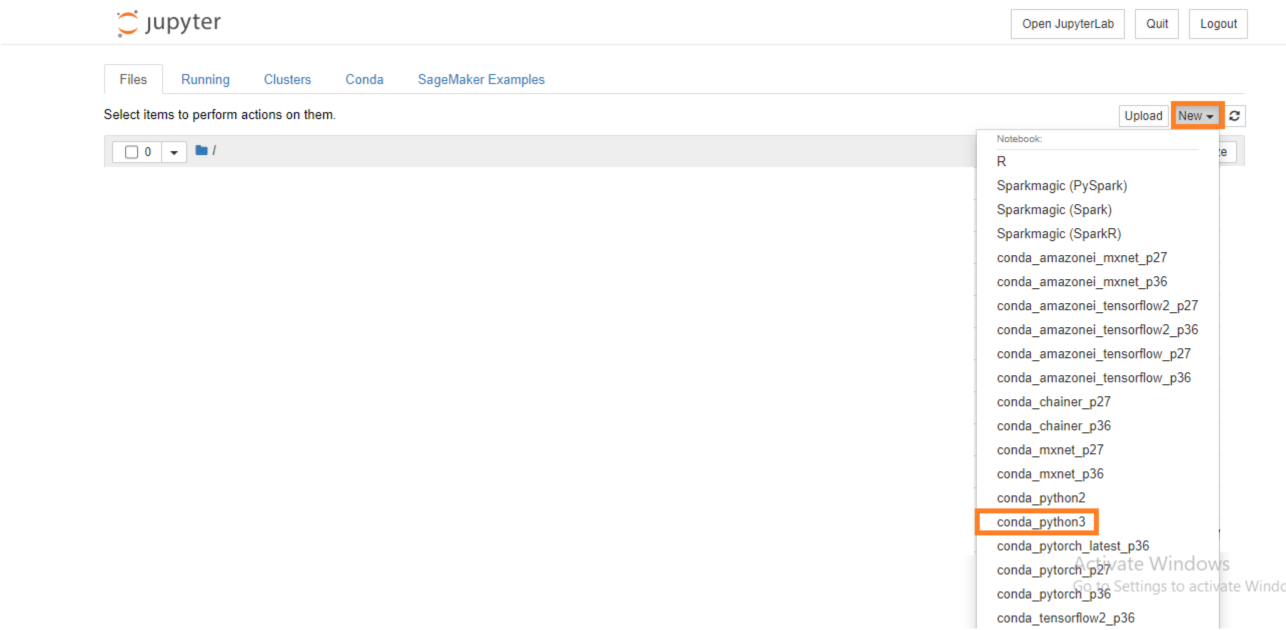


Figure 16: Create New Python3 File

Click on the title of the notebook stated as 'Untitled'. Pop up emerges to rename the file. Rename the file to 'Data_Acquisition' and click on Rename.

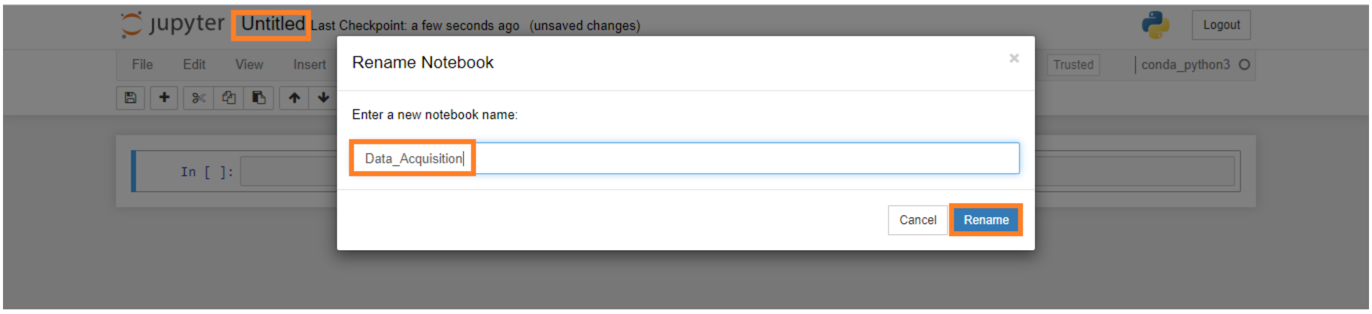


Figure 17: Rename File

Import libraries. The Environment agency API is used to retrieve the sensor data using pandas and saved as a CSV file. Public Access Rights is provided by the website [1].

```
In [ ]: #importing libraries

        import pandas as pd
```

```
In [ ]: ng datasets

        = pd.read_csv("https://environment.data.gov.uk/flood-monitoring/id/stations/550148/readings.csv?since=2017-01-01&_limit=10000")
        ure = pd.read_csv("https://environment.data.gov.uk/flood-monitoring/id/stations/3901/readings.csv?since=2017-01-01&_limit=10000")
        = pd.read_csv("https://environment.data.gov.uk/flood-monitoring/id/stations/694670/readings.csv?since=2017-01-01&_limit=10000")
```

```
In [ ]: #converting to csv

        rainfall.to_csv("Rainfall.csv",index=False)
        temperature.to_csv("Temperature.csv",index=False)
        rest_all.to_csv("Wind Speed Direction and River level.csv",index=False)
```

Figure 18: Acquire Data from API and save as CSV

Boto3 SDK [2] is used to access the S3 bucket and store the CSV file.

```
In [ ]: import datetime
        import tarfile

        import boto3 # AWS SDK for python. Provides low-level access to AWS services
        from sagemaker import get_execution_role
        import sagemaker

        m_boto3 = boto3.client('sagemaker')

        sess = sagemaker.Session()

        region = sess.boto_session.region_name

        bucket = 'flood-prediction-master-dataset'  #  specify the S3 bucket to save the file

        print('Using bucket ' + bucket)
```

```
In [ ]: # send data to S3. key_prefix is the directory path. path is the local file name which will be saved in S3 with same file name
        # and bucket is the bucket name

        upload = sess.upload_data(path='Rainfall.csv', bucket=bucket, key_prefix='raw-datasets/')

        upload = sess.upload_data(path='Temperature.csv', bucket=bucket, key_prefix='raw-datasets/')

        upload = sess.upload_data(path='Wind Speed Direction and River level.csv', bucket=bucket, key_prefix='raw-datasets/')
```

Activate Windows
Go to Settings to activat

Figure 19: Save to S3 using Boto3 SDK

[1] Public Access: https://environment.data.gov.uk/flood-monitoring/doc/reference
[2] Boto3 Documentation: https://boto3.amazonaws.com/v1/documentation/api/latest/guide/migrations3.html

3 Data Transformation and Formatting

Click on the services drop down and navigate to the Analytics section. Click on Athena. Athena provides SQL based data querying on S3 objects.

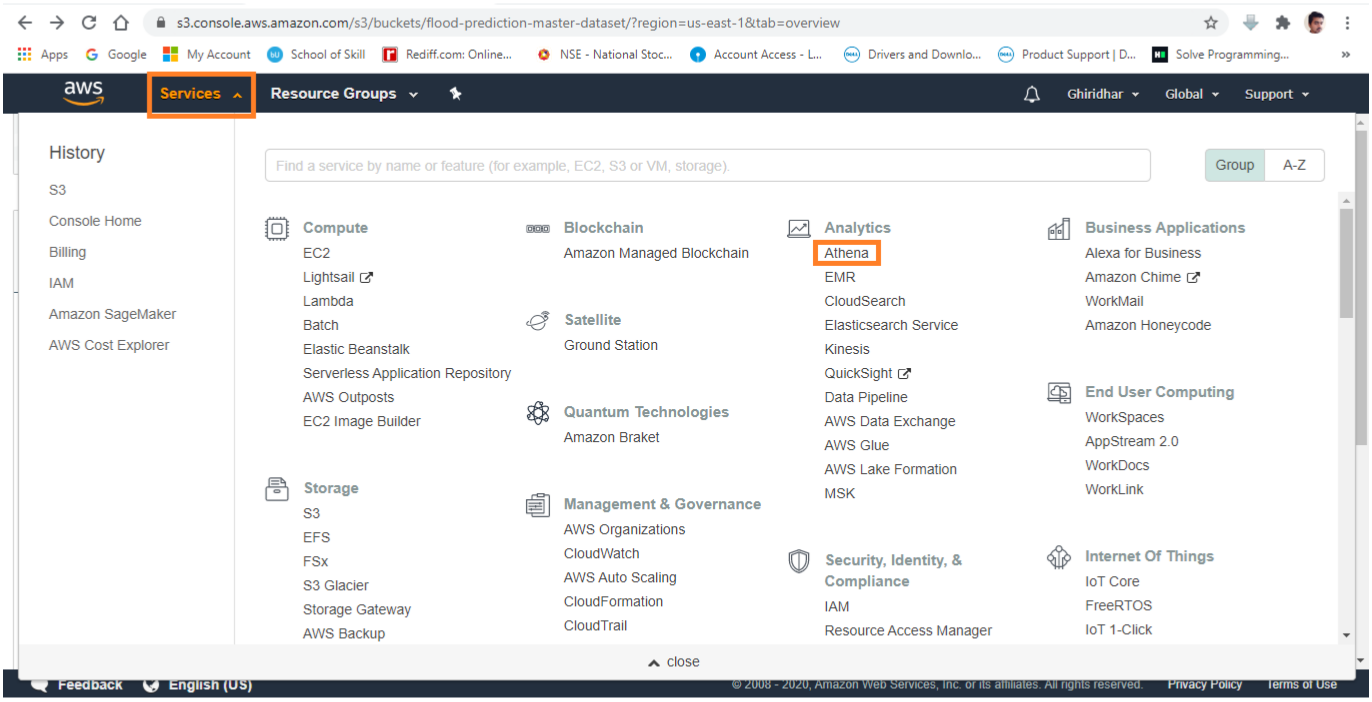


Figure 20: Click on Athena

Choose the Data source as S3 and metadata as AWS Glue. Click on Next.

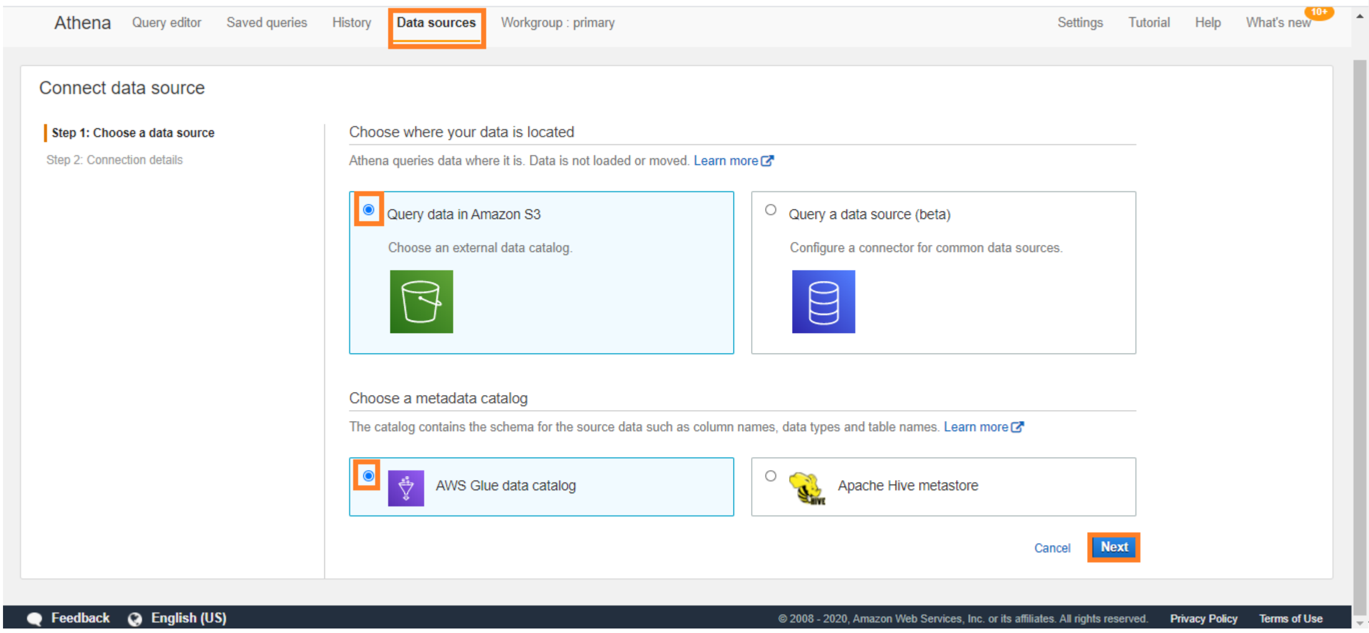


Figure 21: Choose S3 as Data Source

Choose to enter table schema manually and click on 'Continue to add Table'.

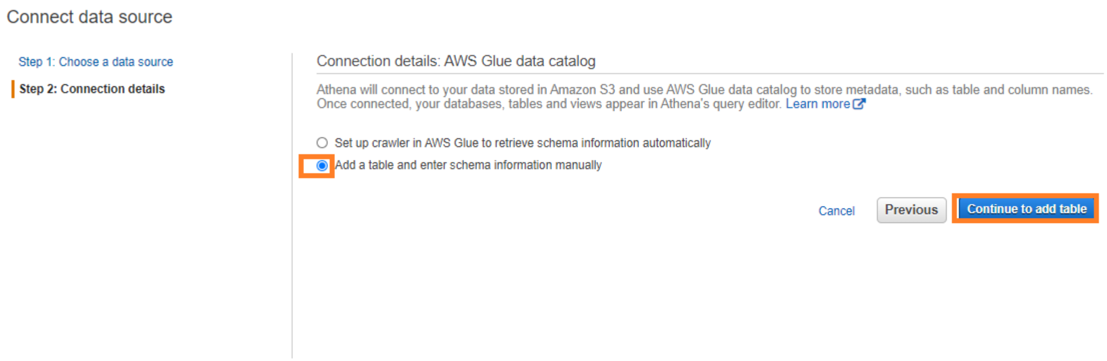


Figure 22: Choose to enter Table Structure Manually

Enter the Dataset name, Table name and S3 path to the raw datasets. The path should be a folder. Athena extracts all the data from the files as a single file. Click on Next.

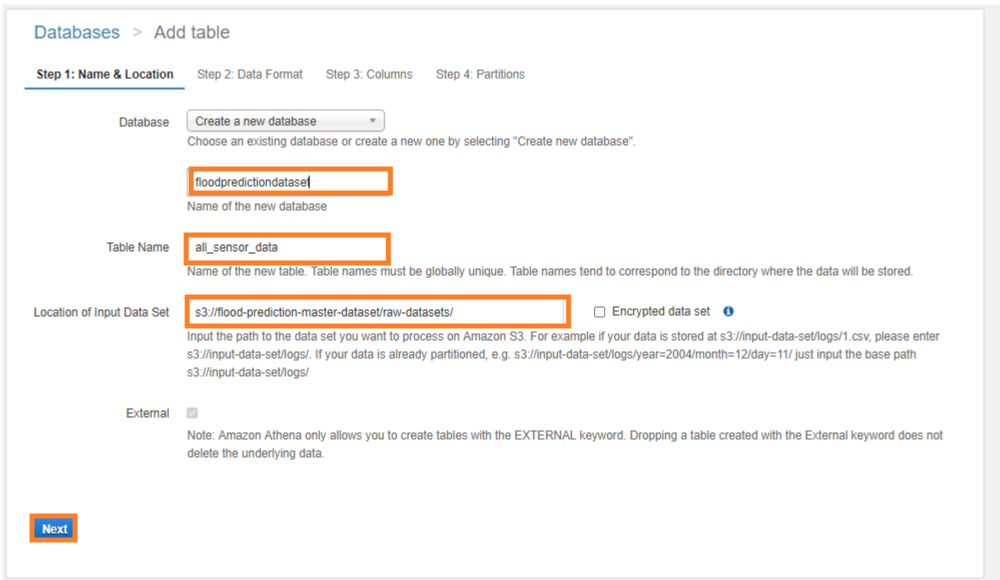


Figure 23: Enter Dataset and Table Name and provide path

Choose Datatype of the Data source as CSV and click on Next.

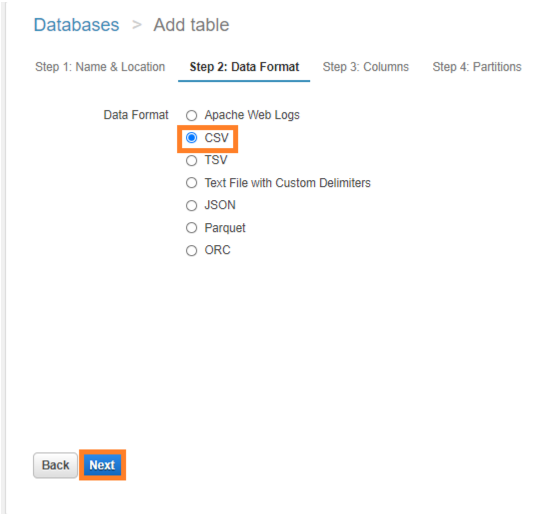


Figure 24: Set Datatype as CSV

Enter the schema of the table. Since the schema of all the three files is same, it does not affect the table schema process. In case the schema of each file differs, additional steps would be involved. Click on Add column to add new column schema. After entering the schema as per the figure below click on Next.

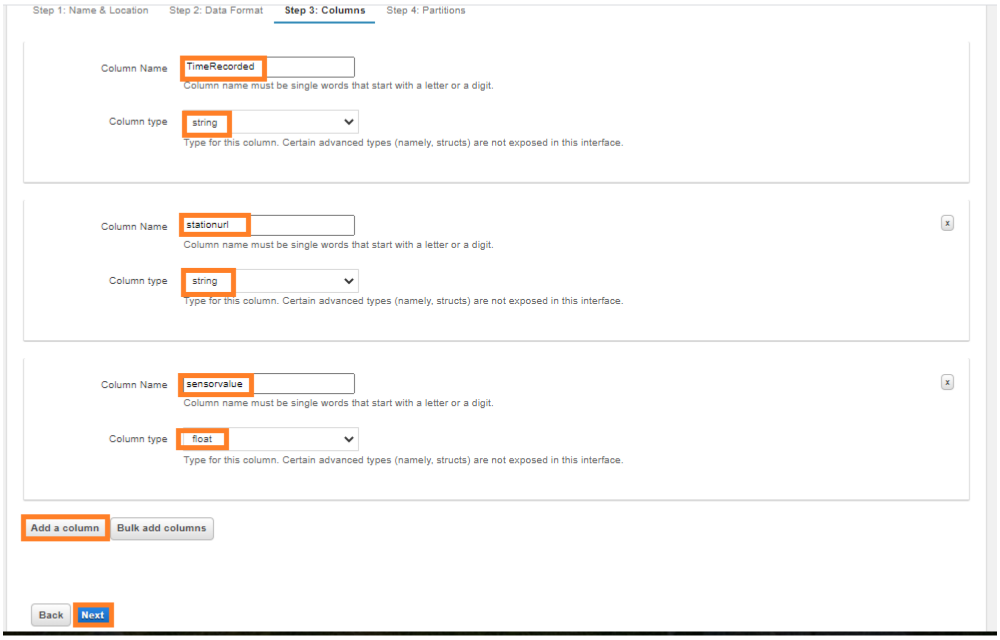


Figure 25: Enter Table Structure

Click on Create Table since Partitioning is not essential.

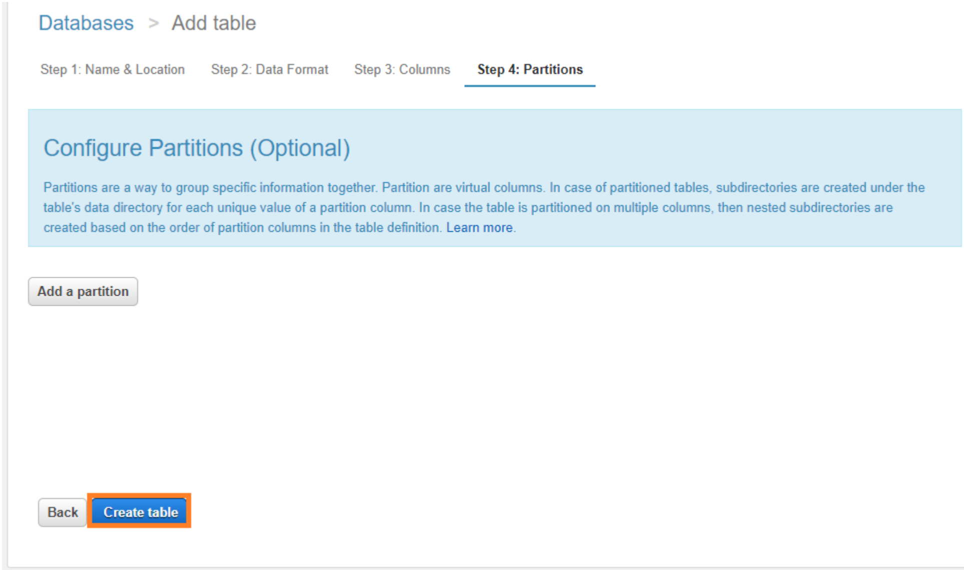


Figure 26: Create Table

In every tab, queries can be executed based on the tables created. Click on the '+' tab to add new tab. The code in the below figure selects the created table which contains all the raw data.

