About This notebook

This notebook has been prepared for our Data Mining course, focusing on applying predictive analytics to solve real-world problems.The contributor to this project is Parisa Tavakoli, a computer science student.

Table Of content

Note For viewers

Each section of this report corresponds to a milestone we set at the project's inception. The report details the approach taken, the technologies used, and the exact locations within our Jupyter Notebook where the related code can be found.

1. Handling Categorical Features with Missing Values: 3.1
2. Handling Numerical Features with Missing Values and Outliers: 3.2
3. Correlation Heatmap: 4.5
4. Label Encoding(Categorical To Numerical): 5.1 and 5.2
5. KNN: 7
6. DT: 8
7. SVM: 9
8. Comaprison: 10

I hope we covered everything that was needed :D

Further Questions?

If you have any further questions, feel free to contact me at Parisa.Tavakoli.81@gmail.com.

1 Importing Essential Libraries for Data Analysis and Visualization in Python

1.1 Library Imports and Dependencies

All required libraries for this notebook's operations are consolidated and imported in the following cell. This setup ensures comprehensive support for data manipulation, visualization, and machine learning tasks, streamlining the environment for seamless execution.

In [7]:
```
pip install tensorflow
```

```
Collecting tensorflow
  Downloading tensorflow-2.17.0-cp311-cp311-macosx_12_0_arm64.whl.metadata (4.1 kB)
Requirement already satisfied: absl-py>=1.0.0 in /opt/anaconda3/lib/python3.11/site-packages (from tensorflow) (2.1.0)
Collecting astunparse>=1.6.0 (from tensorflow)
  Downloading astunparse-1.6.3-py2.py3-none-any.whl.metadata (4.4 kB)
Collecting flatbuffers>=24.3.25 (from tensorflow)
  Downloading flatbuffers-24.3.25-py2.py3-none-any.whl.metadata (850 bytes)
Collecting gast!=0.5.0,!=0.5.1,!=0.5.2,>=0.2.1 (from tensorflow)
  Downloading gast-0.6.0-py3-none-any.whl.metadata (1.3 kB)
Collecting google-pasta>=0.1.1 (from tensorflow)
  Downloading google_pasta-0.2.0-py3-none-any.whl.metadata (814 bytes)
Collecting h5py>=3.10.0 (from tensorflow)
  Downloading h5py-3.11.0-cp311-cp311-macosx_11_0_arm64.whl.metadata (2.5 kB)
Collecting libclang>=13.0.0 (from tensorflow)
  Downloading libclang-18.1.1-1-py2.py3-none-macosx_11_0_arm64.whl.metadata (5.2 kB)
Requirement already satisfied: ml-dtypes<0.5.0,>=0.3.1 in /opt/anaconda3/lib/python3.11/site-packages (from tensorflow) (0.4.0)
Collecting opt-einsum>=2.3.2 (from tensorflow)
  Downloading opt_einsum-3.3.0-py3-none-any.whl.metadata (6.5 kB)
Requirement already satisfied: packaging in /opt/anaconda3/lib/python3.11/site-packages (from tensorflow) (23.1)
Requirement already satisfied: protobuf!=4.21.0,!=4.21.1,!=4.21.2,!=4.21.3,!=4.21.4,!=4.21.5,<5.0.0dev,>=3.20.3 in /opt/anaconda3/lib/python3.11/site-packa
ges (from tensorflow) (3.20.3)
Requirement already satisfied: requests<3,>=2.21.0 in /opt/anaconda3/lib/python3.11/site-packages (from tensorflow) (2.31.0)
Requirement already satisfied: setuptools in /opt/anaconda3/lib/python3.11/site-packages (from tensorflow) (68.2.2)
Requirement already satisfied: six>=1.12.0 in /opt/anaconda3/lib/python3.11/site-packages (from tensorflow) (1.16.0)
Collecting termcolor>=1.1.0 (from tensorflow)
  Downloading termcolor-2.4.0-py3-none-any.whl.metadata (6.1 kB)
Requirement already satisfied: typing-extensions>=3.6.6 in /opt/anaconda3/lib/python3.11/site-packages (from tensorflow) (4.9.0)
Requirement already satisfied: wrapt>=1.11.0 in /opt/anaconda3/lib/python3.11/site-packages (from tensorflow) (1.14.1)
Collecting grpcio<2.0,>=1.24.3 (from tensorflow)
  Downloading grpcio-1.64.1-cp311-cp311-macosx_10_9_universal2.whl.metadata (3.3 kB)
Collecting tensorboard<2.18,>=2.17 (from tensorflow)
  Downloading tensorboard-2.17.0-py3-none-any.whl.metadata (1.6 kB)
Requirement already satisfied: keras>=3.2.0 in /opt/anaconda3/lib/python3.11/site-packages (from tensorflow) (3.4.1)
Collecting tensorflow-io-gcs-filesystem>=0.23.1 (from tensorflow)
  Downloading tensorflow_io_gcs_filesystem-0.37.1-cp311-cp311-macosx_12_0_arm64.whl.metadata (14 kB)
Requirement already satisfied: numpy<2.0.0,>=1.23.5 in /opt/anaconda3/lib/python3.11/site-packages (from tensorflow) (1.26.4)
Requirement already satisfied: wheel<1.0,>=0.23.0 in /opt/anaconda3/lib/python3.11/site-packages (from astunparse>=1.6.0->tensorflow) (0.41.2)
Requirement already satisfied: rich in /opt/anaconda3/lib/python3.11/site-packages (from keras>=3.2.0->tensorflow) (13.3.5)
Requirement already satisfied: namex in /opt/anaconda3/lib/python3.11/site-packages (from keras>=3.2.0->tensorflow) (0.0.8)
Requirement already satisfied: optree in /opt/anaconda3/lib/python3.11/site-packages (from keras>=3.2.0->tensorflow) (0.12.1)
Requirement already satisfied: charset-normalizer<4,>=2 in /opt/anaconda3/lib/python3.11/site-packages (from requests<3,>=2.21.0->tensorflow) (2.0.4)
Requirement already satisfied: idna<4,>=2.5 in /opt/anaconda3/lib/python3.11/site-packages (from requests<3,>=2.21.0->tensorflow) (3.4)
Requirement already satisfied: urllib3<3,>=1.21.1 in /opt/anaconda3/lib/python3.11/site-packages (from requests<3,>=2.21.0->tensorflow) (2.0.7)
Requirement already satisfied: certifi>=2017.4.17 in /opt/anaconda3/lib/python3.11/site-packages (from requests<3,>=2.21.0->tensorflow) (2024.2.2)
Requirement already satisfied: markdown>=2.6.8 in /opt/anaconda3/lib/python3.11/site-packages (from tensorboard<2.18,>=2.17->tensorflow) (3.4.1)
Collecting tensorboard-data-server<0.8.0,>=0.7.0 (from tensorboard<2.18,>=2.17->tensorflow)
  Downloading tensorboard_data_server-0.7.2-py3-none-any.whl.metadata (1.1 kB)
Requirement already satisfied: werkzeug>=1.0.1 in /opt/anaconda3/lib/python3.11/site-packages (from tensorboard<2.18,>=2.17->tensorflow) (2.2.3)
Requirement already satisfied: MarkupSafe>=2.1.1 in /opt/anaconda3/lib/python3.11/site-packages (from werkzeug>=1.0.1->tensorboard<2.18,>=2.17->tensorflow)
(2.1.3)
Requirement already satisfied: markdown-it-py<3.0.0,>=2.2.0 in /opt/anaconda3/lib/python3.11/site-packages (from rich->keras>=3.2.0->tensorflow) (2.2.0)
Requirement already satisfied: pygments<3.0.0,>=2.13.0 in /opt/anaconda3/lib/python3.11/site-packages (from rich->keras>=3.2.0->tensorflow) (2.15.1)
Requirement already satisfied: mdurl~=0.1 in /opt/anaconda3/lib/python3.11/site-packages (from markdown-it-py<3.0.0,>=2.2.0->rich->keras>=3.2.0->tensorflo
w) (0.1.0)
Downloading tensorflow-2.17.0-cp311-cp311-macosx_12_0_arm64.whl (236.2 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 236.2/236.2 MB 3.0 MB/s eta 0:00:0000:0100:02
Downloading astunparse-1.6.3-py2.py3-none-any.whl (12 kB)
Downloading flatbuffers-24.3.25-py2.py3-none-any.whl (26 kB)
Downloading gast-0.6.0-py3-none-any.whl (21 kB)
Downloading google_pasta-0.2.0-py3-none-any.whl (57 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 57.5/57.5 kB 6.1 MB/s eta 0:00:00
Downloading grpcio-1.64.1-cp311-cp311-macosx_10_9_universal2.whl (10.4 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 10.4/10.4 MB 3.9 MB/s eta 0:00:0000:0100:01
Downloading h5py-3.11.0-cp311-cp311-macosx_11_0_arm64.whl (2.9 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 2.9/2.9 MB 4.0 MB/s eta 0:00:00a 0:00:01
Downloading libclang-18.1.1-1-py2.py3-none-macosx_11_0_arm64.whl (25.8 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 25.8/25.8 MB 3.8 MB/s eta 0:00:0000:0100:01
Downloading opt_einsum-3.3.0-py3-none-any.whl (65 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 65.5/65.5 kB 2.7 MB/s eta 0:00:00
Downloading tensorboard-2.17.0-py3-none-any.whl (5.5 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 5.5/5.5 MB 4.0 MB/s eta 0:00:0000:0100:01m
Downloading tensorflow_io_gcs_filesystem-0.37.1-cp311-cp311-macosx_12_0_arm64.whl (3.5 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 3.5/3.5 MB 4.0 MB/s eta 0:00:0000:0100:01
Downloading termcolor-2.4.0-py3-none-any.whl (7.7 kB)
Downloading tensorboard_data_server-0.7.2-py3-none-any.whl (2.4 kB)
Installing collected packages: libclang, flatbuffers, termcolor, tensorflow-io-gcs-filesystem, tensorboard-data-server, opt-einsum, h5py, grpcio, google-pa
sta, gast, astunparse, tensorboard, tensorflow
  Attempting uninstall: h5py
    Found existing installation: h5py 3.9.0
    Uninstalling h5py-3.9.0:
      Successfully uninstalled h5py-3.9.0
Successfully installed astunparse-1.6.3 flatbuffers-24.3.25 gast-0.6.0 google-pasta-0.2.0 grpcio-1.64.1 h5py-3.11.0 libclang-18.1.1 opt-einsum-3.3.0 tensor
board-2.17.0 tensorboard-data-server-0.7.2 tensorflow-2.17.0 tensorflow-io-gcs-filesystem-0.37.1 termcolor-2.4.0
Note: you may need to restart the kernel to use updated packages.
```

In [8]:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import time
import pickle
import datetime

from sklearn.linear_model import LogisticRegression, LinearRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn import svm
from sklearn.model_selection import GridSearchCV, train_test_split
from sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn.metrics import (
    jaccard_score, f1_score, log_loss, confusion_matrix,
    accuracy_score, precision_score, recall_score, classification_report
)
```

```
from keras.models import Sequential
from keras.layers import Dense, BatchNormalization, Dropout, LSTM
from keras.utils import to_categorical
from keras.optimizers import Adam
from keras import callbacks
from tensorflow.keras import regularizers

# Setting random seed for reproducibility
np.random.seed(0)

# Configuring inline plotting via matplotlib
%matplotlib inline
```

1.2 Suppression of Warnings

To ensure clarity in output and focus on essential results, non-critical Python warnings were suppressed during the execution of the project. This was achieved through the following code snippet:

In [9]:
```
# Surpressing warnings:
def warn(*args, **kwargs):
    pass
import warnings
warnings.warn = warn
```

This approach helps in maintaining a clean output by preventing the display of routine warnings related to deprecations or minor issues, thus allowing for uninterrupted analysis and presentation.

1.3 Setting a Global Color Palette

To enhance visual consistency across all charts in the analysis, a global color palette is defined. This palette will be used for all subsequent visualizations to maintain uniformity and improve the aesthetic appeal of the report.

By setting the palette with `sns.set_palette()`, you ensure that seaborn plots throughout the notebook will automatically use these colors. This eliminates the need to specify the `palette` parameter in each plotting function, simplifying the code and ensuring consistency.

In [10]:
```
# Define a global color palette
global_palette = ["#C2C4E2", "#EED4E5"]

# Set the color palette in seaborn for all plots
sns.set_palette(sns.color_palette(global_palette))
```

2 Loading and Previewing the Dataset: Initial Insights into Rain Data

2.1 Loading the Dataset

The dataset, containing rain data, is loaded from a CSV file named Dataset.csv into a Pandas DataFrame called rain. This initial step is crucial for the subsequent data handling and analysis tasks, setting the foundation for exploring and manipulating the data.

In [11]:
```
# Load the dataset
rain = pd.read_csv('Dataset.csv')
```

2.2 Previewing the Dataset

To get an initial understanding of the dataset's structure and the types of data it contains, the first few rows are displayed using the `head()` method on the DataFrame `rain`. This provides a quick snapshot of the columns and the values they hold, which is essential for assessing data quality and preparing for further data exploration and cleaning.

In [12]:
```
# Display the first few rows of the dataset
rain.head()
```

Out[12]:

	Unnamed: 0	Date	Weather Station	Minimum Temperature	Maximum Temperature	Rainfall	Evaporation	Sunshine	Gust Trajectory	Air Velocity	...	Moisture Level at 9 AM	Moisture Level at 3 PM	Atmospheric Pressure at 9 AM	Atmospheric Pressure at 3 PM	Cloudi at 9
0	0	12/1/2008	Station 2	13.4	22.9	0.6	NaN	NaN	W	44.0	...	71.0	22.0	1007.7	1007.1	
1	1	12/2/2008	Station 2	7.4	25.1	0.0	NaN	NaN	WNW	44.0	...	44.0	25.0	1010.6	1007.8	
2	2	12/3/2008	Station 2	12.9	25.7	0.0	NaN	NaN	WSW	46.0	...	38.0	30.0	1007.6	1008.7	
3	3	12/4/2008	Station 2	9.2	28.0	0.0	NaN	NaN	NE	24.0	...	45.0	16.0	1017.6	1012.8	
4	4	12/5/2008	Station 2	17.5	32.3	1.0	NaN	NaN	W	41.0	...	82.0	33.0	1010.8	1006.0	

5 rows × 24 columns

This command outputs the top five rows of the dataset, offering immediate insight into the dataset's format, the nature of the attributes, and preliminary data characteristics such as the presence of any missing values or the data types of each column.

2.3 Understanding Dataset Dimensions

To assess the size of the dataset, the `shape` attribute of the DataFrame `rain` is used. This attribute provides the total number of rows and columns, helping to understand the scale of the data and guiding subsequent data processing decisions.

In [13]:
```
rain.shape
```

Out[13]: (145460, 24)

This line of code outputs a tuple representing the number of rows (data entries) and columns (features) in the dataset. Knowing the dimensions is crucial for planning the data analysis, including handling missing values, scaling data, and selecting appropriate machine learning models.

2.4 Dataset Structure and Content Overview

To further explore the dataset, the `info()` method is called on the DataFrame `rain`. This method provides a concise summary of the DataFrame, including the number of non-null entries in each column, the data type of each column, and memory usage. It is particularly useful for quickly identifying columns with missing values and understanding the data types that need to be converted or handled differently during preprocessing.

In [14]:
```
rain.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 145460 entries, 0 to 145459
Data columns (total 24 columns):
 #   Column                        Non-Null Count   Dtype  
---  ------                        --------------   -----  
 0   Unnamed: 0                    145460 non-null  int64  
 1   Date                          145460 non-null  object 
 2   Weather Station               145460 non-null  object 
 3   Minimum Temperature           143975 non-null  float64
 4   Maximum Temperature           144199 non-null  float64
 5   Rainfall                      142199 non-null  float64
 6   Evaporation                   82670 non-null   float64
 7   Sunshine                      75625 non-null   float64
 8   Gust Trajectory               135134 non-null  object 
 9   Air Velocity                  135197 non-null  float64
 10  Gust Trajectory at  9 AM      134894 non-null  object 
 11  Gust Trajectory at  3 PM      141232 non-null  object 
 12  Air Velocity at  9 AM         143693 non-null  float64
 13  Air Velocity at  3 PM         142398 non-null  float64
 14  Moisture Level at 9 AM        142806 non-null  float64
 15  Moisture Level at 3 PM        140953 non-null  float64
 16  Atmospheric Pressure at 9 AM  130395 non-null  float64
 17  Atmospheric Pressure at 3 PM  130432 non-null  float64
 18  Cloudiness at 9 AM            89572 non-null   float64
 19  Cloudiness at 3 PM            86102 non-null   float64
 20  Recorded Temperature at 9 AM  143693 non-null  float64
 21  Recorded Temperature at 3 PM  141851 non-null  float64
 22  Rain that day                 142199 non-null  object 
 23  Rain the day after            142193 non-null  object 
dtypes: float64(16), int64(1), object(7)
memory usage: 26.6+ MB
```

This function is essential for preliminary data checks, helping to guide decisions on data cleaning and type conversions which are crucial for effective data analysis and model training. It ensures that all features are appropriately formatted and ready for further analytical steps.

2.5 Removing Redundant Columns

In the process of data cleaning, the column labeled `'Unnamed: 0'` is removed from the DataFrame `rain`. This column, typically generated during data export or import when an index column is included without a name, is redundant and does not hold any useful information for analysis.

In [15]:
```
rain.drop('Unnamed: 0', axis=1, inplace=True)
```

By setting `axis=1`, the operation targets columns (not rows), and `inplace=True` modifies the DataFrame directly. Removing this column simplifies the dataset, enhancing clarity and reducing memory usage, which can improve processing efficiency in subsequent data manipulation and analysis steps. This step is crucial for maintaining a clean and focused dataset for accurate and efficient analysis.

In [16]:
```
print(rain.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 145460 entries, 0 to 145459
Data columns (total 23 columns):
 #   Column                        Non-Null Count   Dtype  
---  ------                        --------------   -----  
 0   Date                          145460 non-null  object 
 1   Weather Station               145460 non-null  object 
 2   Minimum Temperature           143975 non-null  float64
 3   Maximum Temperature           144199 non-null  float64
 4   Rainfall                      142199 non-null  float64
 5   Evaporation                   82670 non-null   float64
 6   Sunshine                      75625 non-null   float64
 7   Gust Trajectory               135134 non-null  object 
 8   Air Velocity                  135197 non-null  float64
 9   Gust Trajectory at  9 AM      134894 non-null  object 
 10  Gust Trajectory at  3 PM      141232 non-null  object 
 11  Air Velocity at  9 AM         143693 non-null  float64
 12  Air Velocity at  3 PM         142398 non-null  float64
 13  Moisture Level at 9 AM        142806 non-null  float64
 14  Moisture Level at 3 PM        140953 non-null  float64
 15  Atmospheric Pressure at 9 AM  130395 non-null  float64
 16  Atmospheric Pressure at 3 PM  130432 non-null  float64
 17  Cloudiness at 9 AM            89572 non-null   float64
 18  Cloudiness at 3 PM            86102 non-null   float64
 19  Recorded Temperature at 9 AM  143693 non-null  float64
 20  Recorded Temperature at 3 PM  141851 non-null  float64
 21  Rain that day                 142199 non-null  object 
 22  Rain the day after            142193 non-null  object 
dtypes: float64(16), object(7)
memory usage: 25.5+ MB
None
```

2.6 Generating Descriptive Statistics and Enhancing Presentation

This segment of the analysis involves generating descriptive statistics for numerical columns in the `rain` DataFrame, excluding columns of type `object`, which typically contain non-numeric data. The `.describe()` method provides key statistics such as mean, median, standard deviation, minimum, and maximum values, essential for gaining insights into the

distribution and variability of data.

To improve the readability and accessibility of these statistics, the DataFrame is then transposed:

Transposing the statistics DataFrame rearranges it so that the statistical measures (mean, count, standard deviation, etc.) are displayed as columns rather than rows. This makes it easier to compare these measures across different features.

This step allows for a more intuitive presentation of the data, enabling quick visual assessments and facilitating easier communication of the data's characteristics. It's particularly useful when dealing with multiple variables, helping to provide a clearer summary of each attribute's statistical properties in the dataset.

In [17]:
```
stats = rain.describe(exclude=[object])

