Team name: `SGS_Tut01_NhatQuang_Team04`

Members:

- Vo Huy Chu (s4075654)
- Tran Gia Khanh (s4041377)
- Vo Dang Khoa (4058809)
- Ngo Huynh Uyen Nhu (s4107237)
- Phan Ngoc Bao Tran (s4028662)
- Cao Vu Thuy Uyen (s4028519)

Foreword

The Python code in this notebook is to be run on Python3.12 and above, which can be found by building a conda environment from the `environment.yml` file. Run the code cell below to install it, then set the kernel used to run the code to the newly-created conda environment.

In [ ]:
```
# Import the `platform` module:
import platform
# Check if the OS running the notebook isn't running Windows:
if platform.system() != "Windows":
    # Install HomeBrew:
    !sudo /bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
    # Install miniconda:
    !sudo brew install --cask miniconda
    # Initialise miniconda with the current shell:
    !sudo conda init "$(basename "${SHELL}")"
    # Create the conda environment from the `environment.yml` file:
    !conda env create --file environment.yml
# Run the following code if the host machine is running Windows:
else:
    # Install miniconda:
    !winget install Anaconda.Miniconda3 --accept-package-agreements --nowarn
    # Create the conda environment from the `environment.yml` file:
    !%LOCALAPPDATA%\miniconda3\Library\bin\conda.bat env create --file environment.yml
```

```
-
\
|
-
\
|
/

Found an existing package already installed. Trying to upgrade the installed package...
No available upgrade found.
No newer package versions are available from the configured sources.
```

Dataset Acquisition

The dataset used in this analysis was obtained from the Kaggle website, which hosts data science projects and public datasets, at the web address: https://www.kaggle.com/datasets/budincsevity/szeged-weather. This dataset is weather data from Szeged, Hungary. It covers a period from 2006 to 2016. The data, which includes different conditions of the atmosphere has been captured for every hour. In this case, we want to come up with a model that predicts the apparent temperature as it relates to human perception under certain weather conditions. A term proposed for prediction is a mean absolute error (MAE) of less than $3.0^{\circ}C$. This value has been estimated prudentially due to the fact that the final beneficiaries of such forecasting results are human beings who sense the temperature through their skin and typically do not feel changes for less than $3.0^{\circ}C$. The MAE threshold balances model complexity with the practical needs of its users in making predictions that are neither too accurate — which would be overly precise — nor too loosely fit.

In [ ]:
```
# Import `pandas`:
import pandas as pd
# Read the dataset:
dataset = pd.read_csv("./weatherHistory.csv")
```

Peform EDA (exploratory data analysis)

In [ ]:
```
# List of non-null values and data type for each feature:
print(dataset.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 96453 entries, 0 to 96452
Data columns (total 12 columns):
 #   Column                    Non-Null Count  Dtype
---  ------                    --------------  -----
 0   Formatted Date            96453 non-null  object
 1   Summary                   96453 non-null  object
 2   Precip Type               95936 non-null  object
 3   Temperature (C)           96453 non-null  float64
 4   Apparent Temperature (C)  96453 non-null  float64
 5   Humidity                  96453 non-null  float64
 6   Wind Speed (km/h)         96453 non-null  float64
 7   Wind Bearing (degrees)    96453 non-null  float64
 8   Visibility (km)           96453 non-null  float64
 9   Loud Cover                96453 non-null  float64
 10  Pressure (millibars)      96453 non-null  float64
 11  Daily Summary             96453 non-null  object
dtypes: float64(8), object(4)
memory usage: 8.8+ MB
None
```

From the results above, we can see that the dataset comprises a total of 96453 samples. While most of the features are complete, note that the `Precip Type` feature has 517 missing values. Notwithstanding, the rest of the features are fully populated; each has 96453 entries. This very low presence of missing data confirms that this is good quality data with only a very small portion of missing values and no major issues related to unbalanced classes. Out of the 12 features used in the dataset, 8 are numeric. These numeric features relate to important weather measurements: two temperature readings, humidity, wind speed, wind bearing, visibility, cloud cover, and atmospheric pressure. Quantitative analysis, which helps in modeling and interpreting weather patterns more effectively, is dependent on the presence of these numeric features.

Details of each feature:

- `Formatted Date` : Date and time the measurements were recorded, in the format of `yyyy-mm-dd hh:mm::ss.ms +0200` .
- `Summary` : A short summary of the weather throughout the hour of recording, up to 2 significant features separated by " `and` ".
- `Precip Type` : The prominent type of precipitation throughout the hour of recording, no precipitation is marked with an empty value.
- `Temperature (C)` : The temperature at the hour of recording, in Celcius.
- `Apparent Temperature (C)` : How the temperature feels to humans based on various factors like humidity, wind speed, etc., measured in degrees Celsius.
- `Humidity` : The humidity measured at the hour of recording, value ranges from 0 to 1.
- `Wind Speed (km/h)` : The velocity of the wind at the hour of recording, measured in `km/h` .
- `Wind Bearing (degrees)` : The direction from which the wind is coming, expressed in degrees from true north.
- `Visibility (km)` : How far away the terrain and objects are visible, measured in `km` .
- `Loud Cover` : How much of the sky is covered by clouds
- `Pressure (millibars)` : The atmospheric pressure at the hour of recording, in `millibars` .
- `Daily Summary` : A long line of string summarising the weather condition throughout the day.