Figure 27: Show all Data

Selects data based on station. This provides five categories as output since there are five sensor data.

Figure 28: Group Data by sensor type

Creates a River level table based on sensor station value.

Figure 29: Create River level Table

Creates a Rainfall table based on sensor station value.

Figure 30: Create rainfall Table

Creates a Temperature table based on sensor station value.

Figure 31: Create temperature Table

Creates a Wind Speed table based on sensor station value.

Figure 32: Create Wind Speed Table

Creates a Wind Direction table based on sensor station value.

Figure 33: Create Wind Direction Table

Creates a table by applying a Join between River level table and rainfall table based on timestamp column.

Figure 34: Joining River level and rainfall

Creates a table by applying a Join with the above table and temperature table based on timestamp column.

Figure 35: Joining Temperature

Creates a table by applying a Join with the above table and Wind Speed table based on timestamp column.

Figure 36: Joining Speed

Creates a table by applying a Join with the above table and Wind Direction table based on timestamp column. This creates a table with 6 columns - one timestamp column and five sensor data columns with a time period of 15 minutes.

Figure 37: Final 15 minutes dataset

Aggregating the above table based on hour. Aggregate wind speed, wind direction and temperature to their average values. Aggregate rainfall to its sum value and river level to its max value.

Figure 38: Aggregated Dataset

Navigate to the S3 bucket inside the 'ml-sensor-data-15-min' folder. The table will be stored in a .gz format. Select the file and go to Actions. Click on 'Make Public' option. Although bucket was made public, objects are not public unless explicitly made public.

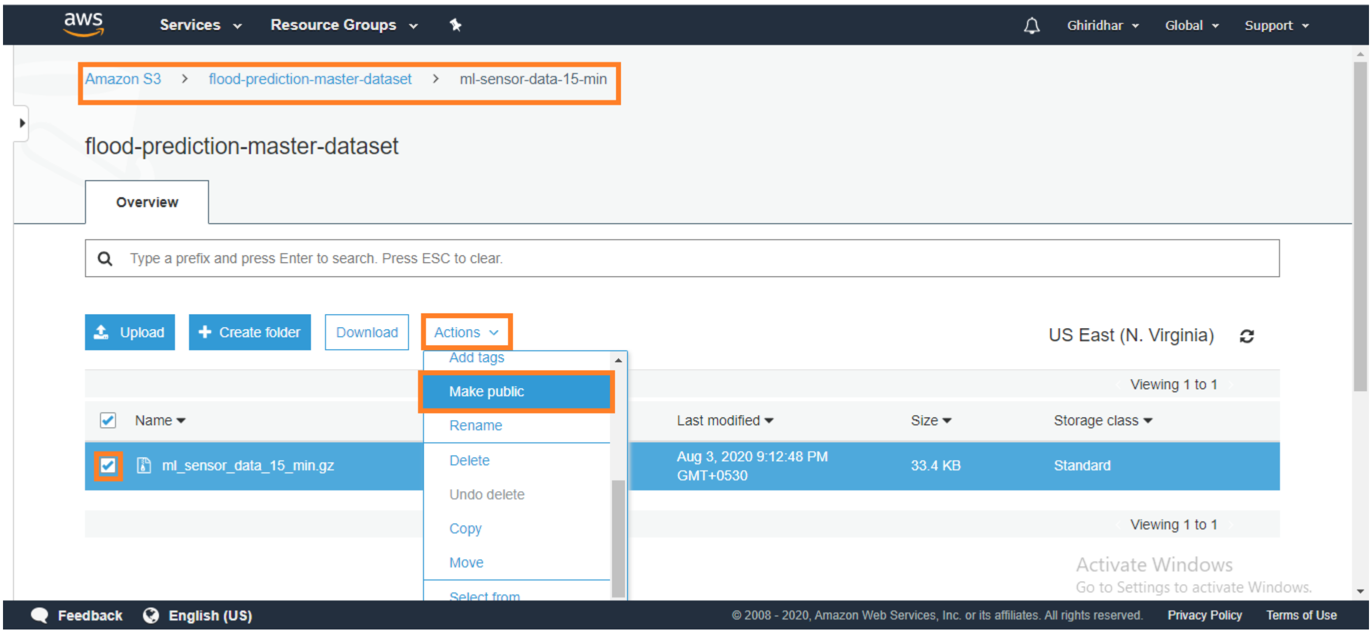


Figure 39: Making 15 Minute dataset public

Click on 'Make Public' to confirm the action. Do the same action for 1 hour dataset in the 'ml-sensor-data-1-hr' folder.

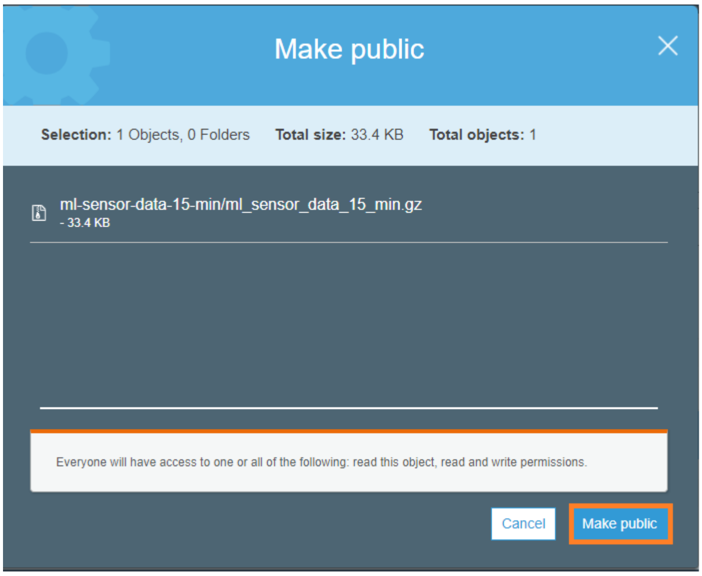


Figure 40: Confirming

Create jupyter python_3 file with file name 'Source_Dataset_gz_to_csv' inside the CT-GAN instance. Retrieve both the .gz files and save them as CSV files.

```python
In [ ]: #import packages

        import pandas as pd
```

```python
In [ ]: #import source file

        hr_file = pd.read_csv("https://flood-prediction-master-dataset.s3.amazonaws.com/ml-sensor-data-1-hr/ml_sensor_data_1_hr.gz", name
        min_file = pd.read_csv("https://flood-prediction-master-dataset.s3.amazonaws.com/ml-sensor-data-15-min/ml_sensor_data_15_min.gz",
```

```python
In [ ]: #saving as csv file

        hr_file.to_csv("ml_sensor_data_1_hr.csv",index=False,header=None)
        min_file.to_csv("ml_sensor_data_15_min.csv",index=False,header=None)
```

Figure 41: retrieve file using pandas

Using Boto3, save the CSV files to their respective folders in S3. This Format conversion was done by making the bucket and the .gz file as public, as Boto3 faces issues while streaming and decoding the .gz file.

```python
In [ ]: import datetime
        import tarfile

        import boto3 # AWS SDK for python. Provides low-level access to AWS services
        from sagemaker import get_execution_role
        import sagemaker

        m_boto3 = boto3.client('sagemaker')

        sess = sagemaker.Session()

        region = sess.boto_session.region_name

        bucket = 'flood-prediction-master-dataset'  #  specify the S3 bucket to save the file

        print('Using bucket ' + bucket)
```

```python
In [ ]: # send data to S3. key_prefix is the directory path. path is the local file name which will be saved in S3 with same file name
        # and bucket is the bucket name

        upload = sess.upload_data(path='ml_sensor_data_1_hr.csv', bucket=bucket, key_prefix='ml-sensor-data-1-hr/')

        upload = sess.upload_data(path='ml_sensor_data_15_min.csv', bucket=bucket, key_prefix='ml-sensor-data-15-min/')
```

Figure 42: Save CSV format to S3

Navigate to the S3 bucket and inside the 'ml-sensor-data-15-min' folder. Since the .gz and .csv have the same name, a no named folder is created within which the CSV file is saved.

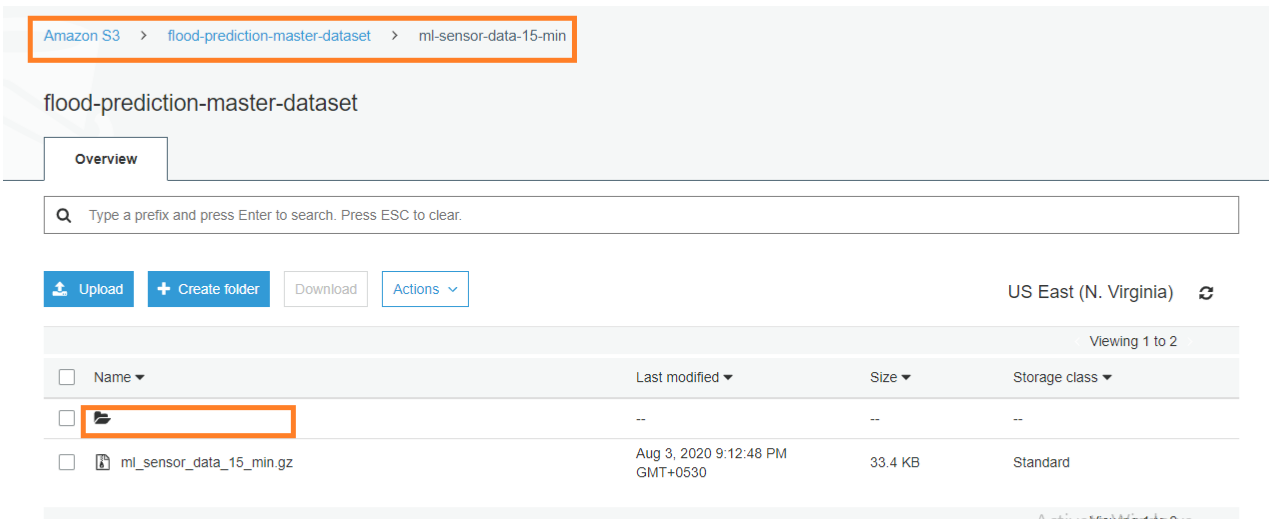


Figure 43: Go inside Empty Folder

Click on the folder. Select the CSV file and go to actions. Click on Move.

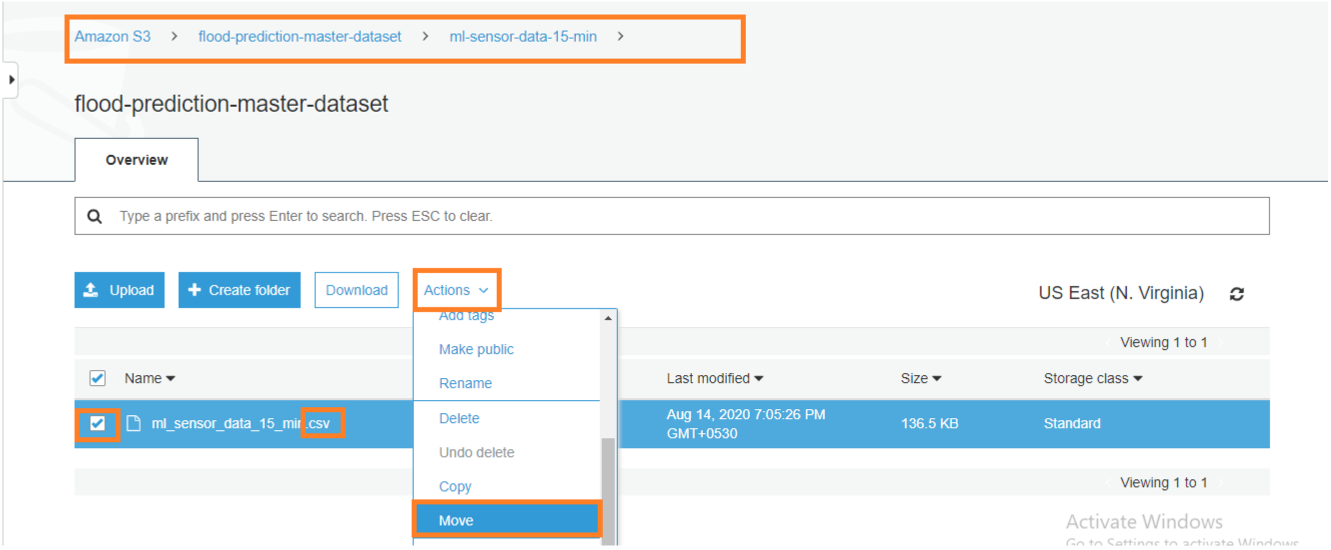


Figure 44: Move File

Click on the 'flood-prediction-master-dataset' S3 bucket. Do not choose the bucket, but click on the bucket name.

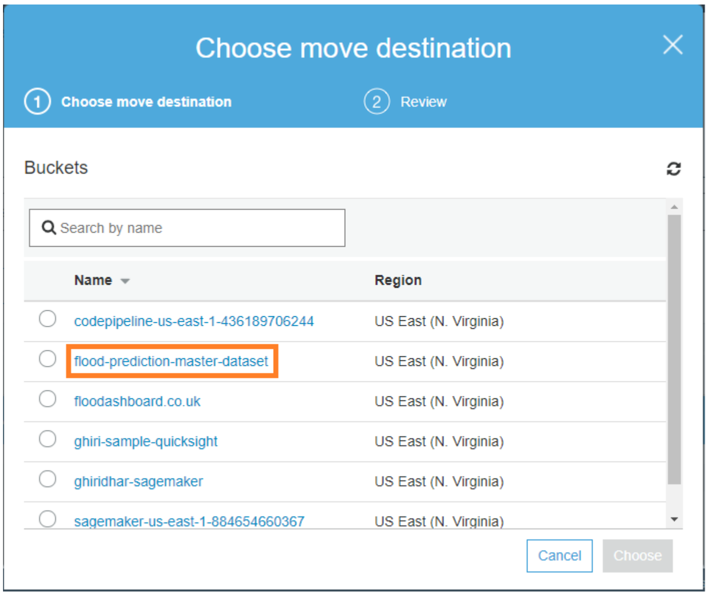


Figure 45: Click on Destination S3 bucket

Choose the folder to move the file. Here select the 'ml-sensor-data-15-min'. Click on Choose.

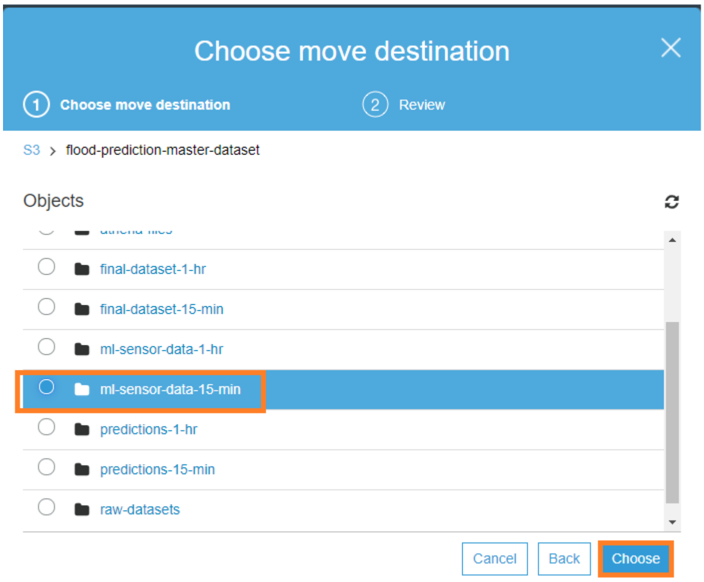


Figure 46: Choose the destination folder

Click on Move to confirm the action.

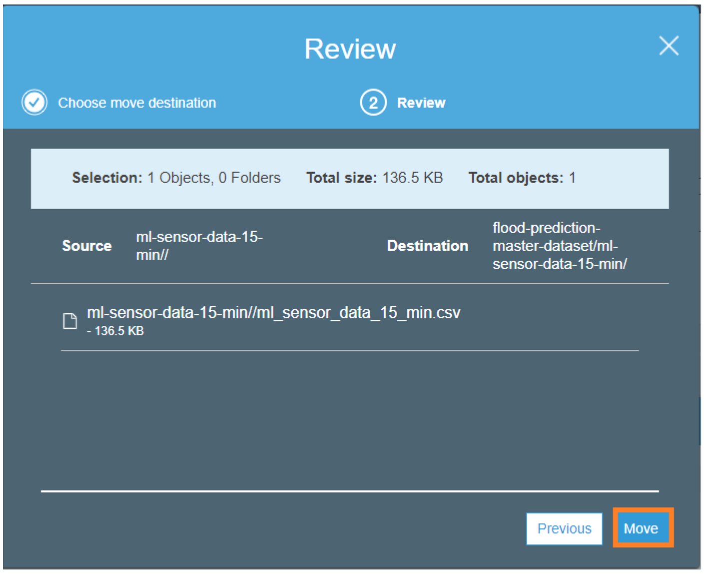


Figure 47: Confirm Move Action

Ensure the process was successful. The no named folder is deleted by itself. Do the same process for the 1 hour CSV file.

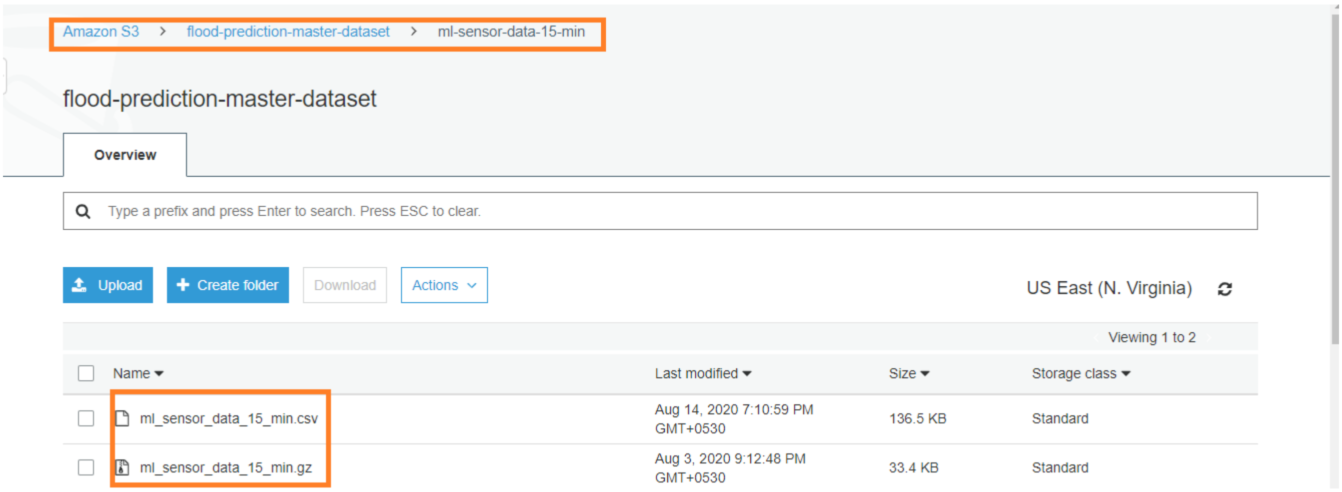


Figure 48: Ensure both files are inside the same directory.

Choose the bucket and click on 'Edit Public Access Settings'.