# Now transpose the DataFrame
transposed_stats = stats.T

# Display the transposed DataFrame
display(transposed_stats)
```

	count	mean	std	min	25%	50%	75%	max
Minimum Temperature	143975.0	12.194034	6.398495	-8.5	7.6	12.0	16.9	33.9
Maximum Temperature	144199.0	23.221348	7.119049	-4.8	17.9	22.6	28.2	48.1
Rainfall	142199.0	2.360918	8.478060	0.0	0.0	0.0	0.8	371.0
Evaporation	82670.0	5.468232	4.193704	0.0	2.6	4.8	7.4	145.0
Sunshine	75625.0	7.611178	3.785483	0.0	4.8	8.4	10.6	14.5
Air Velocity	135197.0	40.035230	13.607062	6.0	31.0	39.0	48.0	135.0
Air Velocity at 9 AM	143693.0	14.043426	8.915375	0.0	7.0	13.0	19.0	130.0
Air Velocity at 3 PM	142398.0	18.662657	8.809800	0.0	13.0	19.0	24.0	87.0
Moisture Level at 9 AM	142806.0	68.880831	19.029164	0.0	57.0	70.0	83.0	100.0
Moisture Level at 3 PM	140953.0	51.539116	20.795902	0.0	37.0	52.0	66.0	100.0
Atmospheric Pressure at 9 AM	130395.0	1017.649940	7.106530	980.5	1012.9	1017.6	1022.4	1041.0
Atmospheric Pressure at 3 PM	130432.0	1015.255889	7.037414	977.1	1010.4	1015.2	1020.0	1039.6
Cloudiness at 9 AM	89572.0	4.447461	2.887159	0.0	1.0	5.0	7.0	9.0
Cloudiness at 3 PM	86102.0	4.509930	2.720357	0.0	2.0	5.0	7.0	9.0
Recorded Temperature at 9 AM	143693.0	16.990631	6.488753	-7.2	12.3	16.7	21.6	40.2
Recorded Temperature at 3 PM	141851.0	21.683390	6.936650	-5.4	16.6	21.1	26.4	46.7

2.7 Descriptive Statistics for Categorical Data

For a comprehensive analysis, it's important to understand not only the numerical attributes but also the categorical ones. This code snippet calculates descriptive statistics for columns of type `object` in the `rain` DataFrame, which are typically categorical or textual data. The `.describe()` method, when used with the parameter `include=[object]`, focuses on statistics relevant to categorical data, such as count, unique count, top, and frequency of the most common category.

To improve the accessibility and presentation of this data, the DataFrame containing the statistics is transposed:

This transposition changes the layout so that each categorical column's descriptive metrics are displayed as rows, making it easier to read and compare across categories.

This visualization strategy enhances the ease with which these statistics can be analyzed and understood, facilitating better data-driven decisions about data preprocessing tasks such as encoding strategies or handling of categorical variables.

In [18]:
```
stats2 = rain.describe(include=[object])

# Now transpose the DataFrame
transposed_stats2 = stats2.T

# Display the transposed DataFrame
display(transposed_stats2)
```

	count	unique	top	freq
Date	145460	3436	11/12/2013	49
Weather Station	145460	49	Station 9	3436
Gust Trajectory	135134	16	W	9915
Gust Trajectory at 9 AM	134894	16	N	11758
Gust Trajectory at 3 PM	141232	16	SE	10838
Rain that day	142199	2	No	110319
Rain the day after	142193	2	No	110316

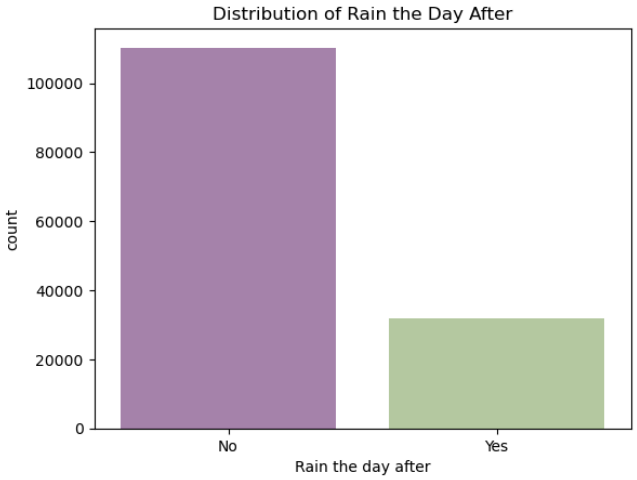
2.8 Evaluating Class Distribution in the Dataset

To assess whether the dataset suffers from class imbalance, which is a common issue in predictive modeling that can significantly impact the performance of a model, a visualization of the target variable `Rain the day after` is created using a countplot. This visualization helps in quickly identifying disparities in the number of instances for each class. Such an evaluation is crucial as it influences the choice of data preprocessing techniques such as resampling or the application of different evaluation metrics that are more appropriate for imbalanced data. By visually assessing the balance of the dataset, one can better prepare for accurate model training and evaluation.

In [19]:
```
# Calculate the counts and percentages of each response
counts = rain['Rain the day after'].value_counts()
percentages = rain['Rain the day after'].value_counts(normalize=True) * 100

# Print the percentages
```

```
print("Percentage of 'No': {:.2f}%".format(percentages['No']))
print("Percentage of 'Yes': {:.2f}%".format(percentages['Yes']))

# Create a count plot
cols = ["#ae7bb5", "#B5D09C"]  # Color palette for the plot
sns.countplot(x=rain["Rain the day after"], palette=cols)
plt.title('Distribution of Rain the Day After')
plt.show()
```

```
Percentage of 'No': 77.58%
Percentage of 'Yes': 22.42%
```



3 Identifying Categorical and Numerical Features in the Rain Dataset

3.1 Identification of Categorical Features

In the preprocessing phase, it is essential to identify and categorize the types of features present in the dataset. This code snippet focuses on identifying categorical features—attributes with data type 'object' (often strings representing categories). Understanding the nature and number of categorical features is crucial for determining the need for data transformation techniques such as encoding.

In [20]:
```
categorical_features = [column_name for column_name in rain.columns if rain[column_name].dtype == 'O']
print("Number of Categorical Features: {}".format(len(categorical_features)))
print("Categorical Features: ",categorical_features)
```

```
Number of Categorical Features: 7
Categorical Features:  ['Date', 'Weather Station', 'Gust Trajectory', 'Gust Trajectory at  9 AM', 'Gust Trajectory at  3 PM', 'Rain that day', 'Rain the da
y after']
```

3.1.1 Cardinality Check for Categorical Features

Before encoding categorical data for use in machine learning models such as Logistic Regression and Support Vector Machines, it's crucial to assess the cardinality of these features. Cardinality refers to the number of unique values in a feature column. High cardinality in features, like a column containing hundreds of unique zip codes, can significantly complicate model training:

- **Challenges of High Cardinality**: Features with high cardinality can lead to a substantial increase in dataset dimensions when encoded, potentially degrading model performance.
- **Strategies for High Cardinality**: To mitigate issues associated with high cardinality, feature engineering can be applied to derive new, more manageable features. Alternatively, features that offer minimal predictive value might be dropped entirely.

This approach ensures the dataset is optimally prepared, maintaining model efficiency and effectiveness.

In [21]:
```
for each_feature in categorical_features:
   unique_values = len(rain[each_feature].unique())
   print("Cardinality(no. of unique values) of {} are: {}".format(each_feature, unique_values))
```

```
Cardinality(no. of unique values) of Date are: 3436
Cardinality(no. of unique values) of Weather Station are: 49
Cardinality(no. of unique values) of Gust Trajectory are: 17
Cardinality(no. of unique values) of Gust Trajectory at  9 AM are: 17
Cardinality(no. of unique values) of Gust Trajectory at  3 PM are: 17
Cardinality(no. of unique values) of Rain that day are: 3
Cardinality(no. of unique values) of Rain the day after are: 3
```

3.1.2 Transforming and Extracting Date Components from the Rain Dataset

Challenges and Feature Engineering for High Cardinality in Date Column

The Date column in our dataset exhibits high cardinality, which can lead to inefficiencies in machine learning models due to increased dimensionality when the dates are converted to numerical formats. This situation necessitates specific preprocessing steps to enhance model performance.

In [22]:
```
rain['Date'].dtype
```

Out[22]:
```
dtype('O')
```

In [23]:
```
# Convert the 'Date' column to datetime
rain['Date'] = pd.to_datetime(rain['Date'])

# Extract year, month, and day
rain['Year'] = rain['Date'].dt.year
```

```
rain['Month'] = rain['Date'].dt.month
rain['Day'] = rain['Date'].dt.day
```

In [24]:
```
# Drop the original 'Date' column
rain.drop('Date', axis=1, inplace=True)
```

In [25]:
```
# Display the updated DataFrame to check the result
rain.head()
```

Out[25]:

	Weather Station	Minimum Temperature	Maximum Temperature	Rainfall	Evaporation	Sunshine	Gust Trajectory	Air Velocity	Gust Trajectory at 9 AM	Gust Trajectory at 3 PM	...	Atmospheric Pressure at 3 PM	Cloudiness at 9 AM	Cloudiness at 3 PM	Recorded Temperature at 9 AM	T
0	Station 2	13.4	22.9	0.6	NaN	NaN	W	44.0	W	WNW	...	1007.1	8.0	NaN	16.9	
1	Station 2	7.4	25.1	0.0	NaN	NaN	WNW	44.0	NNW	WSW	...	1007.8	NaN	NaN	17.2	
2	Station 2	12.9	25.7	0.0	NaN	NaN	WSW	46.0	W	WSW	...	1008.7	NaN	2.0	21.0	
3	Station 2	9.2	28.0	0.0	NaN	NaN	NE	24.0	SE	E	...	1012.8	NaN	NaN	18.1	
4	Station 2	17.5	32.3	1.0	NaN	NaN	W	41.0	ENE	NW	...	1006.0	7.0	8.0	17.8	

5 rows × 25 columns

In [26]:
```
# categorical data:

categorical_features = [column_name for column_name in rain.columns if rain[column_name].dtype == 'O']
print("Number of Categorical Features: {}".format(len(categorical_features)))
print("Categorical Features: ",categorical_features)
```

```
Number of Categorical Features: 6
Categorical Features:  ['Weather Station', 'Gust Trajectory', 'Gust Trajectory at  9 AM', 'Gust Trajectory at  3 PM', 'Rain that day', 'Rain the day afte
r']
```

In [27]:
```
# Numerical Features:

numerical_features = [column_name for column_name in rain.columns if rain[column_name].dtype != 'O']
#rain.select_dtypes(include=['float64','int64']).columns
print("Number of Numerical Features: {}".format(len(numerical_features)))
print("Numerical Features: ",numerical_features)
```

```
Number of Numerical Features: 19
Numerical Features:  ['Minimum Temperature', 'Maximum Temperature', 'Rainfall', 'Evaporation', 'Sunshine', 'Air Velocity', 'Air Velocity at  9 AM', 'Air Ve
locity at  3 PM', 'Moisture Level at 9 AM', 'Moisture Level at 3 PM', 'Atmospheric Pressure at 9 AM', 'Atmospheric Pressure at 3 PM', 'Cloudiness at 9 AM',
'Cloudiness at 3 PM', 'Recorded Temperature at 9 AM', 'Recorded Temperature at 3 PM', 'Year', 'Month', 'Day']
```

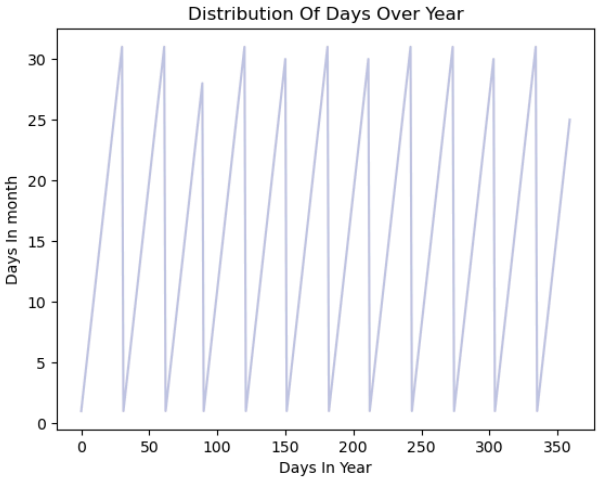
As we can see after chnaging Date we have three more Numerical Feature and one less categorical.

3.1.3 Visualization of Daily Distribution Over a Year

To understand the temporal distribution within the dataset, a visualization is created that represents roughly a year's span of data from the rain DataFrame. This visualization helps in assessing any seasonal patterns or anomalies in the dataset over a year.

In [28]:
```
# roughly a year's span section
section = rain[:360]
tm = section["Day"].plot(color="#C2C4E2")
tm.set_title("Distribution Of Days Over Year")
tm.set_ylabel("Days In month")
tm.set_xlabel("Days In Year")
```

Out[28]:
```
Text(0.5, 0, 'Days In Year')
```



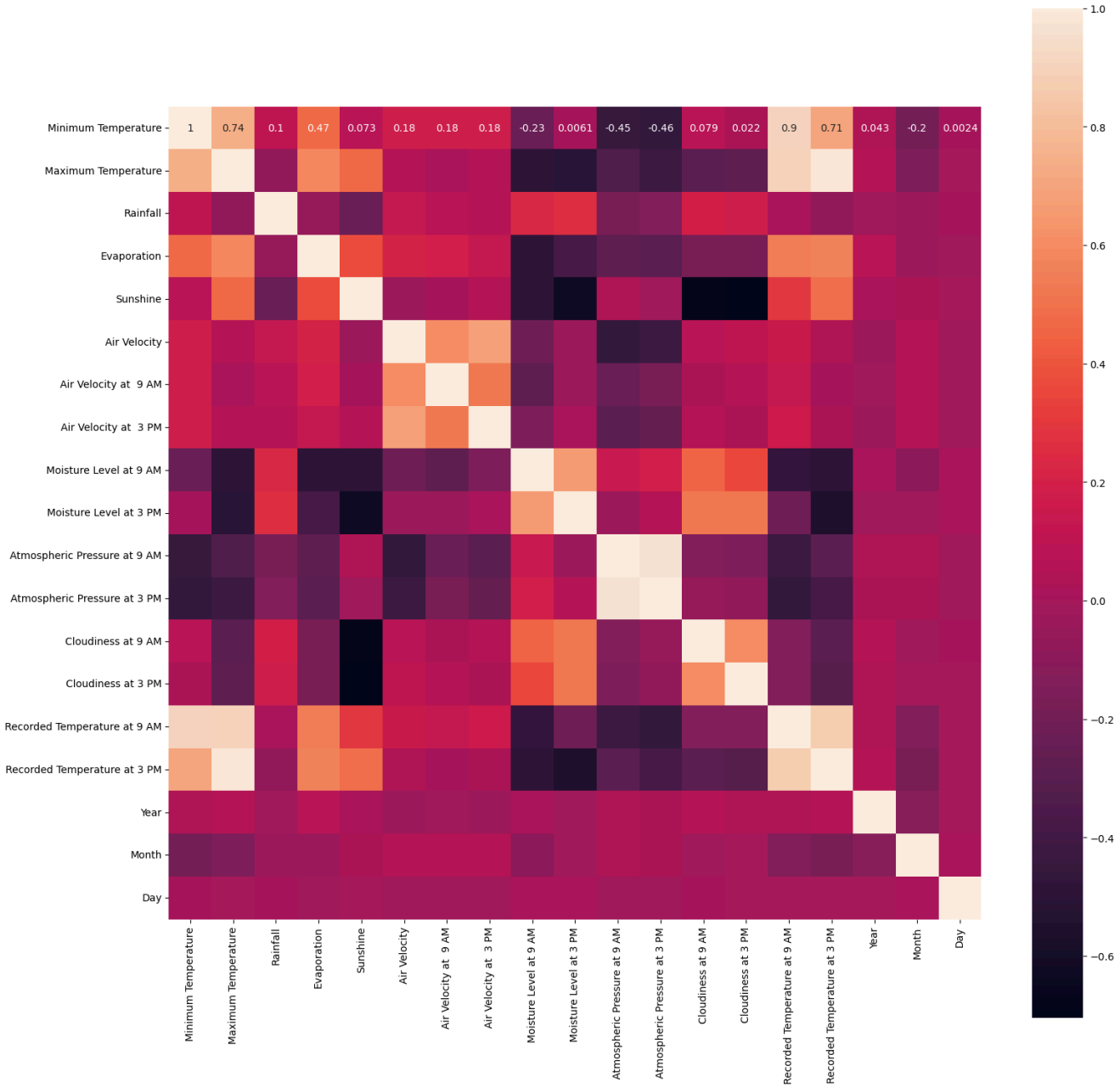
3.1.4 Correlation Analysis Among Numerical Features

Understanding the interrelationships among numerical features in the dataset is crucial for both feature selection and modeling. A correlation analysis helps identify how closely changes in one feature are associated with changes in another, which can influence model building and feature engineering.

In [29]:
```
# Select only numeric columns from the DataFrame
numeric_data = rain.select_dtypes(include=[np.number])

# Calculate the correlation matrix for numeric attributes
corrmat = numeric_data.corr()
```

```
# Generate a heatmap to visualize the correlation matrix
plt.subplots(figsize=(18, 18))
sns.heatmap(corrmat, annot=True, square=True)
```

Out[29]: <Axes: >



3.1.5 Assessment of Missing Values in Categorical Features

Identifying and quantifying missing data within categorical features is a critical step in data preprocessing. This action ensures that the dataset is adequately prepared for further analysis, such as feature encoding or model training, where missing values could lead to errors or biased results.

