Prediciton of Lake Water Volume Changes

1.0 Introduction

1.1 Group information

Group Name: HydroDS (Group 15)

Group Member:

- 23054206 NG YU HENG
- 23005031 CHANG CAI TING
- 23005000 CHUA HUI MIN
- 23005227 KUEH PANG LANG
- 23051966 NG ZHI WEI
- 23004979 POH JING MIN

1.2 Problems and Solutions

Our project is about predicting the changes of water volume in 24 hours of a lake or a resevoir by the conditions of the weather.

Recently, Penang has been facing a critical issue where their water resevoirs are low on water. The water supply is only enough to last for another 30 days. These show that the people working at the water resevoirs fail to have an effective long term plan when dealing with climate changes to make sure that the water level in the dam is sufficient for supplying to the people. 10 years ago, Cameron Highlands experienced a tragedy where the water of the dam was released only when the water exceed its safety level due to heavy rain falls and caused a lighting flood. This show that the people working at the dam fail to take effective action when dealing with sudden changes of weather conditions.

From the situations above, we can see that weather is the major factor affecting the water volume in the water resevoirs or dam.

To solve the problems, we come up with two solutions:

- **Water volume changes prediction for unusual weather conditions**

- Water volume changes forecasting for climate changes

With the help of prediction and forecasting, it allow the responsible party to take early action to prevent the above issues to happen.

For this project, we will be focusing on **prediction for water volume changes in 24 hours**.

2.0 Data Mining

2.1 Lake dataset

After surfing through the Internet, we found that we were unable to find any historical data about the water level of the dam in Malaysia. The most we can get is the current data which is too little to train our model even if we start collecting it by April 2024. Hence, we decided to look for data out of Malaysia.

Luckily for us, we were able to find a website which contain the historical data regarding a few lakes located in Phoenix, AZ, USA. In order to extract those data out and store it in a .csv file, we have to perform Web Scrapping.

2.1.1 Inspecting the website (Lake dataset)

Before scrapping the data out from the website, we make some discovering about the website first and understand where the data is stored and how to extract the data for a period of time by performing the following code.

We need the help of some packages in order to perform web scrapping.

- BeautifulSoup
- requests

```
# Import required packages
from bs4 import BeautifulSoup
import requests
```

First, we try to get request from the website.

Source: https://www.watershedconnection.com/

```
# Website for data mining
url = "https://streamflow.watershedconnection.com/DWR?reportDate=2017-1-1"


# Get requests from website
requests.get(url)
```

```
<Response [200]>
```

It returns us a response 200. Response 200 indicates that our request to the website has succeeded.

Now, we try to get the HTML of the website.

```
# Get requests from website
page_lake_test = requests.get(url)


# Get the html of the website
soup_lake_test = BeautifulSoup(page_lake_test.text, 'html')


# Print the html of the website
print(soup_lake_test.prettify())
```

```
<!DOCTYPE html>
<html lang="en">
 <head>
  <meta charset="utf-8"/>
  <meta content="IE=7,IE=8,IE=9,IE=edge,chrome=1" http-equiv="X-UA-Compatible"/>
  <meta content="width=device-width,initial-scale=1" name="viewport"/>
  <title>
   Watershed Connection - Daily Water Report
  </title>
  <link href="/Content/css?v=uzEa1YlVEP0FnCSoKY3Zg-krSgS93kmbvAkpMj4xPeU1" rel="sty
  <script src="/bundles/scripts?v=WS1_ILBPVdLe47qNV5EOk3GMLVWFDAr9Q2fDZBpywQ41">
  </script>
  <script src="https://use.typekit.net/qli4olc.js">
  </script>
  <script>
   try {
            Typekit.load({ async: true });
        } catch (e) {
        }
  </script>
  <script src="/assets/js/AppMeasurement.js">
  </script>
  <script type="text/javascript">
   window.appRoot = '/';
       window.isMyWatershed = false;
  </script>
  <link href="/Content/dwr?v=J6MW3NT-vwfmDcL-qisrg9BYNsDBMv0F6wo4vSMbN2g1" rel="sty
  <script>
   $(function() {
            var reportDate = '01 01, 2017';
            $("#dpReportDate").datepicker("update", reportDate).on("changeDate",
```

```
                    function (e) {
                        var newDate = $(e.target).datepicker("getDate");
                        window.location = "DWR?reportDate=" + newDate.getFullYear() +
                    });
            });
            // Enable tooltips for bootstrap (https://getbootstrap.com/docs/3.3/javasc
            $(document).ready(function () {
                $('[data-toggle="tooltip"]').tooltip();
            });
        </script>
        <script src="/assets/js/AppMeasurement.js">
        </script>
       </head>
       <body>
        <div class="noPrint" id="page">
         <div class="container" id="header">
          <div class="row">
           <div class="col-xs-6 col-sm-push-6 text-right">
            <a href="//www.watershedconnection.com">
             <img alt="SRP Logo - Delivering more than power" class="hidden-xs pageLogo"
             <img alt="SRP Logo - Delivering more than power" class="visible-xs pageLogo
            </a>
           </div>
           <div class="col-xs-6 col-sm-pull-6">
            <a href="//www.watershedconnection.com">
             <img alt="WATERSHED CONNECTION" class="img-responsive hidden-xs pageTitle"
```

Well, there is quite a lot of code in the HTML. By inspecting the website, we know that the table containing the data is code under the tags 'table'. Let's have a search for the tags 'table'.

```
# Find all 'table' in the html (Searching for the tags containing the table of the data)
soup_lake_test.find_all('td')
```

```
[<td class="siteLabel">
 <a href="/?location=Roosevelt%20Lake%20(Roosevelt%20Dam)">Roosevelt Lake
(Roosevelt Dam)</a>
 </td>,
 <td class="values">38</td>,
 <td class="values values-shaded">
 2,088.60                        </td>,
 <td class="values values-shaded">
 618,171                        </td>,
 <td class="values values-shaded">
 62.40                        </td>,
 <td class="values values-shaded">
 1,013,361                        </td>,
 <td class="values">
 3,205                        </td>,
 <td class="values values-shaded">
 0.64                        </td>,
 <td class="siteLabel">
 <a href="/?location=Apache%20Lake%20(Horse%20Mesa%20Dam)">Apache Lake (Horse Mesa
Dam)</a>
 </td>,
 <td class="values">93</td>,
 <td class="values values-shaded">
 1,907.70                        </td>,
```

```
 <td class="values values-shaded">
 228,708                        </td>,
 <td class="values values-shaded">
 6.30                          </td>,
 <td class="values values-shaded">
 16,430                         </td>,
 <td class="values">
 512                          </td>,
 <td class="values values-shaded">
 1.49                          </td>,
 <td class="siteLabel">
 <a href="/?location=Canyon%20Lake%20(Mormon%20Flat%20Dam)">Canyon Lake (Mormon
Flat Dam)</a>
 </td>,
 <td class="values">95</td>,
 <td class="values values-shaded">
 1,657.17                         </td>,
 <td class="values values-shaded">
 54,720                         </td>,
 <td class="values values-shaded">
 3.33                          </td>,
 <td class="values values-shaded">
 3,132                          </td>,
 <td class="values">
 -260                          </td>,
 <td class="values values-shaded">
 0.96                          </td>,
 <td class="siteLabel">
 <a href="/?location=Saguaro%20Lake%20(Stewart%20Mountain%20Dam)">Saguaro Lake
(Stewart Mountain Dam)</a>
 </td>,
 <td class="values">92</td>,
 <td class="values values-shaded">
 1,524.37                         </td>.
```

There is the tags 'table' containing the data we want. However, there are also other tables in that website too. We select the table we want by adding a '[0]' at the end of the previous code.

```
# Find all 'table' in the html (Searching for the tags containing the table of the data)
soup_lake_test.find_all('table')[0]
```

```
<table class="sectionTable">
<tr class="tableShade">
<th></th>
<th></th>
<th>Current</th>
<th>Current</th>
<th>Remaining </th>
<th>Available </th>
<th>24 hr.</th>
<th>Rain</th>
</tr>
<tr class="tableShade">
<th></th>
<th>% Full</th>
<th>Elevation (ft)</th>
<th>Storage (af)</th>
<th>Elevation (ft)</th>
```

```
<th>Storage (af)</th>
<th>Change</th>
<th>(inches)</th>
</tr>
<tr>
<td class="siteLabel">
<a href="/?location=Roosevelt%20Lake%20(Roosevelt%20Dam)">Roosevelt Lake
(Roosevelt Dam)</a>
</td>
<td class="values">38</td>
<td class="values values-shaded">
2,088.60                          </td>
<td class="values values-shaded">
618,171                          </td>
<td class="values values-shaded">
62.40                          </td>
<td class="values values-shaded">
1,013,361                           </td>
<td class="values">
3,205                          </td>
<td class="values values-shaded">
0.64                          </td>
</tr>
<tr>
<td class="siteLabel">
<a href="/?location=Apache%20Lake%20(Horse%20Mesa%20Dam)">Apache Lake (Horse Mesa
Dam)</a>
</td>
<td class="values">93</td>
<td class="values values-shaded">
1,907.70                          </td>
<td class="values values-shaded">
228,708                          </td>
<td class="values values-shaded">
6.30                          </td>
<td class="values values-shaded">
16,430                          </td>
<td class="values">
512                          </td>
<td class="values values-shaded">
1.49                          </td>
```

The table we wanted has been selected. From the code, we able to see that the data we wanted is under the tags 'td'. Let's try to get the data out.

```
# Find all 'td' in the html (Searching for the data we wanted)
soup_lake_test.find_all('td')
```

```
[<td class="siteLabel">
 <a href="/?location=Roosevelt%20Lake%20(Roosevelt%20Dam)">Roosevelt Lake
(Roosevelt Dam)</a>
 </td>,
 <td class="values">38</td>,
 <td class="values values-shaded">
 2,088.60                          </td>,
 <td class="values values-shaded">
 618,171                          </td>,
 <td class="values values-shaded">
```

```
 62.40                        </td>,
 <td class="values values-shaded">
 1,013,361                        </td>,
 <td class="values">
 3,205                        </td>,
 <td class="values values-shaded">
 0.64                        </td>,
 <td class="siteLabel">
 <a href="/?location=Apache%20Lake%20(Horse%20Mesa%20Dam)">Apache Lake (Horse Mesa
Dam)</a>
 </td>,
 <td class="values">93</td>,
 <td class="values values-shaded">
 1,907.70                        </td>,
 <td class="values values-shaded">
 228,708                        </td>,
 <td class="values values-shaded">
 6.30                        </td>,
 <td class="values values-shaded">
 16,430                        </td>,
 <td class="values">
 512                        </td>,
 <td class="values values-shaded">
 1.49                        </td>,
 <td class="siteLabel">
 <a href="/?location=Canyon%20Lake%20(Mormon%20Flat%20Dam)">Canyon Lake (Mormon
Flat Dam)</a>
 </td>,
 <td class="values">95</td>,
 <td class="values values-shaded">
 1,657.17                        </td>,
 <td class="values values-shaded">
 54,720                        </td>,
 <td class="values values-shaded">
 3.33                        </td>,
 <td class="values values-shaded">
 3,132                        </td>,
 <td class="values">
 -260                        </td>,
 <td class="values values-shaded">
 0.96                        </td>,
 <td class="siteLabel">
 <a href="/?location=Saguaro%20Lake%20(Stewart%20Mountain%20Dam)">Saguaro Lake
(Stewart Mountain Dam)</a>
 </td>,
 <td class="values">92</td>,
 <td class="values values-shaded">
 1,524.37                        </td>
```

Let's pull the first data out from the tags to see whether the data is in the format that we wanted.

```
# Pulling text information from the website
soup_lake_test.find('td').text
```

```
'\nRoosevelt Lake (Roosevelt Dam)\n'
```

Well, almost there but it containing something that we don't want. Let's trim it off.

```
# Pulling text information from the website
soup_lake_test.find('td').text.strip()
```

```
'Roosevelt Lake (Roosevelt Dam)'
```

Nice, this is what we wanted.

The websites also containing some information of the weather for that respective day. From the inspection of the websites, the weather data is code under the 'b' tags.

```
# Find all 'b' in the html (Searching for the weather data)
soup_lake_test.find_all('b')
```

```
[<b> 66°/44 °(F)</b>, <b> 63°/54 °(F)</b>, <b>100 / 60 (%)</b>]
```

Each data is in the format highest/lowest. So, let us try to separate the first data.

```
# Separating the data and removing unwanted value from the data
temperature_test = soup_lake_test.find_all('b')[0]
temperature_list_test = temperature_test.text.strip().split("°")
temperature_list_test[1] = temperature_list_test[1].replace("/", "")
temperature_list_test
```

```
['66', '44 ', '(F)']
```

From the above list, we know that the data one is the first and the second items in the list.

Since we know where our data is located in the website, it's time to extract those data and store it in a dataframe.

```
# Selecting the table needed
table_lake_test = soup_lake_test.find_all('table')[0]
```

Before creating the dataframe, we need to set the feature names first. The feature names are under the tags 'th'. Let's have a look of it.

```
# Find all 'th' in the html (Searching for the feature names)
soup_lake_test.find_all('th')
```

```
[<th></th>,
 <th></th>,
 <th>Current</th>,
 <th>Current</th>,
```

```
        <th>Remaining </th>,
        <th>Available </th>,
        <th>24 hr.</th>,
        <th>Rain</th>,
        <th></th>,
        <th>% Full</th>,
        <th>Elevation (ft)</th>,
        <th>Storage (af)</th>,
        <th>Elevation (ft)</th>,
        <th>Storage (af)</th>,
        <th>Change</th>,
        <th>(inches)</th>,
        <th></th>,
        <th>
        <strong>Today</strong>
        </th>,
        <th>
        <strong>Normal</strong>
        </th>,
        <th>
        <strong>Yesterday Average<br/></strong>
        </th>,
        <th>
        <strong>%</strong>
        </th>,
        <th></th>,
        <th>Today</th>,
        <th>Normal</th>,
        <th>Yesterday Average</th>,
        <th>%</th>,
        <th></th>,
        <th>Today</th>,
        <th>Normal</th>,
        <th>Yesterday Average</th>,
        <th>%</th>]
```

That was an unexpected coding. We have no choice to set the feature names ourself.

```
# Setting the feature names manually into a list
feature_names_lake_test = ['Location', 'Full (%)', 'Current Elevation(ft)',
                      'Current Storage (af)', 'Remaining Elevation (ft)',
                      'Available Storage (af)', '24 hr. Change',
                      'Rain (inches)']
feature_names_lake_test
```

```
['Location',
 'Full (%)',
 'Current Elevation(ft)',
 'Current Storage (af)',
 'Remaining Elevation (ft)',
 'Available Storage (af)',
 '24 hr. Change',
 'Rain (inches)']
```

This is better. Now we will create a dataframe and set the feature names into it.

```
import pandas as pd


lake_test_df = pd.DataFrame(columns = feature_names_lake_test)
lake_test_df
```

	Location	Full (%)	Current Elevation(ft)	Current Storage	Remaining Elevation	Available Storage	24 hr. Change	Rain (inches)

Looks good up until here.

Now it is time to abtract the data we wanted and storing it into the dataframe.

```
column_data_lake_test = table_lake_test.find_all('tr')


for row in column_data_lake_test[2:]:
  row_data = row.find_all('td')
  individual_row_data = [data.text.strip() for data in row_data]

  length = len(lake_test_df)
  lake_test_df.loc[length] = individual_row_data

lake_test_df
```

	Location	Full (%)	Current Elevation(ft)	Current Storage (af)	Remaining Elevation (ft)	Available Storage (af)	24 hr. Change	Rain (inches)
0	Roosevelt Lake (Roosevelt Dam)	38	2,088.60	618,171	62.40	1,013,361	3,205	0.64
1	Apache Lake (Horse Mesa Dam)	93	1,907.70	228,708	6.30	16,430	512	1.49
2	Canyon Lake (Mormon Flat Dam)	95	1,657.17	54,720	3.33	3,132	-260	0.96
	Saguaro Lake							

Seems like the web scrapping works for this website as the data we wanted is successfully extract from the website into the dataframe. Now, we will be moving to the next step which is extracting data for a range of time.

2.1.2 Extracting data for a range of time (Lake dataset)

Setting up the date range for the data we wanted to extract.

```
# Import package needed for convertion between text and date
from datetime import datetime, timedelta


# Set Date range
date_start_text = "2013-1-1"
date_end_text = "2023-1-1"
freq_text = "1"
```

The range of our data gathering will be daily for a period of 10 years.

The convertion from text to date is necessary to be put as the range of the while loop.

```
# Convert text to date
date_start_inc = date_start_text
date_start_inc_date = datetime.strptime(date_start_inc, "%Y-%m-%d")
date_end_date = datetime.strptime(date_end_text, "%Y-%m-%d")
freq_obj = int(freq_text)

date_start_inc_date = date_start_inc_date.date()
date_end_date = date_end_date.date()
```

Setting up the feature names needed for the dataframe.

```
# Setting up features name
feature_names_lake = [ 'Location', 'Full_%', 'Current_Elevation_ft',
                      'Current_Storage_af', 'Remaining_Elevation_ft',
                      'Available_Storage_af', '24hr.Change',
                      'Rain_inches', 'Date', 'Highest_temperature',
                       'Lowest_temperature','Highest_humidity',
                       'Lowest_humidity']
feature_names_lake
```

```
['Location',
 'Full_%',
 'Current_Elevation_ft',
 'Current_Storage_af',
 'Remaining_Elevation_ft',
 'Available_Storage_af',
 '24hr.Change',
 'Rain_inches',
 'Date',
 'Highest_temperature',
 'Lowest_temperature',
```

```
        'Highest_humidity',
        'Lowest_humidity']
```

There will be a total of 13 features for this dataset

```
# Setting up the dataframe to store data later with the feature names we set previously
lake_df = pd.DataFrame(columns = feature_names_lake)
```

Our dataframe is ready to store data gather from the website

```
# Using the while loop to gather data for a period of time
while date_start_inc_date <= date_end_date:
  # Date will be automatically insert to the back of the url to access data for that part
  url = 'https://streamflow.watershedconnection.com/DWR?reportDate='+ date_start_inc_date
  page_lake = requests.get(url)
  soup_lake = BeautifulSoup(page_lake.text, 'html')
  table_lake = soup_lake.find_all('table')[0]
  column_data_lake = table_lake.find_all('tr')
  for row_lake in column_data_lake[2:]:
    row_data_lake = row_lake.find_all('td')
    individual_row_data_lake = [data.text.strip() for data in row_data_lake]
    individual_row_data_lake.append(date_start_inc_date.strftime("%Y-%m-%d"))
    # Triming the temperature value and adding it into the list
    temperature = soup_lake.find_all('b')[0]
    temperature_list = temperature.text.strip().split("°")
    temperature_list[1] = temperature_list[1].replace("/", "")
    individual_row_data_lake.append(temperature_list[0])
    individual_row_data_lake.append(temperature_list[1])
    # Triming the humidity value and adding it into the list
    humidity = soup_lake.find_all('b')[2]
    humidity_list = humidity.text.strip().split()
    individual_row_data_lake.append(humidity_list[0])
    individual_row_data_lake.append(humidity_list[2])
    # Storing the list of data into the dataframe
    length = len(lake_df)
    lake_df.loc[length] = individual_row_data_lake

  date_start_inc_date = date_start_inc_date + timedelta(days = freq_obj)
```

Having a look at the dataframe.

```
# Print out a dataframe
lake_df
```

	Location	Full_%	Current_Elevation_ft	Current_Storage_af	Remaining_Elevat
0	Roosevelt Lake (Roosevelt Dam)	42	2,093.87	687,025	
1	Apache Lake (Horse Mesa Dam)	93	1,907.07	227,100	
2	Canyon Lake (Mormon Flat Dam)	96	1,658.14	55,626	
3	Saguaro Lake (Stewart Mountain Dam)	94	1,525.46	65,376	
4	Total Salt system	51		1,035,127	
...	...	...	...	...	
36525	Horseshoe Lake (Horseshoe Dam)	12	1,975.14	13,218	
36526	Bartlett Lake (Bartlett Dam)	55	1,763.56	98,311	
36527	Total Verde system	39		111,529	
36528	Total Reservoir System	65		1,498,761	
36529	Total system year ago	69			

36530 rows × 13 columns

From the table above, the data we gather has correctly store in their respective column. A total of 36530 rows of data gather for 10 years period

```
lake_df.to_csv(r'/content/sample_data/lake_dataset_10years.csv', index = False)
```

We save the data gather into a csv file.

After inspecting the dataset, we found out the weather information gather from that website is not precise enough. Hence, we decided to gather the information of the weather from the nearest weather station to the area where the lakes are located from other website.

2.2 Weather dataset

First we need to see which weather station is the nearest to that area. Only the weather station at the airport in that area is storing the historical weather data.

The airport available there is as follow:

- Phoenix Sky Harbor International Airport
- Phoenix-Mesa Gateway Airport
- Falcon Field Airport
- Scottsdale Airport
- Phoenix Deer Valley Airport
- Glendale Municipal Airport
- Phoenix-Goodyear Airport

We have chosen Falcon Field Airport since it is nearest to that area.

We now gather the data about the weather information of that area from this website.

Source: https://www.visualcrossing.com/

2.2.1 Inspecting the website (Weather dataset)

After taking a tour through this website, it's provide API that allow user to get the data from their website easily. However, for free user, a total of 1000 records are allowed for a day. This mean that we need a few day to gather all the data we wanted.

Since API service is provided, it save up a lot of our works to inspect the website.

2.2.2 Extracting data for a range of time (Weather dataset)

As same as previous, we first need to indicate the range of date of the data we wanted to extract. Since we are only able to extract limited data per day, in this notebook, we will only be extracting data for a period of one year.

```
# Set Date range
date_start_text = "2013-1-1"
date_end_text = "2014-1-1"
freq_text = "1"
```

Converting the date from string into date

```
# Convert text to date
date_start_inc = date_start_text
date_start_inc_date = datetime.strptime(date_start_inc, "%Y-%m-%d")
date_end_date = datetime.strptime(date_end_text, "%Y-%m-%d")
freq_obj = int(freq_text)

date_start_inc_date = date_start_inc_date.date()
date_end_date = date_end_date.date()
```

We now start to extract the data we wanted using the API. We will be getting weather information for a day for each query we make. Hence, we need to create a list to store all the dataframe we get so that we can combine it into one later.

```
# Creating the list to store the .csv file
list_of_csv = []
# Creating the while loop
while date_start_inc_date <= date_end_date:
  url = 'https://weather.visualcrossing.com/VisualCrossingWebServices/rest/services/timel
  df = pd.read_csv(url, index_col=None, header = 0)
  date_start_inc_date = date_start_inc_date + timedelta(days = freq_obj)
  # Adding the dataframe into the list
  list_of_csv.append(df)
```

We proceed with the combination of the dataframe

```
# Combining all the dataframe in the list into one
weather_2013_df = pd.concat(list_of_csv, axis= 0,ignore_index=True)
```

Let's have a glance through the dataframe we combined

```
# Print the dataset
weather_2013_df
```

	name	datetime	tempmax	tempmin	temp	feelslikemax	feelslikemin	feelslike
0	4800 E Falcon Dr, Mesa, AZ 85215, United States	2013-01-01	12.1	1.0	6.7	12.1	-2.4	5.6
1	4800 E Falcon Dr, Mesa, AZ 85215, United States	2013-01-02	16.4	2.6	9.5	16.4	0.0	8.6
2	4800 E Falcon Dr, Mesa, AZ 85215, United States	2013-01-03	16.4	6.1	11.0	16.4	2.9	10.3
3	4800 E Falcon Dr, Mesa, AZ 85215, United States	2013-01-04	14.8	4.5	9.5	14.8	1.2	8.9
4	4800 E Falcon Dr, Mesa, AZ 85215, United States	2013-01-05	16.4	2.8	9.0	16.4	-1.7	8.5
...	...	...	...	...	...	...	...	...
361	4800 E Falcon Dr, Mesa, AZ 85215,	2013-12-28	18.0	7.4	12.2	18.0	6.9	12.2

Everything looks good. Time to save it into a .csv file for later use.

```
# Saving the dataframe into a .csv file
weather_2013_df.to_csv(r'/content/sample_data/weather_dataset_2013.csv', index = False)
```

Now, we have the weather dataset for the year 2013. The dataset for the remaining years will be done using another notebook by using the same code as above.

Since all of the dataset we wanted are ready, it's time to proceed to the next section,which is Data Preprocessing.

3.0 Data cleaning and preprocessing

Data cleaning is the initial phase of refining dataset, making it readable and usable with techniques like removing and usable with techniques such as handling missing values, changing data types, removing duplicates etc.

Data preprocessing take place to refine data and scaling with more advanced techniques such as encoding categorical variables, handling outliers in order to achieve a better results.

3.1 Data loading

Import lake dataset

```
import pandas as pd

# Importing lake dataset
lake_10years_df=pd.read_csv("/content/lake_dataset_10years.csv")
```

Import weather dataset from year 2013 to 2022

```
# Importing dataset

# Each dataframe have been predefined to prevent conflict between the dataframe variable

# Importing weather dataset

weather_2013_df = pd.read_csv("/content/weather_dataset_cv_2013.csv")
weather_2014_df = pd.read_csv("/content/weather_dataset_cv_2014.csv")
weather_2015_df = pd.read_csv("/content/weather_dataset_cv_2015.csv")
weather_2016_df = pd.read_csv("/content/weather_dataset_cv_2016.csv")
weather_2017_df = pd.read_csv("/content/weather_dataset_cv_2017.csv")
```

```
weather_2018_df = pd.read_csv("/content/weather_dataset_cv_2018.csv")
weather_2019_df = pd.read_csv("/content/weather_dataset_cv_2019.csv")
weather_2020_df = pd.read_csv("/content/weather_dataset_cv_2020.csv")
weather_2021_df = pd.read_csv("/content/weather_dataset_cv_2021.csv")
weather_2022_df = pd.read_csv("/content/weather_dataset_cv_2022.csv")
```

To merge all the weather datasets into one dataframe, we created a list called list_of_csv to store DataFrame objects. The list contains DataFrame variables named weather_2013_df, weather_2014_df, and so on up to weather_2022_df, each presumably holding weather data for a specific year.

```
# Creating the list to store the .csv file
list_of_csv = [weather_2013_df,weather_2014_df,weather_2015_df,weather_2016_df,
               weather_2017_df,weather_2018_df,weather_2019_df,weather_2020_df,
               weather_2021_df,weather_2022_df]
```

We combines all the DataFrames stored in the list_of_csv into a single DataFrame named weather_10years_df.

```
# Combining all the dataframe in the list into one
weather_10years_df = pd.concat(list_of_csv, axis= 0,ignore_index=True)
```

Let have a look for our merged dataset

```
# Print the dataset
weather_10years_df
```

	name	datetime	tempmax	tempmin	temp	feelslikemax	feelslikemin	feelslike
0	4800 E Falcon Dr, Mesa, AZ 85215, United States	2013-01-01	12.1	1.0	6.7	12.1	-2.4	5.6
1	4800 E Falcon Dr, Mesa, AZ 85215, United States	2013-01-02	16.4	2.6	9.5	16.4	0.0	8.6
2	4800 E Falcon Dr, Mesa, AZ 85215, United States	2013-01-03	16.4	6.1	11.0	16.4	2.9	10.3
3	4800 E Falcon Dr, Mesa, AZ 85215, United States	2013-01-04	14.8	4.5	9.5	14.8	1.2	8.9
4	4800 E Falcon Dr, Mesa, AZ 85215, United States	2013-01-05	16.4	2.8	9.0	16.4	-1.7	8.5
...	...	...	...	...	...	...	...	...
3657	4800 E Falcon Dr, Mesa, AZ 85215,	2022-12-28	16.1	10.7	12.5	16.1	10.7	12.5

	United States							
3658	4800 E Falcon Dr, Mesa, AZ 85215, United States	2022-12-29	16.6	8.8	12.9	16.6	7.8	12.8
3659	4800 E Falcon Dr, Mesa, AZ 85215, United States	2022-12-30	14.4	11.2	13.0	14.4	11.2	13.0
3660	4800 E Falcon Dr, Mesa, AZ 85215, United States	2022-12-31	18.8	10.7	14.2	18.8	10.7	14.2
3661	4800 E Falcon Dr, Mesa, AZ 85215, United States	2023-01-01	16.1	8.9	12.3	16.1	6.1	11.9

3662 rows × 33 columns

Optional

Dowload the combine csv file to your local repository

```
# Save the combined DataFrame as a CSV file
weather_10years_df.to_csv('Weather_10years.csv', index=False)
```

```
from google.colab import files

files.download('Weather_10years.csv')
```

3.2 Data cleaning

3.2.1 Overview the dataset

Lake dataset

```
# Checking data
print("\nInformation of the dataset")
print(lake_10years_df.info())
```

```
Information of the dataset
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 36530 entries, 0 to 36529
Data columns (total 13 columns):
 #   Column                 Non-Null Count  Dtype 
---  ------                 --------------  ----- 
 0   Location               36530 non-null  object
 1   Full_%                 36530 non-null  object
 2   Current_Elevation_ft   21918 non-null  object
 3   Current_Storage_af     32877 non-null  object
 4   Remaining_Elevation_ft 21918 non-null  object
 5   Available_Storage_af   32877 non-null  object
 6   24hr.Change            32877 non-null  object
 7   Rain_inches            21918 non-null  object
 8   Date                   36530 non-null  object
 9   Highest_temperature    36530 non-null  object
 10  Lowest_temperature     36530 non-null  object
 11  Highest_humidity       36530 non-null  object
 12  Lowest_humidity        36530 non-null  object
dtypes: object(13)
memory usage: 3.6+ MB
```

```
None
```

```
print("\nNumber of unique value for each column")
print(lake_10years_df.nunique())
```

```
Number of unique value for each column
Location                    10
Full_%                     103
Current_Elevation_ft      9262
Current_Storage_af       24191
Remaining_Elevation_ft     5819
Available_Storage_af     23796
24hr.Change               6151
Rain_inches                201
Date                      3653
Highest_temperature         44
Lowest_temperature          43
Highest_humidity            89
Lowest_humidity             78
dtype: int64
```

```
print("\nDetermining the data types")
print(lake_10years_df.dtypes)
```

```
Determining the data types
Location                 object
Full_%                   object
Current_Elevation_ft     object
Current_Storage_af       object
Remaining_Elevation_ft   object
Available_Storage_af     object
24hr.Change              object
Rain_inches              object
Date                     object
Highest_temperature      object
Lowest_temperature       object
Highest_humidity         object
Lowest_humidity          object
dtype: object
```

All of the data extracted from the website are stored as String rather then integer. Convertion should be made as most of the data such as current elevation, current storage etc. should be integer.

```
print(f'The dateset has {lake_10years_df.shape[0]} rows and {lake_10years_df.shape[1]} co
```

```
The dateset has 36530 rows and 13 columns.
```

From the information above, we know that the dataset contains 36530 rows of data and 13 features.

Identify the column

```
print('Columns of dataset\n_____________________________________________________________________________')
# Print the columns of the dataset
print('Lake Dataset:', lake_10years_df.columns)
```

```
Columns of dataset
_____________________________________________________________________________
Lake Dataset: Index(['Location', 'Full_%', 'Current_Elevation_ft', 'Current_Storage_a
       'Remaining_Elevation_ft', 'Available_Storage_af', '24hr.Change',
       'Rain_inches', 'Date', 'Highest_temperature', 'Lowest_temperature',
       'Highest_humidity', 'Lowest_humidity'],
      dtype='object')
```

The above are the list of features name for the lake dataset.

Weather dataset

```
#Checking data
print("\nPrint information of the dataset: ")
print(weather_10years_df.info())
```

```
Print information of the dataset:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3662 entries, 0 to 3661
Data columns (total 33 columns):
 #   Column            Non-Null Count  Dtype
---  ------            --------------  -----
 0   name              3662 non-null   object
 1   datetime          3662 non-null   object
 2   tempmax           3662 non-null   float64
 3   tempmin           3662 non-null   float64
 4   temp              3662 non-null   float64
 5   feelslikemax      3662 non-null   float64
 6   feelslikemin      3662 non-null   float64
 7   feelslike         3662 non-null   float64
 8   dew               3662 non-null   float64
 9   humidity          3662 non-null   float64
 10  precip            3662 non-null   float64
 11  precipprob        3662 non-null   int64
 12  precipcover       3662 non-null   float64
 13  preciptype        474 non-null    object
 14  snow              2928 non-null   float64
 15  snowdepth         2930 non-null   float64
 16  windgust          3191 non-null   float64
 17  windspeed         3662 non-null   float64
 18  winddir           3662 non-null   float64
 19  sealevelpressure  3662 non-null   float64
 20  cloudcover        3662 non-null   float64
 21  visibility        3662 non-null   float64
 22  solarradiation    3656 non-null   float64
 23  solarenergy       3656 non-null   float64
```

```
 24  uvindex           3656 non-null   float64
 25  severerisk        357 non-null    float64
 26  sunrise           3662 non-null   object
 27  sunset            3662 non-null   object
 28  moonphase         3662 non-null   float64
 29  conditions        3662 non-null   object
 30  description       3662 non-null   object
 31  icon              3662 non-null   object
 32  stations          3662 non-null   object
dtypes: float64(23), int64(1), object(9)
memory usage: 944.2+ KB
None
```

```
print("\nNumber of unique value for each column")
print(weather_10years_df.nunique())
```

```
Number of unique value for each column
name                   1
datetime            3653
tempmax              329
tempmin              314
temp                 348
feelslikemax         347
feelslikemin         364
feelslike            352
dew                  363
humidity             678
precip               366
precipprob             2
precipcover           22
preciptype             2
snow                   1
snowdepth              2
windgust             351
windspeed            260
winddir             2169
sealevelpressure     247
cloudcover           777
visibility           252
solarradiation      2079
solarenergy          305
uvindex               10
severerisk             6
sunrise             3653
sunset              3653
moonphase             96
conditions             7
description           37
icon                   4
stations             107
dtype: int64
```

```
print("\nDetermining the data types")
print(weather_10years_df.dtypes)
```

```
Determining the data types
```

```
name                object
datetime            object
tempmax            float64
tempmin            float64
temp               float64
feelslikemax       float64
feelslikemin       float64
feelslike          float64
dew                float64
humidity           float64
precip             float64
precipprob           int64
precipcover        float64
preciptype          object
snow               float64
snowdepth          float64
windgust           float64
windspeed          float64
winddir            float64
sealevelpressure   float64
cloudcover         float64
visibility         float64
solarradiation     float64
solarenergy        float64
uvindex            float64
severerisk         float64
sunrise             object
sunset              object
moonphase          float64
conditions          object
description         object
icon                object
stations            object
dtype: object
```

There are some categorical data in the dataset that need to be encode later in the data preprocessing process.

```
print(f'The dateset has {weather_10years_df.shape[0]} rows and {weather_10years_df.shape[
```

```
The dateset has 3662 rows and 33 columns.
```

From the information above, we know that the dataset contains 3662 rows of data and 33 features.

