SCHOOL OF COMPUTER SCIENCE

Machine Learning and Parallel Computing

Assessment 2

Module Code	ITS 66604
Module Name	Machine Learning and Parallel Computing
Assessment	Assessment 2

Student Name	Hng Qi Yean
Student ID	0364483

Table of Content

1.0 Introduction

1.1 Background

With climate change and rapid urbanization, instances of extreme weather events such as heatwaves, heavy storms, flooding, and poor air quality are increasing in occurrence and intensity. Such events pose serious threats to public health, infrastructure, agriculture, and the environment. With weather becoming increasingly unpredictable, machine learning and data-driven systems must be developed to identify and respond to extreme conditions in a timely manner. Weather sensors and environmental monitoring produce large volumes of data that can be supplied to machine learning systems to provide intelligent classification and detection of abnormal weather patterns. The accurate classification of extreme weather events, or timely warnings thereof, can help governments, emergency services, and the public in the prompt exercise of informed decision-making and damage mitigation.

1.2 Problem Statement

Despite the tremendous amounts of weather data that sensors and monitoring systems pour in, the exact classification of meteorological conditions into normal weather or extreme conditions remains one of the biggest challenges in the domain. The fact of the matter is that there exist overlapping patterns of features between classes, that sensor readings are often inconsistent, that noise is present in the data, and that there is an imbalanced distribution of classes, with extreme weather being very rare. Methods based on simplistic rules may not appreciate subtle variations, or perhaps combinations of weather factors, that constitute extremes. Hence, a solution based on data classification needs to be developed that is capable of working with noisy inputs, learning the complexity of patterns, and disambiguating between normal and extreme meteorological events.

1.3 Objectives

The objective of this project is to develop a machine learning model that is capable of classifying weather conditions as either normal or extreme based on the chosen dataset. It also aims to evaluate and compare the performance of Decision Tree and Random Forest models to identify which model is more suitable for classification tasks.

2.0 Dataset Selection

2.1 Dataset Source

The dataset used in this project is called **World Weather Repository (Daily Updating)**. It was obtained from the Kaggle website, and the author of the dataset is **Nidula Elgiriyewithana**. (Elgiriyewithana, 2023) The dataset offers daily weather information gathered from various locations around the world. After downloading the dataset in CSV format from Kaggle, it was uploaded to **Google Drive** and integrated with **Google Colab** for further processing and model training.

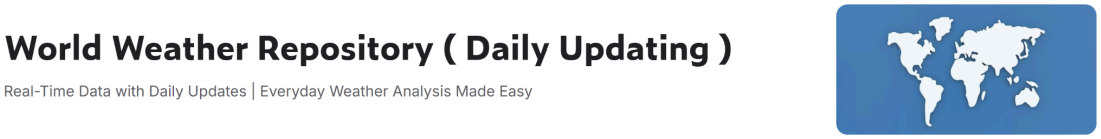

Figure 1. Title of the Dataset

2.2 Dataset Overview

The dataset is a collection of structured daily meteorological data collected from global capital cities. It contains up to 40 unique features, including temperature, humidity, wind speed, precipitation, air pressure, and various air quality indices like PM2.5 and Carbon Monoxide. Additionally, it also includes categorical data such as wind direction and descriptive weather conditions (e.g., “Cloudy,” “Rain,” “Thunderstorm”). The data are in a clean tabular form, which makes preprocessing easy and makes the dataset suitable for supervised machine learning.

country Country of the weather data	location_name Name of the location (city)	temperature_cels... Temperature in degrees Celsius	condition_text Weather condition description	wind_mph Wind speed in miles per hour	humidity Humidity as a percentage	air_quality_
Bulgaria 1% Indonesia 1% Other (79932) 98%	248 unique values	-24.9 … 49.2	Partly cloudy 32% Sunny 31% Other (30159) 37%	2.2 … 1.84k	2 … 100	0.17
Afghanistan	Kabul	26.6	Partly Cloudy	8.3	24	8.4
Albania	Tirana	19.0	Partly cloudy	6.9	94	1.1
Algeria	Algiers	23.0	Sunny	9.4	29	10.4
Andorra	Andorra La Vella	6.3	Light drizzle	7.4	61	0.7
Angola	Luanda	26.0	Partly cloudy	8.1	89	183.4
Antigua and Barbuda	Saint John's	26.0	Partly cloudy	5.6	84	1.2
Argentina	Buenos Aires	8.0	Clear	2.2	93	4.0
Armenia	Yerevan	19.0	Partly cloudy	4.3	40	0.8
Australia	Canberra	9.0	Clear	2.5	87	3.7
Austria	Vienna	16.0	Partly cloudy	12.5	63	3.7
Azerbaijan	Baku	17.0	Partly cloudy	4.3	68	1.9
Bahamas	Nassau	27.0	Partly cloudy	16.1	89	12.2
Bahrain	Manama	36.0	Sunny	13.6	33	39.0

Figure 2. Overview of the Dataset

2.3 Dataset Selection Justification

This dataset was chosen due to its wide and diverse set of features, making it highly suitable for classifying extreme weather events. It provides daily recorded weather data, which ensures greater accuracy and reliability for training supervised machine learning models. Moreover, with its focus on capital cities worldwide, the dataset offers global coverage while maintaining a consistent data structure across locations.

Most importantly, this dataset is particularly valuable because of its combination of over 40 features, including temperature, wind speed, air pressure, humidity, visibility, precipitation, and multiple air quality indices such as PM2.5 and Carbon Monoxide. This variety allows machine learning models to perform deeper analysis of the environmental factors contributing to extreme conditions.

Additionally, the presence of both numerical sensor readings and descriptive labels enables powerful feature engineering and supports hybrid analysis, which combines quantitative trends with qualitative observations. This makes it an ideal choice for exploring real-world weather classification tasks with a strong foundation for building interpretable and effective models.