```
In [30]: rain[categorical_features].isnull().sum()
```

```
Out[30]: Weather Station               0
         Gust Trajectory           10326
         Gust Trajectory at  9 AM  10566
         Gust Trajectory at  3 PM   4228
         Rain that day              3261
         Rain the day after         3267
         dtype: int64
```

3.1.5.1 Identification and Imputation of Categorical Features with Missing Values

Early detection of missing values in categorical features is crucial for ensuring data quality and effectiveness in subsequent preprocessing steps. This identification helps prioritize which features need immediate attention either for imputation or for decision-making regarding their utility in the dataset.

```
In [31]: # list of categorical features which has null values:

categorical_features_with_null = [feature for feature in categorical_features if rain[feature].isnull().sum()]
```

To ensure data integrity and maintain the robustness of the dataset for analysis and modeling, missing values in categorical features are imputed with the most frequent value, known as the mode. This approach is commonly used to handle missing data where the deletion of rows or features could lead to significant information loss.

In [32]:
```
# Filling the missing(Null) categorical features with most frequent value(mode)

for each_feature in categorical_features_with_null:
    mode_val = rain[each_feature].mode()[0]
    rain[each_feature].fillna(mode_val,inplace=True)
```

In [33]:
```
rain[categorical_features].isnull().sum()
```

Out[33]:
```
Weather Station             0
Gust Trajectory             0
Gust Trajectory at  9 AM    0
Gust Trajectory at  3 PM    0
Rain that day               0
Rain the day after          0
dtype: int64
```

as we can see now there is no missing value here.

3.2 Checking for Missing Values in Numerical Features

To ensure data completeness and prepare for statistical analysis or machine learning modeling, it's crucial to identify and quantify missing values in numerical features. This step is fundamental for evaluating the need for further data cleaning or imputation strategies.

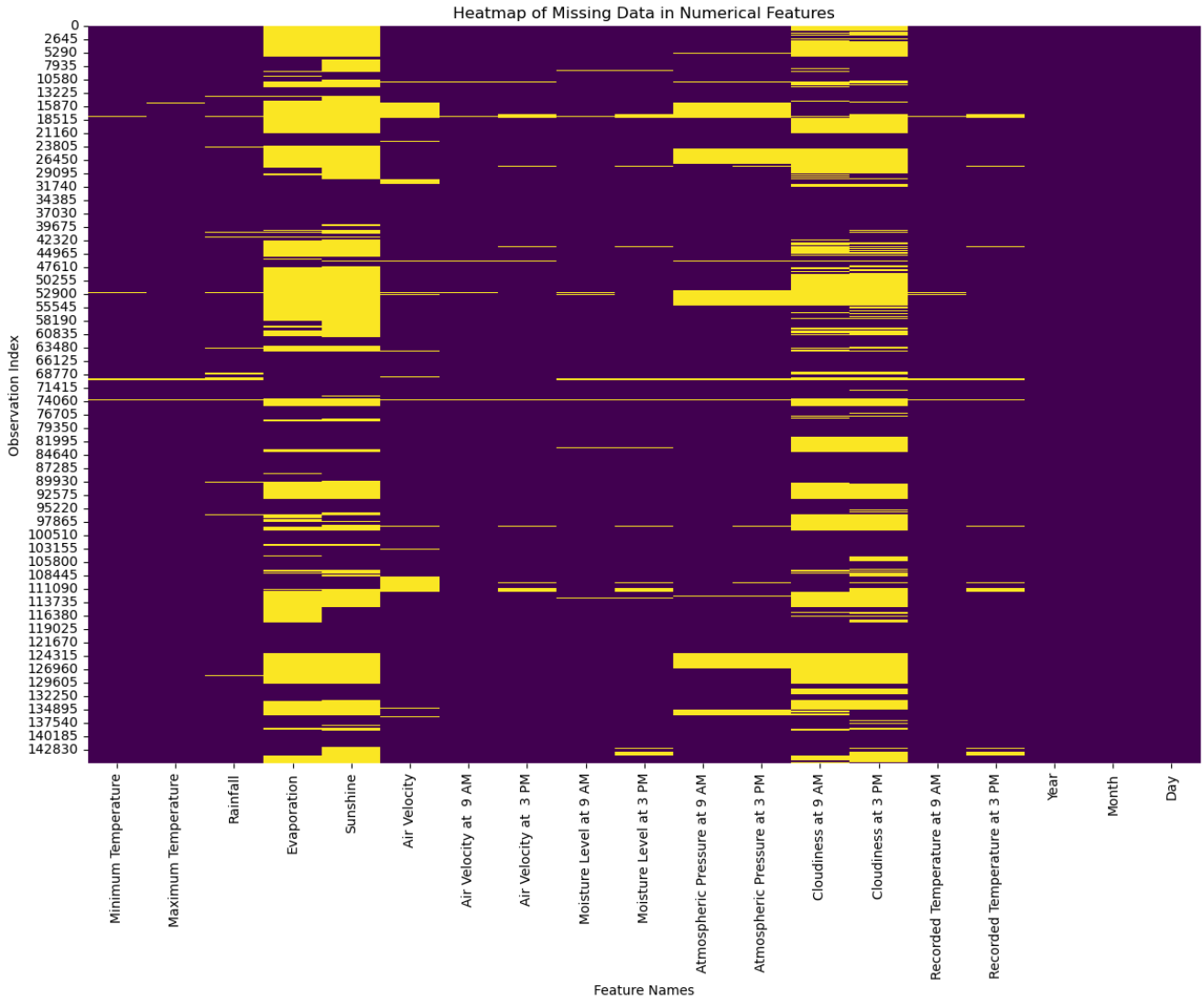
In [34]:
```
# checking null values in numerical features

rain[numerical_features].isnull().sum()
```

Out[34]:
```
Minimum Temperature              1485
Maximum Temperature              1261
Rainfall                         3261
Evaporation                     62790
Sunshine                        69835
Air Velocity                    10263
Air Velocity at  9 AM            1767
Air Velocity at  3 PM            3062
Moisture Level at 9 AM           2654
Moisture Level at 3 PM           4507
Atmospheric Pressure at 9 AM    15065
Atmospheric Pressure at 3 PM    15028
Cloudiness at 9 AM              55888
Cloudiness at 3 PM              59358
Recorded Temperature at 9 AM     1767
Recorded Temperature at 3 PM     3609
Year                                0
Month                               0
Day                                 0
dtype: int64
```

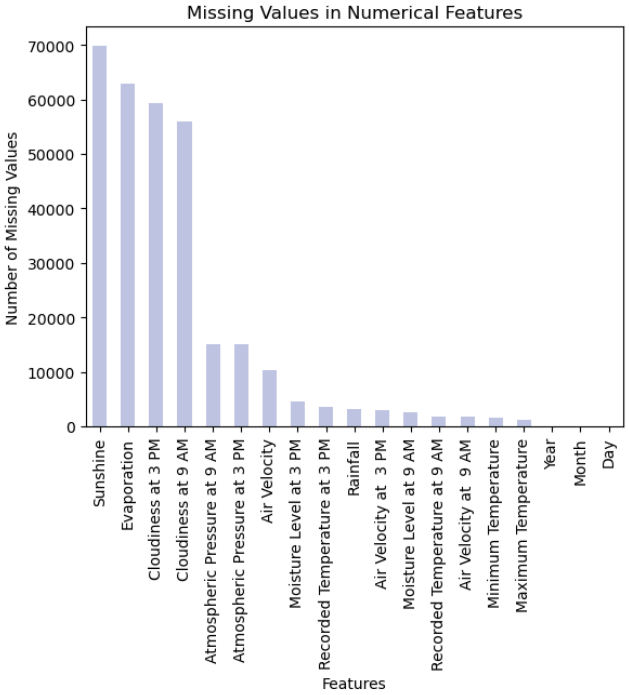
3.2.1 Visualization of Missing Values in Numerical Features

Visualizing missing data can provide intuitive insights into the pattern and extent of missing values across numerical features. A heatmap is an effective tool for this visualization, as it allows for the immediate identification of data incompleteness across multiple variables.

In [35]:
```
# Create a heatmap to visualize missing values in numerical features
plt.figure(figsize=(15,10))
sns.heatmap(rain[numerical_features].isnull(), cmap='viridis', linecolor='white', cbar=False)

# Adding labels for clarity
plt.title('Heatmap of Missing Data in Numerical Features')
plt.xlabel('Feature Names')  # Label for columns
plt.ylabel('Observation Index')  # Label for rows
plt.show()
```



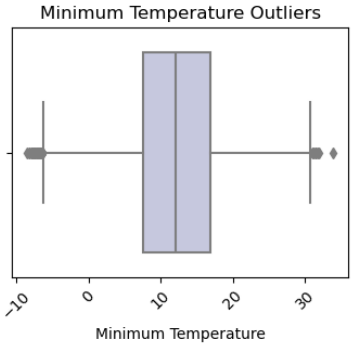
In [36]:
```
# Visualize missing values in numerical features with a bar plot
missing_values_plot = rain[numerical_features].isnull().sum().sort_values(ascending=False)
missing_values_plot.plot(kind='bar', color='#C2C4E2')
plt.title('Missing Values in Numerical Features')
plt.xlabel('Features')
plt.ylabel('Number of Missing Values')
plt.show()
```

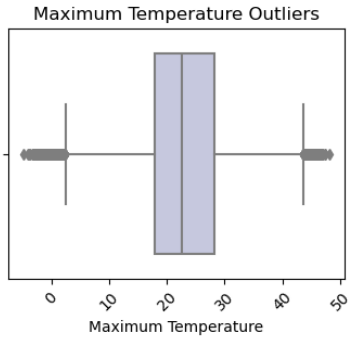


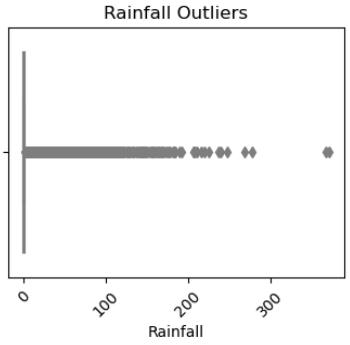
3.2.2 Outlier Detection in Numerical Features

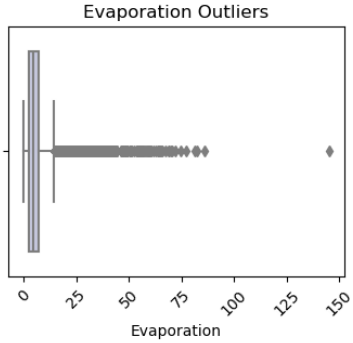
Outliers can significantly affect the performance of many statistical models by skewing the results. Thus, detecting and potentially addressing outliers is an essential step in data preprocessing. Box plots are highly effective for this purpose as they visually summarize the distribution of data, highlighting potential outliers.

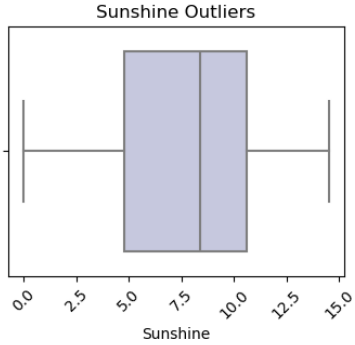
In [37]:
```
# Generate box plots for each numerical feature to identify outliers, optimized for smaller display
for feature in numerical_features:
    plt.figure(figsize=(4, 3))
    sns.boxplot(x=rain[feature])
    plt.title(feature + ' Outliers')
    plt.xticks(rotation=45)
    plt.show()
```

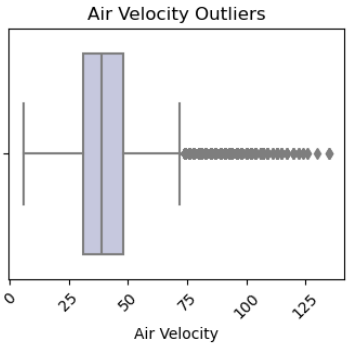


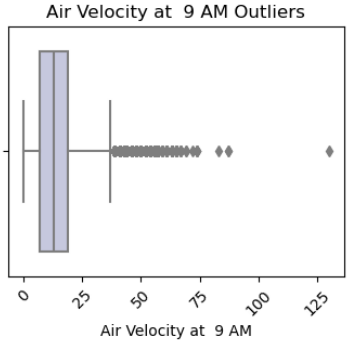


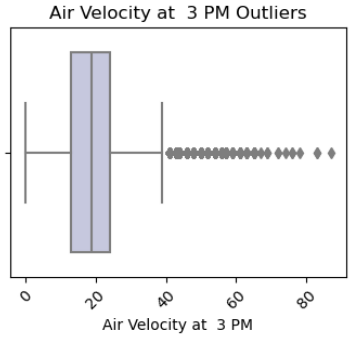

Rainfall Outliers
0
100
200
300
Rainfall

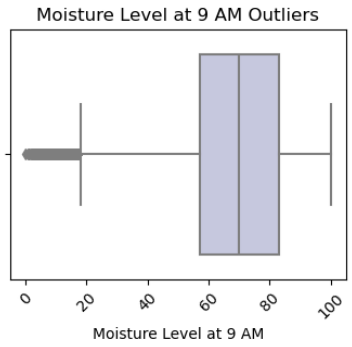

Evaporation Outliers
0
25
50
75
100
125
150
Evaporation

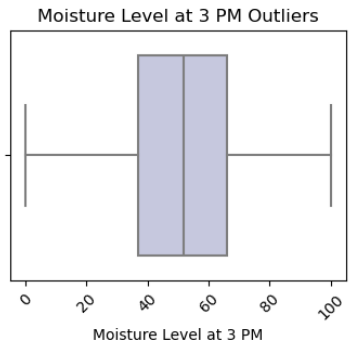

Sunshine Outliers
0.0
2.5
5.0
7.5
10.0
12.5
15.0
Sunshine

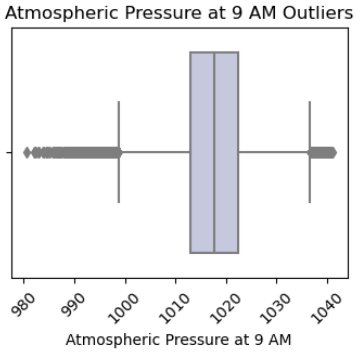

Air Velocity Outliers
0
25
50
75
100
125
Air Velocity

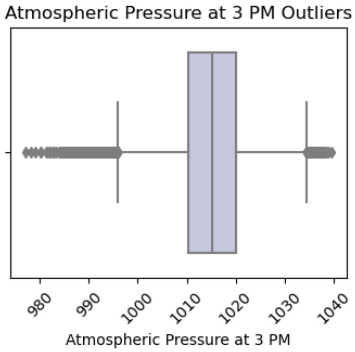

Air Velocity at 9 AM Outliers
0
25
50
75
100
125
Air Velocity at 9 AM

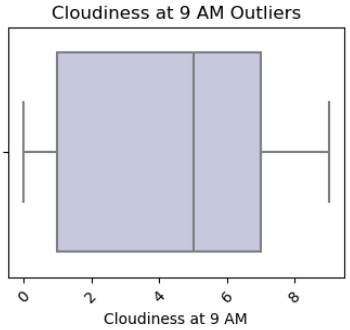

Air Velocity at 3 PM Outliers
0
20
40
60
80
Air Velocity at 3 PM

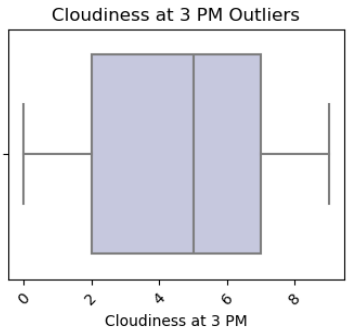

Moisture Level at 9 AM Outliers
0
20
40
60
80
100
Moisture Level at 9 AM

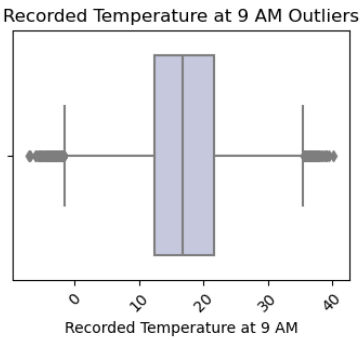

Moisture Level at 3 PM Outliers
0
20
40
60
80
100
Moisture Level at 3 PM

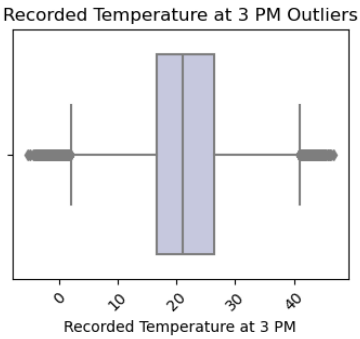

Atmospheric Pressure at 9 AM Outliers
980
990
1000
1010
1020
1030
1040
Atmospheric Pressure at 9 AM

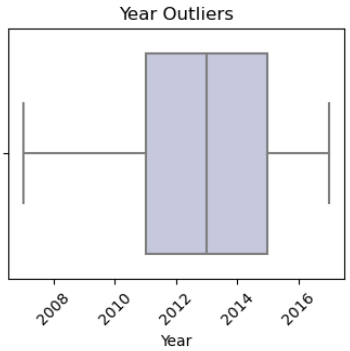

Atmospheric Pressure at 3 PM Outliers
980
990
1000
1010
1020
1030
1040
Atmospheric Pressure at 3 PM

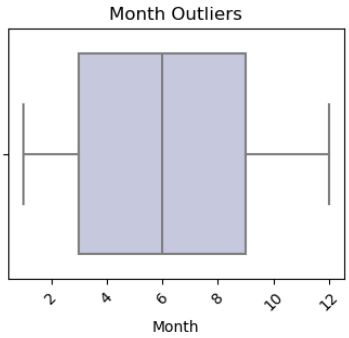

Cloudiness at 9 AM Outliers
0
2
4
6
8
Cloudiness at 9 AM

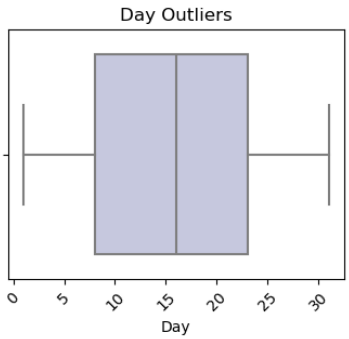

Cloudiness at 3 PM Outliers
0
2
4
6
8
Cloudiness at 3 PM


Recorded Temperature at 9 AM Outliers
0
10
20
30
40
Recorded Temperature at 9 AM


Recorded Temperature at 3 PM Outliers
0
10
20
30
40
Recorded Temperature at 3 PM


Year Outliers
2008
2010
2012
2014
2016
Year





In [38]:
```
# checking for outliers using the statistical formulas:

rain[numerical_features].describe()
```

Out[38]:

	Minimum Temperature	Maximum Temperature	Rainfall	Evaporation	Sunshine	Air Velocity	Air Velocity at 9 AM	Air Velocity at 3 PM	Moisture Level at 9 AM	Moisture Level at 3 PM	Atmosphe Pressure a
count	143975.000000	144199.000000	142199.000000	82670.000000	75625.000000	135197.000000	143693.000000	142398.000000	142806.000000	140953.000000	130395.00(
mean	12.194034	23.221348	2.360918	5.468232	7.611178	40.035230	14.043426	18.662657	68.880831	51.539116	1017.64
std	6.398495	7.119049	8.478060	4.193704	3.785483	13.607062	8.915375	8.809800	19.029164	20.795902	7.10
min	-8.500000	-4.800000	0.000000	0.000000	0.000000	6.000000	0.000000	0.000000	0.000000	0.000000	980.50
25%	7.600000	17.900000	0.000000	2.600000	4.800000	31.000000	7.000000	13.000000	57.000000	37.000000	1012.90
50%	12.000000	22.600000	0.000000	4.800000	8.400000	39.000000	13.000000	19.000000	70.000000	52.000000	1017.60
75%	16.900000	28.200000	0.800000	7.400000	10.600000	48.000000	19.000000	24.000000	83.000000	66.000000	1022.40
max	33.900000	48.100000	371.000000	145.000000	14.500000	135.000000	130.000000	87.000000	100.000000	100.000000	1041.00

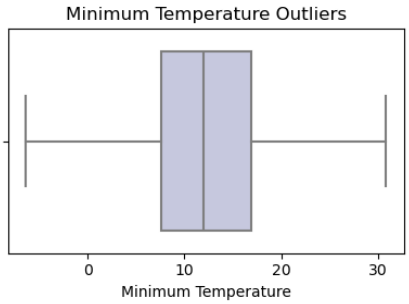
In [39]:
```
# features which has outliers:

features_with_outliers = ['Minimum Temperature', 'Maximum Temperature', 'Rainfall', 'Evaporation', 'Air Velocity','Air Velocity at  9 AM', 'Air Velocity a
```

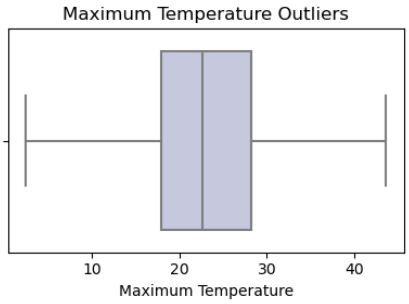
3.2.2.1 Replacing Outliers in Numerical Features Using IQR

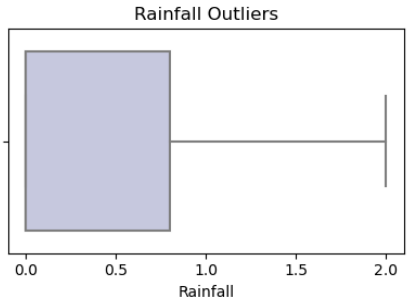
Outliers can significantly impact statistical analyses and predictive modeling. To mitigate this issue, outliers in the dataset are replaced using the Interquartile Range (IQR) method. This technique involves calculating the IQR, which helps in defining what constitutes an "outlier" and then capping these outliers to reduce their impact.

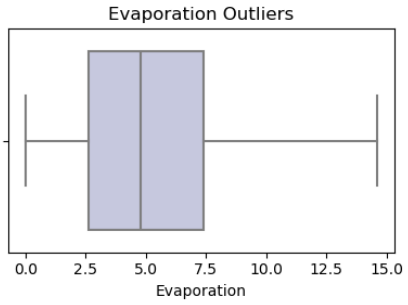
In [40]:
```
# Replacing outliers using IQR:

for feature in features_with_outliers:
    q1 = rain[feature].quantile(0.25)
    q3 = rain[feature].quantile(0.75)
    IQR = q3-q1
    lower_limit = q1 - (IQR*1.5)
    upper_limit = q3 + (IQR*1.5)
    rain.loc[rain[feature]<lower_limit,feature] = lower_limit
    rain.loc[rain[feature]>upper_limit,feature] = upper_limit
```

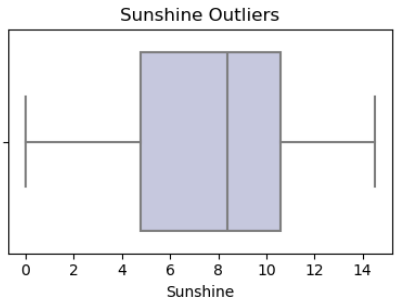
In [41]:
```
for feature in numerical_features:
    plt.figure(figsize=(4, 3))  # Reduced figure size to 4x3 inches
    sns.boxplot(x=rain[feature])
    plt.title(feature + ' Outliers')
    plt.tight_layout()  # Ensure everything fits without overlapping
    plt.show()
```

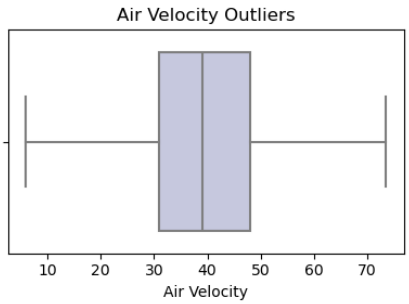

Minimum Temperature Outliers
0
10
20
30
Minimum Temperature

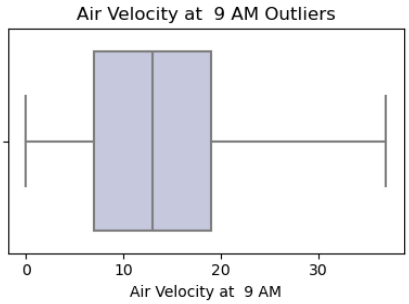

Maximum Temperature Outliers
10
20
30
40
Maximum Temperature

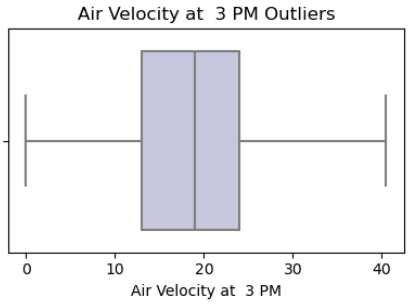

Rainfall Outliers
0.0
0.5
1.0
1.5
2.0
Rainfall

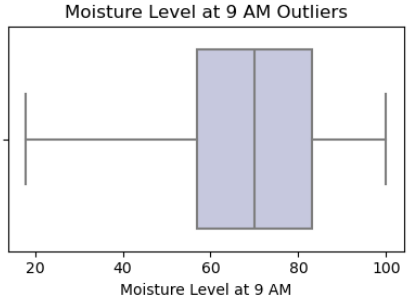

Evaporation Outliers
0.0
2.5
5.0
7.5
10.0
12.5
15.0
Evaporation

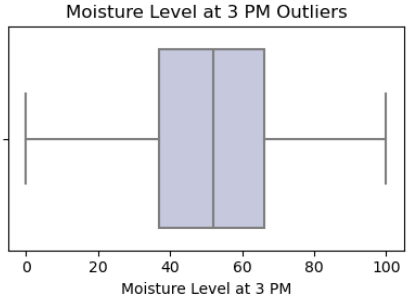

Sunshine Outliers
0
2
4
6
8
10
12
14
Sunshine

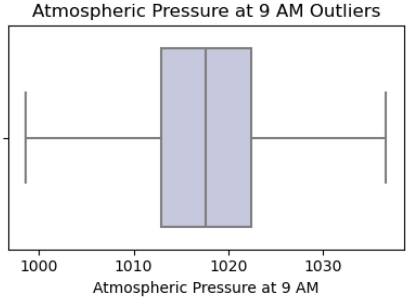

Air Velocity Outliers
10
20
30
40
50
60
70
Air Velocity

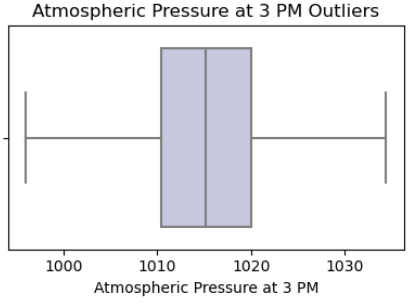

Air Velocity at 9 AM Outliers
0
10
20
30
Air Velocity at 9 AM

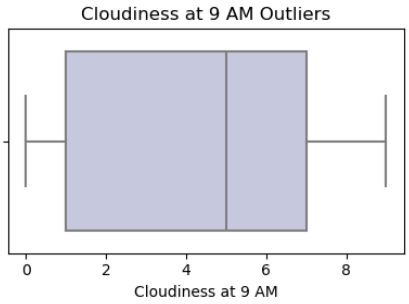

Air Velocity at 3 PM Outliers
0
10
20
30
40
Air Velocity at 3 PM

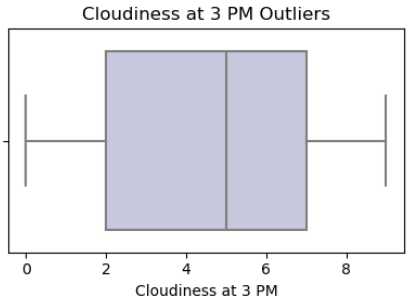

Moisture Level at 9 AM Outliers
20
40
60
80
100
Moisture Level at 9 AM

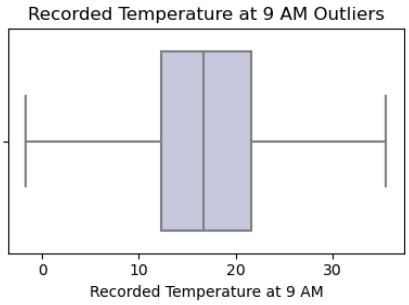

Moisture Level at 3 PM Outliers
0
20
40
60
80
100
Moisture Level at 3 PM

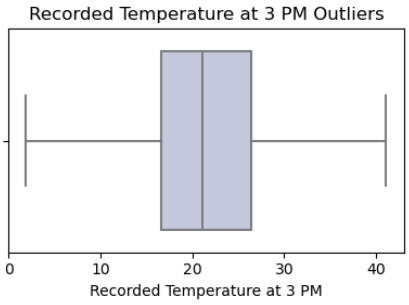

Atmospheric Pressure at 9 AM Outliers
1000
1010
1020
1030
Atmospheric Pressure at 9 AM

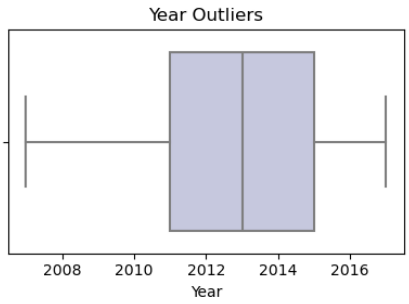

Atmospheric Pressure at 3 PM Outliers
1000
1010
1020
1030
Atmospheric Pressure at 3 PM

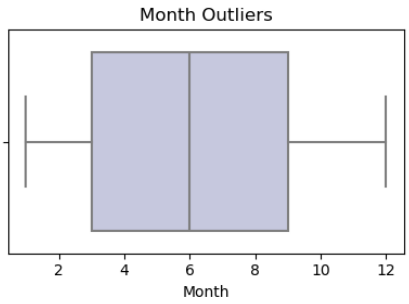

Cloudiness at 9 AM Outliers
0
2
4
6
8
Cloudiness at 9 AM

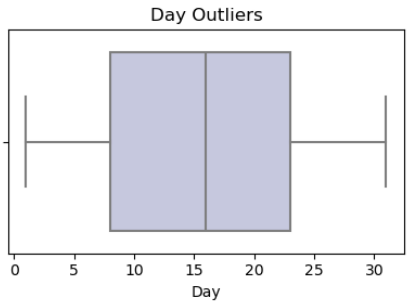

Cloudiness at 3 PM Outliers
0
2
4
6
8
Cloudiness at 3 PM


Recorded Temperature at 9 AM Outliers
0
10
20
30
Recorded Temperature at 9 AM


Recorded Temperature at 3 PM Outliers
0
10
20
30
40
Recorded Temperature at 3 PM


Year Outliers
2008
2010
2012
2014
2016
Year


Month Outliers
2
4
6
8
10
12
Month



In [42]:
```
# list of numerical Features with Null values:

numerical_features_with_null = [feature for feature in numerical_features if rain[feature].isnull().sum()]
numerical_features_with_null
```

Out[42]:
```
['Minimum Temperature',
 'Maximum Temperature',
 'Rainfall',
 'Evaporation',
 'Sunshine',
 'Air Velocity',
 'Air Velocity at  9 AM',
 'Air Velocity at  3 PM',
 'Moisture Level at 9 AM',
 'Moisture Level at 3 PM',
 'Atmospheric Pressure at 9 AM',
 'Atmospheric Pressure at 3 PM',
 'Cloudiness at 9 AM',
 'Cloudiness at 3 PM',
 'Recorded Temperature at 9 AM',
 'Recorded Temperature at 3 PM']
```

3.2.3 Imputation of Missing Values in Numerical Features Using Mean

Missing values in numerical features can disrupt statistical analysis and machine learning models. Imputing these missing values with the mean of the respective feature is a common strategy that helps maintain the overall distribution of the data.

In [43]:
```
# Filling null values uisng mean:

for feature in numerical_features_with_null:
    mean_value = rain[feature].mean()
    rain[feature].fillna(mean_value,inplace=True)
```

In [44]:
```
rain.isnull().sum()
```

Out[44]:
```
Weather Station                0
Minimum Temperature            0
Maximum Temperature            0
Rainfall                       0
Evaporation                    0
Sunshine                       0
Gust Trajectory                0
Air Velocity                   0
Gust Trajectory at  9 AM       0
Gust Trajectory at  3 PM       0
Air Velocity at  9 AM          0
Air Velocity at  3 PM          0
Moisture Level at 9 AM         0
Moisture Level at 3 PM         0
Atmospheric Pressure at 9 AM   0
Atmospheric Pressure at 3 PM   0
Cloudiness at 9 AM             0
Cloudiness at 3 PM             0
Recorded Temperature at 9 AM   0
Recorded Temperature at 3 PM   0
Rain that day                  0
Rain the day after             0
Year                           0
Month                          0
Day                            0
dtype: int64
```

As we can see there is no NULL value left in our DataFrame.

In [45]:
```
rain.head()
```

Out[45]:

	Weather Station	Minimum Temperature	Maximum Temperature	Rainfall	Evaporation	Sunshine	Gust Trajectory	Air Velocity	Gust Trajectory at 9 AM	Gust Trajectory at 3 PM	...	Atmospheric Pressure at 3 PM	Cloudiness at 9 AM	Cloudiness at 3 PM	Recorded Temperature at 9 AM	T
0	Station 2	13.4	22.9	0.6	5.318667	7.611178	W	44.0	W	WNW	...	1007.1	8.000000	4.50993	16.9	
1	Station 2	7.4	25.1	0.0	5.318667	7.611178	WNW	44.0	NNW	WSW	...	1007.8	4.447461	4.50993	17.2	
2	Station 2	12.9	25.7	0.0	5.318667	7.611178	WSW	46.0	W	WSW	...	1008.7	4.447461	2.00000	21.0	
3	Station 2	9.2	28.0	0.0	5.318667	7.611178	NE	24.0	SE	E	...	1012.8	4.447461	4.50993	18.1	
4	Station 2	17.5	32.3	1.0	5.318667	7.611178	W	41.0	ENE	NW	...	1006.0	7.000000	8.00000	17.8	

5 rows × 25 columns

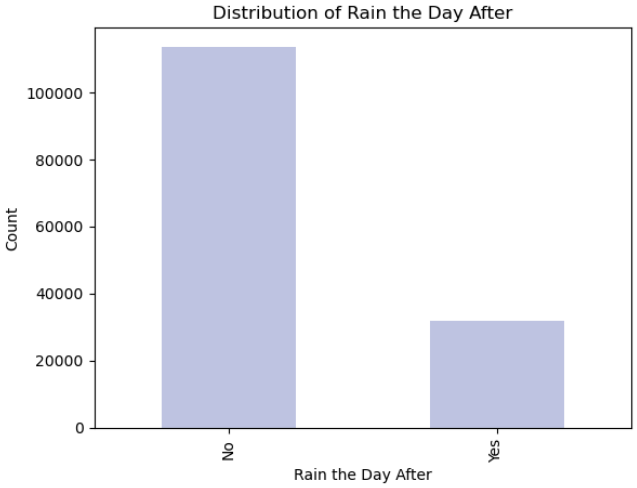
4 Exploring the Rain Labels

To understand the distribution and relationships within the dataset, we explore the target label Rain the day after and related variable Rain that day. Visualizing these labels provides insights into their distributions and potential patterns related to weather stations.

4.1 Distribution of Rain the day after

The distribution of the Rain the day after label is visualized to assess class balance and identify potential imbalances which may impact modeling.

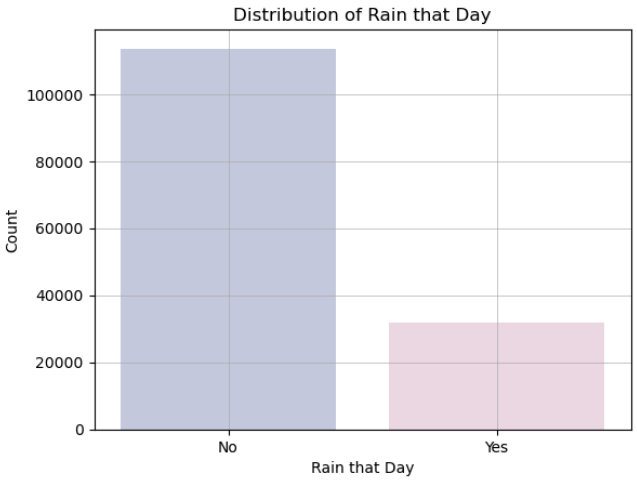
In [46]:
```
# Visualize the distribution of the 'Rain the day after' label
rain['Rain the day after'].value_counts().plot(kind='bar', color='#C2C4E2')
plt.title('Distribution of Rain the Day After')
plt.xlabel('Rain the Day After')
plt.ylabel('Count')
plt.show()
```



4.2 Analysis of Rain that day

The Rain that day variable is examined to understand its distribution within the dataset.

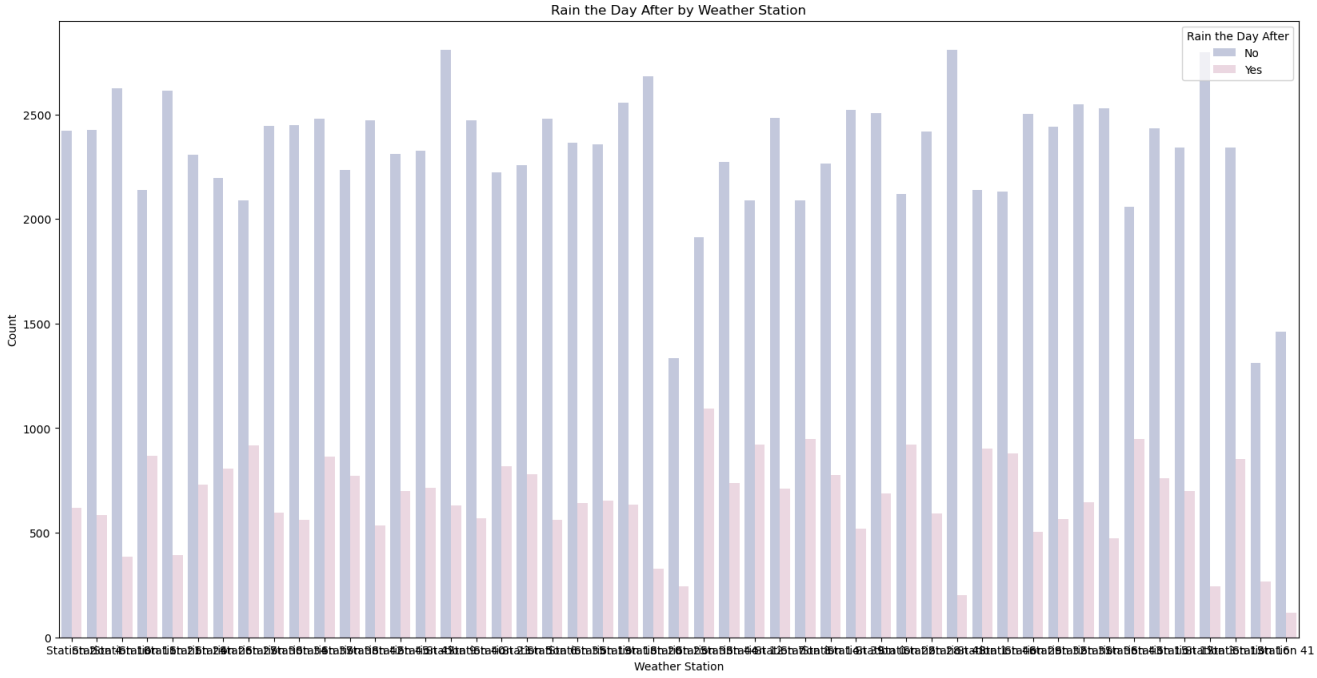
In [47]:
```
# Visualize the distribution of the 'Rain that day' variable
sns.countplot(data=rain, x="Rain that day", palette=["#C2C4E2", "#EED4E5"])
plt.title('Distribution of Rain that Day')
plt.xlabel('Rain that Day')
plt.ylabel('Count')
plt.grid(linewidth=0.5)
plt.show()
```



4.3 Distribution by Weather Station

The relationship between Rain the day after and different weather stations is visualized to identify any station-specific patterns or biases.

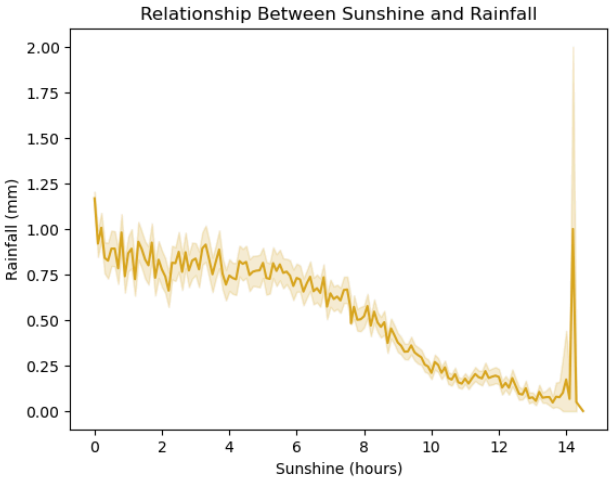
In [48]:
```
# Visualize the relationship between 'Rain the day after' and different weather stations
plt.figure(figsize=(20,10))
ax = sns.countplot(x="Weather Station", hue="Rain the day after", data=rain, palette=["#C2C4E2", "#EED4E5"])
plt.title('Rain the Day After by Weather Station')
plt.xlabel('Weather Station')
plt.ylabel('Count')
plt.legend(title='Rain the Day After', loc='upper right')
plt.show()
```



4.4 Exploring the Relationship Between Sunshine and Rainfall

Understanding the relationship between different weather variables is crucial for accurate predictive modeling. Here, we explore the relationship between Sunshine and Rainfall using a line plot, which can reveal trends and correlations between these two features.

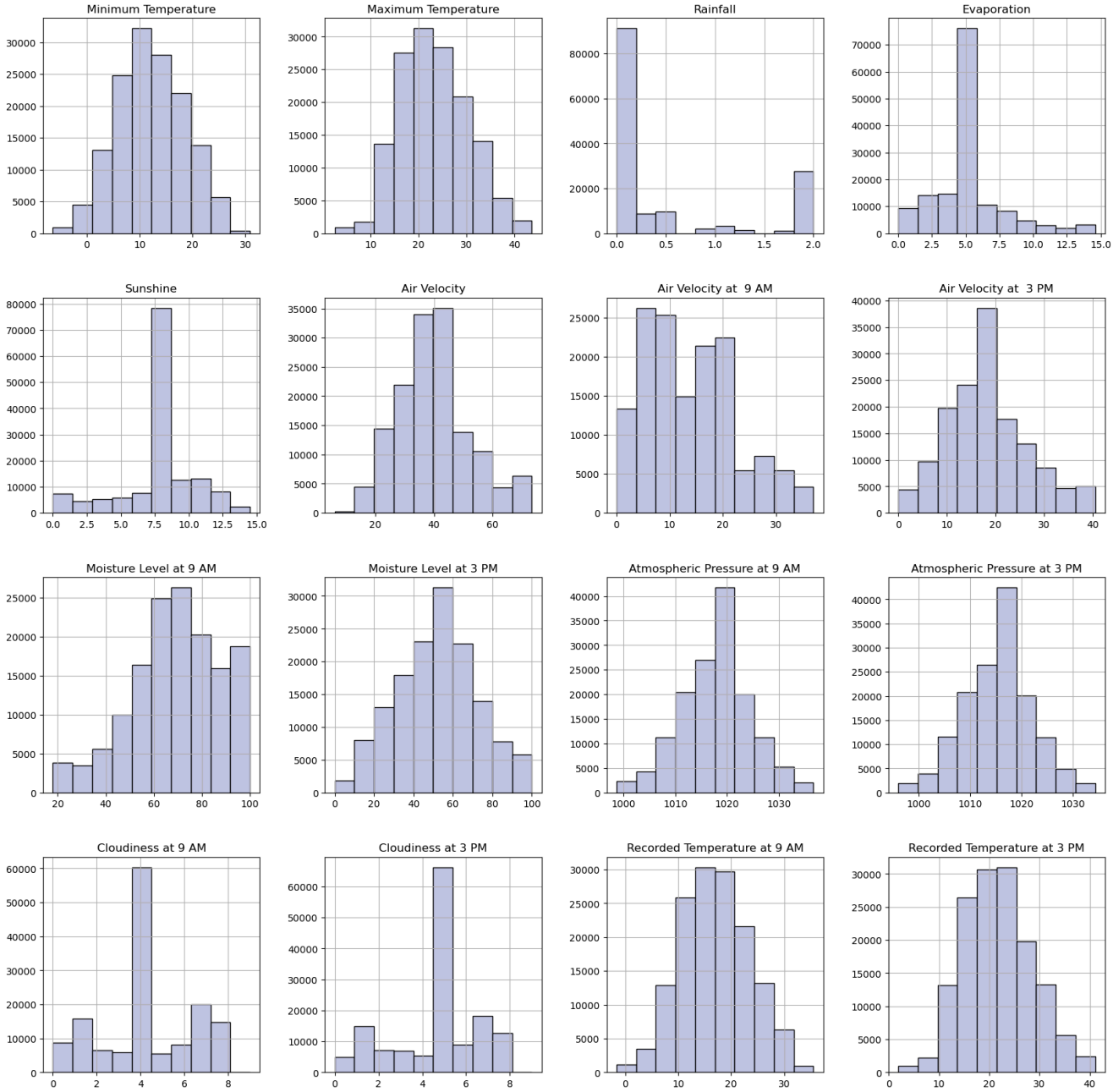
In [49]:
```
# Visualize the relationship between Sunshine and Rainfall
sns.lineplot(data=rain, x='Sunshine', y='Rainfall', color='goldenrod')
plt.title('Relationship Between Sunshine and Rainfall')
plt.xlabel('Sunshine (hours)')
plt.ylabel('Rainfall (mm)')
plt.show()
```



4.5 Distribution Analysis of Numerical Features

Understanding the distribution of numerical features is a fundamental step in data exploration. Histograms provide a clear visual representation of the frequency distribution of these features, helping to identify their underlying patterns, skewness, and potential outliers.

In [50]:
```
# Define the numerical features to be visualized
num_features = ['Minimum Temperature', 'Maximum Temperature', 'Rainfall', 'Evaporation', 'Sunshine', 'Air Velocity',
                'Air Velocity at  9 AM', 'Air Velocity at  3 PM', 'Moisture Level at 9 AM', 'Moisture Level at 3 PM',
                'Atmospheric Pressure at 9 AM', 'Atmospheric Pressure at 3 PM', 'Cloudiness at 9 AM',
                'Cloudiness at 3 PM', 'Recorded Temperature at 9 AM', 'Recorded Temperature at 3 PM']

# Visualize the distribution of numerical features using histograms
rain[num_features].hist(bins=10, figsize=(20, 20), color='#C2C4E2', edgecolor='black')
plt.suptitle('Histograms of Numerical Features', fontsize=20)
plt.show()
```



4.6 Data Type Verification and Correlation Analysis

Understanding the data types of each column and analyzing the correlations among numerical features are essential steps in data preprocessing. This process helps in identifying relationships between features that can be crucial for predictive modeling and feature selection.

Correlation is a statistic that helps to measure the strength of relationship between features.

In [51]:
```
# Check the data types of all columns in the DataFrame
print(rain.dtypes)
```

```
Weather Station                 object
Minimum Temperature            float64
Maximum Temperature            float64
Rainfall                       float64
Evaporation                    float64
Sunshine                       float64
Gust Trajectory                 object
Air Velocity                   float64
Gust Trajectory at  9 AM        object
Gust Trajectory at  3 PM        object
Air Velocity at  9 AM          float64
Air Velocity at  3 PM          float64
Moisture Level at 9 AM         float64
Moisture Level at 3 PM         float64
Atmospheric Pressure at 9 AM   float64
Atmospheric Pressure at 3 PM   float64
Cloudiness at 9 AM             float64
Cloudiness at 3 PM             float64
Recorded Temperature at 9 AM   float64
Recorded Temperature at 3 PM   float64
Rain that day                   object
Rain the day after              object
Year                             int32
Month                            int32
Day                              int32
dtype: object
```

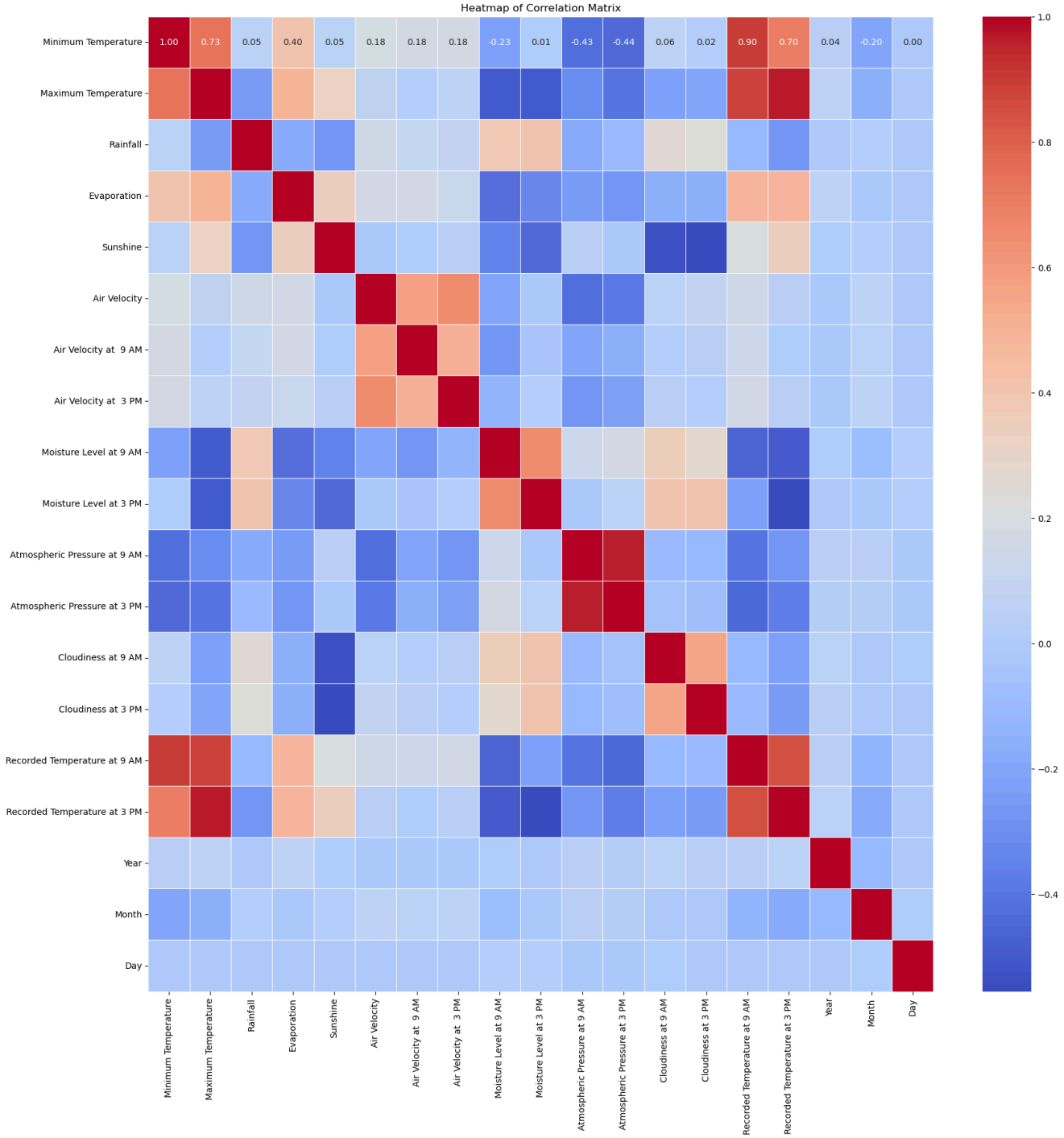
In [52]:
```
# Exclude non-numeric columns from correlation calculation
numeric_columns = rain.select_dtypes(include=['float64', 'int32'])