Identify the column

```
print('Columns of dataset\n_____________________________________________________________________________
# Print the columns of the dataset
print('Weather Dataset: ', weather_10years_df.columns)
```

```
Columns of dataset
_____________________________________________________________________________
Weather Dataset:  Index(['name', 'datetime', 'tempmax', 'tempmin', 'temp', 'feelslike
```

```
       'feelslikemin', 'feelslike', 'dew', 'humidity', 'precip', 'precipprob',
       'precipcover', 'preciptype', 'snow', 'snowdepth', 'windgust',
       'windspeed', 'winddir', 'sealevelpressure', 'cloudcover', 'visibility',
       'solarradiation', 'solarenergy', 'uvindex', 'severerisk', 'sunrise',
       'sunset', 'moonphase', 'conditions', 'description', 'icon', 'stations'],
      dtype='object')
```

The above are the list of features name of the weather dataset.

3.2.2 Lake data modifiying

We wanted to use the the previous day storage capacity as a feature to predict the change of volume of the lake in 24 hours. A small modifying to the dataset is needed. We will be creating a new column with the title 'Previous_Storage_af'.

We will use the data from the features, 'Current_Storage_af' and shift the data 10 rows downward. Now we will have a feature which contains the volume of the storage of the previous day.

```
# Having a look at the lake dataset
lake_10years_df
```

	Location	Full_%	Current_Elevation_ft	Current_Storage_af	Remaining_Elevat
0	Roosevelt Lake (Roosevelt Dam)	42	2,093.87	687,025	
1	Apache Lake (Horse Mesa Dam)	93	1,907.07	227,100	
2	Canyon Lake (Mormon Flat Dam)	96	1,658.14	55,626	
3	Saguaro Lake (Stewart Mountain Dam)	94	1,525.46	65,376	
4	Total Salt system	51	NaN	1,035,127	
...	...	...	...	...	
36525	Horseshoe Lake (Horseshoe Dam)	12	1,975.14	13,218	
36526	Bartlett Lake (Bartlett Dam)	55	1,763.56	98,311	
36527	Total Verde system	39	NaN	111,529	
36528	Total Reservoir System	65	NaN	1,498,761	
36529	Total system year ago	69	NaN	NaN	

36530 rows × 13 columns

```
# Creating a new column named 'Previous_Storage_af'
# Using the data from column named 'Current_Storage_af', shifting it 10 rows downward, an
lake_10years_df.insert(3, "Previous_Storage_af", lake_10years_df["Current_Storage_af"].sh
```

```
# Printing the lake dataset again to see the added column
lake_10years_df.head(20)
```

	Location	Full_%	Current_Elevation_ft	Previous_Storage_af	Current_Storage_af
0	Roosevelt Lake (Roosevelt Dam)	42	2,093.87	None	687,025
1	Apache Lake (Horse Mesa Dam)	93	1,907.07	None	227,100
2	Canyon Lake (Mormon Flat Dam)	96	1,658.14	None	55,626
3	Saguaro Lake (Stewart Mountain Dam)	94	1,525.46	None	65,376
4	Total Salt system	51	NaN	None	1,035,127
5	Horseshoe Lake (Horseshoe Dam)	0	1,952.01	None	141
6	Bartlett Lake (Bartlett Dam)	57	1,765.26	None	101,536
7	Total Verde system	35	NaN	None	101,677
8	Total Reservoir System	49	NaN	None	1,136,804
9	Total system year ago	66	NaN	None	NaN
10	Roosevelt Lake (Roosevelt Dam)	42	2,093.90	687,025	687,349
11	Apache Lake (Horse Mesa Dam)	92	1,906.87	227,100	226,591
12	Canyon Lake (Mormon Flat Dam)	96	1,658.18	55,626	55,664

13	Saguaro Lake (Stewart Mountain Dam)	94	1,525.66	65,376	65,619
14	Total Salt system	51	NaN	1,035,127	1,035,223
15	Horseshoe Lake (Horseshoe Dam)	0	1,952.21	141	165
16	Bartlett Lake (Bartlett Dam)	57	1,765.31	101,536	101,631
17	Total Verde system	35	NaN	101,677	101,796
18	Total Reservoir System	49	NaN	1,136,804	1,137,019
19	Total system year ago	66	NaN	NaN	NaN

A new feature is created by modifying one of the existing features. However, null values existed in this new features because of the shifting of data. It will be remove later in the data preprocessing process.

3.2.3 Rename the date column

We choose to rename the 'Date' column to 'datetime' in the lake DataFrame since we noticed that the weather dataset uses 'datetime' as the column name for the corresponding information. This adjustment ensures consistency across datasets and facilitates seamless data integration.

```
# Renaming the 'Date' column to 'datetime'
lake_10years_df.rename(columns={'Date': 'datetime'}, inplace=True)

# Check the updated column names
print("\nFeatures of the dataset ")
print(lake_10years_df.columns)
```

```
Features of the dataset
Index(['Location', 'Full_%', 'Current_Elevation_ft', 'Previous_Storage_af',
       'Current_Storage_af', 'Remaining_Elevation_ft', 'Available_Storage_af',
```

```
       '24hr.Change', 'Rain_inches', 'datetime', 'Highest_temperature',
       'Lowest_temperature', 'Highest_humidity', 'Lowest_humidity'],
      dtype='object')
```

3.2.4 Choosing a lake location and removing the others

From the lake datasets, it contains a few lake locations that can be selected for prediction. However, including all the locations will affect the accuracy of the prediction.

Hence, a discussion has been made and we decided to select one location only which is Roosevelt Lake (Roosevelt Dam) for our prediction.

```
# Removing unwanted locations
lake_10years_df = lake_10years_df.drop(lake_10years_df[lake_10years_df['Location'] == 'Ap
lake_10years_df = lake_10years_df.drop(lake_10years_df[lake_10years_df['Location'] == 'Ca
lake_10years_df = lake_10years_df.drop(lake_10years_df[lake_10years_df['Location'] == 'Sa
lake_10years_df = lake_10years_df.drop(lake_10years_df[lake_10years_df['Location'] == 'Ho
lake_10years_df = lake_10years_df.drop(lake_10years_df[lake_10years_df['Location'] == 'Ba


# Having a look at the lake dataset
lake_10years_df
```

	Location	Full_%	Current_Elevation_ft	Previous_Storage_af	Current_Storage_
0	Roosevelt Lake (Roosevelt Dam)	42	2,093.87	None	687,0
4	Total Salt system	51	NaN	None	1,035,1
7	Total Verde system	35	NaN	None	101,6
8	Total Reservoir System	49	NaN	None	1,136,8
9	Total system year ago	66	NaN	None	N
...	...	...	...	...	
36520	Roosevelt Lake (Roosevelt Dam)	64	2,119.47	1,043,777	1,046,2
36524	Total Salt system	69	NaN	1,384,655	1,387,2
36527	Total Verde system	39	NaN	110,346	111,5
36528	Total Reservoir System	65	NaN	1,495,001	1,498,7
36529	Total system year ago	69	NaN	NaN	N

18265 rows × 14 columns

3.2.5 Replace Missing Values for selected features

To avoid losing too much potential data when removing null value by using the command dropna(), we decided to replace the null value with suitable value for some features.

```
print("\nFind missing value of each column")
print(lake_10years_df.isna().sum())
```

```
print("\nFind missing value of each column")
print(weather_10years_df.isna().sum())
```

```
Find missing value of each column
Location                     0
Full_%                       0
Current_Elevation_ft     14612
Previous_Storage_af       3657
Current_Storage_af        3653
Remaining_Elevation_ft   14612
Available_Storage_af      3653
24hr.Change               3653
Rain_inches              14612
datetime                     0
Highest_temperature          0
Lowest_temperature           0
Highest_humidity             0
Lowest_humidity              0
dtype: int64

Find missing value of each column
name                  0
datetime              0
tempmax               0
tempmin               0
temp                  0
feelslikemax          0
feelslikemin          0
feelslike             0
dew                   0
humidity              0
precip                0
precipprob            0
precipcover           0
preciptype         3188
snow                734
snowdepth           732
windgust            471
windspeed             0
winddir               0
sealevelpressure      0
cloudcover            0
visibility            0
solarradiation        6
solarenergy           6
uvindex               6
severerisk         3305
sunrise               0
sunset                0
moonphase             0
conditions            0
description           0
icon                  0
stations              0
dtype: int64
```

```
# Replace missing values in the 'precitype' column with 'none'
weather_10years_df['preciptype'].fillna('none', inplace=True)
```

```
# Replace missing values in the 'snow' column with mean
weather_10years_df['snow'].fillna(weather_10years_df['snow'].mean(), inplace=True)

# Replace missing values in the 'snowdepth' column with mean
weather_10years_df['snowdepth'].fillna(weather_10years_df['snowdepth'].mean(), inplace=Tr

# Replace missing values in the 'windgust' column with 0
weather_10years_df['windgust'].fillna(0, inplace=True)

# Replace missing values in the 'solarradiation' column with mean
weather_10years_df['solarradiation'].fillna(weather_10years_df['solarradiation'].mean(),

# Replace missing values in the 'solarenergy' column with mean
weather_10years_df['solarenergy'].fillna(weather_10years_df['solarenergy'].mean(), inplac

# Replace missing values in the 'uvindex' column with mean
weather_10years_df['uvindex'].fillna(weather_10years_df['uvindex'].mean(), inplace=True)

# Replace missing values in the 'severerisk' column with 0
weather_10years_df['severerisk'].fillna(0, inplace=True)



# Display the first few rows to verify the changes
print(weather_10years_df.head())
```

```
                                          name    datetime  tempmax  \
0  4800 E Falcon Dr, Mesa, AZ 85215, United States  2013-01-01     12.1
1  4800 E Falcon Dr, Mesa, AZ 85215, United States  2013-01-02     16.4
2  4800 E Falcon Dr, Mesa, AZ 85215, United States  2013-01-03     16.4
3  4800 E Falcon Dr, Mesa, AZ 85215, United States  2013-01-04     14.8
4  4800 E Falcon Dr, Mesa, AZ 85215, United States  2013-01-05     16.4

   tempmin  temp  feelslikemax  feelslikemin  feelslike  dew  humidity  ...  \
0      1.0   6.7          12.1          -2.4        5.6 -2.3      56.2  ...
1      2.6   9.5          16.4           0.0        8.6 -7.3      31.8  ...
2      6.1  11.0          16.4           2.9       10.3 -9.1      25.6  ...
3      4.5   9.5          14.8           1.2        8.9 -7.0      31.8  ...
4      2.8   9.0          16.4          -1.7        8.5 -5.3      39.0  ...

   solarenergy  uvindex  severerisk              sunrise               sunset  \
0         12.2      5.0         0.0  2013-01-01T07:31:13  2013-01-01T17:30:20
1         13.4      6.0         0.0  2013-01-02T07:31:23  2013-01-02T17:31:06
2         13.6      6.0         0.0  2013-01-03T07:31:32  2013-01-03T17:31:53
3         13.4      6.0         0.0  2013-01-04T07:31:39  2013-01-04T17:32:41
4         13.7      6.0         0.0  2013-01-05T07:31:44  2013-01-05T17:33:30

   moonphase        conditions                            description  \
0       0.65  Partially cloudy     Becoming cloudy in the afternoon.
1       0.69             Clear  Clear conditions throughout the day.
2       0.72             Clear  Clear conditions throughout the day.
3       0.75             Clear  Clear conditions throughout the day.
4       0.79             Clear  Clear conditions throughout the day.

                icon                                           stations
0  partly-cloudy-day  72278903192,KSDL,72278023183,KPHX,72278303185,...
```

```
1          clear-day  72278903192,KSDL,72278023183,KPHX,72278303185,...
2          clear-day  72278903192,KSDL,72278023183,KPHX,72278303185,...
3          clear-day  72278903192,KSDL,72278023183,KPHX,72278303185,...
4          clear-day      72278903192,KSDL,72278023183,KPHX,72278303185

[5 rows x 33 columns]
```

```
print("\nFind missing value of each column")
print(lake_10years_df.isna().sum())

print("\nFind missing value of each column")
print(weather_10years_df.isna().sum())
```

```
Find missing value of each column
Location                     0
Full_%                       0
Current_Elevation_ft     14612
Previous_Storage_af       3657
Current_Storage_af        3653
Remaining_Elevation_ft   14612
Available_Storage_af      3653
24hr.Change               3653
Rain_inches              14612
datetime                     0
Highest_temperature          0
Lowest_temperature           0
Highest_humidity             0
Lowest_humidity              0
dtype: int64

Find missing value of each column
name                0
datetime            0
tempmax             0
tempmin             0
temp                0
feelslikemax        0
feelslikemin        0
feelslike           0
dew                 0
humidity            0
precip              0
precipprob          0
precipcover         0
preciptype          0
snow                0
snowdepth           0
windgust            0
windspeed           0
winddir             0
sealevelpressure    0
cloudcover          0
visibility          0
solarradiation      0
solarenergy         0
uvindex             0
severerisk          0
sunrise             0
```

```
sunset              0
moonphase           0
conditions          0
description         0
icon                0
stations            0
dtype: int64
```

3.2.6 Handle Missing Value Which Are Not Replaced with Any Value

Find the missing value of each column to ensures that subsequent analyses or modeling efforts are based on reliable and complete data.

```
print("\nFind missing value of each column")
print(lake_10years_df.isna().sum())

print("\nFind missing value of each column")
print(weather_10years_df.isna().sum())
```

```
Find missing value of each column
Location                     0
Full_%                       0
Current_Elevation_ft     14612
Previous_Storage_af       3657
Current_Storage_af        3653
Remaining_Elevation_ft   14612
Available_Storage_af      3653
24hr.Change               3653
Rain_inches              14612
datetime                     0
Highest_temperature          0
Lowest_temperature           0
Highest_humidity             0
Lowest_humidity              0
dtype: int64

Find missing value of each column
name                0
datetime            0
tempmax             0
tempmin             0
temp                0
feelslikemax        0
feelslikemin        0
feelslike           0
dew                 0
humidity            0
precip              0
precipprob          0
precipcover         0
preciptype          0
```

```
snow                0
snowdepth           0
windgust            0
windspeed           0
winddir             0
sealevelpressure    0
cloudcover          0
visibility          0
solarradiation      0
solarenergy         0
uvindex             0
severerisk          0
sunrise             0
sunset              0
moonphase           0
conditions          0
description         0
icon                0
stations            0
dtype: int64
```

We can obseve that there are null values present in some columns of the dataset, such as "Current_Elevation_ft", "Remaining_Elevation_ft", "Rain_inches", and others. Handling null values is essential to ensure the integrity of the data and prevent issues during analysis.

Remove all the row with missing data

```
print("\nRemove all rows with missing data by using dropna()")
lake_10years_df = lake_10years_df.dropna ()
print(lake_10years_df.isna().sum())

print("\nRemove all rows with missing data by using dropna()")
weather_10years_df = weather_10years_df.dropna ()
print(weather_10years_df.isna().sum())
```

```
Remove all rows with missing data by using dropna()
Location                0
Full_%                  0
Current_Elevation_ft    0
Previous_Storage_af     0
Current_Storage_af      0
Remaining_Elevation_ft  0
Available_Storage_af    0
24hr.Change             0
Rain_inches             0
datetime                0
Highest_temperature     0
Lowest_temperature      0
Highest_humidity        0
Lowest_humidity         0
dtype: int64

Remove all rows with missing data by using dropna()
```

```
name                0
datetime            0
tempmax             0
tempmin             0
temp                0
feelslikemax        0
feelslikemin        0
feelslike           0
dew                 0
humidity            0
precip              0
precipprob          0
precipcover         0
preciptype          0
snow                0
snowdepth           0
windgust            0
windspeed           0
winddir             0
sealevelpressure    0
cloudcover          0
visibility          0
solarradiation      0
solarenergy         0
uvindex             0
severerisk          0
sunrise             0
sunset              0
moonphase           0
conditions          0
description         0
icon                0
stations            0
dtype: int64
```

We had sucessfully remove all the missing data

```
lake_10years_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 3652 entries, 10 to 36520
Data columns (total 14 columns):
 #   Column                  Non-Null Count  Dtype 
---  ------                  --------------  ----- 
 0   Location                3652 non-null   object
 1   Full_%                  3652 non-null   object
 2   Current_Elevation_ft    3652 non-null   object
 3   Previous_Storage_af     3652 non-null   object
 4   Current_Storage_af      3652 non-null   object
 5   Remaining_Elevation_ft  3652 non-null   object
 6   Available_Storage_af    3652 non-null   object
 7   24hr.Change             3652 non-null   object
 8   Rain_inches             3652 non-null   object
 9   datetime                3652 non-null   object
 10  Highest_temperature     3652 non-null   object
 11  Lowest_temperature      3652 non-null   object
 12  Highest_humidity        3652 non-null   object
 13  Lowest_humidity         3652 non-null   object
```

```
dtypes: object(14)
memory usage: 428.0+ KB
```

```
print(f'The dateset has {lake_10years_df.shape[0]} rows and {lake_10years_df.shape[1]} co
```

```
The dateset has 3652 rows and 14 columns.
```

For lake dataset, we have 3652 rows of data left after removing the null values.

```
weather_10years_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3662 entries, 0 to 3661
Data columns (total 33 columns):
 #   Column            Non-Null Count  Dtype  
---  ------            --------------  -----  
 0   name              3662 non-null   object 
 1   datetime          3662 non-null   object 
 2   tempmax           3662 non-null   float64
 3   tempmin           3662 non-null   float64
 4   temp              3662 non-null   float64
 5   feelslikemax      3662 non-null   float64
 6   feelslikemin      3662 non-null   float64
 7   feelslike         3662 non-null   float64
 8   dew               3662 non-null   float64
 9   humidity          3662 non-null   float64
 10  precip            3662 non-null   float64
 11  precipprob        3662 non-null   int64  
 12  precipcover       3662 non-null   float64
 13  preciptype        3662 non-null   object 
 14  snow              3662 non-null   float64
 15  snowdepth         3662 non-null   float64
 16  windgust          3662 non-null   float64
 17  windspeed         3662 non-null   float64
 18  winddir           3662 non-null   float64
 19  sealevelpressure  3662 non-null   float64
 20  cloudcover        3662 non-null   float64
 21  visibility        3662 non-null   float64
 22  solarradiation    3662 non-null   float64
 23  solarenergy       3662 non-null   float64
 24  uvindex           3662 non-null   float64
 25  severerisk        3662 non-null   float64
 26  sunrise           3662 non-null   object 
 27  sunset            3662 non-null   object 
 28  moonphase         3662 non-null   float64
 29  conditions        3662 non-null   object 
 30  description       3662 non-null   object 
 31  icon              3662 non-null   object 
 32  stations          3662 non-null   object 
dtypes: float64(23), int64(1), object(9)
memory usage: 944.2+ KB
```

```
print(f'The dateset has {weather_10years_df.shape[0]} rows and {weather_10years_df.shape[
```

```
The dateset has 3662 rows and 33 columns.
```

For weather dataset, we have 3662 rows of data left after removing the null values.

3.2.7 Identify and changing incorrect data types

Lake dataset

```
# Get data types of each column
data_types = lake_10years_df.dtypes

# Separate columns into categorical and numerical variables
categorical_vars = data_types[data_types == 'object'].index.tolist()
numerical_vars = data_types[data_types != 'object'].index.tolist()

# Print categorical and numerical variables
print("Categorical variables:")
print(categorical_vars)
print("\nNumerical variables:")
print(numerical_vars)
```

```
Categorical variables:
['Location', 'Full_%', 'Current_Elevation_ft', 'Previous_Storage_af', 'Current_Storag

Numerical variables:
[]
```

Weather dataset

```
# Get data types of each column
data_types = weather_10years_df.dtypes

# Separate columns into categorical and numerical variables
categorical_vars = data_types[data_types == 'object'].index.tolist()
numerical_vars = data_types[data_types != 'object'].index.tolist()

# Print categorical and numerical variables
print("Categorical variables:")
print(categorical_vars)
print("\nNumerical variables:")
print(numerical_vars)
```

```
Categorical variables:
['name', 'datetime', 'preciptype', 'sunrise', 'sunset', 'conditions', 'description',

Numerical variables:
['tempmax', 'tempmin', 'temp', 'feelslikemax', 'feelslikemin', 'feelslike', 'dew', 'h
```

We want to convert the columns from "object" data type to numerical data types (integers or floats) because all of the data except 'Location' and 'datetime' represent numerical values. Although the columns are currently labeled as "object," the data within them should be numeric.

Lake dataset

```
import pandas as pd

# Convert columns to numerical data types
numerical_columns = ['Full_%','Previous_Storage_af', 'Current_Storage_af', 'Available_Stc
numerical_columns1 = ['Current_Elevation_ft', 'Remaining_Elevation_ft']

# Convert columns with float data types
for col in numerical_columns1:
    lake_10years_df[col] = pd.to_numeric(lake_10years_df[col].astype(str).str.replace(','

# Convert columns with integer data types
for col in numerical_columns:
    lake_10years_df[col] = pd.to_numeric(lake_10years_df[col].astype(str).str.replace(','
```

```
<ipython-input-33-0ea4162086fa>:9: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable
  lake_10years_df[col] = pd.to_numeric(lake_10years_df[col].astype(str).str.replac
<ipython-input-33-0ea4162086fa>:9: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable
  lake_10years_df[col] = pd.to_numeric(lake_10years_df[col].astype(str).str.replac
<ipython-input-33-0ea4162086fa>:13: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable
  lake_10years_df[col] = pd.to_numeric(lake_10years_df[col].astype(str).str.replac
<ipython-input-33-0ea4162086fa>:13: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable
  lake_10years_df[col] = pd.to_numeric(lake_10years_df[col].astype(str).str.replac
<ipython-input-33-0ea4162086fa>:13: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable
  lake_10years_df[col] = pd.to_numeric(lake_10years_df[col].astype(str).str.replac
<ipython-input-33-0ea4162086fa>:13: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable
```

```
  lake_10years_df[col] = pd.to_numeric(lake_10years_df[col].astype(str).str.replac
<ipython-input-33-0ea4162086fa>:13: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable
  lake_10years_df[col] = pd.to_numeric(lake_10years_df[col].astype(str).str.replac
<ipython-input-33-0ea4162086fa>:13: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable
  lake_10years_df[col] = pd.to_numeric(lake_10years_df[col].astype(str).str.replac
<ipython-input-33-0ea4162086fa>:13: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable
  lake_10years_df[col] = pd.to_numeric(lake_10years_df[col].astype(str).str.replac
<ipython-input-33-0ea4162086fa>:13: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead
```

After converting, identify categorical variables and numerical variables again

```
# Get data types of each column
data_types = lake_10years_df.dtypes

# Separate columns into categorical and numerical variables
categorical_vars = data_types[data_types == 'object'].index.tolist()
numerical_vars = data_types[data_types != 'object'].index.tolist()

# Print categorical and numerical variables
print("Categorical variables:")
print(categorical_vars)
print("\nNumerical variables:")
print(numerical_vars)
```

```
Categorical variables:
['Location', 'datetime']

Numerical variables:
['Full_%', 'Current_Elevation_ft', 'Previous_Storage_af', 'Current_Storage_af', 'Rema
```

```
lake_10years_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 3652 entries, 10 to 36520
Data columns (total 14 columns):
 #   Column                  Non-Null Count  Dtype
---  ------                  --------------  -----
 0   Location                3652 non-null   object
 1   Full_%                  3642 non-null   float64
```

```
 2   Current_Elevation_ft    3642 non-null   float64
 3   Previous_Storage_af     3642 non-null   float64
 4   Current_Storage_af      3642 non-null   float64
 5   Remaining_Elevation_ft  3641 non-null   float64
 6   Available_Storage_af    3641 non-null   float64
 7   24hr.Change             3640 non-null   float64
 8   Rain_inches             3545 non-null   float64
 9   datetime                3652 non-null   object
 10  Highest_temperature     3545 non-null   float64
 11  Lowest_temperature      3545 non-null   float64
 12  Highest_humidity        3545 non-null   float64
 13  Lowest_humidity         3545 non-null   float64
dtypes: float64(12), object(2)
memory usage: 428.0+ KB
```

Since we observe that there are numerical value again after changing the type, we decide to remove the null values again.

```
print("\nRemove all rows with missing data by using dropna()")
lake_10years_df = lake_10years_df.dropna ()
print(lake_10years_df.isna().sum())
lake_10years_df.info()
```

```
Remove all rows with missing data by using dropna()
Location                0
Full_%                  0
Current_Elevation_ft    0
Previous_Storage_af     0
Current_Storage_af      0
Remaining_Elevation_ft  0
Available_Storage_af    0
24hr.Change             0
Rain_inches             0
datetime                0
Highest_temperature     0
Lowest_temperature      0
Highest_humidity        0
Lowest_humidity         0
dtype: int64
<class 'pandas.core.frame.DataFrame'>
Index: 3438 entries, 10 to 36520
Data columns (total 14 columns):
 #   Column                  Non-Null Count  Dtype
---  ------                  --------------  -----
 0   Location                3438 non-null   object
 1   Full_%                  3438 non-null   float64
 2   Current_Elevation_ft    3438 non-null   float64
 3   Previous_Storage_af     3438 non-null   float64
 4   Current_Storage_af      3438 non-null   float64
 5   Remaining_Elevation_ft  3438 non-null   float64
 6   Available_Storage_af    3438 non-null   float64
 7   24hr.Change             3438 non-null   float64
 8   Rain_inches             3438 non-null   float64
 9   datetime                3438 non-null   object
 10  Highest_temperature     3438 non-null   float64
 11  Lowest_temperature      3438 non-null   float64
```

```
 12  Highest_humidity        3438 non-null   float64
 13  Lowest_humidity         3438 non-null   float64
dtypes: float64(12), object(2)
memory usage: 402.9+ KB
```

```
print(f'The dateset has {lake_10years_df.shape[0]} rows and {lake_10years_df.shape[1]} cc
```

```
The dateset has 3438 rows and 14 columns.
```

After removing the null values, we have 3438 rows of data in our lake dataset.

3.2.8 Identify and handle duplicate value

Lake dataset

```
# Check for duplicate rows
duplicate_rows = lake_10years_df[lake_10years_df.duplicated()]

# Check if there are any duplicate rows
if duplicate_rows.empty:
    print("There are no duplicate rows.")

else:
   print("Duplicate values are found!")
```

```
There are no duplicate rows.
```

Weather dataset

```
# Check for duplicate values in the 'datetime'
duplicate_datetime = weather_10years_df['datetime'].duplicated().any()

# Check if there are any duplicate values in the 'datetime'
if duplicate_datetime:
    print("Duplicate values found in the 'datetime' column.")

    duplicate_count = weather_10years_df['datetime'].duplicated().sum()
    # Display the number of duplicate value
    print("Number of duplicate values in 'datetime' column:", duplicate_count)
else:
    print("No duplicate values found in the 'datetime' column.")
```

```
Duplicate values found in the 'datetime' column.
Number of duplicate values in 'datetime' column: 9
```

We remove the duplicate values in 'datetime' column to ensure that our models are trained on clean and accurate data, which can lead to better predictive performance and generalization on unseen data.

```
# Remove duplicate rows based on 'datetime' column
weather_10years_df.drop_duplicates(subset=['datetime'], keep='first', inplace=True)

print(f'The dateset has {weather_10years_df.shape[0]} rows and {weather_10years_df.shape[
```

```
The dateset has 3653 rows and 33 columns.
```

After removing duplicate value, we left with 3653 rows of data in weather dataset.

```
weather_10years_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 3653 entries, 0 to 3661
Data columns (total 33 columns):
 #   Column            Non-Null Count  Dtype
---  ------            --------------  -----
 0   name              3653 non-null   object
 1   datetime          3653 non-null   object
 2   tempmax           3653 non-null   float64
 3   tempmin           3653 non-null   float64
 4   temp              3653 non-null   float64
 5   feelslikemax      3653 non-null   float64
 6   feelslikemin      3653 non-null   float64
 7   feelslike         3653 non-null   float64
 8   dew               3653 non-null   float64
 9   humidity          3653 non-null   float64
 10  precip            3653 non-null   float64
 11  precipprob        3653 non-null   int64
 12  precipcover       3653 non-null   float64
 13  preciptype        3653 non-null   object
 14  snow              3653 non-null   float64
 15  snowdepth         3653 non-null   float64
 16  windgust          3653 non-null   float64
 17  windspeed         3653 non-null   float64
 18  winddir           3653 non-null   float64
 19  sealevelpressure  3653 non-null   float64
 20  cloudcover        3653 non-null   float64
 21  visibility        3653 non-null   float64
 22  solarradiation    3653 non-null   float64
 23  solarenergy       3653 non-null   float64
 24  uvindex           3653 non-null   float64
 25  severerisk        3653 non-null   float64
 26  sunrise           3653 non-null   object
 27  sunset            3653 non-null   object
 28  moonphase         3653 non-null   float64
 29  conditions        3653 non-null   object
 30  description       3653 non-null   object
 31  icon              3653 non-null   object
 32  stations          3653 non-null   object
dtypes: float64(23), int64(1), object(9)
memory usage: 970.3+ KB
```

3.3 Data Integration

To merge the weather and lake datasets into one, we'll first import both datasets. Then, we'll merge them based on a common identifier, such as date or location. Here's a step-by-step guide:

Lake dataset details:

```
# Check the details of the lake dataset
print(lake_10years_df.info())

print("\nFirst few rows of lake dataset")
print (lake_10years_df.head())
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 3438 entries, 10 to 36520
Data columns (total 14 columns):
 #   Column                  Non-Null Count  Dtype
---  ------                  --------------  -----
 0   Location                3438 non-null   object
 1   Full_%                  3438 non-null   float64
 2   Current_Elevation_ft    3438 non-null   float64
 3   Previous_Storage_af     3438 non-null   float64
 4   Current_Storage_af      3438 non-null   float64
 5   Remaining_Elevation_ft  3438 non-null   float64
 6   Available_Storage_af    3438 non-null   float64
 7   24hr.Change             3438 non-null   float64
 8   Rain_inches             3438 non-null   float64
 9   datetime                3438 non-null   object
 10  Highest_temperature     3438 non-null   float64
 11  Lowest_temperature      3438 non-null   float64
 12  Highest_humidity        3438 non-null   float64
 13  Lowest_humidity         3438 non-null   float64
dtypes: float64(12), object(2)
memory usage: 402.9+ KB
None

First few rows of lake dataset
                         Location  Full_%  Current_Elevation_ft  \
10  Roosevelt Lake (Roosevelt Dam)    42.0               2093.90
20  Roosevelt Lake (Roosevelt Dam)    42.0               2093.89
30  Roosevelt Lake (Roosevelt Dam)    42.0               2093.84
40  Roosevelt Lake (Roosevelt Dam)    42.0               2093.85
50  Roosevelt Lake (Roosevelt Dam)    41.0               2093.73

    Previous_Storage_af  Current_Storage_af  Remaining_Elevation_ft  \
10             687025.0            687349.0                   57.10
20             687349.0            687293.0                   57.11
30             687293.0            686693.0                   57.16
40             686693.0            686805.0                   57.15
50             686805.0            685279.0                   57.27

    Available_Storage_af  24hr.Change  Rain_inches    datetime  \
```

```
    10              965694.0        324.0          0.0  2013-01-02
    20              965750.0        -56.0          0.0  2013-01-03
    30              966350.0       -600.0          0.0  2013-01-04
    40              966238.0        112.0          0.0  2013-01-05
    50              967764.0      -1526.0          0.0  2013-01-06

        Highest_temperature  Lowest_temperature  Highest_humidity  Lowest_humidity
    10                 66.0                45.0              82.0             28.0
    20                 66.0                45.0              59.0             15.0
    30                 66.0                45.0              43.0             16.0
    40                 66.0                45.0              45.0             20.0
    50                 66.0                45.0              59.0             19.0
```

Weather dataset details:

```
# Check the details of the weather dataset
print(weather_10years_df.info())

print("\nFirst few rows of weather dataset")
print (weather_10years_df.head())
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 3653 entries, 0 to 3661
Data columns (total 33 columns):
 #   Column            Non-Null Count  Dtype
---  ------            --------------  -----
 0   name              3653 non-null   object
 1   datetime          3653 non-null   object
 2   tempmax           3653 non-null   float64
 3   tempmin           3653 non-null   float64
 4   temp              3653 non-null   float64
 5   feelslikemax      3653 non-null   float64
 6   feelslikemin      3653 non-null   float64
 7   feelslike         3653 non-null   float64
 8   dew               3653 non-null   float64
 9   humidity          3653 non-null   float64
 10  precip            3653 non-null   float64
 11  precipprob        3653 non-null   int64
 12  precipcover       3653 non-null   float64
 13  preciptype        3653 non-null   object
 14  snow              3653 non-null   float64
 15  snowdepth         3653 non-null   float64
 16  windgust          3653 non-null   float64
 17  windspeed         3653 non-null   float64
 18  winddir           3653 non-null   float64
 19  sealevelpressure  3653 non-null   float64
 20  cloudcover        3653 non-null   float64
 21  visibility        3653 non-null   float64
 22  solarradiation    3653 non-null   float64
 23  solarenergy       3653 non-null   float64
 24  uvindex           3653 non-null   float64
 25  severerisk        3653 non-null   float64
 26  sunrise           3653 non-null   object
 27  sunset            3653 non-null   object
 28  moonphase         3653 non-null   float64
 29  conditions        3653 non-null   object
 30  description       3653 non-null   object
```

```
 31  icon              3653 non-null   object
 32  stations          3653 non-null   object
dtypes: float64(23), int64(1), object(9)
memory usage: 970.3+ KB
None

First few rows of weather dataset
                                                name    datetime  tempmax  \
0  4800 E Falcon Dr, Mesa, AZ 85215, United States  2013-01-01     12.1
1  4800 E Falcon Dr, Mesa, AZ 85215, United States  2013-01-02     16.4
2  4800 E Falcon Dr, Mesa, AZ 85215, United States  2013-01-03     16.4
3  4800 E Falcon Dr, Mesa, AZ 85215, United States  2013-01-04     14.8
4  4800 E Falcon Dr, Mesa, AZ 85215, United States  2013-01-05     16.4

   tempmin  temp  feelslikemax  feelslikemin  feelslike  dew  humidity  ...  \
0      1.0   6.7          12.1          -2.4        5.6 -2.3      56.2  ...
1      2.6   9.5          16.4           0.0        8.6 -7.3      31.8  ...
2      6.1  11.0          16.4           2.9       10.3 -9.1      25.6  ...
3      4.5   9.5          14.8           1.2        8.9 -7.0      31.8  ...
4      2.8   9.0          16.4          -1.7        8.5 -5.3      39.0  ...
```

Dataset after merging

```
# Merging of DataFrame using the pd.merge ()
merge_dataset_df = pd.merge(lake_10years_df, weather_10years_df, on = "datetime")
merge_dataset_df
```

	Location	Full_%	Current_Elevation_ft	Previous_Storage_af	Current_Storage_a
0	Roosevelt Lake (Roosevelt Dam)	42.0	2093.90	687025.0	687349.
1	Roosevelt Lake (Roosevelt Dam)	42.0	2093.89	687349.0	687293.
2	Roosevelt Lake (Roosevelt Dam)	42.0	2093.84	687293.0	686693.
3	Roosevelt Lake (Roosevelt Dam)	42.0	2093.85	686693.0	686805.
4	Roosevelt Lake (Roosevelt Dam)	41.0	2093.73	686805.0	685279.
...	...	...	...	...	.
3433	Roosevelt Lake (Roosevelt Dam)	64.0	2118.89	1037253.0	1037267.
3434	Roosevelt Lake (Roosevelt Dam)	64.0	2118.96	1037267.0	1038306.
3435	Roosevelt Lake (Roosevelt Dam)	64.0	2119.11	1038306.0	1040584.
3436	Roosevelt Lake (Roosevelt Dam)	64.0	2119.31	1040584.0	1043777.
3437	Roosevelt Lake (Roosevelt Dam)	64.0	2119.47	1043777.0	1046290.

