Weather Prediction

Authors: Feifan Jiang, Liza Kostina, Judy Wu, Daniel Zou

Date: November 2024

Objective:

The objective of this project is to accurately predict the minimum, average, and maximum daily temperatures (in Fahrenheit) for each of 20 selected cities in the United States, over five future days, across nine consecutive days. These predictions aim to minimize the mean squared error and are submitted daily at noon, starting November 26, 2024, and concluding on December 4, 2024. A total of 2,700 temperature predictions will be made, considering 3 temperature variables, 20 cities, 5 prediction days, and 9 submission days. The project focuses on implementing robust forecasting techniques to achieve precision across diverse geographic locations, from Anchorage to Miami, and varying climates.

Project timeline

- Monday, Nov 25: Model finalized, code committed to Github, Docker image uploaded to Dockerhub
- Tuesday, Nov 26: Begin making daily predictions, due at noon daily
- Tuesday, Dec 3: Presentations
- Monday, Dec 9: Final day making predictions; report due

Data

Data Sources

To utilize the strengths of publicly available datasets, we sourced data from NOAA and an additional OpenWeather dataset, leveraging them as follows:

- 1. **NOAA Dataset**
  The NOAA dataset provided us with historical weather data dating back to January 1, 1948. Specifically, we utilized the **Global Historical Climatology Network - Daily (GHCN-D)** dataset, which integrates daily climate observations from approximately 30 sources. This comprehensive and consistent dataset includes measurements which we further used in our models:

  - Minimum, maximum, and average temperatures,
  - Precipitation,
  - Snowfall,
  - Snow depth.

  The NOAA dataset served as the backbone of our analysis, offering a long-term historical perspective essential for training robust prediction models.

- 1. **OpenWeather Dataset**
  To complement the historical data provided by NOAA, we utilized the OpenWeather History API for recent weather data. This dataset offers hourly historical weather data dating back to January 1, 1979, and includes a wide range of weather parameters such as:

  - Minimum, maximum, and average temperatures,

  - Humidity,
  - Precipitation details (rainfall and snowfall),
  - Wind speed and direction.

For our analysis, we focused on data from 20 locations, specified by their geographic coordinates (longitude and latitude), covering the period from November 12, 2024, to the day of making predictions. The data was retrieved using Python's `requests` module, downloaded in weekly chunks to stay within the API limits. The raw data was provided in JSON format, requiring additional processing for integration with our models.

The OpenWeather dataset offered high-resolution, recent weather data, providing a contemporary perspective to complement the long-term historical insights from NOAA.

Data Preprocessing

Preprocessing both datasets was a critical step in preparing the historical weather data for analysis and model training.

- **NOAA Dataset** For NOAA dataset the main steps of the preprocessing included:

1. **Format Conversion**
   - The raw data, provided in `.dly` format, was converted into a tabular format to enable easier analysis and integration with our workflow.
2. **Data Cleaning**
   - **Invalid Dates:** Removed entries with invalid dates to ensure consistency.
   - **Missing Average Temperatures:** Filled missing average temperature values by taking the average of the maximum and minimum temperatures for the same day.
   - **Snowfall and Precipitation:** Replaced missing values for snowfall and precipitation with zeros, assuming no snowfall or rainfall occurred.
   - **Invalid Temperature Values:** Fixed anomalous temperature readings (e.g., values below -1000) by replacing them with the previous day's valid temperature.
3. **Unit Conversion**
   - Converted all temperature values from Celsius to Fahrenheit to align with standard U.S. weather reporting conventions.
4. **Feature Selection**
   - Retained only the core features, including temperature (minimum, maximum, and average), precipitation and snowfall for subsequent model training.

These preprocessing steps ensured the NOAA dataset was clean, consistent, and ready for effective integration into the predictive models.