3.0 Exploratory Data Analysis (EDA)

3.1 Summary Statistics

After uploading the dataset, the first step we do is to explore the dataset. Basic exploratory methods such as df.shape, df.describe(), and df.info() were used to understand the structure and distribution of the dataset.

```
[ ]  # Show the shape of the dataset (rows, columns)
     df.shape

     (81450, 41)
```

Figure 3. df.shape

By executing df.shape, we can observe that in the dataset, there are 81,450 rows (entries) and 41 columns (features).

```
[ ] # Generate summary statistics for numeric columns
    df.describe()
```

	latitude	longitude	last_updated_epoch	temperature_celsius	temperature_fahrenheit	wind_mph	wind_kph	wind_degree	pressure_mb
count	81450.00000	81450.000000	8.145000e+04	81450.000000	81450.000000	81450.000000	81450.000000	81450.000000	81450.000000
mean	19.12799	22.180716	1.733991e+09	22.468751	72.445419	8.308431	13.374532	170.019804	1014.027612
std	24.47210	65.829918	1.048374e+07	9.210042	16.577830	8.352727	13.440224	103.365641	12.089679
min	-41.30000	-175.200000	1.715849e+09	-24.900000	-12.800000	2.200000	3.600000	1.000000	947.000000
25%	3.75000	-6.836100	1.725020e+09	17.300000	63.100000	4.000000	6.500000	81.000000	1010.000000
50%	17.25000	23.316700	1.734000e+09	24.900000	76.800000	6.900000	11.200000	162.000000	1013.000000
75%	40.40000	50.580000	1.743067e+09	28.400000	83.100000	11.400000	18.400000	257.000000	1018.000000
max	64.15000	179.220000	1.752051e+09	49.200000	120.600000	1841.200000	2963.200000	360.000000	3006.000000

Figure 4. df.describe()

By executing df.describe(), we can obtain summary statistics for each numerical feature of the dataset, including count, mean, standard deviation and more.

```
[ ] # Step 3: Data Exploration
    # ========================

    # Display column names, data types, and non-null counts
    df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 81450 entries, 0 to 81449
Data columns (total 41 columns):
 #   Column                        Non-Null Count  Dtype
---  ------                        --------------  -----
 0   country                       81450 non-null  object
 1   location_name                 81450 non-null  object
 2   latitude                      81450 non-null  float64
 3   longitude                     81450 non-null  float64
 4   timezone                      81450 non-null  object
 5   last_updated_epoch            81450 non-null  int64
 6   last_updated                  81450 non-null  object
 7   temperature_celsius           81450 non-null  float64
 8   temperature_fahrenheit        81450 non-null  float64
 9   condition_text                81450 non-null  object
 10  wind_mph                      81450 non-null  float64
 11  wind_kph                      81450 non-null  float64
 12  wind_degree                   81450 non-null  int64
 13  wind_direction                81450 non-null  object
```

Figure 5. df.info()

By executing df.info(), we can know the concise summary of the dataset, including the number of entries, column names, data types, and the number of non-null values in each feature.

3.2 Checking Missing Values

To check the missing values of the dataset, we use df.isnull().sum() command. This step helps to identify the number of missing or null entries in each column (feature), which is very essential for handling data quality.

```
[ ]  # Count missing (null) values in each column
     df.isnull().sum()
```

	0
country	0
location_name	0
latitude	0
longitude	0
timezone	0
last_updated_epoch	0
last_updated	0
temperature_celsius	0
temperature_fahrenheit	0
condition_text	0
wind_mph	0
wind_kph	0
wind_degree	0

Figure 6. Checking Missing Values

3.3 Correlation Matrix Heatmap

To understand the relationships between variables, a correlation matrix heatmap was also created by using Seaborn. This makes it easier to detect multicollinearity and figure out which features might have a bigger impact on the machine learning model's predictions. (Chip, 2023)

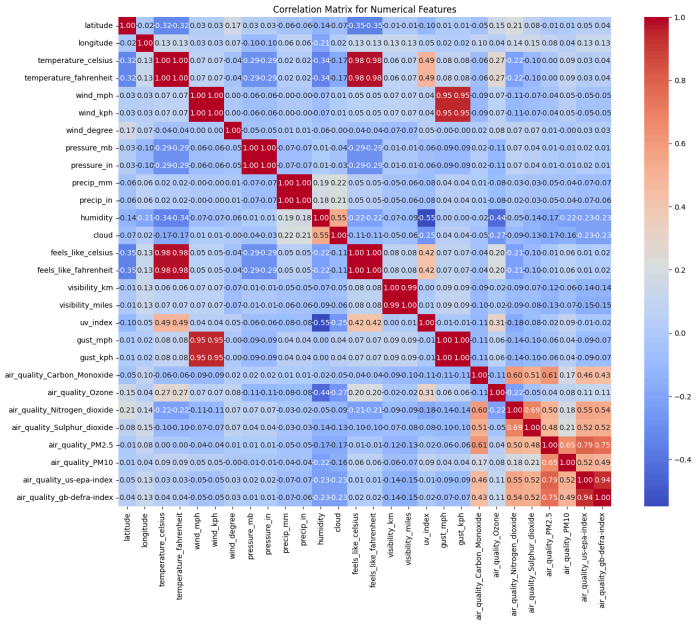

Figure 7. Correlation Matrix Heatmap

4.0 Data Preprocessing

4.1 Dropping Irrelevant Columns

In order to facilitate the process of training the models, we have to go through an important step, which is dataset cleaning, also known as data preprocessing. Even though the dataset consists of 40 features, some of them are irrelevant or not very important, which will only affect the results of testing and reduce the model performance. These include features such as location_name, last_updated, sunrise, moon_phase and moon_illumination. Although these features might hold contextual value in other scenarios, they do not contribute significantly to the classification task. Therefore, these columns were dropped using the df.drop() command.

```
# Define a list of non-essential or redundant columns to remove
drop_cols = [
    'location_name', 'last_updated', 'last_updated_epoch',
    'sunrise', 'sunset', 'moonrise', 'moonset',
    'moon_phase', 'moon_illumination'
]

# Drop the specified columns from the dataset to simplify the feature set
df.drop(columns=drop_cols, inplace=True)
```