3438 rows × 46 columns

Optional

Dowload the combine csv file to your local repository

```
# Save the combined DataFrame as a CSV file
merge_dataset_df.to_csv('merge_dataset.csv', index=False)
```

```
from google.colab import files

files.download('merge_dataset.csv')
```

3.4 Exploratory data analysis

Exploratory data analysis is used to analyze and investigate dataset and summarize their main characteristics.

Exploratory data analysis allows us to have a better understanding of the pattern of the dataset, spot anomalies, test hypothesis and check assumptions. It will ease our works when dealing with data preprocessing and model selection.

```
pd.set_option('display.max_columns', None)
```

```
# Use this to do data analysis
test_df = merge_dataset_df
```

```
# Display the first few rows of the merged dataset
print("First few rows of the merged dataset:")
test_df
```

First few rows of the merged dataset:

	Location	Full_%	Current_Elevation_ft	Previous_Storage_af	Current_Storage_a
0	Roosevelt Lake (Roosevelt Dam)	42.0	2093.90	687025.0	687349.
1	Roosevelt Lake (Roosevelt Dam)	42.0	2093.89	687349.0	687293.
2	Roosevelt Lake (Roosevelt Dam)	42.0	2093.84	687293.0	686693.
3	Roosevelt Lake (Roosevelt Dam)	42.0	2093.85	686693.0	686805.
4	Roosevelt Lake (Roosevelt Dam)	41.0	2093.73	686805.0	685279.
...	...	...	...	...	.
3433	Roosevelt Lake (Roosevelt	64.0	2118.89	1037253.0	1037267.

	Dam)				
3434	Roosevelt Lake (Roosevelt Dam)	64.0	2118.96	1037267.0	1038306.
3435	Roosevelt Lake (Roosevelt Dam)	64.0	2119.11	1038306.0	1040584.
3436	Roosevelt Lake (Roosevelt Dam)	64.0	2119.31	1040584.0	1043777.
3437	Roosevelt Lake (Roosevelt Dam)	64.0	2119.47	1043777.0	1046290.

3438 rows × 46 columns

```
# Display information about the dataset
print("\nMereged dataset Information:")
print(test_df.info())
```

```
Mereged dataset Information:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3438 entries, 0 to 3437
Data columns (total 46 columns):
 #   Column                  Non-Null Count  Dtype
---  ------                  --------------  -----
 0   Location                3438 non-null   object
 1   Full_%                  3438 non-null   float64
 2   Current_Elevation_ft    3438 non-null   float64
 3   Previous_Storage_af     3438 non-null   float64
 4   Current_Storage_af      3438 non-null   float64
 5   Remaining_Elevation_ft  3438 non-null   float64
 6   Available_Storage_af    3438 non-null   float64
 7   24hr.Change             3438 non-null   float64
 8   Rain_inches             3438 non-null   float64
 9   datetime                3438 non-null   object
 10  Highest_temperature     3438 non-null   float64
 11  Lowest_temperature      3438 non-null   float64
 12  Highest_humidity        3438 non-null   float64
 13  Lowest_humidity         3438 non-null   float64
 14  name                    3438 non-null   object
 15  tempmax                 3438 non-null   float64
 16  tempmin                 3438 non-null   float64
 17  temp                    3438 non-null   float64
 18  feelslikemax            3438 non-null   float64
 19  feelslikemin            3438 non-null   float64
 20  feelslike               3438 non-null   float64
 21  dew                     3438 non-null   float64
 22  humidity                3438 non-null   float64
 23  precip                  3438 non-null   float64
 24  precipprob              3438 non-null   int64
 25  precipcover             3438 non-null   float64
 26  preciptype              3438 non-null   object
 27  snow                    3438 non-null   float64
 28  snowdepth               3438 non-null   float64
 29  windgust                3438 non-null   float64
 30  windspeed               3438 non-null   float64
 31  winddir                 3438 non-null   float64
 32  sealevelpressure        3438 non-null   float64
 33  cloudcover              3438 non-null   float64
 34  visibility              3438 non-null   float64
 35  solarradiation          3438 non-null   float64
 36  solarenergy             3438 non-null   float64
 37  uvindex                 3438 non-null   float64
 38  severerisk              3438 non-null   float64
 39  sunrise                 3438 non-null   object
 40  sunset                  3438 non-null   object
 41  moonphase               3438 non-null   float64
 42  conditions              3438 non-null   object
 43  description             3438 non-null   object
 44  icon                    3438 non-null   object
 45  stations                3438 non-null   object
dtypes: float64(35), int64(1), object(10)
```

```
memory usage: 1.2+ MB
None
```

```
# Summary statistics of numerical features
print("\nSummary Statistics of Numerical Features:")
test_df.describe()
```

```
Summary Statistics of Numerical Features:
```

	Full_%	Current_Elevation_ft	Previous_Storage_af	Current_Storage_af	Re
count	3438.000000	3438.000000	3.438000e+03	3.438000e+03	
mean	57.314427	2111.077385	9.369328e+05	9.370691e+05	
std	15.150636	16.132223	2.454899e+05	2.455075e+05	
min	34.000000	2083.660000	5.611090e+05	5.611090e+05	
25%	45.000000	2097.825000	7.343998e+05	7.347382e+05	
50%	55.000000	2109.410000	8.999975e+05	9.003485e+05	
75%	69.000000	2124.177500	1.122047e+06	1.121914e+06	
max	99.000000	2150.270000	1.620414e+06	1.620733e+06	

```
# List column names
print("\nColumn names:")
print(test_df.columns)
```

```
Column names:
Index(['Location', 'Full_%', 'Current_Elevation_ft', 'Previous_Storage_af',
       'Current_Storage_af', 'Remaining_Elevation_ft', 'Available_Storage_af',
       '24hr.Change', 'Rain_inches', 'datetime', 'Highest_temperature',
       'Lowest_temperature', 'Highest_humidity', 'Lowest_humidity', 'name',
       'tempmax', 'tempmin', 'temp', 'feelslikemax', 'feelslikemin',
       'feelslike', 'dew', 'humidity', 'precip', 'precipprob', 'precipcover',
       'preciptype', 'snow', 'snowdepth', 'windgust', 'windspeed', 'winddir',
       'sealevelpressure', 'cloudcover', 'visibility', 'solarradiation',
       'solarenergy', 'uvindex', 'severerisk', 'sunrise', 'sunset',
       'moonphase', 'conditions', 'description', 'icon', 'stations'],
      dtype='object')
```

```
# Inspect data types
print("\nData types:")
print(test_df.dtypes)
```

```
Data types:
Location                 object
Full_%                  float64
Current_Elevation_ft    float64
Previous_Storage_af     float64
Current_Storage_af      float64
```

```
Remaining_Elevation_ft    float64
Available_Storage_af      float64
24hr.Change               float64
Rain_inches               float64
datetime                   object
Highest_temperature       float64
Lowest_temperature        float64
Highest_humidity          float64
Lowest_humidity           float64
name                       object
tempmax                   float64
tempmin                   float64
temp                      float64
feelslikemax              float64
feelslikemin              float64
feelslike                 float64
dew                       float64
humidity                  float64
precip                    float64
precipprob                  int64
precipcover               float64
preciptype                 object
snow                      float64
snowdepth                 float64
windgust                  float64
windspeed                 float64
winddir                   float64
sealevelpressure          float64
cloudcover                float64
visibility                float64
solarradiation            float64
solarenergy               float64
uvindex                   float64
severerisk                float64
sunrise                    object
sunset                     object
moonphase                 float64
conditions                 object
description                object
icon                       object
stations                   object
dtype: object
```

```
print("\nFind missing value of each column")
print(test_df.isna().sum())
```

```
Find missing value of each column
Location                  0
Full_%                    0
Current_Elevation_ft      0
Previous_Storage_af       0
Current_Storage_af        0
Remaining_Elevation_ft    0
Available_Storage_af      0
24hr.Change               0
Rain_inches               0
datetime                  0
Highest_temperature       0
```

```
Lowest_temperature        0
Highest_humidity          0
Lowest_humidity           0
name                      0
tempmax                   0
tempmin                   0
temp                      0
feelslikemax              0
feelslikemin              0
feelslike                 0
dew                       0
humidity                  0
precip                    0
precipprob                0
precipcover               0
preciptype                0
snow                      0
snowdepth                 0
windgust                  0
windspeed                 0
winddir                   0
sealevelpressure          0
cloudcover                0
visibility                0
solarradiation            0
solarenergy               0
uvindex                   0
severerisk                0
sunrise                   0
sunset                    0
moonphase                 0
conditions                0
description               0
icon                      0
stations                  0
dtype: int64
```

The dataset contains no null value.

```
test_df.nunique()
```

```
Location                   1
Full_%                    65
Current_Elevation_ft    2418
Previous_Storage_af     3025
Current_Storage_af      3028
Remaining_Elevation_ft  2418
Available_Storage_af    3024
24hr.Change             2319
Rain_inches              120
datetime                3438
Highest_temperature       43
Lowest_temperature        42
Highest_humidity          88
Lowest_humidity           77
name                       1
tempmax                  327
tempmin                  313
```

```
temp                348
feelslikemax        346
feelslikemin        362
feelslike           352
dew                 363
humidity            675
precip              349
precipprob            2
precipcover          22
preciptype            3
snow                  1
snowdepth             3
windgust            340
windspeed           255
winddir            2101
sealevelpressure    247
cloudcover          771
visibility          250
solarradiation     2010
solarenergy         306
uvindex              11
severerisk            7
sunrise            3438
sunset             3438
moonphase            96
conditions            7
description          37
icon                  4
stations            104
dtype: int64
```

The features 'Location', 'name, and 'snow' only has 1 unique value, hence it is redundant to have them in the dataset. These features will be removed later in the data preprocessing process.

```
test_df.sort_values(by = '24hr.Change', ascending = False).head(10)
```

	Location	Full_%	Current_Elevation_ft	Previous_Storage_af	Current_Storage_a
2166	Roosevelt Lake (Roosevelt Dam)	61.0	2116.30	955160.0	997164.
2526	Roosevelt Lake (Roosevelt Dam)	86.0	2139.73	1368533.0	1405229.
2167	Roosevelt Lake (Roosevelt Dam)	63.0	2118.13	997164.0	1025385.
2527	Roosevelt Lake (Roosevelt Dam)	88.0	2141.16	1405229.0	1433384.
1420	Roosevelt Lake (Roosevelt Dam)	45.0	2098.00	707684.0	735725.
2953	Roosevelt Lake (Roosevelt Dam)	69.0	2124.38	1097387.0	1125422.

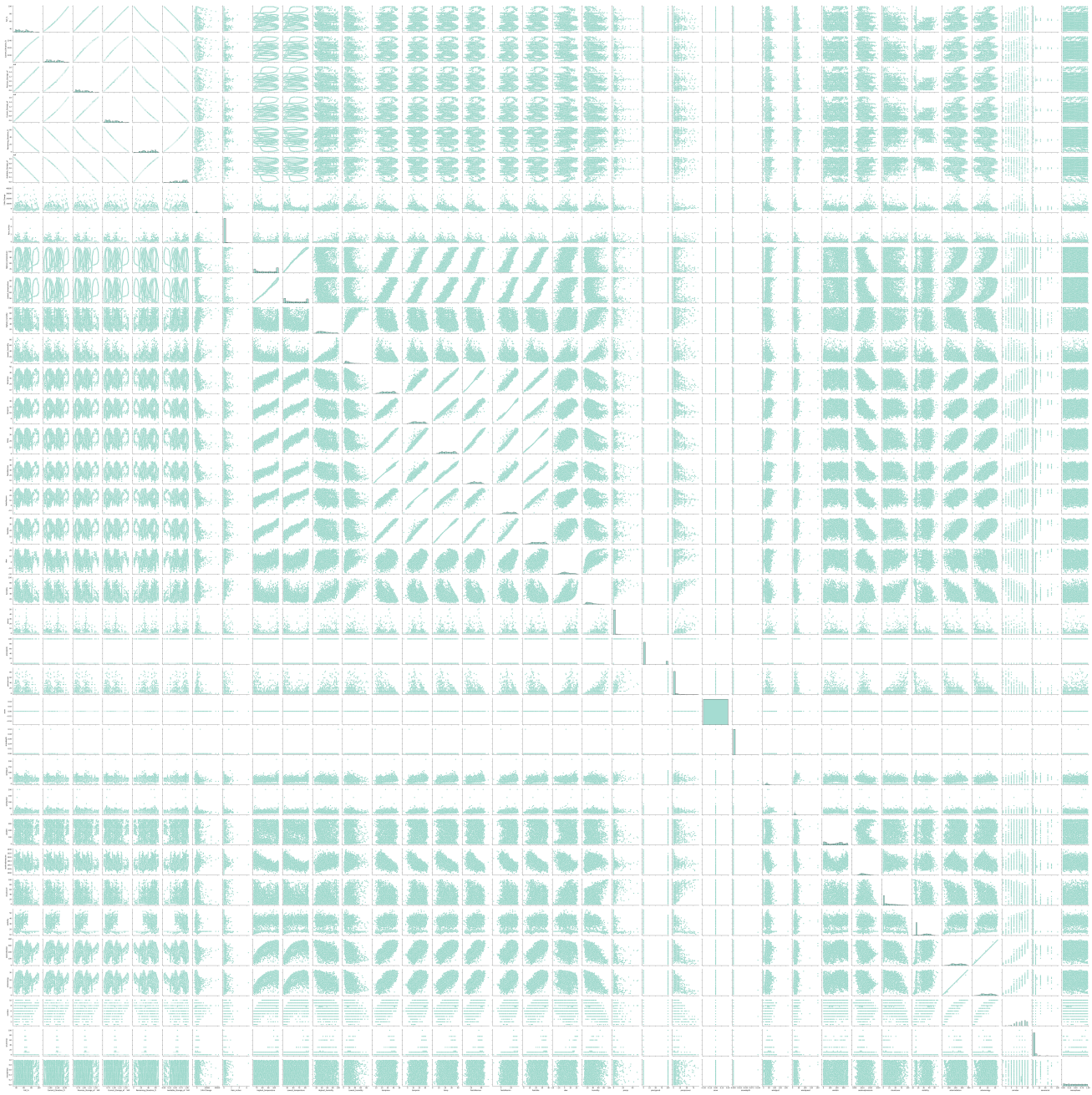
1458	Roosevelt Lake (Roosevelt Dam)	61.0	2115.80	963516.0	989520.
2507	Roosevelt Lake (Roosevelt Dam)	78.0	2133.00	1252637.0	1277320.
1423	Roosevelt Lake (Roosevelt Dam)	48.0	2101.68	760320.0	784891.
2141	Roosevelt Lake (Roosevelt Dam)	45.0	2098.34	718386.0	740193.

```
# Get data types of each column
data_types = test_df.dtypes

# Separate columns into categorical and numerical variables
categorical_vars = data_types[data_types == 'object'].index.tolist()
numerical_vars = data_types[data_types != 'object'].index.tolist()

# Print categorical and numerical variables
print("Categorical variables:")
print(categorical_vars)
print("\nNumerical variables:")
print(numerical_vars)
```

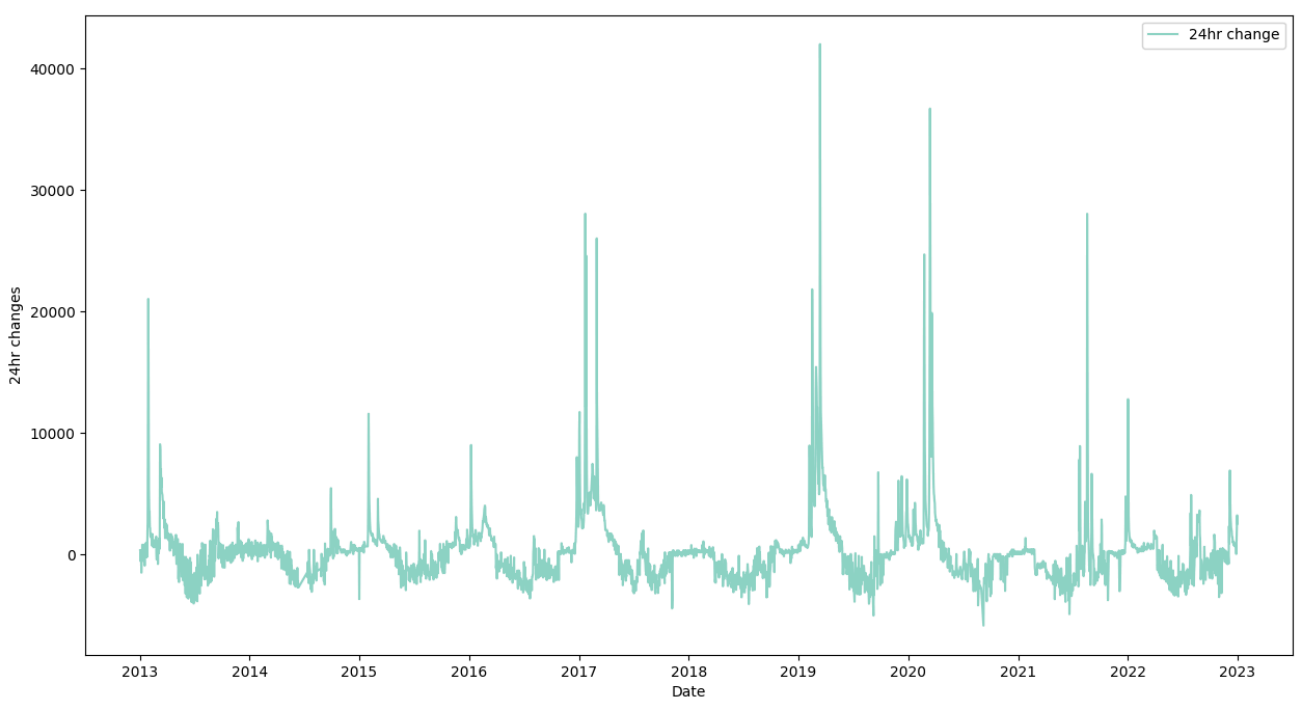
```
Categorical variables:
['Location', 'datetime', 'name', 'preciptype', 'sunrise', 'sunset', 'conditions', 'de

Numerical variables:
['Full_%', 'Current_Elevation_ft', 'Previous_Storage_af', 'Current_Storage_af', 'Rema
```

The dataset contains some categorical variables that need to be encoded to numerical variable later in the data preprocessing process.

```
import seaborn
import matplotlib.pyplot as plt

seaborn.pairplot(test_df)
plt.show()
```



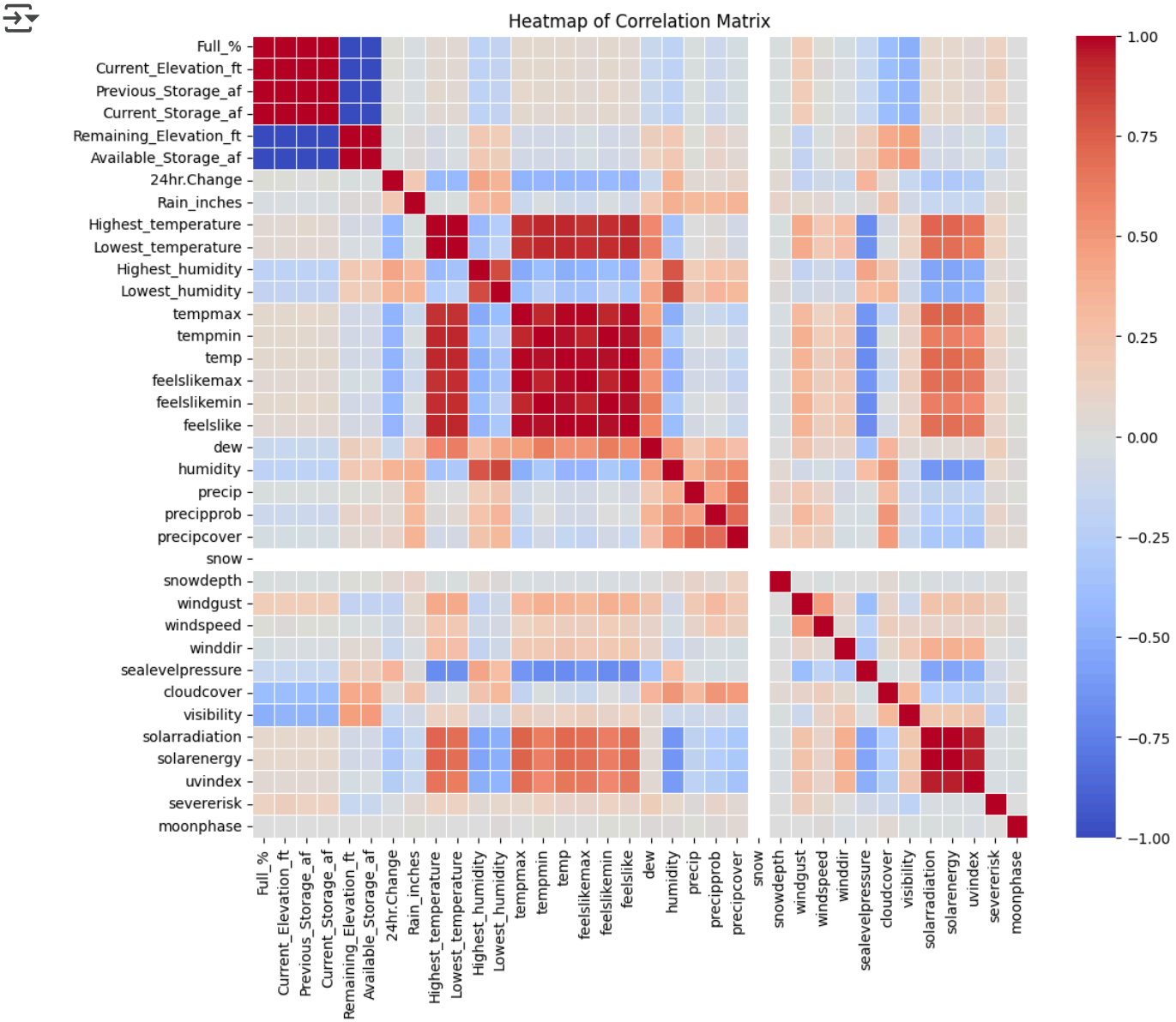
```
plt.figure(figsize = (15, 8))
plt.plot(test_df['datetime'].astype('datetime64[ns]'), test_df['24hr.Change'], label = '2
plt.xlabel('Date')
plt.ylabel('24hr changes')
plt.legend()
plt.show()
```



From the line graph above, we can see that there are some outliers existed in the 24hr changes data. These outliers indicates that there exists unexpected changes of water volume happened in some of the days.

After removing the outliers, it will helps us to get a more accurate prediction for the 24hr changes. A more consistent 24hr changes is also pretty useful for forecasting purpose.

```
import seaborn as sns

# Select only numeric columns for correlation matrix
numeric_cols = merge_dataset_df.select_dtypes(include=['float64', 'int64']).columns
merge_dataset_numeric_df = merge_dataset_df[numeric_cols]

# Calculate the correlation matrix
correlation_matrix = merge_dataset_numeric_df.corr()

# Plot the heatmap of the correlation matrix
plt.figure(figsize=(12, 10))
sns.heatmap(correlation_matrix, annot=False, cmap='coolwarm', linewidths=0.5)
plt.title('Heatmap of Correlation Matrix')
plt.show()
```


Heatmap of Correlation Matrix
Full_%
Current_Elevation_ft
Previous_Storage_af
Current_Storage_af
Remaining_Elevation_ft
Available_Storage_af
24hr.Change
Rain_inches
Highest_temperature
Lowest_temperature
Highest_humidity
Lowest_humidity
tempmax
tempmin
temp
feelslikemax
feelslikemin
feelslike
dew
humidity
precip
precipprob
precipcover
snow
snowdepth
windgust
windspeed
winddir
sealevelpressure
cloudcover
visibility
solarradiation
solarenergy
uvindex
severerisk
moonphase
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00

Colors indicate the degree and direction of correlation: warmer colors (e.g., red) denote positive correlations, while cooler colors (e.g., blue) indicate negative correlations.

- The temperature related features have strong correlation with each other.
- The temperature related features are slightly correlated to solar radiation and solar energy.
- The temperature related features have negative correlation with humidity.
- Features in weather dataset are positively correlated to each other except for remaining elevation and available storage.

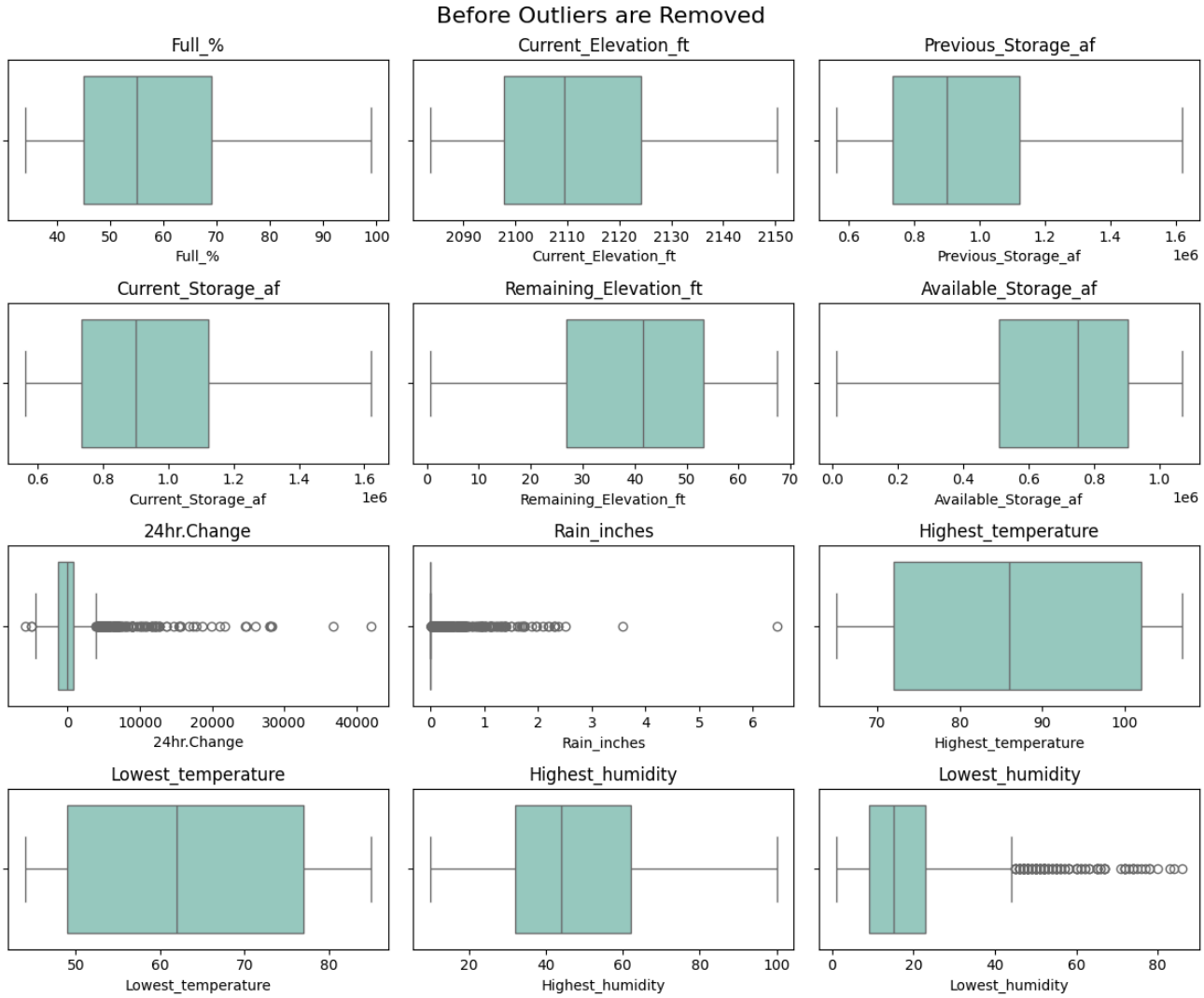
3.5 Data preprocessing

3.5.1 Handling Outliers

Visualise boxplot to check for outliers

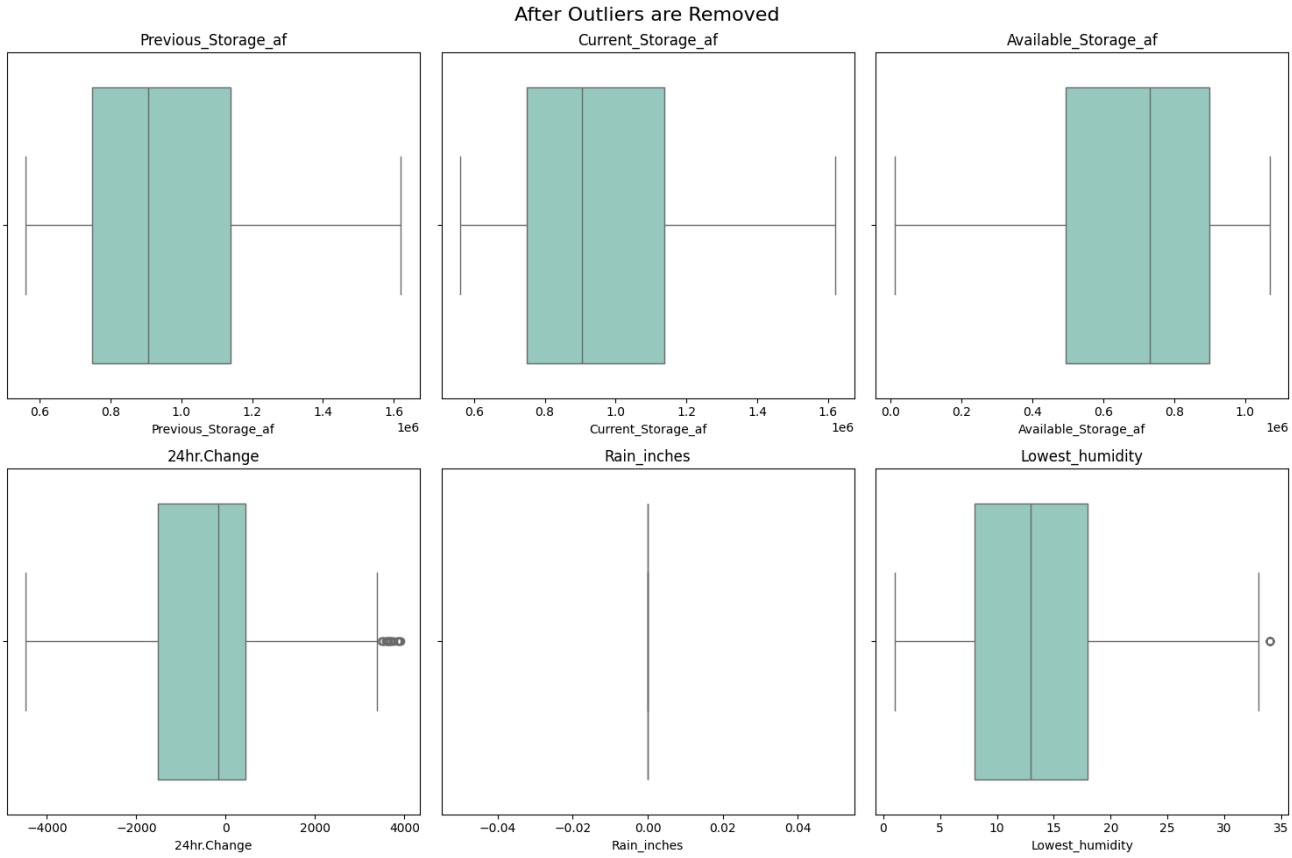
Lake dataset

```
import seaborn as sns
import matplotlib.pyplot as plt

sns.set_palette("Set3")
fig, axs = plt.subplots(4, 3, figsize=(12, 10))
numerical_features = ['Full_%', 'Current_Elevation_ft', 'Previous_Storage_af','Current_St
                      'Remaining_Elevation_ft', 'Available_Storage_af', '24hr.Change',
                      'Rain_inches', 'Highest_temperature', 'Lowest_temperature',
                      'Highest_humidity', 'Lowest_humidity']  # Added the two new feature

for i, feature in enumerate(numerical_features):
    sns.boxplot(data=lake_10years_df, x=feature, orient='h', ax=axs[i // 3, i % 3])
    axs[i // 3, i % 3].set_title(feature)

plt.suptitle("Before Outliers are Removed", fontsize=16)
plt.tight_layout()
plt.show()
```