In [ ]:
```
# A summary of the statistics for the numerical features:
dataset.describe()
```

Out[ ]:

	Temperature (C)	Apparent Temperature (C)	Humidity	Wind Speed (km/h)	Wind Bearing (degrees)	Visibility (km)	Loud Cover	Pressure (millibars)
count	96453.000000	96453.000000	96453.000000	96453.000000	96453.000000	96453.000000	96453.0	96453.000000
mean	11.932678	10.855029	0.734899	10.810640	187.509232	10.347325	0.0	1003.235956
std	9.551546	10.696847	0.195473	6.913571	107.383428	4.192123	0.0	116.969906
min	-21.822222	-27.716667	0.000000	0.000000	0.000000	0.000000	0.0	0.000000
25%	4.688889	2.311111	0.600000	5.828200	116.000000	8.339800	0.0	1011.900000
50%	12.000000	12.000000	0.780000	9.965900	180.000000	10.046400	0.0	1016.450000
75%	18.838889	18.838889	0.890000	14.135800	290.000000	14.812000	0.0	1021.090000
max	39.905556	39.344444	1.000000	63.852600	359.000000	16.100000	0.0	1046.380000

The `Loud Cover` feature, as shown by summary statistics, has a value of 0.0 for all 96453 samples. Such uniformity indicates that there is no variation in this feature across the entire dataset. Because `Loud Cover` does not bring any discriminating information or variability to the data that will help in prediction but only duplicate information, it is redundant. Therefore, it can safely be dispensed from the dataset to enhance tidiness during subsequent analysis and lower computational costs without loss of essential information.

In [ ]:
```
# A few samples from the beginning of the dataset:
dataset.head()
```

Out[ ]:

	Formatted Date	Summary	Precip Type	Temperature (C)	Apparent Temperature (C)	Humidity	Wind Speed (km/h)	Wind Bearing (degrees)	Visibility (km)	Loud Cover	Pressure (millibars)	Daily Summary
0	2006-04-01 00:00:00.000 +0200	Partly Cloudy	rain	9.472222	7.388889	0.89	14.1197	251.0	15.8263	0.0	1015.13	Partly cloudy throughout the day.
1	2006-04-01 01:00:00.000 +0200	Partly Cloudy	rain	9.355556	7.227778	0.86	14.2646	259.0	15.8263	0.0	1015.63	Partly cloudy throughout the day.
2	2006-04-01 02:00:00.000 +0200	Mostly Cloudy	rain	9.377778	9.377778	0.89	3.9284	204.0	14.9569	0.0	1015.94	Partly cloudy throughout the day.
3	2006-04-01 03:00:00.000 +0200	Partly Cloudy	rain	8.288889	5.944444	0.83	14.1036	269.0	15.8263	0.0	1016.41	Partly cloudy throughout the day.
4	2006-04-01 04:00:00.000 +0200	Mostly Cloudy	rain	8.755556	6.977778	0.83	11.0446	259.0	15.8263	0.0	1016.51	Partly cloudy throughout the day.

Inspecting a few initial samples of the dataset via `dataset.head()` , we note that the four features namely `Formatted Date` , `Summary` , `Precip Type` , and `Daily Summary` are essentially of non-numeric data type, i.e., `string` , in this case. In particular, these features do not directly supply numeric values that can be analysed or modeled. For us to determine whether these features can be treated as categorical variables and thereby establish their potential for incorporation into our model, we have to check the number of unique values in each feature. Those features containing a low number of unique values are good candidates for conversion into categorical variables. "Label Encoding", where each unique value is assigned its numerical code, or "One-Hot Encoding" which transforms each unique value into a separate binary $(0/1)$ feature are some techniques used for such conversions. The uniqueness of the values within these features is very important in deciding how to best inject them into the model to make it more effective by retaining crucial categorical information.

In [ ]:
```
print(f"{dataset["Formatted Date"].value_counts()}\n")
print(f"{dataset["Summary"].value_counts()}\n")
print(f"{dataset["Precip Type"].value_counts()}\n")
print(f"{dataset["Daily Summary"].value_counts()}\n")
```

```
Formatted Date
2010-08-02 23:00:00.000 +0200    2
2010-08-02 22:00:00.000 +0200    2
2010-08-02 21:00:00.000 +0200    2
2010-08-02 20:00:00.000 +0200    2
2010-08-02 19:00:00.000 +0200    2
                                ..
2016-09-09 03:00:00.000 +0200    1
2016-09-09 04:00:00.000 +0200    1
2016-09-09 05:00:00.000 +0200    1
2016-09-09 06:00:00.000 +0200    1
2006-04-01 05:00:00.000 +0200    1
Name: count, Length: 96429, dtype: int64

Summary
Partly Cloudy                           31733
Mostly Cloudy                           28094
Overcast                                16597
Clear                                   10890
Foggy                                    7148
Breezy and Overcast                       528
Breezy and Mostly Cloudy                  516
Breezy and Partly Cloudy                  386
Dry and Partly Cloudy                      86
Windy and Partly Cloudy                    67
Light Rain                                 63
Breezy                                     54
Windy and Overcast                         45
Humid and Mostly Cloudy                    40
Drizzle                                    39
Breezy and Foggy                           35
Windy and Mostly Cloudy                    35
Dry                                        34
Humid and Partly Cloudy                    17
Dry and Mostly Cloudy                      14
Rain                                       10
Windy                                       8
Humid and Overcast                          7
Windy and Foggy                             4
Dangerously Windy and Partly Cloudy         1
Windy and Dry                               1
Breezy and Dry                              1
Name: count, dtype: int64

Precip Type
rain    85224
snow    10712
Name: count, dtype: int64

Daily Summary
Mostly cloudy throughout the day.                                      20085
Partly cloudy throughout the day.                                       9981
Partly cloudy until night.                                              6169
Partly cloudy starting in the morning.                                  5184
Foggy in the morning.                                                   4201
                                                                       ...
Drizzle until morning.                                                    24
Light rain overnight.                                                     24
Rain until morning.                                                       24
Rain until afternoon.                                                     24
Foggy starting overnight continuing until morning and breezy in the afternoon.    23
Name: count, Length: 214, dtype: int64
```

