Master Project

Prediction of extreme events using images from precipitation radars

Institute for Software and Systems Engineering (ISSE)

Director of the Institute:

Prof. Dr. Andreas Rausch

Technical University Clausthal

MS.c Computer Science

Tutors:

M. Sc. Fidae El Morer

Dhananjeyan Jeyaraj

Supervisor:

Dr. Stefan Wittek

Author:

Arman Abouali

SS 2023 – WS 2023 / 24

Introduction:

Extreme weather events, magnified by the ongoing challenges of climate change, continue to have devastating impacts on communities, infrastructure, and human lives. A case in point is the heavy precipitation that led to catastrophic flooding in northern Germany on July 26, 2017. The worst of the flooding was concentrated in the Harz region, including the city of Goslar, a UNESCO World Heritage Site, which was severely inundated. In the state of Lower Saxony alone, at least four rivers reached record highs, leaving cities isolated, disrupting public services, and necessitating evacuations.

Despite existing information systems providing some level of warning, the 20-to-30-minute lead time was insufficient to prevent substantial damage and loss. For example, the Innerste River reached unprecedented levels of 7.14 meters in Heinde, surpassing its previous record set in 2007. This raises questions about the adequacy of current early warning systems and emphasizes the need for more reliable and effective solutions.

In this context, the potential of Artificial Intelligence (AI) and machine learning-based predictive models is considerable for enhancing early warning systems. Leveraging historical sensor data and additional variable such as precipitation radar imagery data from Deutsche Wetterdienst (DWD), these advanced systems aim to extend the narrow window of preparation time. Nonetheless, these models face the inherent challenge of accurately predicting rare events based on historical data alone. Therefore, this research focuses on integrating additional data sources into machine learning algorithms to develop a more robust and reliable early warning system for rare but devastating events such as floods.

Consequently, the primary objective of this master's project is to develop a functional framework that employs radar-based precipitation data from Deutsche Wetterdienst (DWD) to forecast water levels as captured by multiple sensors situated in the city of Goslar. This research aims to address the following question:

Is it possible to enhance the accuracy of flood predictions by incorporating supplementary data sources such as precipitation measurements, thereby surpassing the capabilities of predictive models that solely rely on historical water level sensor data?

Data Exploration:

This research employs two time-series datasets:

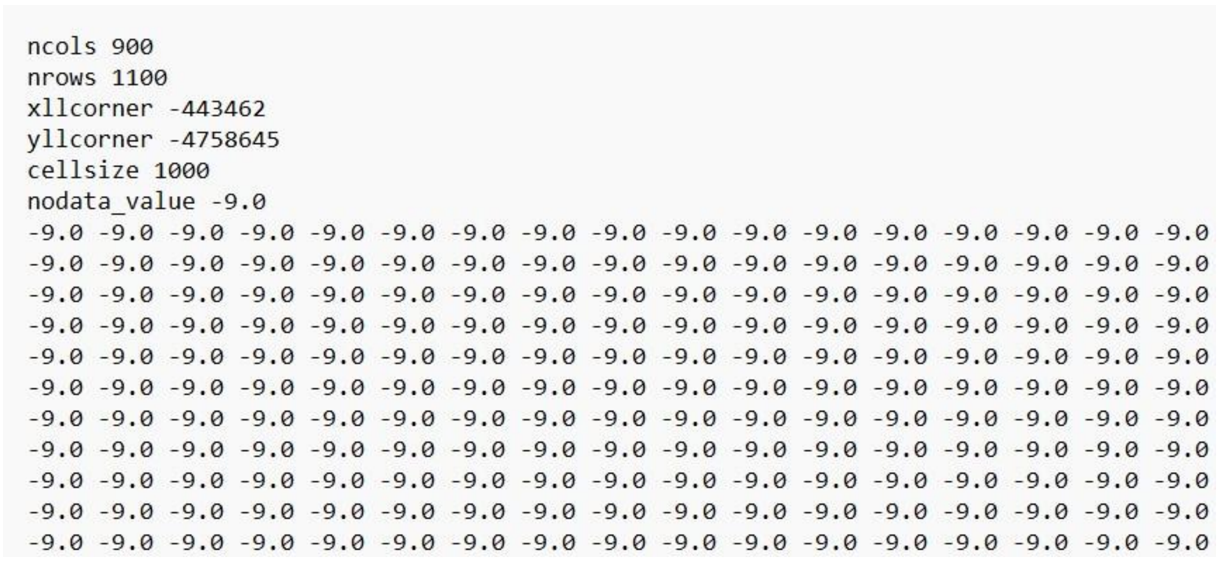
1. **Radar Imagery Data:** Obtained from Deutsche Wetterdienst (DWD), this dataset offers radar-based precipitation measurements.
2. **Water Level Captured by Sensors**: Features sequential time-series water level readings from sensors placed at various rivers around the target area.

Both datasets serve as key components for developing an advanced early warning system. Consequently, for a more comprehensive understanding of these datasets, each will be examined in detail in this section.

- **Radar Imagery Data**

The dataset under consideration originates from the Climate-Data-Center (CDC) section of the Deutsche Wetterdienst (DWD) website. Classified by date ranges, it offers a robust historical record of precipitation levels of the entire country (grids_germany), accessible from the year 2001 to 2023. Considering their accuracy and precision, this dataset stands as an additional source applied to enhance the performance of the predictive model.

A noteworthy aspect of this data is its granularity; files are generated every five minutes. These files, stored in ASCII (.asc) file format, are identical in their dimensions, measuring 1100 x 900 pixels with an approximate size of 4,000 KB each. The image below is a preview of a single file, which can be viewed via any text reader software, featuring six rows of information of the file to provide a better insight into the specific structure of that each of those files.



```
ncols 900
nrows 1100
xllcorner -443462
yllcorner -4758645
cellsize 1000
nodata_value -9.0
-9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0
-9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0
-9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0
-9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0
-9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0
-9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0
-9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0
-9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0
-9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0
-9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0
-9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0 -9.0
```

Interesting to note that these ASCII files can be transformed into image formats such as .tiff or .JPEG, offering a visual representation of the data contained within them, similar to the image below, which is a converted version (.tiff) of a random image among those ASCII files.

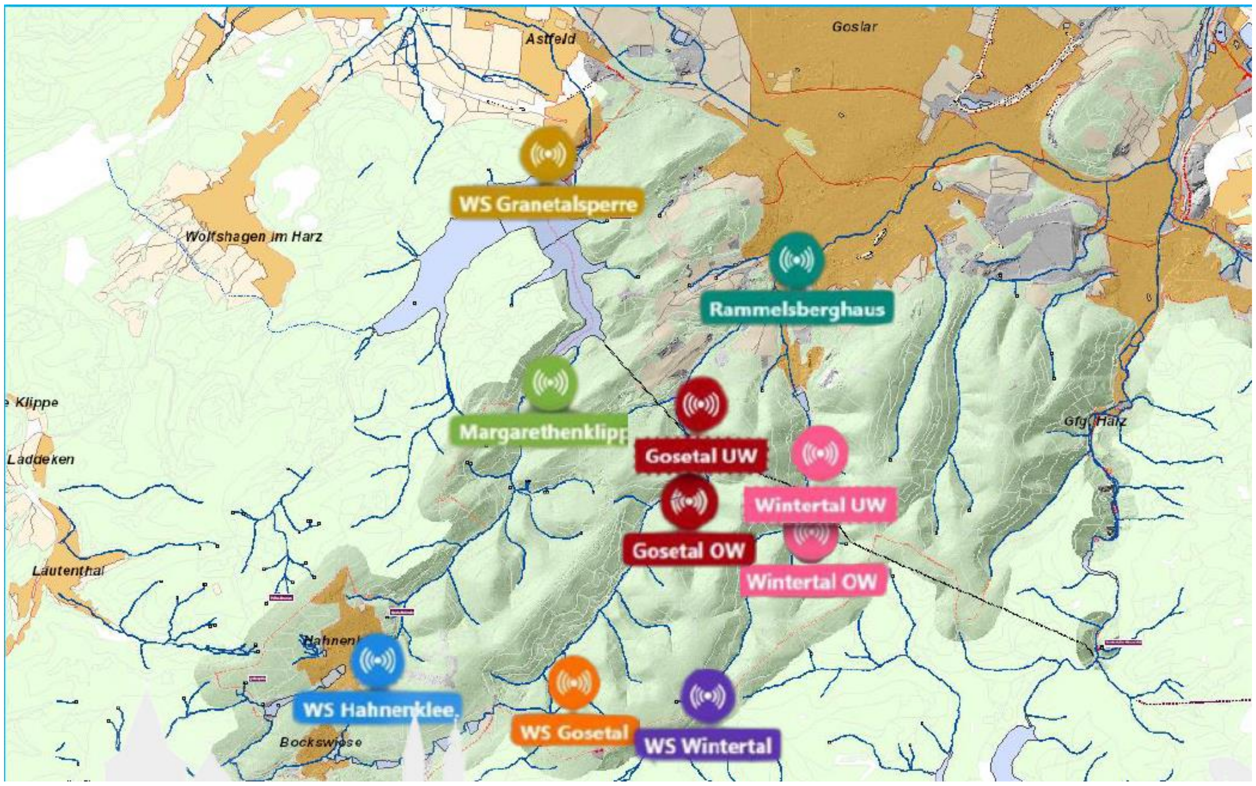
As depicted in the image below, areas marked by lighter spectrum hues signify regions with precipitation in that particular time. Lastly, concerning the nomenclature of these image files, each ASCII file is assigned a unique identifier. For example, consider the file name "YW_2017.002_20031219_1520":

"YW_2017.002_" serves as the consistent element in the naming of all files, specifying the data category.

The subsequent segment, "20031219_1520," acts as a timestamp, indicating that this particular file was captured on December 19, 2003, at 15:20.

- **Water Level Captured by Sensors**

On the other hand, there sensor-captured data to measure river water levels, the locations of which are illustrated in the image below. This dataset encompasses several attributes and a total of 514,176 observations from November 2003 to July 2018. Each observation is recorded at 15-minute intervals. The first two features serve as a timestamp, labeled "Datum" for the date and "Zeit" for the time. The remaining features contain crucial information about river water levels and rainfall amounts captured by the sensors.



Project's Overview:

Despite the existence of a system capable of providing advance notification to authorities around 20 to 30 minutes before a flooding catastrophe, the circumstance of July 26, 2017, provides convincing proof that this timeframe was inadequate to effectively prevent such catastrophe.

The existing system suggests that relying on sensor data used to measure water levels may not be adequate for predicting infrequent events such as floods. Consequently, incorporating precipitation data as a contributing factor to flood occurrences and anticipating an improvement in the performance of the Early Warning System can be viewed as a logical proposition.

The pressing need for an advanced predictive model is evident today. The main goal of this project is to assess how radar imagery data can serve as a valuable additional resource for creating and improving a comprehensive Flood Early Warning System. In order to achieve this objective, the goal is to substantially increase the duration of advance notice for flood warnings, therefore enabling authorities to implement more effective preventive measures.

The hypothesis driving this research is as follows: Can the prediction of extreme and rare events like floods be improved by incorporating additional data sources such as radar-based precipitation measurements? The primary focus of this study is to investigate the potential efficiency of multi-source predictive models over older ones that only depend on water level sensor data.

To accomplish this objective, we initially developed a bot—referred to as "DWD_Bot"—to automatically download and preprocess the radar imagery data from the DWD website. The bot is equipped with several functions to get and preprocess the project-specific data and save those data to the local disk. Additionally, a pipeline has been implemented to enable data collection in larger time intervals (when required) from the source.

One challenge we encountered was optimizing data formats and features for memory efficiency (occupying less memory storage) without compromising data quality, which is considered in the respective steps of this project.

The second challenge of this project is to find the target location —the District of Goslar in Niedersachsen, Germany— on the map and transform those coordinates into dimensions of the Radar data. It is essential to ensure that the dimensions used to segment the radar data, in terms of area of coverage, should precisely cover the locations where the sensor data is available, neither more nor less. in the coming section, I will explain in detail, the method and the procedure that is applied in this project to overcome this challenge.

Prerequisite:

Section 1: Adjust and find target locations using QGIS

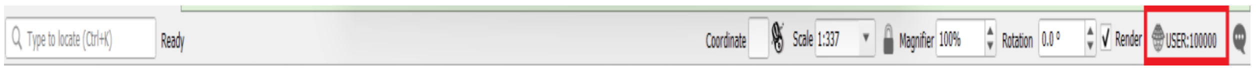
This section will clarify the procedure of accurately determining the target location, the District of Goslar in Niedersachsen, Germany, through the utilization of QGIS Desktop 3.30.3 software. The objective is to establish a meticulous and precise methodology for recognizing and extracting a new window of the target location from the original images. The objective is to ascertain the coordinates of the specific location and subsequently transform these into a four-cornered square frame that aligns with the original image dimensions of 1100 x 900 pixels. To offer a more comprehensive understanding, this methodology will be broken down into step-by-step sections to provide more clarity.

- **Step 1:**

The initial step in realizing our project objectives involves customizing the Coordinate Reference System (CRS) in the QGIS software to align with RADOLAN's grid. RADOLAN stands for RAdar-OnLine-ANeichung, a German acronym that translates to Radar-Online-Adjustment.