# Calculate correlation matrix
correlation_matrix = numeric_columns.corr()

# Transpose the correlation matrix
transposed_correlation_matrix = correlation_matrix.T

# Display the transposed correlation matrix
transposed_correlation_matrix
```

Out[52]:

	Minimum Temperature	Maximum Temperature	Rainfall	Evaporation	Sunshine	Air Velocity	Air Velocity at 9 AM	Air Velocity at 3 PM	Moisture Level at 9 AM	Moisture Level at 3 PM	Atmospheric Pressure at 9 AM	Atmospheric Pressure at 3 PM	Cloudiness at 9 AM
Minimum Temperature	1.000000	0.733392	0.051203	0.404632	0.051295	0.180336	0.177534	0.177176	-0.230718	0.006033	-0.428982	-0.437161	0.062154
Maximum Temperature	0.733392	1.000000	-0.255978	0.501465	0.329668	0.073229	0.017140	0.053117	-0.497111	-0.498544	-0.314920	-0.402168	-0.225781
Rainfall	0.051203	-0.255978	1.000000	-0.188968	-0.261889	0.151293	0.102499	0.080050	0.387454	0.406205	-0.179751	-0.095155	0.270881
Evaporation	0.404632	0.501465	-0.188968	1.000000	0.339024	0.170903	0.157072	0.113275	-0.421244	-0.318548	-0.248039	-0.269624	-0.168963
Sunshine	0.051295	0.329668	-0.261889	0.339024	1.000000	-0.019597	0.006782	0.041356	-0.349320	-0.443121	0.029821	-0.016637	-0.532497
Air Velocity	0.180336	0.073229	0.151293	0.170903	-0.019597	1.000000	0.578562	0.659350	-0.214349	-0.028868	-0.424191	-0.381720	0.050920
Air Velocity at 9 AM	0.177534	0.017140	0.102499	0.157072	0.006782	0.578562	1.000000	0.507874	-0.273399	-0.033036	-0.214427	-0.164884	0.018119
Air Velocity at 3 PM	0.177176	0.053117	0.080050	0.113275	0.041356	0.659350	0.507874	1.000000	-0.146602	0.015137	-0.276327	-0.237763	0.041044
Moisture Level at 9 AM	-0.230718	-0.497111	0.387454	-0.421244	-0.349320	-0.214349	-0.273399	-0.146602	1.000000	0.658850	0.133181	0.177319	0.354042
Moisture Level at 3 PM	0.006033	-0.498544	0.406205	-0.318548	-0.443121	-0.028868	-0.033036	0.015137	0.658850	1.000000	-0.024338	0.050405	0.398762
Atmospheric Pressure at 9 AM	-0.428982	-0.314920	-0.179751	-0.248039	0.029821	-0.424191	-0.214427	-0.276327	0.133181	-0.024338	1.000000	0.959981	-0.101138
Atmospheric Pressure at 3 PM	-0.437161	-0.402168	-0.095155	-0.269624	-0.016637	-0.381720	-0.164884	-0.237763	0.177319	0.050405	0.959981	1.000000	-0.047053
Cloudiness at 9 AM	0.062154	-0.225781	0.270881	-0.168963	-0.532497	0.050920	0.018119	0.041044	0.354042	0.398762	-0.101138	-0.047053	1.000000
Cloudiness at 3 PM	0.016724	-0.213214	0.234198	-0.167229	-0.553853	0.078253	0.039635	0.018156	0.273774	0.406605	-0.114087	-0.065433	0.559800
Recorded Temperature at 9 AM	0.897765	0.879365	-0.114011	0.477917	0.208714	0.154054	0.131738	0.165858	-0.468540	-0.216576	-0.403417	-0.446363	-0.109048
Recorded Temperature at 3 PM	0.699147	0.968905	-0.260283	0.485623	0.347710	0.039421	0.007197	0.031299	-0.489627	-0.555212	-0.272115	-0.366309	-0.232396
Year	0.043007	0.061734	-0.014480	0.060169	0.006072	-0.029014	-0.017855	-0.029658	0.009745	-0.009463	0.028290	0.023126	0.051533
Month	-0.202744	-0.163190	0.011964	-0.024221	0.017589	0.058281	0.051481	0.058906	-0.089070	-0.018568	0.034314	0.025093	-0.007930
Day	0.002342	0.000655	0.003290	-0.005429	-0.000286	-0.009631	-0.008879	-0.010331	0.015121	0.012813	-0.019915	-0.020277	0.005702

4.6.1 Create heatmap

In [53]:
```
# Create a heatmap to visualize the correlation matrix
plt.figure(figsize=(20, 20))
sns.heatmap(correlation_matrix, linewidths=0.5, annot=True, fmt=".2f", cmap='coolwarm')
plt.title('Heatmap of Correlation Matrix')
plt.show()
```



As we can see We have Correlation, But we Decided to Keep data as it is. I Googled it and they said If we are using inear models (e.g., linear regression or logistic regression) or Random Forests Correlation Features may Cause error. I visited these sites:

1. What we should do with highly correlated features?
2. What should I do if I have very little correlation between my all feature variable and target variable on my dataset?
3. In supervised learning, why is it bad to have correlated features?
4. https://medium.com/@abdallahashraf90x/all-you-need-to-know-about-correlation-for-machine-learning-e249fec292e9

and from things I read we can only delete datas if there is a column with high Correlation(we Don't have it here) :D

5 Encoding

In [54]:
```
rain.head()
```

Out[54]:

	Weather Station	Minimum Temperature	Maximum Temperature	Rainfall	Evaporation	Sunshine	Gust Trajectory	Air Velocity	Gust Trajectory at 9 AM	Gust Trajectory at 3 PM	...	Atmospheric Pressure at 3 PM	Cloudiness at 9 AM	Cloudiness at 3 PM	Recorded Temperature at 9 AM	T
0	Station 2	13.4	22.9	0.6	5.318667	7.611178	W	44.0	W	WNW	...	1007.1	8.000000	4.50993	16.9	
1	Station 2	7.4	25.1	0.0	5.318667	7.611178	WNW	44.0	NNW	WSW	...	1007.8	4.447461	4.50993	17.2	
2	Station 2	12.9	25.7	0.0	5.318667	7.611178	WSW	46.0	W	WSW	...	1008.7	4.447461	2.00000	21.0	
3	Station 2	9.2	28.0	0.0	5.318667	7.611178	NE	24.0	SE	E	...	1012.8	4.447461	4.50993	18.1	
4	Station 2	17.5	32.3	1.0	5.318667	7.611178	W	41.0	ENE	NW	...	1006.0	7.000000	8.00000	17.8	

5 rows × 25 columns

In [55]:
```
categorical_features
```

Out[55]:
```
['Weather Station',
 'Gust Trajectory',
 'Gust Trajectory at  9 AM',
 'Gust Trajectory at  3 PM',
 'Rain that day',
 'Rain the day after']
```

5.1 One-Hot Encoding of Categorical Features

To prepare the dataset for machine learning models, categorical features must be converted into a numerical format. One-hot encoding is employed to transform these categorical variables into a format that can be used effectively by most machine learning algorithms.

In [56]:
```
# One-hot encode categorical features
rain_proc = pd.get_dummies(data=rain, columns=['Rain that day', 'Gust Trajectory', 'Gust Trajectory at  9 AM', 'Gust Trajectory at  3 PM'])
rain_proc
```

Out[56]:

	Weather Station	Minimum Temperature	Maximum Temperature	Rainfall	Evaporation	Sunshine	Air Velocity	Air Velocity at 9 AM	Air Velocity at 3 PM	Moisture Level at 9 AM	...	Gust Trajectory at 3 PM_NNW	Gust Trajectory at 3 PM_NW	Gust Trajectory at 3 PM_S	Gust Trajectory at 3 PM_SE	Traje PM
0	Station 2	13.4	22.900000	0.6	5.318667	7.611178	44.000000	20.0	24.0	71.0	...	False	False	False	False	
1	Station 2	7.4	25.100000	0.0	5.318667	7.611178	44.000000	4.0	22.0	44.0	...	False	False	False	False	
2	Station 2	12.9	25.700000	0.0	5.318667	7.611178	46.000000	19.0	26.0	38.0	...	False	False	False	False	
3	Station 2	9.2	28.000000	0.0	5.318667	7.611178	24.000000	11.0	9.0	45.0	...	False	False	False	False	
4	Station 2	17.5	32.300000	1.0	5.318667	7.611178	41.000000	7.0	20.0	82.0	...	False	True	False	False	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
145455	Station 41	2.8	23.400000	0.0	5.318667	7.611178	31.000000	13.0	11.0	51.0	...	False	False	False	False	
145456	Station 41	3.6	25.300000	0.0	5.318667	7.611178	22.000000	13.0	9.0	56.0	...	False	False	False	False	
145457	Station 41	5.4	26.900000	0.0	5.318667	7.611178	37.000000	9.0	9.0	53.0	...	False	False	False	False	
145458	Station 41	7.8	27.000000	0.0	5.318667	7.611178	28.000000	13.0	7.0	51.0	...	False	False	False	False	
145459	Station 41	14.9	23.224781	0.0	5.318667	7.611178	39.837792	17.0	17.0	62.0	...	False	False	False	False	

145460 rows × 71 columns

5.2 Binary Encoding of Target Variable and Selection of Key Columns

To facilitate numerical analysis and modeling, the binary target variable Rain the day after is encoded from 'No'/'Yes' to 0/1. This transformation ensures that the target variable is in a suitable format for machine learning algorithms.

In [57]:
```
# Replace 'No'/'Yes' with 0/1 in the dataset
rain_proc.replace(['No', 'Yes'], [0, 1], inplace=True)

# Select and display specific columns for further analysis
selected_columns = rain_proc[['Year', 'Month', 'Day', 'Rain the day after']]
selected_columns.head()
```

Out[57]:

	Year	Month	Day	Rain the day after
0	2008	12	1	0
1	2008	12	2	0
2	2008	12	3	0
3	2008	12	4	0
4	2008	12	5	0

5.3 Encoding Categorical Data to Numerical Values

To facilitate numerical analysis and machine learning modeling, categorical data needs to be converted into numerical format. The encode_data function is designed to create a mapping dictionary for this purpose, systematically replacing categorical values with unique numerical identifiers.

In [58]:
```
def encode_data(feature_name):
    '''
    Function to map categorical data to numerical data.
```

```
    Parameters:
    feature_name (str): The name of the categorical feature to be encoded.

    Returns:
    dict: A dictionary mapping categorical values to numerical values.
    '''

    mapping_dict = {}
    unique_values = list(rain[feature_name].unique())
    for idx in range(len(unique_values)):
        mapping_dict[unique_values[idx]] = idx
    print(mapping_dict)
    return mapping_dict
```

5.4 Encoding the 'Weather Station' Feature

To prepare the 'Weather Station' feature for numerical analysis and machine learning models, it is essential to convert its categorical values into numerical format. The encode_data function is used to generate a mapping dictionary for this transformation.

In [59]:
```
# Encode the 'Weather Station' feature
rain_proc['Weather Station'].replace(encode_data('Weather Station'), inplace = True)
```

```
{'Station 2': 0, 'Station 4': 1, 'Station 10': 2, 'Station 11': 3, 'Station 21': 4, 'Station 24': 5, 'Station 26': 6, 'Station 27': 7, 'Station 30': 8, 'St
ation 34': 9, 'Station 37': 10, 'Station 38': 11, 'Station 42': 12, 'Station 45': 13, 'Station 47': 14, 'Station 9': 15, 'Station 40': 16, 'Station 23': 1
7, 'Station 5': 18, 'Station 6': 19, 'Station 35': 20, 'Station 19': 21, 'Station 18': 22, 'Station 20': 23, 'Station 25': 24, 'Station 33': 25, 'Station 4
4': 26, 'Station 12': 27, 'Station 7': 28, 'Station 8': 29, 'Station 14': 30, 'Station 39': 31, 'Station 0': 32, 'Station 22': 33, 'Station 28': 34, 'Stati
on 48': 35, 'Station 1': 36, 'Station 46': 37, 'Station 29': 38, 'Station 32': 39, 'Station 31': 40, 'Station 36': 41, 'Station 43': 42, 'Station 15': 43,
'Station 17': 44, 'Station 3': 45, 'Station 13': 46, 'Station 16': 47, 'Station 41': 48}
```

In [60]:
```
rain_proc.head()
```

Out[60]:

	Weather Station	Minimum Temperature	Maximum Temperature	Rainfall	Evaporation	Sunshine	Air Velocity	Air Velocity at 9 AM	Air Velocity at 3 PM	Moisture Level at 9 AM	...	Gust Trajectory at 3 PM_NNW	Gust Trajectory at 3 PM_NW	Gust Trajectory at 3 PM_S	Gust Trajectory at 3 PM_SE	Gust Trajectory at 3 PM_SSE	Tr
0	0	13.4	22.9	0.6	5.318667	7.611178	44.0	20.0	24.0	71.0	...	False	False	False	False	False	
1	0	7.4	25.1	0.0	5.318667	7.611178	44.0	4.0	22.0	44.0	...	False	False	False	False	False	
2	0	12.9	25.7	0.0	5.318667	7.611178	46.0	19.0	26.0	38.0	...	False	False	False	False	False	
3	0	9.2	28.0	0.0	5.318667	7.611178	24.0	11.0	9.0	45.0	...	False	False	False	False	False	
4	0	17.5	32.3	1.0	5.318667	7.611178	41.0	7.0	20.0	82.0	...	False	True	False	False	False	

5 rows × 71 columns

6 Split Data into Training and Testing Set

6.1 Separation of Features and Target Variable

To prepare the dataset for machine learning model training, it is essential to separate the features (independent variables) from the target variable (dependent variable). This step ensures that the dataset is correctly structured for model training and evaluation.

In [61]:
```
# Separate features and target variable
features = rain_proc.drop(columns='Rain the day after', axis=1)
Y = rain_proc['Rain the day after']
```

In [62]:
```
rain.drop('Weather Station', axis=1, inplace=True)
```

In [63]:
```
x_train, x_test, y_train, y_test = train_test_split(features, Y, test_size=0.2, random_state=10)
```

7 Training and Evaluation of K-Nearest Neighbors (KNN) Classifier

The K-Nearest Neighbors (KNN) algorithm is used to classify the target variable based on the feature set. This section outlines the steps taken to train the KNN model, make predictions, and evaluate its performance using various metrics.

In [ ]:
```
from sklearn.model_selection import cross_val_score
import matplotlib.pyplot as plt
from sklearn.neighbors import KNeighborsClassifier

# Define the range of k values to test
k_range = range(1, 31)
k_scores = []

# Perform cross-validation for each value of k
for k in k_range:
    knn = KNeighborsClassifier(n_neighbors=k)
    scores = cross_val_score(knn, x_train, y_train, cv=10, scoring='accuracy')
    k_scores.append(scores.mean())

# Plot the results
plt.figure(figsize=(10, 6))
plt.plot(k_range, k_scores, marker='o', color='blue')
plt.title('Cross-Validation Accuracy for Different k Values')
plt.xlabel('Number of Neighbors (k)')
plt.ylabel('Cross-Validated Accuracy')
plt.grid(True)
plt.show()

# Determine the best k
best_k = k_range[k_scores.index(max(k_scores))]
print(f'The best k value is {best_k} with a cross-validated accuracy of {max(k_scores):.4f}')
```

In [ ]:
```
# Train and evaluate K-Nearest Neighbors (KNN) classifier
knn = KNeighborsClassifier(n_neighbors=best_k)
knn.fit(x_train, y_train)
```

In [ ]:
```
knn_predictions = knn.predict(x_test)
```

In [ ]:
```
# Calculate performance metrics
knn_accuracy = accuracy_score(y_test, knn_predictions)
knn_precision = precision_score(y_test, knn_predictions, average='weighted')
knn_recall = recall_score(y_test, knn_predictions, average='weighted')
knn_f1_score = f1_score(y_test, knn_predictions, average='weighted')

# Display performance metrics
print('KNN Accuracy: ', knn_accuracy)
print('KNN Precision: ', knn_precision)
print('KNN Recall: ', knn_recall)
print('KNN F1 Score: ', knn_f1_score)
```

7.1 Training, Validation, and Testing of K-Nearest Neighbors (KNN) Classifier

The K-Nearest Neighbors (KNN) algorithm is evaluated using separate training, validation, and test sets. This comprehensive evaluation provides insights into the model's performance at different stages of the training process.

In [ ]:
```
# Split the data into training and test sets
x_train, x_test, y_train, y_test = train_test_split(features, Y, test_size=0.2, random_state=10)

# Further split the training set into training and validation sets
x_train1, x_val, y_train1, y_val = train_test_split(x_train, y_train, test_size=0.1, random_state=10)

# Train the KNN classifier on the training set
knn = KNeighborsClassifier(n_neighbors=best_k)
knn.fit(x_train1, y_train1)

# Evaluate the KNN classifier on the training set
knn_predictions_train = knn.predict(x_train1)
knn_accuracy_train = accuracy_score(y_train1, knn_predictions_train)

# Evaluate the KNN classifier on the validation set
knn_predictions_val = knn.predict(x_val)
knn_accuracy_val = accuracy_score(y_val, knn_predictions_val)

# Retrain the KNN classifier on the combined training and validation sets
knn.fit(x_train, y_train)

# Evaluate the KNN classifier on the test set
knn_predictions_test = knn.predict(x_test)
knn_accuracy_test = accuracy_score(y_test, knn_predictions_test)