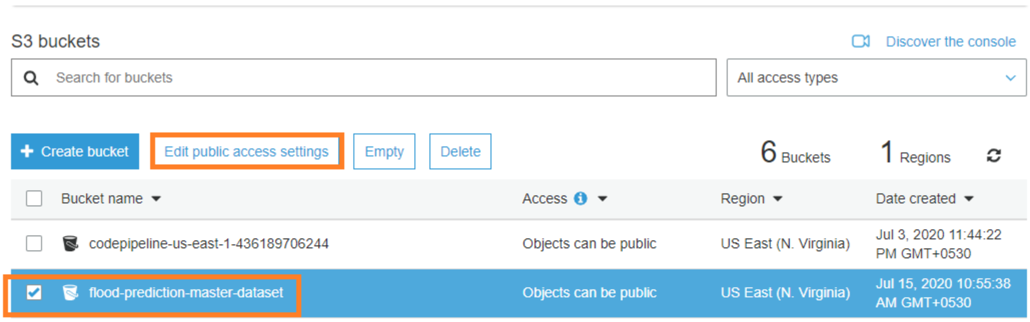


Figure 49: Edit Bucket Public Access

Check on the 'Block all public access' option and click on Save.

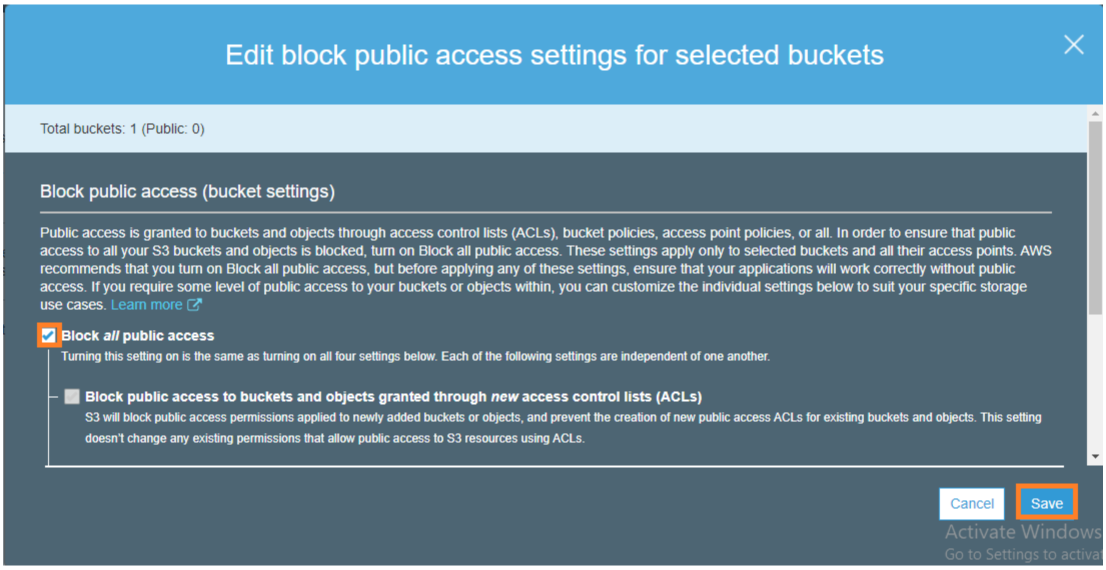


Figure 50: Block access

Type 'confirm' in the textbox and click on Confirm. This converts all the public objects inside the bucket as private.

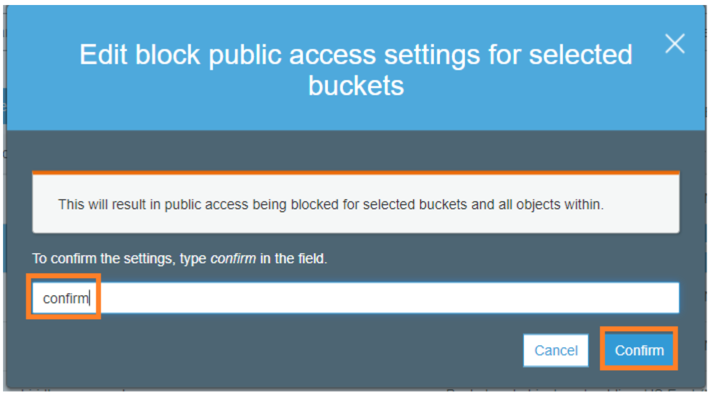


Figure 51: Confirm Action

4 GAN creation and Merging

Create a Python3 file named GAN_generator_15_min in CTGAN instance. Go to Kernel and go to Change Kernel and Choose 'conda_mxnet_p36'. This Kernel enables installing Third Party libraries in AWS SageMaker environment.

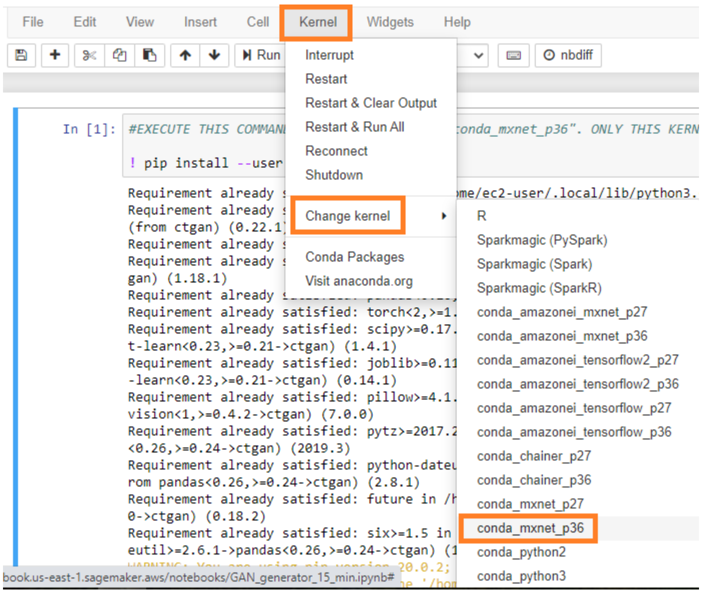


Figure 52: Change Kernal

Install the ctgan library using the pip command.

```
In [ ]: #EXECUTE THIS COMMAND WITH KERNAL SET TO "conda_mxnet_p36". ONLY THIS KERNALCAN INSTALL EXTERNAL LIBRARIES.
        ! pip install --user ctgan
```

Figure 53: Install CTGAN library

Change the kernel back to 'conda_python3'.

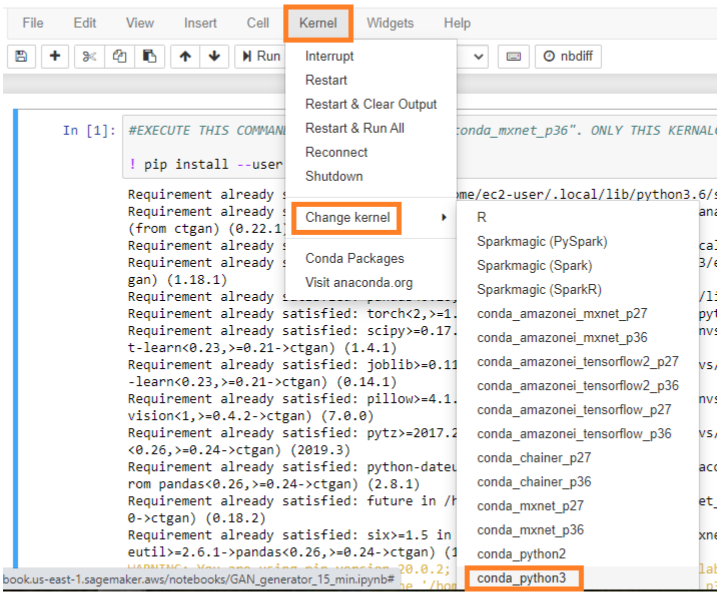


Figure 54: Change Kernal to Python3

Import all the required libraries. Stream the 15 minutes file from S3 and specify the column names.

```python
#import packages

from ctgan import CTGANSynthesizer
import pandas as pd
import boto3
```

```python
#import source file

s3 = boto3.client('s3')

bucket = 'flood-prediction-master-dataset'
key = 'ml-sensor-data-15-min/ml_sensor_data_15_min.csv'

obj = s3.get_object(Bucket= bucket,Key= key)

file = pd.read_csv(obj['Body'],names=["time","river","rain","temperature","wind_direction","wind_speed"])

file
```

```python
#specify the headers to GAN

discrete_columns = [
    'time',
    'river',
    'rain',
    'temperature',
    'wind_direction',
    'wind_speed'
]
```

Activate Windows
Go to Settings to activate

Figure 55: Retrieve file and mention columns

Declare the CTGAN, train the model and generate records.

```python
#creating instance

ctgan = CTGANSynthesizer()
```

```python
#training

ctgan.fit(file, discrete_columns)
```

```python
#sample creation

samples = ctgan.sample(2758)
```

```python
import datetime
import tarfile

import boto3 # AWS SDK for python. Provides low-level access to AWS services
from sagemaker import get_execution_role
import sagemaker

m_boto3 = boto3.client('sagemaker')

sess = sagemaker.Session()

region = sess.boto_session.region_name

bucket = 'flood-prediction-master-dataset'  #  specify the S3 bucket to save the file

print('Using bucket ' + bucket)
```

Activate Windows
Go to Settings to activate

Figure 56: Train and Create samples.

Save the Generated file as CSV and using Boto3 save it to S3.

```python
#save the file as csv locally

samples.to_csv("gan_data_15_min.csv")
```

```python
# send data to S3. key_prefix is the directory path. path is the local file name which will be saved in S3 with same file name
# and bucket is the bucket name

upload = sess.upload_data(path='gan_data_15_min.csv', bucket=bucket, key_prefix='ml-sensor-data-15-min/')
```

Figure 57: Convert to CSV and save to S3

Create a python3 file named GAN_generator_1_hr in CTGAN instance. Repeat the same process as in the above figures 52 to 55. Stream the 1 hour dataset and train the model.

```
In [ ]: #EXECUTE THIS COMMAND WITH KERNAL SET TO "conda_mxnet_p36". ONLY THIS KERNALCAN INSTALL EXTERNAL LIBRARIES.

        ! pip install --user ctgan

In [ ]: #import packages

        from ctgan import CTGANSynthesizer
        import pandas as pd
        import boto3

In [ ]: #import source file

        s3 = boto3.client('s3')

        bucket = 'flood-prediction-master-dataset'
        key = 'ml-sensor-data-1-hr/ml_sensor_data_1_hr.csv'

        obj = s3.get_object(Bucket= bucket,Key= key)

        file = pd.read_csv(obj['Body'],names=["time","river","rain","temperature","wind_direction","wind_speed"])

        file

In [ ]: #specify the headers to GAN

        discrete_columns = [
            'time',
            'river',
            'rain',
            'temperature',
            'wind_direction',
            'wind_speed'
        ]

In [ ]: #creating instance

        ctgan = CTGANSynthesizer()

In [ ]: #training

        ctgan.fit(file, discrete_columns)
```

Figure 58: GAN generation for 1 Hour data

Generate sample records, save the file as CSV and stream it to S3 using Boto3.

```
In [ ]: #sample creation

        samples = ctgan.sample(692)

In [ ]: import datetime
        import tarfile

        import boto3 # AWS SDK for python. Provides low-level access to AWS services
        from sagemaker import get_execution_role
        import sagemaker

        m_boto3 = boto3.client('sagemaker')

        sess = sagemaker.Session()

        region = sess.boto_session.region_name

        bucket = 'flood-prediction-master-dataset'  #  specify the S3 bucket to save the file

        print('Using bucket ' + bucket)

In [ ]: #save the file as csv locally

        samples.to_csv("gan_data_1_hr.csv")

In [ ]: # send data to S3. key_prefix is the directory path. path is the local file name which will be saved in S3 with same file name
        # and bucket is the bucket name

        upload = sess.upload_data(path='gan_data_1_hr.csv', bucket=bucket, key_prefix='ml-sensor-data-1-hr/'
```

Figure 59: Create sample and upload to S3

Now create a Python3 notebook named Sensor_and_GAN_merger_15_min in 'gan-and-sensor-data-merge' instance. Import libraries and Stream the 15 minutes sensor and GAN data.

```
In [ ]: #packages

        import pandas as pd
        import numpy as np
        import boto3
```

```
In [ ]: #file import from S3

        s3 = boto3.client('s3')

        bucket = 'flood-prediction-master-dataset'
        key = 'ml-sensor-data-15-min/ml_sensor_data_15_min.csv'

        obj = s3.get_object(Bucket= bucket,Key= key)
        sensor_file = pd.read_csv(obj['Body'],names=["time","river","rain","temperature","wind_direction","wind_speed"])

        key = 'ml-sensor-data-15-min/gan_data_15_min.csv'

        obj = s3.get_object(Bucket= bucket,Key= key)
        gan_file = pd.read_csv(obj['Body'])  #GAN was saved with header. hence no need for stating column names
```

Figure 60: Retrieve both files

Add DateTime column and source column. Source column states the source of the record - GAN or sensor. Combine both files

```
In [ ]: #converting the string datatype of time values to datetime

        sensor_file['timerecorded'] = pd.to_datetime(sensor_file['time'])
        gan_file['timerecorded'] = pd.to_datetime(gan_file['time'])

In [ ]: #dropping the redundant column

        sensor_file.drop(['time'],axis=1,inplace=True)
        gan_file.drop(['time'],axis=1,inplace=True)

In [ ]: #sorting by datetime column

        gan_file.sort_values(by=['timerecorded'],inplace=True)
        sensor_file.sort_values(by=['timerecorded'],inplace=True)

In [ ]: #quicksight imports all data from all the csv files from the folder and files specified.
        #Hence, a distinguishing column is required.

        gan_file['source'] = 'GAN'
        sensor_file['source'] = 'SENSOR'

In [ ]: #merging both datasets

        final_file_15_min = gan_file.append(sensor_file)
```

Activate Windows

Figure 61: Add Time and source column

Find the mean of each column for sensor and GAN data to assess how close the values are. Save the file as CSV.

```
In [ ]: print("Mean of River GAN: "+str(round(gan_file['river'].mean(),4))+" Mean of River Sensor: "+str(round(sensor_file['river'].mean(

        print("Mean of Rainfall GAN: "+str(round(gan_file['rain'].mean(),4))+" Mean of Rainfall Sensor: "+str(round(sensor_file['rain'].m

        print("Mean of Temperature GAN: "+str(round(gan_file['temperature'].mean(),4))+" Mean of Temperature Sensor: "+str(round(sensor_f

        print("Mean of wind direction GAN: "+str(round(gan_file['wind_direction'].mean(),4))+" Mean of wind direction Sensor: "+str(rounc

        print("Mean of wind speed GAN: "+str(round(gan_file['wind_speed'].mean(),4))+" Mean of wind speed Sensor: "+str(round(sensor_file

In [ ]: final_file_15_min

In [ ]: #reset file index to current dataframe

        final_file_15_min.reset_index(drop=True, inplace=True)

In [ ]: #save as a csv file locally

        final_file_15_min.to_csv("final_data_15_min.csv")
```

Activate Windows
Go to Settings to activat

Figure 62: Mean comparison and convert to CSV

Using Boto3, save the file to S3.

```
In [ ]: import datetime
        import tarfile

        import boto3 # AWS SDK for python. Provides low-level access to AWS services
        from sagemaker import get_execution_role
        import sagemaker

        m_boto3 = boto3.client('sagemaker')

        sess = sagemaker.Session()

        region = sess.boto_session.region_name

        bucket = 'flood-prediction-master-dataset'  #  Bucket name where the file is to be saved

        print('Using bucket ' + bucket)

        # save data to S3. key_prefix is the directory path. path is the local file name which will be saved in S3. It has to be
        # the same file name as saved locally and bucket is the bucket name

        upload = sess.upload_data(path='final data 15 min.csv', bucket=bucket, key_prefix='final-dataset-15-min')
```

Figure 63: save to S3 folder

Similarly, create a Python3 notebook named Sensor_and_GAN_merger_1_hr in 'gan-and-sensor-data-merge' instance. Import libraries and Stream the 1 hour sensor and GAN data.

```
In [ ]: #packages

        import pandas as pd
        import numpy as np
        import boto3

In [ ]: #file import from S3

        s3 = boto3.client('s3')

        bucket = 'flood-prediction-master-dataset'
        key = 'ml-sensor-data-1-hr/ml_sensor_data_1_hr.csv'

        obj = s3.get_object(Bucket= bucket,Key= key)
        sensor_file = pd.read_csv(obj['Body'],names=["time","river","rain","temperature","wind_direction","wind_speed"])

        key = 'ml-sensor-data-1-hr/gan_data_1_hr.csv'

        obj = s3.get_object(Bucket= bucket,Key= key)
        gan_file = pd.read_csv(obj['Body'])  #GAN was saved with header. hence no need for stating column names
```

Figure 64: 1 hour data import

Create DateTime column and source column. Combine both files.

```
In [ ]: #converting the string datatype of time values to datetime

        sensor_file['timerecorded'] = pd.to_datetime(sensor_file['time'])
        gan_file['timerecorded'] = pd.to_datetime(gan_file['time'])

In [ ]: #dropping the redundant column

        sensor_file.drop(['time'],axis=1,inplace=True)
        gan_file.drop(['time'],axis=1,inplace=True)

In [ ]: #sorting by datetime column

        gan_file.sort_values(by=['timerecorded'],inplace=True)
        sensor_file.sort_values(by=['timerecorded'],inplace=True)

In [ ]: #quicksight imports all data from all the csv files from the folder and files specified.
        #Hence, a distinguishing column is required.

        gan_file['source'] = 'GAN'
        sensor_file['source'] = 'SENSOR'

In [ ]: #merge both datasets

        final_file_1_hr = gan_file.append(sensor_file)
```

Activate Windows

Figure 65: Adding source and time columns

Save the file as CSV and save the file to S3.

```
In [ ]: #reset file index to current dataframe

        final_file_1_hr.reset_index(drop=True, inplace=True)
```

```
In [ ]: #save as a csv file locally

        final_file_1_hr.to_csv("final_data_1_hr.csv")
```

```
In [ ]: import datetime
        import tarfile

        import boto3 # AWS SDK for python. Provides low-level access to AWS services
        from sagemaker import get_execution_role
        import sagemaker

        m_boto3 = boto3.client('sagemaker')

        sess = sagemaker.Session()

        region = sess.boto_session.region_name

        bucket = 'flood-prediction-master-dataset'  #  Bucket name where the file is to be saved

        print('Using bucket ' + bucket)

        # save data to S3. key_prefix is the directory path. path is the local file name which will be saved in S3. It has to be
        # the same file name as saved locally and bucket is the bucket name

        upload = sess.upload_data(path='final_data_1_hr.csv', bucket=bucket, key_prefix='final-dataset-1-hr')
```

Figure 66: Save 1 Hour Final dataset to S3

Find the mean for each column in Sensor and GAN file to assess how close the values are.

```
In [ ]: print("Mean of River GAN: "+str(round(gan_file['river'].mean(),4))+" Mean of River Sensor: "+str(round(sensor_file['river'].mean(

        print("Mean of Rainfall GAN: "+str(round(gan_file['rain'].mean(),4))+" Mean of Rainfall Sensor: "+str(round(sensor_file['rain'].m

        print("Mean of Temperature GAN: "+str(round(gan_file['temperature'].mean(),4))+" Mean of Temperature Sensor: "+str(round(sensor_f

        print("Mean of wind direction GAN: "+str(round(gan_file['wind_direction'].mean(),4))+" Mean of wind direction Sensor: "+str(rounc

        print("Mean of wind speed GAN: "+str(round(gan_file['wind_speed'].mean(),4))+" Mean of wind speed Sensor: "+str(round(sensor_file
```

Figure 67: Mean comparison for 1 Hour Data

5 QuickSight configuration and Visualization

QuickSight expects a JSON file named as 'manifest.json' which states the list of files to be extracted along with Upload Settings for the same. The below figure states the manifest file for generating a QuickSight visualization for sensor and GAN data for the 15 minutes time period.

```
{
  "fileLocations": [
    {
      "URIs": [
        "https://flood-prediction-master-dataset.s3.amazonaws.com/final-dataset-15-min/final_data_15_min.csv"
      ]
    }
  ],
  "globalUploadSettings": {
    "format": "CSV",
    "delimiter": ",",
    "containsHeader": "true"
  }
}
```

Figure 68: manifest file of 15 minutes sensor and GAN data

Upload the file by navigating to the 'final-dataset-15-min' by clicking on Upload.