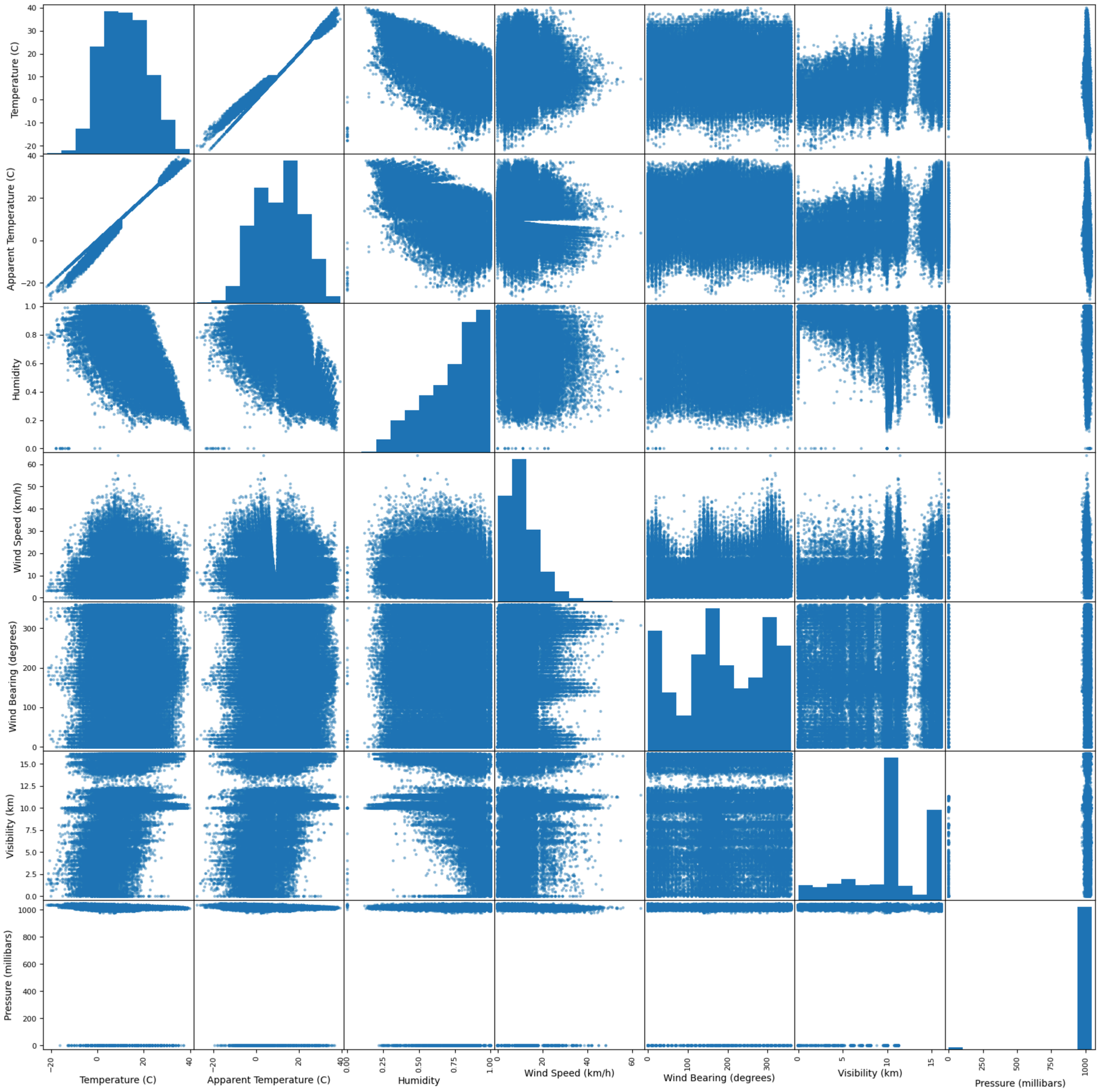
From our dataset, we find `Precip Type` is appropriate for categorical conversion as it holds just three unique values: `rain`, `snow` and `none` (we introduced a custom value here to use in filling in missing entries). `Precip Type` is great for "Label" or "One-Hot Encoding" because of the small number of unique values it has. This will allow us to appropriately represent this feature in numeric form for our model. The `Summary` feature also makes itself available to be converted into a categorical variable. Because this feature is made up of textual descriptions, it can be encoded to reflect various weather conditions. Let every distinct weather summary be represented as a one-dimensional vector. If a given summary describes two properties — say "Partly Cloudy" and "Windy" — then it can be taken to be a vector that combines the other component vectors into one, based on the presence of these separate dimensions, i.e., attributes on their own and not just on the individual values each component represents. In addition, vectors representing different property intensities ("Light Rain" and "Heavy Rain") can be encoded by using values that point in similar directions but have different magnitudes. A mild condition is represented by $[1, 0]$; an equally mild equivalent is $[2, 0]$ — only that it gives a stronger impact. By transforming `Precip Type` and `Summary` in this manner, we retain the most important information regarding weather conditions but make these features usable with machine learning algorithms that require numerical input.

Looking for outliers

This step is very important as outliers can negatively affect the accuracy of the model, making it inaccurate and inconsistent.

We look for outliers by viewing and singling out the dots that are out of place and do not quite follow the trend on the scatter plots with every pair of features:

In [ ]:
```
# Import the `scatter_matrix` class from the `plotting` module of `pandas`:
from pandas.plotting import scatter_matrix
# Import the `pyplot` module from `matplotlib`
import matplotlib.pyplot as pplt
# List of features to plot:
features_to_plot = ["Temperature (C)", "Apparent Temperature (C)", "Humidity", "Wind Speed (km/h)", "Wind Bearing (degrees)", "Visibility (km)", "Pressure (millibars)"]
# Perform the `scatter_matrix` action on the dataset with the features in the list of `features_to_plot` above:
scatter_matrix(dataset[features_to_plot], figsize=(20, 20))
# Show the scatter matrix and the histograms on the topleft-bottomright diagonal:
pplt.show()
```



- The various points far from the blue clusters seen in the graphs above are outliers.
- Additionally, in the histograms on display, the bars separated from the main clusters are outliers.