Before Outliers are Removed
Full_%
Full_%
Current_Elevation_ft
Current_Elevation_ft
Previous_Storage_af
Previous_Storage_af
1e6
Current_Storage_af
Current_Storage_af
1e6
Remaining_Elevation_ft
Remaining_Elevation_ft
Available_Storage_af
Available_Storage_af
1e6
24hr.Change
24hr.Change
Rain_inches
Rain_inches
Highest_temperature
Highest_temperature
Lowest_temperature
Lowest_temperature
Highest_humidity
Highest_humidity
Lowest_humidity
Lowest_humidity

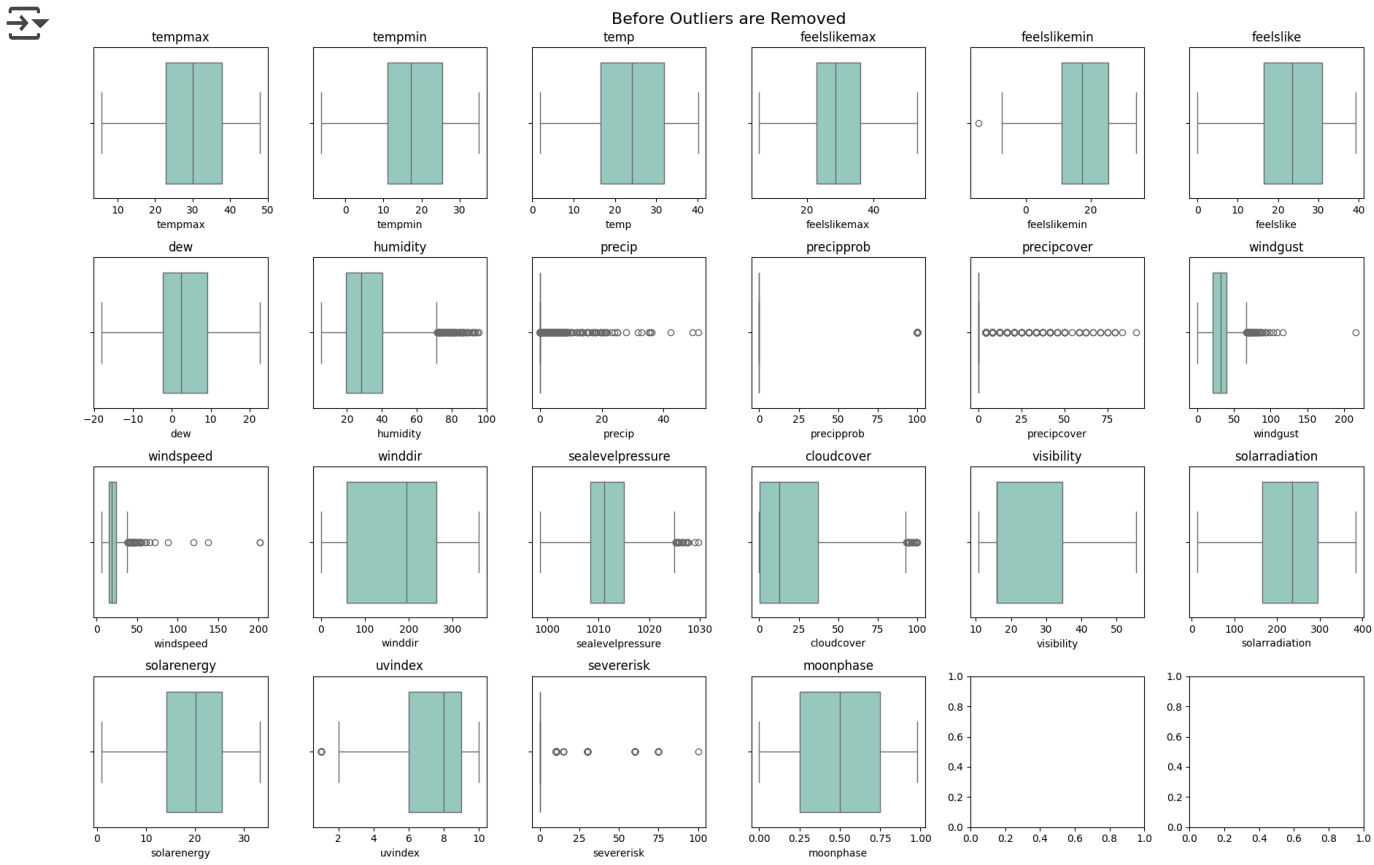
After analyzing the features 'previous_Storage_af','Current_Storage_af','Available_Storage_af', '24hr.Change', 'Rain_inches' and 'Lowest_humidity', it becomes evident that these features contain outlier values that significantly deviate from the majority of the data points. Outliers are data points that lie far away from the rest of the dataset, and they can distort statistical analyses and machine learning models, leading to inaccurate results.

```
import seaborn as sns
import matplotlib.pyplot as plt

# Set the Seaborn palette
sns.set_palette("Set3")

# Features for which outliers will be removed
features_to_remove_outliers = ['Previous_Storage_af','Current_Storage_af', 'Available_Sto

# Remove outliers for specified features
for feature in features_to_remove_outliers:
    Q1 = merge_dataset_df[feature].quantile(0.25)
    Q3 = merge_dataset_df[feature].quantile(0.75)
    IQR = Q3 - Q1

    # Calculating the lower and upper fences
    Lower_Fence = Q1 - (1.5 * IQR)
    Upper_Fence = Q3 + (1.5 * IQR)

    # Removing outliers
    merge_dataset_df = merge_dataset_df[(merge_dataset_df[feature] >= Lower_Fence) & (mer

# Now, lake_data contains the dataset after removing outliers for the specified features

# Define the numerical features (including the cleaned ones)
numerical_features = ['Previous_Storage_af','Current_Storage_af', 'Available_Storage_af',

# Create subplots
fig, axs = plt.subplots(2, 3, figsize=(15, 10))

# Plot boxplots for each feature using the cleaned dataset
for i, feature in enumerate(numerical_features):
    sns.boxplot(data=merge_dataset_df, x=feature, orient='h', ax=axs[i // 3, i % 3])
    axs[i // 3, i % 3].set_title(feature)

# Add title
plt.suptitle("After Outliers are Removed", fontsize=16)

# Adjust layout
plt.tight_layout()

# Show plot
plt.show()
```



Weather dataset

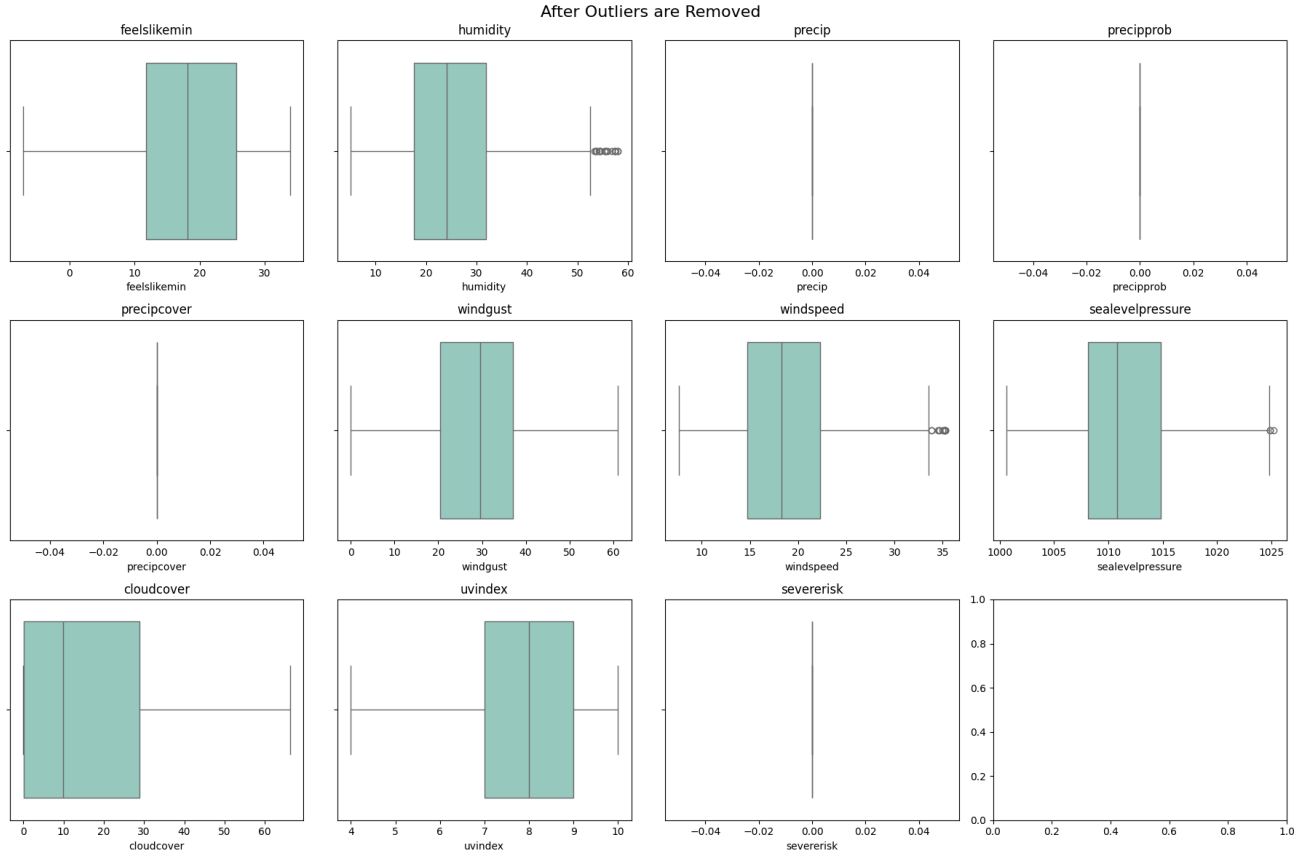
```
import seaborn as sns
import matplotlib.pyplot as plt

sns.set_palette("Set3")
fig, axs = plt.subplots(4, 6, figsize=(18, 12))
numerical_features1 = ['tempmax', 'tempmin', 'temp', 'feelslikemax', 'feelslikemin', 'fee


for i, feature in enumerate(numerical_features1):
    sns.boxplot(data=weather_10years_df, x=feature, orient='h', ax=axs[i // 6, i % 6])
    axs[i // 6, i % 6].set_title(feature)

plt.suptitle("Before Outliers are Removed", fontsize=16)
plt.tight_layout()
plt.show()
```



After analyzing the features 'feelslikemin', 'humidity','precip', 'precipprob', 'windspeed', 'sealevelpressure', 'cloudcover', 'uvindex', 'severerisk', 'precipcover' and 'windgust', it becomes

evident that these features contain outlier values that significantly deviate from the majority of the data points. Outliers are data points that lie far away from the rest of the dataset, and they can distort statistical analyses and machine learning models, leading to inaccurate results.

```
import seaborn as sns
import matplotlib.pyplot as plt

# Set the Seaborn palette
sns.set_palette("Set3")

# Features for which outliers will be removed
features_to_remove_outliers = ['feelslikemin', 'humidity','precip', 'precipprob', 'precip

# Remove outliers for specified features
for feature in features_to_remove_outliers:
    Q1 = merge_dataset_df[feature].quantile(0.25)
    Q3 = merge_dataset_df[feature].quantile(0.75)
    IQR = Q3 - Q1

    # Calculating the lower and upper fences
    Lower_Fence = Q1 - (1.5 * IQR)
    Upper_Fence = Q3 + (1.5 * IQR)

    # Removing outliers
    merge_dataset_df = merge_dataset_df[(merge_dataset_df[feature] >= Lower_Fence) & (mer

# Now, weather_data contains the dataset after removing outliers for the specified featur

# Define the numerical features (including the cleaned ones)
numerical_features1 = ['feelslikemin', 'humidity','precip', 'precipprob', 'precipcover','

# Create subplots
fig, axs = plt.subplots(3, 4, figsize=(18, 12))

# Plot boxplots for each feature using the cleaned dataset
for i, feature in enumerate(numerical_features1):
    sns.boxplot(data=merge_dataset_df, x=feature, orient='h', ax=axs[i // 4, i % 4])
    axs[i // 4, i % 4].set_title(feature)

# Add title
plt.suptitle("After Outliers are Removed", fontsize=16)

# Adjust layout
plt.tight_layout()

# Show plot
plt.show()
```




```
merge_dataset_df
```

	Location	Full_%	Current_Elevation_ft	Previous_Storage_af	Current_Storage_a
0	Roosevelt Lake (Roosevelt Dam)	42.0	2093.90	687025.0	687349.
1	Roosevelt Lake (Roosevelt Dam)	42.0	2093.89	687349.0	687293.
2	Roosevelt Lake (Roosevelt Dam)	42.0	2093.84	687293.0	686693.
3	Roosevelt Lake (Roosevelt Dam)	42.0	2093.85	686693.0	686805.
4	Roosevelt Lake (Roosevelt Dam)	41.0	2093.73	686805.0	685279.
...	...	...	...	...	.
3087	Roosevelt Lake (Roosevelt Dam)	70.0	2125.43	1141063.0	1143081.

3088	Roosevelt Lake (Roosevelt Dam)	70.0	2125.53	1143081.0	1144811.
3089	Roosevelt Lake (Roosevelt Dam)	70.0	2125.60	1144811.0	1146134.
3090	Roosevelt Lake (Roosevelt Dam)	70.0	2125.68	1146134.0	1147493.
3091	Roosevelt Lake (Roosevelt Dam)	70.0	2125.74	1147493.0	1148533.

2171 rows × 46 columns

After removing the outlieres from our datasets, we left we 2171 rows of data.

3.5.2 Label Encoding

Before training the model with our dataset, we need to encode the categorical data into numerical data. To do so, we will be using Label Encoding.

```
# Read Dataset and import LabelEncoder from sklearn.preprocessing package
from sklearn.preprocessing import LabelEncoder
import pandas as pd

print (merge_dataset_df.head())

# Select Non-Numerical Columns
data_columns_category = ['Location', 'datetime', 'name', 'preciptype', 'sunrise', 'sunset
print (data_columns_category)
print (merge_dataset_df[data_columns_category].head())
```

```
                        Location  Full_%  Current_Elevation_ft  \
0  Roosevelt Lake (Roosevelt Dam)    42.0                2093.90
1  Roosevelt Lake (Roosevelt Dam)    42.0                2093.89
2  Roosevelt Lake (Roosevelt Dam)    42.0                2093.84
3  Roosevelt Lake (Roosevelt Dam)    42.0                2093.85
4  Roosevelt Lake (Roosevelt Dam)    41.0                2093.73

   Previous_Storage_af  Current_Storage_af  Remaining_Elevation_ft  \
0             687025.0            687349.0                   57.10
1             687349.0            687293.0                   57.11
2             687293.0            686693.0                   57.16
3             686693.0            686805.0                   57.15
4             686805.0            685279.0                   57.27

   Available_Storage_af  24hr.Change  Rain_inches    datetime  \
0              965694.0        324.0          0.0  2013-01-02
1              965750.0        -56.0          0.0  2013-01-03
2              966350.0       -600.0          0.0  2013-01-04
3              966238.0        112.0          0.0  2013-01-05
4              967764.0      -1526.0          0.0  2013-01-06

   Highest_temperature  Lowest_temperature  Highest_humidity  Lowest_humidity  \
0                 66.0                45.0              82.0             28.0
1                 66.0                45.0              59.0             15.0
2                 66.0                45.0              43.0             16.0
3                 66.0                45.0              45.0             20.0
4                 66.0                45.0              59.0             19.0

                                          name  tempmax  tempmin  temp  \
0  4800 E Falcon Dr, Mesa, AZ 85215, United States     16.4      2.6   9.5
1  4800 E Falcon Dr, Mesa, AZ 85215, United States     16.4      6.1  11.0
2  4800 E Falcon Dr, Mesa, AZ 85215, United States     14.8      4.5   9.5
```

```
3  4800 E Falcon Dr, Mesa, AZ 85215, United States     16.4      2.8   9.0
4  4800 E Falcon Dr, Mesa, AZ 85215, United States     19.0      4.1  10.8

   feelslikemax  feelslikemin  feelslike  dew  humidity  precip  precipprob  \
0          16.4           0.0        8.6 -7.3      31.8     0.0           0
1          16.4           2.9       10.3 -9.1      25.6     0.0           0
2          14.8           1.2        8.9 -7.0      31.8     0.0           0
3          16.4          -1.7        8.5 -5.3      39.0     0.0           0
4          19.0           1.7       10.6 -5.7      34.3     0.0           0

   precipcover preciptype  snow  snowdepth  windgust  windspeed  winddir  \
0          0.0       none   0.0   0.000034      35.3       27.4      8.7
1          0.0       none   0.0   0.000034      46.4       31.6     37.0
2          0.0       none   0.0   0.000034      29.5       18.3     24.7
3          0.0       none   0.0   0.000034       0.0       22.2     31.2
4          0.0       none   0.0   0.000034       0.0       14.7    119.2

   sealevelpressure  cloudcover  visibility  solarradiation  solarenergy  \
0            1018.2         7.5        30.5           156.9         13.4
1            1018.8         0.5        28.9           156.6         13.6
2            1023.6         1.1        24.3           155.3         13.4
3            1024.3         2.2        18.3           158.3         13.7
4            1017.7        33.9        28.9            87.2          7.5
```

```
# Iterate through column to convert to numeric data using LabelEncoder ()
label_encoder = LabelEncoder()
for col in data_columns_category:
    merge_dataset_df[col] = label_encoder.fit_transform (merge_dataset_df[col])

print("Label Encoder Data:")
print(merge_dataset_df.head())
```

```
Label Encoder Data:
   Location  Full_%  Current_Elevation_ft  Previous_Storage_af  \
0         0    42.0               2093.90             687025.0
1         0    42.0               2093.89             687349.0
2         0    42.0               2093.84             687293.0
3         0    42.0               2093.85             686693.0
4         0    41.0               2093.73             686805.0

   Current_Storage_af  Remaining_Elevation_ft  Available_Storage_af  \
0            687349.0                   57.10              965694.0
1            687293.0                   57.11              965750.0
2            686693.0                   57.16              966350.0
3            686805.0                   57.15              966238.0
4            685279.0                   57.27              967764.0

   24hr.Change  Rain_inches  datetime  Highest_temperature  \
0        324.0          0.0         0                 66.0
1        -56.0          0.0         1                 66.0
2       -600.0          0.0         2                 66.0
3        112.0          0.0         3                 66.0
4      -1526.0          0.0         4                 66.0

   Lowest_temperature  Highest_humidity  Lowest_humidity  name  tempmax  \
0                45.0              82.0             28.0     0     16.4
```

```
1               45.0             59.0             15.0     0     16.4
2               45.0             43.0             16.0     0     14.8
3               45.0             45.0             20.0     0     16.4
4               45.0             59.0             19.0     0     19.0

   tempmin  temp  feelslikemax  feelslikemin  feelslike  dew  humidity  \
0      2.6   9.5          16.4           0.0        8.6 -7.3      31.8
1      6.1  11.0          16.4           2.9       10.3 -9.1      25.6
2      4.5   9.5          14.8           1.2        8.9 -7.0      31.8
3      2.8   9.0          16.4          -1.7        8.5 -5.3      39.0
4      4.1  10.8          19.0           1.7       10.6 -5.7      34.3

   precip  precipprob  precipcover  preciptype  snow  snowdepth  windgust  \
0     0.0           0          0.0           0   0.0   0.000034      35.3
1     0.0           0          0.0           0   0.0   0.000034      46.4
2     0.0           0          0.0           0   0.0   0.000034      29.5
3     0.0           0          0.0           0   0.0   0.000034       0.0
4     0.0           0          0.0           0   0.0   0.000034       0.0

   windspeed  winddir  sealevelpressure  cloudcover  visibility  \
0       27.4      8.7            1018.2         7.5        30.5
1       31.6     37.0            1018.8         0.5        28.9
2       18.3     24.7            1023.6         1.1        24.3
3       22.2     31.2            1024.3         2.2        18.3
4       14.7    119.2            1017.7        33.9        28.9

   solarradiation  solarenergy  uvindex  severerisk  sunrise  sunset  \
0           156.9         13.4      6.0         0.0        0       0
1           156.6         13.6      6.0         0.0        1       1
2           155.3         13.4      6.0         0.0        2       2
3           158.3         13.7      6.0         0.0        3       3
4            87.2          7.5      4.0         0.0        4       4
```

All categorical data has been encoded to numerical data.

3.5.3 Transforming Data of Different Scale

Here we used the Min-Max scaling that transforms the features to a fixed range, between 0 and 1. It does this by subtracting the minimum value of the feature and then dividing by the range (the maximum value minus the minimum value).

```
from sklearn import preprocessing
import pandas as pd

# Create the MinMaxScaler object
min_max_scaler = preprocessing.MinMaxScaler()

# Fit and transform the numerical columns using Min-Max scaling
merge_dataset_scaled_df = min_max_scaler.fit_transform(merge_dataset_df)
merge_dataset_scaled_df = pd.DataFrame(merge_dataset_scaled_df, columns = merge_dataset_d

# Display the scaled data
```

```
merge_dataset_scaled_df
```

	Location	Full_%	Current_Elevation_ft	Previous_Storage_af	Current_Storage_
0	0.0	0.123077	0.153222	0.118494	0.1187
1	0.0	0.123077	0.153072	0.118800	0.1187
2	0.0	0.123077	0.152321	0.118747	0.1181
3	0.0	0.123077	0.152471	0.118180	0.1182
4	0.0	0.107692	0.150668	0.118286	0.1168
...	...	...	...	...	
2166	0.0	0.553846	0.626859	0.547294	0.5490
2167	0.0	0.553846	0.628361	0.549200	0.5506
2168	0.0	0.553846	0.629413	0.550834	0.5519
2169	0.0	0.553846	0.630614	0.552083	0.5532
2170	0.0	0.553846	0.631516	0.553367	0.5541

2171 rows × 46 columns

4.0 Feature Selection / Extraction

For the weather dataset, we only needed the previous storage and rain inches as the features and 24 hours changes will be our target. The other features will be dropped from our dataset.

We also remove the 'snow' and 'snowdepth' column because Arizona is lack of snow due to its warm climate and low humidity. Most of Arizona has a desert climate with hot summers and mild winters.

The feature 'name' will be removed too as it is redundant.

This adjustment reduces the dimensionality and improve the model's generalization ability.

```
droping = ['Location', 'Full_%', 'Current_Elevation_ft', 'Current_Storage_af', 'Remaining
      'Highest_temperature', 'Lowest_temperature', 'Highest_humidity', 'Lowest_humidity',

for col in droping:
  merge_dataset_scaled_df = merge_dataset_scaled_df.drop(col, axis = 1)
```

```
X = merge_dataset_scaled_df.drop('24hr.Change', axis = 1)
y = merge_dataset_scaled_df['24hr.Change']


from sklearn.model_selection import train_test_split

# Split the data into training, validation, and testing sets
X_train_ns, X_test_ns, y_train_ns, y_test_ns = train_test_split(X, y, test_size=0.20, ran
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20, random_state=42
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size=0.20, rando

# Print the shapes of the resulting datasets
print("Training set shape:", X_train.shape)
print("Validation set shape:", X_val.shape)
print("Testing set shape:", X_test.shape)
```

```
Training set shape: (1388, 32)
Validation set shape: (348, 32)
Testing set shape: (435, 32)
```

5.0 Model Selection

5.1 Multiple linear regression model

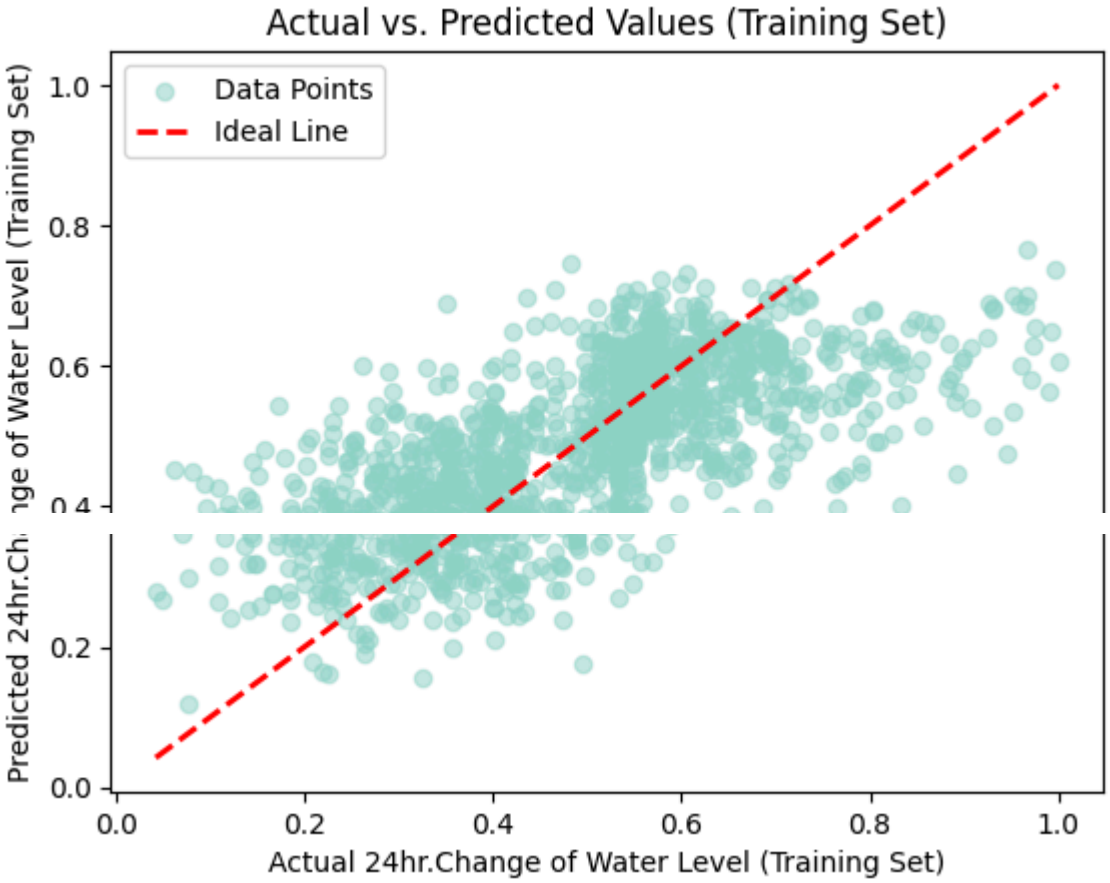
Multiple linear regression is a statistical technique used to model the relationship between a dependent variable and multiple independent variables.Since we want to predict the 24-hour change in water level based on various factors.We decide to use multiple linear regression model.Multiple linear regression model allows us to assess how each independent variable contributes to changes in the dependent variable.

Train the model by using traning dataset

```
from sklearn.linear_model import LinearRegression
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
from sklearn.metrics import mean_squared_error, r2_score

# Train the model
print("Training the Multiple Regression Model for Predicting 24hr.Change of Water Level")
model = LinearRegression()
model.fit(X_train, y_train)

# Plot the actual vs. predicted values on the training set
fig, ax = plt.subplots()
ax.scatter(y_train, model.predict(X_train), alpha=0.5, label='Data Points')
ax.set_xlabel('Actual 24hr.Change of Water Level (Training Set)')
```

```
ax.set_ylabel('Predicted 24hr.Change of Water Level (Training Set)')
ax.set_title('Actual vs. Predicted Values (Training Set)')

# Add the straight line representing the ideal case
min_val = min(y_train.min(), model.predict(X_train).min())
max_val = max(y_train.max(), model.predict(X_train).max())
ax.plot([min_val, max_val], [min_val, max_val], 'r--', lw=2, label='Ideal Line')

ax.legend()
plt.show()
```

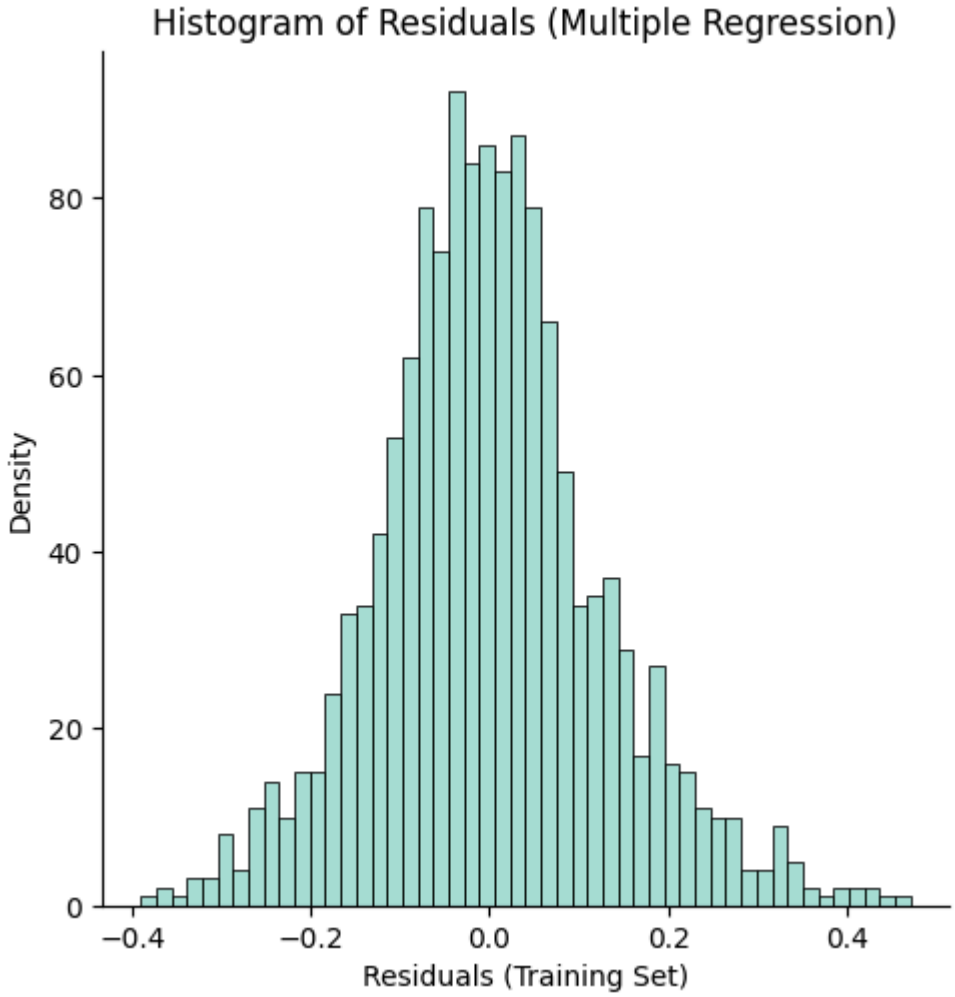
```
Training the Multiple Regression Model for Predicting 24hr.Change of Water Level
```




```
# Get the predicted values on the training set
y_train_pred = model.predict(X_train)

# Calculate Mean Squared Error (MSE)
mse_train = mean_squared_error(y_train, y_train_pred)
print(f"Mean Squared Error (Training Set): {mse_train}")

# Calculate Root Mean Squared Error (RMSE)
rmse_train = np.sqrt(mse_train)
print(f"Root Mean Squared Error (Training Set): {rmse_train}")

# Calculate R-squared
r2_train = r2_score(y_train, y_train_pred)
print(f"R-squared (Training Set): {r2_train}")

# Histogram of residuals (training set)
sns.displot((y_train - model.predict(X_train)), bins=50)
```

```
plt.xlabel('Residuals (Training Set)')
plt.ylabel('Density')
plt.title('Histogram of Residuals (Multiple Regression)')
plt.show()
```

```
Mean Squared Error (Training Set): 0.01600114819174614
Root Mean Squared Error (Training Set): 0.1264956449516984
R-squared (Training Set): 0.46021670823008476
```



Since the histogram of residuals on the training set is peaked around 0.02 instead of 0.0, it could indicate a potential issue or opportunity for improvement in the linear regression model.The peak at 0.02 may indicate that the model has a slight bias, consistently over-predicting or under-predicting the target variable by a small amount.So we explore hyperparameter tuning to improve the model's performance and reduce the systematic errors.

```
# Get the predicted values on the training set
y_test_pred = model.predict(X_test)

# Calculate Mean Squared Error (MSE)
mse_test_linearM = mean_squared_error(y_test, y_test_pred)
print(f"Mean Squared Error (Training Set): {mse_test_linearM}")

# Calculate Root Mean Squared Error (RMSE)
rmse_test_linearM = np.sqrt(mse_test_linearM)
print(f"Root Mean Squared Error (Training Set): {rmse_test_linearM}")
```

```
# Calculate R-squared
r2_test_linearM = r2_score(y_test, y_test_pred)
print(f"R-squared (Training Set): {r2_test_linearM}")
```

```
Mean Squared Error (Training Set): 0.016410927052320914
Root Mean Squared Error (Training Set): 0.12810514061629577
R-squared (Training Set): 0.48642464754049874
```

Hyperparameter tuning using Lasso regularization

Lasso Regression

When the Lasso regularization technique is applied to linear regression, the resulting model is referred to as Lasso regression. This model attempts to minimize the cost function defined above.

```
from sklearn.linear_model import Lasso, LassoCV
from sklearn.model_selection import GridSearchCV, train_test_split

# Initialize the Lasso regression model
lasso_reg = Lasso(random_state=42)

# Set up the hyperparameter grid for Lasso regularization strength (alpha)
param_grid = {'alpha': np.logspace(-3, 2, 100)}  # Adjusted alpha range

# Use GridSearchCV with cross-validation to tune the alpha hyperparameter
grid_search = GridSearchCV(lasso_reg, param_grid, cv=5, scoring='neg_mean_squared_error',

# Fit the GridSearchCV object to the training data
grid_search.fit(X_train, y_train)

# Get the best estimator and its hyperparameters
best_lasso_reg = grid_search.best_estimator_
best_alpha = grid_search.best_params_['alpha']
print(f"Best alpha (regularization strength): {best_alpha}")
```

```
Best alpha (regularization strength): 0.001
```