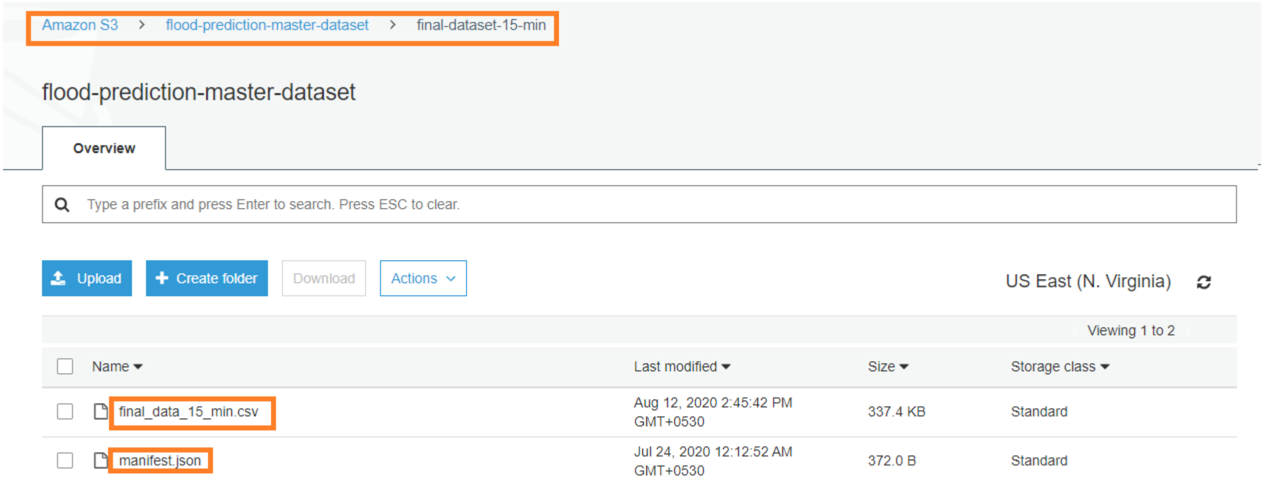


Figure 69: S3 Folder of 15 minutes data

The below figure shows the manifest file for 1 hour time period file.


```
{
  "fileLocations": [
    {
      "URIs": [
        "https://flood-prediction-master-dataset.s3.amazonaws.com/final-dataset-1-hr/final_data_1_hr.csv"
      ]
    }
  ],
  "globalUploadSettings": {
    "format": "CSV",
    "delimiter": ",",
    "containsHeader": "true"
  }
}
```


Figure 70: Manifest file for 1 hour time period for sensor and GAN data comparison

The 'final-dataset-1-hr' should seem like the figure below.

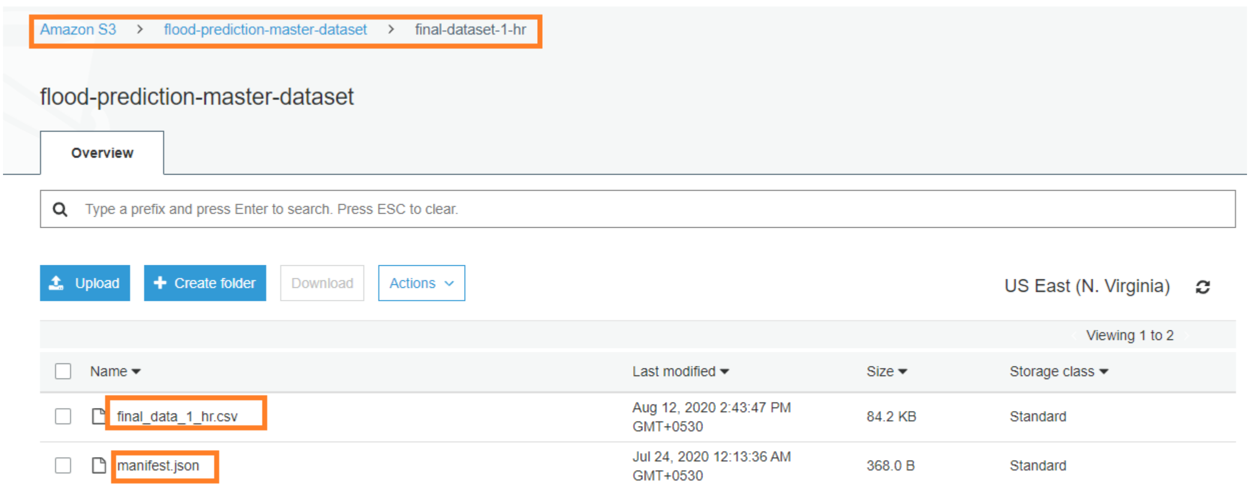


Figure 71: S3 folder of 1 hour data

Click on Services and navigate to the Analytics Section. Click on QuickSight.

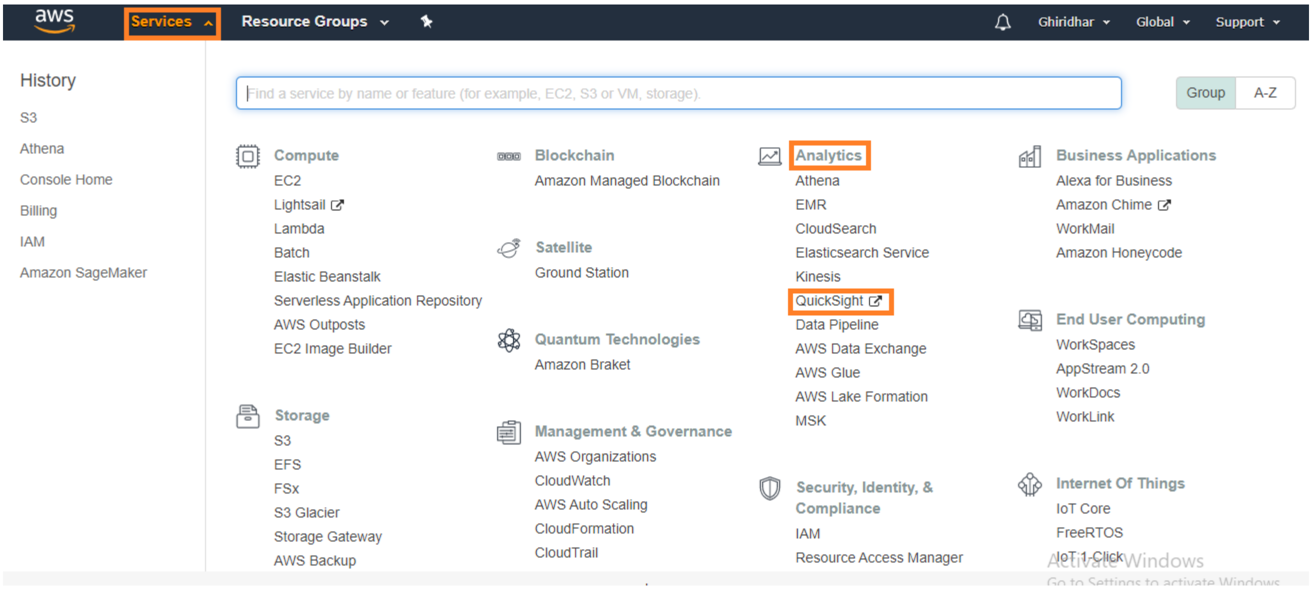


Figure 72: QuickSight Menu

The following screens appear only once. A different account has been used to guide the process. Click on 'Sign Up for QuickSight'. The AWS Account number is unique for every account.

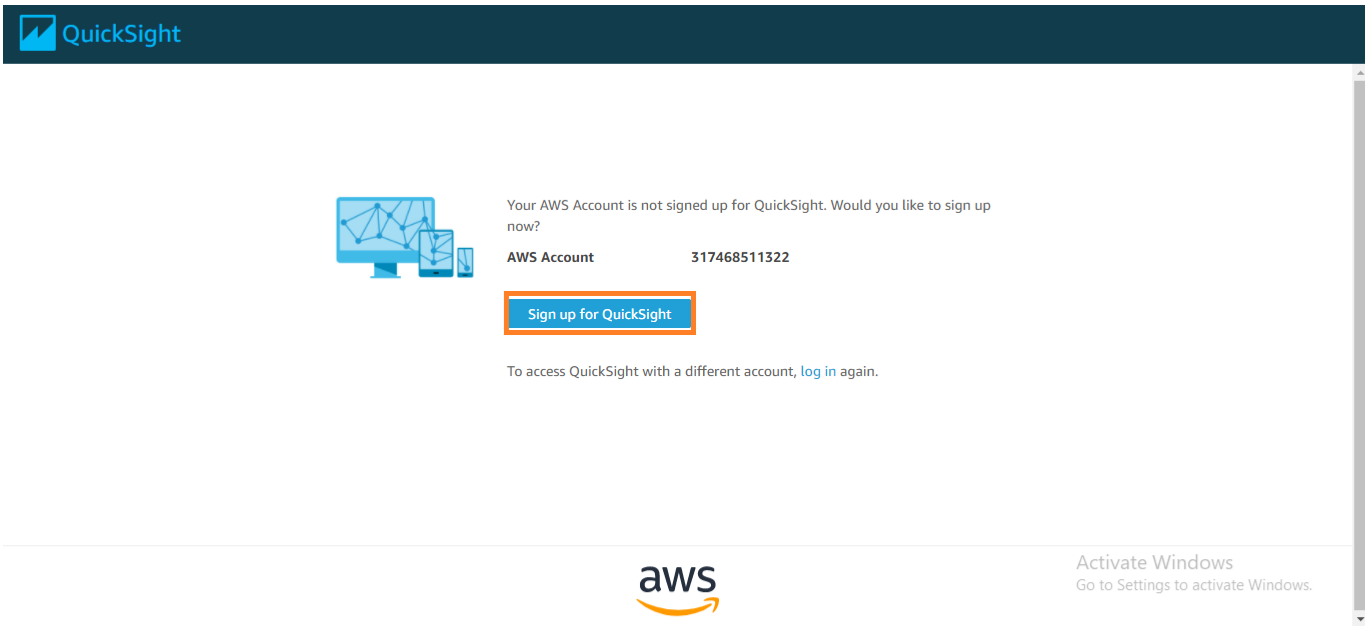


Figure 73: Sign Up to QuickSight

Click on Standard. Choose Enterprise in case you plan to avail those additional services. Click on Continue.

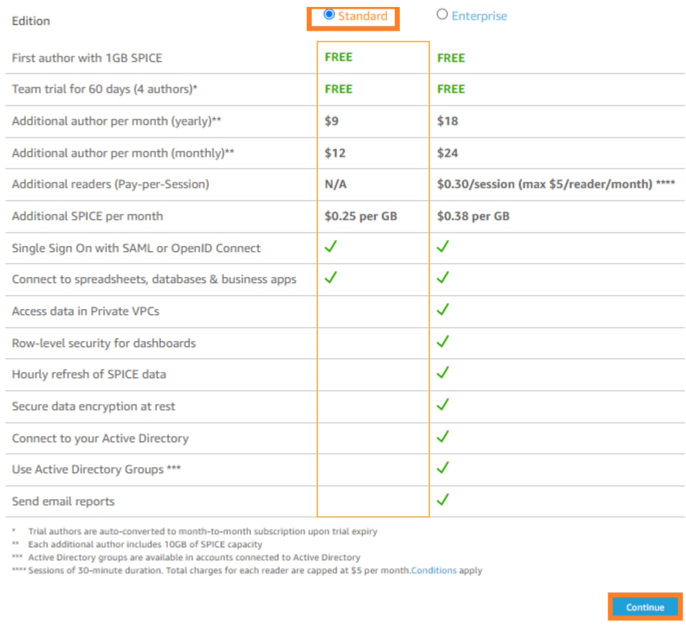


Figure 74: Choose Plan

Enter the account name and email address. Check the Amazon S3 to select the buckets to be provided access to QuickSight.

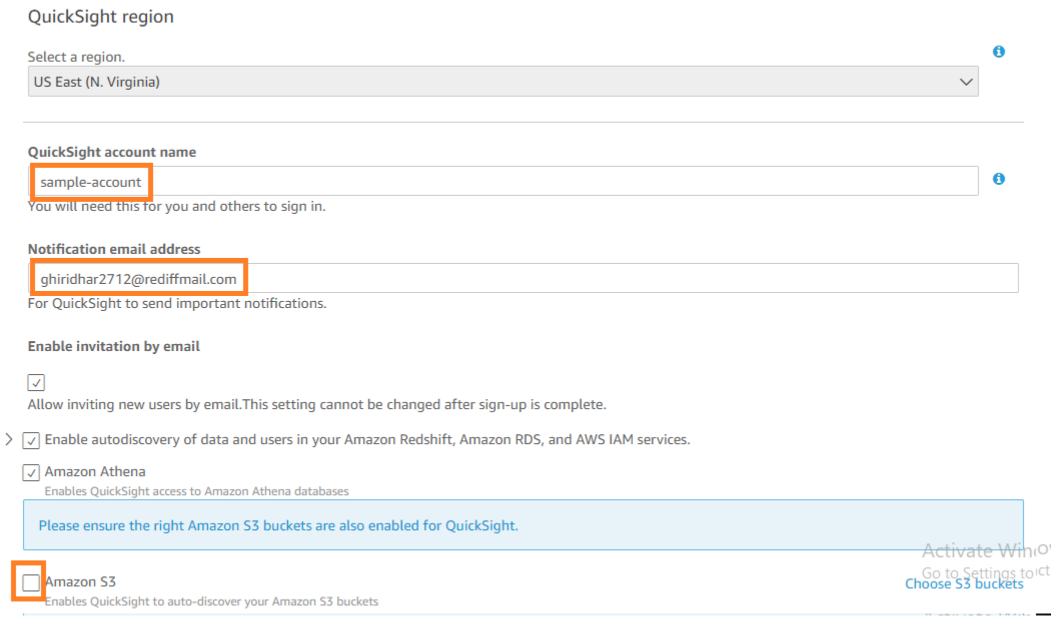


Figure 75: Set Account Name and email address

A Sample Bucket has been selected. Click on 'flood-prediction-master-dataset' and click on Finish. This process was already executed by the main account. Since this cannot be reiterated, another account was used for showing these steps.

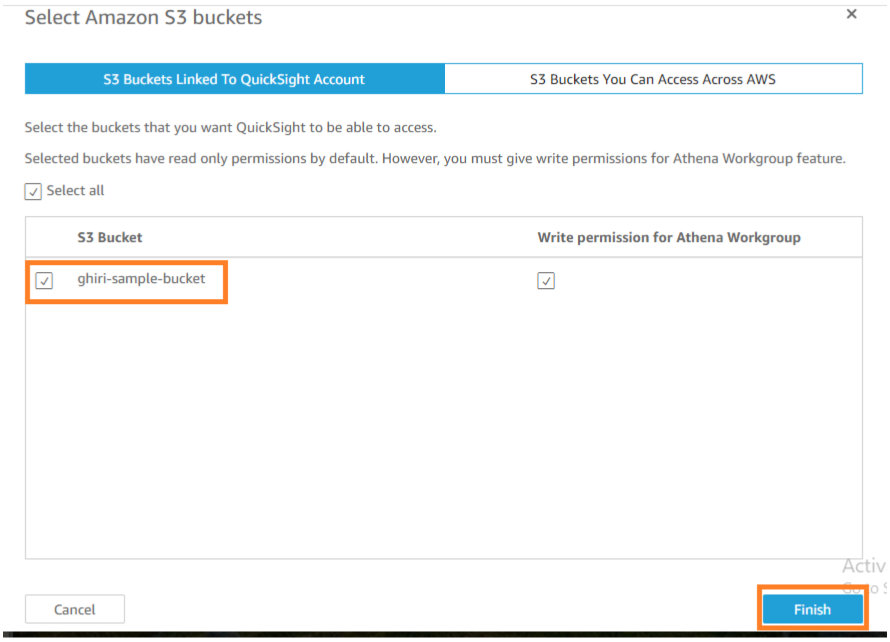


Figure 76: Choose S3 buckets to provide access

Click on Go to Amazon QuickSight.

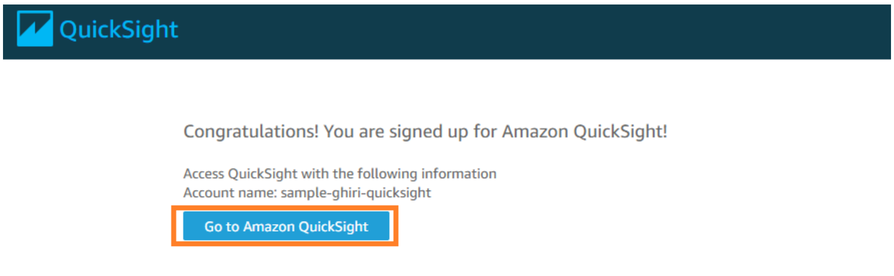


Figure 77: Go to QuickSight Page

The below figure shows the QuickSight Analyses tab.

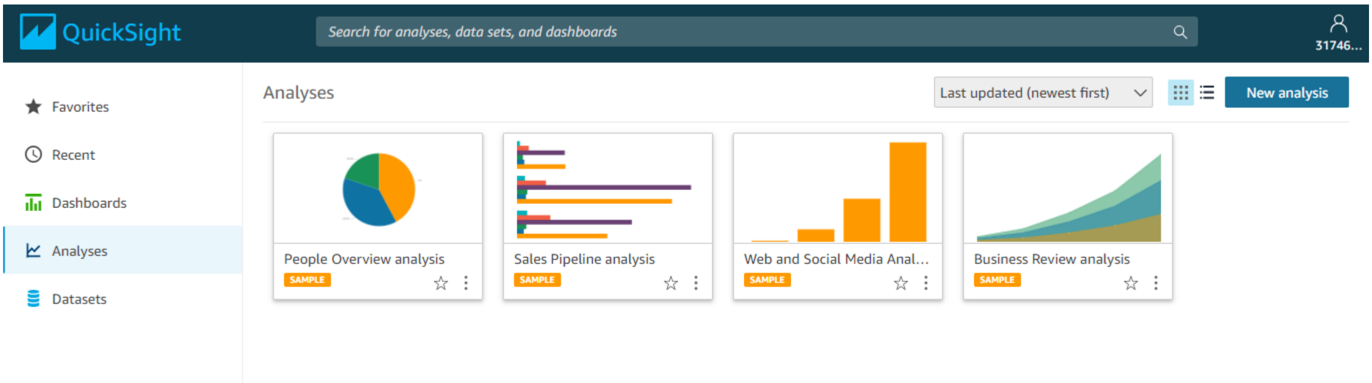


Figure 78: QuickSight main page

Click on Datasets tab to the left.

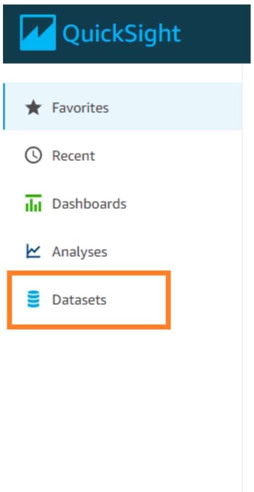


Figure 79: Click on Datasets

Click on 'New dataset'.

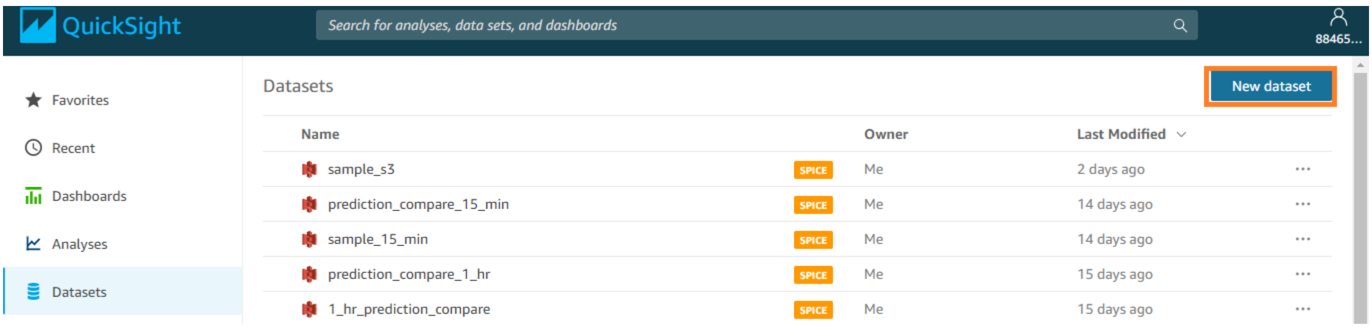


Figure 80: Adding a New Dataset

Click on 'S3' to enter the manifest path.