To accomplish this in QGIS, navigate to the 'Settings' tab on the top menu and select 'Custom Projections.' A window will appear with a sidebar featuring 'User Defined CRS' where you can input your customized coordinates. First, assign a name to your specific project. Under the 'Parameters' section, enter *"+proj=stere +lat_0=90.0 +lon_0=10.0 +lat_ts=60.0 +a=6370040 +b=6370040 +units=m*" to tailor the CRS to German standards. Finally, specify the format as "WKT" (Well-Known Text) to ensure your spatial data aligns correctly with the real-world coordinates. This is essential for the integrity and accuracy of your GIS analysis and visual representation.

Then set the current CRS from the right-most option of the bottom ribbon to the defined CRS.



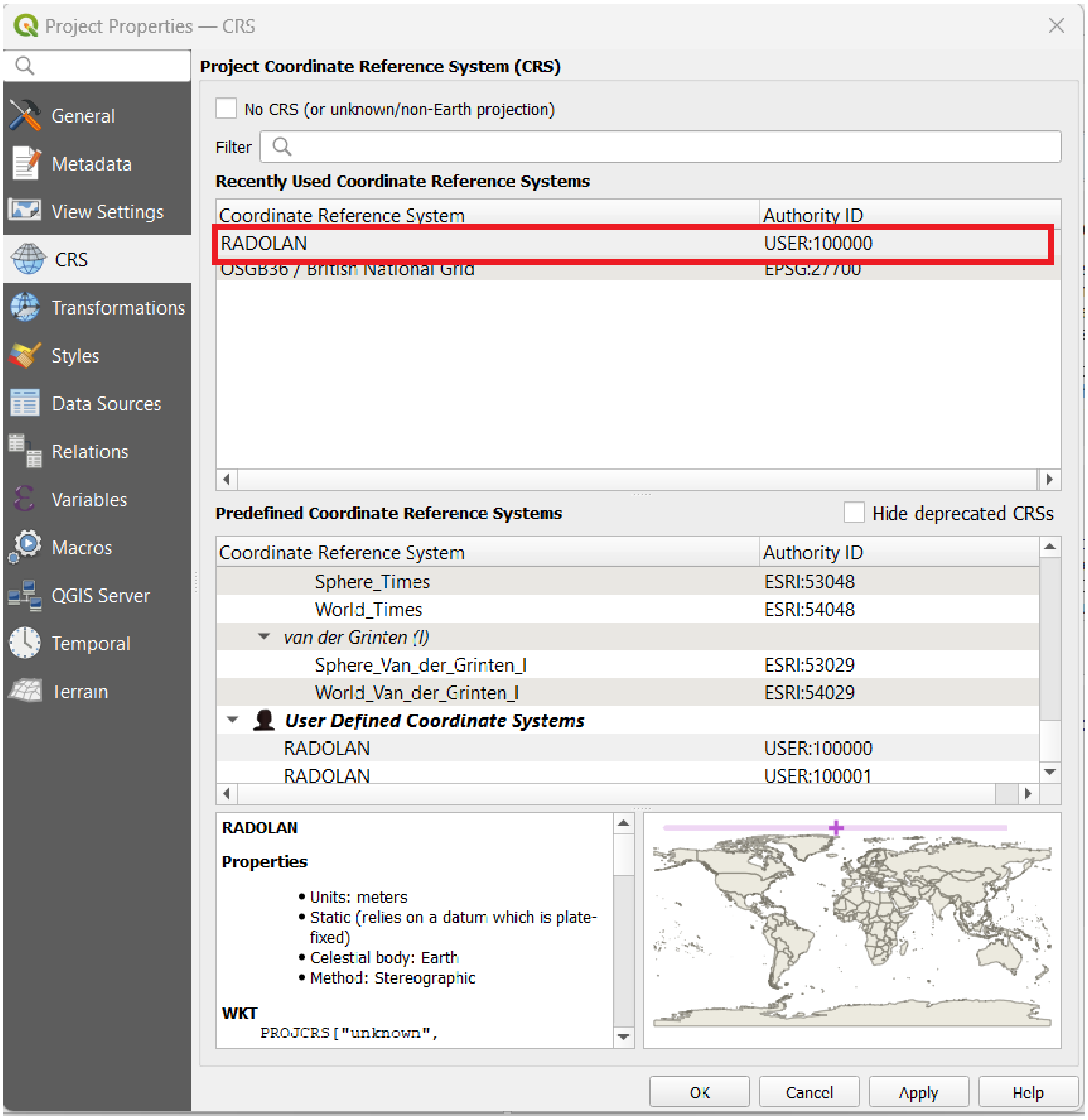
Scale 1:337

Magnifier 100%

Rotation 0.0 °

Render ✓

USER:100000



Project Coordinate Reference System (CRS)

☐ No CRS (or unknown/non-Earth projection)

Filter

Recently Used Coordinate Reference Systems

Coordinate Reference System	Authority ID
RADOLAN	USER:100000
OSGB36 / British National Grid	EPSG:27700

Predefined Coordinate Reference Systems ☐ Hide deprecated CRSs

Coordinate Reference System	Authority ID
Sphere_Times	ESRI:53048
World_Times	ESRI:54048
van der Grinten (I)	
Sphere_Van_der_Grinten_I	ESRI:53029
World_Van_der_Grinten_I	ESRI:54029
User Defined Coordinate Systems	
RADOLAN	USER:100000
RADOLAN	USER:100001

RADOLAN

Properties

- Units: meters
- Static (relies on a datum which is plate-fixed)
- Celestial body: Earth
- Method: Stereographic

WKT

```
PROJCRS["unknown",
```

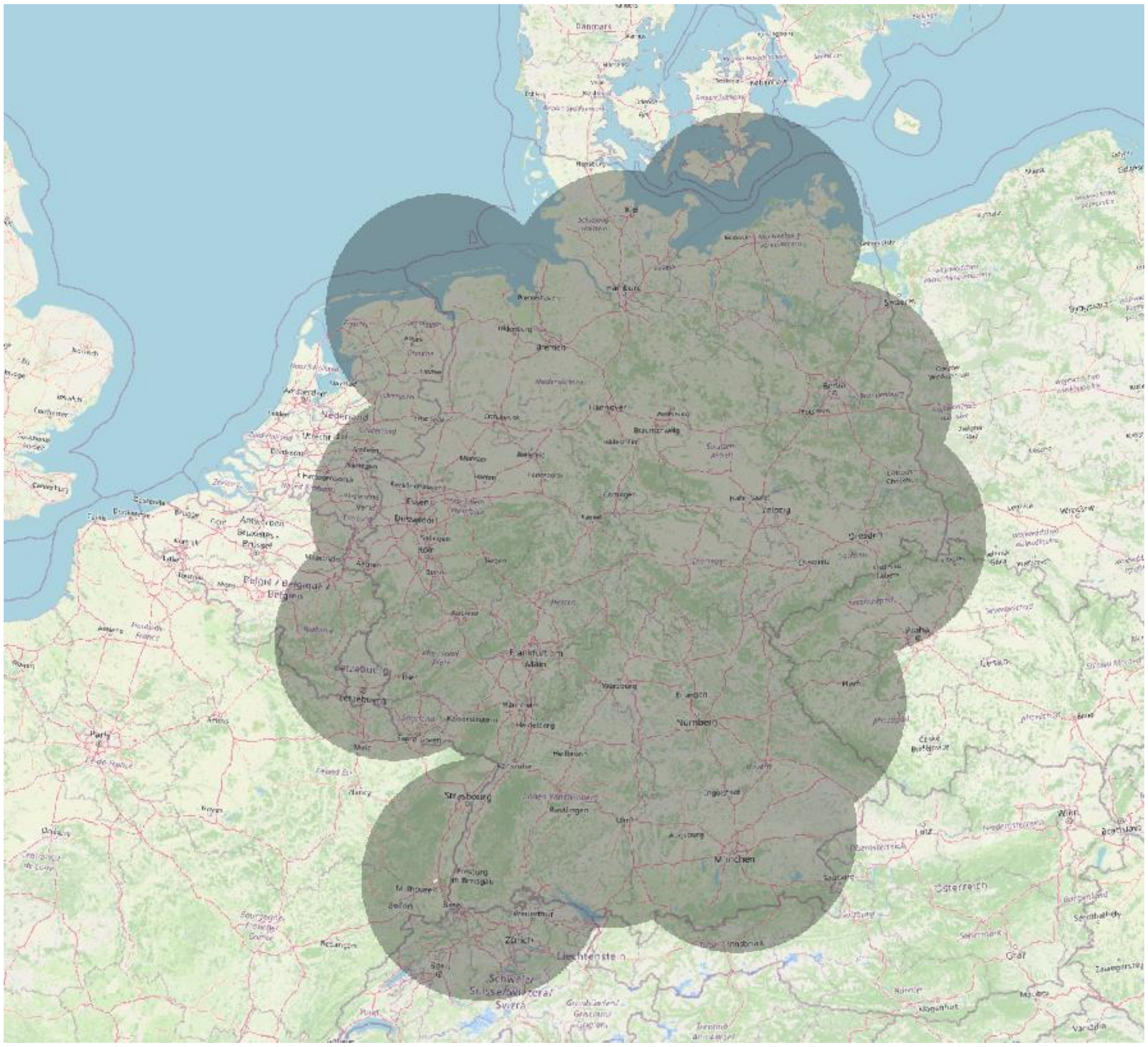
- **Step 2:**

This step requires a single TIFF file that was previously converted from an ASCII data source. The file name I have used for this task is "YW_2017.002_20031219_0010"; this file can be selected at random.

In the preceding phase, a foundational layer that encompasses Germany within a larger geographic context has already been incorporated into the GIS workspace. This created layer is shown as an "Open Street Map" (OSM standard), in the layers box, accessible via the left-side panel of the software.

With the CRS and base map set, it's time to import a converted TIFF file into the layer box of the GIS project. By dragging and dropping the chosen TIFF file into the layers box the new layer can be imported to the project.

This imported layer of the TIFF file automatically aligns over Germany due to the fact that both the TIFF and the base map layer of Germany are using the same CRS. The Coordinate Reference System is a framework that helps to ensure that the coordinates of the data match those of the real-world location, enabling seamless overlay and alignment. When the CRS of the TIFF layer matches that of the base map layer (which is set to the geographical coordinates of Germany), the imported data will fall into place correctly on the map.



- **Step 3**

In this phase of the project, the goal is to fine-tune the geographical coordinates for the target location, which is the district of Goslar in Germany. The advantage of the TIFF layer created in the previous step is to access the coordinates of the entire map of Germany. However, what we actually need in this project is to analyze the district of Goslar, therefore, we need to narrow down the dimensions to achieve that specific location.

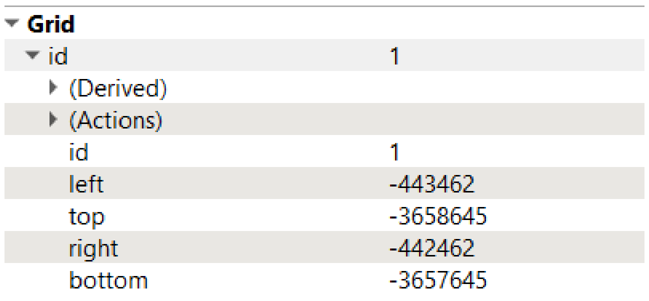
Considering the dimensions of the TIFF file are specifically 1100x900 pixels, the next task is to add a grid layer on top of the TIFF file within the working project. This grid will also have dimensions of 1100x900 to align perfectly with the TIFF layer. The purpose of creating this grid is to facilitate the precise identification of the district of Goslar within the larger geographical context of Germany.

This process ensures that the map can be 'graded' at a scale that makes it easier to pinpoint the exact coordinates of the target location, using the same 1100x900 dimensions as the TIFF file.

To overlay a grid on the TIFF file, navigate to the "Vector" option in QGIS's top menu and choose "Research Tools," followed by selecting "Create Grid." In the window that appears, switch to the "Parameters" tab and set the Grid Type to "Point." Next, you'll need to specify the grid's extent. To do so, use the drop-down menu to choose "Calculate from layer" and then pick the TIFF file layer relevant to your project, which in this case is the layer named "YW_2017.002_20031219_0010".

Adjust the vertical and horizontal spacing of the grids to 1000 meters to align with the image dimensions. Ensure that the Grid CRS is set to RADOLAN coordinates. Finally, click "Run" to produce the grid.

As a result of completing this step, you will obtain a grid with dimensions of 1100 x 900. Each point within this grid is equipped with both an identification number and geographical coordinates. The identification numbers are assigned in a column-wise, top-down manner, starting with the top-left corner point labeled as ID 1, and culminating with the bottom-right corner point carrying the ID of 990000. This identification and coordinate information for the grid point can be accessed through the software's "Identify Feature" tool on the top ribbon, recognizable by this specific icon shape.

