COS30049
Computing Technology Innovation Project
Assignment 2

Weather Data Analysis and Modelling

Table of Contents

Introduction

This project involves using machine learning techniques to analyze and model weather data from multiple cities worldwide. The project has several goals:

- Predict weather components such as minimum and maximum temperatures (tmin and tmax).
- Classify weather into categories like 'Rainy', 'Windy', or 'Sunny'.
- Simulate daily energy usage for heating and cooling needs based on weather conditions.
- Use regression models to predict next day temperatures.

The project will walk through data preparation, model training, prediction generation, and evaluation, explaining each step in detail so that users can replicate the entire workflow.

Project Structure

- **CleanedDatasets/**: Contains the cleaned weather datasets for different cities combined.
- **UncleanedDatasets/**: Has the datasets that have to be cleaned.
- **EnhancedML.ipynb:** Contains the entire code from data cleaning to weather prediction.
- **Visualisation.**ipynb: Contains visualisation tools which are helpful for comparing cleaned and uncleaned data.
- **CleaningTokyoandBombay.ipynb:** Code to clean Bombay and Tokyo datasets.
- **README.md**: Documentation for the project (shorter for github).
- **READMe.pdf**: Has detailed steps for every code snippet.

Datasets

In this project, we are using weather datasets from several cities around the world:

- **New York** (UncleanedDatasets/NewYork.csv)
- **Dubai** (UncleanedDatasets /Dubai.csv)
- **Melbourne** (UncleanedDatasets /Melbourne.csv)
- **Sydney** (UncleanedDatasets /Sydney.csv)
- **Tokyo** (UncleanedDatasets /Tokyo.csv)
- **Bombay** (UncleanedDatasets /Bombay.csv)

Each dataset includes daily weather observations with features like:

- tavg: Average daily temperature
- tmin: Minimum daily temperature
- tmax: Maximum daily temperature
- prcp: Precipitation
- wspd: Wind speed
- pres: Atmospheric pressure
- city: Name of the city

Environment Setup

To replicate this project, it is essential to configure your environment correctly.

Prerequisites

- **Python 3.10 or higher**: This is the programming language we use for data analysis and machine learning.
- **pip**: Python's package installer, which helps us install libraries like pandas and scikit-learn.

Clone the Repository

```
git clone https://github.com/SanyaBaweja/CTIP-Assignment2.git
cd Desktop
```

This command will download the project files, including data and scripts, into a folder on your machine.

Create a Virtual Environment

Conda Environment Setup for Weather Data Analysis

To ensure that your environment is correctly set up for your project, follow these steps.

Step 1: Install Conda

Before you proceed, make sure you have Conda installed on your machine.
Download and Install Miniconda or Anaconda

Miniconda Installation

- Go to the Miniconda download page.
- Select the appropriate installer for your operating system (Windows, macOS, or Linux).
- Download the installer for your system and follow the installation instructions.

Anaconda Installation

- Go to the Anaconda download page.
- Select the appropriate installer for your operating system.
- Download the installer for your system and follow the installation instructions.

Step 2: Create a New Conda Environment

Creating a separate environment allows you to isolate the project dependencies, making it easier to manage without affecting global installations.
Run the following command to create a new Conda environment:

```
conda create --name weather_project python=3.10
```

This command:

- Creates a new environment called `weather_project`.
- Installs **Python 3.10** (or the latest compatible version) in this environment.

Step 3: Activate the Environment

Once the environment is created, activate it by running:

```
conda activate weather_project
```

This command:

- Switches you into the `weather_project` environment, so any Python libraries you install or code you run will be isolated to this environment.

Step 4: Install Required Packages

In this step, you will install all the required libraries for your project, including packages for data manipulation, visualization, and machine learning.
Run the following command to install the necessary libraries:

```
conda install pandas seaborn matplotlib scikit-learn joblib numpy
```

This command installs:

- **pandas**: For data manipulation and analysis.
- **seaborn** & **matplotlib**: For data visualization and plotting.
- **scikit-learn**: For machine learning models and algorithms.
- **joblib**: For saving and loading machine learning models.
- **numpy**: For numerical computations.

Step 5: Install Jupyter Notebook

If you plan to run your code interactively or create visual reports, installing **Jupyter Notebook** is highly recommended. It allows you to write and execute code in an interactive environment, ideal for data analysis and visualization.