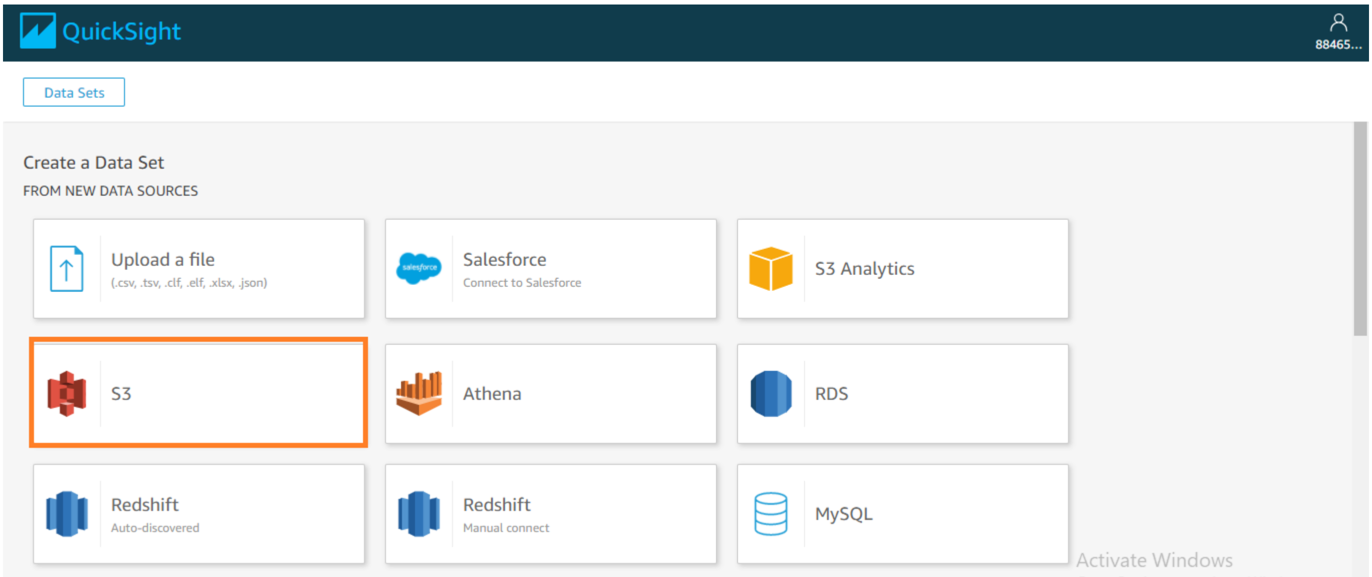


Figure 81: S3 as data source

Enter a name for the visualization process. Provide the path for the manifest.json file.

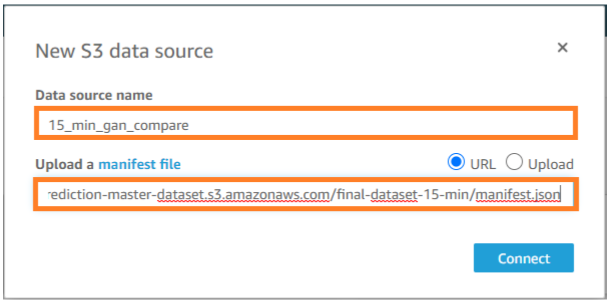


Figure 82: Provide Manifest file path

Click on Visualize.

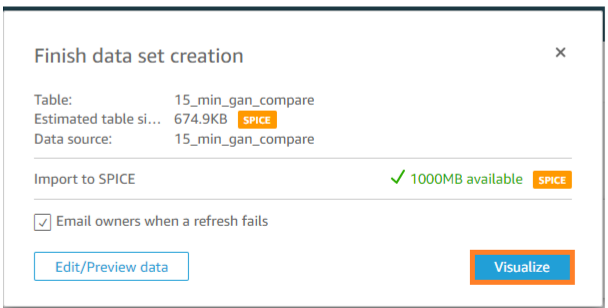


Figure 83: Visualize

Click on the line chart from the list at the left bottom. Drag and Drop 'timerecorded' column to the X-Axis, 'river' to Value and 'source' to Color. The visualization is created below. QuickSight extracts all the file mentioned in the manifest.json and dumps the data into the SPICE storage (storage of QuickSight). Hence a differentiating column is required. Hence, source column was created. The same process is to be followed for obtaining a visualization for sensor and GAN data with a time period of 1 hour. Click on Print at the top right to print/save the visualization.

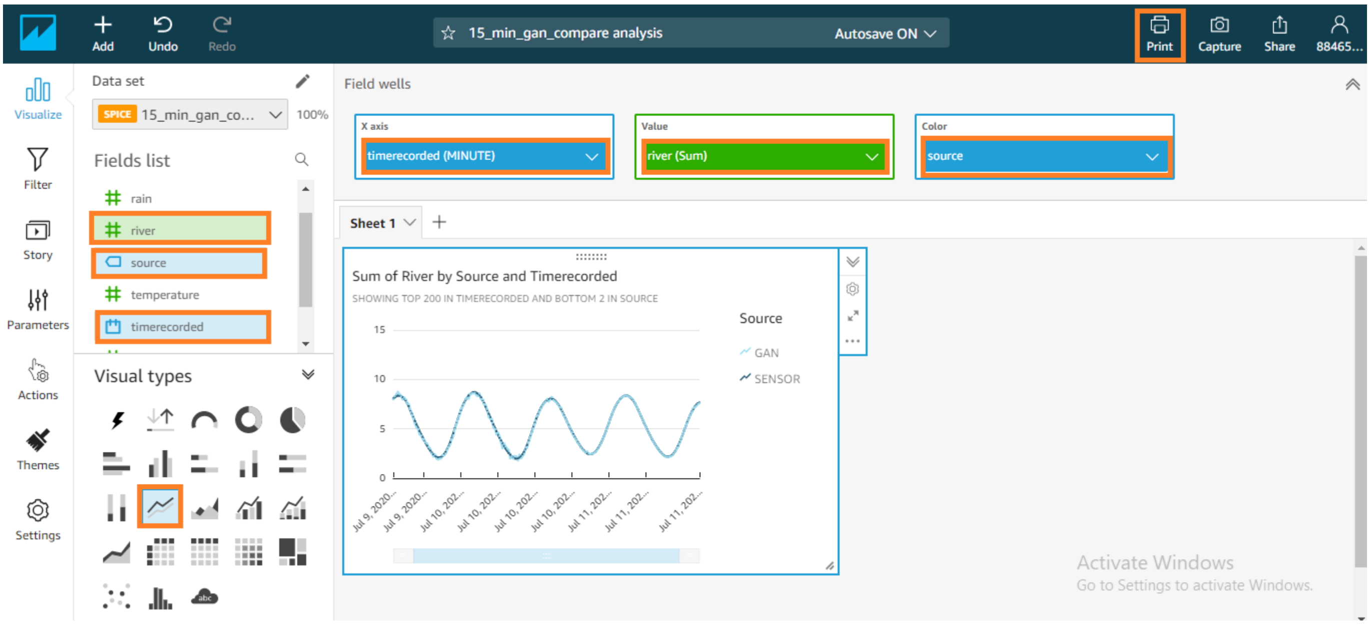


Figure 84: Select Visualization and columns

Click on 'Go to Preview'.

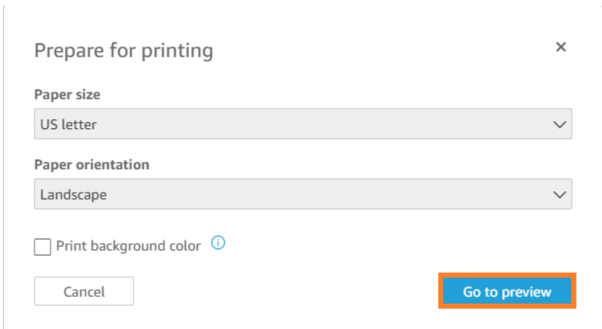


Figure 85: Preview Print

Click on 'Print'.

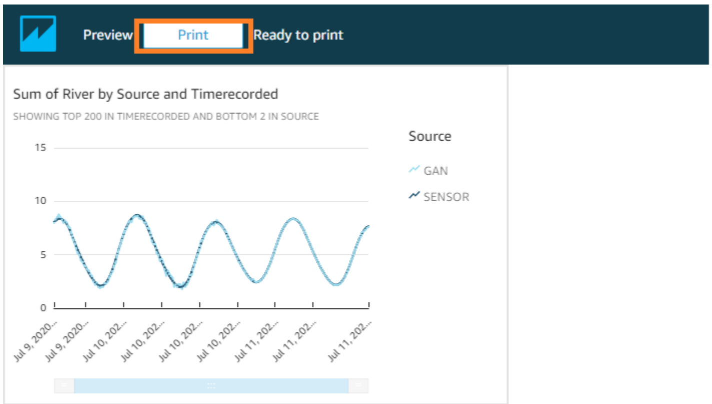


Figure 86: Preview Print

A pop up opens which provides an option to print or save as PDF.

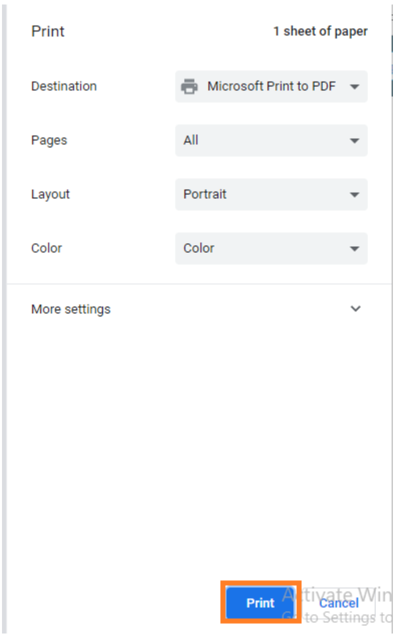


Figure 87: Print Visual

Print or Save the file as PDF.

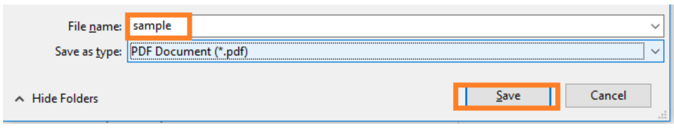


Figure 88: Save the Visual as PDF

6 Model Creation, Prediction & Visualization

Create a Python3 file named 'Random_Forest_Prediction_15_min' inside time-series-algorithms instance. Import libraries and stream the dataset.

```
In [ ]: #import libraries

        import pandas as pd
        import numpy as np
        import boto3
```

```
In [ ]: #import data

        s3 = boto3.client('s3')

        bucket = 'flood-prediction-master-dataset'
        key = 'final-dataset-15-min/final_data_15_min.csv'

        obj = s3.get_object(Bucket= bucket,Key= key)

        dataset_15min = pd.read_csv(obj['Body'])
```

Figure 89: Stream 15 Minutes dataset for Random Forest

Remove the index column and rearrange the column order. GAN and sensor data are stored into two files. GAN data's timestamp is incremented by month and appended to the sensor data.

```
In [ ]: #drop existing index column

        dataset_15min.drop(['Unnamed: 0'],axis=1,inplace=True)

        #setting datetime datatype from string. Default is string when reading from csv

        dataset_15min['timerecorded'] = pd.to_datetime(dataset_15min['timerecorded'])

        #rearranging columns

        dataset_15min = dataset_15min[['timerecorded','river','rain','temperature','wind_direction','wind_speed','source']]

        #splitting GAN and sensor data

        gan_file = dataset_15min.loc[dataset_15min['source']=='GAN']
        sensor_file = dataset_15min.loc[dataset_15min['source']=='SENSOR']
```

```
In [ ]: #makes the GAN datetime go ahead by 1 month. June - July sensor data. June to august is summer. Hence GAN 1 month ahead.

        gan_file['timerecorded']  = gan_file['timerecorded'] + pd.DateOffset(months=1)

        #merging both files and resetting index

        dataset_15min = sensor_file.append(gan_file)
        dataset_15min.reset_index(drop=True, inplace=True)
```

Figure 90: DateTime conversion and GAN timestamp Incrementation

Time Features are added to the dataset. Dataset is splitted in a ratio of 95:5.

```
In [ ]: # adding the datetime column value as a feature. River level being time dependent, datetime column value is saved as
        # continuous columns.

        dataset_15min['dayofweek'] = dataset_15min['timerecorded'].dt.dayofweek
        dataset_15min['hour'] = dataset_15min['timerecorded'].dt.hour
        dataset_15min['minute'] = dataset_15min['timerecorded'].dt.minute
        dataset_15min['month'] = dataset_15min['timerecorded'].dt.month
        dataset_15min['year'] = dataset_15min['timerecorded'].dt.year
        dataset_15min['dayofmonth'] = dataset_15min['timerecorded'].dt.day
        dataset_15min['dayofyear'] = dataset_15min['timerecorded'].dt.dayofyear

In [ ]: dataset_15min.shape

In [ ]: #splitting into train and test dataset

        train_dataset = dataset_15min[:5240]
        test_dataset = dataset_15min[5240:]
```

Figure 91: Data Splitting for Random Forest of 15 Minutes Time Period

Training and testing files are saved as CSV and bucket name is provided to Boto3.

```
In [ ]: # removing dependent columns from test dataset. timerecorded is not required for prediction but for further processes.

        y_test = test_dataset[['timerecorded','river']]
        test_dataset.drop(['timerecorded','river'],axis=1,inplace=True)

        # converting training and testing datasets into csv files

        train_dataset.to_csv("train_dataset.csv")
        test_dataset.to_csv("test_dataset.csv")

In [ ]: import datetime
        import tarfile

        import boto3 # AWS SDK for python. Provides low-level access to AWS services
        from sagemaker import get_execution_role
        import sagemaker

        m_boto3 = boto3.client('sagemaker')

        sess = sagemaker.Session()

        region = sess.boto_session.region_name

        bucket = 'flood-prediction-master-dataset'  #  Bucket to store and retrieve data

        print('Using bucket ' + bucket)
```

Activate Window
Go to Settings to activ

Figure 92: Saving Files as CSV

Training and Testing datasets are uploaded to S3. SageMaker ML either accepts streaming data or S3 path as input.

```
In [ ]: # saving data to S3. SageMaker will take training data from s3

        trainpath = sess.upload_data(
            path='train_dataset.csv', bucket=bucket,
            key_prefix='predictions-15-min')

        testpath = sess.upload_data(
            path='test_dataset.csv', bucket=bucket,
            key_prefix='predictions-15-min')
```

Figure 93: Uploading files to S3

Random Forest is Scripted based on SageMaker Python SDK [3]. Libraries are imported and a model is loaded if already present. Arguments are defined for the script.

[3] https://sagemaker.readthedocs.io/en/stable/frameworks/sklearn/using_sklearn.html

```python
In [ ]: %%writefile rftimeseries15min.py

#doing by scripting

import argparse
import os

import numpy as np
import pandas as pd
from sklearn.ensemble import RandomForestRegressor
import joblib

def model_fn(model_dir):
    clf = joblib.load(os.path.join(model_dir, "rfmodel.joblib"))
    return clf

if __name__ =='__main__':

    print('extracting arguments')
    parser = argparse.ArgumentParser()

    # hyperparameters sent by the client are passed as command-line arguments to the script.

    parser.add_argument('--n-estimators', type=int, default=1500)
    parser.add_argument('--max-leaf-nodes', type=int, default=15)
```

Figure 94: Scripting the Random Forest algorithm - Model Loading

The environmental variables are used to retrieve the Datasets and models within the AWS EC2 ML Instance. The datasets are retrieved and unnecessary columns are removed.

```python
    # Data, model, and output directories
    parser.add_argument('--model_dir', type=str, default=os.environ.get('SM_MODEL_DIR'))
    parser.add_argument('--train', type=str, default=os.environ.get('SM_CHANNEL_TRAIN'))
    parser.add_argument('--test', type=str, default=os.environ.get('SM_CHANNEL_TEST'))
    parser.add_argument('--train-file', type=str, default='train_dataset.csv')
    parser.add_argument('--test-file', type=str, default='test_dataset.csv')

    args, _ = parser.parse_known_args()


    print('reading data')
    train_df = pd.read_csv(os.path.join(args.train, args.train_file))
    test_df = pd.read_csv(os.path.join(args.test, args.test_file))

    print('building training and testing datasets')


    #since only one file is accepted as a script parameter, the predictors and target are segregated here
    y_train = train_df['river']

    # remove unrequired columns
    train_df.drop(['river'],axis=1,inplace=True)
    train_df.drop(['Unnamed: 0'],axis=1,inplace=True)
    train_df.drop(['timerecorded'],axis=1,inplace=True)
    train_df.drop(['source'],axis=1,inplace=True)

    X_train = train_df

    test_df.drop(['Unnamed: 0'],axis=1,inplace=True)
    test_df.drop(['source'],axis=1,inplace=True)
```

Figure 95: Using Environmental Variables to retrieve the required files for prediction

Training and prediction is defined in the script. Model is saved to a ML Instance Folder.

```
X_test = test_df

# train
print('training model')
model = RandomForestRegressor(
    n_estimators=args.n_estimators,
    max_leaf_nodes =args.max_leaf_nodes,
    n_jobs=-1)

model.fit(X_train,y_train)

# persist model

path = os.path.join(args.model_dir, "rfmodel.joblib")
joblib.dump(model, path)
print('model persisted at ' + path)

# predicting value. It will not predict from the below code when deployed to AWS ML EC2.
# but is required so that it can have a code when predict is called. It analyses the no of test parameters and its dtypes
# The print in this script are shown in CloudWatch.

print('validating model')
predictions = model.predict(X_test)
```

Figure 96: Training, Prediction and Saving Model

The Sagemaker estimator is provided the script, instance type and script argument values. Datasets and Training is executed. Estimator is passed the Environment Folder where the model persists.

```
In [ ]: # use of Estimator from the SageMaker Python SDK. stating the script and hyperparameters

        from sagemaker.sklearn.estimator import SKLearn

        sklearn_estimator = SKLearn(
            entry_point='rftimeseries15min.py',
            role = get_execution_role(),
            train_instance_count=1,
            train_instance_type='ml.m4.xlarge',
            framework_version='0.23-1',
            base_job_name='randomforest-15-min',
            hyperparameters = {
                                'n-estimators': 2000,
                                'max-leaf-nodes': 20
                               })

In [ ]: # launch training job, with asynchronous call

        sklearn_estimator.fit({'train':trainpath, 'test': testpath}, wait=False)

In [ ]: # after training the model is created which is used for prediction. Here the model is generated. The path is displayed.

        sklearn_estimator.latest_training_job.wait(logs='None')
        artifact = m_boto3.describe_training_job(
            TrainingJobName=sklearn_estimator.latest_training_job.name)['ModelArtifacts']['S3ModelArtifacts']

        print('Model artifact persisted at ' + artifact)
```

Activate Win
Go to Settings to

Figure 97: Random Forest Model Deployment

The model is deployed to an AWS Endpoint. Test dataset is passed to get the predictions. The timestamp column values from the test dataset and predictions from the algorithms are saved as a dataframe. A source column is created with value 'RF' to identify the prediction source. The file is saved to S3.

```
In [ ]: # An EC2 model is deployed based on the script and model

        predictor = sklearn_estimator.deploy(instance_type='ml.m4.xlarge',initial_instance_count=1)

In [ ]: # removing unrequired columns

        test_dataset.drop(['source'],axis=1,inplace=True)

In [ ]: # "outcome" contains ML predictions. rf_final has the datetime value for each prediction taken from y_test.
        # "rf_final" is then provided prediction values saved as a column. Also source column states the algorithm name.
        # By just counting the number of predictions above flood level, a better algorithm can be decided,
        # but the datetime column will help analyze the delay between two algorithms.

        outcome = pd.DataFrame(predictor.predict(test_dataset))
        outcome.rename(columns={0:"river"},inplace=True)


        rf_final = y_test['timerecorded'].to_frame()
        rf_final.reset_index(drop=True,inplace=True)
        rf_final['river'] = outcome['river'].astype(float)
        rf_final['source'] = 'RF'

        # saving as a csv file locally
        rf_final.to_csv("random_forest_predictions_15_min.csv")

        # saving file to s3
        sess.upload_data(
            path='random_forest_predictions_15_min.csv', bucket=bucket,
            key_prefix='predictions-15-min')

        print("Success!")
```

Activate Wi
Go to Settings t

Figure 98: Prediction and saving it to S3

Create a Python3 file named 'XGBoost_Prediction_15_min' inside time-series-algorithms instance. Import libraries and stream the dataset.

```
In [ ]: #import libraries

        import pandas as pd
        import numpy as np
        import boto3

In [ ]: #import data

        s3 = boto3.client('s3')

        bucket = 'flood-prediction-master-dataset'
        key = 'final-dataset-15-min/final_data_15_min.csv'

        obj = s3.get_object(Bucket= bucket,Key= key)

        dataset_15min = pd.read_csv(obj['Body'])
```