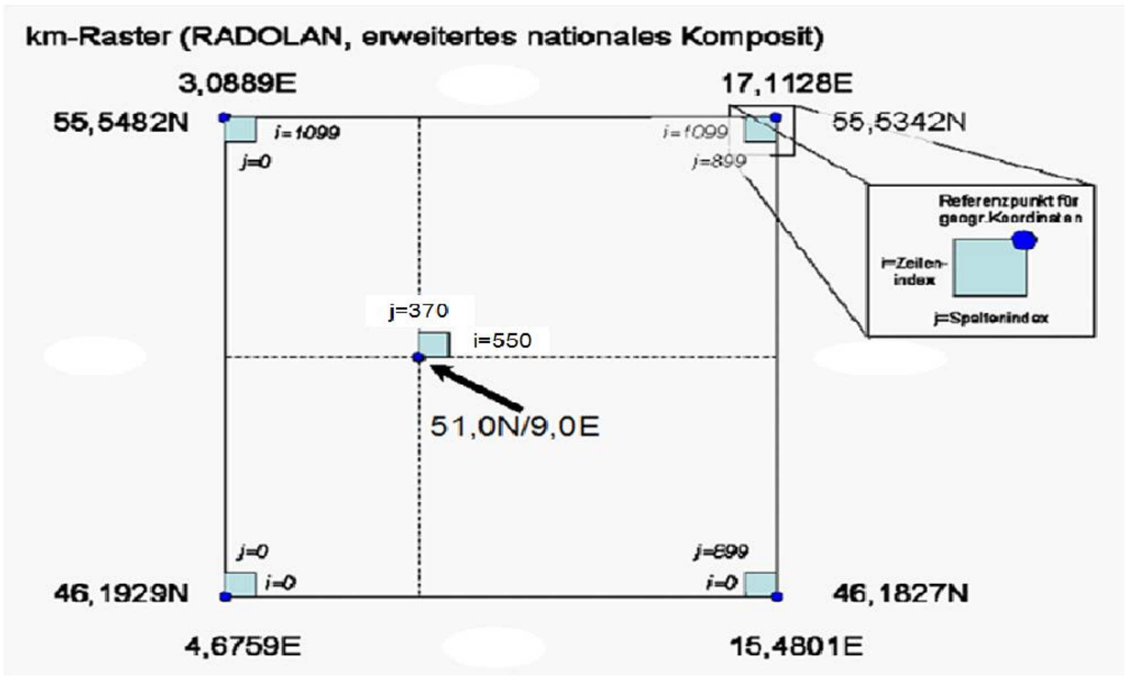


Grid	
id	1
(Derived)	
(Actions)	
id	1
left	-443462
top	-3658645
right	-442462
bottom	-3657645

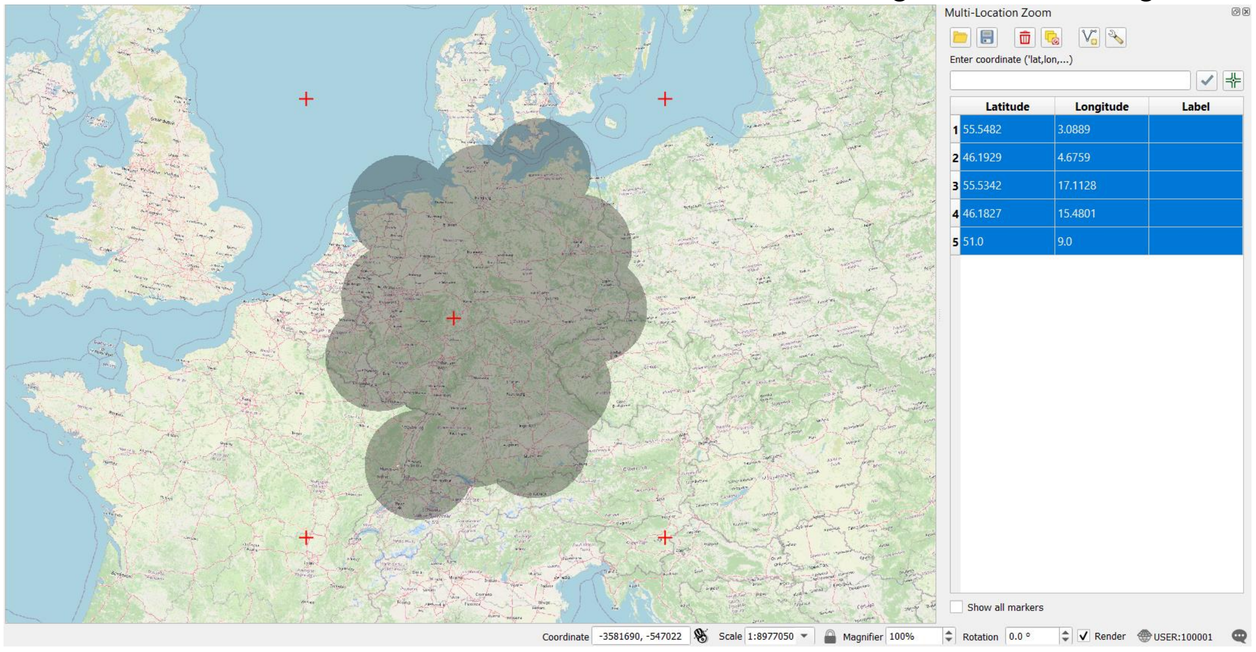
- **Step 4:**

In the RADOLAN documentation provided by the Deutscher Wetterdienst (DWD), it became evident that a precise alignment of our QGIS grid with the actual coordinates from the RADOLAN data was crucial for accurate analysis. Therefore, these coordinates have to be compared with the calculated grid on the ASCII (tiff) file to ensure the correct and accurate location of the grid for further data extraction.

To accurately map these coordinates in the defined QGIS project, we utilize the capabilities of the 'Lat Lon Tools' plugin. This robust plugin enables us to search for, pinpoint, and display the specified latitude and longitude coordinates specified in the documentation within QGIS. This function is instrumental in achieving a precise match between our map and the RADOLAN data.



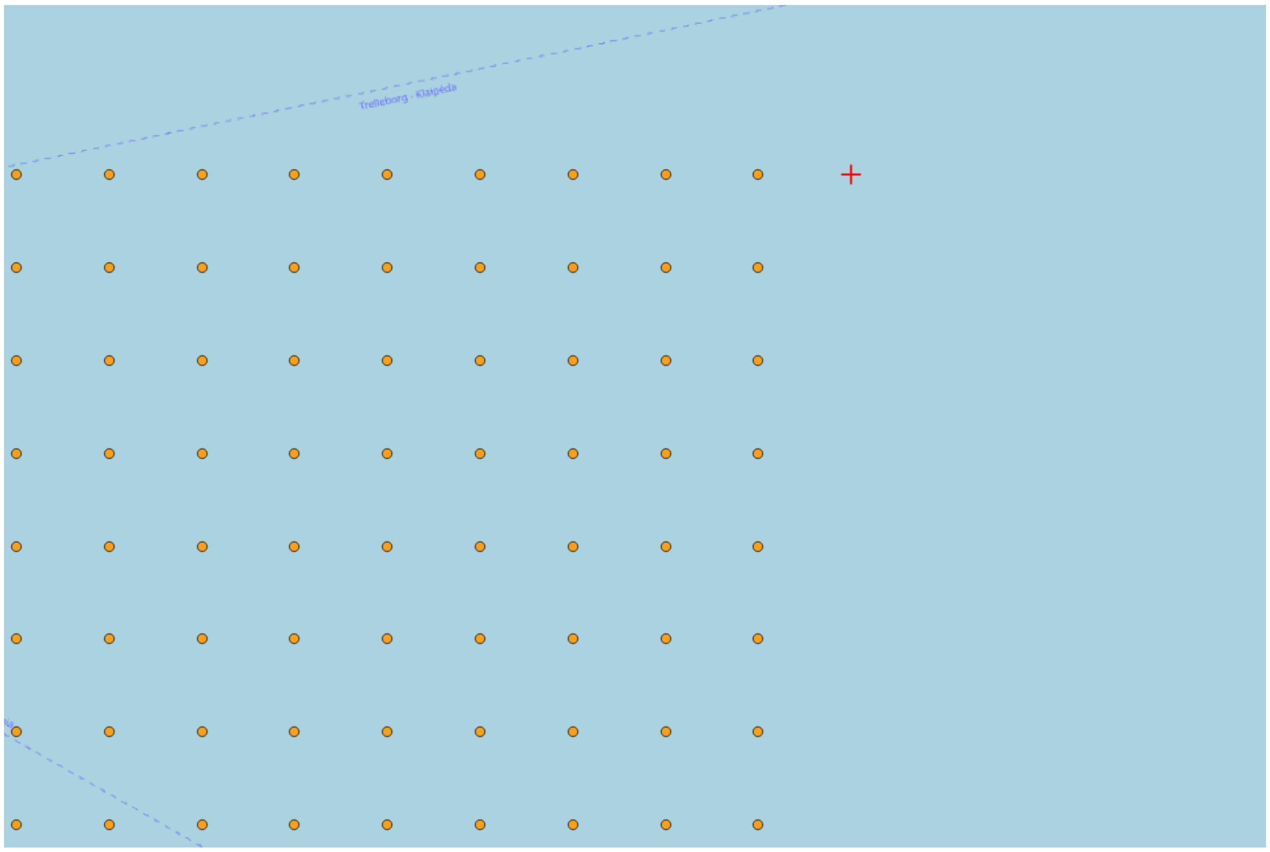
By using the 'Lat Lon Tools' plugin as mentioned before, the coordinates shown in the aforementioned picture may be identified and then compared with the radar image data, as shown in the following figure.

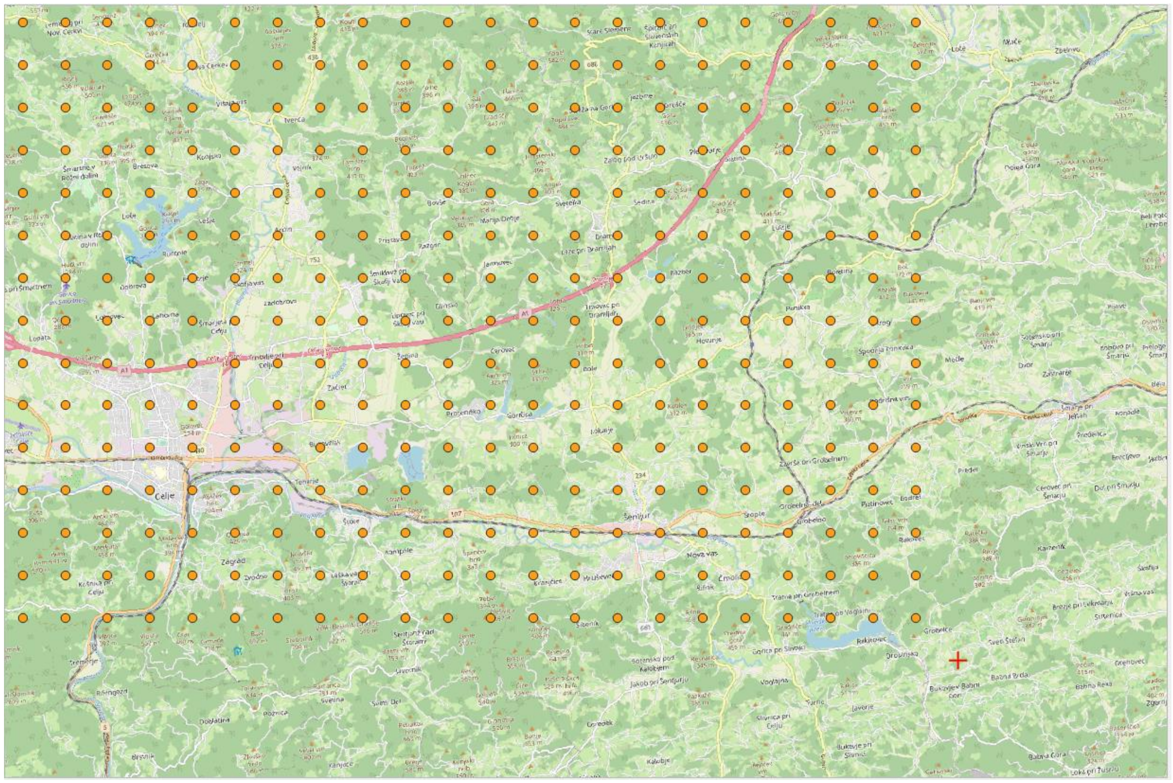


	Latitude	Longitude	Label
1	55.5482	3.0889	
2	46.1929	4.6759	
3	55.5342	17.1128	
4	46.1827	15.4801	
5	51.0	9.0	

Coordinate -3581690, -547022 Scale 1:8977050 Magnifier 100% Rotation 0.0 ° ✓ Render USER:100001

Next is to compare coordinates with the grid corners.