- **OpenWeather Dataset**

The OpenWeather dataset also underwent several preprocessing steps to prepare it for integration with the NOAA data and subsequent model training. The main steps included:

1. **Format Conversion**
   - Raw data retrieved in `.JSON` format was converted into `.csv` format to facilitate analysis and storage.
2. **Data Aggregation**
   - **Time Conversion:** The UTC timestamps in the dataset were converted to the local time of each city to align with region-specific weather patterns.
   - **Feature Extraction:** Key features, including temperature (minimum, maximum, and average), precipitation, and snowfall records, were extracted for analysis.
   - **Daily Aggregation:** The hourly data was aggregated into daily data by:
     - Calculating the daily average, minimum, and maximum temperatures.
     - Summing total precipitation and snowfall for each day.
3. **Dataset Integration**
   - The processed OpenWeather data, covering the period from November 12 to November 25, 2024, was merged with the processed NOAA data to create a unified dataset for model training.

These preprocessing steps ensured consistency between the two datasets and prepared the data for effective model development.

The code used for data downloading and preprocessing is provided in the Appendix section of this report for reference. Additionally, the original code can be found in the analysis folder, and the raw and processed data are stored in the data folder of our GitHub repository.

Regression-Based Weather Forecasting

After obtaining and preprocessing the data, we framed weather forecasting as a regression problem. By using past weather data as features and future temperature data as labels, we were able to approach this task systematically within a supervised learning framework.

The primary objective of the project was to minimize the Mean Squared Error between the predicted and actual temperature values. Thus, we continued our analysis with preparing regression dataset in the following way:

Regression Dataset Structure

1. **Features**:
   - A flattened array of the previous 30 days' data, including:
     - Minimum temperature (TMIN)
     - Average temperature (TAVG)
     - Maximum temperature (TMAX)
     - Snowfall (SNOW)
     - Precipitation (PRCP)
2. **Labels**:

- A flattened array of the next 5 days' data, focusing on:
  - Minimum temperature (TMIN)
  - Average temperature (TAVG)
  - Maximum temperature (TMAX)

Data Range To ensure robust model evaluation and fair predictions, the data range was divided as follows:
- **Model Training**:
- Models were trained on data excluding the evaluation days, with a buffer to prevent any data leakage.
- **Model Evaluation**:
- To simulate real-world test conditions, evaluation was conducted on data from November 25 to December 10 for each of the past five years.
- For each evaluation day, predictions were made for the subsequent 5 days.
- This approach ensured that the evaluation set closely resembled the actual test scenario in timing and structure.
- **Final Prediction**:
- After selecting the best-performing model, it was trained on the full dataset from January 1, 2014, to October 31, 2024, to generate the final predictions.

This structured approach to training, evaluation, and prediction ensured a clear separation between training and testing phases while leveraging historical and recent data to create a robust predictive framework.

Prediction Models

To address the weather forecasting challenge, we trained a series of predictors using diverse methodologies:

- **Previous Day Predictor**
- **Average Last Week Predictor**
- **Linear Regression**
- **Ridge Regression**
- **LASSO**
- **Random Forest**

Additionally, we developed a **Weighted Average Predictor**, which combines the outputs of the above predictors using optimized weights to improve overall accuracy.

Baseline Predictors To establish a benchmark for comparison, we implemented two simple baseline predictors:

1. **Previous Day Predictor**
   - **Approach**: Uses the last recorded day's weather data to predict the next 5 days.
   - **Mechanism**: Repeats today's values (TMIN, TAVG, TMAX) for the subsequent 5 days.
   - **Strengths**:

    - Simplicity: The method requires no training or parameter tuning, making it extremely straightforward and computationally efficient.
    - Quick Deployment: Since it relies solely on the most recent data, it can be easily implemented and does not require historical data beyond the last recorded day.
  - **Limitations**:
    - Assumes static weather patterns, making it less effective during rapid weather changes.

2. **Average Last Week Predictor**
  - **Approach**: Averages the last 7 days of weather data to generate predictions.
  - **Mechanism**: Calculates the mean TMIN, TAVG, and TMAX over the past week and repeats these averages for the next 5 days.
  - **Strengths**:
    - More robust to daily fluctuations compared to the Previous Day Predictor.
  - **Limitations**:
    - Assumes weekly averages accurately reflect future trends, which might not always hold especially during abrupt weather changes.