Run the following command to install Jupyter Notebook:
```
conda install jupyter
```

Once installed, you can start the notebook server by running:
```
jupyter notebook
```

This command will:
- Open a Jupyter Notebook interface in your default web browser.
- Allow you to write, run, and visualize your Python code interactively.

EnhancedML.ipynb

Data Preprocessing

Data preprocessing is a crucial step before training machine learning models. It ensures that the data is clean, consistent, and ready for modeling.

1. Load the Datasets

We begin by loading weather datasets for multiple cities using `pandas`.

```
import pandas as pd

# Load datasets for multiple cities
ny_weather = pd.read_csv('UncleanedDatasets/NewYork.csv', index_col="date")
dubai_weather = pd.read_csv('UncleanedDatasets/Dubai.csv', index_col="date")
melbourne_weather = pd.read_csv('UncleanedDatasets /Melbourne.csv',
index_col="date")
sydney_weather = pd.read_csv('UncleanedDatasets /Sydney.csv', index_col="date")
```

Here’s what this step does:

- `pd.read_csv` loads the CSV files into a `pandas` DataFrame, a table-like data structure.
- The `index_col="date"` argument sets the `date` column as the index for the DataFrame, so every row is indexed by its date.

2. Add 'city' Column and Combine Datasets

Each dataset belongs to a specific city. We add a new column (`city`) to label the rows from each city and then combine all the datasets into one DataFrame.

```
# Add 'city' column to each dataset
ny_weather['city'] = 'New York'
dubai_weather['city'] = 'Dubai'
melbourne_weather['city'] = 'Melbourne'
sydney_weather['city'] = 'Sydney'

# Combine datasets from all cities
```

```
combined_weather_data = pd.concat([ny_weather, dubai_weather, melbourne_weather,
sydney_weather])
```

- `pd.concat()` merges all the datasets into one, stacking them vertically. This results in one DataFrame with all the weather data, with a new `city` column that indicates which rows come from which city.

3. Handle Missing Values

Missing data is common in real-world datasets. We need to handle it carefully to avoid errors in machine learning models.

Calculate Missing Values Percentage

```
null_pct = combined_weather_data.isnull().sum() / combined_weather_data.shape[0]
```

This step calculates the percentage of missing values in each column:

- `isnull().sum()` gives the total number of missing values in each column.
- Dividing by the total number of rows (`shape[0]`) gives the proportion of missing data for each column.

Drop Columns with More than 50% Missing Values

Columns with too many missing values provide little information, so we drop them.

```
valid_columns = combined_weather_data.columns[null_pct < 0.5]
combined_weather_data = combined_weather_data[valid_columns].copy()
```

- This keeps only the columns with less than 50% missing values.
- `.copy()` ensures that we work on a clean copy of the data, preventing changes from affecting the original DataFrame.

Fill Remaining Missing Values with Mean

For the remaining columns, we replace missing values with the column's mean. This method is simple and prevents the loss of too much data.

```
for col in ['tavg', 'tmin', 'tmax', 'prcp', 'wspd', 'wdir', 'pres']:
    combined_weather_data[col].fillna(combined_weather_data[col].mean(),
inplace=True)
```

- `fillna()` replaces missing values.
- We're using the column mean (`combined_weather_data[col].mean()`) to fill in gaps, a standard practice when dealing with numerical data.

4. Remove Outliers

Outliers can distort model training, so we remove them using the Interquartile Range (IQR) method.

```
# Calculate Q1 and Q3
Q1 = combined_weather_data.quantile(0.25)
Q3 = combined_weather_data.quantile(0.75)
IQR = Q3 - Q1
```

```
# Filter out outliers
combined_weather_data = combined_weather_data[~((combined_weather_data < (Q1 - 1.5
* IQR)) | (combined_weather_data > (Q3 + 1.5 * IQR))).any(axis=1)]
```

- The Interquartile Range (IQR) is the difference between the 75th percentile (Q3) and the 25th percentile (Q1).
- We remove rows that have values beyond 1.5 times the IQR from Q1 or Q3, as they are considered outliers.

5. Feature Engineering

We create new columns to provide additional useful information to our models.