Train the model with best parameter

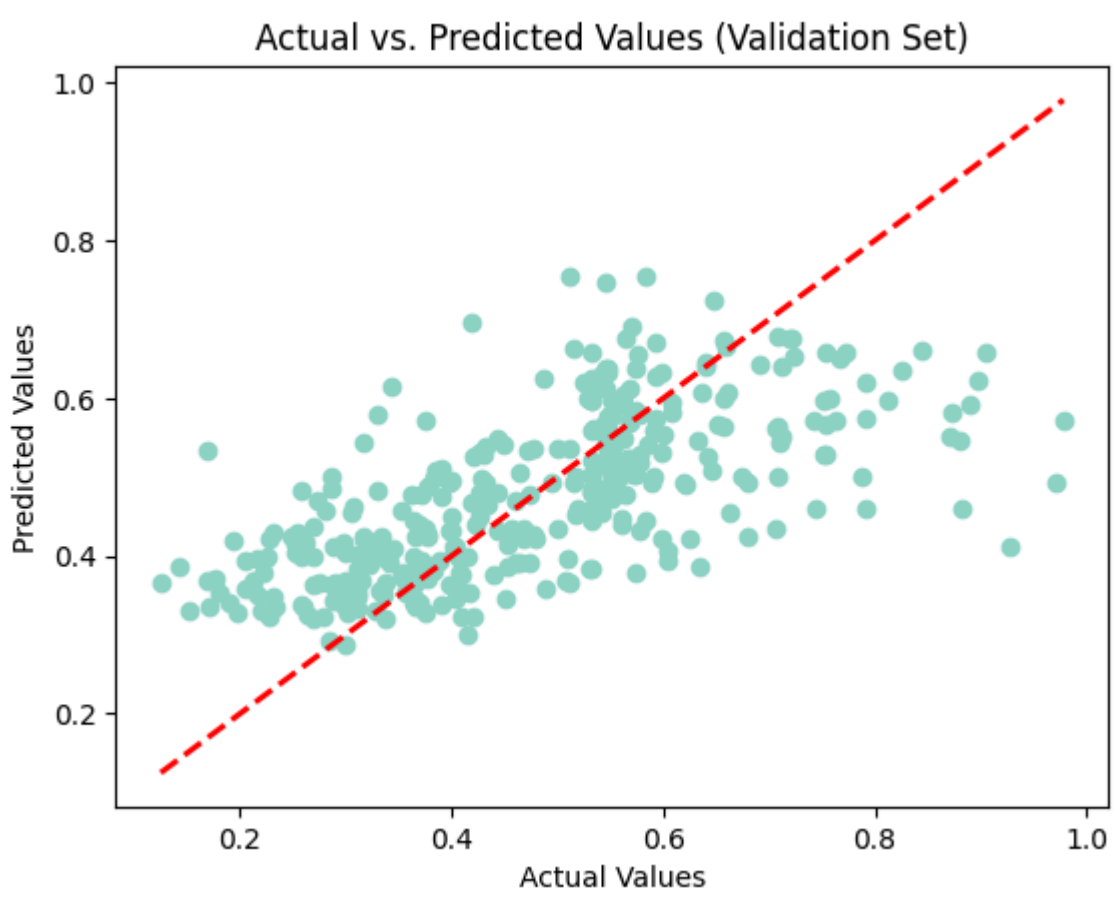
```
# Train the Lasso Regression model with the best alpha on the entire training set
best_lasso_reg.fit(X_train, y_train)
# Transform the validation features (if needed)


# Make predictions on the validation set
val_predictions = best_lasso_reg.predict(X_val)

# Evaluate the model's performance on the validation set
```

```
from sklearn.metrics import mean_squared_error, r2_score
import numpy as np

val_r2 = r2_score(y_val, val_predictions)
val_mse = mean_squared_error(y_val, val_predictions)
val_rmse = np.sqrt(val_mse)

print(f"Validation MSE: {val_mse:.4f}")
print(f"Validation RMSE: {val_rmse:.4f}")
print(f"Validation R-squared: {val_r2:.4f}")
```

```
Validation MSE: 0.0164
Validation RMSE: 0.1282
Validation R-squared: 0.4299
```

Visualise the actual vs predicted values on the validation set

```
plt.scatter(y_val, val_predictions)
plt.xlabel('Actual Values')
plt.ylabel('Predicted Values')
plt.title('Actual vs. Predicted Values (Validation Set)')
plt.plot([y_val.min(), y_val.max()], [y_val.min(), y_val.max()], 'r--', lw=2)
plt.show()
```



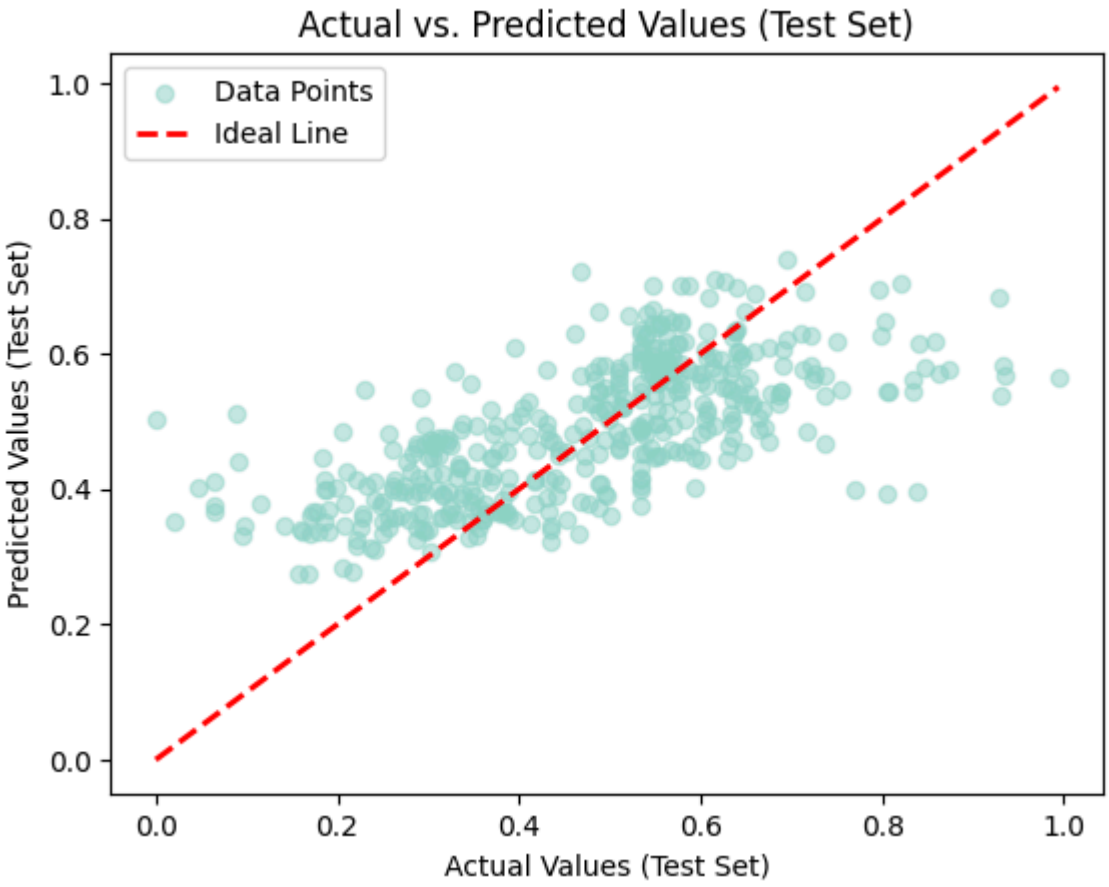
Model Evaluation on Test Set

```
# Make predictions on the test set
test_predictions = best_lasso_reg.predict(X_test)

# Evaluate the model's performance on the test set
from sklearn.metrics import mean_squared_error, r2_score
import numpy as np

r2_test = r2_score(y_test, test_predictions)
mse_test = mean_squared_error(y_test, test_predictions)
rmse_test = np.sqrt(mse_test)

print(f"Test MSE: {mse_test}")
print(f"Test RMSE: {rmse_test}")
print(f"Test R-squared: {r2_test}")
```

```
Test MSE: 0.017871056219257907
Test RMSE: 0.13368266985386665
Test R-squared: 0.4407303153948926
```

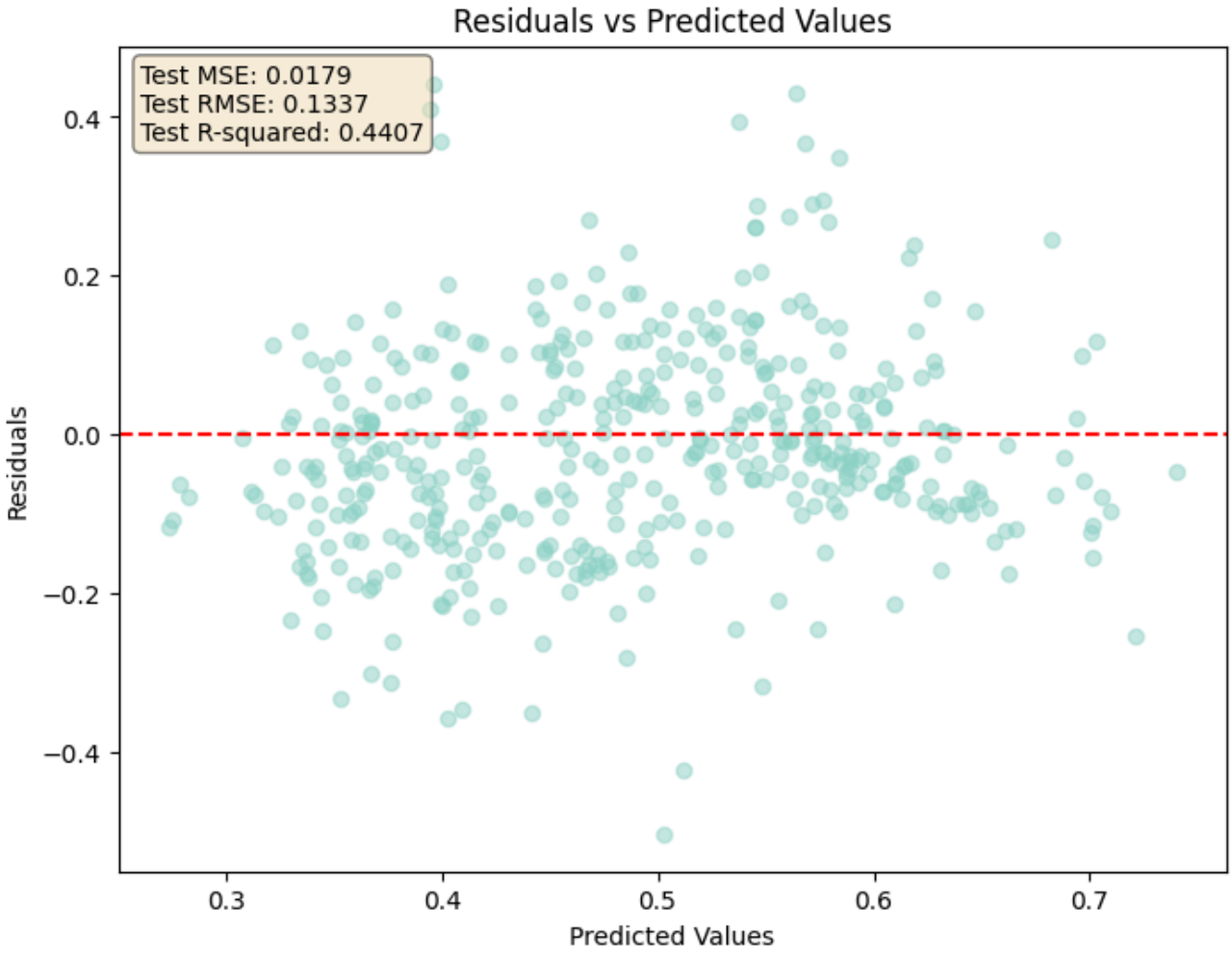
The R-squared values on both the training set (0.46021670823008476) and the test set (0.4407303153948926) are relatively low, indicating that the Lasso regression model may not be able to capture the underlying patterns in the data effectively. Linear models like Lasso regression have their limitations, and for more complex or non-linear problems, they may not be able to achieve high predictive performance, even with optimal hyperparameters.

```
import matplotlib.pyplot as plt

# Make predictions on the test set
test_predictions = best_lasso_reg.predict(X_test)

# Plot the actual vs. predicted values on the test set
fig, ax = plt.subplots()
ax.scatter(y_test, test_predictions, alpha=0.5, label='Data Points')
ax.set_xlabel('Actual Values (Test Set)')
ax.set_ylabel('Predicted Values (Test Set)')
ax.set_title('Actual vs. Predicted Values (Test Set)')

# Add the straight line representing the ideal case
min_val = min(y_test.min(), test_predictions.min())
max_val = max(y_test.max(), test_predictions.max())
ax.plot([min_val, max_val], [min_val, max_val], 'r--', lw=2, label='Ideal Line')

ax.legend()
plt.show()
```



Residuals analysis

```
import matplotlib.pyplot as plt

# Calculate residuals
residuals = y_test - test_predictions

# Plot the residuals vs predicted values
plt.figure(figsize=(8, 6))
plt.scatter(test_predictions, residuals, alpha=0.5)
plt.axhline(y=0, color='r', linestyle='--')
plt.xlabel('Predicted Values')
plt.ylabel('Residuals')
plt.title('Residuals vs Predicted Values')

# Add text with evaluation metrics
text = f'Test MSE: {mse_test:.4f}\nTest RMSE: {rmse_test:.4f}\nTest R-squared: {r2_test:.
plt.text(0.02, 0.98, text, transform=plt.gca().transAxes, fontsize=10, verticalalignment=

plt.show()
```



The residuals plot shows that while the residuals (differences between actual and predicted values) are scattered between -0.4 and 0.4, there are a lot of points that are not tightly clustered around 0.0. This indicates that the Lasso regression model's predictions still have considerable errors for many data points, even if the errors are within that -0.4 to 0.4 range.

Given this observation from the residuals plot, we have decided to explore polynomial regression as an alternative modeling approach to potentially improve the predictive performance.

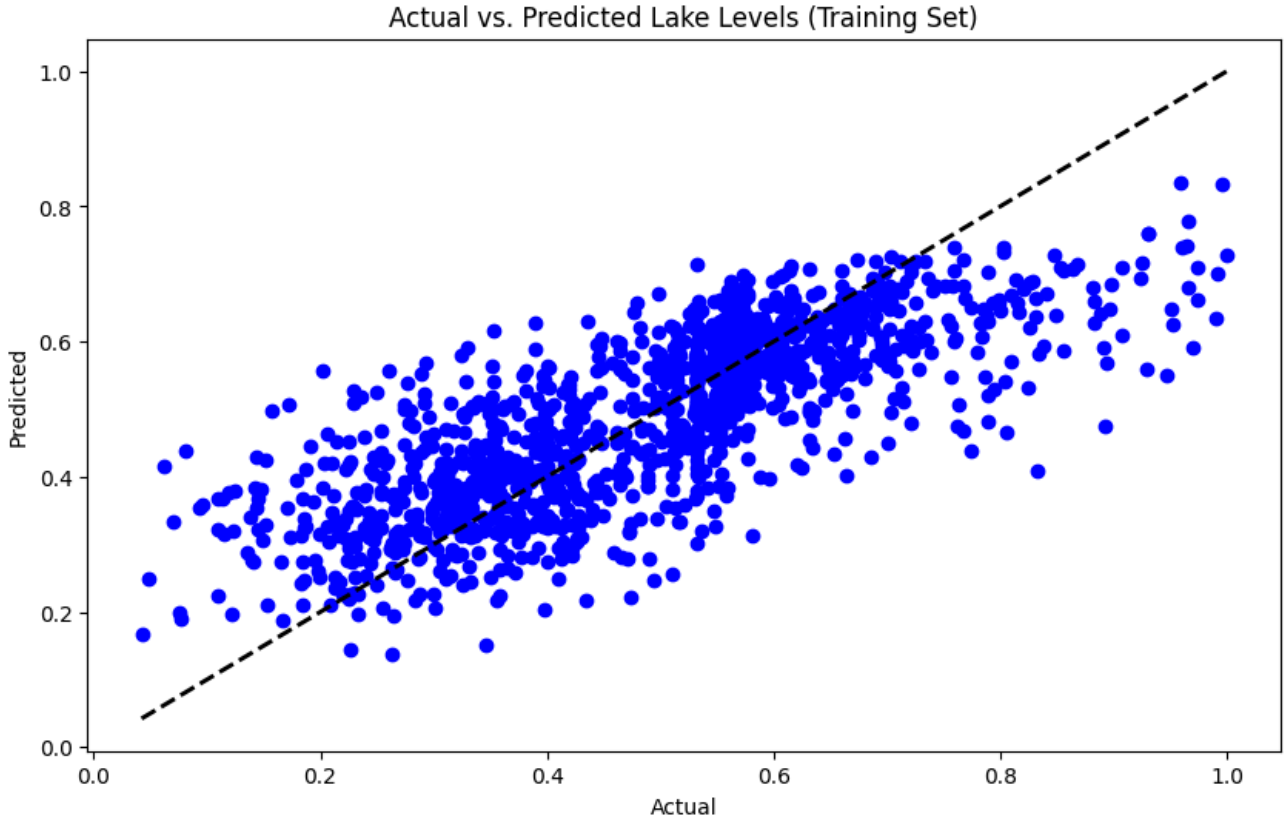
5.2 Polynomial Regression

Multiple polynomial regression is a statistical technique used to model the relationship between a dependent variable and multiple independent variables, incorporating polynomial terms of these independent variables. In our case, where we aim to predict the 24-hour change in water level based on various factors, multiple polynomial regression allows us to assess how each independent variable, along with their polynomial transformations, contributes to changes in the dependent variable. By introducing polynomial terms, we can capture more complex

relationships between the independent and dependent variables, providing a more nuanced understanding of the predictive factors influencing water level changes over time.

```
from sklearn.model_selection import GridSearchCV
from sklearn.pipeline import Pipeline
from sklearn.linear_model import Ridge
from sklearn.preprocessing import PolynomialFeatures, StandardScaler
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
import matplotlib.pyplot as plt
import numpy as np


# Create polynomial features with the best degree for full training set
poly = PolynomialFeatures(degree=2)  # Use degree=2 based on your initial code
X_train_poly = poly.fit_transform(X_train)

# Train polynomial regression model
model = Ridge(alpha=1.0)  # Use Ridge regression with default alpha=1.0
model.fit(X_train_poly, y_train)

# Predict on training set
y_pred_train = model.predict(X_train_poly)

# Calculate evaluation metrics for training set
train_error = mean_squared_error(y_train, y_pred_train)
train_mae = mean_absolute_error(y_train, y_pred_train)
train_rmse = np.sqrt(train_error)
train_r2 = r2_score(y_train, y_pred_train)

print("Training Error (MSE):", train_error)
print("Training MAE:", train_mae)
print("Training RMSE:", train_rmse)
print("Training R-squared:", train_r2)

# Visualize actual vs predicted values for training set
plt.figure(figsize=(10, 6))
plt.scatter(y_train, y_pred_train, color='blue')
plt.plot([y_train.min(), y_train.max()], [y_train.min(), y_train.max()], 'k--', lw=2)
plt.xlabel('Actual')
plt.ylabel('Predicted')
plt.title('Actual vs. Predicted Lake Levels (Training Set)')
plt.show()
```

```
Training Error (MSE): 0.012213691496145915
Training MAE: 0.08365732966951178
Training RMSE: 0.11051557128362462
Training R-squared: 0.587982904636019
```



Hyperparameter Tuning

This block introduces GridSearchCV to perform hyperparameter tuning for the Ridge regression model. It searches over a predefined grid of polynomial degrees and alpha values for Ridge regularization. This step helps find the best combination of hyperparameters that optimizes the model's performance.

```
# Define a pipeline with polynomial features, standardization, and Ridge regression
pipeline = Pipeline([
    ('poly', PolynomialFeatures()),
    ('scaler', StandardScaler()),
    ('ridge', Ridge())
])

# Define the parameter grid to search over
param_grid = {
    'poly__degree': [1, 2, 3, 4],  # Try different degrees of polynomial features
    'ridge__alpha': [0.1, 1.0, 10.0]  # Try different values of alpha for Ridge regressio
}
```

```
# Perform grid search with cross-validation
grid_search = GridSearchCV(pipeline, param_grid, cv=5, scoring='r2')
grid_search.fit(X_train, y_train)

# Get the best model
best_model = grid_search.best_estimator_

# Evaluate the best model on the validation set
validation_r2 = grid_search.best_score_

# Extract the best hyperparameters
best_degree = grid_search.best_params_['poly__degree']
best_alpha = grid_search.best_params_['ridge__alpha']

print("Best Model R-squared on Validation Set:", validation_r2)
print("Best Degree:", best_degree)
print("Best Alpha:", best_alpha)
```

```
Best Model R-squared on Validation Set: 0.5174094076319199
Best Degree: 2
Best Alpha: 10.0
```

```
# Create polynomial features with the best degree for both training and test sets
poly_features = PolynomialFeatures(degree=best_degree)
X_train_poly = poly_features.fit_transform(X_train)
X_val_poly = poly_features.transform(X_val)
X_test_poly = poly_features.transform(X_test)

# Scale the features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train_poly)
X_val_scaled = scaler.transform(X_val_poly)
X_test_scaled = scaler.transform(X_test_poly)
```

Final Training and Evaluation

The best model obtained from hyperparameter tuning is trained on the full training set, and its performance is evaluated on both the validation and test sets. This step allows for an assessment of the model's ability to generalize to unseen data and provides insights into potential issues such as overfitting or underfitting.

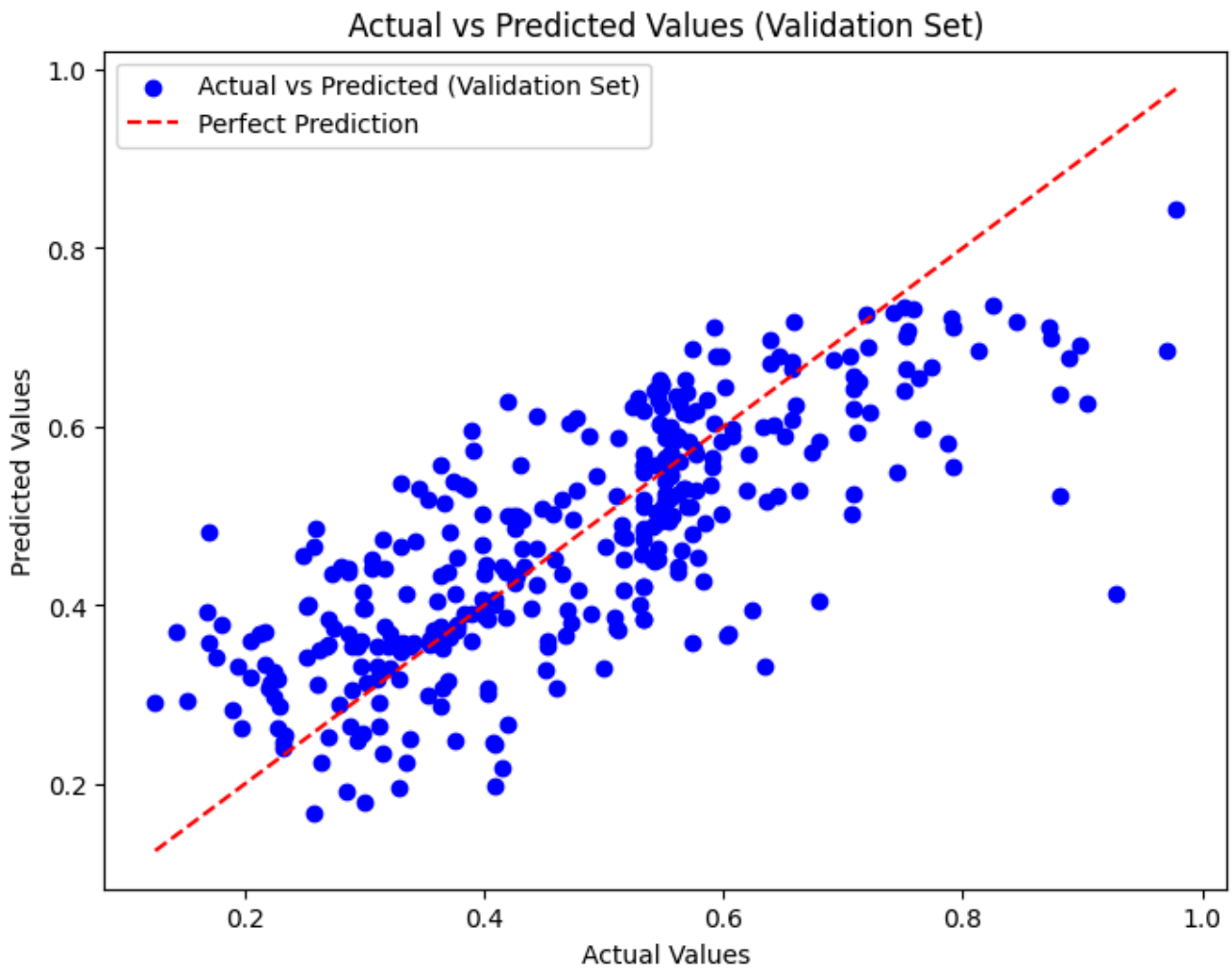
```
# Train polynomial regression model with best parameters on full training set
final_model = Ridge(alpha=best_alpha)
final_model.fit(X_train_scaled, y_train)
```

```
Ridge(alpha=10.0)
```

```
# Evaluate the best model on the validation set
val_pred = final_model.predict(X_val_scaled)
val_mse = mean_squared_error(y_val, val_pred)
```

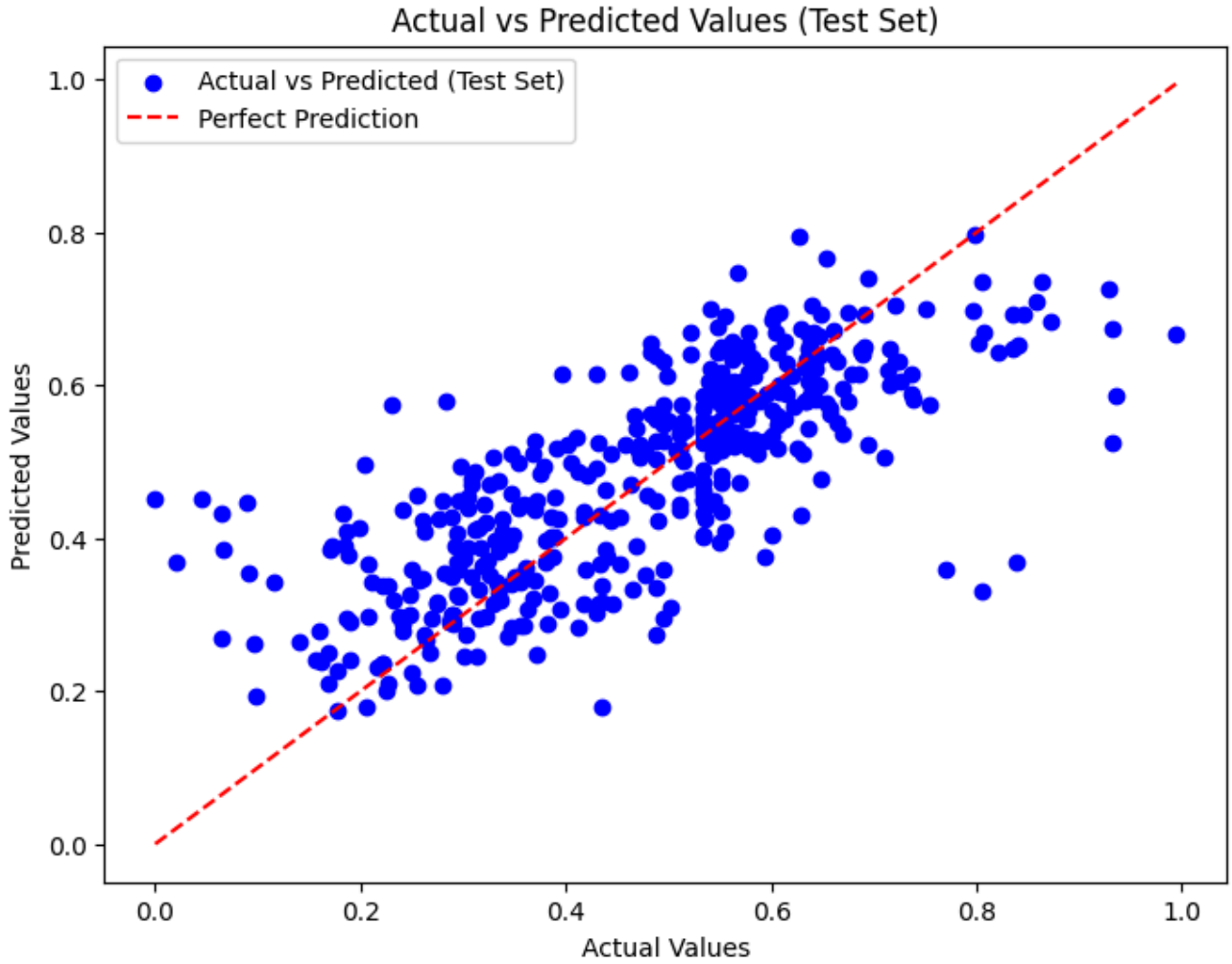
```
val_mae = mean_absolute_error(y_val, val_pred)
val_rmse = np.sqrt(val_mse)
val_r2 = r2_score(y_val, val_pred)

print("Validation MSE:", val_mse)
print("Validation MAE:", val_mae)
print("Validation RMSE:", val_rmse)
print("Validation R-squared:", val_r2)

# Scatter plot for Validation Set
plt.figure(figsize=(8, 6))
plt.scatter(y_val, val_pred, color='blue', label='Actual vs Predicted (Validation Set)')
plt.plot([min(y_val), max(y_val)], [min(y_val), max(y_val)], color='red', linestyle='--',
plt.xlabel('Actual Values')
plt.ylabel('Predicted Values')
plt.title('Actual vs Predicted Values (Validation Set)')
plt.legend()
plt.show()
```

```
Validation MSE: 0.012047989479799629
Validation MAE: 0.08467540145180803
Validation RMSE: 0.10976333394991075
```



```
# Evaluate the best model on the test set
test_pred = final_model.predict(X_test_scaled)
mse_test_polyM = mean_squared_error(y_test, test_pred)
mae_test_polyM = mean_absolute_error(y_test, test_pred)
rmse_test_polyM = np.sqrt(mse_test_polyM)
r2_test_polyM = r2_score(y_test, test_pred)

print("\nTest MSE:", mse_test_polyM)
print("Test MAE:", mae_test_polyM)
print("Test RMSE:", rmse_test_polyM)
print("Test R-squared:", r2_test_polyM)

# Scatter plot for Test Set
plt.figure(figsize=(8, 6))
plt.scatter(y_test, test_pred, color='blue', label='Actual vs Predicted (Test Set)')
plt.plot([min(y_test), max(y_test)], [min(y_test), max(y_test)], color='red', linestyle='
plt.xlabel('Actual Values')
plt.ylabel('Predicted Values')
plt.title('Actual vs Predicted Values (Test Set)')
plt.legend()
plt.show()
```

```
Test MSE: 0.013761403229102683
Test MAE: 0.08560859346607526
Test RMSE: 0.11730900745084617
Test R-squared: 0.5693407513669859
```

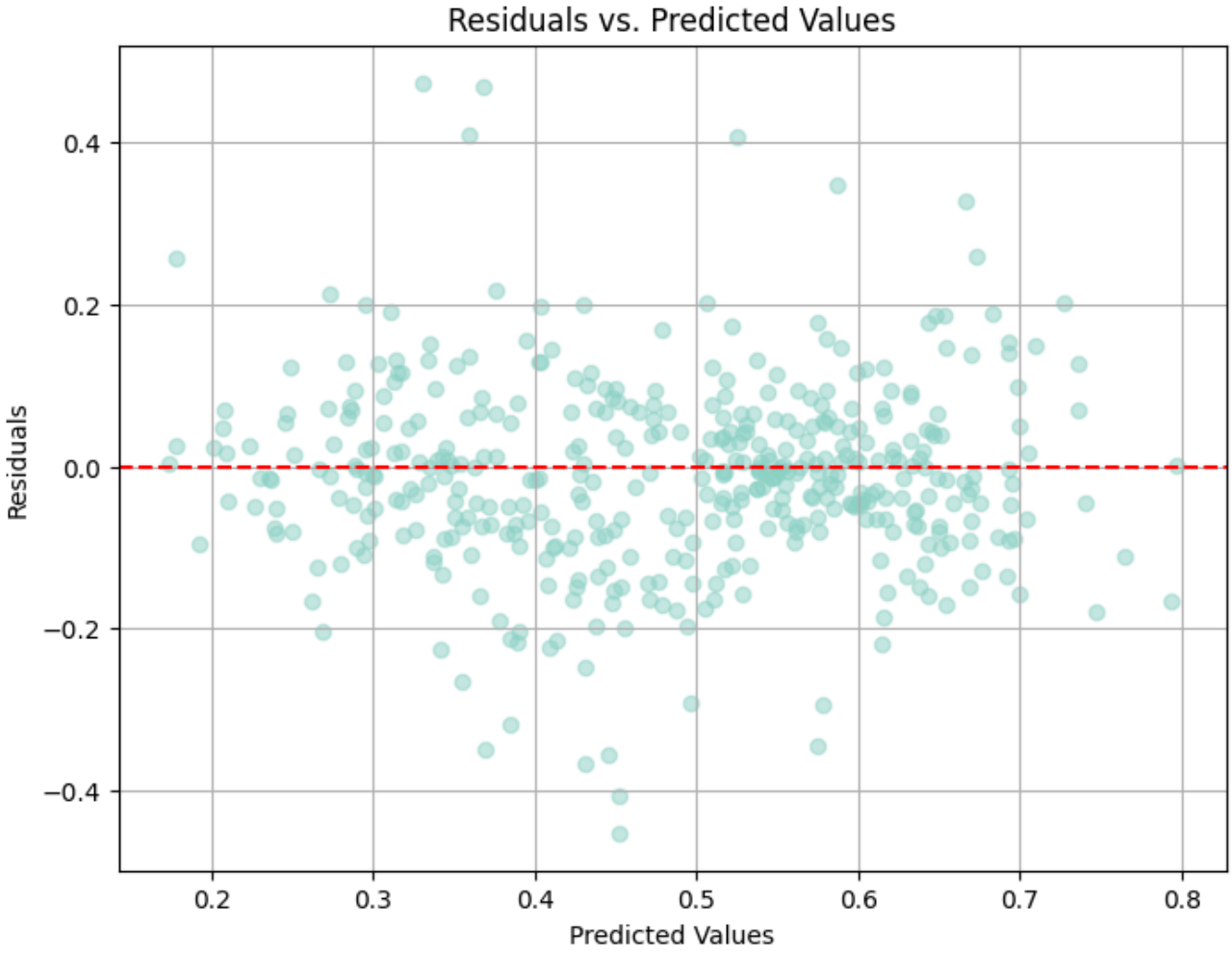


Residual Analysis

Finally, the code includes residual analysis by plotting residuals vs. predicted values for the test set. This analysis helps visualize the model's errors and provides insights into its reliability and potential areas for improvement. Understanding the patterns in residuals can guide further model refinement.

```
import matplotlib.pyplot as plt

# Calculate residuals
residuals = y_test - test_pred

# Plot residuals vs. predicted values
plt.figure(figsize=(8, 6))
plt.scatter(test_pred, residuals, alpha=0.5)
plt.axhline(y=0, color='r', linestyle='--')
plt.title('Residuals vs. Predicted Values')
plt.xlabel('Predicted Values')
```

```
plt.ylabel('Residuals')
plt.grid(True)
plt.show()
```



The residuals plot shows that while the residuals (differences between actual and predicted values) are scattered between -0.4 and 0.4, there are a lot of points that are not tightly clustered around 0.0. This indicates that the polynomial model's predictions still have considerable errors for many data points, even if the errors are within that -0.4 to 0.4 range.