While both baseline predictors are straightforward to implement, they come with inherent limitations, particularly in their assumptions about weather patterns. These predictors served as useful references, enabling us to evaluate the performance of more sophisticated models.

Regression-Based Predictors

After implementing the baseline predictors, we advanced to regression-based models to better capture complex weather patterns. For these models, the features consisted of the last 30 days of weather data, specifically TMIN, TAVG, TMAX, SNOW, and PRCP (precipitation), which were flattened into a single feature vector of size 30 x 5 = 150.

Linear Regression

- **Strength**:
  - Simple, interpretable, and computationally efficient.
  - Provides clear coefficients that explain the relationship between features and the target variable.
- **Limitation**:
  - Assumes a linear relationship between features and the target, which limits its ability to model more complex weather patterns or interactions between variables.

Ridge Regression (5-fold with RidgeCV)

- **Approach**:
  - Ridge regression was applied with a regularization parameter (α) tuned using RidgeCV via 5-fold cross-validation.
- **Strength**:

  - Handles multicollinearity effectively, reducing the potential for overfitting by shrinking the coefficients.

- **Limitation**:
  - Despite regularization, it still assumes linear relationships between the features and target, which may not fully capture nonlinear weather patterns.

LASSO Regression (5-fold with LassoCV)

- **Approach**:
  - LASSO regression was also tuned using LassoCV with 5-fold cross-validation.

- **Strength**:
  - Particularly useful for feature selection, as it tends to shrink coefficients of less important features to zero.

- **Limitation**:
  - LASSO may drop important features if they are highly correlated with others, as it retains only one feature from a correlated group, which could result in the loss of valuable predictors.

These regression-based models introduced more complexity compared to the baseline predictors, and their strengths and limitations reflect the trade-off between interpretability and the ability to capture complex relationships in the data.

Random Forest and Weighted Average Predictor

After exploring linear and regularized regression models, we turned to more complex machine learning models, such as Random Forest, to capture nonlinear relationships in the data.

Random Forest (GridSearchCV with 5-fold Cross-Validation)

- **Features**: Same as Linear Regression (150 features, including TMIN, TAVG, TMAX, SNOW, and PRCP for the past 30 days).

- **Strength**:
  - Random Forests excel at handling complex patterns and interactions between features, which are common in weather data.
  - Provides feature importance, allowing us to identify which variables contribute most to the model's predictions.

- **Limitation**:
  - Computationally intensive, especially as the number of trees and data size increases. Training and prediction times may be longer compared to simpler models.

Weighted Average Predictor (Custom Cross-Validation over Weights)

- **Approach**:
  - The Weighted Average Predictor combines predictions from multiple models by calculating a weighted average of their outputs.
  - Weights for each model's prediction are generated randomly using a Dirichlet distribution, and each weight combination is evaluated based on the Mean Squared Error (MSE). The best-performing weights are then selected.
- **Features and Targets**:
  - Features and targets are the same as those used in the individual models, with weights applied to the predictions of each model.
- **Strength**:
  - By combining predictions from multiple models, this approach improves accuracy and captures complementary patterns that individual models may miss.
- **Limitation**:
  - The quality of the model depends heavily on the choice of weights, which could lead to suboptimal performance if not carefully optimized.

These models represent a natural progression from simpler linear models to more sophisticated ensemble methods, with the goal of improving prediction accuracy by capturing complex relationships in the data. The Random Forest provides a powerful tool for identifying complex patterns, while the Weighted Average Predictor serves as a way to combine multiple model outputs for enhanced performance.

Conclusion

After training and evaluating the Weighted Average Predictor, we determined the optimal weights to assign to each model in the ensemble. Our final configuration gave full weight to the Ridge Regression model, with a weight of 1.0, and assigned zero weight to the other models (Lasso, Random Forest, and Baselines). This weight configuration was chosen to minimize the Mean Squared Error (MSE), which resulted in an optimal MSE of 49.97.

The historical NOAA data, supplemented with current OpenWeather data, served as the foundation for our predictions. For evaluation, we used the period from Nov 25 to Dec 10 of the past five years, ensuring that the evaluation set was representative of real-world forecasting conditions. Our weighted model, which included a variety of predictors such as Linear Regression and Random Forest, provided a robust forecasting tool. The final result was a Ridge Regression model trained on temperature, snowfall, and precipitation from the past 30 days.