Create New Features: Temperature Range and Wind Chill

- **Temperature Range**: The difference between `tmax` (maximum temperature) and `tmin` (minimum temperature).
- **Wind Chill**: A rough estimate of how cold it feels when factoring in wind speed, calculated as `tavg - (wspd * 0.7)`.

```
combined_weather_data['temp_range'] = combined_weather_data['tmax'] -
combined_weather_data['tmin']
combined_weather_data['wind_chill'] = combined_weather_data['tavg'] -
(combined_weather_data['wspd'] * 0.7)
```

These engineered features help capture additional patterns in the data.

6. Save the Cleaned Data

Finally, we save the cleaned and processed dataset to a CSV file for further use.

```
combined_weather_data.to_csv('CleanedDatasets/NewCombined.csv', index=True)
```

This makes it easy to load and reuse the processed data later in the project.

Model Training

With the cleaned data ready, we now train machine learning models to predict temperatures and classify weather.

1. Prepare Data for Training

One-Hot Encode the 'city' Column

Machine learning models can’t work directly with text data like city names. We convert the `city` column into numerical form using **one-hot encoding**.

```
combined_weather_data = pd.get_dummies(combined_weather_data, columns=['city'],
drop_first=False)
```

- `pd.get_dummies()` creates separate columns for each city, with `0` or `1` values indicating whether the city applies to each row.
- This ensures that the model can understand the `city` column numerically.

Define Features (X) and Target Variables (y)

Now, we separate the data into:

- **Features (`X`)**: The input variables used for predictions (excluding the target variables `tmax` and `tmin`).
- **Target (`y`)**: The variables we want to predict (`tmax` and `tmin`).

```
X = combined_weather_data.drop(columns=['tmax', 'tmin'])
y = combined_weather_data[['tmax', 'tmin']]
```

Split Data into Training and Testing Sets

We split the data into two parts: 80% for training the model and 20% for testing it.

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)
```

- **Training data**: The model learns patterns from this data.
- **Testing data**: Used to evaluate the model's performance on unseen data.
- The `random_state=42` ensures that the data split is reproducible (you'll get the same split each time).

2. Train Regression Models

Now, we train machine learning models to predict `tmin` and `tmax` using our prepared data.

Linear Regression

```
from sklearn.linear_model import LinearRegression

# Initialize and train the Linear Regression model
linear_model = LinearRegression()
linear_model.fit(X_train, y_train)
```

- **Linear Regression**: A simple model that predicts target values by finding the best-fit straight line through the data.
- `.fit()` trains the model on the training data (`X_train`, `y_train`).

Random Forest Regressor

```
from sklearn.ensemble import RandomForestRegressor

# Initialize and train the Random Forest Regressor
rf_model = RandomForestRegressor(n_estimators=100, random_state=42)
rf_model.fit(X_train, y_train)
```

- **Random Forest**: An ensemble model that builds multiple decision trees and averages their predictions. This improves accuracy and reduces overfitting.
- `n_estimators=100` specifies that the model will build 100 decision trees.

Gradient Boosting Regressor

For more complex relationships, we also train **Gradient Boosting** models for both `tmin` and `tmax`.

```
from sklearn.ensemble import GradientBoostingRegressor
```

```
gbr_model_tmin = GradientBoostingRegressor(n_estimators=100, random_state=42)
gbr_model_tmax = GradientBoostingRegressor(n_estimators=100, random_state=42)

gbr_model_tmin.fit(X_train, y_train['tmin'])
gbr_model_tmax.fit(X_train, y_train['tmax'])
```

- **Gradient Boosting**: A model that builds decision trees sequentially, where each tree corrects the errors of the previous one. This often results in highly accurate models.
- We train separate models for `tmin` and `tmax` since these are separate targets.

Lasso and Ridge Regressor

To experiment with more models and to get the least error for model selection we train 2 more models which are Lasso and Ridge Regressor.