Figure 8. Dropping Irrelevant Columns

4.2 Handling Anomalies

After dropping irrelevant columns, the next step is to handle anomaly data in the dataset. Sensor-based weather datasets often contain missing values or fault readings. In our dataset, a few extreme outliers such as -9999 and -1848.15 were detected in air quality columns, for instance, air_quality_Carbon_Monoxide, air_quality_Sulphur_dioxide, and air_quality_PM10. These values are clearly not realistic and would only negatively affect the calculations and predictions of the machine learning model. To handle this, these values were replaced with NaN using the .replace() function. Once flagged as missing, these NaN values were imputed with the median of each respective column. This step ensures cleaner input data and thereby improves both model accuracy and reliability.

```
# Anomaly Handling for Sensor Data

# Replace known placeholder anomalies with NaN
anomaly_cols = ['air_quality_Carbon_Monoxide', 'air_quality_Sulphur_dioxide', 'air_quality_PM10']
for col in anomaly_cols:
    df[col] = df[col].replace(-9999, np.nan)
    df[col] = df[col].replace(-1848.15, np.nan)  # for PM10

# Impute missing values with median (more robust than mean)
for col in anomaly_cols:
    if df[col].isnull().sum() > 0:
        median_val = df[col].median()
        df[col].fillna(median_val, inplace=True)
```

Figure 9. Handling Anomalies

5.0 Label Engineering

5.1 Custom Definition of "Extreme Weather"

Next, we need to classify weather conditions, but before that, we have to establish the threshold that identifies **"Extreme"** weather. A rule-based function was created to define the **"Extreme"** weather and it is according to the scores of each row based on how many environmental thresholds it exceeds, for example, temperature above 32°C or below 0°C , wind speed over 35 kph, etc. If a data point meets three or above of these thresholds, it is classified as **"Extreme"** weather; otherwise, it is labeled as **"Normal"**.

```
# Function to assign 'Extreme' or 'Normal' based on weather thresholds
def label_extreme_harder(row):
    score = 0

    # Check if each weather condition exceeds threshold and add to score
    if row['temperature_celsius'] > 32 or row['temperature_celsius'] < 0:
        score += 1
    if row['air_quality_PM2.5'] > 65:
        score += 1
    if row['humidity'] > 85 or row['humidity'] < 25:
        score += 1
    if row['wind_kph'] > 35:
        score += 1
    if row['precip_mm'] > 8:
        score += 1
    if row['air_quality_Carbon_Monoxide'] > 4:
        score += 1

    # Label as 'Extreme' if 3 or more conditions are met, else 'Normal'
    return 'Extreme' if score >= 3 else 'Normal'

# Apply labeling function to the DataFrame
df['weather_extreme'] = df.apply(label_extreme_harder, axis=1)
```

Figure 10. Custom Definition of "Extreme Weather"

After that, a count plot was created to show the distribution of **"Normal"** vs **"Extreme"** weather labels. As we can see from the plot below, the dataset is imbalanced, with significantly more **"Normal"** examples than the **"Extreme",** highlighting the need for resampling techniques in later steps.

```
[13] # Plot the distribution of 'weather_extreme' labels using a count plot
     plt.figure(figsize=(6, 4))
     sns.countplot(data=df, x='weather_extreme', palette='viridis')
     plt.title("Distribution of Weather Labels")
     plt.xlabel("Weather Label")
     plt.ylabel("Count")
     plt.tight_layout()
     plt.show()
```

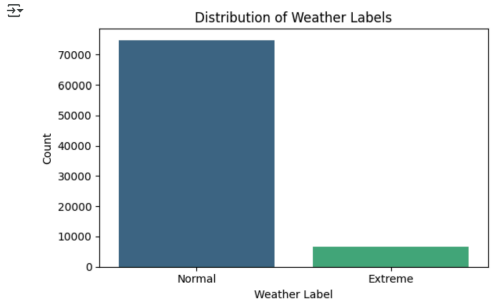

Figure 11. Distribution of Weather Labels

6.0 Feature Engineering

6.1 Noise Injection

Next, in order to prevent overfitting and ensure the models not only generalize well on training data but also on test data, we have to input some noise into the dataset. By adding Gaussian noise into features, it enables the machine learning model to simulate real-world sensor variability. This noise injection technique helps to improve model performance by exposing it to slight fluctuations that resemble natural measurement errors, making the model more generalizable. A total of 20% noise scale was added to the original data to ensure meaningful yet controlled distortion.

```
# 1. Add Gaussian noise to selected numeric features to simulate sensor variation
for col in ['humidity', 'wind_kph', 'precip_mm']:
    std_dev = df[col].std()
    noise = np.random.normal(loc=0.0, scale=0.2 * std_dev, size=len(df))  # Apply stronger noise (20% of std)
    df[col] += noise  # Add noise to the original data
```

Figure 12. Noise Injection

6.2 Interaction and Distractor Features

To improve the learnability of the models, we generated new features from the data. For example, the humidity_temp_ratio and wind_temp_product were computed using temperature, humidity, and wind speed, which included relationships between different weather variables. Besides, a precip_shifted column was generated with a slight amount of random noise to introduce random shifts to mimic unpredictable rain shifts in precipitation patterns.

```
# 2. Create interaction-based noisy features to increase complexity
df['humidity_temp_ratio'] = df['humidity'] / (df['temperature_celsius'] + 1)  # Avoid division by zero
df['wind_temp_product'] = df['wind_kph'] * df['temperature_celsius']         # Multiplicative interaction
df['precip_shifted'] = df['precip_mm'] + np.random.normal(0, 0.3, len(df))   # Add more subtle noise
```

Figure 13. Example of Newly Generated Features

Two columns of random noise (random_noise_1, random_noise_2) were also added as distractors. This helps to measure how quickly and how accurately the model distinguishes informative features from irrelevant features and builds resilience against noise.

```
# 3. Add purely random features to act as noise/distractors for the model
df['random_noise_1'] = np.random.normal(0, 1, len(df))        # Gaussian noise (mean=0, std=1)
df['random_noise_2'] = np.random.randint(0, 100, len(df))     # Random integers between 0 and 100
```

Figure 14. Example of Distractor Features

7.0 Class Imbalance Handling

7.1 Problem of Imbalance

In many real-world classification tasks, including weather event detection, datasets are often **imbalanced**, meaning that one class significantly outnumbers the other one. In this project, the majority of data points were labeled as **"Normal"** weather, while the **"Extreme"** weather instances were only a few. This imbalanced circumstances will lead to a serious challenge because most of the machine learning algorithms assume that classes are relatively balanced. If this is not addressed, the model may become biased toward the dominant class, leading to high accuracy but poor detection of the minority class (Ahmad, 2024), in this case, the rare **"Extreme"** events we aim to classify. Such skewed learning methods can lead to inaccurate model assessments and poor operational performance in crucial applications such as weather warnings. Therefore, it is essential to balance the dataset when we want to ensure that the model learns to classify both classes effectively and fairly.