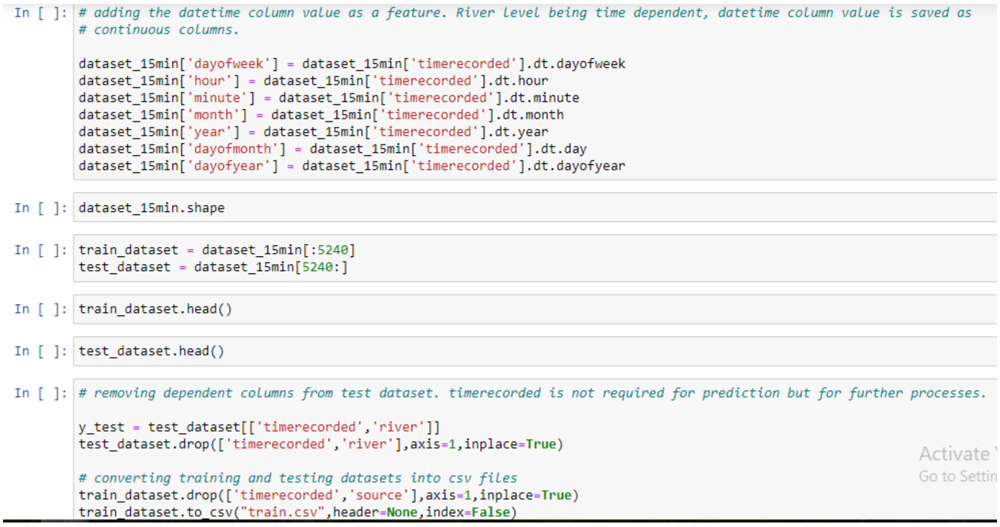
Figure 99: Stream 15 Minutes Data for XGBoost

The below figures shows the process of streaming the data, imcrementing the GAN timestamp value, feature generation and splitting the file in a 95:5 ratio.

```
In [ ]: #drop existing index column

        dataset_15min.drop(['Unnamed: 0'],axis=1,inplace=True)

        #setting datetime datatype from string. Default is string when reading from csv

        dataset_15min['timerecorded'] = pd.to_datetime(dataset_15min['timerecorded'])

        #rearranging columns

        dataset_15min = dataset_15min[['timerecorded','river','rain','temperature','wind_direction','wind_speed','source']]

        #splitting GAN and sensor data

        gan_file = dataset_15min.loc[dataset_15min['source']=='GAN']
        sensor_file = dataset_15min.loc[dataset_15min['source']=='SENSOR']

In [ ]: #makes the GAN datetime go ahead by 1 month. June - July sensor data. June to august is summer. Hence GAN 1 month ahead.

        gan_file['timerecorded']  = gan_file['timerecorded'] + pd.DateOffset(months=1)

        #merging both files and resetting index

        dataset_15min = sensor_file.append(gan_file)
        dataset_15min.reset_index(drop=True, inplace=True)
```

Activate W

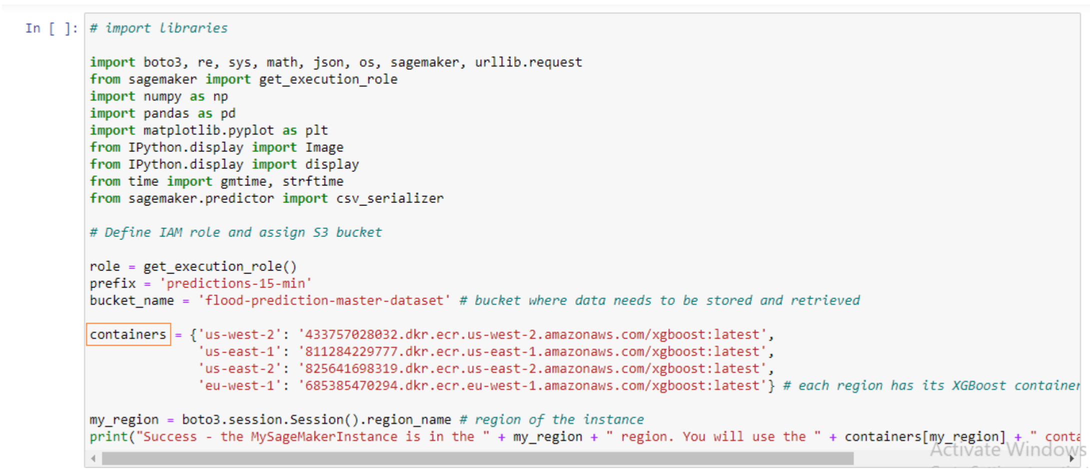
Figure 100: DateTime conversion and GAN Timestamp incrementation



```
In [ ]: # adding the datetime column value as a feature. River level being time dependent, datetime column value is saved as
        # continuous columns.

        dataset_15min['dayofweek'] = dataset_15min['timerecorded'].dt.dayofweek
        dataset_15min['hour'] = dataset_15min['timerecorded'].dt.hour
        dataset_15min['minute'] = dataset_15min['timerecorded'].dt.minute
        dataset_15min['month'] = dataset_15min['timerecorded'].dt.month
        dataset_15min['year'] = dataset_15min['timerecorded'].dt.year
        dataset_15min['dayofmonth'] = dataset_15min['timerecorded'].dt.day
        dataset_15min['dayofyear'] = dataset_15min['timerecorded'].dt.dayofyear

In [ ]: dataset_15min.shape

In [ ]: train_dataset = dataset_15min[:5240]
        test_dataset = dataset_15min[5240:]

In [ ]: train_dataset.head()

In [ ]: test_dataset.head()

In [ ]: # removing dependent columns from test dataset. timerecorded is not required for prediction but for further processes.

        y_test = test_dataset[['timerecorded','river']]
        test_dataset.drop(['timerecorded','river'],axis=1,inplace=True)

        # converting training and testing datasets into csv files
        train_dataset.drop(['timerecorded','source'],axis=1,inplace=True)
        train_dataset.to_csv("train.csv",header=None,index=False)
```

Figure 101: Feature Generation and File Splitting

Import libraries, set the source bucket and select an AWS container comprising of XG-Boost algorithm. AWS provides containers with preloaded XGBoost algorithm. Training and deployment processes are not benefitted but the scripting time is saved.



```
In [ ]: # import libraries

        import boto3, re, sys, math, json, os, sagemaker, urllib.request
        from sagemaker import get_execution_role
        import numpy as np
        import pandas as pd
        import matplotlib.pyplot as plt
        from IPython.display import Image
        from IPython.display import display
        from time import gmtime, strftime
        from sagemaker.predictor import csv_serializer

        # Define IAM role and assign S3 bucket

        role = get_execution_role()
        prefix = 'predictions-15-min'
        bucket_name = 'flood-prediction-master-dataset' # bucket where data needs to be stored and retrieved

        containers = {'us-west-2': '433757028032.dkr.ecr.us-west-2.amazonaws.com/xgboost:latest',
                      'us-east-1': '811284229777.dkr.ecr.us-east-1.amazonaws.com/xgboost:latest',
                      'us-east-2': '825641698319.dkr.ecr.us-east-2.amazonaws.com/xgboost:latest',
                      'eu-west-1': '685385470294.dkr.ecr.eu-west-1.amazonaws.com/xgboost:latest'} # each region has its XGBoost container

        my_region = boto3.session.Session().region_name # region of the instance
        print("Success - the MySageMakerInstance is in the " + my_region + " region. You will use the " + containers[my_region] + " conta
```

Figure 102: Choose AWS XGBoost Containers

Instance type, Hyperparameters and path to the datasets are provided.

```
In [ ]: # setting hyperparameters, bucket and session data

        sess = sagemaker.Session()

        xgb = sagemaker.estimator.Estimator(containers[my_region],
                                            role,
                                            train_instance_count=1,
                                            train_instance_type='ml.m4.xlarge',
                                            output_path='s3://{}/{}/output'.format(bucket_name, prefix),
                                            sagemaker_session=sess)
        xgb.set_hyperparameters(eta=0.06,
                                silent=0,
                                early_stopping_rounds=5,
                                objective='reg:linear',
                                num_round=1000)

In [ ]: # saving data to S3. SageMaker will take training data from s3
        boto3.Session().resource('s3').Bucket(bucket_name).Object(os.path.join(prefix, 'train/train.csv')).upload_file('train.csv')

        trainpath = sagemaker.s3_input(s3_data='s3://{}/{}/train'.format(bucket_name, prefix), content_type='csv')

In [ ]: s3 = boto3.client('s3')
        s3.get_object(Bucket=bucket_name)
```

Figure 103: Uploading Datasets to S3 and defining the estimator

The model is trained and deployed.

```
In [ ]: # training the model

        xgb.fit({'train': trainpath})

In [ ]: # deploying to a endpoint

        xgb_predictor = xgb.deploy(initial_instance_count=1,instance_type='ml.m4.xlarge')

In [ ]: xgb_predictor.content_type = 'text/csv' # set the data type for an inference
        xgb_predictor.serializer = csv_serializer # set the serializer type

In [ ]: # removing unrequired columns

        test_dataset.drop(['source'],axis=1,inplace=True)
```

Figure 104: XGBoost Training and Deployment 15 Minutes Time Period Data

The prediction is obtained which is combined with the test dataset timestamp. The 'source' column is created with value 'XGB'. The test dataset is added a 'source' column with value 'ACTUAL'. Both files are saved to S3.

```
In [ ]: #predictions contains ML predictions
        #see how to add column name to prediction output
        predictions = xgb_predictor.predict(test_dataset.values).decode('utf-8') # prediction
        predictions_array = np.fromstring(predictions[1:], sep=',') # and turn the prediction into an array

        outcome = pd.DataFrame(predictions_array)
        outcome.rename(columns={0:"river"},inplace=True)


        rf_final = y_test['timerecorded'].to_frame()
        rf_final.reset_index(drop=True,inplace=True)
        rf_final['river'] = outcome['river'].astype(float)
        rf_final['source'] = 'XGB'

        rf_final.to_csv("xgboost_predictions_15_min.csv")

        y_test['source'] = 'ACTUAL'

        y_test.to_csv("actual_15_min.csv")

        sess.upload_data(
            path='xgboost_predictions_15_min.csv', bucket=bucket_name,
            key_prefix='predictions-15-min')

        sess.upload_data(
            path='actual_15_min.csv', bucket=bucket_name,
            key_prefix='predictions-15-min')

        print("Success!")
```

Figure 105: Predicting and Uploading the file to S3

Manifest file for visualizing the actual river level and algorithm predictions for 15 minutes time period is provided below.

```
{
  "fileLocations": [
    {
      "URIs": [
        "https://flood-prediction-master-dataset.s3.amazonaws.com/predictions-15-min/actual_15_min.csv",
        "https://flood-prediction-master-dataset.s3.amazonaws.com/predictions-15-min/random_forest_predictions_15_min.csv",
        "https://flood-prediction-master-dataset.s3.amazonaws.com/predictions-15-min/xgboost_predictions_15_min.csv"
      ]
    }
  ],
  "globalUploadSettings": {
    "format": "CSV",
    "delimiter": ",",
    "containsHeader": "true"
  }
}
```


Figure 106: 15 Minutes Prediction Manifest File

Below figure shows the 'predictions-15-min' folder wherein three datasets and manifest file is present.

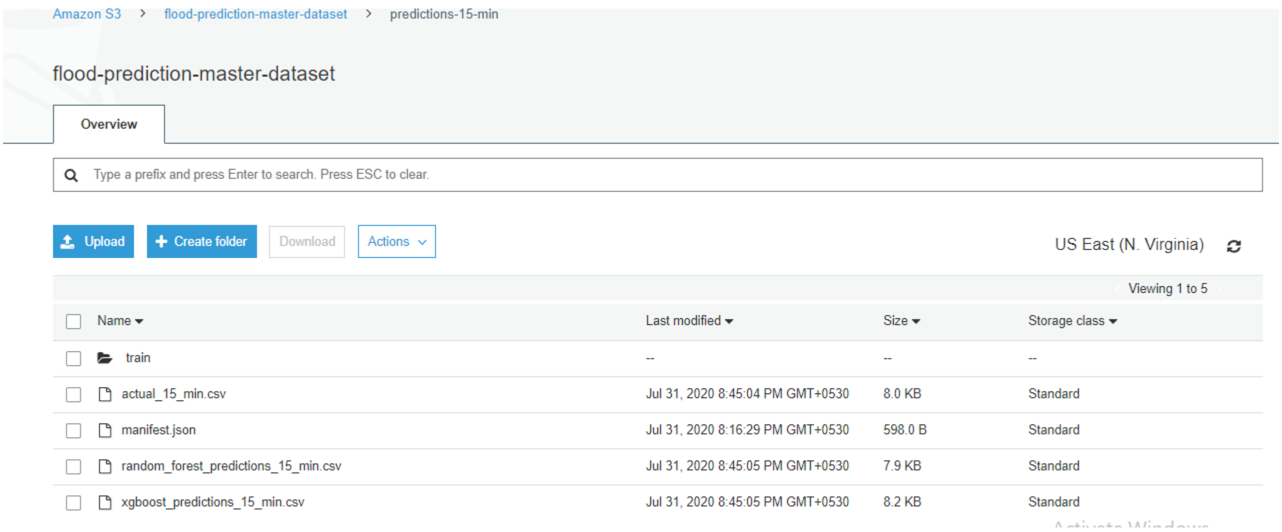


Figure 107: S3 bucket folder of 15 minute predictions

Provide QuickSight the manifest file's link and configure the QuickSight as shown below.

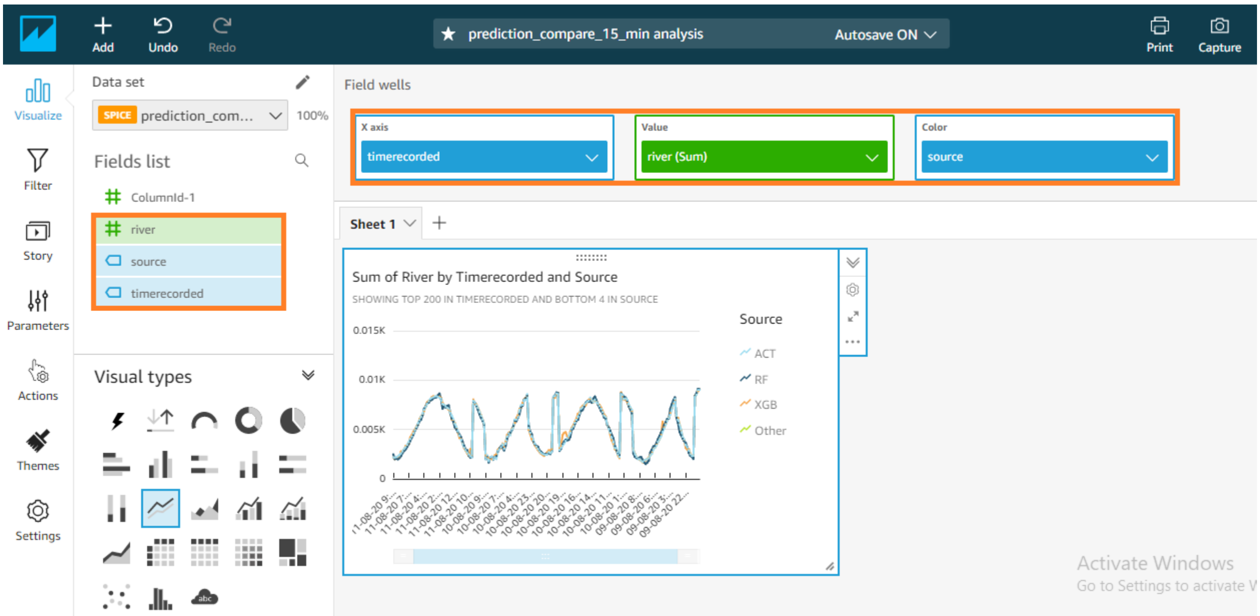


Figure 108: Visualization of 15 Minutes predictions

Create a Python3 file named 'XGBoost_Prediction_1_hr' inside time-series-algorithms instance. The below code predicts the river level based on 1 hour dataset and follows the same flow as in the above XGBoost file.

```
In [ ]: #import libraries

import pandas as pd
import numpy as np
import boto3
```

```
In [ ]: #import data

s3 = boto3.client('s3')

bucket = 'flood-prediction-master-dataset'
key = 'final-dataset-1-hr/final_data_1_hr.csv'

obj = s3.get_object(Bucket= bucket,Key= key)

dataset_1hr = pd.read_csv(obj['Body'])
```

Figure 109: Streaming the 1 hour time period datasets for XGBoost

```
In [ ]: #drop existing index column

dataset_1hr.drop(['Unnamed: 0'],axis=1,inplace=True)

#setting datetime datatype from string. Default is string when reading from csv

dataset_1hr['timerecorded'] = pd.to_datetime(dataset_1hr['timerecorded'])

#rearranging columns

dataset_1hr = dataset_1hr[['timerecorded','river','rain','temperature','wind_direction','wind_speed','source']]

#splitting GAN and sensor data

gan_file = dataset_1hr.loc[dataset_1hr['source']=='GAN']
sensor_file = dataset_1hr.loc[dataset_1hr['source']=='SENSOR']
```

```
In [ ]: #makes the GAN datetime go ahead by 1 month. June - July sensor data. June to august is summer. Hence GAN 1 month ahead.

gan_file['timerecorded']  = gan_file['timerecorded'] + pd.DateOffset(months=1)

#merging both files and resetting index

dataset_1hr = sensor_file.append(gan_file)
dataset_1hr.reset_index(drop=True, inplace=True)
```

Activate Wi

Figure 110: DateTime conversion and GAN Timestamp incrementation

```
In [ ]: # adding the datetime column value as a feature. River level being time dependent, datetime column value is saved as
        # continuous columns.

        dataset_1hr['dayofweek'] = dataset_1hr['timerecorded'].dt.dayofweek
        dataset_1hr['hour'] = dataset_1hr['timerecorded'].dt.hour
        dataset_1hr['minute'] = dataset_1hr['timerecorded'].dt.minute
        dataset_1hr['month'] = dataset_1hr['timerecorded'].dt.month
        dataset_1hr['year'] = dataset_1hr['timerecorded'].dt.year
        dataset_1hr['dayofmonth'] = dataset_1hr['timerecorded'].dt.day
        dataset_1hr['dayofyear'] = dataset_1hr['timerecorded'].dt.dayofyear

In [ ]: dataset_1hr.shape

In [ ]: train_dataset = dataset_1hr[:1315]
        test_dataset = dataset_1hr[1315:]

In [ ]: train_dataset.tail()

In [ ]: test_dataset.head()

In [ ]: # removing dependent columns from test dataset. timerecorded is not required for prediction but for further processes.

        y_test = test_dataset[['timerecorded','river']]
        test_dataset.drop(['timerecorded','river'],axis=1,inplace=True)

        # converting training and testing datasets into csv files
        train_dataset.drop(['timerecorded','source'],axis=1,inplace=True)
        train_dataset.to_csv("train.csv",header=None,index=False)
```

Activate
Go to Setti

Figure 111: Feature Generation and Splitting

```
In [ ]: # import libraries

        import boto3, re, sys, math, json, os, sagemaker, urllib.request
        from sagemaker import get_execution_role
        import numpy as np
        import pandas as pd
        import matplotlib.pyplot as plt
        from IPython.display import Image
        from IPython.display import display
        from time import gmtime, strftime
        from sagemaker.predictor import csv_serializer

        # Define IAM role and assign S3 bucket

        role = get_execution_role()
        prefix = 'predictions-1-hr'
        bucket_name = 'flood-prediction-master-dataset' # bucket where data needs to be stored and retrieved

        containers = {'us-west-2': '433757028032.dkr.ecr.us-west-2.amazonaws.com/xgboost:latest',
                      'us-east-1': '811284229777.dkr.ecr.us-east-1.amazonaws.com/xgboost:latest',
                      'us-east-2': '825641698319.dkr.ecr.us-east-2.amazonaws.com/xgboost:latest',
                      'eu-west-1': '685385470294.dkr.ecr.eu-west-1.amazonaws.com/xgboost:latest'} # each region has its XGBoost container

        my_region = boto3.session.Session().region_name # region of the instance
        print("Success - the MySageMakerInstance is in the " + my_region + " region. You will use the " + containers[my_region] + " conta
```