```
from sklearn.linear_model import Lasso, Ridge
from sklearn.metrics import mean_squared_error, r2_score

# Separate the target variables for tmin and tmax
y_train_tmin = y_train['tmin']
y_train_tmax = y_train['tmax']
y_test_tmin = y_test['tmin']
y_test_tmax = y_test['tmax']

# Initialize Lasso and Ridge models for tmin and tmax
lasso_model_tmin = Lasso(alpha=1.0, random_state=42)
lasso_model_tmax = Lasso(alpha=1.0, random_state=42)

ridge_model_tmin = Ridge(alpha=1.0, random_state=42)
ridge_model_tmax = Ridge(alpha=1.0, random_state=42)

# ---- Lasso Regression ----
# Train the Lasso models on the respective targets
lasso_model_tmin.fit(X_train, y_train_tmin)
lasso_model_tmax.fit(X_train, y_train_tmax)

# ---- Ridge Regression ----
# Train the Ridge models on the respective targets
ridge_model_tmin.fit(X_train, y_train_tmin)
ridge_model_tmax.fit(X_train, y_train_tmax)
```

3. Evaluate Models

We use **Mean Squared Error (MSE)** and **R² Score** to evaluate the models:

- **MSE** measures how close the predicted values are to the actual values (lower is better).
- **R²** measures how well the model explains the variability in the target values (higher is better, with a maximum of 1.0).

Linear Regression Evaluation

```
from sklearn.metrics import mean_squared_error, r2_score

# Make predictions and evaluate the model
y_pred_linear = linear_model.predict(X_test)
mse_linear_tmin = mean_squared_error(y_test['tmin'], y_pred_linear[:, 1])
mse_linear_tmax = mean_squared_error(y_test['tmax'], y_pred_linear[:, 0])
r2_linear_tmin = r2_score(y_test['tmin'], y_pred_linear[:, 1])
r2_linear_tmax = r2_score(y_test['tmax'], y_pred_linear[:, 0])
```

Random Forest Regressor Evaluation

```
# Make predictions and evaluate the model
y_pred_rf = rf_model.predict(X_test)
mse_rf_tmin = mean_squared_error(y_test['tmin'], y_pred_rf[:, 1])
mse_rf_tmax = mean_squared_error(y_test['tmax'], y_pred_rf[:, 0])
r2_rf_tmin = r2_score(y_test['tmin'], y_pred_rf[:, 1])
r2_rf_tmax = r2_score(y_test['tmax'], y_pred_rf[:, 0])
```

Gradient Boosting Regressor Evaluation

```
# Make predictions and evaluate the model
y_pred_gbr_tmin = gbr_model_tmin.predict(X_test)
y_pred_gbr_tmax = gbr_model_tmax.predict(X_test)
mse_gbr_tmin = mean_squared_error(y_test['tmin'], y_pred_gbr_tmin)
mse_gbr_tmax = mean_squared_error(y_test['tmax'], y_pred_gbr_tmax)
r2_gbr_tmin = r2_score(y_test['tmin'], y_pred_gbr_tmin)
r2_gbr_tmax = r2_score(y_test['tmax'], y_pred_gbr_tmax)
```

Lasso and Ridge Regressor

```
# Train the Lasso models on the respective targets
lasso_model_tmin.fit(X_train, y_train_tmin)
lasso_model_tmax.fit(X_train, y_train_tmax)

# Make predictions on the test data
y_pred_lasso_tmin = lasso_model_tmin.predict(X_test)
y_pred_lasso_tmax = lasso_model_tmax.predict(X_test)

# Calculate MSE and R² for Lasso Regression
mse_lasso_tmin = mean_squared_error(y_test_tmin, y_pred_lasso_tmin)
mse_lasso_tmax = mean_squared_error(y_test_tmax, y_pred_lasso_tmax)

r2_lasso_tmin = r2_score(y_test_tmin, y_pred_lasso_tmin)
r2_lasso_tmax = r2_score(y_test_tmax, y_pred_lasso_tmax)
```

Models are compared using MSE and R^2 Method and we choose the model with the least error. For predicting Tmin and Tmax, we use Gradient Boost Regressor whereas to classify the weather into those with rainy, windy, or sunny, we use random forest classifier. To add complexity, we added another Random Forest Regressor to predict energy consumption for the day.

Weather Classification

In this section, we classify the weather into categories like 'Rainy', 'Windy', or 'Sunny'.

1. Classify Weather Types

We define a function to classify weather based on certain conditions:

```
def classify_weather(row):
    if row['prcp'] > 0.1:
        return 'Rainy'
    elif row['wspd'] > 15:  # Example threshold for a windy day
        return 'Windy'
    else:
        return 'Sunny'

# Apply the function to create a new column 'weather_type'
```

```
combined_weather_data['weather_type'] =
combined_weather_data.apply(classify_weather, axis=1)
```

This function categorizes weather based on precipitation (`prcp`) and wind speed (`wspd`).

2. Train Classification Model

We train a Random Forest Classifier to predict the weather type.