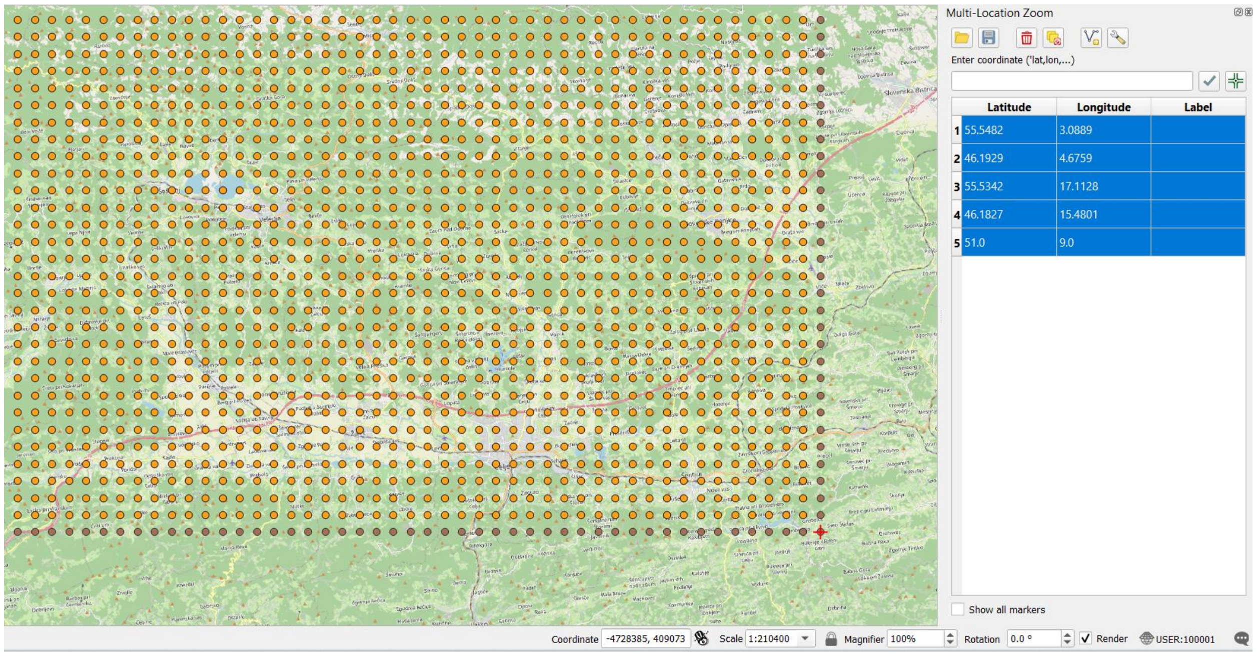




In order to achieve proper correspondence between the picture layer grid and the RADOLAN coordinates, it is necessary to make adjustments to the southern border of the grid. Based on the provided documentation, it is recommended to augment the (Ymin) value by 1000 meters to align the coordinates and achieve conformance with the corner points of the grid. [2].

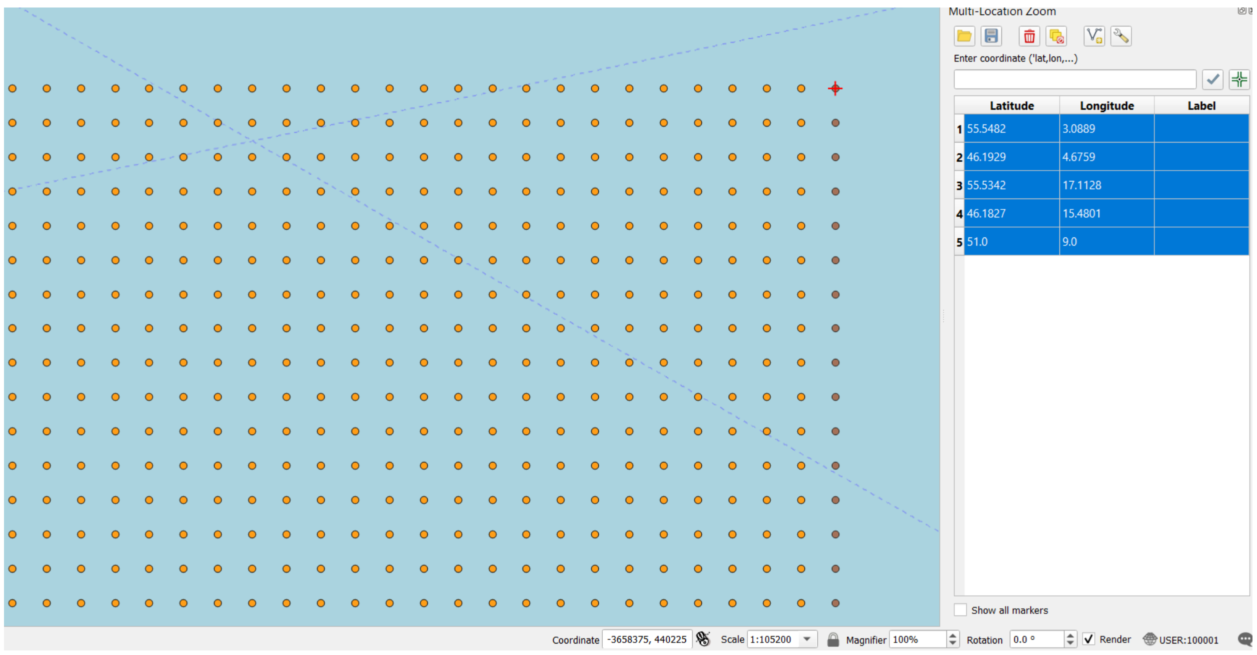
Therefore, applying the following coordinate is providing accurate match between actual coordinates of the image and the grid points:

[-443462.0000,456538.0000, -4758645.0000, -3658645.0000 [USER:100001].



	Latitude	Longitude	Label
1	55.5482	3.0889	
2	46.1929	4.6759	
3	55.5342	17.1128	
4	46.1827	15.4801	
5	51.0	9.0	

Coordinate -4728385, 409073 Scale 1:210400 Magnifier 100% Rotation 0.0 ° ✓ Render USER:100001



	Latitude	Longitude	Label
1	55.5482	3.0889	
2	46.1929	4.6759	
3	55.5342	17.1128	
4	46.1827	15.4801	
5	51.0	9.0	

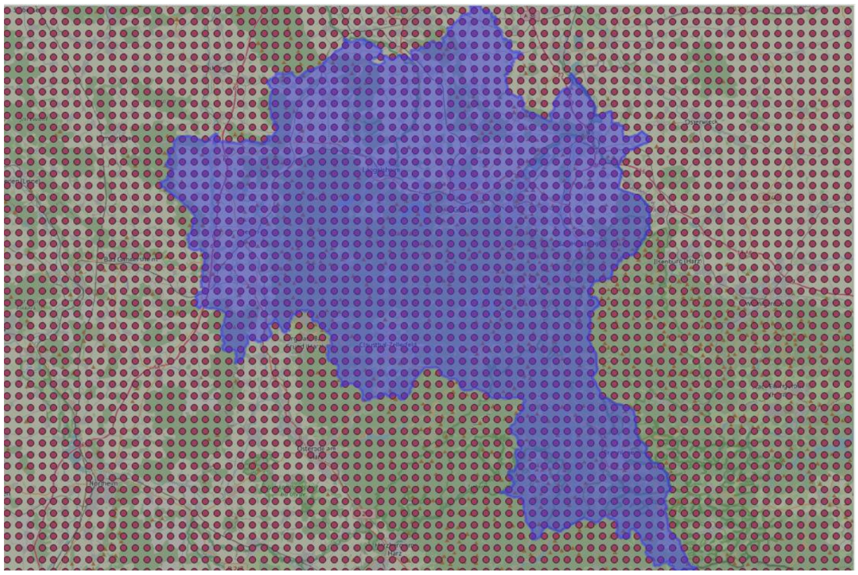
Coordinate -3658375, 440225 Scale 1:105200 Magnifier 100% Rotation 0.0 ° ✓ Render USER:100001

However, this matching results in the addition of an extra row and column to the grid point. Instead of possessing a grid consisting of 1100 * 900 points, we now possess a grid consisting of 1101 * 901 points. This alteration may initially be seen as a failure, but upon closer examination, it is not. This inquiry may be addressed by recognizing that pictures are composed of pixels, whereas grids consist of points. In other words, to get a single pixel inside an image, it is necessary to determine the values of the four corner points within the grid. For example, if a grid consists of 5 * 5 or 25 points, it will result in 16 pixels. Consequently, to get 25 pixels, a grid with dimensions of 6 * 6 or 36 points is required.

- **Step 5:**

In QGIS, a range of built-in tools are available to meet various user needs. One such useful tool is the "OSM Place Search," which can be installed via the "Plugins" tab located in the top menu bar. After installing this plugin, you will see the "OSM Place Search" box appear in the software's left-hand column.

To use this feature, simply type "Goslar" into the search box. A list of matching locations will be displayed in the text box below. Locate the one that aligns with your specific needs and double-click it to zoom into your desired location on the map.



- **Step 6:**

Passing all the previous steps, you can now narrow down your focus to the specific area of interest. To do this, we'll utilize QGIS's polygon-drawing feature to identify the grid IDs that correspond to your target location.

Start by navigating to the second top ribbon and selecting "New Shapefile Layer." This will open a new window where you can define the attributes of your shapefile. Begin by naming the shapefile, which will be saved with a .shp extension, and choosing a location on your local disk to store it. Then, choose a name for your shape, and set the Geometry Type to "Polygon." After clicking "Add to Field List," make sure to remove any previous filenames from the list and click "OK". Then your newly defined shape is visible in the layers list.

From the same toolbar, locate the "Toggle Editing" option to activate additional features on the ribbon. Once activated, click on "Add Polygon Feature," which will allow you to start drawing your polygon on the map.

Create your preferred polygon—either rectangular or square—so that it encompasses the area of interest. Once you have defined all four corners, right-click to finalize the shape. A dialog box will appear, allowing you to select a numerical value that identifies the color of the polygon. Lastly, write down the ID numbers assigned to the edges of your polygon.

Section 2: Introducing the DWD Bot

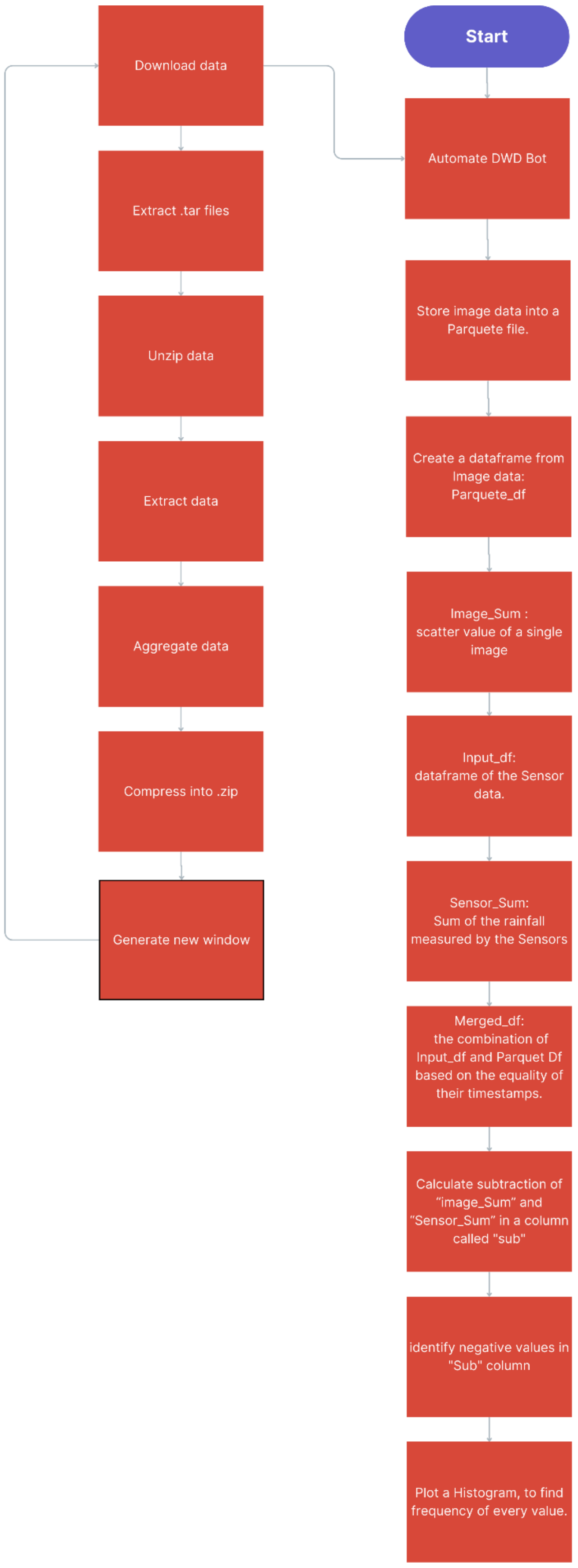
Introduction:

In order to acquaint yourself with the DWD Bot, it is essential to first familiarize yourself with the protocol for downloading and extracting picture data from the CDC section of DWD. When obtaining the data with the provided URL: "*https://opendata.dwd.de/climate_environment/CDC/grids_germany/5_minutes/radolan/reproc/2017_002/asc/*", The provided information displays a compilation of years during which radar picture data is accessible for download and storage on a local disk, without any associated cost.

By selecting any given year, users will be able to see a comprehensive list of the twelve months associated with that specific year. Each month will offer the option to download image files in ASCII format, conveniently packaged into a .tar file. Therefore, you won't have the possibility of downloading data for the entire year, instead unit of download a file is data of a month.

In order to gain access to the aforementioned files, it is imperative to initiate the process by downloading the data pertaining to the specific month. Subsequently, the file must be extracted into a newly created directory. Following this, each individual file, symbolizing a day within the given month, must be unzipped from its compressed .gz (GZip) format. Once this step is completed, each file must be further extracted and unzipped to unveil a total of 288 .asc (ASCII) files, which serve as representations of images for a singular day. The aforementioned technique must be iterated for each day's data in order to get and retrieve all image data for a selected month.

The emergence of the DWD Bot presents an opportunity to streamline and simplify the process of retrieving image data from a given source within a certain time frame. By automating this technique, the bot minimizes the need for manual manipulation and offers a more efficient means for obtaining the desired data. In order to provide a better insight into the bot, I have provided a flowchart, explaining structure of the bot.