Activate Windows

Figure 112: Choosing the AWS XGBoost Containers

```
In [ ]: # setting hyperparameters, bucket and session data

        sess = sagemaker.Session()

        xgb = sagemaker.estimator.Estimator(containers[my_region],
                                            role,
                                            train_instance_count=1,
                                            train_instance_type='ml.m4.xlarge',
                                            output_path='s3://{}/{}/output'.format(bucket_name, prefix),
                                            sagemaker_session=sess)
        xgb.set_hyperparameters(eta=0.06,
                                silent=0,
                                early_stopping_rounds=5,
                                objective='reg:linear',
                                num_round=1000)

In [ ]: # saving data to S3. SageMaker will take training data from s3
        boto3.Session().resource('s3').Bucket(bucket_name).Object(os.path.join(prefix, 'train/train.csv')).upload_file('train.csv')

        trainpath = sagemaker.s3_input(s3_data='s3://{}/{}/train'.format(bucket_name, prefix), content_type='csv')

In [ ]: # training the model

        xgb.fit({'train': trainpath})
```

Activate Windows

Figure 113: Defining the estimators, uploading the files and training the model

```
In [ ]: # deploying to a endpoint

        xgb_predictor = xgb.deploy(initial_instance_count=1,instance_type='ml.m4.xlarge')

In [ ]: xgb_predictor.content_type = 'text/csv' # set the data type for an inference
        xgb_predictor.serializer = csv_serializer # set the serializer type

In [ ]: # removing unrequired columns

        test_dataset.drop(['source'],axis=1,inplace=True)
```

Figure 114: Deploying the XGBoost model

```
In [ ]: #predictions contains ML predictions
        #see how to add column name to prediction output
        predictions = xgb_predictor.predict(test_dataset.values).decode('utf-8') # prediction
        predictions_array = np.fromstring(predictions[1:], sep=',') # and turn the prediction into an array

        outcome = pd.DataFrame(predictions_array)
        outcome.rename(columns={0:"river"},inplace=True)


        rf_final = y_test['timerecorded'].to_frame()
        rf_final.reset_index(drop=True,inplace=True)
        rf_final['river'] = outcome['river'].astype(float)
        rf_final['source'] = 'XGB'

        rf_final.to_csv("xgboost_predictions_1_hr.csv")

        y_test['source'] = 'ACTUAL'

        y_test.to_csv("actual_1_hr.csv")

        sess.upload_data(
            path='xgboost_predictions_1_hr.csv', bucket=bucket_name,
            key_prefix='predictions-1-hr')

        sess.upload_data(
            path='actual_1_hr.csv', bucket=bucket_name,
            key_prefix='predictions-1-hr')

        print("Success!")
```

Figure 115: Predicting and uploading the file to S3

Create a Python3 file named 'Random_Forest_Prediction_1_hr' inside time-series-algorithms instance. The below code predicts the river level based on 1 hour dataset and follows the same flow as in the above Random Forest file.

```
In [ ]: #import libraries

        import pandas as pd
        import numpy as np
        import boto3

In [ ]: #import data

        s3 = boto3.client('s3')

        bucket = 'flood-prediction-master-dataset'
        key = 'final-dataset-1-hr/final_data_1_hr.csv'

        obj = s3.get_object(Bucket= bucket,Key= key)

        dataset_1hr = pd.read_csv(obj['Body'])
```

Figure 116: Streaming the 1 hour time period dataset for Random Forest

```python
In [ ]: #drop existing index column

        dataset_1hr.drop(['Unnamed: 0'],axis=1,inplace=True)

        #setting datetime datatype from string. Default is string when reading from csv

        dataset_1hr['timerecorded'] = pd.to_datetime(dataset_1hr['timerecorded'])

        #rearranging columns

        dataset_1hr = dataset_1hr[['timerecorded','river','rain','temperature','wind_direction','wind_speed','source']]

        #splitting GAN and sensor data

        gan_file = dataset_1hr.loc[dataset_1hr['source']=='GAN']
        sensor_file = dataset_1hr.loc[dataset_1hr['source']=='SENSOR']

In [ ]: #makes the GAN datetime go ahead by 1 month. June - July sensor data. June to august is summer. Hence GAN 1 month ahead.

        gan_file['timerecorded']  = gan_file['timerecorded'] + pd.DateOffset(months=1)

        #merging both files and resetting index

        dataset_1hr = sensor_file.append(gan_file)
        dataset_1hr.reset_index(drop=True, inplace=True)
```

Figure 117: DateTime conversion and GAN Timestamp incrementation

```python
In [ ]: # adding the datetime column value as a feature. River level being time dependent, datetime column value is saved as
        # continuous columns.
        dataset_1hr['dayofweek'] = dataset_1hr['timerecorded'].dt.dayofweek
        dataset_1hr['hour'] = dataset_1hr['timerecorded'].dt.hour
        dataset_1hr['minute'] = dataset_1hr['timerecorded'].dt.minute
        dataset_1hr['month'] = dataset_1hr['timerecorded'].dt.month
        dataset_1hr['year'] = dataset_1hr['timerecorded'].dt.year
        dataset_1hr['dayofmonth'] = dataset_1hr['timerecorded'].dt.day
        dataset_1hr['dayofyear'] = dataset_1hr['timerecorded'].dt.dayofyear

In [ ]: dataset_1hr.shape

In [ ]: train_dataset = dataset_1hr[1315:]
        test_dataset = dataset_1hr[:1315]

In [ ]: train_dataset.tail()

In [ ]: test_dataset.head()

In [ ]: # removing dependent columns from test dataset. timerecorded is not required for prediction but for further processes.

        y_test = test_dataset[['timerecorded','river']]
        test_dataset.drop(['timerecorded','river'],axis=1,inplace=True)

        # converting training and testing datasets into csv files

        train_dataset.to_csv("train_dataset.csv")
        test_dataset.to_csv("test_dataset.csv")
```

Figure 118: Feature Generation and Dataset Splitting

```python
In [ ]: import datetime
        import tarfile

        import boto3 # AWS SDK for python. Provides low-level access to AWS services
        from sagemaker import get_execution_role
        import sagemaker

        m_boto3 = boto3.client('sagemaker')

        sess = sagemaker.Session()

        region = sess.boto_session.region_name

        bucket = 'flood-prediction-master-dataset'  #  Bucket to store and retrieve data

        print('Using bucket ' + bucket)

In [ ]: # send data to S3. SageMaker will take training data from s3
        trainpath = sess.upload_data(
            path='train_dataset.csv', bucket=bucket,
            key_prefix='predictions-1-hr')

        testpath = sess.upload_data(
            path='test_dataset.csv', bucket=bucket,
            key_prefix='predictions-1-hr')
```

Figure 119: Uploading file to S3

```
In [ ]: %%writefile rftimeseries1hr.py

#doing by scripting

import argparse
import os

import numpy as np
import pandas as pd
from sklearn.ensemble import RandomForestRegressor
import joblib

def model_fn(model_dir):
    clf = joblib.load(os.path.join(model_dir, "rfmodel.joblib"))
    return clf

if __name__ =='__main__':

    print('extracting arguments')
    parser = argparse.ArgumentParser()

    # hyperparameters sent by the client are passed as command-line arguments to the script.

    parser.add_argument('--n-estimators', type=int, default=1500)
    parser.add_argument('--max-leaf-nodes', type=int, default=15)

    # Data, model, and output directories
    parser.add_argument('--model_dir', type=str, default=os.environ.get('SM_MODEL_DIR'))
    parser.add_argument('--train', type=str, default=os.environ.get('SM_CHANNEL_TRAIN'))
    parser.add_argument('--test', type=str, default=os.environ.get('SM_CHANNEL_TEST'))
```

Figure 120: Scripting the 1 hour time period Random Forest

```
    parser.add_argument('--train-file', type=str, default='train_dataset.csv')
    parser.add_argument('--test-file', type=str, default='test_dataset.csv')

    args, _ = parser.parse_known_args()


    print('reading data')
    train_df = pd.read_csv(os.path.join(args.train, args.train_file))
    test_df = pd.read_csv(os.path.join(args.test, args.test_file))

    print('building training and testing datasets')

    #since only one file is accepted as a script parameter, the predictors and target are segregated here
    y_train = train_df['river']

    # remove unrequired columns
    train_df.drop(['river'],axis=1,inplace=True)
    train_df.drop(['Unnamed: 0'],axis=1,inplace=True)
    train_df.drop(['timerecorded'],axis=1,inplace=True)
    train_df.drop(['source'],axis=1,inplace=True)

    X_train = train_df

    test_df.drop(['Unnamed: 0'],axis=1,inplace=True)
    test_df.drop(['source'],axis=1,inplace=True)

    X_test = test_df
```

Figure 121: Creating Script arguments and splitting dataset

```
    # train
    print('training model')
    model = RandomForestRegressor(
        n_estimators=args.n_estimators,
        max_leaf_nodes =args.max_leaf_nodes,
        n_jobs=-1)

    model.fit(X_train,y_train)

    # persist model

    path = os.path.join(args.model_dir, "rfmodel.joblib")
    joblib.dump(model, path)
    print('model persisted at ' + path)

    # predicting value. It will not predict from the below code when deployed to AWS ML EC2.
    # but is required so that it can have a code when predict is called. It analyses the no of test parameters and its dtypes
    # The print in this script are shown in CloudWatch.

    print('validating model')
    predictions = model.predict(X_test)
```

Figure 122: Random Forest Training, Model Creation and Storage

```
In [ ]: # use of Estimator from the SageMaker Python SDK. stating the script and hyperparameters

        from sagemaker.sklearn.estimator import SKLearn

        sklearn_estimator = SKLearn(
            entry_point='rftimeseries1hr.py',
            role = get_execution_role(),
            train_instance_count=1,
            train_instance_type='ml.m4.xlarge',
            framework_version='0.23-1',
            base_job_name='randomforest-1-hr',
            hyperparameters = {
                                'n-estimators': 2000,
                                'max-leaf-nodes': 20
                               })
```

```
In [ ]: # launch training job, with asynchronous call

        sklearn_estimator.fit({'train':trainpath, 'test': testpath}, wait=False)
```

```
In [ ]: # after training the model is created which is used for prediction. Here the model is generated. The path is displayed.

        sklearn_estimator.latest_training_job.wait(logs='None')
        artifact = m_boto3.describe_training_job(
            TrainingJobName=sklearn_estimator.latest_training_job.name)['ModelArtifacts']['S3ModelArtifacts']

        print('Model artifact persisted at ' + artifact)
```

Activate W
Go to Settings

Figure 123: Random Forest for 1 Hour Time Period Training and Deployment

```
In [ ]: # An EC2 model is deployed based on the script and model

        predictor = sklearn_estimator.deploy(instance_type='ml.m4.xlarge',initial_instance_count=1)
```

```
In [ ]: # removing unrequired columns

        test_dataset.drop(['source'],axis=1,inplace=True)
```

```
In [ ]: # "outcome" contains ML predictions. rf_final has the datetime value for each prediction taken from y_test.
        # "rf_final" is then provided prediction values saved as a column. Also source column states the algorithm name.
        # By just counting the number of predictions above flood level, a better algorithm can be decided,
        # but the datetime column will help analyze the delay between two algorithms.

        outcome = pd.DataFrame(predictor.predict(test_dataset))
        outcome.rename(columns={0:"river"},inplace=True)


        rf_final = y_test['timerecorded'].to_frame()
        rf_final.reset_index(drop=True,inplace=True)
        rf_final['river'] = outcome['river'].astype(float)
        rf_final['source'] = 'RF'

        # saving as a csv file locally
        rf_final.to_csv("random_forest_predictions_1_hr.csv")

        # saving file to s3
        sess.upload_data(
            path='random_forest_predictions_1_hr.csv', bucket=bucket,
            key_prefix='predictions-1-hr')

        print("Success!")
```

Figure 124: Random Forest Prediction and Upload to S3

Manifest file for visualizing the actual river level and algorithm predictions for 1 hour time period is provided below. Follow the same process as above to generate a QuickSight visualization.

```
{
    "fileLocations": [
        {
            "URIs": [
                "https://flood-prediction-master-dataset.s3.amazonaws.com/predictions-1-hr/actual_1_hr.csv",
                "https://flood-prediction-master-dataset.s3.amazonaws.com/predictions-1-
hr/random_forest_predictions_1_hr.csv",
                "https://flood-prediction-master-dataset.s3.amazonaws.com/predictions-1-hr/xgboost_predictions_1_hr.csv"
            ]
        }
    ],
    "globalUploadSettings": {
        "format": "CSV",
        "delimiter": ",",
        "containsHeader": "true"
    }
}
```

Figure 125: Manifest File for 1 hour prediction data visualization

7 Flood Trigger Evaluation

The Environmental agency also provides publicly the flood warning trigger level for each river. Based on this value, the Evaluation of the prediction is assessed.

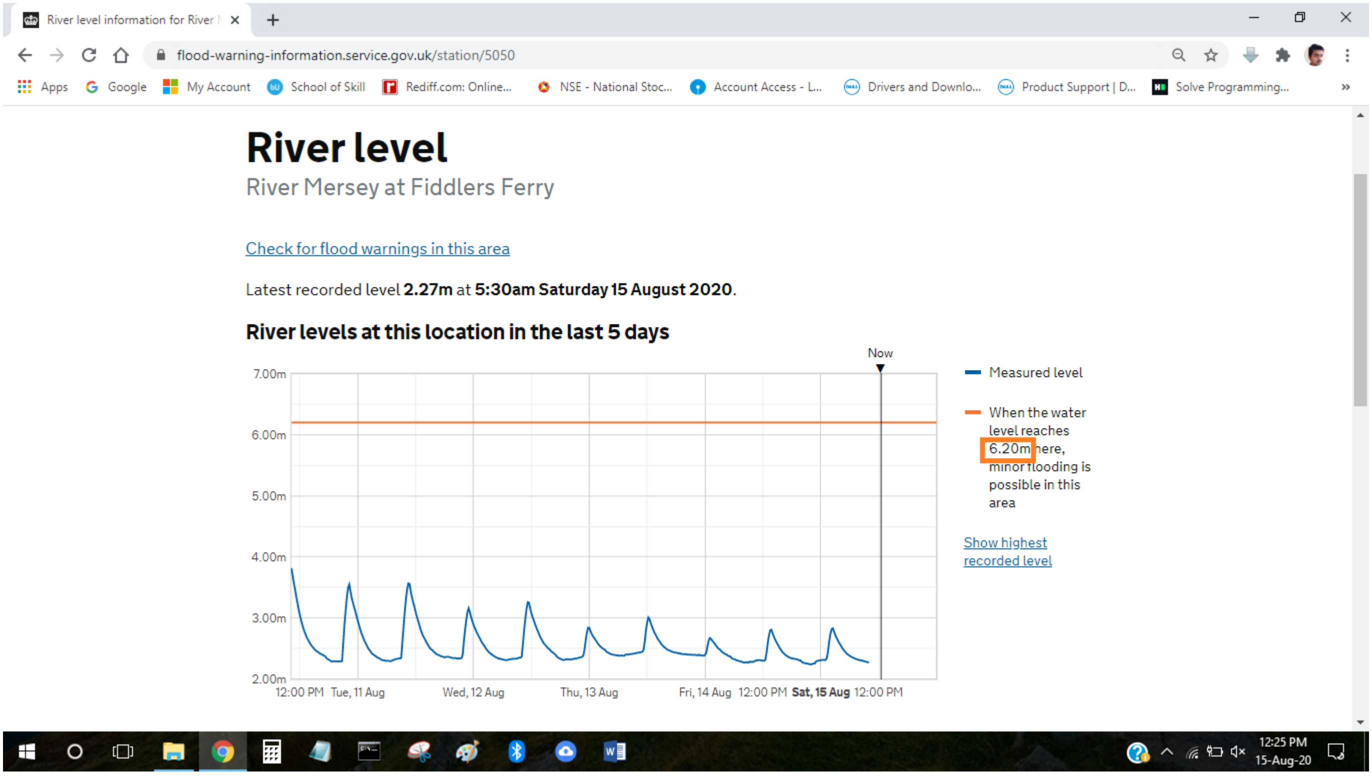


Figure 126: Flood Trigger River Level Value

Create a Python3 file named 'Flood_Trigger_Comparison_1_Hour' inside the time-series-algorithms instance. Import the libraries and stream the CSV files from the S3 'predictions-1-hr' folder.

```
In [ ]: import pandas as pd
        import numpy as np
        from sklearn.metrics import r2_score,mean_absolute_error
        import boto3

In [ ]: #importing predicted data

        s3 = boto3.client('s3')

        bucket = 'flood-prediction-master-dataset'

        key = 'predictions-1-hr/actual_1_hr.csv'
        obj = s3.get_object(Bucket= bucket,Key= key)
        actual = pd.read_csv(obj['Body'])

        key = 'predictions-1-hr/xgboost_predictions_1_hr.csv'
        obj = s3.get_object(Bucket= bucket,Key= key)
        xgb_predict = pd.read_csv(obj['Body'])

        key = 'predictions-1-hr/random_forest_predictions_1_hr.csv'
        obj = s3.get_object(Bucket= bucket,Key= key)
        rf_predict = pd.read_csv(obj['Body'])
```

Figure 127: Importing 1 Hour Time Period Files

Difference between actual and prediction values (can also be termed as Prediction Error) is saved for both algorithms seperately. The sum and max of prediction error for both algorithms is calculated. Also the first flood trigger time is calculated. If the value is within 6 hours, then flash flood prediction is successful.

```
In [ ]: #dropping unnexessary columns

        actual.drop(['Unnamed: 0','source'],axis=1,inplace=True)
        xgb_predict.drop(['Unnamed: 0','source'],axis=1,inplace=True)
        rf_predict.drop(['Unnamed: 0','source'],axis=1,inplace=True)

In [ ]: actual

In [ ]: xgb_predict

In [ ]: rf_predict

In [ ]:

In [ ]: # saving difference between actual and predicted river value

        xgb_difference = pd.DataFrame(actual['river'] - xgb_predict['river'])

        rf_difference = pd.DataFrame(actual['river'] - rf_predict['river'])

        print("Sum of Prediction Error in XGBoost: "+str(xgb_difference['river'].sum())+
              " Highest Error in Prediction in XGBoost: "+str(xgb_difference['river'].max()))

        print("Sum of Prediction Error in Random Forest: "+str(rf_difference['river'].sum())+
              " Highest Error in Prediction in Random Forest: "+str(rf_difference['river'].max()))

        print("Flood Triggered by XGBoost within "+str(xgb_predict[xgb_predict["river"]>=6.2000].index[0])+" Hours."+
              " Flood Triggered by Random Forest within "+str(rf_predict[rf_predict["river"]>=6.2000].index[0])+" Hours.")
```

Activate Wir
Go to Settings t

Figure 128: Assessing Prediction Error

All three datasets are joined without any join condition since all datasets have the same number of records and timestamp. Columns, either actual or prediction greater than 6.2 are retrieved and stored to a different variable. Status column is added to define the validity of the trigger by the predictions.

```
In [ ]: # CREATING TABLE OF TRIGGERS. filtering columns/instances with river level above flood level

        actual_flood = actual[['timerecorded','river']]

In [ ]: # joining xgboost and random forest columns with same index to compare the triggers

        xgb_compare = actual_flood.join(xgb_predict,lsuffix='_actual',rsuffix='_xgboost')

        all_combined = xgb_compare.join(rf_predict)

        all_combined.drop(['timerecorded_xgboost','timerecorded'],axis=1,inplace=True)
        all_combined = all_combined.rename(columns={"river":"river_rf"})

In [ ]: # creating table that has either actual, xgboost prediction or random forest prediction greater than flood threshold value

        possible_flood = all_combined[(all_combined['river_actual']>=6.200) | (all_combined['river_xgboost']>=6.200) |
                                      (all_combined['river_rf']>=6.200)]

In [ ]: possible_flood['xgb_status'] = 'NONE'
        possible_flood['rf_status'] = 'NONE'
```