```
from sklearn.ensemble import RandomForestClassifier

# Features and target
X_classification = combined_weather_data.drop(columns=['weather_type'])
y_classification = combined_weather_data['weather_type']

# Split data into training and testing sets
X_train_cls, X_test_cls, y_train_cls, y_test_cls =
train_test_split(X_classification, y_classification, test_size=0.2,
random_state=42)

# Train the classifier
rf_classifier = RandomForestClassifier(n_estimators=100, random_state=42)
rf_classifier.fit(X_train_cls, y_train_cls)
```

3. Evaluate Classification Model

We evaluate the classifier using accuracy and other classification metrics like precision, recall, and F1-score.

```
from sklearn.metrics import classification_report, accuracy_score

y_pred_cls = rf_classifier.predict(X_test_cls)
print(f"Accuracy: {accuracy_score(y_test_cls, y_pred_cls)}")
print(classification_report(y_test_cls, y_pred_cls))
```

Energy Usage Simulation

Next, we simulate energy usage based on weather conditions, using a function to estimate energy consumption for heating or cooling.

1. Simulate Energy Usage

We define a function that calculates energy consumption based on temperature and wind speed.

```
def simulate_energy_usage(row):
    tavg = row['tavg']
    wspd = row['wspd']
    prcp = row['prcp']

    # Higher energy consumption on very cold or hot days
    if tavg < 10:  # Cold day, more heating needed
        return 100 + (10 - tavg) * 5 + wspd * 2
    elif tavg > 25:  # Hot day, more cooling needed
        return 100 + (tavg - 25) * 4 + wspd * 1.5
    else:  # Moderate temperature, less energy usage
        return 50 + prcp * 0.5

# Apply the function to create a new column 'energy_usage'
combined_weather_data['energy_usage'] =
combined_weather_data.apply(simulate_energy_usage, axis=1)
```

This function returns a simulated energy usage value based on weather conditions.

2. Train Energy Usage Model

We train a Random Forest Regressor to predict energy usage.

```
# Define features and target for energy usage prediction
X_energy = combined_weather_data[['tavg', 'tmin', 'tmax', 'prcp', 'wspd']]
y_energy = combined_weather_data['energy_usage']

# Split data into training and testing sets
X_train_energy, X_test_energy, y_train_energy, y_test_energy =
train_test_split(X_energy, y_energy, test_size=0.2, random_state=42)

# Train the Random Forest Regressor
energy_model = RandomForestRegressor(n_estimators=100, random_state=42)
energy_model.fit(X_train_energy, y_train_energy)
```

3. Evaluate Energy Usage Model

We evaluate the model by calculating **Mean Squared Error (MSE)** and **R² score**.

```
y_pred_energy = energy_model.predict(X_test_energy)
mse_energy = mean_squared_error(y_test_energy, y_pred_energy)
r2_energy = r2_score(y_test_energy, y_pred_energy)

print(f"Mean Squared Error: {mse_energy}")
print(f"R² Score: {r2_energy}")
```

Extending the Model

We can extend the project by adding more datasets to make our models more robust.

1. Load Additional Datasets

For example, we can add weather data from Tokyo and Bombay:

```
tokyo_weather = pd.read_csv('UncleanedDatasets/Tokyo.csv', index_col="date")
bombay_weather = pd.read_csv('UncleanedDatasets/Bombay.csv', index_col="date")

tokyo_weather['city'] = 'Tokyo'
bombay_weather['city'] = 'Bombay'

# Combine with the existing data
additional_datasets = pd.concat([tokyo_weather, bombay_weather])
```

2. Retrain Models

After adding new data, we need to repeat the preprocessing and retrain the models to improve their accuracy. The Random Forest and Gradient Boost are retrained using the additional datasets and their accuracy is again measured. This will significantly reduce the MSE making the model even better for prediction.

Predicting Next Day's Temperatures

One of the project goals is to predict tomorrow's `tmin` and `tmax`. To do this, we shift the temperature columns by one day.