Removing outlier rows

Outliers can be removed by adding a limitting line (Often horizontal or vertical) outside the cluster to remove small dots that are out of place. Through direct observation and analysis, we shall form conditions for each feature which will limit the values in the dataset to specific ranges, thus excluding outliers:

- `Temperature (C)` : Only allow for values above -15 degrees Celcius as most of the dots are above that temperature
- `Apparent Temperature (C)` : Only allow for values above -20 degrees Celcius because there are only a few dots at the bottom
- `Humidity` : Only take in data with the humidity value above 0.2
- `Wind Speed (km/h)` : The wind speeed cannot reach 45 `km/h` since the majority of the spots do not go any higher than 45 `km/h`
- `Visibility (km)` : The visibility cannot be between 12.5 `km` and 13.75 `km` because the amount of dot between that 2 values do not seem to be enough to consider valid
- `Pressure (millibar)` : The pressure must be over 1 `bar` since it's evident that there's a huge gap between the 2 milestones

In [ ]:
```python
# Temperature condition being higher than -15 degrees C:
temperature_condition = dataset["Temperature (C)"] > -15
# Apparent temperature being higher than -20 degrees:
apparent_temperature_condition = dataset["Apparent Temperature (C)"] > -20
# Humidity higher than 0.2:
hudmidity_condition = dataset["Humidity"] > 0.2
# A wind speed lower than 45 km/h:
wind_speed_condition = dataset["Wind Speed (km/h)"] < 45
# A visibility not between 12.5 km and 13.75 km:
visibility_condition = (dataset["Visibility (km)"] <= 12.5) | (dataset["Visibility (km)"] >= 13.75)
# A pressure above 1000 millibars:
pressure_condition = dataset["Pressure (millibars)"] > 1000
# Apply all the above conditions:
filtered_dataset = dataset[temperature_condition & apparent_temperature_condition & hudmidity_condition & wind_speed_condition & visibility_condition & pressure_condition]
```

Preprocess data

Removing unecessary features

In our data, there are some features that do not help in predicting the apparent temperature, which is the main goal of our task. These features include `Formatted Date` and `Daily Summary`. Since these features do not provide useful information for our analysis, we will remove them from the dataset. Additionally, we have already removed the feature `Loud Cover` earlier because it only contained a single value (0.0), making it unhelpful for our analysis. By removing these unnecessary features, we can focus on the data that will actually contribute to achieving accurate results.

In [ ]:
```python
# Remove the columns using drop():
filtered_dataset = filtered_dataset.drop(columns=["Formatted Date", "Daily Summary", "Loud Cover"])
```

Splitting training and testing data

There are two methods to accomplish this, being "randomised sample selection" and "stategic sample selection". The prior will randomly select 20% of data for the test set, useful in scenarios where the dataset is large. The second is used when we want to maintain the distributions of important features. As our dataset is large (96453 samples in total), "randomised sample selection" is appropriate for our task.

In [ ]:
```python
# Import the `train_test_split` class from the `model_selection` module from Scikit-Learn:
from sklearn.model_selection import train_test_split
# Split the dataset:
training_set, test_set = train_test_split(filtered_dataset, test_size=0.2, random_state=69)
"""
`test_size=0.2`: 20% of the dataset will be distributed to the test set
`random_state`: Used to get the same training set when the same number is used
"""
```

Out[ ]:
```
'\n`test_size=0.2`: 20% of the dataset will be distributed to the test set\n`random_state`: Used to get the same training set when the same number is used\n'
```

Removing the labels

A supervised machine model's dataset would typically compose of features and labels. Features are the input variables while the labels are target variables. Generally, we train a ML model to predict the labels based on the features. In order to do that, our dataset is split into two main components: the feature matrix and the label vector.

In our weather prediction program, the parameter `Apparent Temperature (C)` would be the target variable or label. This parameter is what our program will learn to predict based on the features provided in the dataset.

In the code below, the training set is used to build the model while the testing test is used to evaluate performance. We remove the labels because the values of the labels are not supposed to take into consideration during the calculation, and the calculations must be made based on the input features alone.

In [ ]:
```python
# Store the label feature from the training set for future use:
training_set_label = training_set["Apparent Temperature (C)"].copy()
# Drop from the training set:
training_set = training_set.drop(columns="Apparent Temperature (C)")
# Store the label feature from the testing set for future use:
test_set_label = test_set["Apparent Temperature (C)"].copy()
# Drop from the test set:
test_set = test_set.drop(columns="Apparent Temperature (C)")
```

We split the features into two categories: Numerical and Categorical. The Categorical features include `Summary` and `Precip Type`, both of which will be handled separately from the other. These categorical features represent non-numeric data that require processing before they can be used to train ML models. In order to do so, we transform the categorical features into a numerical format through a process called vectorisation.