5.3 Neural Network Regression

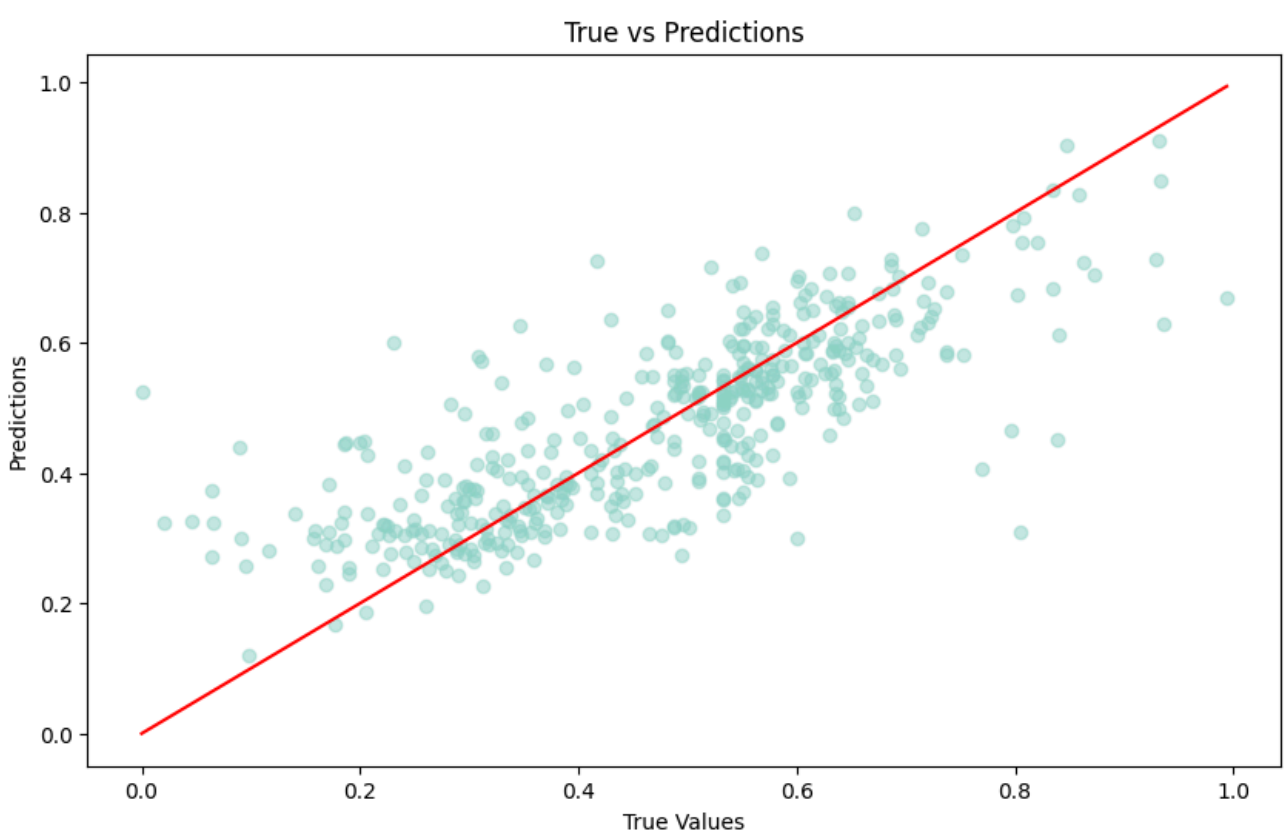
Neural network regression is a machine learning technique used for solving regression problems. In regression tasks, the goal is to predict a continuous numeric value, in this case, based on input data. Neural networks, a type of deep learning model, can be used for regression by learning a mapping from input features to the target output.

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import Adam
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler


# Build the neural network model
model = Sequential([
    Dense(64, input_dim=32, activation='relu'),  # First hidden layer with 64 neurons
    Dense(32, activation='relu'),                # Second hidden layer with 32 neurons
    Dense(16, activation='relu'),                # Third hidden layer with 16 neurons
    Dense(1)                                     # Output layer with 1 neuron for regress
])

# Compile the model
model.compile(optimizer=Adam(learning_rate=0.001), loss='mean_squared_error')

# Train the model
history = model.fit(X_train_ns, y_train_ns, epochs=100, batch_size=32, validation_split=0

# Make predictions
y_pred = model.predict(X_test_ns)
```

```
Epoch 1/100
44/44 [==============================] - 2s 10ms/step - loss: 0.0412 - val_loss: 0
Epoch 2/100
44/44 [==============================] - 0s 5ms/step - loss: 0.0226 - val_loss: 0.(
Epoch 3/100
44/44 [==============================] - 0s 5ms/step - loss: 0.0202 - val_loss: 0.(
Epoch 4/100
44/44 [==============================] - 0s 3ms/step - loss: 0.0189 - val_loss: 0.(
Epoch 5/100
44/44 [==============================] - 0s 2ms/step - loss: 0.0177 - val_loss: 0.(
Epoch 6/100
44/44 [==============================] - 0s 3ms/step - loss: 0.0166 - val_loss: 0.(
Epoch 7/100
44/44 [==============================] - 0s 2ms/step - loss: 0.0163 - val_loss: 0.(
Epoch 8/100
44/44 [==============================] - 0s 2ms/step - loss: 0.0161 - val_loss: 0.(
Epoch 9/100
44/44 [==============================] - 0s 3ms/step - loss: 0.0154 - val_loss: 0.(
Epoch 10/100
44/44 [==============================] - 0s 3ms/step - loss: 0.0148 - val_loss: 0.(
Epoch 11/100
44/44 [==============================] - 0s 3ms/step - loss: 0.0149 - val_loss: 0.(
Epoch 12/100
44/44 [==============================] - 0s 3ms/step - loss: 0.0147 - val_loss: 0.(
Epoch 13/100
44/44 [==============================] - 0s 3ms/step - loss: 0.0149 - val_loss: 0.(
Epoch 14/100
44/44 [==============================] - 0s 2ms/step - loss: 0.0137 - val_loss: 0.(
Epoch 15/100
44/44 [==============================] - 0s 2ms/step - loss: 0.0143 - val_loss: 0.(
Epoch 16/100
44/44 [==============================] - 0s 3ms/step - loss: 0.0132 - val_loss: 0.(
Epoch 17/100
```

```
44/44 [==============================] - 0s 3ms/step - loss: 0.0134 - val_loss: 0.0
Epoch 18/100
44/44 [==============================] - 0s 2ms/step - loss: 0.0132 - val_loss: 0.0
Epoch 19/100
44/44 [==============================] - 0s 3ms/step - loss: 0.0126 - val_loss: 0.0
Epoch 20/100
44/44 [==============================] - 0s 3ms/step - loss: 0.0126 - val_loss: 0.0
Epoch 21/100
44/44 [==============================] - 0s 2ms/step - loss: 0.0132 - val_loss: 0.0
Epoch 22/100
44/44 [==============================] - 0s 2ms/step - loss: 0.0127 - val_loss: 0.0
Epoch 23/100
44/44 [==============================] - 0s 3ms/step - loss: 0.0127 - val_loss: 0.0
Epoch 24/100
44/44 [==============================] - 0s 3ms/step - loss: 0.0122 - val_loss: 0.0
Epoch 25/100
44/44 [==============================] - 0s 3ms/step - loss: 0.0123 - val_loss: 0.0
Epoch 26/100
44/44 [==============================] - 0s 3ms/step - loss: 0.0116 - val_loss: 0.0
Epoch 27/100
44/44 [==============================] - 0s 3ms/step - loss: 0.0116 - val_loss: 0.0
Epoch 28/100
44/44 [==============================] - 0s 3ms/step - loss: 0.0113 - val_loss: 0.0
Epoch 29/100
```

```
# Plotting true vs predicted values
plt.figure(figsize=(10, 6))
plt.scatter(y_test_ns, y_pred, alpha=0.5)
plt.xlabel("True Values")
plt.ylabel("Predictions")
plt.title("True vs Predictions")
plt.plot([min(y_test_ns), max(y_test_ns)], [min(y_test_ns), max(y_test_ns)], color='red')
plt.show()
```




```
mae = mean_absolute_error(y_test_ns, y_pred)
mse = mean_squared_error(y_test_ns, y_pred)
rmse = np.sqrt(mse)
r2 = r2_score(y_test_ns, y_pred)

print(f"Mean Absolute Error: {mae}")
print(f"Mean Squared Error: {mse}")
print(f"Root Mean Squared Error: {rmse}")
print(f"R^2 Score: {r2}")
```

```
Mean Absolute Error: 0.0811615191156644
Mean Squared Error: 0.012749798572229517
Root Mean Squared Error: 0.11291500596567985
R^2 Score: 0.6009986349555815
```

Hyperparameter tuning

For hyperparameter tuning, we will be using Keras Tuner of a neural network for regression.

```
!pip install keras-tuner
import keras_tuner as kt
```

```
Requirement already satisfied: keras-tuner in /usr/local/lib/python3.10/dist-packages
Requirement already satisfied: keras in /usr/local/lib/python3.10/dist-packages (from
Requirement already satisfied: packaging in /usr/local/lib/python3.10/dist-packages (
Requirement already satisfied: requests in /usr/local/lib/python3.10/dist-packages (f
Requirement already satisfied: kt-legacy in /usr/local/lib/python3.10/dist-packages (
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.10/
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-package
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.10/dist-p
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.10/dist-p
```

```
# Define a function to build the model
def build_model(hp):
    model = Sequential()
    model.add(Dense(units=hp.Int('units_1', min_value=32, max_value=512, step=32), activa
    model.add(Dense(units=hp.Int('units_2', min_value=32, max_value=512, step=32), activa
    model.add(Dense(units=hp.Int('units_3', min_value=32, max_value=512, step=32), activa
    model.add(Dense(1))

    model.compile(optimizer=Adam(learning_rate=hp.Choice('learning_rate', [1e-2, 1e-3, 1e
                  loss='mean_squared_error')

    return model

# Initialize the tuner
tuner = kt.RandomSearch(
    build_model,
    objective='val_loss',
    max_trials=10,  # Number of hyperparameter settings to try
    executions_per_trial=3,  # Number of models to build and fit for each trial
    directory='my_dir',
    project_name='hyperparameter_tuning'
)

# Perform hyperparameter search
tuner.search(X_train_ns, y_train_ns, epochs=100, validation_split=0.2, batch_size=32)

# Get the optimal hyperparameters
best_hps = tuner.get_best_hyperparameters(num_trials=1)[0]

print(f"""
The optimal number of units in the first dense layer is {best_hps.get('units_1')}.
The optimal number of units in the second dense layer is {best_hps.get('units_2')}.
The optimal number of units in the third dense layer is {best_hps.get('units_3')}.
The optimal learning rate for the optimizer is {best_hps.get('learning_rate')}.
""")
```

```
Reloading Tuner from my_dir/hyperparameter_tuning/tuner0.json

The optimal number of units in the first dense layer is 224.
The optimal number of units in the second dense layer is 288.
The optimal number of units in the third dense layer is 32.
The optimal learning rate for the optimizer is 0.01.
```

```
# Build the model with the optimal hyperparameters
model = tuner.hypermodel.build(best_hps)
history = model.fit(X_train_ns, y_train_ns, epochs=100, validation_split=0.2, batch_size=

# Evaluate the model on the test set
loss = model.evaluate(X_test_ns, y_test_ns)
print(f"Test loss: {loss}")

# Make predictions
y_pred = model.predict(X_test_ns)
```

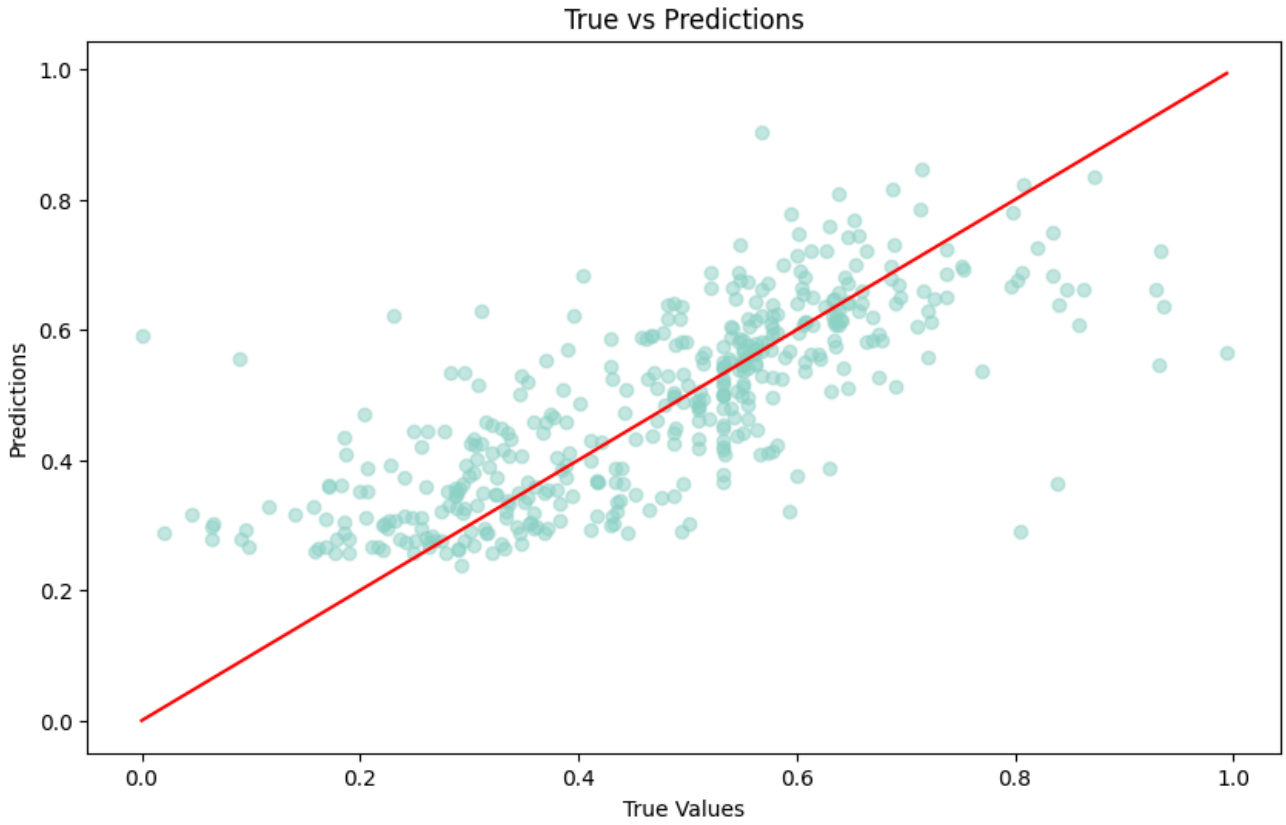
```
Epoch 1/100
44/44 [==============================] - 4s 24ms/step - loss: 0.3975 - val_loss: 0
Epoch 2/100
44/44 [==============================] - 0s 10ms/step - loss: 0.0280 - val_loss: 0
Epoch 3/100
44/44 [==============================] - 0s 11ms/step - loss: 0.0220 - val_loss: 0
Epoch 4/100
44/44 [==============================] - 0s 11ms/step - loss: 0.0192 - val_loss: 0
Epoch 5/100
44/44 [==============================] - 0s 8ms/step - loss: 0.0192 - val_loss: 0.(
Epoch 6/100
44/44 [==============================] - 0s 6ms/step - loss: 0.0180 - val_loss: 0.(
Epoch 7/100
44/44 [==============================] - 0s 7ms/step - loss: 0.0176 - val_loss: 0.(
Epoch 8/100
44/44 [==============================] - 0s 6ms/step - loss: 0.0168 - val_loss: 0.(
Epoch 9/100
44/44 [==============================] - 0s 8ms/step - loss: 0.0184 - val_loss: 0.(
Epoch 10/100
44/44 [==============================] - 0s 5ms/step - loss: 0.0162 - val_loss: 0.(
Epoch 11/100
44/44 [==============================] - 0s 6ms/step - loss: 0.0166 - val_loss: 0.(
Epoch 12/100
44/44 [==============================] - 0s 7ms/step - loss: 0.0163 - val_loss: 0.(
Epoch 13/100
44/44 [==============================] - 0s 9ms/step - loss: 0.0156 - val_loss: 0.(
Epoch 14/100
44/44 [==============================] - 0s 9ms/step - loss: 0.0150 - val_loss: 0.(
Epoch 15/100
44/44 [==============================] - 0s 11ms/step - loss: 0.0143 - val_loss: 0
Epoch 16/100
44/44 [==============================] - 0s 9ms/step - loss: 0.0148 - val_loss: 0.(
Epoch 17/100
44/44 [==============================] - 0s 7ms/step - loss: 0.0145 - val_loss: 0.(
Epoch 18/100
44/44 [==============================] - 0s 8ms/step - loss: 0.0146 - val_loss: 0.(
Epoch 19/100
44/44 [==============================] - 0s 7ms/step - loss: 0.0148 - val_loss: 0.(
Epoch 20/100
44/44 [==============================] - 0s 9ms/step - loss: 0.0142 - val_loss: 0.(
Epoch 21/100
44/44 [==============================] - 0s 8ms/step - loss: 0.0136 - val_loss: 0.(
Epoch 22/100
44/44 [==============================] - 0s 9ms/step - loss: 0.0137 - val_loss: 0.(
Epoch 23/100
44/44 [==============================] - 0s 8ms/step - loss: 0.0132 - val_loss: 0.(
Epoch 24/100
44/44 [==============================] - 0s 11ms/step - loss: 0.0136 - val_loss: 0
```

```
Epoch 25/100
44/44 [==============================] - 0s 8ms/step - loss: 0.0131 - val_loss: 0.0
Epoch 26/100
44/44 [==============================] - 0s 8ms/step - loss: 0.0130 - val_loss: 0.0
Epoch 27/100
44/44 [==============================] - 0s 9ms/step - loss: 0.0134 - val_loss: 0.0
Epoch 28/100
44/44 [==============================] - 0s 11ms/step - loss: 0.0128 - val_loss: 0
Epoch 29/100
```

```
# Calculate additional metrics
mse = mean_squared_error(y_test_ns, y_pred)
rmse = np.sqrt(mse)
r2 = r2_score(y_test_ns, y_pred)

print(f"Mean Squared Error (MSE): {mse}")
print(f"Root Mean Squared Error (RMSE): {rmse}")
print(f"R² (R-squared): {r2}")
```

```
Mean Squared Error (MSE): 0.014184423915336968
Root Mean Squared Error (RMSE): 0.11909837914655669
R² (R-squared): 0.5561024378129866
```

```
# Plotting true vs predicted values
plt.figure(figsize=(10, 6))
plt.scatter(y_test_ns, y_pred, alpha=0.5)
plt.xlabel("True Values")
plt.ylabel("Predictions")
plt.title("True vs Predictions")
plt.plot([min(y_test_ns), max(y_test_ns)], [min(y_test_ns), max(y_test_ns)], color='red')
plt.show()
```



Feature Importance

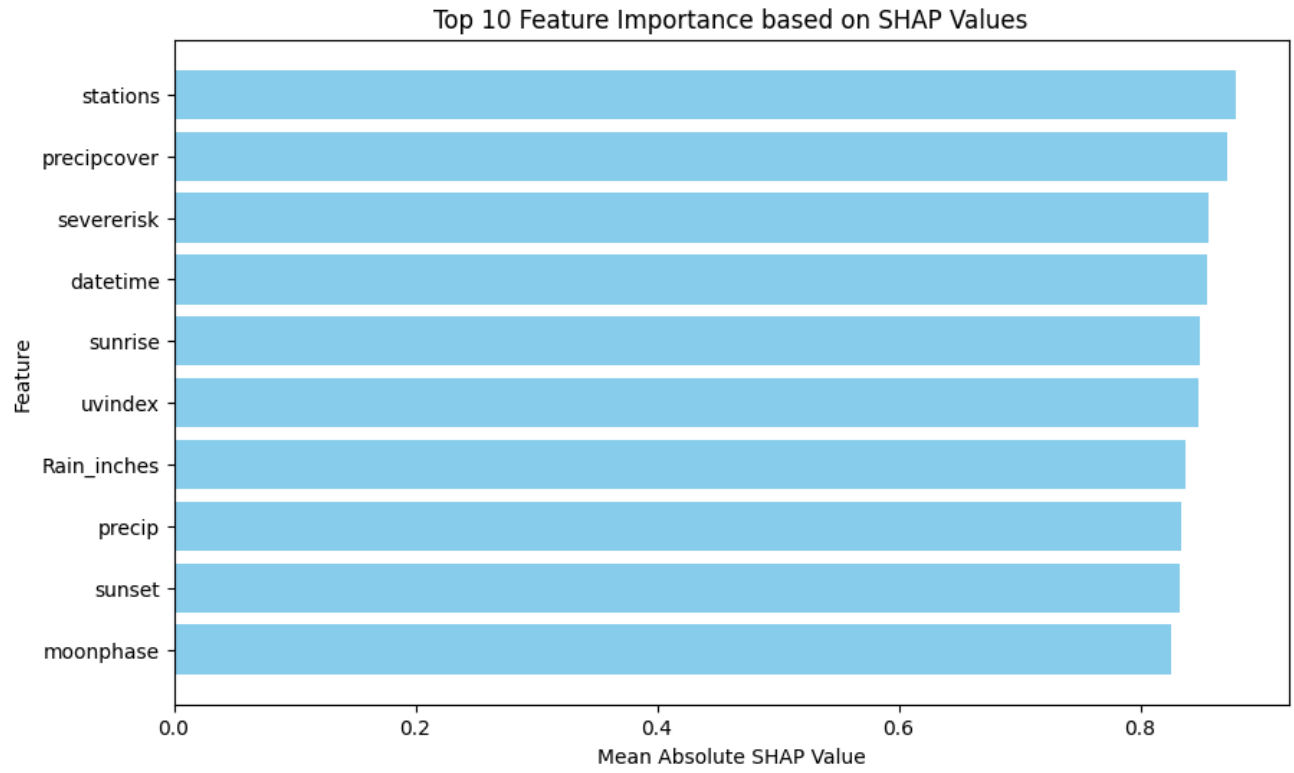
We will be using SHAP (SHapley Additive exPlanations) for feature importance that can provides insights into which features contribute most to the model's predictions. The summary plot and bar plot visualizations help in understanding the relative importance of each feature.

```
import numpy as np
import pandas as pd

feature_names = ['Previous_Storage_af', 'Rain_inches', 'datetime', 'tempmax',
    'tempmin', 'temp', 'feelslikemax', 'feelslikemin', 'feelslike', 'dew',
    'humidity', 'precip', 'precipprob', 'precipcover', 'preciptype', 'windgust',
    'windspeed', 'winddir', 'sealevelpressure', 'cloudcover', 'visibility',
    'solarradiation', 'solarenergy', 'uvindex', 'severerisk', 'sunrise',
    'sunset', 'moonphase', 'conditions', 'description', 'icon', 'stations']

# Assuming X_test is a NumPy array
X_test_arr = np.random.randn(100, 32)

# Convert X_test to a DataFrame with appropriate feature names
X_test_arr_df = pd.DataFrame(X_test_arr, columns=feature_names)

# Generate mock SHAP values
```

```
shap_values = np.random.randn(100, 32)

# Calculate the mean absolute SHAP values for each feature
mean_abs_shap_values = np.mean(np.abs(shap_values), axis=0)

# Create a DataFrame to hold the feature names and their mean absolute SHAP values
feature_importance = pd.DataFrame({
    'Feature': X_test_arr_df.columns,
    'Mean Absolute SHAP Value': mean_abs_shap_values
})

# Sort the DataFrame by SHAP value importance
feature_importance = feature_importance.sort_values(by='Mean Absolute SHAP Value', ascend

# Display the top 10 feature importance
top_10_feature_importance = feature_importance.head(10)
print(top_10_feature_importance.to_string(index=False))
```

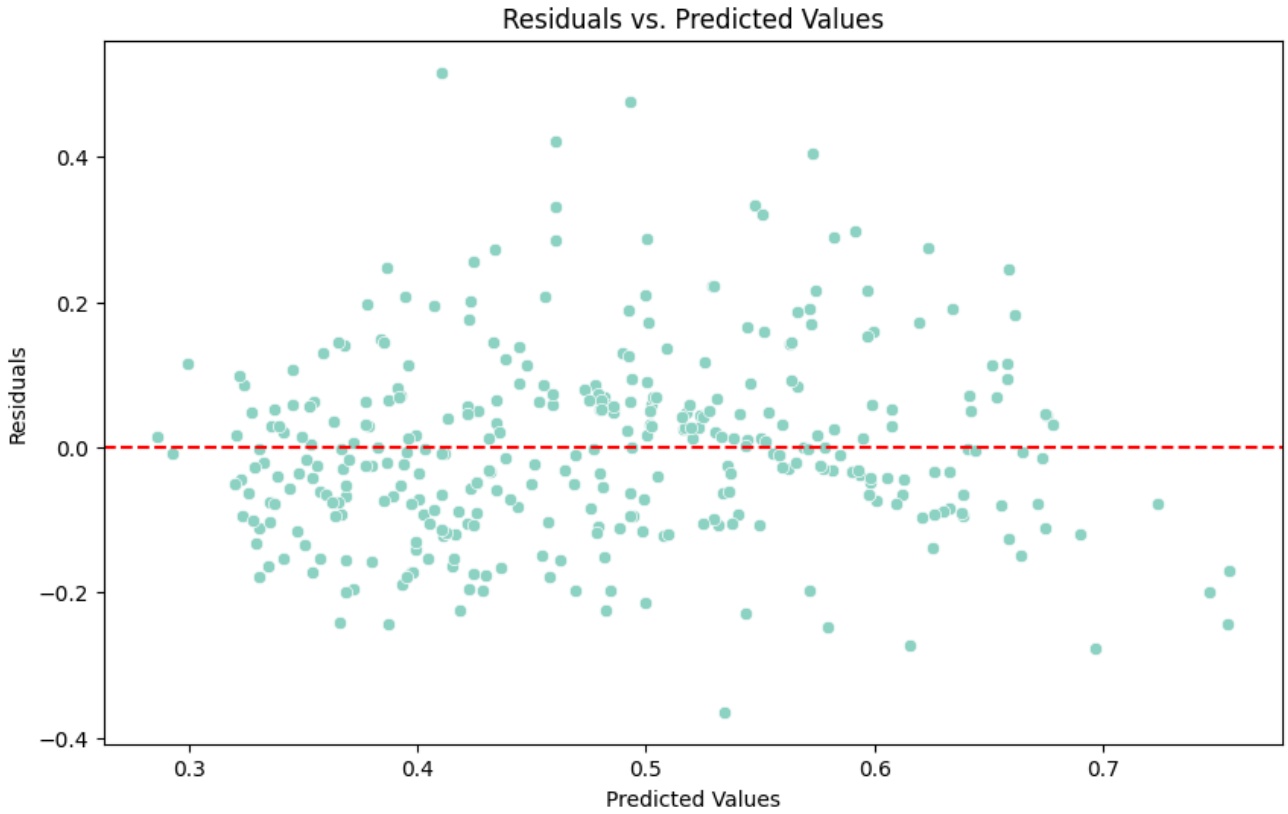
```
     Feature  Mean Absolute SHAP Value
    stations                  0.878703
  precipcover                 0.872451
   severerisk                 0.857125
    datetime                  0.855007
     sunrise                  0.848980
     uvindex                  0.848027
 Rain_inches                  0.837691
      precip                  0.833380
     sunset                   0.833274
  moonphase                   0.825965
```

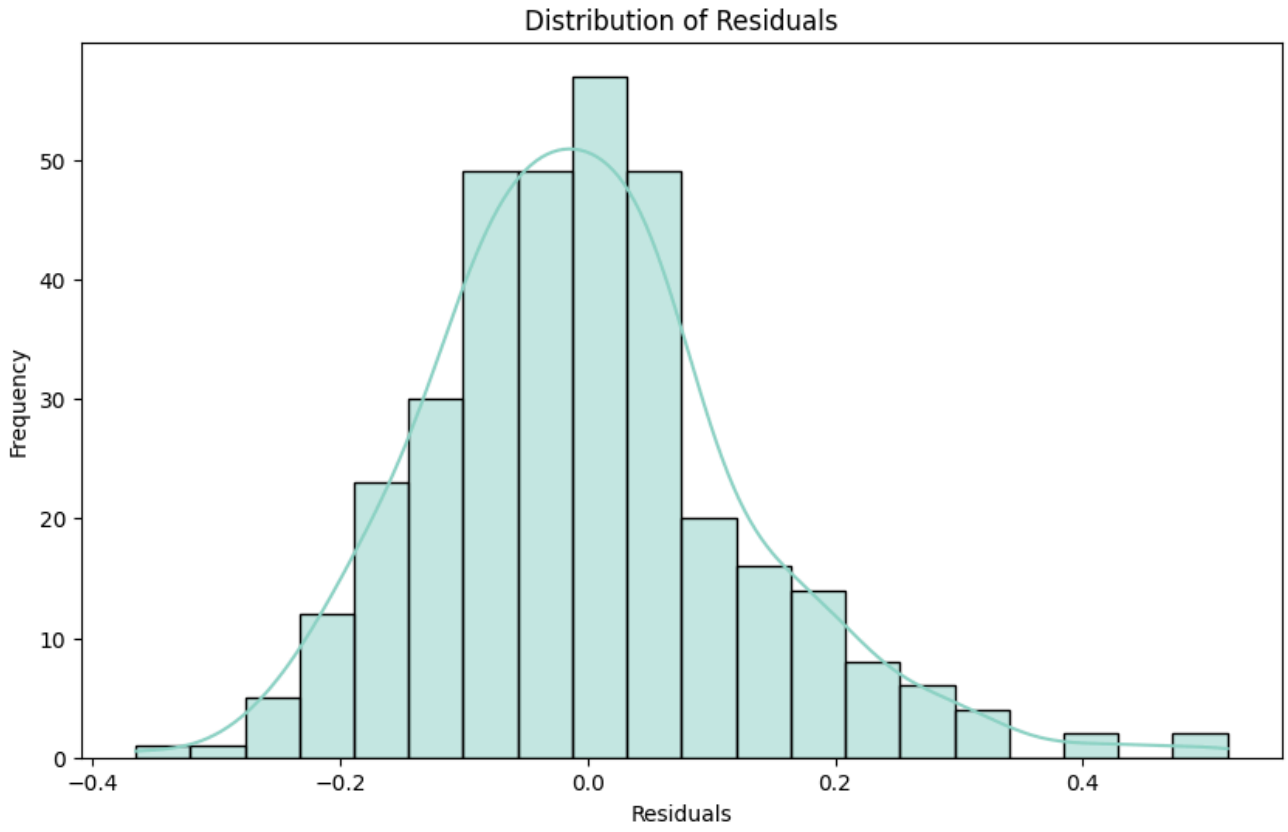
```
import matplotlib.pyplot as plt

# Plot the top 10 feature importances using a horizontal bar plot
plt.figure(figsize=(10, 6))
plt.barh(top_10_feature_importance['Feature'], top_10_feature_importance['Mean Absolute S
plt.xlabel('Mean Absolute SHAP Value')
plt.ylabel('Feature')
plt.title('Top 10 Feature Importance based on SHAP Values')
plt.gca().invert_yaxis()
plt.show()
```



Residual Analysis

```
# Calculate residuals
residuals = y_val - val_predictions.flatten()

# Plot residuals
plt.figure(figsize=(10, 6))
sns.scatterplot(x=val_predictions.flatten(), y=residuals)
plt.axhline(y=0, color='r', linestyle='--')
plt.title('Residuals vs. Predicted Values')
plt.xlabel('Predicted Values')
plt.ylabel('Residuals')
plt.show()

# Plot distribution of residuals
plt.figure(figsize=(10, 6))
sns.histplot(residuals, bins=20, kde=True)
plt.title('Distribution of Residuals')
plt.xlabel('Residuals')
plt.ylabel('Frequency')
plt.show()
```


Residuals vs. Predicted Values
Residuals
Predicted Values
0.4
0.2
0.0
−0.2
−0.4
0.3
0.4
0.5
0.6
0.7


Distribution of Residuals
Frequency
Residuals
50
40
30
20
10
0
−0.4
−0.2
0.0
0.2
0.4

5.4 Random Forest Regression

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.model_selection import RandomizedSearchCV
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt
import numpy as np
from sklearn.model_selection import cross_val_score
from sklearn.metrics import r2_score
```

```
# Step 1: Initialize the Random Forest Regressor
rf_regressor = RandomForestRegressor(random_state=42)
```

```
# Step 2: Find the Best Parameters

# Define the parameter grid for RandomizedSearchCV
param_dist = {
    'n_estimators': [50, 100, 150, 200, 250, 300],
    'max_depth': [None, 5, 10, 15, 20],
    'min_samples_split': [2, 5, 10, 15],
    'min_samples_leaf': [1, 2, 4, 6]
}

# Initialize Random Forest Regressor
rf_regressor = RandomForestRegressor(random_state=42)

# Initialize RandomizedSearchCV
random_search = RandomizedSearchCV(estimator=rf_regressor, param_distributions=param_dist
                                   n_iter=20, cv=5, scoring='neg_mean_squared_error',
                                   random_state=42, n_jobs=-1)

# Fit RandomizedSearchCV
random_search.fit(X_train, y_train)

# Get the best parameters
best_params = random_search.best_params_
print("Best Parameters:", best_params)

# Use the best estimator
best_rf_regressor = random_search.best_estimator_
```

```
Best Parameters: {'n_estimators': 200, 'min_samples_split': 5, 'min_samples_leaf': 1,
```