```
 # Shift the 'tmin' and 'tmax' columns
combined_weather_data['tmin_next_day'] = combined_weather_data['tmin'].shift(-1)
combined_weather_data['tmax_next_day'] = combined_weather_data['tmax'].shift(-1)

# Drop rows with missing values in the next day columns
combined_weather_data.dropna(subset=['tmin_next_day', 'tmax_next_day'],
inplace=True)
```

Now we can use the features from today (`tavg`, `prcp`, etc.) to predict tomorrow's minimum and maximum temperatures.

Results

The final comparison table for actual vs predicted values is generated using Gradient Booster Regressor, and the results are saved to CSV for further analysis.

```
python
Copy code
results = pd.DataFrame({
    'Actual Tmin': y_test['tmin_next_day'].values,
    'Predicted Tmin (RF)': y_pred_rf[:, 0],
    'Predicted Tmax (RF)': y_pred_rf[:, 1]
})
```

For each model, we display key metrics (MSE, R², accuracy) and visualizations like:

- **Correlation Heatmaps**: Show relationships between features.
- **Actual vs. Predicted Values**: To see how well the model's predictions align with reality.

Visualisation.ipynb

Running the Notebook

1. **Open the Jupyter Notebook**: Make sure your environment is activated. Open the `Visualisation.ipynb` notebook.
2. **Load Data**: The notebook loads two datasets:
    - **`Uncleaned Data.csv`**: The raw weather data before cleaning.
    - **`NewCombined.csv`**: The cleaned weather data after handling missing values, removing outliers, and feature engineering.
3. **Generate Visualizations**:
    - The code generates **histograms** and **scatter plots** to visualize the weather data.
    - You can compare how data cleaning affects the distribution of values and relationships between variables.

Explanation of Visualizations

1. **Distribution of Weather Variables (Before and After Cleaning)**:
    - **Histograms** show the frequency of values for different weather variables (`tavg`, `tmin`, `tmax`, `prcp`, `wspd`, `pres`) both before and after the data has been cleaned.

    - This comparison reveals the impact of handling missing values, outliers, and normalizing data ranges.
2. **Scatter Plots (Tmin vs. Tmax)**:
    - **Scatter plots** show the relationship between minimum (`tmin`) and maximum (`tmax`) temperatures.
    - The color of the points represents wind speed (`wspd`), offering insights into whether wind affects temperature relationships.
    - The comparison between cleaned and uncleaned data highlights how cleaning the data improves the visual clarity of patterns in the dataset.

Code Overview

1. **Loading Datasets**:

```
data_before_cleaning = pd.read_csv("Uncleaned Data.csv", index_col="date")
data_after_cleaning = pd.read_csv("NewCombined.csv", index_col="date")
```

2. **Histograms for Data Distribution**: The notebook first displays histograms for both the uncleaned and cleaned datasets:

```
data_before_cleaning[['tavg', 'tmin', 'tmax', 'prcp', 'wspd',
'pres']].hist(bins=20, figsize=(10, 8), layout=(2, 3))
plt.suptitle('Distribution of Weather Variables')
plt.show()

data_after_cleaning[['tavg', 'tmin', 'tmax', 'prcp', 'wspd',
'pres']].hist(bins=20, figsize=(10, 8), layout=(2, 3))
plt.suptitle('Distribution of Weather Variables')
plt.show()
```

3. **Scatter Plots to Explore Relationships**: Scatter plots are used to show the relationship between `tmin` and `tmax`, colored by wind speed (`wspd`):

```
python
Copy code
sns.scatterplot(x='tmin', y='tmax', data=data_before_cleaning, hue='wspd',
palette='coolwarm')
plt.title('Scatter Plot of Tmin vs Tmax (Colored by Windspeed)')
plt.xlabel('Min Temperature (°C)')
plt.ylabel('Max Temperature (°C)')
plt.show()
```

The same plot is then generated for the cleaned data, allowing you to compare the before and after state of the dataset.

Conclusion

This project demonstrates how to clean and preprocess weather data, perform feature engineering, and build machine learning models for both regression and classification tasks. By extending the project with additional datasets, we improve the generalizability of our models.