In [ ]:
```
# Import the `NumPy` library for use:
import numpy as np
# The numerical features are automatically selected from the training set, based on their data type:
numerical_features = list(training_set.select_dtypes(include=[np.number]))
```

A pipeline to preprocess categorical data

`Summary`

To process data of the `Summary` feature, we shall vectorise the sample values as the following:

- `Overcast` : [3, 0, 0, 0, 0]
- `Mostly Cloudy` : [2, 0, 0, 0, 0]
- `Partly Cloudy` : [1, 0, 0, 0, 0]
- `Clear` : [0, 0, 0, 0, 0] (No significant weather conditions)
- `Foggy` : [0, 2, 0, 0, 0]
- `Dangerously Windy` : [0, 0, 3, 0, 0]
- `Windy` : [0, 0, 2, 0, 0]
- `Breezy` : [0, 0, 1, 0, 0]
- `Humid` : [0, 0, 0, 2, 0]
- `Dry` : [0, 0, 0, -2, 0]
- `Rain` : [0, 0, 0, 0, 3]
- `Light Rain` : [0, 0, 0, 0, 2]
- `Drizzle` : [0, 0, 0, 0, 1]

  The pipeline also allows combinations of the weather effects as the sum of the above vectors. For example: a "Windy and Rainy" day could be represented as the sum of `Windy`: [0, 0, 2, 0, 0] and `Rain`: [0, 0, 0, 0, 3] vectors, resulting in [0, 0, 2, 0, 3]. This ablity of summing up vectors for combined weather conditions offer a flexible way to handle real life scenarios where multiple weather contions occur simultaneously.

The `SummaryVectoriser` class:

Inherits from `BaseEstimator` and `TransformerMixin`. `SummaryVectoriser` will transform the original text data into the vectors as defined above.

Class Inheritance

We will use class inheritance from `BaseEstimator` and `TransformerMixin` (part of the scikit-learn library) in order to seamlessly integrate in the scikit-learn pipeline. `BaseEstimator` provides the functionality for managing parameters. By inheriting from `BaseEstimator`, the 'SummaryVectoriser' class gains compatibility with scikit-learn pipeline and the upcoming grid search function. `TransformerMixin` allows us to call `fit()` and `transform()` in a single step, allowing data to be processed in one streamlined step.

Class Constructor

The `__init__()` method initialises a dictionary `self.vectors` that maps the weather conditions to NumPy arrays. Each array is a numerical vector represents a specific weather condition.

The `fit()` Method

The `fit()` method is a placeholder in this class which returns `self`. This is necessary for the compatibility within the pipelines. Here, the `fit()` method does not need to store any parameters because the transformation logic is independent to the dataset as the same weather condition will always have the same vector.

The `transform()` Method

This is the core of the `SummaryVectoriser` class as the `transform` method will convert an array of weather summaries into a matrix of corresponding numerical vectors. The method begins by creating a NumPy array `transformed_data` filled with 0s. The array will be formatted as (number of data points, length of a vector) where a row represents a weather summary and a column represents a component of the vector. This method will loop and iterate over each summary in the input data then check if each weather condition exists in `self.vectors`. If it does, the selected vector is added to the cumulative vector and stored in the appropriate row of `transformed_data`. This process is looped for all weather summaries in the dataset. After the process of all summaries, the `transform()` method returns the `transformed_data` array. This array can now be used as input features for our ML model.

In [ ]:
```
# Import the `BaseEstimator`and `TransformerMinxin` classes for use:
from sklearn.base import BaseEstimator, TransformerMixin
# Define the class `SummaryVectoriser`:
class SummaryVectoriser(BaseEstimator, TransformerMixin):
    # Constructor for the class:
    def __init__(self):
        # `vectors` will be the list of vectors sent to the constructor:
        self.vectors = {
            "Overcast": np.array([3, 0, 0, 0, 0]),
            "Mostly Cloudy": np.array([2, 0, 0, 0, 0]),
            "Partly Cloudy": np.array([1, 0, 0, 0, 0]),
            "Clear": np.array([0, 0, 0, 0, 0]),
            "Foggy": np.array([0, 2, 0, 0, 0]),
            "Dangerously Windy": np.array([0, 0, 3, 0, 0]),
            "Windy": np.array([0, 0, 2, 0, 0]),
            "Breezy": np.array([0, 0, 1, 0, 0]),
            "Humid": np.array([0, 0, 0, 2, 0]),
            "Dry": np.array([0, 0, 0, -2, 0]),
            "Rain": np.array([0, 0, 0, 0, 3]),
            "Light Rain": np.array([0, 0, 0, 0, 2]),
            "Drizzle": np.array([0, 0, 0, 0, 1])
        }
    # The `fit()` method:
    def fit(self, data, labels=None):
        # Make no changes, simply return `self`:
        return self
    # The `transform()` method:
    def transform(self, data):
        # Create an NumPy array filled with `0`s to hold the vectors for each weather condition found in `data`:
        transformed_data = np.zeros((len(data), len(next(iter(self.vectors.values())))))
        # Loop through each element in `data`:
        for index, array in enumerate(data):
            condition_string = array[0]
            # Split the string for the weather condition:
            conditions = [condition.strip() for condition in condition_string.split(" and ")]
            # Initialise a vector to hold the values corresponding to each weather condition in the `conditions` list:
            vector = np.zeros(len(next(iter(self.vectors.values()))))
            # Loop over each condition in `conditions`:
            for condition in conditions:
                # Check if the condition is in `self.vectors`
                if condition in self.vectors:
                    # Add the value of the corresponding vector into the original `vector`:
                    vector += self.vectors[condition]
            # Put the added-up vector at the index of `index` in the `transformed_data` array:
            transformed_data[index] = vector
        # Return the transfomed data:
        return transformed_data
```

`Precip Type`

We first handle the missing values in this column, which occur when there are no precipitations to be found. As such, they will be replaced with "`none`". This is done using the `SimpleImputer()` transformer which will fill the empty values with a constant string, being "`none`".

From our selected dataset, the `Precip Type` columns have occasional missing values. These values embody the scenarios in which precipitation wasn't recognized, which means that there was neither rain nor snow. To confront these issues, `SimpleImputer()` from the Scikit-Learn module will be utilised, which can make sure that the dataset is fully complete so that there are no missing data for the model to deal with.