# Display accuracy for different sets
print('KNN Accuracy for training set: ', knn_accuracy_train)
print('KNN Accuracy for validation set: ', knn_accuracy_val)
print('KNN Accuracy for test set: ', knn_accuracy_test)
```

Procedure Explanation:

- **Data Splitting**: The dataset is split into training (80%) and test (20%) sets. The training set is further split into training (90% of 80%) and validation (10% of 80%) sets.
- **Model Training**: The KNN classifier is trained on the initial training set (`x_train1` , `y_train1`).
- **Evaluation on Training Set**: Predictions are made on the training set, and accuracy is calculated.
- **Evaluation on Validation Set**: Predictions are made on the validation set, and accuracy is calculated.
- **Retraining and Final Evaluation**: The KNN classifier is retrained on the combined training and validation sets (`x_train` , `y_train`), and final predictions are made on the test set.

Implications for Model Performance:

- Evaluating the KNN model on separate training, validation, and test sets provides a comprehensive understanding of its performance and robustness.
- The validation set accuracy helps in tuning the model to achieve better performance and generalization.

7.2 Balancing Data Using Oversampling with Cross Validation

7.2.1 Introduction

In machine learning, class imbalance can pose challenges for predictive modeling, particularly in classification tasks. In this section, we explore the use of oversampling with SMOTE (Synthetic Minority Over-sampling Technique) coupled with cross-validation to address class imbalance.

7.2.2 Methodology

7.2.2.1 Data Splitting

The dataset is split into training and testing sets using the `train_test_split` function from the `sklearn.model_selection` module. Stratification based on the target variable ensures that the class distribution is preserved in both sets.

7.2.2.2 Oversampling with SMOTE

To address class imbalance, we employ SMOTE (Synthetic Minority Over-sampling Technique) to oversample the minority class. This is achieved using the `SMOTE` function from the `imblearn.over_sampling` module, followed by the `fit_resample` method to obtain a balanced dataset.

7.2.2.3 Cross-Validation

Stratified k-fold cross-validation is performed to assess the model's performance. We use the `StratifiedKFold` function to split the data into k folds while maintaining class proportions. The `cross_val_score` function calculates accuracy scores for each fold using the K-Nearest Neighbors (KNN) classifier with the optimal k value.

In [ ]:
```
# Balancing data using oversampling with cross validation
from sklearn.model_selection import StratifiedKFold, cross_val_score
from imblearn.over_sampling import SMOTE

x_train, x_test, y_train, y_test = train_test_split(features, Y, test_size=0.2, random_state=10, stratify=Y)

smote = SMOTE(random_state=10)
x_train_balanced, y_train_balanced = smote.fit_resample(x_train, y_train)

knn = KNeighborsClassifier(n_neighbors=best_k)

# Perform stratified k-fold cross-validation
skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=10)

cross_val_scores = cross_val_score(knn, x_train_balanced, y_train_balanced, cv=skf, scoring='accuracy')

print('Cross-validation accuracies: ', cross_val_scores)
print('Mean cross-validation accuracy: ', np.mean(cross_val_scores))
```

In [ ]:
```
knn.fit(x_train_balanced, y_train_balanced)

knn_test_predictions = knn.predict(x_test)
knn_accuracy_test = accuracy_score(y_test, knn_test_predictions)

print('KNN Accuracy for test set: ', knn_accuracy_test)
```

7.2.3 Results

7.2.3.1 Cross-Validation Accuracies

The accuracy scores obtained for each fold during cross-validation are as follows: [0.8039, 0.8060, 0.8068, 0.8052, 0.8040]. These scores reflect the model's performance on different subsets of the data.

7.2.3.2 Mean Cross-Validation Accuracy

The mean cross-validation accuracy, calculated as the average of the accuracy scores across folds, is approximately 0.8052. This metric provides an overall measure of the model's performance on unseen data.

7.2.4 KNN Accuracy for Test Set

The accuracy of the KNN classifier on the test set is 0.7471. This value indicates the model's performance on previously unseen data and serves as a practical evaluation metric.

7.3 Balancing Data Using Undersampling

7.3.1 Introduction

While oversampling techniques like SMOTE are effective for addressing class imbalance by generating synthetic samples for the minority class, undersampling methods involve reducing the number of instances in the majority class. In this section, we explore the use of undersampling with RandomUnderSampler to balance the dataset.

7.3.2 Methodology

7.3.2.1 Data Splitting

Similar to the previous section, the dataset is split into training and testing sets using the `train_test_split` function. Stratification based on the target variable ensures proportional representation of classes in both sets.

7.3.2.2 Undersampling with RandomUnderSampler

To address class imbalance, RandomUnderSampler is used to randomly remove instances from the majority class until a balanced dataset is obtained. This is achieved using the `RandomUnderSampler` function from the `imblearn.under_sampling` module, followed by the `fit_resample` method.

7.3.2.3 Cross-Validation

Stratified k-fold cross-validation is performed to assess the model's performance. The dataset is divided into k folds while maintaining class proportions. The K-Nearest Neighbors (KNN) classifier with a predefined number of neighbors (k=4) is used. Accuracy scores for each fold are calculated using the `cross_val_score` function.

In [ ]:
```
# Balacing but this time with undersampling
from imblearn.under_sampling import RandomUnderSampler

x_train, x_test, y_train, y_test = train_test_split(features, Y, test_size=0.2, random_state=10, stratify=Y)

under_sampler = RandomUnderSampler(random_state=10)
x_train_balanced, y_train_balanced = under_sampler.fit_resample(x_train, y_train)

knn = KNeighborsClassifier(n_neighbors=4)

skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=10)

cross_val_scores = cross_val_score(knn, x_train_balanced, y_train_balanced, cv=skf, scoring='accuracy')

print('Cross-validation accuracies: ', cross_val_scores)
print('Mean cross-validation accuracy: ', np.mean(cross_val_scores))
```

In [ ]:
```
knn.fit(x_train_balanced, y_train_balanced)

knn_test_predictions = knn.predict(x_test)
knn_accuracy_test = accuracy_score(y_test, knn_test_predictions)

print('KNN Accuracy for test set: ', knn_accuracy_test)
```

7.3.3 Results

7.3.3.1 Cross-Validation Accuracies

The accuracy scores obtained for each fold during cross-validation are as follows: [0.7288, 0.7277, 0.7325, 0.7167, 0.7330]. These scores reflect the model's performance on different subsets of the data.

7.3.3.2 Mean Cross-Validation Accuracy

The mean cross-validation accuracy, calculated as the average of the accuracy scores across folds, is approximately 0.7277. This metric provides an overall measure of the model's performance on unseen data.

7.3.4 KNN Accuracy for Test Set

The accuracy of the KNN classifier on the test set is 0.7914. This value indicates the model's performance on previously unseen data and serves as a practical evaluation metric.

7.4 Evaluating KNN Performance with Balanced Datasets

7.4.1 Introduction

This section focuses on assessing the performance of a K-Nearest Neighbors (KNN) classifier trained on balanced datasets obtained through oversampling and undersampling techniques. We compare the effectiveness of SMOTE (Synthetic Minority Over-sampling Technique) for oversampling and RandomUnderSampler for undersampling.

7.4.2 Methodology

7.4.2.1 Data Splitting

The dataset is split into training and testing subsets using the `train_test_split` function. Stratification ensures that the class distribution is maintained across both sets.

7.4.2.2 Balancing with Oversampling (SMOTE)

SMOTE is employed to generate synthetic samples for the minority class, resulting in a balanced training dataset. The KNN classifier is then trained on this balanced dataset.

7.4.2.3 Balancing with Undersampling (RandomUnderSampler)

RandomUnderSampler is utilized to reduce the size of the majority class in the training set, achieving a balanced dataset. Subsequently, the KNN classifier is trained on this balanced dataset.

In [ ]:
```
from imblearn.over_sampling import SMOTE
from imblearn.under_sampling import RandomUnderSampler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split

# Splitting the data into training and testing sets
x_train, x_test, y_train, y_test = train_test_split(features, Y, test_size=0.2, random_state=10, stratify=Y)

# Balancing data using oversampling (SMOTE)
smote = SMOTE(random_state=10)
x_train_balanced, y_train_balanced = smote.fit_resample(x_train, y_train)

# Training KNN on the balanced dataset
knn = KNeighborsClassifier(n_neighbors=5)  # You can adjust the value of k here
knn.fit(x_train_balanced, y_train_balanced)

# Evaluating on the test set
knn_test_predictions = knn.predict(x_test)
knn_accuracy_test = accuracy_score(y_test, knn_test_predictions)
print('KNN Accuracy for test set: ', knn_accuracy_test)


# Balancing data using undersampling (RandomUnderSampler)
under_sampler = RandomUnderSampler(random_state=10)
x_train_balanced, y_train_balanced = under_sampler.fit_resample(x_train, y_train)

# Training KNN on the balanced dataset
knn = KNeighborsClassifier(n_neighbors=5)  # You can adjust the value of k here
knn.fit(x_train_balanced, y_train_balanced)
```

```
# Evaluating on the test set
knn_test_predictions = knn.predict(x_test)
knn_accuracy_test = accuracy_score(y_test, knn_test_predictions)
print('KNN Accuracy for test set: ', knn_accuracy_test)
```

7.4.3 Results

The performance of the K-Nearest Neighbors (KNN) classifier on the test set after training on balanced datasets using oversampling (SMOTE) and undersampling (RandomUnderSampler) techniques is as follows:

- **KNN Accuracy for Oversampled Test Set:** 0.7442
- **KNN Accuracy for Undersampled Test Set:** 0.7543

These results indicate that the KNN classifier trained on the dataset balanced using undersampling achieved slightly higher accuracy on the test set compared to the oversampled dataset. However both of them are lower Than imbalanced result which was 0.8456577436169272.

8 Training and Evaluation of Decision Tree Classifier

The Decision Tree algorithm is employed to classify the target variable based on the feature set. This section outlines the steps taken to train the Decision Tree model, make predictions, and evaluate its performance using various metrics.

In [ ]:
```
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import cross_val_score
import matplotlib.pyplot as plt

# Define the range of max_depth values to test
max_depth_range = range(1, 21)
depth_scores = []

# Perform cross-validation for each value of max_depth
for depth in max_depth_range:
    tree = DecisionTreeClassifier(criterion="entropy", max_depth=depth)
    scores = cross_val_score(tree, x_train, y_train, cv=10, scoring='accuracy')
    depth_scores.append(scores.mean())

# Plot the results
plt.figure(figsize=(10, 6))
plt.plot(max_depth_range, depth_scores, marker='o', color='blue')
plt.title('Cross-Validation Accuracy for Different Maximum Depths')
plt.xlabel('Maximum Depth')
plt.ylabel('Cross-Validated Accuracy')
plt.grid(True)
plt.show()

# Determine the best max_depth
best_max_depth = max_depth_range[np.argmax(depth_scores)]
print(f'The best max_depth value is {best_max_depth} with a cross-validated accuracy of {max(depth_scores):.4f}')
```

In [ ]:
```
# Train and evaluate Decision Tree classifier
tree = DecisionTreeClassifier(criterion="entropy", max_depth=best_max_depth)
tree.fit(x_train, y_train)
```

In [ ]:
```
tree_predictions = tree.predict(x_test)
```

In [ ]:
```
# Calculate performance metrics
tree_accuracy = accuracy_score(y_test, tree_predictions)
tree_precision = precision_score(y_test, tree_predictions, average='weighted')
tree_recall = recall_score(y_test, tree_predictions, average='weighted')
tree_f1_score = f1_score(y_test, tree_predictions, average='weighted')

# Display performance metrics
print('Decision Tree Accuracy: ', tree_accuracy)
print('Decision Tree Precision: ', tree_precision)
print('Decision Tree Recall: ', tree_recall)
print('Decision Tree F1 Score: ', tree_f1_score)
```

8.1 Training, Validation, and Testing of Decision Tree Classifier

The Decision Tree algorithm is evaluated using separate training, validation, and test sets. This comprehensive evaluation provides insights into the model's performance at different stages of the training process.

In [ ]:
```
# Split the data into training and test sets
x_train, x_test, y_train, y_test = train_test_split(features, Y, test_size=0.2, random_state=10)

# Further split the training set into training and validation sets
x_train1, x_val, y_train1, y_val = train_test_split(x_train, y_train, test_size=0.1, random_state=10)

# Train the Decision Tree classifier on the training set
tree = DecisionTreeClassifier(criterion="entropy", max_depth=best_max_depth)
tree.fit(x_train1, y_train1)

# Evaluate the Decision Tree classifier on the training set
tree_predictions_train = tree.predict(x_train1)
tree_accuracy_train = accuracy_score(y_train1, tree_predictions_train)

# Evaluate the Decision Tree classifier on the validation set
tree_predictions_val = tree.predict(x_val)
tree_accuracy_val = accuracy_score(y_val, tree_predictions_val)

# Retrain the Decision Tree classifier on the combined training and validation sets
tree.fit(x_train, y_train)

# Evaluate the Decision Tree classifier on the test set
tree_predictions_test = tree.predict(x_test)
```

```
tree_accuracy_test = accuracy_score(y_test, tree_predictions_test)

# Calculate additional performance metrics on the test set
tree_precision = precision_score(y_test, tree_predictions_test, average='weighted')
tree_recall = recall_score(y_test, tree_predictions_test, average='weighted')
tree_f1_score = f1_score(y_test, tree_predictions_test, average='weighted')

# Display accuracy for different sets
print('Decision Tree Accuracy for training set: ', tree_accuracy_train)
print('Decision Tree Accuracy for validation set: ', tree_accuracy_val)
print('Decision Tree Accuracy for test set: ', tree_accuracy_test)
print('Decision Tree Precision: ', tree_precision)
print('Decision Tree Recall: ', tree_recall)
print('Decision Tree F1 Score: ', tree_f1_score)
```

Procedure Explanation:

- **Data Splitting**: The dataset is split into training (80%) and test (20%) sets. The training set is further split into training (90% of 80%) and validation (10% of 80%) sets.
- **Model Training**: The Decision Tree classifier is initialized with `criterion="entropy"` and `max_depth=8`, and trained on the initial training set (`x_train1`, `y_train1`).
- **Evaluation on Training Set**: Predictions are made on the training set, and accuracy is calculated.
- **Evaluation on Validation Set**: Predictions are made on the validation set, and accuracy is calculated.
- **Retraining and Final Evaluation**: The Decision Tree classifier is retrained on the combined training and validation sets (`x_train`, `y_train`), and final predictions are made on the test set.
- **Additional Metrics**: Precision, recall, and F1 score are calculated for the test set to provide a comprehensive performance evaluation.

8.2 Balancing Data Using Oversampling with Cross-Validation

8.2.1 Introduction

Class imbalance is a common issue in machine learning datasets that can significantly affect model performance. In this section, we address class imbalance by employing oversampling with SMOTE (Synthetic Minority Over-sampling Technique) combined with cross-validation. We use a Decision Tree classifier and optimize its hyperparameter, max_depth, to achieve better model performance.

8.2.2 Methodology

8.2.2.1 Data Splitting

The dataset is split into training and testing sets using the `train_test_split` function. Stratification ensures that the class distribution is preserved in both sets.

8.2.2.2 Oversampling with SMOTE

SMOTE is applied to the training set to generate synthetic samples for the minority class, resulting in a balanced dataset.

8.2.2.3 Hyperparameter Tuning

We define a range of values for the `max_depth` hyperparameter of the Decision Tree classifier and perform cross-validation for each value. The goal is to find the optimal `max_depth` that maximizes the model's performance.

In [ ]:
```
# Balancing data using oversampling with cross-validation
x_train, x_test, y_train, y_test = train_test_split(features, Y, test_size=0.2, random_state=10, stratify=Y)

smote = SMOTE(random_state=10)
x_train_balanced, y_train_balanced = smote.fit_resample(x_train, y_train)

# Define the range of max_depth values to test
max_depth_range = range(1, 11)
depth_scores = []

# Perform cross-validation for each value of max_depth
for depth in max_depth_range:
    tree = DecisionTreeClassifier(criterion="entropy", max_depth=depth)
    skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=10)
    scores = cross_val_score(tree, x_train_balanced, y_train_balanced, cv=skf, scoring='accuracy')
    depth_scores.append(scores.mean())

# Determine the best max_depth
best_max_depth = max_depth_range[np.argmax(depth_scores)]
print(f'The best max_depth value is {best_max_depth} with a cross-validated accuracy of {max(depth_scores):.4f}')

# Train Decision Tree with best max_depth
tree = DecisionTreeClassifier(criterion="entropy", max_depth=best_max_depth)
tree.fit(x_train_balanced, y_train_balanced)

# Evaluate on test set
tree_test_predictions = tree.predict(x_test)
tree_accuracy_test = accuracy_score(y_test, tree_test_predictions)
print('Decision Tree Accuracy for test set: ', tree_accuracy_test)
```

8.2.3 Results

8.2.3.1 Best Max Depth Selection

For the Decision Tree classifier, the best `max_depth` value is determined to be 10 with a cross-validated accuracy of 0.8309.

8.2.3.2 Decision Tree Accuracy on Test Set

The Decision Tree classifier trained with the optimal `max_depth` is evaluated on the test set, achieving an accuracy of 0.8083.

8.3 Balancing Data Using Undersampling with Cross-Validation

8.3.1 Introduction

Class imbalance is a common challenge in machine learning, and undersampling techniques are one approach to address it. In this section, we explore the use of RandomUnderSampler combined with cross-validation to balance the dataset and improve model performance.

8.3.2 Methodology

8.3.2.1 Data Splitting

The dataset is split into training and testing sets using the `train_test_split` function. Stratification ensures that the class distribution is preserved in both sets.

8.3.2.2 Undersampling with RandomUnderSampler

RandomUnderSampler is applied to the training set to reduce the number of instances in the majority class, resulting in a balanced dataset.

8.3.2.3 Hyperparameter Tuning

We define a range of values for the `max_depth` hyperparameter of the Decision Tree classifier and perform cross-validation for each value. The goal is to find the optimal `max_depth` that maximizes the model's performance.

In [ ]:
```
# Balancing data using undersampling with cross-validation
x_train, x_test, y_train, y_test = train_test_split(features, Y, test_size=0.2, random_state=10, stratify=Y)

under_sampler = RandomUnderSampler(random_state=10)
x_train_balanced, y_train_balanced = under_sampler.fit_resample(x_train, y_train)

# Define the range of max_depth values to test
max_depth_range = range(1, 11)
depth_scores = []

# Perform cross-validation for each value of max_depth
for depth in max_depth_range:
    tree = DecisionTreeClassifier(criterion="entropy", max_depth=depth)
    skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=10)
    scores = cross_val_score(tree, x_train_balanced, y_train_balanced, cv=skf, scoring='accuracy')
    depth_scores.append(scores.mean())