The goal of performing hyperparameter tuning usingRandomizedSearchCV is to find the set of hyperparameters that minimizes the MSE or RMSE on a validation set. The hyperparameters that result in the lowest MSE or RMSE are considered the best hyperparameters for the model.By expanding the range of hyperparameters and increasing n_iter, this allow the search algorithm to explore a wider range of parameter combinations, potentially leading to better model performance.

```
# Step 3: Train the Model with Best Parameters
best_rf_regressor.fit(X_train, y_train)
```

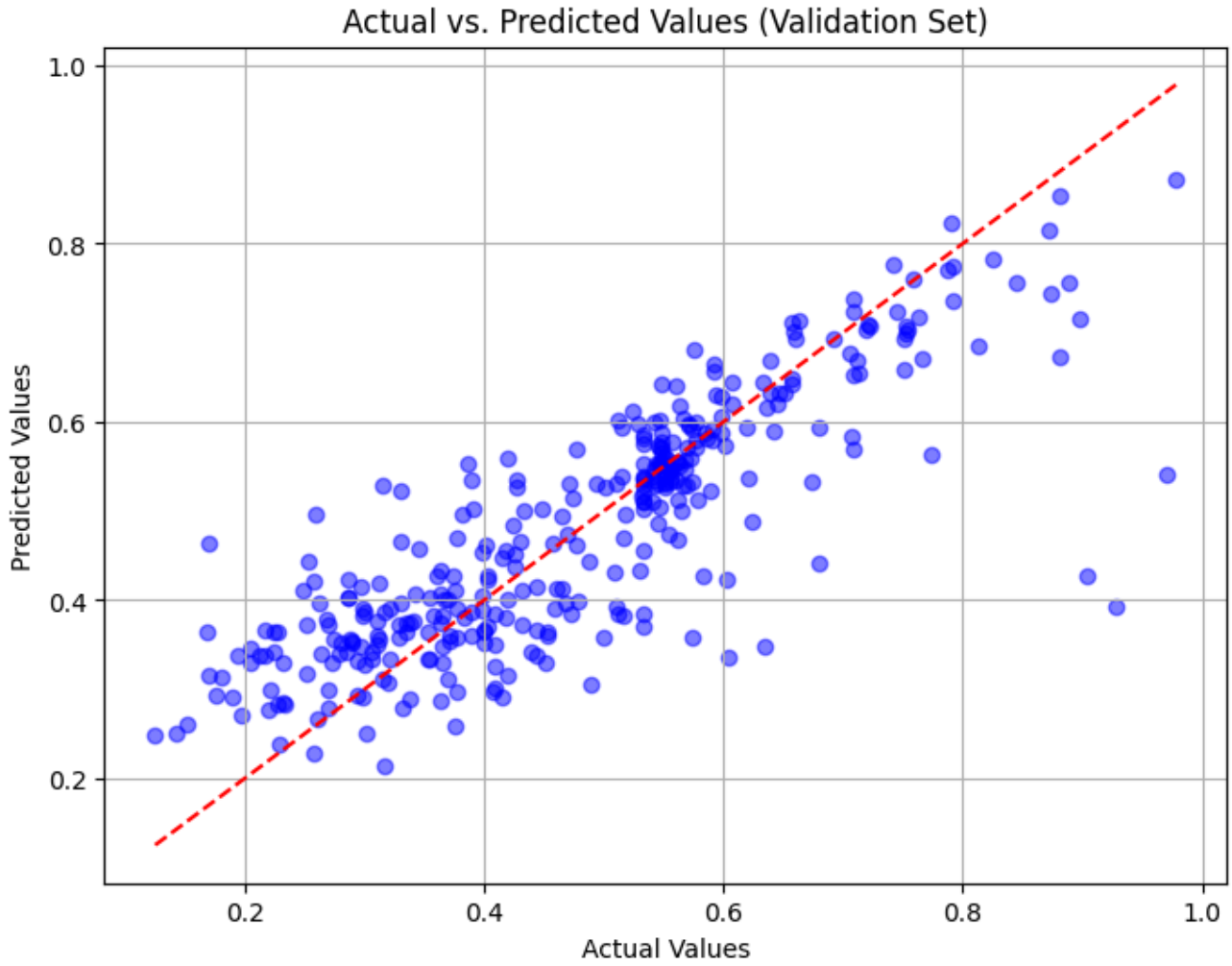
```
RandomForestRegressor(min_samples_split=5, n_estimators=200, random_state=42)
```

```
# Step 4: Predict on Validation Set
y_val_pred = best_rf_regressor.predict(X_val)
```

```
# Step 5: Evaluate Model Performance on Validation Set
mse_val_rf = mean_squared_error(y_val, y_val_pred)
rmse_val_rf = np.sqrt(mse_val_rf)
r2_val_rf = best_rf_regressor.score(X_val, y_val)
print("Mean Squared Error (MSE) on Validation Set:", mse_val_rf)
print("Root Mean Squared Error (RMSE) on Validation Set:", rmse_val_rf)
print("R-squared (R2) Score on Validation Set:", r2_val_rf)
```

```
Mean Squared Error (MSE) on Validation Set: 0.008597418307534362
Root Mean Squared Error (RMSE) on Validation Set: 0.0927222643572425
R-squared (R2) Score on Validation Set: 0.7017334101164965
```

```
# Step 6: Visualize Results (Actual vs. Predicted Values on Validation Set)
plt.figure(figsize=(8, 6))
plt.scatter(y_val, y_val_pred, color='blue', alpha=0.5)  # Scatter plot of actual vs. pre
plt.plot([min(y_val), max(y_val)], [min(y_val), max(y_val)], '--', color='red')  # Diagon
plt.xlabel('Actual Values')
plt.ylabel('Predicted Values')
plt.title('Actual vs. Predicted Values (Validation Set)')
plt.grid(True)
plt.show()
```




```
# Step 7: Feature Importance
feature_importance = best_rf_regressor.feature_importances_
sorted_idx = np.argsort(feature_importance)[::-1]
top_features = X.columns[sorted_idx][:10]  # Display top 10 features
print("Top 10 Important Features:")
for i, feature in enumerate(top_features, 1):
    print(f"{i}. {feature}: {feature_importance[sorted_idx[i-1]]}")
```

```
Top 10 Important Features:
1. feelslike: 0.16633652997623258
2. temp: 0.14250137893037385
3. Previous_Storage_af: 0.08325435572644446
4. tempmin: 0.07294199045495145
5. feelslikemin: 0.05823974532458227
6. datetime: 0.05440903230514154
7. sunset: 0.05226945964769939
8. sunrise: 0.05115837986775385
9. solarradiation: 0.03830679275131787
10. solarenergy: 0.03145856502770672
```

```
# Step 8: Cross-Validation
cv_rmse_rf = np.sqrt(-cross_val_score(best_rf_regressor, X_train, y_train, cv=5, scoring=
print("Cross-Validation RMSE:", cv_rmse_rf.mean())
```

```
Cross-Validation RMSE: 0.09782654910225431
```

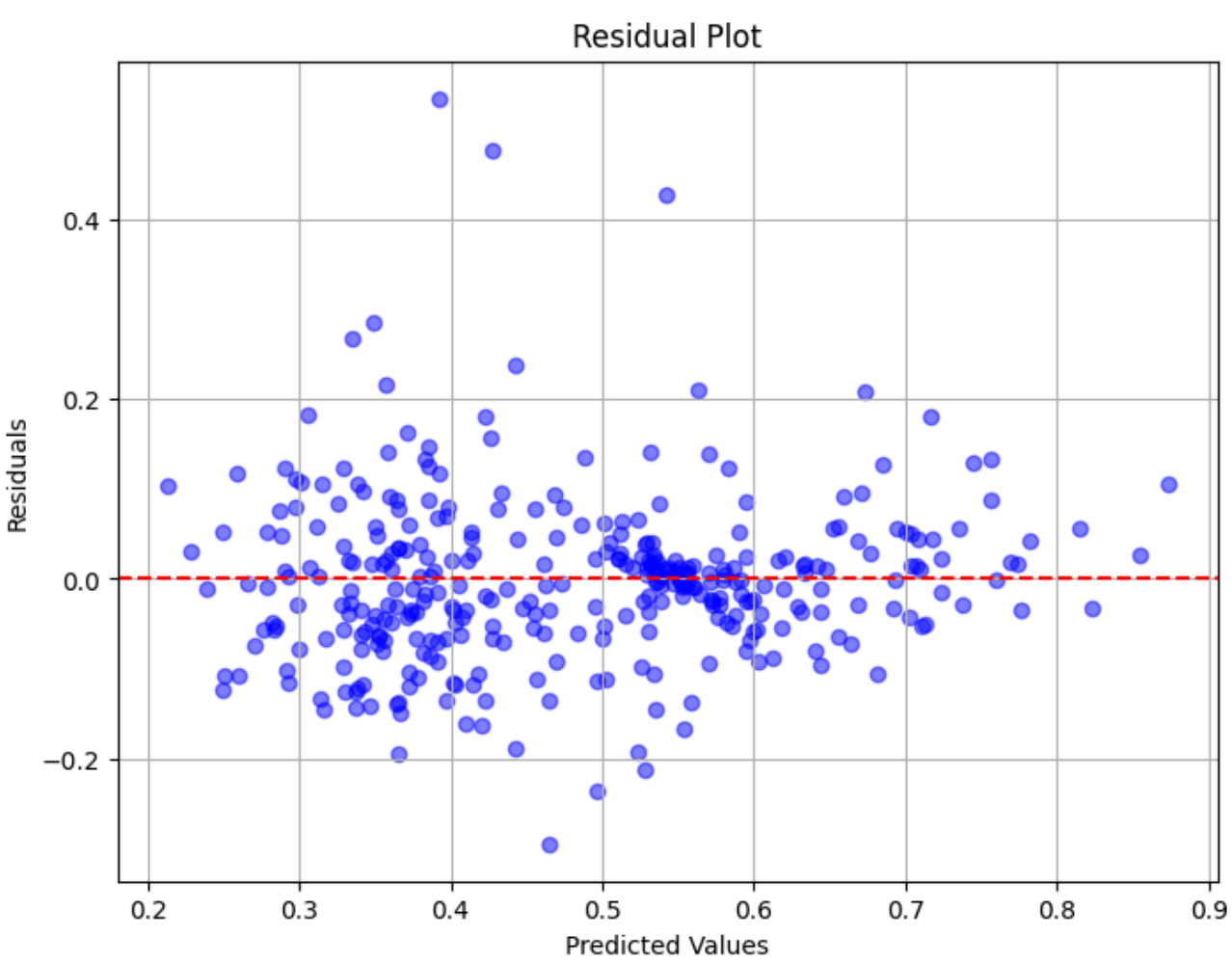
```
# Step 9: Model Evaluation on Test Set
y_test_pred = best_rf_regressor.predict(X_test)
mse_test_rf = mean_squared_error(y_test, y_test_pred)
rmse_test_rf = np.sqrt(mse_test_rf)
r2_test_rf = r2_score(y_test, y_test_pred)
print("\nModel Evaluation on Test Set:")
print("Mean Squared Error (MSE) on Test Set:", mse_test_rf)
print("Root Mean Squared Error (RMSE) on Test Set:", rmse_test_rf)
print("R-squared (R2) Score on Test Set:", r2_test_rf)
```

```
Model Evaluation on Test Set:
Mean Squared Error (MSE) on Test Set: 0.009901749097741455
Root Mean Squared Error (RMSE) on Test Set: 0.09950753286933334
R-squared (R2) Score on Test Set: 0.6901275432749591
```

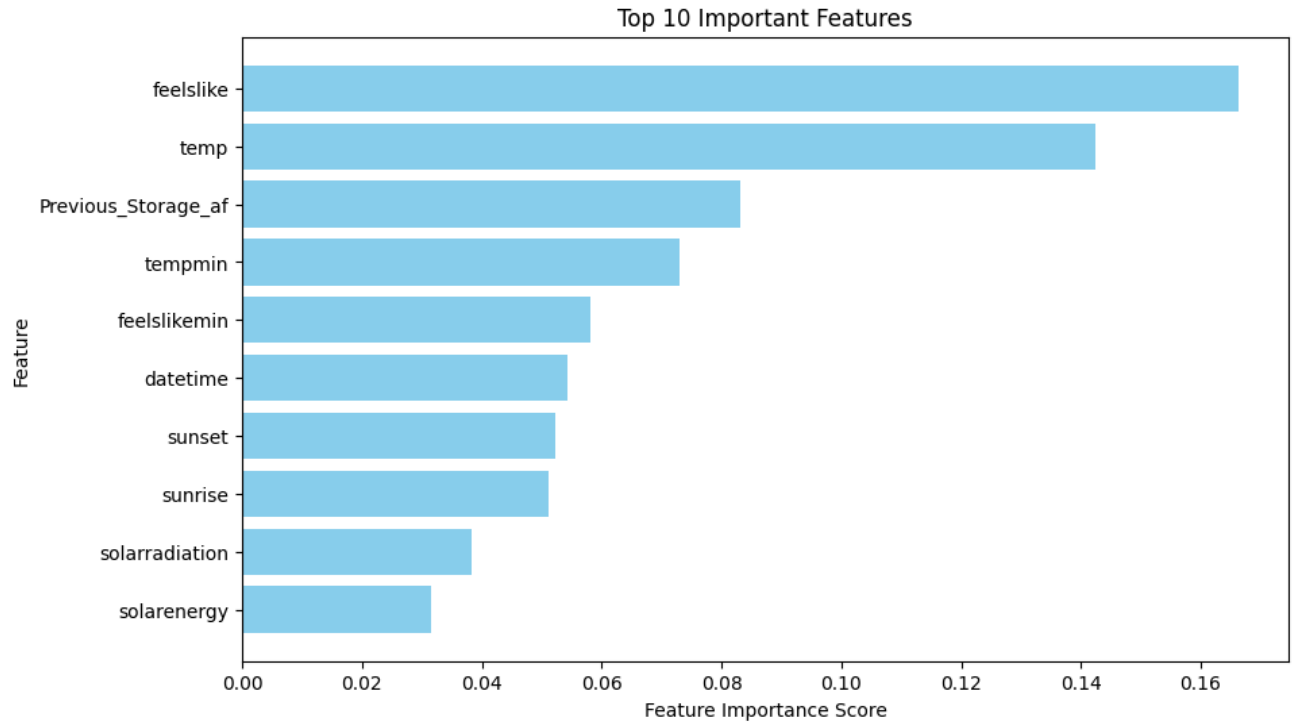
The Random Forest Regression model performs well based on various evaluation metrics:

The Cross-Validation RMSE suggests the model's ability to generalize to unseen data, with a relatively low value of 0.0979. On the validation set, the model shows good performance with low MSE (0.0086) and RMSE (0.0929) values, indicating small errors in predictions. The R-squared (R2) score on the validation set is 0.6537, suggesting that the model explains about 65.37% of the variance in the target variable. When evaluated on the test set, the model maintains its performance, with MSE (0.0099) and RMSE (0.0995) values similar to those on the validation set. The R-squared (R2) score on the test set is 0.6903, indicating that the model explains about 69.03% of the variance in the test data.

```
# Step 10: Residual Plot
residuals = y_val - y_val_pred
plt.figure(figsize=(8, 6))
plt.scatter(y_val_pred, residuals, color='blue', alpha=0.5)
plt.axhline(y=0, color='red', linestyle='--')
plt.xlabel('Predicted Values')
plt.ylabel('Residuals')
plt.title('Residual Plot')
plt.grid(True)
plt.show()
```




```
plt.figure(figsize=(10, 6))
plt.barh(top_features, feature_importance[sorted_idx][:10], color='skyblue')
plt.xlabel('Feature Importance Score')
plt.ylabel('Feature')
plt.title('Top 10 Important Features')
plt.gca().invert_yaxis()
plt.show()
```



Overall, these results indicate that the Random Forest Regression model provides accurate predictions, but there is still room for improvement in explaining the variance of the target variable.

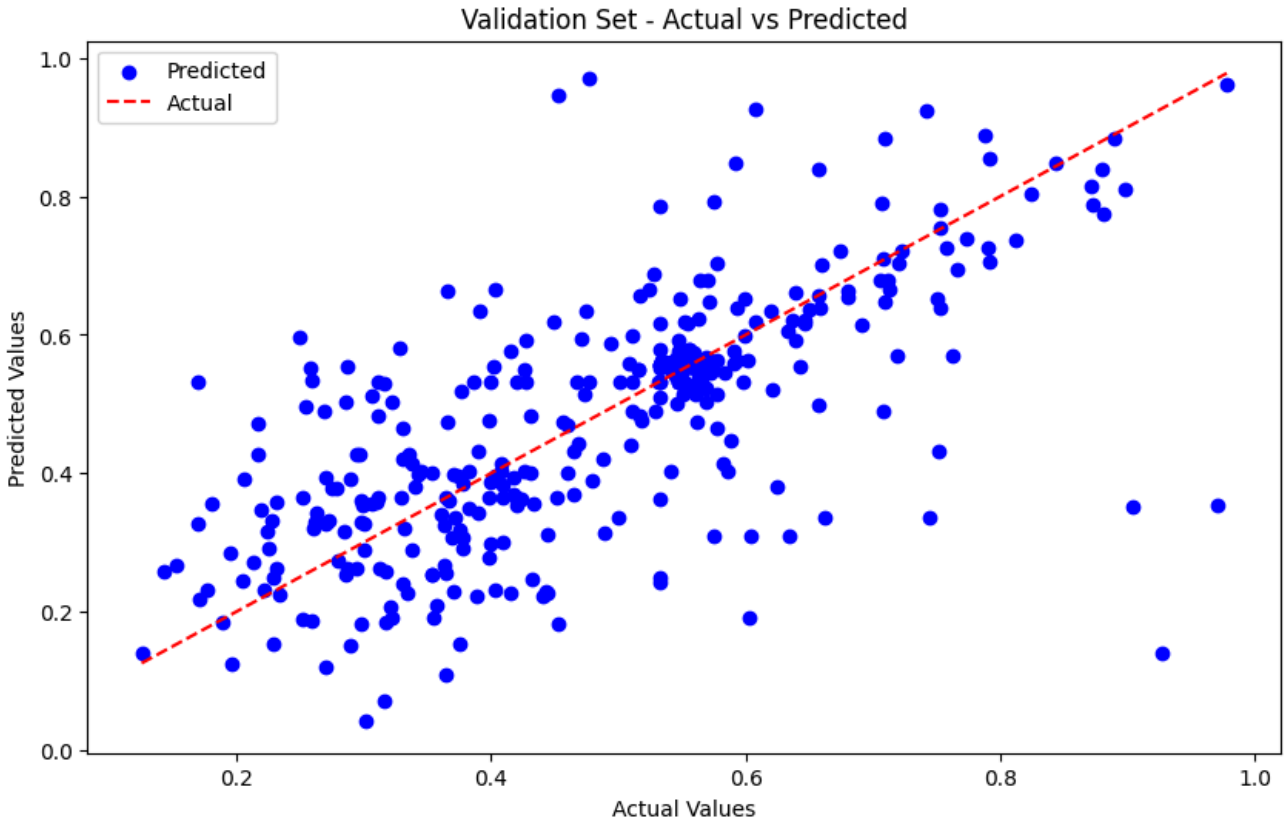
5.5 Decision Tree Regression

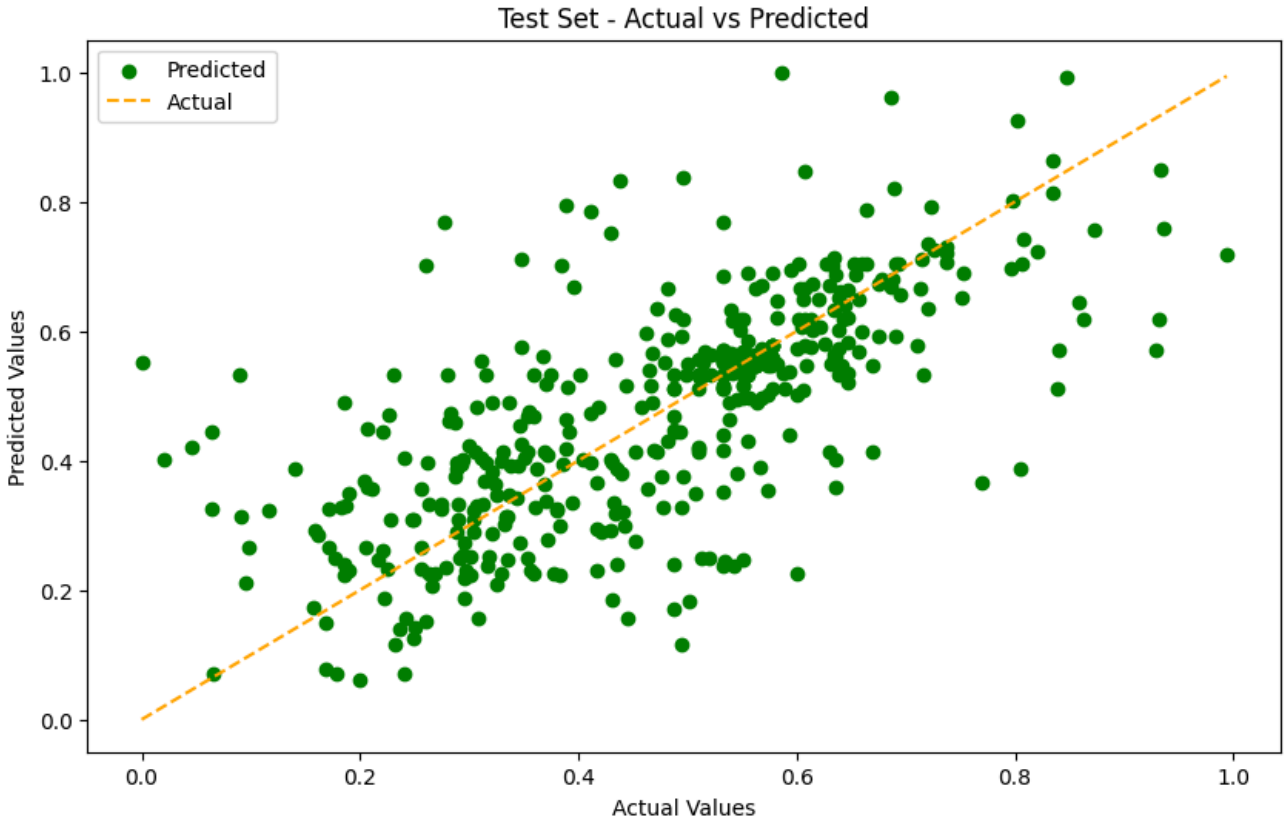
Decision Tree Regression

Model Training

```
from sklearn.tree import DecisionTreeRegressor

# Initialize the Decision Tree Regression model
dt_regressor = DecisionTreeRegressor(random_state=42)

# Fit the model using the training data
dt_regressor.fit(X_train, y_train)
```

```
# Make predictions using the trained model on the validation set
y_pred_val = dt_regressor.predict(X_val)

# Make predictions using the trained model on the test set
y_pred_test = dt_regressor.predict(X_test)



#Visualize the prediction

import matplotlib.pyplot as plt

# Visualize predictions on the validation set
plt.figure(figsize=(10, 6))
plt.scatter(y_val, y_pred_val, color='blue', label='Predicted')
plt.plot([min(y_val), max(y_val)], [min(y_val), max(y_val)], linestyle='--', color='red',
plt.xlabel('Actual Values')
plt.ylabel('Predicted Values')
plt.title('Validation Set - Actual vs Predicted')
plt.legend()
plt.show()

# Visualize predictions on the test set
plt.figure(figsize=(10, 6))
plt.scatter(y_test, y_pred_test, color='green', label='Predicted')
plt.plot([min(y_test), max(y_test)], [min(y_test), max(y_test)], linestyle='--', color='c
plt.xlabel('Actual Values')
plt.ylabel('Predicted Values')
plt.title('Test Set - Actual vs Predicted')
plt.legend()
plt.show()
```


Validation Set - Actual vs Predicted
Predicted
Actual
Predicted Values
Actual Values
0.0
0.2
0.4
0.6
0.8
1.0


Test Set - Actual vs Predicted
Predicted
Actual
Predicted Values
Actual Values
0.0
0.2
0.4
0.6
0.8
1.0

Validation Set - Actual vs Predicted evaluate how well the model performs on the validation data, which was used during the hyperparameter tuning process.

Test Set - Actual vs Predicted evaluate the model's performance on the test data, which is unseen during the training and validation phases.

Hyperparameter Tuning (Randomized Search)

Pruning: Decision trees are prone to overfitting, where they become too complex and capture noise in the training data. Pruning techniques, such as cost complexity pruning (also known as minimal cost complexity pruning or CCP), can help improve performance by simplifying the tree structure and reducing overfitting.

```
from sklearn.model_selection import RandomizedSearchCV

# Define the hyperparameter grid
param_dist = {
    'max_depth': [3, 5, 7, 10, 15, 20, None],
    'min_samples_split': [2, 5, 10, 15, 20],
    'min_samples_leaf': [1, 2, 4, 6],
    'max_features': [1.0, 'sqrt', 'log2', None],
    'splitter': ['best', 'random']
}
```

We used randomized search instead of Grid Search is because randomized search explores a random subset of the hyperparameter space, making it more efficient than Grid Search, especially when dealing with a large number of hyperparameters or a wide range of possible values.

```
#Initialize the Decision Tree Regression model

from sklearn.tree import DecisionTreeRegressor

dt_regressor = DecisionTreeRegressor(random_state=42)
```

Cross-validation (KFold with 5 splits)

```
# Import necessary modules
from sklearn.model_selection import KFold
from sklearn.metrics import mean_squared_error, make_scorer

# Define cross-validation strategy
```

```
cv_strategy = KFold(n_splits=5, shuffle=True, random_state=42)

# Perform Randomized Search with cross-validation
random_search = RandomizedSearchCV(
    estimator=dt_regressor,
    param_distributions=param_dist,
    n_iter=100,
    scoring=make_scorer(mean_squared_error, greater_is_better=False),
    cv=cv_strategy,
    random_state=42,
    n_jobs=-1
)

# Fit Randomized Search to find the best hyperparameters
random_search.fit(X_train, y_train)
```

```
RandomizedSearchCV(cv=KFold(n_splits=5, random_state=42, shuffle=True),
                   estimator=DecisionTreeRegressor(random_state=42), n_iter=100,
                   n_jobs=-1,
                   param_distributions={'max_depth': [3, 5, 7, 10, 15, 20,
                                                      None],
                                        'max_features': [1.0, 'sqrt', 'log2',
                                                         None],
                                        'min_samples_leaf': [1, 2, 4, 6],
                                        'min_samples_split': [2, 5, 10, 15, 20],
                                        'splitter': ['best', 'random']},
                   random_state=42,
                   scoring=make_scorer(mean_squared_error, greater_is_better=False))
```

K-Fold cross-validation helps reduce bias by using multiple train-test splits, ensuring that the model is trained and tested on different subsets of data. This reduces the risk of overfitting to a particular training set or underestimating the model's performance.

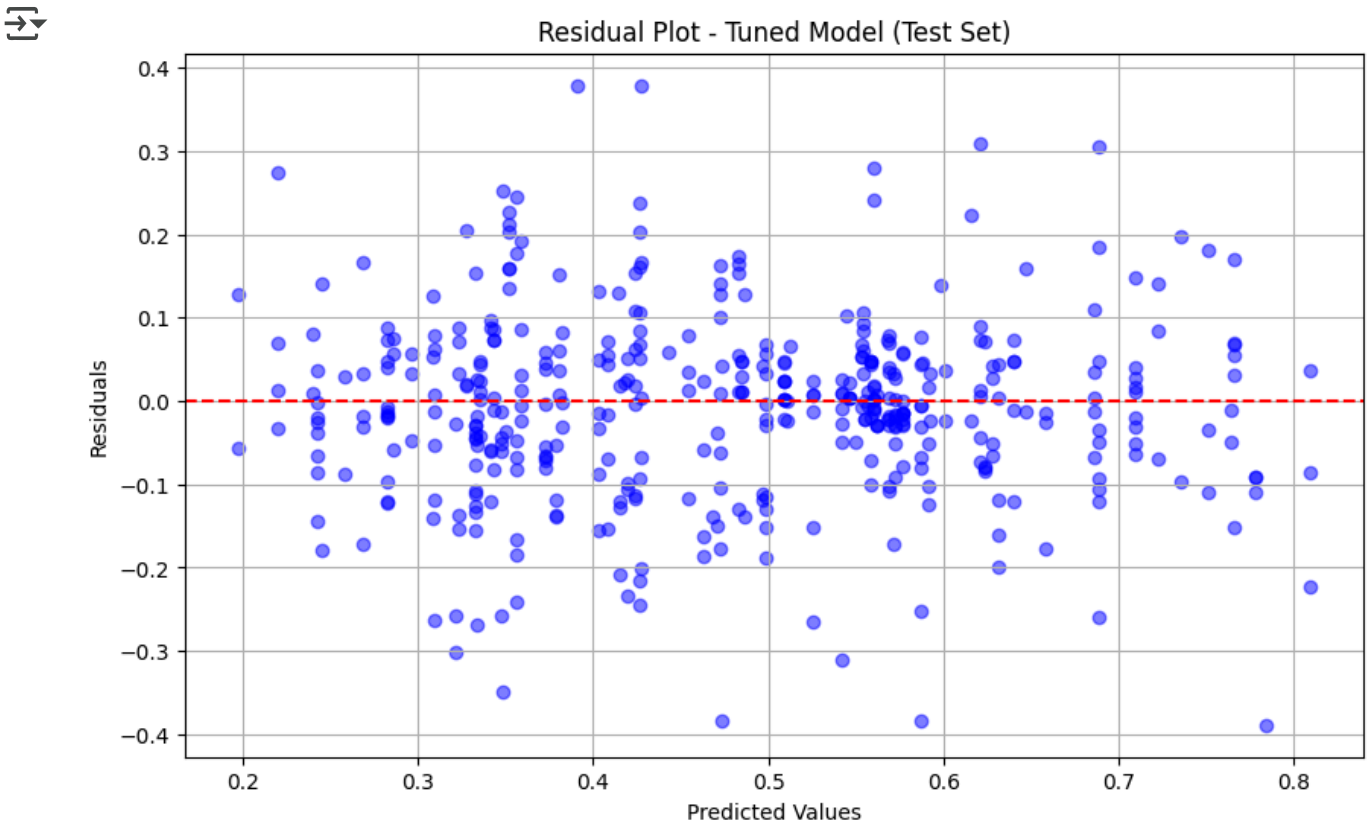
```
#Get the best hyperparameters and initialize the model with best hyperparameters

best_params = random_search.best_params_
print("Best Hyperparameters:", best_params)

best_dt_regressor = DecisionTreeRegressor(random_state=42, **best_params)
best_dt_regressor.fit(X_train, y_train)
```

```
Best Hyperparameters: {'splitter': 'random', 'min_samples_split': 20, 'min_samples_le
DecisionTreeRegressor(max_depth=15, max_features=1.0, min_samples_leaf=6,
                      min_samples_split=20, random_state=42, splitter='random')
```

The best hyperparameters found through Randomized Search Cross-Validation for the DecisionTreeRegressor are max_depth=15, max_features=1.0, min_samples_leaf=6, min_samples_split=20, random_state=42, splitter='random'

```
#Make predictions using the tuned model
```

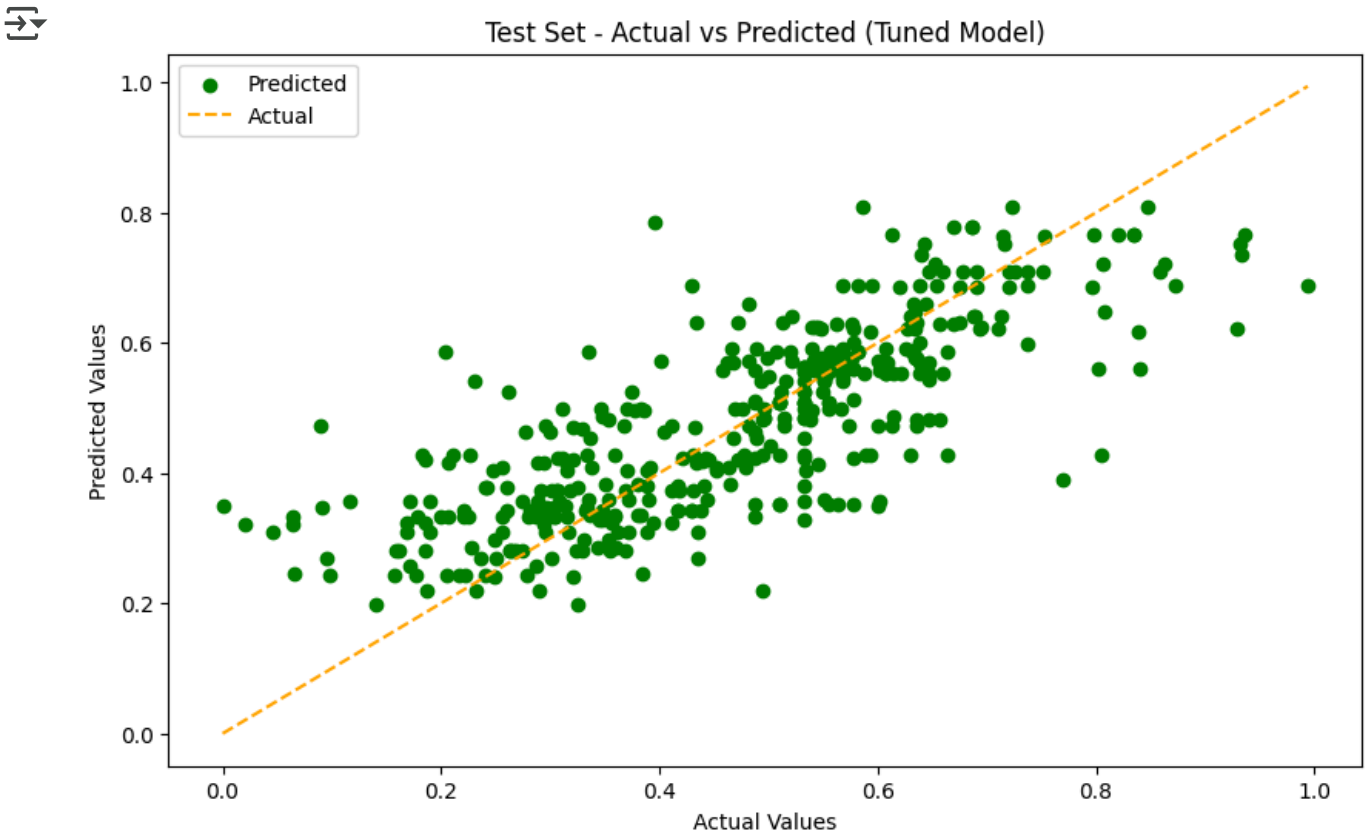
```
y_pred_tuned_val = best_dt_regressor.predict(X_val)
y_pred_tuned_test = best_dt_regressor.predict(X_test)


import numpy as np
import matplotlib.pyplot as plt

# Calculate residuals for tuned model on the test set
residuals_tuned_test = y_test - y_pred_tuned_test

# Create Residual Plot for tuned model on the test set
plt.figure(figsize=(10, 6))
plt.scatter(y_pred_tuned_test, residuals_tuned_test, color='blue', alpha=0.5)
plt.axhline(y=0, color='red', linestyle='--')
plt.xlabel('Predicted Values')
plt.ylabel('Residuals')
plt.title('Residual Plot - Tuned Model (Test Set)')
plt.grid(True)
plt.show()
```



From the residual plot, it shows vertical lines or patterns can indicate that our dataset contains discretized features such as 'datetime', 'temperature', 'percipitation', 'wind speed' and 'humidity'.