7.2 Oversampling

To address the imbalance problem, we implemented the **random oversampling** method to the minority class (**"Extreme"**). Firstly, we separate the target variable (weather_extreme) from the features and use the resample() function to duplicate rows from the minority class until it matches the number of **"Normal"** samples. Then, the oversampled data was combined and shuffled using sample() function to ensure the randomness of the new data. This balancing step helps to prevent the model from being biased toward the majority class and helps it learn from both classes equally. (Brownlee, 2020)

```
# Separate input features and target variable
X = df.drop(columns=['weather_extreme'])
y = df['weather_extreme']

# Combine features and target into one DataFrame for easy resampling
df_combined = pd.concat([X, y], axis=1)

# Split data by class labels
df_normal = df_combined[df_combined['weather_extreme'] == 'Normal']     # Majority class
df_extreme = df_combined[df_combined['weather_extreme'] == 'Extreme']   # Minority class

# Upsample the minority class to match the majority class size
df_extreme_upsampled = resample(
    df_extreme,
    replace=True,                       # Allow duplicates
    n_samples=len(df_normal),           # Match number of Normal samples
    random_state=42                     # Ensure reproducibility
)

# Combine the balanced data
df_balanced = pd.concat([df_normal, df_extreme_upsampled])

# Shuffle the rows to mix Normal and Extreme samples
df_balanced = df_balanced.sample(frac=1, random_state=42).reset_index(drop=True)
```

Figure 15. Oversampling

After oversampling, we plotted a bar chart using the Seaborn's countplot() command to confirm that the dataset is now balanced. The bar chart below clearly shows an equal number of samples for both **"Normal"** and **"Extreme"** weather labels. The visual inspection of class distribution ensures that the model will avoid bias during training and provide equal treatment to both classes.

```
# Create a bar plot to show the distribution of labels in the balanced dataset

plt.figure(figsize=(6, 4))
sns.countplot(data=df_balanced, x='weather_extreme', palette='Set2')
plt.title("Distribution of Weather Labels After Resampling")
plt.xlabel("Weather Label")
plt.ylabel("Count")
plt.tight_layout()
plt.show()
```

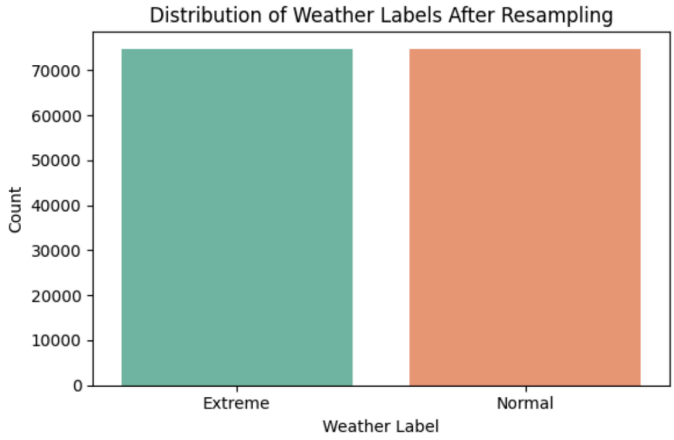

Figure 16. Distribution of Weather Labels after Resampling

8.0 Data Encoding and Scaling

8.1 Encoding Categorical Features

Since machine learning models only accept and understand numerical input to perform or execute tasks, there is a need to transform all categorical input into numerical form. In this case, **LabelEncoder** was used to convert categorical columns into numerical values. This step is essential because models such as Decision Trees and Random Forests require numerical input to perform splits and make decisions. (Sethi, 2020)

```
# Make a copy of the features to perform encoding
X_encoded = X_bal.copy()

# Label encode categorical features (if any) for machine learning compatibility
for col in X_encoded.select_dtypes(include='object').columns:
    le = LabelEncoder()
    X_encoded[col] = le.fit_transform(X_encoded[col])
```

Figure 17. LabelEncoder

Furthermore, the feature humidity_temp_ratio was intentionally removed to reduce overfitting, as it was found to be the most important feature related to the classification. This helps promote better generalization in the models.

```
# Drop a highly correlated or redundant feature to reduce model bias
X_bal.drop(columns=['humidity_temp_ratio'], inplace=True)
```

Figure 18. Removal of the feature - humidity_temp_ratio

8.2 Handling Missing/Infinities

After encoding, all infinite values were replaced with NaN to maintain data consistency and ensure there were no runtime errors while training the models. This is because these infinities may result from mathematical operations such as division by zero or logarithmic transformations. Then, we imputed each column containing missing values (NaN) with the mean of that column. This simple imputation strategy helps to maintain dataset integrity while avoiding the loss of valuable data during row deletion.

```
# Replace infinite values with NaN (to handle any unexpected math operations)
X_encoded = X_encoded.replace([np.inf, -np.inf], np.nan)

# Fill missing values (NaNs) with the mean of each column
for col in X_encoded.columns:
    if X_encoded[col].isnull().any():
        mean_val = X_encoded[col].mean()
        X_encoded[col].fillna(mean_val, inplace=True)
```

Figure 19. Handling Missing/Infinities Values

8.3 Feature Scaling

Next, to ensure that all numerical features are contributed equally during model training, feature scaling was applied using StandardScaler. This method transforms the features so they have a mean of 0 and a standard deviation of 1. (Bhandari, 2020c) Even though tree-based models such as Decision Trees and Random Forests are generally insensitive to feature scaling, applying it helps to maintain consistency and can be useful if additional models are tested later.

```
# Standardize the features to have zero mean and unit variance
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X_encoded)
```

Figure 20. Feature Scaling

9.0 Train-Test Split and Model Preparation

9.1 Train / Test Split

Moving on, in order to evaluate each model's performance effectively, the dataset was split into training and testing sets with a ratio of 80:20. The stratify parameter was employed to maintain the same class distribution in both sets. A random_state was set to ensure the split remains consistent across multiple runs. This kind of split allows for a more accurate representation of how well each model will generalize on previously unseen data.

```
# Split the dataset into training and testing sets
# - test_size=0.2 means 20% of the data will be used for testing
# - stratify=y_bal ensures the class distribution is preserved in both sets
# - random_state ensures reproducibility of the split

X_train, X_test, y_train, y_test = train_test_split(
    X_scaled, y_bal, test_size=0.2, stratify=y_bal, random_state=42
)
```

Figure 21. Train/Test Split

9.2 Model Selection and Rationale (Justification)

In this project, we had selected and implemented two tree-based classification algorithms, which are **Decision Tree** and **Random Forest.** They were chosen because of their strong effectiveness for tabular data, their easy interpretation, and their strong ability to work with numerical as well as categorical data without the need for intense preprocessing.