# Determine the best max_depth
best_max_depth = max_depth_range[np.argmax(depth_scores)]
print(f'The best max_depth value is {best_max_depth} with a cross-validated accuracy of {max(depth_scores):.4f}')

# Train Decision Tree with best max_depth
tree = DecisionTreeClassifier(criterion="entropy", max_depth=best_max_depth)
tree.fit(x_train_balanced, y_train_balanced)

# Evaluate on test set
tree_test_predictions = tree.predict(x_test)
tree_accuracy_test = accuracy_score(y_test, tree_test_predictions)
print('Decision Tree Accuracy for test set: ', tree_accuracy_test)
```

8.3.3 Results

8.3.3.1 Best Max Depth Selection

For the Decision Tree classifier, the best `max_depth` value is determined to be 8 with a cross-validated accuracy of 0.7458.

8.3.3.2 Decision Tree Accuracy on Test Set

The Decision Tree classifier trained with the optimal `max_depth` is evaluated on the test set, achieving an accuracy of 0.7313.

8.4 Evaluating Decision Tree Performance with Balanced Datasets

8.4.1 Introduction

In this section, we examine the performance of a Decision Tree classifier trained on balanced datasets achieved through both oversampling and undersampling techniques. We compare the effectiveness of SMOTE (Synthetic Minority Over-sampling Technique) for oversampling and RandomUnderSampler for undersampling.

8.4.2 Methodology

8.4.2.1 Data Splitting

The dataset is partitioned into training and testing subsets using the `train_test_split` function. Stratification ensures that the class distribution remains consistent across both sets.

8.4.2.2 Balancing with Oversampling (SMOTE)

SMOTE is utilized to generate synthetic samples for the minority class, resulting in a balanced training dataset. The Decision Tree classifier is then trained on this balanced dataset.

8.4.2.3 Balancing with Undersampling (RandomUnderSampler)

RandomUnderSampler is employed to reduce the size of the majority class in the training set, achieving a balanced dataset. Subsequently, the Decision Tree classifier is trained on this balanced dataset.

In [ ]:
```
from imblearn.over_sampling import SMOTE
from imblearn.under_sampling import RandomUnderSampler
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split
```

```
# Splitting the data into training and testing sets
x_train, x_test, y_train, y_test = train_test_split(features, Y, test_size=0.2, random_state=10, stratify=Y)

# Balancing data using oversampling (SMOTE)
smote = SMOTE(random_state=10)
x_train_balanced, y_train_balanced = smote.fit_resample(x_train, y_train)

# Training Decision Tree on the balanced dataset
tree = DecisionTreeClassifier(criterion="entropy", max_depth=8)  # You can adjust the max_depth here
tree.fit(x_train_balanced, y_train_balanced)

# Evaluating on the test set
tree_test_predictions = tree.predict(x_test)
tree_accuracy_test = accuracy_score(y_test, tree_test_predictions)
print('Decision Tree Accuracy for test set: ', tree_accuracy_test)


# Balancing data using undersampling (RandomUnderSampler)
under_sampler = RandomUnderSampler(random_state=10)
x_train_balanced, y_train_balanced = under_sampler.fit_resample(x_train, y_train)

# Training Decision Tree on the balanced dataset
tree = DecisionTreeClassifier(criterion="entropy", max_depth=8)  # You can adjust the max_depth here
tree.fit(x_train_balanced, y_train_balanced)

# Evaluating on the test set
tree_test_predictions = tree.predict(x_test)
tree_accuracy_test = accuracy_score(y_test, tree_test_predictions)
print('Decision Tree Accuracy for test set: ', tree_accuracy_test)
```

8.4.3 Results

The performance of the Decision Tree classifier on the test set after training on balanced datasets using oversampling (SMOTE) and undersampling (RandomUnderSampler) techniques is as follows:

- **Decision Tree Accuracy for Oversampled Test Set:** 0.7967
- **Decision Tree Accuracy for Undersampled Test Set:** 0.7313

These results indicate that the Decision Tree classifier trained on the dataset balanced using oversampling achieved higher accuracy on the test set compared to the undersampled dataset. However both of them are lower Than imbalanced result which was 0.8380598992843891

9 Training and Evaluation of Support Vector Machine (SVM) Classifier

The Support Vector Machine (SVM) algorithm is employed to classify the target variable based on the feature set. This section outlines the steps taken to train the SVM model, make predictions, and evaluate its performance using various metrics.

In [ ]:
```
# Train and evaluate Support Vector Machine (SVM) classifier
svm_model = svm.SVC(kernel='linear')
svm_model.fit(x_train, y_train)
```

In [ ]:
```
svm_predictions = svm_model.predict(x_test)
```

In [ ]:
```
# Calculate performance metrics
svm_accuracy = accuracy_score(y_test, svm_predictions)
svm_precision = precision_score(y_test, svm_predictions, average='weighted')
svm_recall = recall_score(y_test, svm_predictions, average='weighted')
svm_f1_score = f1_score(y_test, svm_predictions, average='weighted')

# Display performance metrics
print('SVM Accuracy: ', svm_accuracy)
print('SVM Precision: ', svm_precision)
print('SVM Recall: ', svm_recall)
print('SVM F1 Score: ', svm_f1_score)
```

9.1 Training, Validation, and Testing of SVM Classifier

The SVM algorithm is evaluated using separate training, validation, and test sets to provide insights into the model's performance at different stages of the training process.

In [ ]:
```
# Split the data into training and test sets
x_train, x_test, y_train, y_test = train_test_split(features, Y, test_size=0.2, random_state=10)

# Further split the training set into training and validation sets
x_train1, x_val, y_train1, y_val = train_test_split(x_train, y_train, test_size=0.1, random_state=10)

# Train the SVM classifier on the training set
svm_model = svm.SVC(kernel='linear')
svm_model.fit(x_train1, y_train1)

# Evaluate the SVM classifier on the training set
svm_predictions_train = svm_model.predict(x_train1)
svm_accuracy_train = accuracy_score(y_train1, svm_predictions_train)

# Evaluate the SVM classifier on the validation set
svm_predictions_val = svm_model.predict(x_val)
svm_accuracy_val = accuracy_score(y_val, svm_predictions_val)

# Retrain the SVM classifier on the combined training and validation sets
svm_model.fit(x_train, y_train)

# Evaluate the SVM classifier on the test set
svm_predictions_test = svm_model.predict(x_test)
svm_accuracy_test = accuracy_score(y_test, svm_predictions_test)

# Calculate additional performance metrics on the test set
svm_precision = precision_score(y_test, svm_predictions_test, average='weighted')
svm_recall = recall_score(y_test, svm_predictions_test, average='weighted')
```

```
svm_f1_score = f1_score(y_test, svm_predictions_test, average='weighted')

# Display accuracy for different sets
print('SVM Accuracy for training set: ', svm_accuracy_train)
print('SVM Accuracy for validation set: ', svm_accuracy_val)
print('SVM Accuracy for test set: ', svm_accuracy_test)
print('SVM Precision: ', svm_precision)
print('SVM Recall: ', svm_recall)
print('SVM F1 Score: ', svm_f1_score)
```

Procedure Explanation:

- **Data Splitting**: The dataset is split into training (80%) and test (20%) sets. The training set is further split into training (90% of 80%) and validation (10% of 80%) sets.
- **Model Training**: The SVM classifier with a linear kernel is trained on the initial training set (`x_train1`, `y_train1`).
- **Evaluation on Training Set**: Predictions are made on the training set, and accuracy is calculated.
- **Evaluation on Validation Set**: Predictions are made on the validation set, and accuracy is calculated.
- **Retraining and Final Evaluation**: The SVM classifier is retrained on the combined training and validation sets (`x_train`, `y_train`), and final predictions are made on the test set.
- **Additional Metrics**: Precision, recall, and F1 score are calculated for the test set to provide a comprehensive performance evaluation.

9.2 Balancing Data Using Oversampling with Cross-Validation for SVM

9.2.1 Introduction

Addressing class imbalance is crucial for robust machine learning models, particularly for algorithms like Support Vector Machines (SVM). In this section, we delve into a methodology that leverages oversampling with SMOTE along with cross-validation to tackle class imbalance and enhance SVM's performance.

9.2.2 Methodology

9.2.2.1 Data Splitting

The dataset undergoes a standard split into training and testing subsets using the `train_test_split` function. Stratification ensures that the class distribution remains consistent across both sets.

9.2.2.2 Oversampling with SMOTE

SMOTE is employed to augment the minority class within the training set, rectifying class imbalance. SMOTE synthesizes new instances by interpolating between existing minority class samples.

9.2.2.3 Hyperparameter Tuning

We embark on a quest to optimize SVM's performance by tuning the hyperparameter `C`, representing the regularization strength. A range of `C` values is tested, and cross-validation is employed to evaluate the model's accuracy for each value.

In [ ]:
```
from sklearn.model_selection import StratifiedKFold, cross_val_score, train_test_split
from imblearn.over_sampling import SMOTE
from imblearn.under_sampling import RandomUnderSampler
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score
import numpy as np

# Balancing data using oversampling with cross-validation
x_train, x_test, y_train, y_test = train_test_split(features, Y, test_size=0.2, random_state=10, stratify=Y)

smote = SMOTE(random_state=10)
x_train_balanced, y_train_balanced = smote.fit_resample(x_train, y_train)

# Define the range of C values to test
C_range = [0.1, 1, 10]
C_scores = []

# Perform cross-validation for each value of C
for C_value in C_range:
    svm = SVC(C=C_value, kernel='linear')
    skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=10)
    scores = cross_val_score(svm, x_train_balanced, y_train_balanced, cv=skf, scoring='accuracy')
    C_scores.append(scores.mean())

# Determine the best C value
best_C = C_range[np.argmax(C_scores)]
print(f'The best C value is {best_C} with a cross-validated accuracy of {max(C_scores):.4f}')

# Train SVM with best C value
svm = SVC(C=best_C, kernel='linear')
svm.fit(x_train_balanced, y_train_balanced)

# Evaluate on test set
svm_test_predictions = svm.predict(x_test)
svm_accuracy_test = accuracy_score(y_test, svm_test_predictions)
print('SVM Accuracy for test set: ', svm_accuracy_test)
```

9.2.3 Results

After applying oversampling with SMOTE and cross-validation to balance the dataset and tune the hyperparameter `C` for the Support Vector Machine (SVM) classifier, the following results were obtained:

- **The best C value:** 0.1
- **Cross-validated accuracy with the best C value:** 0.8894
- **SVM Accuracy for test set:** 0.8388

These results indicate that the SVM classifier achieved a high accuracy on the test set after balancing the data using oversampling with SMOTE and selecting the optimal value of the hyperparameter `C`. However it's lower Than imbalanced result which was 0.8440674971287216

9.3 Balancing Data Using Undersampling with Cross-Validation for SVM

9.3.1 Introduction

Class imbalance can significantly impact the performance of machine learning models, necessitating effective strategies to address it. In this section, we explore the utilization of undersampling with cross-validation to mitigate class imbalance specifically for Support Vector Machines (SVM).

9.3.2 Methodology

9.3.2.1 Data Splitting

The dataset is partitioned into training and testing subsets using the `train_test_split` function. Stratification ensures that the class distribution remains consistent across both sets.

9.3.2.2 Undersampling with RandomUnderSampler

RandomUnderSampler is employed to reduce the size of the majority class within the training set, thereby achieving a balanced dataset. This step is crucial for alleviating class imbalance.

9.3.2.3 Hyperparameter Tuning

We embark on the optimization of SVM's hyperparameter `C`, which denotes the regularization strength. A range of `C` values is explored, and cross-validation is utilized to assess the model's accuracy for each value.

In [ ]:
```
# Balancing data using undersampling with cross-validation
x_train, x_test, y_train, y_test = train_test_split(features, Y, test_size=0.2, random_state=10, stratify=Y)

under_sampler = RandomUnderSampler(random_state=10)
x_train_balanced, y_train_balanced = under_sampler.fit_resample(x_train, y_train)

# Define the range of C values to test
C_range = [0.1, 1, 10]
C_scores = []

# Perform cross-validation for each value of C
for C_value in C_range:
    svm = SVC(C=C_value, kernel='linear')
    skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=10)
    scores = cross_val_score(svm, x_train_balanced, y_train_balanced, cv=skf, scoring='accuracy')
    C_scores.append(scores.mean())

# Determine the best C value
best_C = C_range[np.argmax(C_scores)]
print(f'The best C value is {best_C} with a cross-validated accuracy of {max(C_scores):.4f}')

# Train SVM with best C value
svm = SVC(C=best_C, kernel='linear')
svm.fit(x_train_balanced, y_train_balanced)

# Evaluate on test set
svm_test_predictions = svm.predict(x_test)
svm_accuracy_test = accuracy_score(y_test, svm_test_predictions)
print('SVM Accuracy for test set: ', svm_accuracy_test)
```

9.3.3 Results

Upon employing undersampling with cross-validation for SVM and tuning the hyperparameter `C`, the following results were obtained:

- **The best C value:** 1
- **Cross-validated accuracy with the best C value:** 0.7662
- **SVM Accuracy for the test set:** 0.7658

In comparison, the SVM's performance without undersampling was as follows:

- **SVM Accuracy for the test set (without undersampling):** 0.8441

These results suggest that undersampling with cross-validation for SVM resulted in a slightly lower accuracy on the test set compared to the SVM model without undersampling. Further analysis and evaluation metrics may be necessary to comprehensively assess the efficacy of undersampling in mitigating class imbalance for SVM. Adjustments can be made based on specific details of your analysis and reporting requirements.

9.4 Evaluating SVM Performance with Balanced Datasets

9.4.1 Introduction

In this section, we assess the performance of a Support Vector Machine (SVM) classifier trained on balanced datasets achieved through both oversampling and undersampling techniques. We compare the efficacy of SMOTE (Synthetic Minority Over-sampling Technique) for oversampling and RandomUnderSampler for undersampling.

9.4.2 Methodology

9.4.2.1 Data Splitting

The dataset is divided into training and testing sets using the `train_test_split` function. Stratification ensures that the class distribution is maintained across both sets.

9.4.2.2 Balancing with Oversampling (SMOTE)

SMOTE is employed to generate synthetic samples for the minority class, resulting in a balanced training dataset. The SVM classifier is then trained on this balanced dataset.

9.4.2.3 Balancing with Undersampling (RandomUnderSampler)

RandomUnderSampler is utilized to reduce the size of the majority class in the training set, achieving a balanced dataset. Subsequently, the SVM classifier is trained on this balanced dataset.

In [ ]:
```
from imblearn.over_sampling import SMOTE
from imblearn.under_sampling import RandomUnderSampler
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split

# Splitting the data into training and testing sets
x_train, x_test, y_train, y_test = train_test_split(features, Y, test_size=0.2, random_state=10, stratify=Y)

# Balancing data using oversampling (SMOTE)
smote = SMOTE(random_state=10)
x_train_balanced, y_train_balanced = smote.fit_resample(x_train, y_train)

# Training SVM on the balanced dataset
svm = SVC(kernel='linear')
svm.fit(x_train_balanced, y_train_balanced)

# Evaluating on the test set
svm_test_predictions = svm.predict(x_test)
svm_accuracy_test = accuracy_score(y_test, svm_test_predictions)
print('SVM Accuracy for test set(oversampling): ', svm_accuracy_test)

# Balancing data using undersampling (RandomUnderSampler)
under_sampler = RandomUnderSampler(random_state=10)
x_train_balanced, y_train_balanced = under_sampler.fit_resample(x_train, y_train)

# Training SVM on the balanced dataset
svm = SVC(kernel='linear')
svm.fit(x_train_balanced, y_train_balanced)

# Evaluating on the test set
svm_test_predictions = svm.predict(x_test)
svm_accuracy_test = accuracy_score(y_test, svm_test_predictions)
print('SVM Accuracy for test set(undersampling): ', svm_accuracy_test)
```

9.4.3 Results

9.4.3 Results

The performance of the Support Vector Machine (SVM) classifier on the test set after training on balanced datasets using oversampling (SMOTE) and undersampling (RandomUnderSampler) techniques is as follows:

- **SVM Accuracy for Oversampled Test Set:** 0.8381
- **SVM Accuracy for Undersampled Test Set:** 0.7658

Comparing these results with the performance before sampling:

- **SVM Accuracy for Test Set (Before Sampling):** 0.8441

It appears that the SVM classifier achieved slightly lower accuracy on the test set after both oversampling and undersampling compared to before sampling. However, further analysis and evaluation metrics may be required to make a comprehensive assessment of the efficacy of each balancing technique. Adjustments can be made based on specific details of your analysis and reporting requirements.

10 Comparison of Model Performance

To compare the performance of different classifiers, the accuracy, precision, recall, and F1 score for each model (KNN, Decision Tree, and SVM) are compiled into a DataFrame. This structured comparison provides a clear overview of each model's effectiveness in predicting the target variable.

In [ ]:
```
# Create a DataFrame to store the metrics
results_df = pd.DataFrame({
    'Model': ['KNN', 'Decision Tree', 'SVM'],
    'Accuracy': [knn_accuracy, tree_accuracy, svm_accuracy],
    'Precision': [knn_precision, tree_precision, svm_precision],
    'Recall': [knn_recall, tree_recall, svm_recall],
    'F1 Score': [knn_f1_score, tree_f1_score, svm_f1_score]
})

# Display the DataFrame
results_df
```

10.1 Updated Comparison of Model Performance

In [ ]:
```
# Create a DataFrame to store the metrics
results_df = pd.DataFrame({
    'Model': ['KNN', 'Decision Tree', 'SVM'],
    'Training Accuracy': [knn_accuracy_train, tree_accuracy_train, svm_accuracy_train],
    'Validation Accuracy': [knn_accuracy_val, tree_accuracy_val, svm_accuracy_val],
    'Test Accuracy': [knn_accuracy_test, tree_accuracy_test, svm_accuracy_test],
    'Precision': [knn_precision, tree_precision, svm_precision],
    'Recall': [knn_recall, tree_recall, svm_recall],
    'F1 Score': [knn_f1_score, tree_f1_score, svm_f1_score]
})

# Display the DataFrame
results_df
```

In [ ]:
```

```