Download data
Extract .tar files
Unzip data
Extract data
Aggregate data
Compress into .zip
Generate new window
Start
Automate DWD Bot
Store image data into a Parquete file.
Create a dataframe from Image data: Parquete_df
Image_Sum : scatter value of a single image
Input_df: dataframe of the Sensor data.
Sensor_Sum: Sum of the rainfall measured by the Sensors
Merged_df: the combination of Input_df and Parquet Df based on the equality of their timestamps.
Calculate subtraction of "image_Sum" and "Sensor_Sum" in a column called "sub"
identify negative values in "Sub" column
Plot a Histogram, to find frequency of every value.

In this part, we will elucidate each function involved in executing this process presented in the flowchart.

- **Step 1: "reshape_1D_to_2D."**

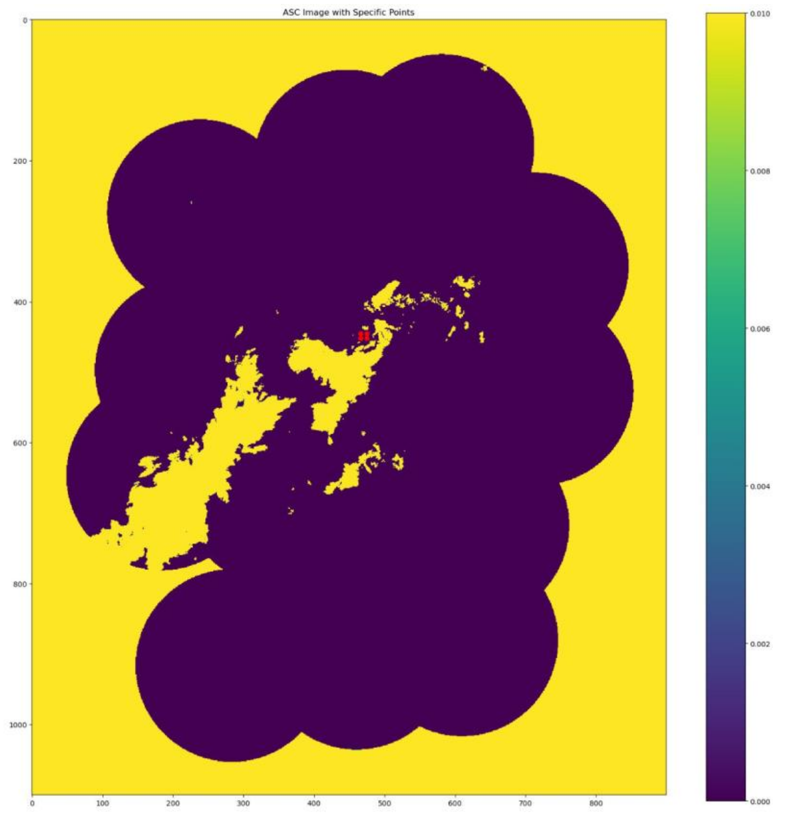
  In order to begin with the bot's procedure, it is necessary to convert the one-dimensional identification numbers (ID numbers) of the grid points obtained in the last step of the previous section in QGIS to a two-dimensional representation to have the coordinates of the target location so that Bot would be able to slice the target windows based on the captured coordinate values. This two-dimensional representation corresponds to the X and Y axes of the image. These coordinate values indicate the area of interest that needs to be sliced and subjected to further analysis. It should be noted that each image encompasses the image data of the entire country. Hence, in order to advance with the algorithm in the Bot, it is necessary to transform the identification numbers of the grid into a two-dimensional representation based on the dimensions of 1101 by 901. This conversion essentially converts a one-dimensional integer into a two-dimensional array; therefore, this step can be implemented once before running the Bot's pipeline to get the coordinates of the sliced window.

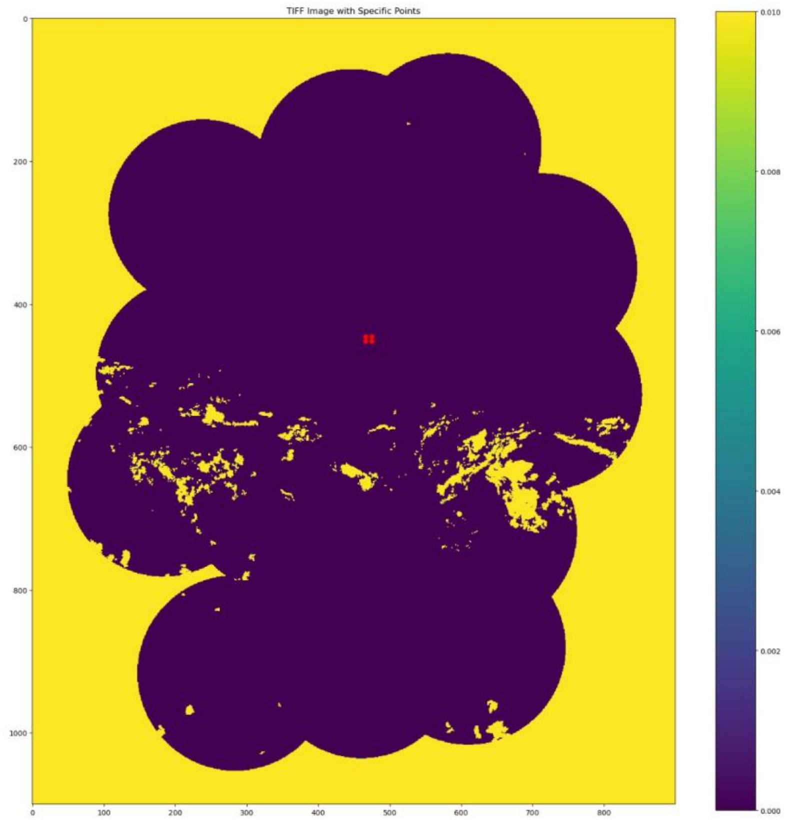
  In order to facilitate the conversion process, a function called "reshape_1D_to_2D" has been developed. This function is designed to perform the necessary transformation.

  This function requests the user to enter the grid ID values of the corners of the location of interest in QGIS and then converts them into an X and Y basis of (1101* 901).

- **Step 2: Visualization**

  Given the preceding process of obtaining the coordinates for the desired location, it is essential to establish a visualization criterion in order to get an approximate representation of the chosen area on a radar image in randomly chosen TIFF format.


ASC Image with Specific Points
0
200
400
600
800
1000
0
100
200
300
400
500
600
700
800
0.010
0.008
0.006
0.004
0.002
0.000


TIFF Image with Specific Points
0
200
400
600
800
1000
0
100
200
300
400
500
600
700
800
0.010
0.008
0.006
0.004
0.002
0.000

- **Step 3: “download_weather_data”**

Prior to delving into the elucidation of the functions, it is imperative to acknowledge the existence of several variables that serve as prerequisites for each of the aforementioned functions.

Directory_path: Indicating the directory where user wants to save the downloaded data.

year: Indicating the 'year' that user wants to download data from that particular year.

month: Indicating the 'month' that user wants to download data from that particular month.

This method includes two functions intended for downloading image data from DWD:

download_weather_data: This function initiates the process of downloading weather data for a specified year and month into a given directory. It accesses a base URL where weather data is stored, verifies the existence of the target directory, changes the current working directory if necessary, and calls the download_file function to retrieve the data.

download_file: This function is called download_weather_data and handles the actual downloading of the data file from a constructed URL. It saves the content to a file within the specified directory and notifies the user upon completion.

In summary, the download_weather_data function prepares the environment for data download and delegates the task of fetching and storing the data to the download_file function. These functions together automate the retrieval of weather data files from the source and their storage on the local file system, simplifying data acquisition for further processing.

- **Step 4: “extract_tar”**

The extract_tar function automates the extraction of .tar files located in a specified directory. It checks each file in the directory, and when it finds a .tar file, it creates a new folder named after the file (minus the .tar extension) and extracts the .tar contents into this folder. After extraction, the original .tar file is deleted to clean up the directory. The function concludes by notifying the user that the extraction and cleanup have been completed.

- **Step 5: “Unzip_gz_files”**

The Unzip_gz_files function traverses a directory and its subdirectories to find files with a .gz extension (gzip compressed files). For each .gz file located, the function:

1. Creates a new folder for the uncompressed content, named after the compressed file, ensuring the folder is created only if it doesn't already exist.
2. Opens the .gz file in binary read mode.

3. Writes the uncompressed content to a new file within the created folder, removing the .gz extension from the file name.
4. Deletes the original .gz file to free up space.
5. Once all .gz files have been processed, it prints a confirmation message that the operation is complete.

- **Step 6: “extract_zip”**

The extract_zip function searches for .tar files within the specified directory and all its subdirectories. For each .tar file found, it:

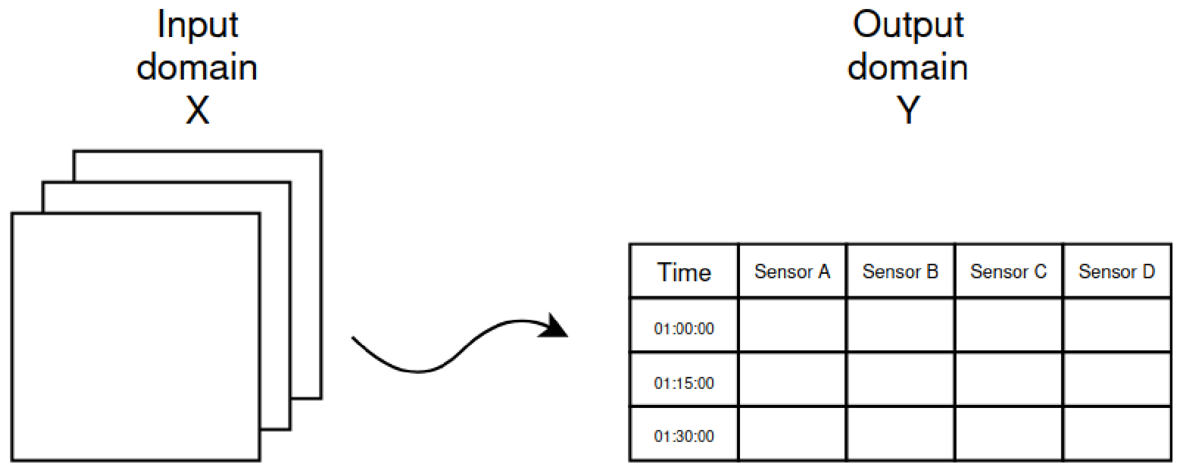
1. Opens the file and extracts its contents to the current directory of the .tar file.
2. Deletes the .tar file after extraction.
3. Prints a message indicating that all .tar files have been successfully extracted once the process is complete.

- **Step 7: “aggregate_array”**

The aggregate_array function performs aggregation of weather data (ASCII files) representing radar images. It processes files within a specified directory for a given year and month and aggregates every three images into one. This is done to synchronize the frequency of the image data (every 5 minutes) with other sensor data captured every 15 minutes. The key steps of the function are:

1. Iterates over each day of the month, constructing a subfolder name expected to contain daily data.
2. Checks if the subfolder exists; if not, it breaks from the loop, assuming no more data is available.
3. Creates an aggregation folder to store the aggregated data, using the prefix "agg_".
4. Collects and sorts all ASCII files within the subfolder.
5. Aggregates every three consecutive ASCII files by summing up the values after ensuring no negative values are present (using np.maximum to compare against 0).
6. Saves the aggregated data into a .npy file for efficient storage and access.
7. Deletes the original ASCII files to free up space.
8. Stacks all aggregated data into a single NumPy array and saves it with a filename indicating the aggregation for the entire day.
9. Deletes the original subfolder containing the ASCII files to clean up the directory.
10. Prints a completion message indicating that the aggregation process is complete.

The provided image offers a more comprehensive understanding of the functionality of the aggregate function. The image demonstrates the aggregation of every three images into a single NumPy file, enabling comparability with every single point of the sensor data.



- **Step 8: "compress_aggregated_folder"**

The compress_aggregated_folder function compresses aggregated data files into a .zip archive to save disk space. Here's how it works:

1. It receives the base directory path, year, and month for the data, as well as the destination path for the output compressed file.
2. Constructs the path to the folder containing the aggregated data based on the provided year and month.
3. Checks if the folder exists. If it doesn't, the function prints a message and returns without doing anything.
4. It creates a .zip file name based on the folder name and sets the path where this zip file will be saved.
5. Begins the compression process, opening a new zip file for writing with compression enabled (ZIP_DEFLATED).
6. Walks through the folder with the aggregated data, adding each file to the zip archive. Files are added in such a way that the directory structure within the zip file mirrors that of the source folder.
7. Once all files are added, the function prints a message indicating that the compression is complete.

This process effectively reduces the storage space required for the aggregated files, making them easier to store and transfer for future use.

- **Step 9: “generate_new_array”**

The generate_new_array function serves as the final step in a data processing pipeline for weather data. Its purpose is to extract a specific region from the original aggregated .npy files based on coordinates determined in a previous step. The function performs the following key operations:

1. Determines the window to slice by calculating the minimum and maximum x (longitude) and y (latitude) values from the provided coordinates.
2. Validates the coordinates to ensure the defined window is correct (minimum values must be less than the maximum values).
3. Iterates over each day of the specified month and year, attempting to locate the corresponding .npy file that contains the aggregated data for that day.
4. If the file exists, it loads the data and slices the array to create a new array representing the window defined by the coordinates.
5. Saves the sliced array as a new .npy file in a structured directory that organizes the data by year and month.
6. After creating the sliced arrays for all days, it cleans up the workspace by deleting the original folder containing the full-size aggregated arrays.
7. Outputs a message for each processed day and a final message to confirm the completion of the task.