9.2.1 Decision Tree Classifier

The Decision Tree model was used because of its inherent simplicity and transparency. It builds a tree-like structure, a sort of flowchart, which makes it easy to understand how the input features are used to classify weather conditions. (Saini, 2021) Through this transparency, it is easier to establish which environmental factors are used to assign the label of **"Extreme"** weather. Additionally, Decision Trees handle non-linear relationships well and are not sensitive to feature scaling or missing values.

9.2.2 Random Forest Classifier

The Random Forest is considered an ensemble method because it builds a number of Decision Trees and averages their results to improve generalization. This method was adopted to reduce the probability of overfitting and variance issues which are typically associated with a single decision tree. Moreover, Random Forest provides model stabilization over time and on average improves accuracy because it relies on multiple trees instead of averaging a single Decision Tree. Random Forest also averages out noise in a dataset which can be very prevalent with real-world weather measurements. (Meenal et al., 2021)

Together, these models provide a balance between interpretability (Decision Tree) and predictive power (Random Forest).

10.0 Decision Tree Model

10.1 Parameters used

The Decision Tree Classifier was structured with specific hyperparameters to control its complexity and enhance generalization. The max_depth was set to 3 to limit the depth of the tree, making the model easier to interpret and reducing the risk of overfitting. Additionally, min_samples_leaf was set to 50 to ensure that each terminal node has a sufficient number of samples, helping the tree avoid learning from noise in small subsets. The min_samples_split was fixed at 80 to prevent the model from splitting unless a node has at least 80 samples. Lastly, a random_state of 42 was used to maintain consistency in results across different runs. Overall, these hyperparameters were used to produce a model that is balanced and interpretable, while capturing relevant weather patterns without excessive complexity.

```
# Initialize the Decision Tree model with limited depth and split constraints to avoid overfitting
dt_model = DecisionTreeClassifier(
    max_depth=3,            # Maximum depth of the tree
    min_samples_leaf=50,    # Minimum number of samples required at a leaf node
    min_samples_split=80,   # Minimum number of samples required to split an internal node
    random_state=42         # For reproducibility
)

# Train the model using the training data
dt_model.fit(X_train, y_train)

# Predict the class labels on the test set
y_pred_dt = dt_model.predict(X_test)
```

Figure 22. Structure of the Decision Tree Classifier

10.2 Accuracy and Classification Report

The performance of the Decision Tree model was assessed based on accuracy, precision, recall and f1-score. Accuracy reflects overall performance, while the classification report indicates performance for each of the classes.

```
Decision Tree Performance:
Accuracy: 0.9332954393473318
Confusion Matrix:
 [[13955   999]
 [  996 13958]]
Classification Report:
              precision    recall  f1-score   support

     Extreme       0.93      0.93      0.93     14954
      Normal       0.93      0.93      0.93     14954

    accuracy                           0.93     29908
   macro avg       0.93      0.93      0.93     29908
weighted avg       0.93      0.93      0.93     29908
```

Figure 23. Accuracy and Classification Report of the Decision Tree Classifier

The Decision Tree classifier reached a maximum accuracy of 93.3%, thus highlighting its significant ability to classify both extreme and normal weather instances. By taking a look at the classification report, it can be found that the classifier performs similarly for the two classes, as the values of precision, recall, and F1-score are all nearly 0.93 for the **"Extreme"** and **"Normal"** classifications. Therefore, the Decision Tree classifier indicates that it generalized well in making relevant distinctions between extreme weather events and normal weather events.

10.3 Confusion Matrix

The confusion matrix below provides a complete overview of both correct and incorrect predictions made by the Decision Tree model. It shows how many instances were accurately classified as either **"Extreme"** or **"Normal"** , as well as the number of misclassification instances that were made. (Bhandari, 2020b)

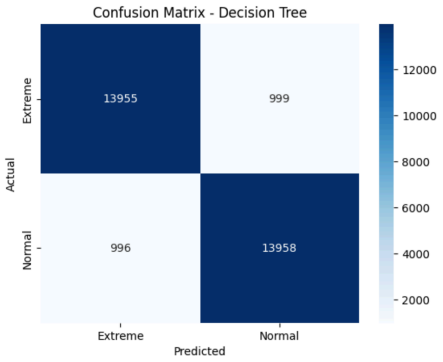

Figure 24. Confusion Matrix of the Decision Tree Classifier

Based on the confusion matrix above, the Decision Tree model correctly classified 13,955 extreme cases and 13,958 normal cases, with only 999 and 996 misclassifications respectively. This almost perfect diagonal configuration indicates that the model has a substantive level of accuracy and balance to its prediction, effectively distinguishing between extreme and normal weather conditions.

10.4 ROC Curve

ROC (Receiver Operating Characteristic) curves determine the Decision Tree model's ability to differentiate **"Extreme"** and **"Normal"** weather events. The ROC curve describes the True Positive Rate, versus the False Positive Rate, at specific thresholds of classification. The AUC (Area Under Curve) score provides a summary measure of how well the classifier performs based on the ROC curve; the higher the score is to 1.0, the better the classification performance. Such a measure reveals information about the model's performance that goes beyond mere accuracy, specifically when encountering datasets that are imbalanced. (Bhandari, 2020a)

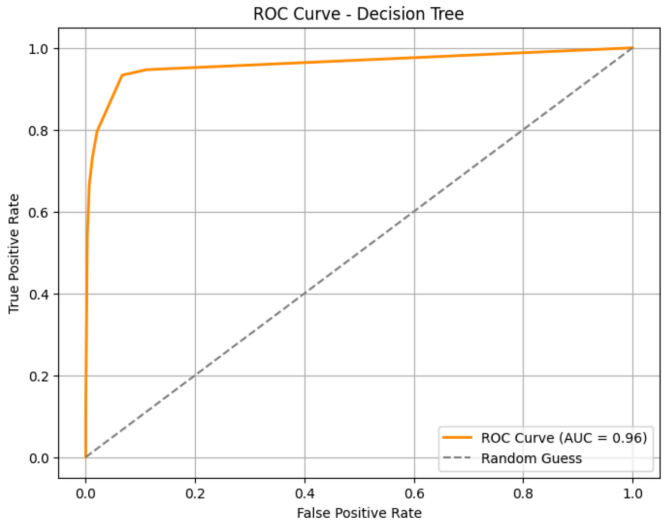

Figure 25. ROC Curve and AUC Score of the Decision Tree Classifier

The ROC curve above shows a strong classification performance, with an AUC (Area Under the Curve) of 0.96. This indicates that the Decision Tree model has a strong ability to distinguish severe weather events from common weather events.