Figure 129: Extracting the Flood Triggered Records

Status is set to HIT, MISS or FALSE. HIT means the prediction triggered rightly, MISS means prediction missed the flood trigger and FALSE means the Trigger is erroneous.

```
In [ ]: # HIT means triggered rightly MISS means not triggered FALSE wrongly triggered

        possible_flood['xgb_status'] = np.where( (possible_flood['river_actual']>=6.200) & (possible_flood['river_xgboost']>=6.200),
                                               'HIT', possible_flood['xgb_status'])
        possible_flood['xgb_status'] = np.where( (possible_flood['river_actual']>=6.200) & (possible_flood['river_xgboost']<6.200),
                                               'MISS', possible_flood['xgb_status'])
        possible_flood['xgb_status'] = np.where( (possible_flood['river_actual']<6.200) & (possible_flood['river_xgboost']>=6.200),
                                               'FALSE', possible_flood['xgb_status'])


        possible_flood['rf_status'] = np.where( (possible_flood['river_actual']>=6.200) & (possible_flood['river_rf']>=6.200),'HIT',
                                              possible_flood['rf_status'])
        possible_flood['rf_status'] = np.where( (possible_flood['river_actual']>=6.200) & (possible_flood['river_rf']<6.200),'MISS',
                                              possible_flood['rf_status'])
        possible_flood['rf_status'] = np.where( (possible_flood['river_actual']<6.200) & (possible_flood['river_rf']>=6.200),'FALSE',
                                              possible_flood['rf_status'])

In [ ]: possible_flood.reset_index(drop=True,inplace=True)
        possible_flood
```

Figure 130: HIT, MISS and Flase Triggers

This status column for both algorithms is assessed using PASS evaluation to determine the performance of the algorithms. Precision, Accuracy, Specificity and Sensitivity (PASS). R-Square value is also calculate to assess its value with respect to the PASS performance.

```
In [ ]: # variables are used to avoid making print statement long and complex

        xgb_hits = possible_flood['xgb_status'].loc[possible_flood['xgb_status'] == 'HIT'].count()
        xgb_misses = possible_flood['xgb_status'].loc[possible_flood['xgb_status'] == 'MISS'].count()
        xgb_false = possible_flood['xgb_status'].loc[possible_flood['xgb_status'] == 'FALSE'].count()

        rf_hits = possible_flood['rf_status'].loc[possible_flood['rf_status'] == 'HIT'].count()
        rf_misses = possible_flood['rf_status'].loc[possible_flood['rf_status'] == 'MISS'].count()
        rf_false = possible_flood['rf_status'].loc[possible_flood['rf_status'] == 'FALSE'].count()

        actual_triggers = xgb_hits + xgb_misses


        print(" Total Actual Flood Triggers are "+ str(actual_triggers)+ ".\n Out of which XGBoost Triggered "+ str(xgb_hits)+
              ",missed " + str(xgb_misses)+ " and false positives are "+ str(xgb_false) + ". "+
              " XGBoost Efficiency in triggering is " + str(round(xgb_hits/actual_triggers*100,2))+"%.\n"+
              " Random Forest Triggered "+ str(rf_hits) + ",missed "+ str(rf_misses) + " and false positives are "+ str(rf_false)+ ". "+
              " Random Forest Efficiency in triggereing is "+ str(round(rf_hits/actual_triggers*100,2)) +"%.\n"
              " Total Time saved by XGBoost by better prediction and triggering is: "+
              str(xgb_hits-rf_hits) +" Hours."
        )

In [ ]: print("r2 score is: "+str(round(r2_score(actual["river"].values,xgb_predict["river"].values),4)*100))

In [ ]: print("r2 score is: "+str(round(r2_score(actual["river"].values,rf_predict["river"].values),4)*100))
```

Activate Windows
Go to Settings to activate

Figure 131: PASS Evaluation

Create a Python3 file named 'Flood_Trigger_Comparison_15_Min' inside the time-series-algorithms instance. Import the libraries and stream the CSV files from the S3 'predictions-15-mins' folder. The below figures examine the predictions for 1 hour time period as performed above.

```
In [ ]: import pandas as pd
        import numpy as np
        from sklearn.metrics import r2_score,mean_absolute_error
        import boto3
```

```
In [ ]: #importing predicted data

        s3 = boto3.client('s3')

        bucket = 'flood-prediction-master-dataset'

        key = 'predictions-15-min/actual_15_min.csv'
        obj = s3.get_object(Bucket= bucket,Key= key)
        actual = pd.read_csv(obj['Body'])

        key = 'predictions-15-min/xgboost_predictions_15_min.csv'
        obj = s3.get_object(Bucket= bucket,Key= key)
        xgb_predict = pd.read_csv(obj['Body'])

        key = 'predictions-15-min/random_forest_predictions_15_min.csv'
        obj = s3.get_object(Bucket= bucket,Key= key)
        rf_predict = pd.read_csv(obj['Body'])
```

Figure 132: Streaming 15 Minutes Time Period Files

```
In [ ]: #dropping unnexessary columns

        actual.drop(['Unnamed: 0','source'],axis=1,inplace=True)
        xgb_predict.drop(['Unnamed: 0','source'],axis=1,inplace=True)
        rf_predict.drop(['Unnamed: 0','source'],axis=1,inplace=True)

In [ ]: actual

In [ ]: xgb_predict

In [ ]: rf_predict
```

Figure 133: Prediction Error in 15 Minutes Predictions

```
In [ ]: # saving difference between actual and predicted river value

        xgb_difference = pd.DataFrame(actual['river'] - xgb_predict['river'])

        rf_difference = pd.DataFrame(actual['river'] - rf_predict['river'])

        print("Sum of Prediction Error in XGBoost: "+str(xgb_difference['river'].sum())+
              " Highest Error in Prediction in XGBoost: "+str(xgb_difference['river'].max()))

        print("Sum of Prediction Error in Random Forest: "+str(rf_difference['river'].sum())+
              " Highest Error in Prediction in Random Forest: "+str(rf_difference['river'].max()))

        print("Flood Triggered by XGBoost within "+str((xgb_predict[xgb_predict["river"]>=6.2000].index[0])//4)+" Hours."+
              " Flood Triggered by Random Forest within "+str((rf_predict[rf_predict["river"]>=6.2000].index[0])//4)+" Hours.")

In [ ]: # CREATING TABLE OF TRIGGERS. filtering columns/instances with river level above flood level

        actual_flood = actual[['timerecorded','river']]

In [ ]: # joining xgboost and random forest columns with same index to compare the triggers

        xgb_compare = actual_flood.join(xgb_predict,lsuffix='_actual',rsuffix='_xgboost')

        all_combined = xgb_compare.join(rf_predict)

        all_combined.drop(['timerecorded_xgboost','timerecorded'],axis=1,inplace=True)
        all_combined = all_combined.rename(columns={"river":"river_rf"})
```

Activate W
Go to Setting

Figure 134: Extracting Flood Triggered Records

```
In [ ]: # creating table that has either actual, xgboost prediction or random forest prediction greater than flood threshold value

        possible_flood = all_combined[(all_combined['river_actual']>=6.200) | (all_combined['river_xgboost']>=6.200) |
                                      (all_combined['river_rf']>=6.200)]

In [ ]: possible_flood['xgb_status'] = 'NONE'
        possible_flood['rf_status'] = 'NONE'

In [ ]: # HIT means triggered rightly MISS means not triggered FALSE wrongly triggered

        possible_flood['xgb_status'] = np.where( (possible_flood['river_actual']>=6.200) & (possible_flood['river_xgboost']>=6.200),
                                                'HIT', possible_flood['xgb_status'])
        possible_flood['xgb_status'] = np.where( (possible_flood['river_actual']>=6.200) & (possible_flood['river_xgboost']<6.200),
                                                'MISS', possible_flood['xgb_status'])
        possible_flood['xgb_status'] = np.where( (possible_flood['river_actual']<6.200) & (possible_flood['river_xgboost']>=6.200),
                                                'FALSE', possible_flood['xgb_status'])


        possible_flood['rf_status'] = np.where( (possible_flood['river_actual']>=6.200) & (possible_flood['river_rf']>=6.200),'HIT',
                                               possible_flood['rf_status'])
        possible_flood['rf_status'] = np.where( (possible_flood['river_actual']>=6.200) & (possible_flood['river_rf']<6.200),'MISS',
                                               possible_flood['rf_status'])
        possible_flood['rf_status'] = np.where( (possible_flood['river_actual']<6.200) & (possible_flood['river_rf']>=6.200),'FALSE',
                                               possible_flood['rf_status'])

In [ ]: possible_flood.reset_index(drop=True,inplace=True)
        possible_flood
```

Activate Windows
Go to Settings to activat

Figure 135: HIT, MISS and FALSE triggers

```
In [ ]: # variables are used to avoid making print statement long and complex

        xgb_hits = possible_flood['xgb_status'].loc[possible_flood['xgb_status'] == 'HIT'].count()
        xgb_misses = possible_flood['xgb_status'].loc[possible_flood['xgb_status'] == 'MISS'].count()
        xgb_false = possible_flood['xgb_status'].loc[possible_flood['xgb_status'] == 'FALSE'].count()

        rf_hits = possible_flood['rf_status'].loc[possible_flood['rf_status'] == 'HIT'].count()
        rf_misses = possible_flood['rf_status'].loc[possible_flood['rf_status'] == 'MISS'].count()
        rf_false = possible_flood['rf_status'].loc[possible_flood['rf_status'] == 'FALSE'].count()

        actual_triggers = xgb_hits + xgb_misses


        print(" Total Actual Flood Triggers are "+ str(actual_triggers)+ ".\n Out of which XGBoost Triggered "+ str(xgb_hits)+
              ",missed " + str(xgb_misses)+ " and false positives are "+ str(xgb_false) + ". "+
              " XGBoost Efficiency in triggering is " + str(round(xgb_hits/actual_triggers*100,2))+"%.\n"+
              " Random Forest Triggered "+ str(rf_hits) + ",missed "+ str(rf_misses) + " and false positives are "+ str(rf_false)+ ". "+
              " Random Forest Efficiency in triggereing is "+ str(round(rf_hits/actual_triggers*100,2)) +"%.\n"
              " Total Time saved by XGBoost by better prediction and triggering is: "+
              str(round((xgb_hits-rf_hits)/4,2)) +" Hours."
        )

In [ ]: print("r2 score is: "+str(round(r2_score(actual["river"].values,xgb_predict["river"].values),4)*100))

In [ ]: print("r2 score is: "+str(round(r2_score(actual["river"].values,rf_predict["river"].values),4)*100))
```

Activate Windows
Go to Settings to activat

Figure 136: PASS Evaluation on 15 Minutes Predictions

8 Results

The visualization of GAN and sensor data is presented below. It shows that the distribution of the values is similar to that of the sensor dataset. The GAN values are not similar but near enough to mimic the sensor values.

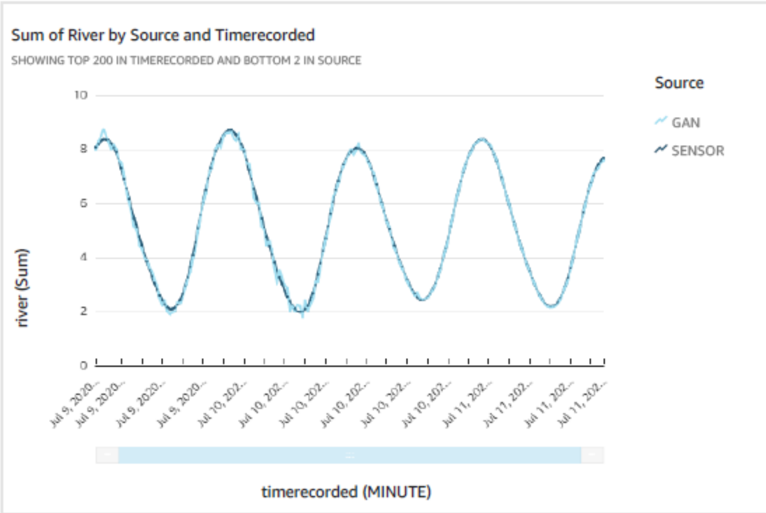


Figure 137: GAN and Sensor Values Comparison for 15 minutes time period

The below graph displays the graph between sensor and GAN data with a time period of 1 hour. These two graphs show good ability of the GAN to imitate the data. But there is one improvement that is yet to be addressed in GAN.

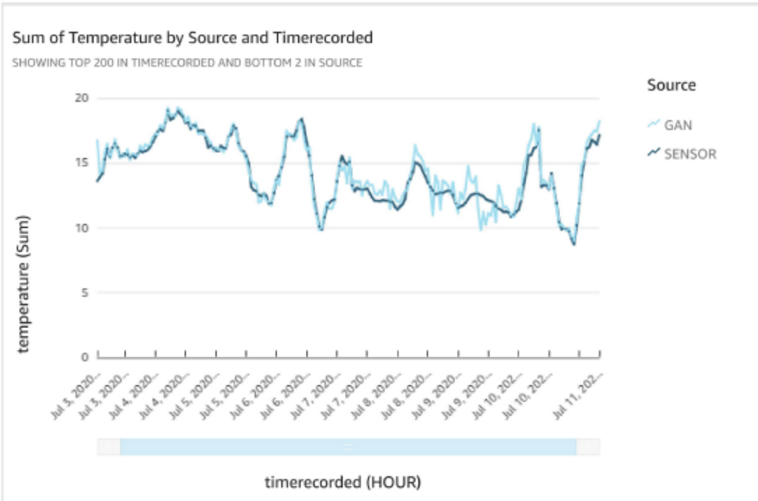


Figure 138: GAN and Sensor Values Comparison for 1 hour time period

The below graph shows the comparison between actual and prediction values for 15 minutes time period. The predictions by both algorithms is very close to the actual value. From the graph it can be concluded that the accuracy of both the algorithms is very high. The trend of the graph is not smooth as the sensor data. The GAN can imitate the distribution of the source data but could not really mimic the trend or smoothness of the sensor data.

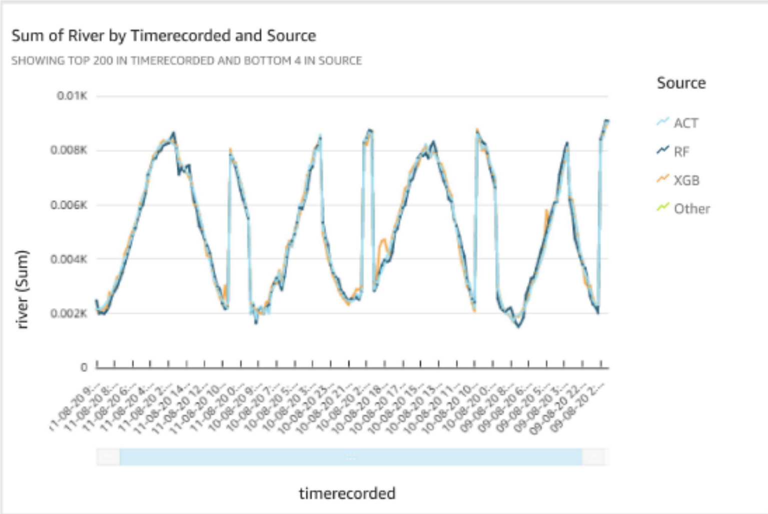


Figure 139: 15 Minutes Predictions and actual values

The below graph is the visualization between actual and prediction values for 1 hour time period. This graph shows distinct gap between actual and prediction values. The accuracy is high but is not that high as in 15 minutes time period. This can be due to less trend details - for 2 hours of data, 1 hour time period has 2 records whereas 15 minutes time period has 8 records. Probably more historical data can assist in understanding the trend and improving the performance of the models.

Since this dataset has all the required parameters like rainfall, temperature, wind, etc. and also a desirable time period, it was an ideal dataset for this research. Since the dataset download was restricted to 1 month by the API, more data could not be obtained.

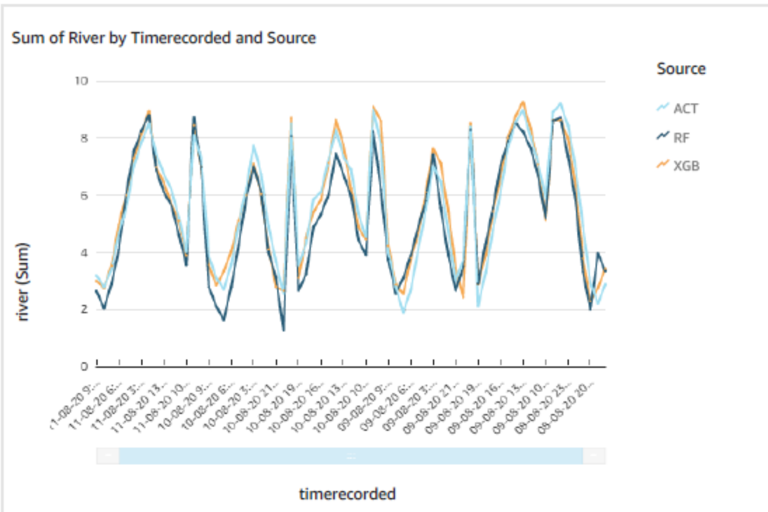


Figure 140: 1 Hour Predictions and actual values

A mission critical prediction model cannot be judged solely based on Statistical Tests. The extent of fit between the forecast and prediction conveys least on the errors and achievements of the prediction model.

Accuracy is the extent of error in the prediction. This is assessed by finding the sum of the difference between actual and predicted values (can be termed as Prediction Error). Sensitivity/Efficiency in this scenario is assessed based on the number of flood warnings triggered correctly. An algorithm can be efficient to trigger the flood warning but should not be erroneous. Specificity/Reliability is assessed based on the number of erroneous flood warnings triggered. Precision is the number of accurate warnings triggered divided by the actual number of warnings. Influenced by Furquim et al. (2018), the below table summarises these details for both algorithms on both time periods. As per Hagen et al. (2020), although the statistical tests indicate that the model is a very accurate fit, but the hit and miss rates of the algorithm conveys room for improvement.

Sum of Prediction Error (Accuracy): XGBoost: -7.98 Random Forest: 21.58
Highest Prediction Error: XGBoost: 0.38 Random Forest: 0.74
Total Number of Flood Warning Triggers: 103
Total Correct Flood Warnings Triggered (Sensitivity): XGBoost: 102 Random Forest: 94
Total Flood Warnings missed: XGBoost: 1 Random Forest: 9
Total Erroneous Flood Warnings Triggered (Specificity): XGBoost: 0 Random Forest: 0
Precision: XGBoost: 99.03 Random Forest: 91.26
R-Square Score: XGBoost: 99.4 Random Forest: 98.75

Table 1: 15 Minute Time Period

Sum of Prediction Error (Accuracy): XGBoost: -1.86 Random Forest: 24.79
Highest Prediction Error: XGBoost: 1.32 Random Forest: 1.60
Total Number of Flood Warning Triggers: 29
Total Correct Flood Warnings Triggered (Sensitivity): XGBoost: 27 Random Forest: 21
Total Flood Warnings missed: XGBoost: 2 Random Forest: 8
Total Erroneous Flood Warnings Triggered (Specificity): XGBoost: 1 Random Forest: 1
Precision: XGBoost: 93.10 Random Forest: 72.41
R-Square Score: XGBoost: 94.01 Random Forest: 85.75

Table 2: 1 Hour Time Period

Figure 141: PASS Evaluation Table

From the above table it is clear that XGBoost has outperformed Random Forest in all aspects. It has saved time and lives of the people. As evident from the graph, the accuracy of 15 minutes time period is greater than the 1 hour time period. Hence, with increasing time period, the accuracy decreases possibly due to less detail of trend. Neural Networks were not implemented due to time and complexity issues. Also a historical dataset with more number of records would enable better accuracy of models. Use of Data assimilation would be required. As mentioned in Hu et al. (2019), as the number of historical records increases with a small time period, there would be instances where river level has least to no change which distort the trend. Hence removal of those field and using Data Assimilation would be required. Also, use of ensemble data in Hagen et al. (2020) enabled prediction beyond one week.

Flash Flood Prediction as well as flood prediction for about 3 days was achieved. Also, XGBoost was implemented for the first time in flood prediction domain which has outperformed Random Forest, a popular algorithm used for flood prediction.

References

Furquim, G., Filho, G. P. R., Jalali, R., Pessin, G., Pazzi, R. W. & Ueyama, J. (2018), 'How to improve fault tolerance in disaster predictions: A case study about flash floods using iot, ml and real data', *Sensors* **18**(3). Impact Factor = 2.475.
URL: *https://www.mdpi.com/1424-8220/18/3/907*

Hagen, J. S., Cutler, A., Trambauer, P., Weerts, A., Suarez, P. & Solomatine, D. (2020), 'Development and evaluation of flood forecasting models for forecast-based financing using a novel model suitability matrix', *Progress in Disaster Science* **6**, 100076. Impact Factor = 2.1.
URL: *http://www.sciencedirect.com/science/article/pii/S2590061720300132*

Hu, R., Fang, F., Pain, C. & Navon, I. (2019), 'Rapid spatio-temporal flood prediction and uncertainty quantification using a deep learning method', *Journal of Hydrology* **575**, 911 – 920. Impact Factor = 3.73.
URL: *http://www.sciencedirect.com/science/article/pii/S0022169419305323*