The outcome of this stage is that while the image dimensions have decreased, this process has effectively reduced the required storage space for image data. In this experiment I have used an (8 * 8) window instead of the larger-sized images with dimensions of (1100 * 900).

- **Step 9.1: “Alternative”**

The rationale for storing aggregate data in a Zip file is to prevent the need for downloading the data again while transitioning to a different coordinate, hence saving time.

In the event of an alteration in interest pertaining to the target area, it is not necessary to do a complete revision of the Bot's operation. Rather, it suffices to retrieve the original data in the compressed Zip format, extract them, and then segment them according to the updated coordinates.

- **Step 10: “pipeline”**

The pipeline function orchestrates a series of steps to process weather data over a range of years and months. The process involves downloading, extracting, and manipulating the data, and is summarized as follows:

1. The function loops through each year and month within the specified range.

2. For each month of each year, it sequentially executes a series of data handling functions:

- download_weather_data: Downloads the weather data archives for the specified year and month to the directory_path.

- extract_tar: Extracts the contents of the downloaded .tar files.

- Unzip_gz_files: Decompresses .gz files within the extracted directories.

- extract_zip: Handles the extraction of .tar files, which appears to be misnamed as it should be dealing with .zip files based on its name.

- aggregate_array: Aggregates the data from individual files into a single array for each day, summing up values to match the temporal resolution of other sensor data.

- compress_aggregated_folder: Compresses the folder containing aggregated data into a .zip file to save space and make the data more manageable.

- generate_new_array: Slices the aggregated data based on the provided coordinates to create new arrays that focus on a specific region of interest.

3. After processing all the months of all the years in the range, the function prints a message indicating successful completion of the pipeline.

The function requires the user to input the start and end years for the data processing. It uses a set of predefined coordinates to slice the data. The directory_path is where the data will be downloaded and processed, while the output_compressed_path is where the compressed aggregated data will be stored.

By calling pipeline, the entire process is initiated, handling each step automatically and sequentially, thus encapsulating the complex data processing workflow into a single, manageable function.

Section 3: Data Analysis

- **Store image data in a Parquet file.**

The create_parquet_file function is designed to compile sliced window data from .npy files into a single Parquet file, which is a columnar storage file format optimized for space efficiency and quick access to data. The function operates as follows:

1. Initializes an empty dictionary to hold the data from .npy files.
2. Iterates over the files in the specified directory path. For each .npy file:
   - Loads the NumPy array from the file.
   - Extracts the date from the file name and converts it to a datetime object.
   - For each time step (assumed to be 15 minutes apart, with 96-time steps for a full day), it stores the corresponding slice of the numpy array in the dictionary, keyed by the channel (time step) and timestamp.
3. Converts the dictionary into a panda DataFrame, with columns for the key (channel), timestamp, and values.

4. Reshapes the DataFrame so that it has a flat structure suitable for conversion into a Parquet file, with keys as timestamps and values as lists of data.

5. Uses Apache Arrow and PyArrow to convert the pandas DataFrame into an Arrow Table, which is then written to a Parquet file at the specified location.

6. Returns the DataFrame for verification or further use.
7. Prints a confirmation message indicating the successful creation of the Parquet file and displays the DataFrame.

This function effectively consolidates the individual .npy files into a single Parquet file, significantly reducing the required storage space and simplifying data access for analysis, particularly when working with pandas DataFrames.

- **Parquet_df**

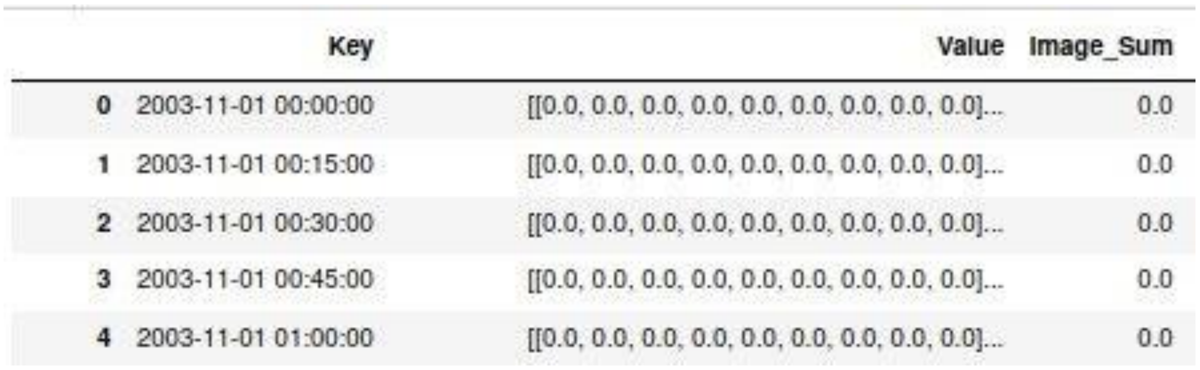
The code snippet provided adds further data manipulation to the pandas DataFrame that was generated from the .npy files and stored in a Parquet file:

1. parquet_df['Image_Sum']: A new column 'Image_Sum' is added to the DataFrame. This column is populated with the sum of all values in each 2D array (which represents a sliced window from the original files). The sum is calculated using a lambda function that iterates through each row of the array.

2. parquet_df['Key']: The 'Key' column, which contains timestamps in string format, is converted into pandas datetime objects for better time series manipulation. This conversion

is done using the pd.to_datetime function with a specified format that matches the timestamp format in the 'Key' column.

After these operations, parquet_df will contain a structured DataFrame with a timestamp column, the original 2D array data, and a new column with the sum of the values in each array. This enriched DataFrame is ready for further analysis or visualization and can be used directly within the Python data science stack.



	Key	Value	Image_Sum
0	2003-11-01 00:00:00	[[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]...	0.0
1	2003-11-01 00:15:00	[[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]...	0.0
2	2003-11-01 00:30:00	[[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]...	0.0
3	2003-11-01 00:45:00	[[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]...	0.0
4	2003-11-01 01:00:00	[[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]...	0.0

- **Input_df**

This data frame reads a CSV file representing sensor data into a panda DataFrame and then performs a series of operations to clean and organize the snesor data:

1. pd.read_csv: The CSV file is read into a pandas DataFrame, with a semicolon ; as the field separator.

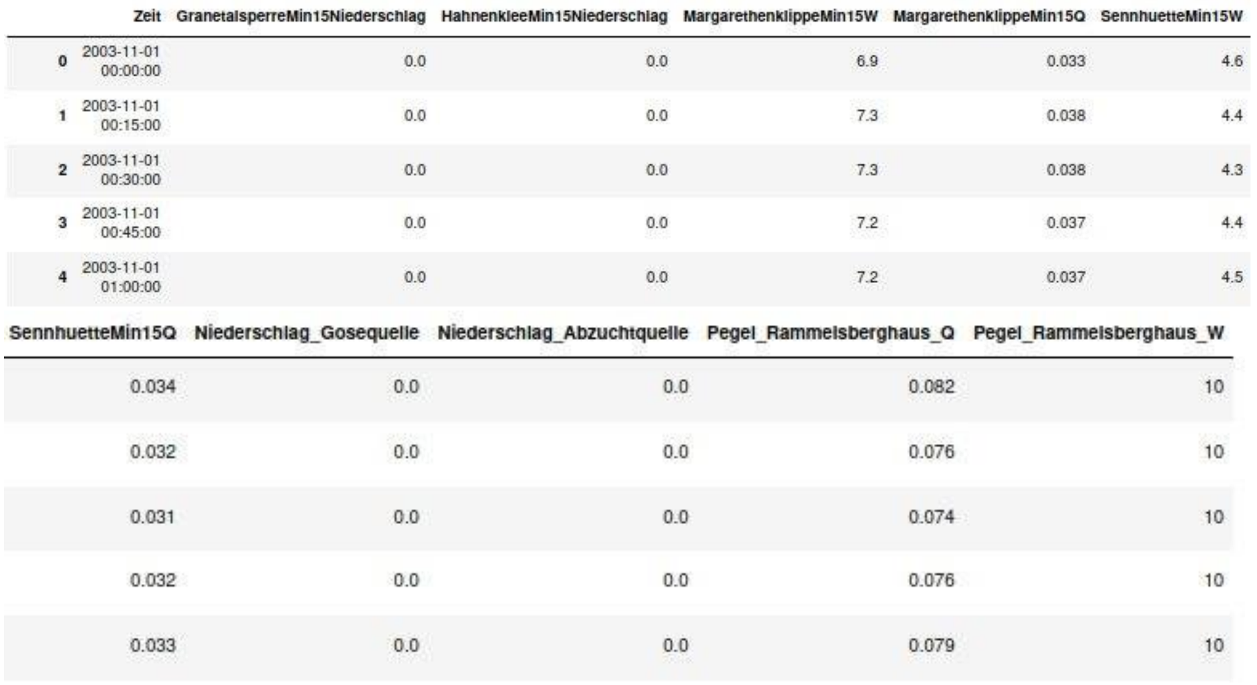
2. The 'Zeit' column, which appears to contain time values, has any occurrence of "24:00:00" replaced with "00:00:00". This is likely done to avoid any confusion or error with date-time conversion, as "24:00:00" is not a standard time representation.

3. The 'Zeit' column is then combined with the 'Datum' column (which likely contains date information) to create a full date-time representation. This combined datetime is formatted according to the specified format '%d.%m.%Y %H:%M:%S', which matches the day-month-year hour:minute:second structure.

4. The 'Datum' column is dropped from the Data Frame, as its data has already been integrated into the 'Zeit' column.

5. Finally, the Data Frame is sorted by the 'Zeit' column in ascending order and reindexed with a reset index, which is a common practice when the order of the rows is changed to ensure the index is a sequential integer starting from 0.

After executing this code, input_df will contain cleaned and chronologically ordered datetime data, with unnecessary columns removed and ready for further processing or analysis.



	Zeit	GranetalsperreMin15Niederschlag	HahnenkleeMin15Niederschlag	MargarethenklippeMin15W	MargarethenklippeMin15Q	SennhuetteMin15W	SennhuetteMin15Q	Niederschlag_Gosequelle	Niederschlag_Abzuchtquelle	Pegel_Rammelsberghaus_Q	Pegel_Rammelsberghaus_W
0	2003-11-01 00:00:00	0.0	0.0	6.9	0.033	4.6	0.034	0.0	0.0	0.082	10
1	2003-11-01 00:15:00	0.0	0.0	7.3	0.038	4.4	0.032	0.0	0.0	0.076	10
2	2003-11-01 00:30:00	0.0	0.0	7.3	0.038	4.3	0.031	0.0	0.0	0.074	10
3	2003-11-01 00:45:00	0.0	0.0	7.2	0.037	4.4	0.032	0.0	0.0	0.076	10
4	2003-11-01 01:00:00	0.0	0.0	7.2	0.037	4.5	0.033	0.0	0.0	0.079	10

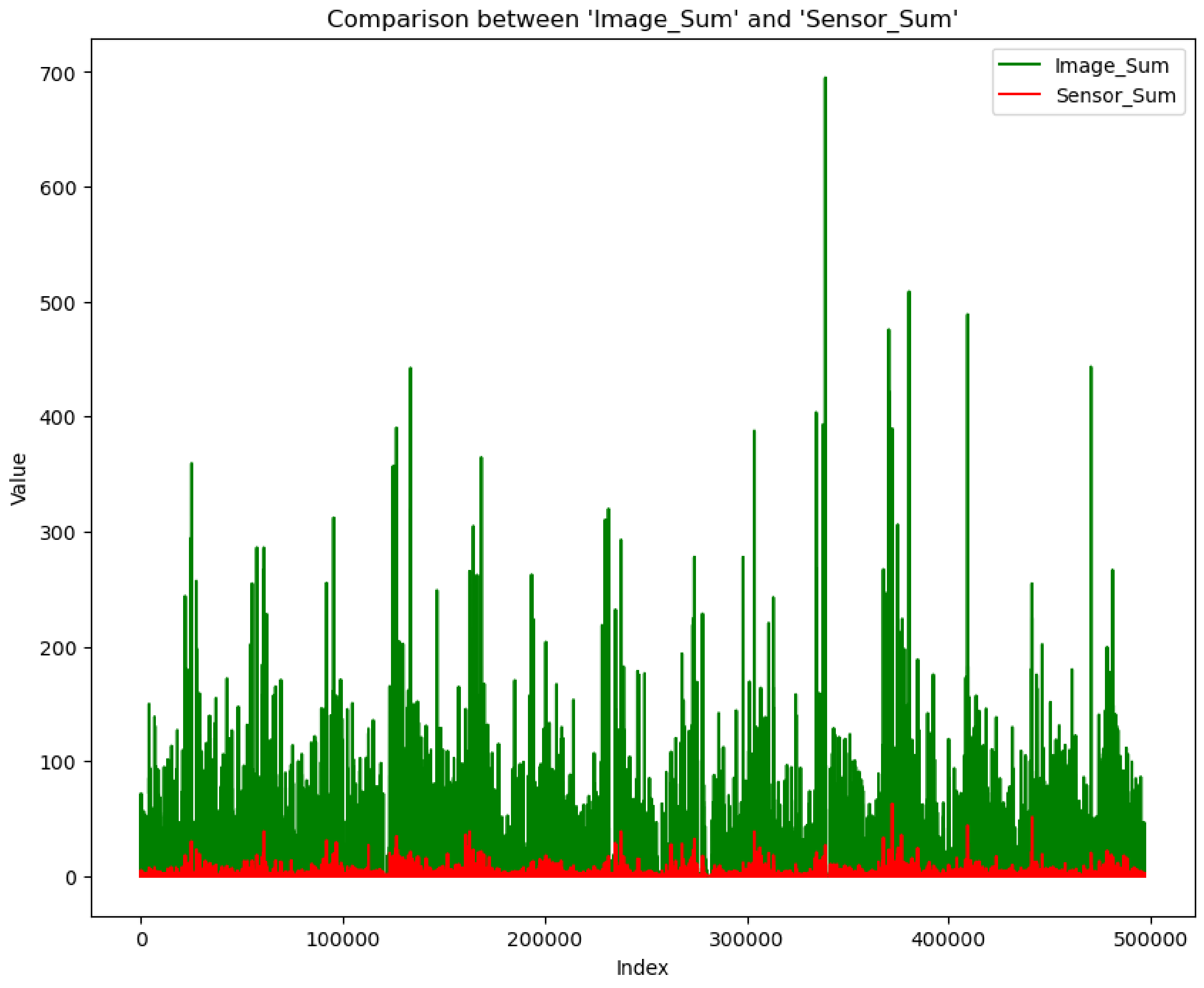
‘Sensor_Sum’a new column to the Data Frame, which is calculated as the sum of the precipitation values from four different sensors/columns: 'GranetalsperreMin15Niederschlag', 'HahnenkleeMin15Niederschlag', 'Niederschlag_Gosequelle', and 'Niederschlag_Abzuchtquelle'. This summation could be used for further analysis, such as comparing with the measurment of the precipitation by the Radar image data 'Image_Sum'.