10.5 Visualized Tree Plot

To facilitate interpretability, the trained Decision Tree model was visualized as a tree diagram. Each node shows a decision based upon a feature and a threshold, thus helping to ease the determination of the logical process the model uses to classify weather conditions.

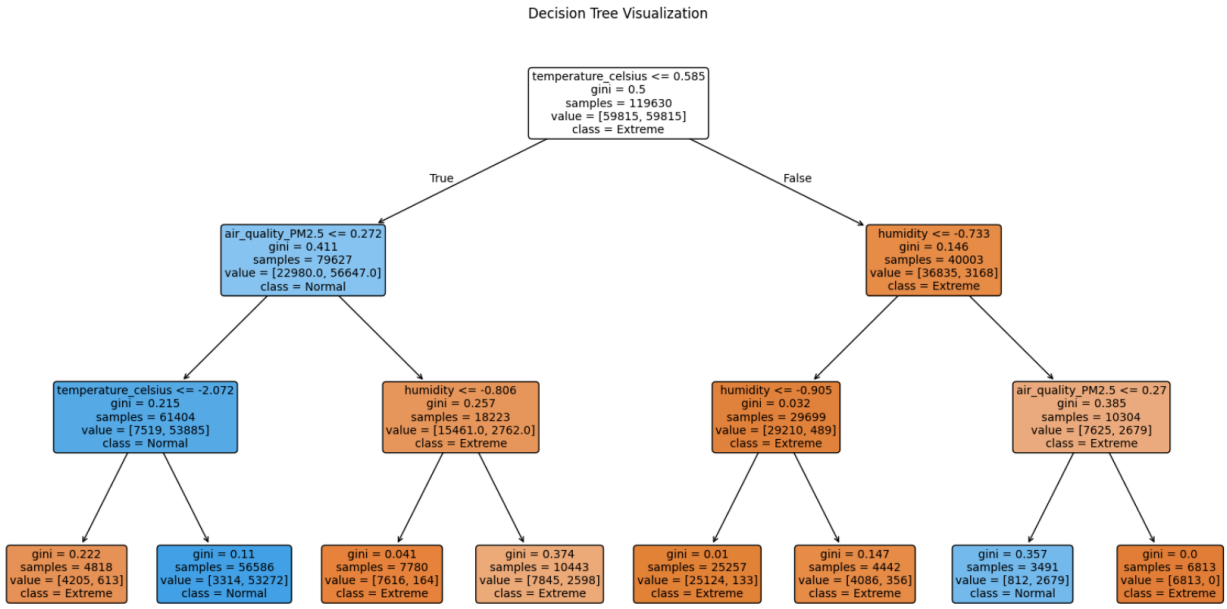

Figure 26. Visualization of the Decision Tree Classifier

The visualized Decision Tree structure is shown in the above figure. The tree begins with a root node that performs the first partitioning according to the feature which produces the greatest information value. Then, the internal nodes represent a condition or a cutting edge that serves to segment the data further, whereas the leaf nodes indicate the final classification result, labeled as **“Extreme”** or **“Normal”**. In the visualization, blue nodes represent instances classified as **“Normal”,** while red nodes indicate **“Extreme”** classifications.

11.0 Random Forest Model

11.1 Parameters used

A Random Forest classifier was implemented as the second model due to its ensemble learning capability, which combines multiple decision trees to improve accuracy and robustness. (Meenal et al., 2021) The model configuration included 30 trees, with a limit of 3 for the maximum depth in order to avoid the risk of overfitting. Furthermore, parameters min_samples_leaf and min_samples_split were set to enhance generalization, while the setting of max_features='sqrt' ensures the diversity of trees by inhibiting the number of features considered during the splitting process. The model was then trained on the training set and used to predict weather classifications on the test set.

```
# Step 11: Model Training 2 - Random Forest Classifier
# =====================================================

# Initialize a Random Forest classifier with limited depth and sample constraints
# to reduce overfitting and encourage generalization
rf_model = RandomForestClassifier(
    n_estimators=30,         # Number of trees in the forest
    max_depth=5,             # Limit tree depth
    min_samples_leaf=10,     # Minimum samples required at a leaf node
    min_samples_split=20,    # Minimum samples required to split an internal node
    max_features='sqrt',     # Use square root of features at each split (faster, more diverse trees)
    random_state=42          # Ensures reproducibility
)

# Train the Random Forest model
rf_model.fit(X_train, y_train)

# Make predictions on the test set
y_pred_rf = rf_model.predict(X_test)
```

Figure 27. Structure of the Random Forest Classifier

11.2 Accuracy and Classification Report

The results of the Random Forest model were evaluated using accuracy, confusion matrix, and classification report. Based on evaluation results, it was determined that the Random Forest model performed better with higher precision, recall, and F1-score compared to the Decision Tree. This is because ensemble methods such as the Random Forest model are superior at identifying complex relationships and reducing variability, ultimately performing better when the input data is noisy and highly unbalanced.

```
Random Forest Performance:
Accuracy: 0.9523204493780928
Confusion Matrix:
 [[14730   224]
 [ 1202 13752]]
Classification Report:
              precision    recall  f1-score   support

     Extreme       0.92      0.99      0.95     14954
      Normal       0.98      0.92      0.95     14954

    accuracy                           0.95     29908
   macro avg       0.95      0.95      0.95     29908
weighted avg       0.95      0.95      0.95     29908
```

Figure 28. Accuracy and Classification Report of the Random Forest Classifier

In a nutshell, the Random Forest model displayed strong performance indicators when run on the test set, with an overall accuracy of almost 0.95. The classification report shows that precision, recall, and F1-scores are closely matched between the two classes, with all values ranging from 0.92 to 0.99. This result implies that the model has an equal effectiveness in classifying both types of weather conditions. The higher F1-score, which is 0.95, implies a strong balance between precision and recall. This performance highlights the strength of ensemble learning, where multiple decision trees work together to reduce overfitting and improve generalization.