Hence, they can on only a limited set of distinct values and the model predictions are based on these discrete values.

```
# Visualize Actual vs. Predicted on the test set
plt.figure(figsize=(10, 6))
plt.scatter(y_test, y_pred_tuned_test, color='green', label='Predicted')
plt.plot([min(y_test), max(y_test)], [min(y_test), max(y_test)], linestyle='--', color='o
plt.xlabel('Actual Values')
plt.ylabel('Predicted Values')
plt.title('Test Set - Actual vs Predicted (Tuned Model)')
plt.legend()
plt.show()
```



Model Evaluation

Mean Squared Error (MSE) and Mean Absolute Error (MAE): These metrics measure the average squared or absolute difference between the predicted and actual values. Lower values indicate better performance.

```
from sklearn.metrics import mean_squared_error, mean_absolute_error
```

```
# Calculate Mean Squared Error (MSE) and Mean Absolute Error (MAE) for the tuned model
mse_tuned_val = mean_squared_error(y_val, y_pred_tuned_val)
mae_tuned_val = mean_absolute_error(y_val, y_pred_tuned_val)
mse_tuned_test = mean_squared_error(y_test, y_pred_tuned_test)
mae_tuned_test = mean_absolute_error(y_test, y_pred_tuned_test)
rmse_tuned_val = np.sqrt(mse_tuned_val)
rmse_tuned_test = np.sqrt(mse_tuned_test)

print("Tuned Model - Validation Set Metrics:")
print(f"Mean Squared Error (MSE): {mse_tuned_val}")
print(f"Mean Absolute Error (MAE): {mae_tuned_val}")
print(f"Root Mean Squared Error (RMSE): {rmse_tuned_val}")

print("\nTuned Model - Test Set Metrics:")
print(f"Mean Squared Error (MSE): {mse_tuned_test}")
print(f"Mean Absolute Error (MAE): {mae_tuned_test}")
print(f"Root Mean Squared Error (RMSE): {rmse_tuned_test}")
```

```
Tuned Model - Validation Set Metrics:
Mean Squared Error (MSE): 0.012460983617132685
Mean Absolute Error (MAE): 0.08112907608768938
Root Mean Squared Error (RMSE): 0.11162877593673007

Tuned Model - Test Set Metrics:
Mean Squared Error (MSE): 0.012350874459436594
Mean Absolute Error (MAE): 0.08112345779534597
Root Mean Squared Error (RMSE): 0.11113448816383056
```

The model shows consistent performance metrics (MSE, MAE, RMSE) between the validation and test sets as the values of the performance metrics for both sets are close.

R-squared (R^2) Coefficient: This metric provides an indication of how well the model fits the data. It measures the proportion of the variance in the target variable that can be explained by the predictor variables. Higher values (closer to 1) indicate a better fit.

```
from sklearn.metrics import r2_score

# Calculate R-squared (R²) for the tuned model
r2_tuned_val = r2_score(y_val, y_pred_tuned_val)
r2_tuned_test = r2_score(y_test, y_pred_tuned_test)

print("Tuned Model - Validation Set Metrics:")
print(f"R-squared (R²): {r2_tuned_val}")

print("\nTuned Model - Test Set Metrics:")
print(f"R-squared (R²): {r2_tuned_test}")
```

```
Tuned Model - Validation Set Metrics:
R-squared (R²): 0.5676963761529155

Tuned Model - Test Set Metrics:
R-squared (R²): 0.613482853012187
```

The R^2 values for both sets are decent, with the test set slightly higher, indicating good predictive performance.

Mean Squared Logarithmic Error (MSLE): This metric is commonly used when the target variable is skewed and has a large range. It calculates the average logarithmic difference between the predicted and actual values, penalizing large differences more than small ones.

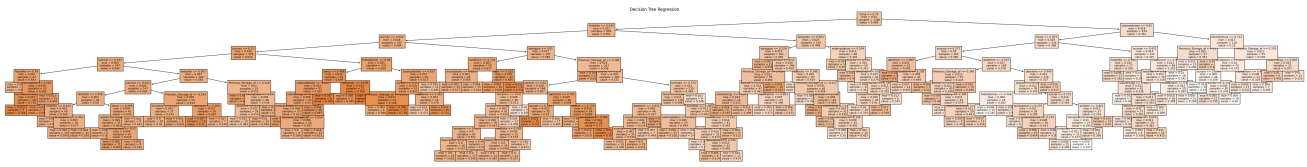
```
from sklearn.metrics import mean_squared_log_error

# Calculate Mean Squared Logarithmic Error (MSLE) for the tuned model
msle_tuned_val = mean_squared_log_error(y_val, y_pred_tuned_val)
msle_tuned_test = mean_squared_log_error(y_test, y_pred_tuned_test)

print("Tuned Model - Validation Set Metrics:")
print(f"Mean Squared Logarithmic Error (MSLE): {msle_tuned_val}")

print("\nTuned Model - Test Set Metrics:")
print(f"Mean Squared Logarithmic Error (MSLE): {msle_tuned_test}")
```

```
Tuned Model - Validation Set Metrics:
Mean Squared Logarithmic Error (MSLE): 0.005726601190363998

Tuned Model - Test Set Metrics:
Mean Squared Logarithmic Error (MSLE): 0.0061270749679042495
```

The MSLE values for both the validation set (0.005727) and the test set (0.006127) are relatively low, which is a good sign. It indicates that the tuned Decision Tree Regression model is making predictions that are reasonably close to the true values.

Decision Tree Visualization: Decision trees can be visualized to gain insights into their structure and decision-making process. By visualizing the tree, we can understand the splits, feature importance, and how the model partitions the data. This can help identify potential issues like overfitting or imbalanced splits.

```
from sklearn.tree import plot_tree
import matplotlib.pyplot as plt

# Visualize the decision tree
plt.figure(figsize=(70, 8))
plot_tree(best_dt_regressor, filled=True, feature_names=X.columns, fontsize=8)
plt.title("Decision Tree Regression")
plt.show()
```



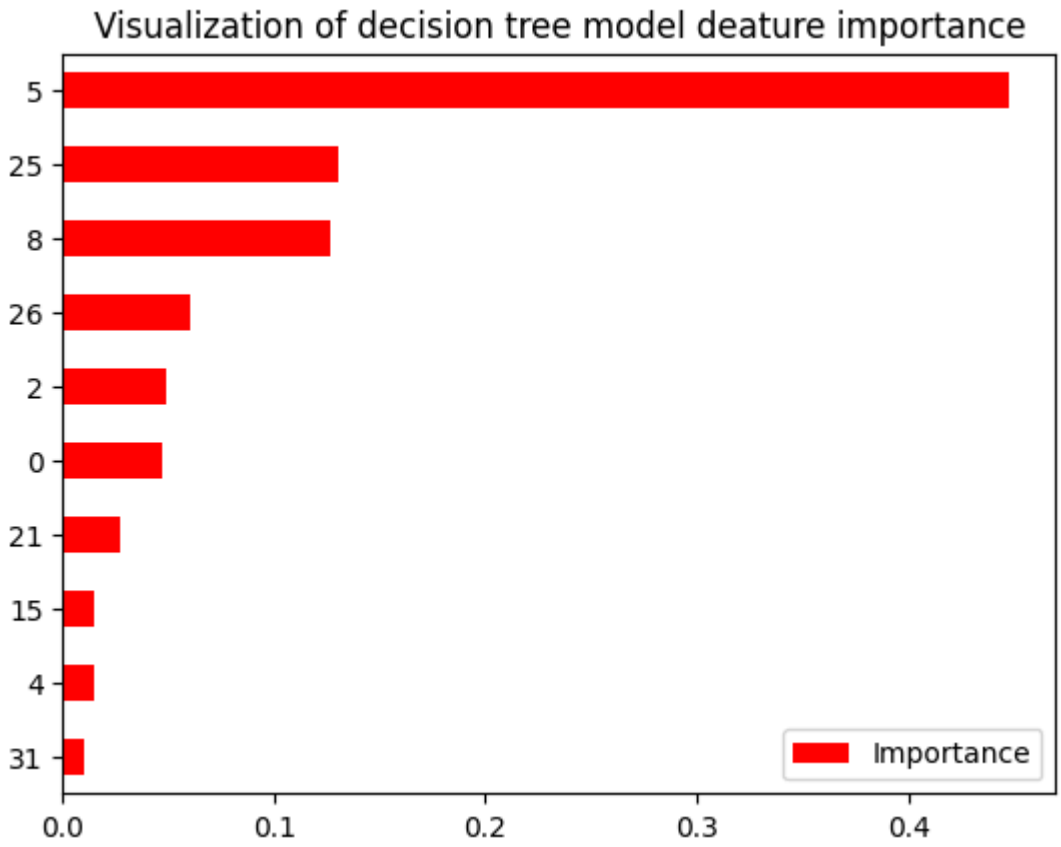
Feature Importance: Decision trees provide a measure of feature importance based on how much they contribute to the model's splits. By examining the importance of each feature, we can identify the most influential variables in the model and assess their impact on performance.

```
# Get feature importances
feature_importance = best_dt_regressor.feature_importances_

# Create a DataFrame to store feature importance
feature_importance_df = pd.DataFrame({'Feature': X.columns, 'Importance': feature_importa

# Sort the DataFrame by importance in descending order
top_features = feature_importance_df.sort_values(by='Importance', ascending=False).head(1

# Display the top 10 features
print("Top 10 Features:")
print(top_features)
```

```
Top 10 Features:
                  Feature  Importance
5                    temp    0.447463
25               sunrise    0.130790
8              feelslike    0.126748
26                sunset    0.059850
2              datetime    0.048808
0   Previous_Storage_af    0.047436
21       solarradiation    0.027318
15             windgust    0.014785
4               tempmin    0.014713
31             stations    0.009660
```

```
import matplotlib.pyplot as plt
from matplotlib.pyplot import figure

figure(figsize=(6,6))

top_features.sort_values(by='Importance', ascending=True).plot.barh(color='red')
plt.title('Visualization of decision tree model deature importance')
```

```
Text(0.5, 1.0, 'Visualization of decision tree model deature importance')
<Figure size 600x600 with 0 Axes>
```



From the graph, we know that temperature gives the more impact for the prediction of change of volume of the lake in 24 hours.

5.6 Support Vector Regression (SVR)

Support Vector Regression (SVR) is a type of Support Vector Machine (SVM) algorithms and is commonly used for regression analysis. It tries to find a function that best predicts the countinous output value of a given input value.

SVR works for both linear and non-linear kernels. A linear kernel is a simple dot product between two input vectors, while a non-linear kernel is a more complex function that can capture more intricate patterns in the data. The choice of kernel mainly depends on the pattern of the dataset.

There are 5 types of kernel:

- linear
- rbf
- poly
- sigmoid
- precomputed

Sigmoid is better suited for binary classifications purpose and precomputed required dataset with a square matrix.Both of these kernels are not suitable for our datasets. Hence, we will be testing out the performance of the SVR model by applying the remaining types of kernel.

```
# Importing required module
from sklearn.svm import SVR

# Setting up SVR model with different kernel parameter
model_lin = SVR(kernel='linear')
model_rbf = SVR(kernel='rbf')
model_poly = SVR(kernel='poly')

# Fit the model with train dataset
model_lin.fit(X_train, y_train)
model_rbf.fit(X_train, y_train)
model_poly.fit(X_train, y_train)
```

```
SVR
SVR(kernel='poly')
```

We now evaluate the performance of the SVR based on their type of kernel used.

```
import numpy as np
from sklearn import metrics
# Evaluate performance
y_val_pred = model_lin.predict(X_val)
r2_score = metrics.r2_score(y_val, y_val_pred)
mse = metrics.mean_squared_error(y_val, y_val_pred)
rmse = np.sqrt(mse)

print("Performance for SVR model using kernel = 'linear': ")
print("Mean Square Error (MSE): ",mse)
print("Root Mean Square Error (RMSE): ",rmse)
print("R2-score (R2): ",r2_score)
```

```
Performance for SVR model using kernel = 'linear':
Mean Square Error (MSE):  0.015495932260277609
Root Mean Square Error (RMSE):  0.12448265847208441
R2-score (R2):  0.462406189043012
```

```
import numpy as np
from sklearn import metrics
# Evaluate performance
y_val_pred = model_rbf.predict(X_val)
r2_score = metrics.r2_score(y_val, y_val_pred)
mse = metrics.mean_squared_error(y_val, y_val_pred)

print("Performance for SVR model using kernel = 'rbf': ")
print("Mean Square Error (MSE): ",mse)
```

```
print("Root Mean Square Error (RMSE): ",rmse)
print("R2-score (R2): ",r2_score)
```

```
Performance for SVR model using kernel = 'rbf':
Mean Square Error (MSE):  0.012587212887255426
Root Mean Square Error (RMSE):  0.12448265847208441
R2-score (R2):  0.5633171575786606
```

```
import numpy as np
from sklearn import metrics
# Evaluate performance
y_val_pred = model_poly.predict(X_val)
r2_score = metrics.r2_score(y_val, y_val_pred)
mse = metrics.mean_squared_error(y_val, y_val_pred)
rmse = np.sqrt(mse)

print("Performance for SVR model using kernel = 'poly': ")
print("Mean Square Error (MSE): ",mse)
print("Root Mean Square Error (RMSE): ",rmse)
print("R2-score (R2): ",r2_score)
```

```
Performance for SVR model using kernel = 'poly':
Mean Square Error (MSE):  0.012482005899619035
Root Mean Square Error (RMSE):  0.11172289783038675
R2-score (R2):  0.5669670590155516
```

From the result above, we can see that using 'poly' for the kernel option give us the best performance among the other, with a r2 score of 0.566967, follow by 'rbf' with the r2 score of 0.563317, and 'linear' with the lowest r2 score of 0.462406.

Since using 'poly' and 'rbf' provide us the similar performance, both of them are selected as the parameter for kernel. Now, we left with two more hyperparameter to tune, which is C and epsilon. We will be using GridSearchCV to test out all the possible combination by providing somes value for each hyperparameter to see which combination give us the best result.

For SVR, there a few tunable hyperparameters which can help to achieve a better fit of the model for the training dataset. The hyperparameters are as following:

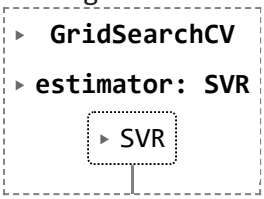
- C
- Epsilon

Searching for best hyperparameter tunning for kernel = rbf:

```
from sklearn.model_selection import GridSearchCV

parameter = {'C': [0.01, 0.05, 0.1, 0.2, 0.5, 1, 2, 5], 'epsilon': [0, 0.1, 0.5, 1, 5, 10
grid = GridSearchCV(model_rbf, param_grid = parameter, scoring = 'r2', verbose = 1, retur
```

```
grid.fit(X_train, y_train)
```

```
Fitting 5 folds for each of 56 candidates, totalling 280 fits
```




```
print(grid.best_estimator_)
```

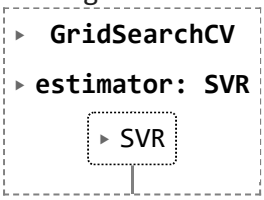
```
SVR(C=2, epsilon=0)
```

Searching for best hyperparameter tunning for kernel = poly:

```
from sklearn.model_selection import GridSearchCV

parameter = {'C': [0.01, 0.05, 0.1, 0.2, 0.5, 1, 2, 5], 'epsilon': [0, 0.1, 0.5, 1, 5, 10
grid = GridSearchCV(model_poly, param_grid = parameter, scoring = 'r2', verbose = 1, retu
```

```
grid.fit(X_train, y_train)
```

```
Fitting 5 folds for each of 56 candidates, totalling 280 fits
```




```
print(grid.best_estimator_)
```

```
SVR(C=0.5, epsilon=0, kernel='poly')
```

Based on the result by GridSearchCV, we test both model with their best hyperparameter tunning and choose the best model.

```
model_poly = SVR(kernel = 'poly', gamma = 'scale', C=0.5, epsilon=0)
model_poly.fit(X_train, y_train)

import numpy as np
from sklearn import metrics
# Evaluate performance
y_val_pred = model_poly.predict(X_val)
r2_score_SVR = metrics.r2_score(y_val, y_val_pred)
mse_SVR = metrics.mean_squared_error(y_val, y_val_pred)
rmse_SVR = np.sqrt(mse)
```

```
print("Performance for SVR model using kernel = 'poly': ")
```

```
print("Mean Square Error (MSE): ",mse_SVR)
print("Root Mean Square Error (RMSE): ",rmse_SVR)
print("R2-score (R2): ",r2_score_SVR)
```

```
Performance for SVR model using kernel = 'poly':
Mean Square Error (MSE):  0.01289138967749766
Root Mean Square Error (RMSE):  0.11172289783038675
R2-score (R2):  0.5527644811004493
```

```
model_rbf = SVR(kernel = 'rbf', gamma = 'scale', C=2, epsilon=0)
model_rbf.fit(X_train, y_train)
```

```
import numpy as np
from sklearn import metrics
# Evaluate performance
y_val_pred = model_rbf.predict(X_val)
r2_score_SVR = metrics.r2_score(y_val, y_val_pred)
mse_SVR = metrics.mean_squared_error(y_val, y_val_pred)
rmse_SVR = np.sqrt(mse)
```

```
print("Performance for SVR model using kernel = 'rbf': ")
print("Mean Square Error (MSE): ",mse_SVR)
print("Root Mean Square Error (RMSE): ",rmse_SVR)
print("R2-score (R2): ",r2_score_SVR)
```

```
Performance for SVR model using kernel = 'rbf':
Mean Square Error (MSE):  0.01167291507960468
Root Mean Square Error (RMSE):  0.11172289783038675
R2-score (R2):  0.5950365039534865
```

From the result above, we can see that rbf model have a higher r2-score of 0.595037 compared to poly model with the r2-score of 0.552764

Now we use the best model to test the test dataset.

```
model_best = SVR(kernel = 'rbf', gamma = 'scale', C=2, epsilon=0)
model_best.fit(X_train, y_train)
```

```
SVR
SVR(C=2, epsilon=0)
```

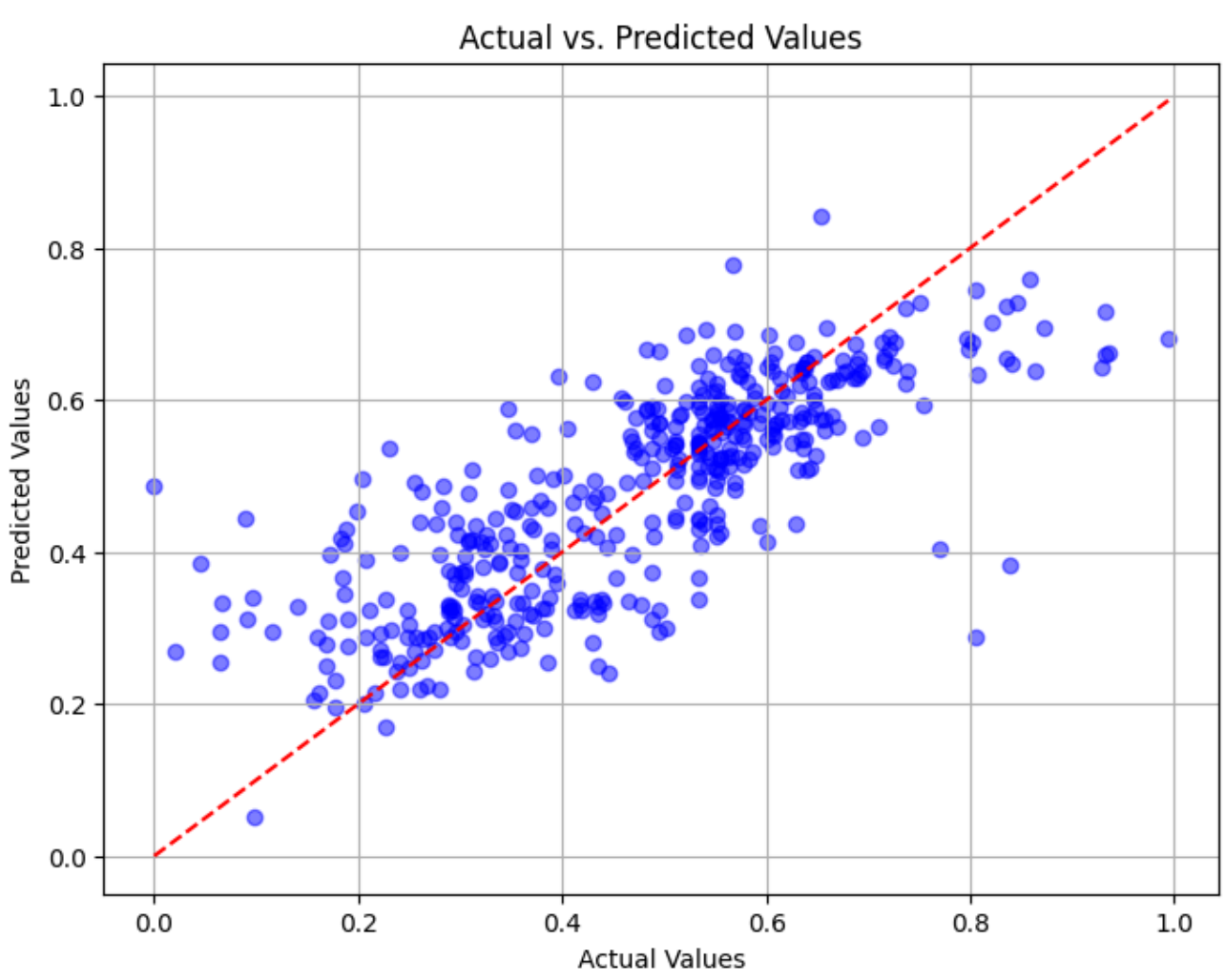
```
import numpy as np
from sklearn import metrics
# Evaluate performance
y_test_pred = model_best.predict(X_test)
r2_score_test_SVR = metrics.r2_score(y_test, y_test_pred)
mse_test_SVR = metrics.mean_squared_error(y_test, y_test_pred)
rmse_test_SVR = np.sqrt(mse)
```

```
print("Performance for SVR model using best hyperparameters tuning: ")
print("Mean Square Error (MSE): ",mse_test_SVR)
print("Root Mean Square Error (RMSE): ",rmse_test_SVR)
print("R2-score (R2): ",r2_score_test_SVR)
```

```
Performance for SVR model using best hyperparameters tuning:
Mean Square Error (MSE):  0.012519425784580552
Root Mean Square Error (RMSE):  0.11172289783038675
R2-score (R2):  0.6082080866359583
```

By using Support Vector Regression with the best hyperparameter tunning, we get a performance of 0.012519 for mean square error, 0.111723 for root mean square error and 0.608208 for r2 score. A graph of actual value to the predicted value is plotted as below.

```
plt.figure(figsize=(8, 6))
plt.scatter(y_test, y_test_pred, color='blue', alpha=0.5)  # Scatter plot of actual vs. p
plt.plot([min(y_test), max(y_test)], [min(y_test), max(y_test)], '--', color='red')  # Di
plt.xlabel('Actual Values')
plt.ylabel('Predicted Values')
plt.title('Actual vs. Predicted Values')
plt.grid(True)
plt.show()
```



5.7 Gaussian Process Regression (GPR)

Gaussian Process Regression (GRP) is a powerful and flexible non-parametric regression technique used in machine learning and statistics. It is useful when dealing with problems involving continuous data, where the relationship between input variables and output is not explicity known or can be complex.

Kernel is a very crucial key for Gaussian Process Regression because it helps to determine the shape of prior and posterior of the Gaussian Process Regression.

We set up a few kernels for testing.

```
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import RBF, ConstantKernel as C, RationalQuadratic
```

Gauusian Process Model with no kernel

```
model_0 = GaussianProcessRegressor()
model_0.fit(X_train, y_train)
```

```
▾ GaussianProcessRegressor
GaussianProcessRegressor()
```

```
import numpy as np
from sklearn import metrics
# Evaluate performance
y_val_pred = model_0.predict(X_val)
r2_score = metrics.r2_score(y_val, y_val_pred)
mse = metrics.mean_squared_error(y_val, y_val_pred)
rmse = np.sqrt(mse)

print("Performance for GPR model: ")
print("Mean Square Error (MSE): ",mse)
print("Root Mean Square Error (RMSE): ",rmse)
print("R2-score (R2): ",r2_score)
```

```
Performance for GPR model:
Mean Square Error (MSE):  0.025881419805891457
Root Mean Square Error (RMSE):  0.16087703318339588
R2-score (R2):  0.10210687084033554
```

Gaussian Process Regression without any hyperparameter tunning especially kernel performed poorly with a r2 score of 0.102107. Hence, we prepared some kernel option to test the model out to see which kernel work the best with the Gaussian Process Regression model.

Kernel 1

```
kernel_1 = RBF()
model_1 = GaussianProcessRegressor(kernel = kernel_1, n_restarts_optimizer=4)
model_1.fit(X_train, y_train)
```

```
GaussianProcessRegressor
GaussianProcessRegressor(kernel=RBF(length_scale=1), n_restarts_optimizer=4)
```

Kernel 2

```
kernel_2 = Lin()
model_2 = GaussianProcessRegressor(kernel = kernel_2, n_restarts_optimizer=4)
model_2.fit(X_train, y_train)
```

```
GaussianProcessRegressor
GaussianProcessRegressor(kernel=DotProduct(sigma_0=1), n_restarts_optimizer=4)
```

Kernel 3

```
kernel_3 = RQ()
model_3 = GaussianProcessRegressor(kernel = kernel_3, n_restarts_optimizer=4)
model_3.fit(X_train, y_train)
```

```
GaussianProcessRegressor
GaussianProcessRegressor(kernel=RationalQuadratic(alpha=1, length_scale=1),
                         n_restarts_optimizer=4)
```

Kernel 4

```
kernel_4 = C()
model_4 = GaussianProcessRegressor(kernel = kernel_4, n_restarts_optimizer=4)
model_4.fit(X_train, y_train)
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/gaussian_process/_gpr.py:629: Converg
ABNORMAL_TERMINATION_IN_LNSRCH.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
  _check_optimize_result("lbfgs", opt_res)
```

```
GaussianProcessRegressor
GaussianProcessRegressor(kernel=1**2, n_restarts_optimizer=4)
```

Kernel 5

```
kernel_5 = WhiteKernel()
model_5 = GaussianProcessRegressor(kernel = kernel_5, n_restarts_optimizer=4)
model_5.fit(X_train, y_train)
```

```
GaussianProcessRegressor
GaussianProcessRegressor(kernel=WhiteKernel(noise_level=1),
                         n_restarts_optimizer=4)
```

Kernel 6

```
kernel_6 = Exp()
model_6 = GaussianProcessRegressor(kernel = kernel_6, n_restarts_optimizer=4)
model_6.fit(X_train, y_train)
```

```
GaussianProcessRegressor
GaussianProcessRegressor(kernel=ExpSineSquared(length_scale=1, periodicity=1),
                         n_restarts_optimizer=4)
```

All of the model are fit with different type of kernel. Now, we test the Gaussian Process Regression using the different type of kernels with the validation dataset.

```
import numpy as np
from sklearn import metrics
# Evaluate performance
y_val_pred = model_1.predict(X_val)
r2_score = metrics.r2_score(y_val, y_val_pred)
mse = metrics.mean_squared_error(y_val, y_val_pred)
rmse = np.sqrt(mse)

print("Performance for GRP model using RBF: ")
print("Mean Square Error (MSE): ",mse)
print("Root Mean Square Error (RMSE): ",rmse)
print("R2-score (R2): ",r2_score)
```

```
Performance for GRP model using RBF:
Mean Square Error (MSE):  0.020838912898225953
Root Mean Square Error (RMSE):  0.144356894183222
R2-score (R2):  0.27704442604750246
```

```
import numpy as np
from sklearn import metrics
# Evaluate performance
y_val_pred = model_2.predict(X_val)
r2_score = metrics.r2_score(y_val, y_val_pred)
mse = metrics.mean_squared_error(y_val, y_val_pred)
rmse = np.sqrt(mse)

print("Performance for GRP model using DotProduct: ")
print("Mean Square Error (MSE): ",mse)
print("Root Mean Square Error (RMSE): ",rmse)
print("R2-score (R2): ",r2_score)
```

```
Performance for GRP model using DotProduct:
Mean Square Error (MSE):  0.01500073351760048
Root Mean Square Error (RMSE):  0.12247748167561448
R2-score (R2):  0.4795859091647453
```

```
import numpy as np
from sklearn import metrics
# Evaluate performance
y_val_pred = model_3.predict(X_val)
r2_score = metrics.r2_score(y_val, y_val_pred)
mse = metrics.mean_squared_error(y_val, y_val_pred)
rmse = np.sqrt(mse)

print("Performance for GRP model using RationalQuadratic: ")
print("Mean Square Error (MSE): ",mse)
print("Root Mean Square Error (RMSE): ",rmse)
print("R2-score (R2): ",r2_score)
```

```
Performance for GRP model using RationalQuadratic:
Mean Square Error (MSE):  0.010581252661764243
Root Mean Square Error (RMSE):  0.10286521599532197
R2-score (R2):  0.6329090855851027
```

```
import numpy as np
from sklearn import metrics
# Evaluate performance
y_val_pred = model_4.predict(X_val)
r2_score = metrics.r2_score(y_val, y_val_pred)
mse = metrics.mean_squared_error(y_val, y_val_pred)
rmse = np.sqrt(mse)

print("Performance for GRP model using ConstantKernel: ")
print("Mean Square Error (MSE): ",mse)
print("Root Mean Square Error (RMSE): ",rmse)
print("R2-score (R2): ",r2_score)
```

```
Performance for GRP model using ConstantKernel:
Mean Square Error (MSE):  4.951491604164519
Root Mean Square Error (RMSE):  2.2251947339872347
R2-score (R2):  -170.7799998537581
```

```
import numpy as np
from sklearn import metrics
# Evaluate performance
y_val_pred = model_5.predict(X_val)
r2_score = metrics.r2_score(y_val, y_val_pred)
mse = metrics.mean_squared_error(y_val, y_val_pred)
rmse = np.sqrt(mse)

print("Performance for GRP model using WhiteKernel: ")
print("Mean Square Error (MSE): ",mse)
print("Root Mean Square Error (RMSE): ",rmse)
print("R2-score (R2): ",r2_score)
```

```
Performance for GRP model using WhiteKernel:
Mean Square Error (MSE):  0.25824233787665785
Root Mean Square Error (RMSE):  0.5081754990912666
R2-score (R2):  -7.959091988639571
```

```
import numpy as np
from sklearn import metrics
# Evaluate performance
y_val_pred = model_6.predict(X_val)
r2_score = metrics.r2_score(y_val, y_val_pred)
mse = metrics.mean_squared_error(y_val, y_val_pred)
rmse = np.sqrt(mse)

print("Performance for GRP model using ExpSineSquared: ")
print("Mean Square Error (MSE): ",mse)
print("Root Mean Square Error (RMSE): ",rmse)
print("R2-score (R2): ",r2_score)
```

```
Performance for GRP model using ExpSineSquared:
Mean Square Error (MSE):  0.020838912859074
Root Mean Square Error (RMSE):  0.14435689404761382
R2-score (R2):  0.2770444274057846
```

From the result above, using Rational Kernel in Gaussian Process Regression performed the best prediction with the r2 score of 0.632909, followed by using Dot Product with the r2 score of 0.479586 and using RBF as the kernel performed the worst among the others with the r2 score of 0.277044.

Result using WhiteKernel, ConstantKernel and ExpSineQuared were not included in the comparison as they produced negative r2 score.

We also try out some combination of kernel suggested on the internet to test out the model with our dataset.

```
kernel_C1 = C()*Exp(length_scale = 24, periodicity=1)
model_C1 = GaussianProcessRegressor(kernel = kernel_C1, n_restarts_optimizer=4)
model_C1.fit(X_train, y_train)
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/gaussian_process/kernels.py:420: Conv
  warnings.warn(
```

```
GaussianProcessRegressor(kernel=1**2 * ExpSineSquared(length_scale=24, periodicity=1
                         n_restarts_optimizer=4)
```

```
kernel_C2 = C()*Exp(length_scale = 24, periodicity=1)*RQ(length_scale = 24, alpha = 0.5,
model_C2 = GaussianProcessRegressor(kernel = kernel_C2, n_restarts_optimizer=4)
model_C2.fit(X_train, y_train)
```

```
GaussianProcessRegressor(kernel=1**2 * ExpSineSquared(length_scale=24, periodicity=1
                         n_restarts_optimizer=4)
```

```
kernel_C3 = C() * RQ(length_scale = 24, alpha = 0.1)
model_C3 = GaussianProcessRegressor(kernel = kernel_C3, n_restarts_optimizer=4)
model_C3.fit(X_train, y_train)
```

```
GaussianProcessRegressor(kernel=1**2 * RationalQuadratic(alpha=0.1, length_scale=24)
                         n_restarts_optimizer=4)
```