- **merged_df**

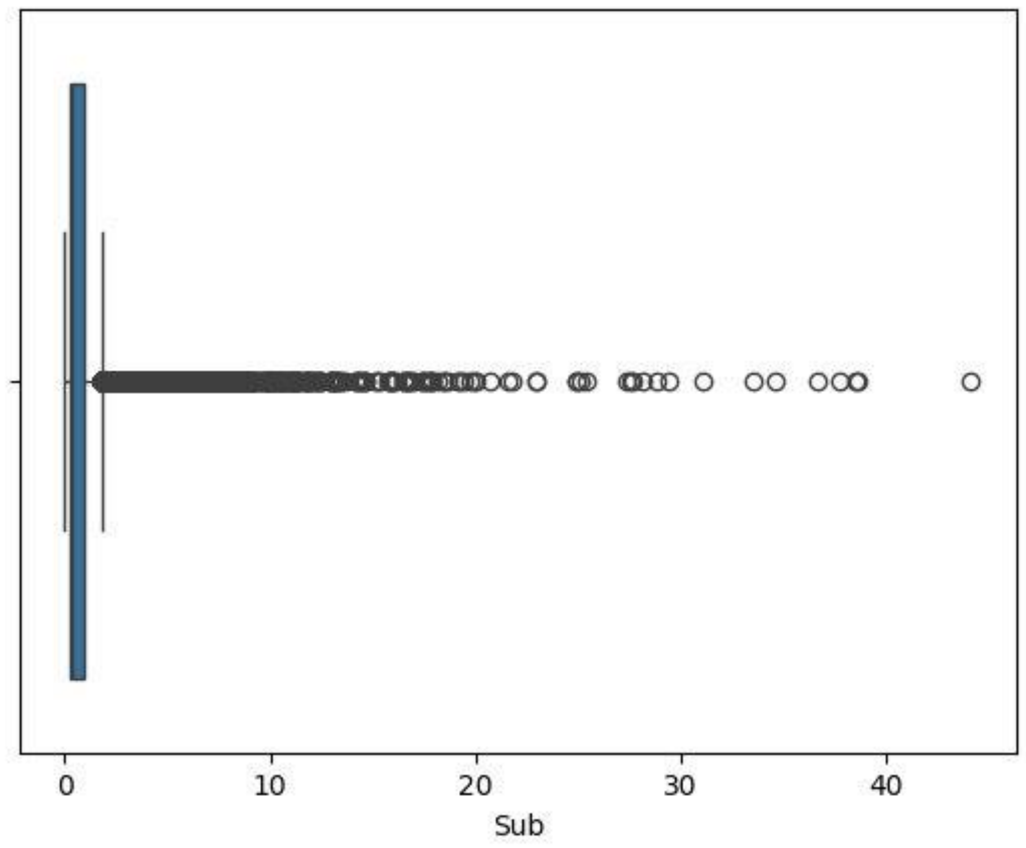
The function merge_parquet_csv is designed to combine two DataFrames, parquet_df and input_df, based on common time stamps. The pd.merge function within pandas is used to join these two DataFrames on matching values between the 'Key' column in parquet_df and the 'Zeit' column in input_df. This merge operation ensures that only the rows with the same timestamp in both DataFrames are combined into a single DataFrame named merged_df.

The resultant merged_df will contain all the columns from both parquet_df and input_df where the timestamps match, facilitating a consolidated view for further analysis.

- **Comparison**



Comparison between 'Image_Sum' and 'Sensor_Sum'
Image_Sum
Sensor_Sum
Value
Index
0
100
200
300
400
500
600
700
0
100000
200000
300000
400000
500000



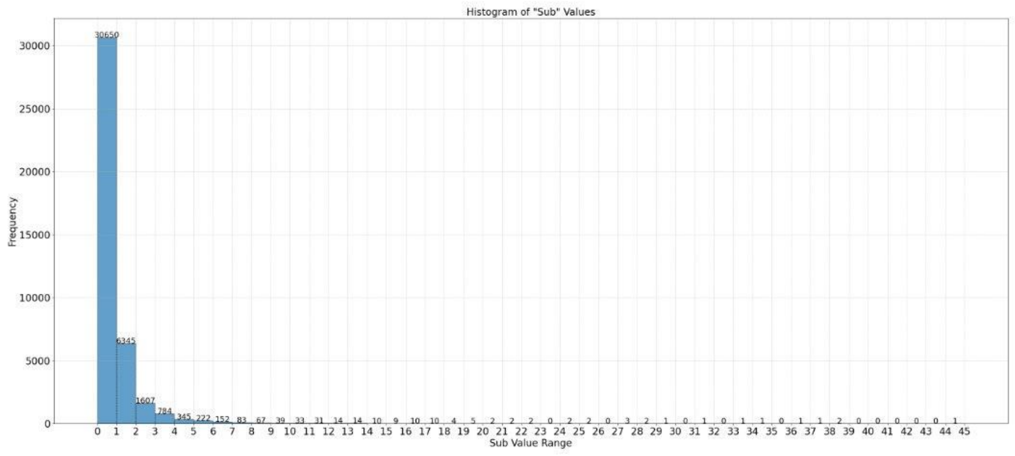
The plot visualizes the comparison between two datasets: 'Image_Sum' depicted in green and 'Sensor_Sum' shown in red. From the graph, it is evident that the 'Image_Sum' values are significantly higher than the 'Sensor_Sum' values across the index range. The 'Image_Sum' data exhibits a high level of variability with several peaks, suggesting periods of high values, which could represent high precipitation events.

The 'Sensor_Sum' data, on the other hand, appears almost constant and considerably lower in comparison to 'Image_Sum', with only minor fluctuations visible. This consistent low value could indicate a baseline or perhaps a malfunctioning sensor if the expected values should be higher.

There are no overlapping areas between the two lines, which means that at a very rare point does the 'Sensor_Sum' exceed the 'Image_Sum' across the shown index range. The index itself is likely a time series, with each point representing a specific time interval (e.g., minutes, hours, or days).

In terms of data analysis, the marked difference between the two sums suggests a need to investigate the reasons behind this discrepancy. Possible explanations could include calibration issues with sensors, different scales or units of measurement, data recording errors, or actual differences in the measured phenomena. Further investigation into the sensors' operation, the data collection process, and additional context surrounding the data points is necessary to understand the cause of these differences. In addition, this analysis indicates that 40458 of the image_sum data are less than sensor_sum, which is approximately 8% of the entire dataset, where precipitation measurement of the radar data was less than sensor data.

- **Plot a Histogram to find frequency of every value:**



The provided histogram represents the distribution of the "Sub" values, which are calculated by subtracting the 'Image_Sum' from 'Sensor_Sum'. The histogram displays a significant concentration of values at the lower end of the range, specifically in the 0 to 5 intervals, indicating that for most of the data points, the precipitation measured by the sensor is relatively close to that estimated from image data. The frequency of 'Sub' values decreases sharply as the difference increases, suggesting that large discrepancies between sensor measurements and image-based estimations are less common. This visualization aids in understanding the variance between the two precipitation measurement methods.

Section 4: Modelling

LSTM Model of Image data:

Part 1: Load Images

Here's the breakdown of the code structure and its functionality:

parquet_idx: Generates a DatetimeIndex starting from November 1st, 2003, until December 31st, 2017, with a frequency of 15 minutes. This creates a time series index which is probably intended to match the timestamps of the image data.

parquet_df.reset_index(drop=True, inplace=True): Resets the index of the DataFrame parquet_df, dropping the current index and replacing it with an integer sequence. This step is essential for reindexing the DataFrame in the next step.

parquet_df = parquet_df.set_index(parquet_idx): Sets the new index of the DataFrame parquet_df using the parquet_idx created in step 1. This aligns the rows of the DataFrame with the new time series index.

parquet_df = parquet_df.reindex(parquet_idx): This reindexes the DataFrame to ensure that it matches the parquet_idx exactly, which can introduce NaN values for any missing time points that were not already in parquet_df.

parquet_df = parquet_df.drop('Key', axis=1): Drops the 'Key' column from the DataFrame. This may be done to remove unnecessary or redundant data before the analysis.

flatten_list_of_arrays_and_divide: A defined function that takes a list of arrays (presumably each array corresponding to image data) and performs the following steps:

Concatenates the arrays into a single array.

Divides every element of this array by 10, which could be for normalization or unit conversion.

Flattens the array to a one-dimensional array, which may be necessary for certain types of analysis or modeling that require a single feature vector per sample.

parquet_df['Value'] = parquet_df['Value'].apply(flatten_list_of_arrays_and_divide): Applies the flatten_list_of_arrays_and_divide function to each element in the 'Value' column of parquet_df. This transforms the data within the 'Value' column to a processed and flattened form, suitable for input into a model.

This code is preparing the data for modeling by ensuring the structure of the DataFrame is correct, the data is properly indexed by time, and the values within it are preprocessed by flattening and normalizing, making them ready for subsequent machine learning tasks.

Part 2: **Load Sensor data**

Here is a detailed explanation of each step within the code:

input_df = pd.read_csv('/home/arman_abouali/Downloads/DWD/input.csv', sep=';'):

Loads a CSV file into a pandas DataFrame called input_df. The CSV file is separated by semicolons (;), as indicated by the sep parameter.

input_df['Zeit'] = input_df['Zeit'].replace("24:00:00", "00:00:00"):

Replaces any time entries in the 'Zeit' column that are "24:00:00" with "00:00:00". This is done because "24:00:00" is not a standard time representation and would cause errors in the datetime conversion.

input_df['Zeit'] = pd.to_datetime(input_df['Datum'] + ' ' + input_df['Zeit'], format='%d.%m.%Y %H:%M:%S'):

Combines the 'Datum' and 'Zeit' columns into a single datetime column. The 'Datum' column contains date information, and 'Zeit' contains time information. The pd.to_datetime function converts the combined string into a pandas datetime object according to the specified format.

input_df = input_df.drop('Datum', axis=1):

Drops the 'Datum' column from the DataFrame since its information is now included in the 'Zeit' column.

input_df = input_df.sort_values(by='Zeit', ascending=True).reset_index(drop=True):

Sorts the DataFrame based on the 'Zeit' column in ascending order and resets the index to reflect the new order, dropping the old index.

input_df['Sensor_Sum'] = input_df['GranetalsperreMin15Niederschlag'] + input_df['HahnenkleeMin15Niederschlag'] + input_df['Niederschlag_Gosequelle'] + input_df['Niederschlag_Abzuchtquelle']:

Creates a new column, 'Sensor_Sum', which is the sum of four other columns that represent different sensor readings for precipitation. This column aggregates the data to a single value, presumably to simplify further analysis.

idx = pd.date_range("2003-11-01 00:00:00", "2018-06-30 23:45:00", freq="15min"):

Generates a DatetimeIndex with a 15-minute frequency, spanning from November 1st, 2003, to June 30th, 2018.

input_df.reset_index(drop=True, inplace=True):

Resets the index of input_df again, which seems redundant since this was already done after sorting the DataFrame.

input_df = input_df.set_index(idx) and input_df = input_df.reindex(idx):

Sets the newly created idx as the index of input_df and then reindexes input_df to match idx exactly. This ensures that the DataFrame has a regular time index, which is crucial for time series analysis.

input_df = input_df.drop('Zeit', axis=1):

Drops the 'Zeit' column since its information is now redundant with the DataFrame's index.

In summary, this code loads sensor precipitation data, cleans it, creates a summed column of all the precipitation readings, and ensures that the DataFrame is properly indexed by time for time series analysis or modeling purposes.