`Precip Type` is considered to be a categorical variable with values such as `rain`, `snow`, and `none`. However, in machine learning models, it must be in the form of a number, which means that these categories must be transformed into a numerical form. In this case, `One-Hot Encoding` is used since categories don't have a natural order (e.g., `rain` doesn't mean more or less than `snow`; they are simply different.), so Label Encoding cannot be used. In addition, if we used Label Encoding, the model would, by chance, incorrectly assume a relationship between the numbers allowing incorrect predictions to occur. In said process, One-Hot Encoding would convert the `Precip Type` column into an ample number of binary columns (0s and 1s).

In [ ]:
```
# Import the `Pipeline` class from Scikit-Learn for use:
from sklearn.pipeline import Pipeline
# Import the simple imputer from Scikit-Learn:
from sklearn.impute import SimpleImputer
# Import the `OneHotEncoder` feature for use:
from sklearn.preprocessing import OneHotEncoder
# Create the pipeline to process `Precip Type`:
precip_type_pipeline = Pipeline(
    # Define the steps the pipeline will take to transform the data:
    steps=[
        # Perform `SimpleImputer` on the dataset:
        ("simple_imputer", SimpleImputer(strategy="constant", fill_value="none")),
        # Perform `OneHotEncoder` on the dataset:
        ("one_hot_encoder", OneHotEncoder())
    ]
)
```

One `ColumnTransformer` to transform the data

- `StandardScaler` will be applied to the numerical features
- `SummaryVectoriser` will be applied to the feature `Summary`
- `OneHotEncoder` will be applied to the feature `Precip Type`

The `ColumnTransformer()` class will enables choosing which transformations are going to be implemented for which columns in the dataset, which makes it easier to apply different preprocessing pipelines to variating kinds of data.

Reminder="passthrough": This will make any columns that are not explicitly listed in the transformers list passed through without any modifications. As a result, all the data is retained even if it's not transformed.

In [ ]:
```
# Import the `ColumnTransformer` class for use:
from sklearn.compose import ColumnTransformer
# Import the `StandardScaler` and `OneHotEncoder` classes for use:
from sklearn.preprocessing import StandardScaler
# Utilise `ColumnTransformer`:
column_transformer = ColumnTransformer(
    # List the transformers to be used:
    transformers=[
        # The numerical features will be processed using `StandardScaler()`:
        ("numerical_transformer", StandardScaler(), numerical_features),
        # The `Summary` feature will be processed using the custom-made `SummaryVectoriser()`:
        ("summary_transformer", SummaryVectoriser(), ["Summary"]),
        # The `Precip Type` feature will be processed using our custom-made pipeline above:
        ("precip_type_transformer", precip_type_pipeline, ["Precip Type"])
```

```
    ],
    # Passthrough the rest of the data (if there are any left) without any modifications:
    remainder="passthrough"
)
```

Transform the data using the defined transformer

In [ ]:
```
# Transform the `training_set` using `fit()` and `transform()` together as `fit_transform()`:
transformed_training_set = column_transformer.fit_transform(training_set)
```

The output is a converted dataset (`translated_training_set`), where each feature has been handled according to the specific transformations.

Inspect the processed data

We view the first few samples of the processed data.

For verifying the preprocessing steps, the first few rows will be printed of the converted dataset. Moreover, the number of rows and columns in the dataset is also printed to display an overview of how the data has been transformed.

In [ ]:
```
# Print out some of the processed data for viewing:
print(transformed_training_set[[0, 1, 2, 3, 4], :])
# Print out the number of rows and columns in the output data:
print(f"(Rows, Columns): {transformed_training_set.shape}")
```

```
[[ 1.07781127 -0.48117906 -0.89707244  0.69431661  1.23015141 -1.05032469
   0.          0.          0.          0.          0.          0.
   1.          0.        ]
 [-0.50312877  0.28857088 -1.09737931  0.30374158  1.13022471 -0.51590358
   0.          0.          0.          0.          0.          0.
   1.          0.        ]
 [-1.83981591  1.31490413 -1.14984063  0.54552612 -2.40179979  1.639733
   0.          0.          0.          0.          0.          0.
   0.          1.        ]
 [-0.68558977  0.8017375   0.44784508 -1.64913354  1.27242809  1.28621413
   0.          0.          0.          0.          0.          0.
   1.          0.        ]
 [ 0.02151947 -1.55882897  0.09492346 -1.64913354 -0.09964236  1.05836016
   0.          0.          0.          0.          0.          0.
   1.          0.        ]]
(Rows, Columns): (74329, 14)
```

Model Training and Evaluation

We shall proceed to train a total of 6 models, those being:

- `Linear Regression` : Fits a straight line to predict the target variable based on a linear relationship with input features.
- `Decision Tree Regressor` : Splits the data into subsets using decision rules to make predictions based on the majority outcome in each subset.
- `Gradient Boosting Regressor` : Builds a series of small models (trees) where each new model corrects the errors of the previous ones.
- `Huber Regressor` : Combines the best of both squared errors (for small errors) and absolute errors (for large errors) to handle outliers better.
- `AdaBoost Regressor` : Starts with a simple model and adjusts it to focus on the mistakes made by the previous models.
- `RANSAC Regressor` : Randomly selects subsets of the data to fit a model, then finds the model that fits the most data points well, ignoring outliers.

Metric: `mean_absolute_error` (MAE): A metric value to compute the average amount of errors between the actual labels and the predictions. This will assist in deciphering due to the fact that it's in the same units as the target variable.

Cross-Validation: `cross_val_score` : a value to examine a model's performance by dividing the data into k folds and using each fold as a validation set during the training of the remaining $k - 1$ folds.

The training, predicting, and evaluating of the model is conducted in a loop: training->prediction->MAE calculation->Cross-Validation->output->training->... This will support the comparison of multiple regression models by utilising both their in-sample MAE and their cross-validated MAE. The cross-validated MAE is more reliable as it gives an assumption of how models perform on unseen data.