11.3 Confusion Matrix

The confusion matrix below provides a complete overview of both correct and incorrect predictions made by the Random Forest model. It shows how many instances were accurately classified as either **"Extreme"** or **"Normal"** , as well as the number of misclassification instances that were made.

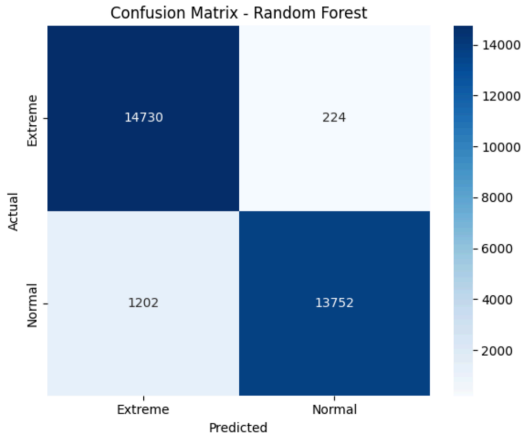

Figure 29. Confusion Matrix of Random Forest Classifier

The confusion matrix for the Random Forest model shows strong classification performance. Out of all actual **"Extreme"** cases, 14,730 were correctly predicted, with only 224 misclassified as **"Normal"**. Similarly, for the **"Normal"** class, 13,752 predictions were correct, while 1,202 were incorrectly labeled as **"Extreme"**. This indicates the model performs well on both classes, though it shows slightly more false positives for the **"Extreme"** category. Overall, the matrix shows strong accuracy and balanced capability of predicting both kinds of meteorological events.

11.4 ROC Curve

For the Random Forest model, the ROC curve shows strong separation between the classes, with a high AUC value indicating excellent performance in identifying extreme weather conditions. This reflects the model's ability to correctly rank predictions and maintain a good balance between sensitivity and specificity, even under class imbalance.

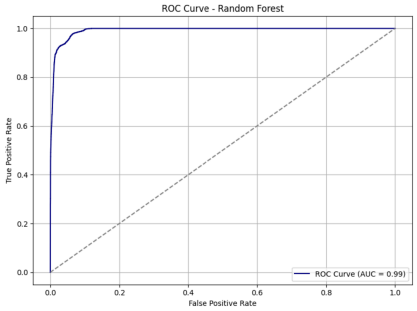

Figure 30. ROC Curve and AUC Score of the Random Forest Classifier

With an AUC (Area Under the Curve) score of 0.99, the model demonstrates near-perfect classification performance. This high AUC value indicates that the model is highly effective at ranking positive instances higher than negative ones, making it a strong tool for detecting extreme weather events with minimal trade-offs between true positives and false positives.

11.5 Feature Importance Visualization

The feature importance plot below shows which variables contributed the most to the predictions of the Random Forest model. While temperature_celsius, humidity, wind speed, and air quality generally appeared high on the list as the most important features, the plot allows one to understand and visualize the model's decision-making and facilitates additional feature enhancement or reduction. (Thorn, 2020)

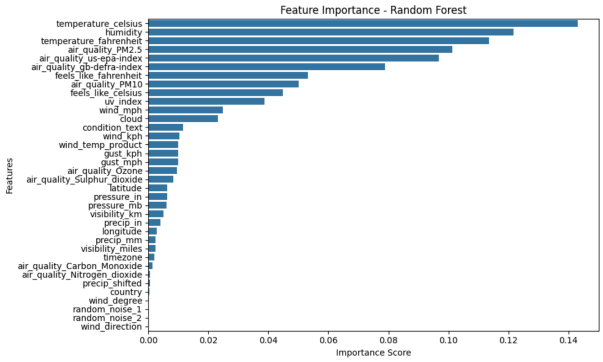

Figure 31. Feature Importance of Random Forest

12.0 Discussion

12.1 Challenges Faced and Applied Optimization Techniques

While developing the model, several key challenges emerged that significantly impacted the efficiency and reliability of the classification model. One key problem that surfaced was **class imbalance**. The dataset started off with an appreciably larger number of instances labeled as **"Normal"** weather compared to instances labeled as **"Extreme"** weather, which might bias the results of the model towards the majority class. To counteract this problem, I applied minority class upsampling via bootstrapping. This approach allowed for an equitable representation of the two classes, thereby boosting the model's capability to effectively distinguish between extreme events.

Another challenge encountered was the presence of noisy data, primarily caused by environmental factors such as humidity, wind speed, and rain, which introduced variability in sensor outputs. Rather than treating all noise as a drawback, this variability was embraced as part of real-world complexity. To further improve the model's robustness and prevent overfitting to ideal or overly clean datasets, Gaussian noise was intentionally added to particular features. This way, the model was able to remain more flexible and capable of adapting to sensor reading variations, hence avoiding the likelihood of overfitting on too clean or idealized datasets.

Additionally, during the training process, the model displayed signs of overfitting, where it performed well on training data but less effectively on unseen test data. After carrying out an analysis of feature importance, I discovered that the feature humidity_temp_ratio was an important factor in impacting the model's decisions. While it was a strong predictor, it also introduced a risk of the model relying too much on a single input. To counter this, I chose to remove the humidity_temp_ratio feature to encourage the model to consider a broader range of inputs. This not only helped to reduce the accuracy, but also resulted in improved generalization and an improved natural performance pattern.

Overall, a combination of preprocessing techniques and strategic feature adjustments allowed the final models to handle real-world data challenges more effectively and perform with increased reliability.

12.2 Performance Comparison

To compare the performance of the Decision Tree and Random Forest models, a number of classification metrics were employed, such as accuracy, precision, recall, F1-score, confusion matrix, and ROC-AUC. Although both models showed competent performance, the Random Forest model outperformed the Decision Tree model on a number of metrics.

The Random Forest's accuracy was a little higher than the Decision Tree's, pointing to superior overall predictive power. More significantly, the F1-score for the "Extreme" class, which measures the tradeoff between precision and recall, was also significantly higher for the Random Forest model. This implies that the ensemble technique was more effective at predicting extreme weather events correctly while avoiding false positives.