Part 3: Build merge_df

This step is to merge two DataFrames, input_df and parquet_df, on their indices and then exports the merged DataFrame to a CSV file. Here's the step-by-step explanation:

merged_df = pd.merge(input_df, parquet_df, left_index=True, right_index=True, how='inner'):

pd.merge is a pandas function that combines two DataFrames based on a common key or index.

left_index=True and right_index=True tell the function to use the index of both input_df and parquet_df as the keys for merging.

how='inner' specifies that the merge should be an inner join, meaning that only the entries that have matching indices in both DataFrames will be included in the resulting merged_df. Rows from either DataFrame that do not have a corresponding index in the other DataFrame will be excluded.

merged_df.to_csv('merged_df.csv', index=False):

This command saves the merged DataFrame merged_df to a CSV file named 'merged_df.csv'.

index=False parameter indicates that the indices should not be written to the CSV file; only the data will be exported.

After executing this code, you will have a new CSV file that combines the data from both input_df and parquet_df where the indices matched, and this file can be used for further analysis or modeling.

Part 4: Model preparation

This step involves reshaping and scaling the data, as well as creating sequences that are used as inputs to the model. Here's a step-by-step explanation:

The image data (X_images) and target variable (y) are extracted from the merged_df DataFrame and converted into NumPy arrays, which are more efficient for numerical computations.

The target variable array y is reshaped to a two-dimensional array with one column to meet the requirements of the scaler object.

The shapes of X_images and y are printed, confirming that X_images is a two-dimensional array with 496800 rows and 64 columns, and y is a two-dimensional array with 496800 rows and 1 column.

A MinMaxScaler is used to normalize the target variable y. This scaler transforms each value in the array such that the minimum value of the transformed data is 0 and the maximum is 1.

The X_images array is reshaped to a two-dimensional array with a single feature column and then scaled similarly to y.

A function create_sequences is defined to transform the image data into sequences of a specified length (sequence_length), which is a common preprocessing step for time series data used in sequence prediction models.

X_images_sequence is created by applying the create_sequences function to the X_images array with sequence_length set to 96. This represents the number of time steps in each input sequence.

The corresponding labels y_sequence is aligned with the sequences in X_images_sequence. Since each sequence ends at an index that is sequence_length steps from the beginning of the sequence, the labels are sliced from y starting from the sequence_length to maintain this alignment.

Finally, the shapes of the newly created X_images_sequence and y_sequence are printed to confirm their dimensions. X_images_sequence is a three-dimensional array with 496704 samples, 96-timesteps, and 64 features per timestep representing flatten images. y_sequence is a two-dimensional array with 496704 samples and 1 column, representing the target variable for each sequence.

This processed and structured data can now be used to train a machine learning model, such as LSTM, for predicting the target variable based on the sequence of image data.

Part 5: Data Splitting

The code snippet is for splitting the dataset into training, validation, and test sets for the purpose of training and evaluating a machine learning model:

train_test_split from sklearn.model_selection is used to divide the dataset into two parts: a temporary set (X_temp, y_temp) and a test set (X_test, y_test). The test set is 10% (test_size=0.1) of the total data and is selected using a random state (seed) of 42 for reproducibility. The shuffle=False parameter is crucial for time series data to maintain the chronological order.

The temporary set is further split into training (X_train, y_train) and validation sets (X_val, y_val). The validation set is 20% (test_size=0.2) of the temporary set, and similarly, it uses a random state of 42 and shuffle=False to preserve the time series order.

The shapes of the training, validation, and test sets are printed to confirm the size of each dataset:

X_train: The training feature set with 357,626 sequences, each sequence having 96 time steps and 64 features per time step.

X_val: The validation feature set with 89,407 sequences, also with 96 time steps and 64 features per time step.

X_test: The test feature set with 49,671 sequences, with the same dimensions as the training and validation sets.

y_train, y_val, and y_test are the corresponding target variable sets for training, validation, and testing. Each has one column since the target variable is a single value per sequence.

These datasets are now ready to be used to train a model with X_train and y_train, tune the model's hyperparameters with X_val and y_val, and finally, evaluate the model's performance with X_test and y_test. This splitting strategy ensures that the model is trained and validated on different subsets of the data, which is important for assessing its generalization capability.

Part 6: LSTM model architecture

This particular model is designed for a time series prediction task using LSTM (Long Short-Term Memory) layers, which are well-suited to learn from sequences of data. Here's a breakdown of the model architecture and training process:

Model Architecture:

A Sequential model is initialized, indicating a linear stack of layers.

Three LSTM layers are added. The first two LSTM layers have return_sequences=True, which means they return the full sequence to the next layer instead of just the output of the last timestep. This is necessary when stacking LSTM layers.

A BatchNormalization layer is included, which normalizes the activations of the previous layer at each batch to improve the stability and performance of the neural network.

A Dense layer with 32 units and 'swish' activation function is used. 'Swish' is a smooth, non-linear activation function that tends to work better than 'relu' in deeper models.

The final output layer is a Dense layer with a single unit and 'swish' activation function, suitable for regression tasks (predicting a continuous value).

Model Compilation:

The model is compiled with the Mean Absolute Error (MAE) as the loss function, which is appropriate for regression problems.

The Adam optimizer is used with a learning rate of 1e-3, which is a reasonable starting point for many models.

Training:

Callbacks are set up for EarlyStopping to prevent overfitting by stopping training if the validation loss doesn't improve for 10 epochs.

ModelCheckpoint is used to save the model that achieves the best performance on the validation set.

The model is trained using the fit method for a maximum of 100 epochs with a batch size of 1024, using the training set X_train and y_train and the validation set X_val and y_val.

The history object captures the loss and metrics values during training.

Saving the Model:

The trained model is saved to a specified path in the Protocol Buffer (.pb) format using model.save(). This format saves the TensorFlow graph and variables and can be used for deployment or further analysis.

Output:

The path where the model is saved is printed out.

When using this code, ensure that TensorFlow is correctly installed in your environment and that you have the necessary computational resources, as training LSTMs can be quite demanding. Also, make sure the model_save_path points to a valid directory where you have written permissions.

Part 7: **Model prediction Result:**

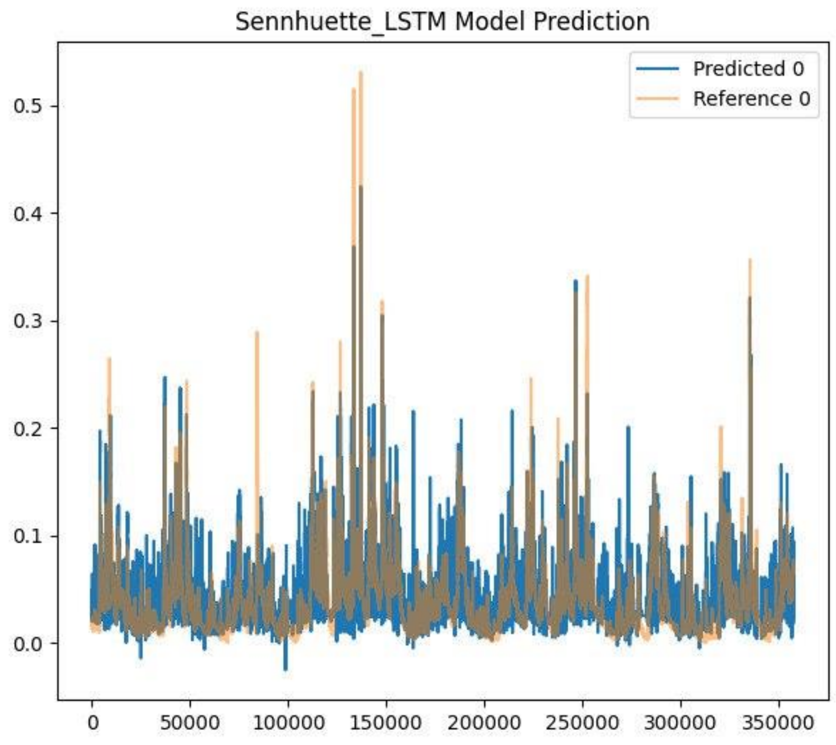
The subsequent three images show the model's predictions compared to the actual reference values over different ranges of data points.

The blue line represents the predicted values by the LSTM model, while the orange line represents the actual reference values.

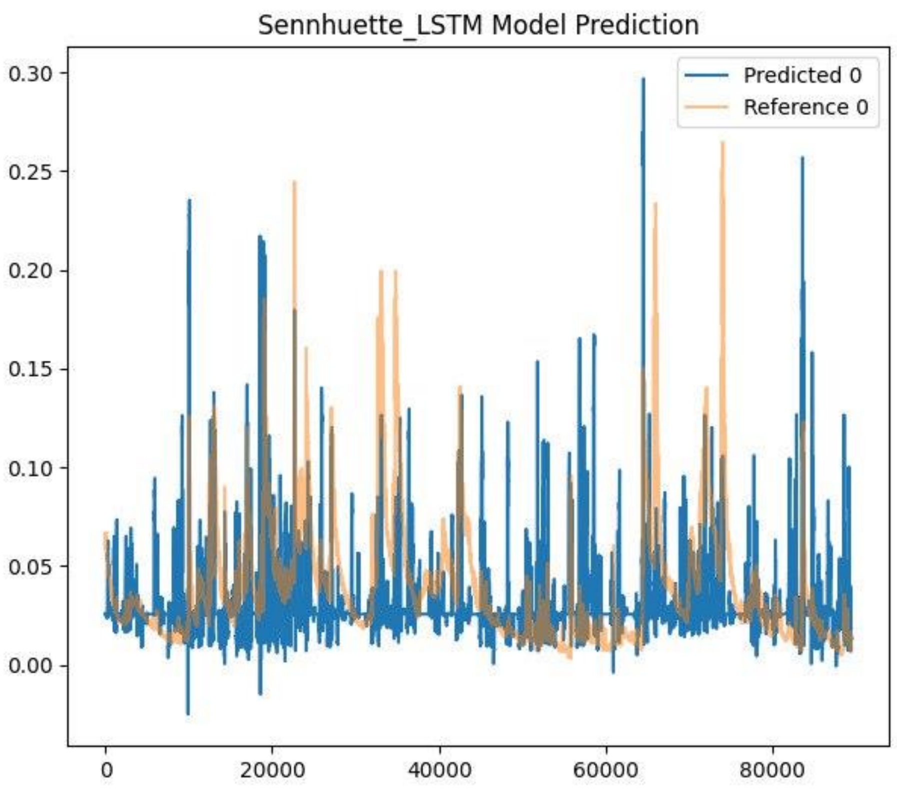
There are instances where the predicted values have spikes, which might correspond to some events or anomalies in the data. The model seems to capture the general trend but struggles with the magnitude of the changes, particularly the peaks.

The differences in scale between the three plots suggest that the model's predictions vary in accuracy across different segments of the test set. Some parts have predictions that are closer to the reference, while others have more significant deviations.

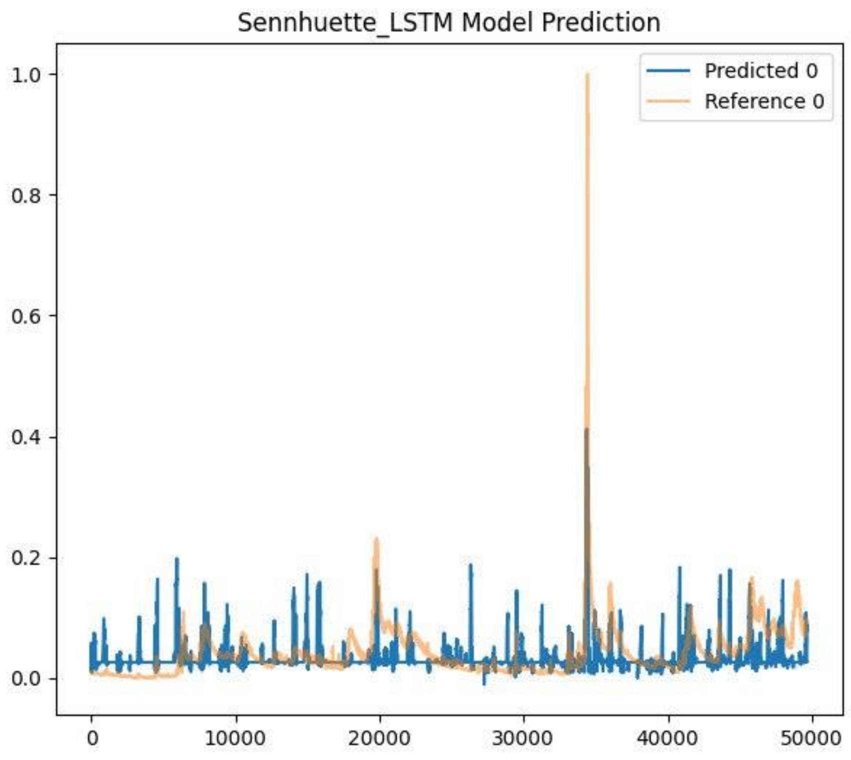
Train set Prediction plot:



Validation Set Prediction plot:



Test Set Prediction plot



:

References:

[1]: https://docs.wradlib.org/en/latest/notebooks/fileio/radolan.html

[2]:
https://opendata.dwd.de/climate_environment/CDC/help/RADOLAN/Unterstuetzungsdokumente/Unterstuetzungsdokument_Verwendung_von_RADOLAN_RADKLIM_Produkten_in_GIS_Software.pdf

[3]:
https://www.dwd.de/DE/leistungen/radolan/radolan_info/radolan_radvor_op_komposit_format_pdf.pdf?__blob=publicationFile&v=23