The `RandomizedSearchCV()` will be implemented as there is no need to check every possible combination but somewhat a random sample of combinations, which is quicker than an exhaustive search (`GridSearchCV()`). Furthermore, both continuous and discrete hyoerparameters can be searched with the former method, which allows for more diverse and possibly better-performing models.

In [ ]:
```
# Import `LinearRegression`, `HuberRegressor` and `RANSACRegressor` from the `linear_model` module:
from sklearn.linear_model import LinearRegression, HuberRegressor, RANSACRegressor
# Import the `DecisionTreeRegressor` class, equivalent to the "Decision Tree Regressor" model:
from sklearn.tree import DecisionTreeRegressor
# Import the models "Gradient Boosting Regressor" and "Ada Boost Regressor" for use:
from sklearn.ensemble import GradientBoostingRegressor, AdaBoostRegressor
# Import the MAE error measurement tool:
from sklearn.metrics import mean_absolute_error
# Import the the cross-evaluation error measurement tool:
from sklearn.model_selection import cross_val_score
"""
MAE was selected over the other metrics as it provides the average error magnitude in the same units as the target variable.
This also means we are able to comprehend the errors of the training by looking at the range of possible values of the target.
"""
# Create a dictionary of models:
models = {
    # Associate the string "Linear Regression" with the model for Linear Regression:
    "Linear Regression": LinearRegression(),
    # Associate the string "Decision Tree Regressor" with the model for Decision Tree Regressor:
    "Decision Tree Regressor": DecisionTreeRegressor(),
    # Associate the string "Gradient Boosting Regressor" with the model for Gradient Boosting Regressor:
    "Gradient Boosting Regressor": GradientBoostingRegressor(),
    # Associate the string "Huber Regressor" with the model for Huber Regressor:
    "Huber Regressor": HuberRegressor(),
    # Associate the string "Ada Boost Regressor" with the model for Ada Boost Regressor:
    "Ada Boost Regressor": AdaBoostRegressor(),
    # Associate the string "RANSAC Regressor" with the model for RANSAC Regressor:
    "RANSAC Regressor": RANSACRegressor()
}
# Loop over the names and the models in the dictionary above:
for name, model in models.items():
    # Perform the training:
    model.fit(transformed_training_set, training_set_label)
    # Make a prediction:
    prediction = model.predict(transformed_training_set)
    # Caculate the mean absolute error:
    mae = mean_absolute_error(training_set_label, prediction)
    # Print the mean absolute error out to the output:
    print(f"{name}'s MAE (mean absolute error): {mae}")
    # Perform K-fold cross-evaluation with K = 5 using the MAE scoring system:
    scores = cross_val_score(model, transformed_training_set, training_set_label, cv=5, scoring="neg_mean_absolute_error")
    """
    `cv=5`: The model is trained on 5 - 1 folds and tested on the final fold.
    5 often strikes a good balance between having a sufficient amount of data for training and testing while minimizing computational cost.
    A fold is a division of the data.
    The cross-validated mean absolute error will be the mean of the scores, each one for a fold.
    `cross_val_score` expects a scoring metric where a higher score indicates a better model performance.
    Since MAE is a metric where lower values indicate better performance (i.e., less error), scoring="neg_mean_absolute_error" is used.
    This way, cross_val_score can use a higher score to indicate a better performance, even though the underlying metric (MAE) actually decreases as performance improves.
    """
    # Print out the cross-validated MAE:
    print(f"{name}'s cross-validated MAE score: {-scores.mean()}.\n")
```

```
Linear Regression's MAE (mean absolute error): 0.8304587647070995
Linear Regression's cross-validated MAE score: 0.8305962847612332.

Decision Tree Regressor's MAE (mean absolute error): 2.641380309987015e-15
Decision Tree Regressor's cross-validated MAE score: 0.028533143860818753.

Gradient Boosting Regressor's MAE (mean absolute error): 0.13850922523802153
Gradient Boosting Regressor's cross-validated MAE score: 0.1322134015236851.

Huber Regressor's MAE (mean absolute error): 0.8171185432843416
Huber Regressor's cross-validated MAE score: 0.8172470599230369.

Ada Boost Regressor's MAE (mean absolute error): 0.9201103132160738
Ada Boost Regressor's cross-validated MAE score: 0.8919317530424904.

RANSAC Regressor's MAE (mean absolute error): 0.8304587647070995
RANSAC Regressor's cross-validated MAE score: 0.8305962847612332.
```

Our target is the `Apparent Temperature (C)` label, whose values lie within the range of $[-27.716667, 39.344444]$. From the results above, we can clearly see that all of the models' MAE, as well as their cross-evaluated MAE scores are insignificant in comparision with this range. Additionally, "Decision Tree Regressor" is the most suitable to perform machine learning on this data, with a mean miniscule compared to its fellow models and a cross-validated MAE score trumping all others of similar variants. In addition, Decision Tree Regressor is one of the faster models to run, in comparison with "Gradient Boosting Regressor" and "Ada Boost Regressor".

Hyperparameter Tuning

We utilise a random search to get random combinations from a defined hyperparameter space.