The confusion matrix also gave a deeper understanding of the performance of the models. The Random Forest had a more balanced classification with less misclassifications in both classes. The Decision Tree, meanwhile, had a tendency to misclassify more of the extreme events as normal, possibly because of its shallowness and propensity to overfit or underfit with slight changes in the data.

The ROC Curve and AUC also attested to the superiority of the Random Forest. It had a greater AUC score than the Decision Tree, proving to have better discriminatory power between the two classes.

In conclusion, although the Decision Tree had the benefit of interpretability and simplicity, the Random Forest model was more reliable and accurate for weather classification problems. Its ensemble approach allowed it to capture complex patterns and reduce variance, making it a better fit for the imbalanced and noisy dataset used in this project.

12.3 Feature Insights

After completing the model training, evaluation, and feature importance visualization, it became evident that certain variables consistently played a major role in predicting extreme weather conditions. Besides the manually removed humidity_temp_ratio, which was excluded on purpose to reduce overfitting, features including temperature_celsius, humidity, and air_quality_PM2.5 stood out as some of the most important predictors in the Random Forest model.

Interestingly, these results aligned when compared with real-world weather phenomena. Temperature and humidity are known to drive many extreme events, for example, heatwaves, storms, or heavy rain. Unusually high or low temperatures can signal abnormal weather, while high humidity is often linked to storm development or heat stress. The PM2.5 feature, which tracks fine particulate matter, also seems rational as it tends to rise during wildfires, smog episodes, or droughts, all of which can be considered extreme weather conditions.

Another key insight was that the combination of features, rather than any single variable, contributed to better model accuracy. For example, moderate humidity might not seem extreme by itself, but when paired with high temperatures or poor air quality, it could indicate an extreme event. This shows how important it is to look at interactions between factors, instead of relying on single-threshold rules.

Additionally, particular features that seemed to be important, like wind direction or location, received much lower importance ratings from the model. This shows that, despite the contextual plausibility of the latter categorical features, they did little to improve the predictive effectiveness of the decision regions defined by the models.

In short, feature importance analysis not only helps interpret the model but also validates domain knowledge, showing that the selected features reflect real-world weather phenomena.

13.0 Conclusion

In conclusion, this assessment successfully developed and evaluated machine learning models for classifying weather conditions as either **“Extreme”** or **“Normal”**. By implementing Decision Tree and Random Forest classifiers, it demonstrated that extreme weather events can be effectively classified based on environmental sensor data, even in the presence of data noise and class imbalance.

Key challenges such as missing values, anomalies, and uneven class distributions were addressed using median imputation, noise injection, and oversampling techniques. Feature engineering, including the creation of interaction terms and handling of categorical variables, played a significant role in improving both model performance and interpretability.

Of the two models created, Random Forest demonstrated the better accuracy, recall, and generalizability of the models primarily due to the ensemble learning of the Random Forest model, which mitigates overfitting while increasing model stability.

In the future, there are opportunities to further improve the model by implementing more sophisticated resampling techniques such as SMOTE, or exploring the seasonal and temporal trends of weather events using time-series models, or applying deep learning algorithms for richer and advanced pattern recognition. (SATPATHY, 2020)

Additionally, this predictive machine learning model could also be integrated into existing weather monitoring applications, allowing for more timely warnings about predicted extreme weather event occurrences.

Overall, the findings from this assessment highlight the potential of machine learning in environmental applications and establish a solid foundation for future improvements and deployment in real-world systems.

14.0 References

Dataset Source - https://www.kaggle.com/datasets/nelgiriyewithana/global-weather-repository

Ahmad, A. (2024). *Class Imbalance Problem - an overview | ScienceDirect Topics*. [online] www.sciencedirect.com. Available at: https://www.sciencedirect.com/topics/computer-science/class-imbalance-problem.

Bhandari, A. (2020a). *AUC-ROC Curve in Machine Learning Clearly Explained*. [online] Analytics Vidhya. Available at: https://www.analyticsvidhya.com/blog/2020/06/auc-roc-curve-machine-learning/.

Bhandari, A. (2020b). *Confusion matrix for machine learning*. [online] Analytics Vidhya. Available at: https://www.analyticsvidhya.com/blog/2020/04/confusion-matrix-machine-learning/.

Bhandari, A. (2020c). *Feature Scaling | Standardization Vs Normalization*. [online] Analytics Vidhya. Available at: https://www.analyticsvidhya.com/blog/2020/04/feature-scaling-machine-learning-normalization-standardization/.

Brownlee, J. (2020). *Random Oversampling and Undersampling for Imbalanced Classification*. [online] Machine Learning Mastery. Available at: https://machinelearningmastery.com/random-oversampling-and-undersampling-for-imbalanced-classification/.

Chip (2023). *How to read a correlation heatmap?* -. [online] quanthub. Available at: https://www.quanthub.com/how-to-read-a-correlation-heatmap/.

Elgiriyewithana, N. (2023). *World Weather Repository (Daily Updates)*. [online] www.kaggle.com. Available at: https://www.kaggle.com/datasets/nelgiriyewithana/global-weather-repository.

Meenal, R., Michael, P.A., Pamela, D. and Rajasekaran, E. (2021). Weather prediction using random forest machine learning model. *Indonesian Journal of Electrical Engineering and Computer Science*, 22(2), p.1208. doi:https://doi.org/10.11591/ijeecs.v22.i2.pp1208-1215.

Saini, A. (2021). *Decision Tree Algorithm - A Complete Guide*. [online] Analytics Vidhya. Available at: https://www.analyticsvidhya.com/blog/2021/08/decision-tree-algorithm/.

SATPATHY, S. (2020). *SMOTE - A Common Technique to Overcome Class Imbalance Problem*. [online] Analytics Vidhya. Available at: https://www.analyticsvidhya.com/blog/2020/10/overcoming-class-imbalance-using-smote-techniques/.

Sethi, A. (2020). *Categorical Encoding | One Hot Encoding vs Label Encoding*. [online] Analytics Vidhya. Available at: https://www.analyticsvidhya.com/blog/2020/03/one-hot-encoding-vs-label-encoding-using-scikit-learn/.

Thorn, J. (2020). *Random Forest for Feature Importance | Towards Data Science*. [online] Towards Data Science. Available at: https://towardsdatascience.com/random-forest-for-feature-importance-ea90852b8fc5/