- Hyperparameter space: Contains criterias used for filtering parameters.
  - `"criterion": ["squared_error"]` : Measure the mean squared error for regression. Using the mean squared error is typically preferred for regression tasks as it provides more accurate models when outliers have a minimal impact because it evaluates the variance of target values and penalizes larger errors more heavily, which helps in capturing the general trends in the data.
  - `"splitter": ["best"]` : The strategy used to split nodes in the decision tree, in this case, it evaluates all possible splits to find the one with the highest quality based on the criterion. This approach generally leads to a more accurate model as it focuses on finding the optimal splits.
  - `"max_depth": list(range(3, 46))` : Possible values for the maximum depth of the tree. In this case, the limits are conservative to avoid overfitting (a case where outliers are taken into account while they shouldn't).
- RandomizedSearchCV object: Samples a fixed number of hyperparameter combinations randomly from the hyperparameter space above.
  - `DecisionTreeRegressor()` : The model to be tuned
  - `param_distributions` : The parameters and their possible values used for filtering
  - `n_iter` : Number of random hyperparameter combinations to try
  - `cv` : Number of folds in cross-evaluation
  - `random_state` : The same value will lead to the same results

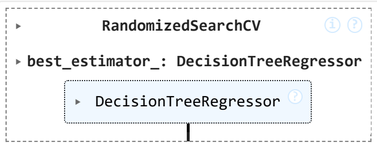
(A decision tree starts at the root node being the entire dataset, it then branches out based on the features' values that affect the target value the most using techniques such as mean-squared error until a certain criteria is met.)

In [ ]:
```
# Imports `RandomizedSearchCV`, which is used for searching hyperparameters of a model using a randomized approach.
from sklearn.model_selection import RandomizedSearchCV

# Create a hyperparameter space:
hyperparameter_space = {
    # Specify possible criteria:
    "criterion": ["squared_error"],
    # Specify strategies:
    "splitter": ["best"],
    # Specify the maximum depth of the tree:
```

```
    "max_depth": list(range(3, 46))
}
# Create a RandomizedSearchCV object:
random_search = RandomizedSearchCV(DecisionTreeRegressor(), param_distributions=hyperparameter_space, n_iter=43, cv=5, random_state=69)
# Perform hyperparameterised training:
random_search.fit(transformed_training_set, training_set_label)
```

Out[ ]:



Hyperparameterised model's testing

We obtain an improved model through a random search with a random max depth. Now, we proceed to perform testing on the training data with new and improved model and evaluate its performance.

In [ ]:

```
# Get the best model with the best hyperparameters:

# `random_search.best_estimator_`: After performing a hyperparameter search (e.g., using RandomizedSearchCV or GridSearchCV)
# the `best_estimator_ attribute` contains the model that performed best during the search, based on the scoring criteria provided.

final_model = random_search.best_estimator_
# Perform the prediction:
# final_model.predict(transformed_training_set): This line uses the best model (final_model) to predict the target values based on the input features from the transformed_training_set.

prediction_2 = final_model.predict(transformed_training_set)
# Get the mean absolute error of the improved model:
# `mean_absolute_error(training_set_label, prediction)`: This function calculates the Mean Absolute Error (MAE), which measures the average absolute difference between the actual values (`training_set_label`) and the predicted values (`prediction`).

mae = mean_absolute_error(training_set_label, prediction)
# Print out the mean absolute error for the improved model's prediction on the training set:
print(f"Final model's MAE on the training set: {mae}")
# Perform K-fold cross-evaluation with K = 5 using the MAE scoring system:

# `cross_val_score(...)``: This function performs K-fold cross-validation where the training data is split into K parts (in this case, 5).
# The model is trained on K-1 parts and tested on the remaining part. This process is repeated K times, each time with a different part as the test set.

# `scoring=neg_mean_absolute_error`: Specifies that the negative mean absolute error should be used as the scoring metric.
# Note that `cross_val_score()` returns negative values for MAE, so later on, the negative sign will be removed to get the actual MAE.

scores = cross_val_score(final_model, transformed_training_set, training_set_label, cv=5, scoring="neg_mean_absolute_error")
# Print out the cross-validated MAE:
print(f"Final model's cross-validated MAE score on the training set: {-scores.mean()}")
```

```
Final model's MAE on the training set: 0.8304587647070995
Final model's cross-validated MAE score on the training set: 0.02842754146180971
```

Model Testing and Analysis

We now move on to performing the prediction on the test set.

In [ ]:

```
# Preprocess the test set:

# column_transformer.fit_transform(test_set):
# This applies the transformations defined in `column_transformer` to `test_set`. It may involve operations like scaling, encoding, or other preprocessing steps.

transformed_test_set = column_transformer.fit_transform(test_set)
# Perform testing on the test data:

# `final_model.predict(transformed_test_set)`: The best model (final_model) obtained after hyperparameter tuning, is used to make predictions on `transformed_test_set`.

final_prediction = final_model.predict(transformed_test_set)
# Print out the performance of the final model:
print(f"Final model's MAE on the test set: {mean_absolute_error(test_set_label, final_prediction)}")
# Perform K-fold cross-evaluation with K = 5 using the MAE scoring system:

# `cross_val_score(...)`: This function performs K-fold cross-validation,
# splitting the test data into `k` parts (here, K = 5).
# The model is trained on K-1 parts and tested on the remaining part, repeated K times with a different part as the test set each time.

# `scoring=neg_mean_absolute_error`: Specifies that the negative mean absolute error should be used as the scoring metric.
# The negative sign is used because, by convention, `cross_val_score()` expects a higher score to be better, but since MAE is an error metric, it's flipped.

scores = cross_val_score(final_model, transformed_test_set, test_set_label, cv=5, scoring="neg_mean_absolute_error")
# Print out the cross-validated MAE:
print(f"Final model's cross-validated MAE score on the test set: {-scores.mean()}")
```

```
Final model's MAE on the test set: 0.12986869719636177
Final model's cross-validated MAE score on the test set: 0.05817570975097201
```

Analysis and suggestions

The resulting mean absolute error of approximately 0.1 and a cross-validated one of only 0.06, despite being less than the minimum error possible, has overwhelmingly surpassed our initial expectation of 3. The dataset used for training comes with excellent quality, allowing for highly effective data analysis and the accuracy of the model. The model's performance may improve with a larger dataset, as 96453 may not be a sufficient number of samples for the model to learn from.

Additionally, the graphs of the various relationships between the features may suggest a linear correlation between the `Temperature (C)` and the `Apparent Temperature (C)` columns. Further analysis might still be possible.