To ensure the accessibility and reproducibility of our results, we packaged the entire workflow into a Docker image. This step allows anyone interested to easily run our models with minimal setup, ensuring that the results can be verified and applied in different environments. This approach highlights the importance of not only developing accurate models but also making them accessible and reproducible for future use.

Further work

While our final Ridge Regression model demonstrated strong performance for the first two days of forecasting, we observed a notable decline in accuracy for days three, four, and five. This suggests that incorporating additional methodologies could enhance the model's robustness over longer prediction horizons.

1. **Combination of Predictors**
   - A hybrid approach that combines the Previous Day Predictor and the Historical Average Predictor could be effective in this longitudinal setting.
   - Historical averages can serve as a stable baseline, especially for capturing long-term patterns, while the Previous Day Predictor provides adaptability to short-term fluctuations.
   - Exploring a weighted combination or dynamic switching mechanism between these predictors based on the prediction horizon might yield improved results.
2. **Advanced Ensemble Methods**
   - Expanding the ensemble approach by incorporating more sophisticated models such as Gradient Boosting Machines (e.g., XGBoost, LightGBM) or Neural Networks could capture more complex patterns in the data.
   - Additionally, using time-series specific models like ARIMA, SARIMA, or LSTM networks could leverage temporal dependencies explicitly.
3. **Feature Engineering and Data Augmentation**
   - Including additional meteorological variables, such as wind direction, atmospheric pressure, and cloud cover, may provide the model with richer contextual information.
   - Data augmentation techniques, such as simulating plausible missing data scenarios, could improve the model's robustness to real-world inconsistencies.
4. **Incorporation of Spatial Information**
   - The current model assumes each location is independent, but incorporating spatial relationships between nearby stations (e.g., using spatial interpolation or graph-based methods) might improve predictions.
   - Models leveraging Geographic Information Systems (GIS) or spatial regression techniques could be explored.
5. **Evaluation of Long-Term Trends**
   - Extend the evaluation period beyond five days to assess the model's capacity to generalize over longer horizons.
   - Investigate trends in model performance across different seasons, as weather patterns can vary significantly by time of year.
6. **Integration with Real-Time Data Streams**
   - Real-time weather updates from APIs like OpenWeather could be dynamically fed into the model for continuous learning and prediction. This would allow the model to adapt to changing conditions.
7. **Optimization of Computational Efficiency**
   - Given the computational demands of ensemble predictors and Random Forests, optimizing model training and inference time is critical for scalability.

- Techniques like model distillation or pruning could make the system more efficient while maintaining accuracy.

By addressing these areas, the forecasting system can become more robust, adaptive, and efficient, making it better suited for both short-term and long-term predictions across varying weather conditions.

Appendix

Code

```
# Downloading data for 20 cities using NOAA dataset

station_code_dict = {
    "PANC": "USW00026451", # Anchorage
    "KBOI": "USW00024131", # Boise
    "KORD": "USW00094846", # Chicago
    "KDEN": "USW00003017", # Denver
    "KDTW": "USW00094847", # Detroit
    "PHNL": "USW00022521", # Honolulu
    "KIAH": "USW00012960", # Houston
    "KMIA": "USW00012839", # Miami
    "KMSP": "USW00014922", # Minneapolis
    "KOKC": "USW00013967", # Oklahoma City
    "KBNA": "USW00013897", # Nashville
    "KJFK": "USW00094789", # New York
    "KPHX": "USW00023183", # Phoenix
    "KPWM": "USW00014764", # Portland ME
    "KPDX": "USW00024229", # Portland OR
    "KSLC": "USW00024127", # Salt Lake City
    "KSAN": "USW00023188", # San Diego
    "KSFO": "USW00023234", # San Francisco
    "KSEA": "USW00024233", # Seattle
    "KDCA": "USW00013743", # Washington DC
}

data_path_url = "https://www.ncei.noaa.gov/pub/data/ghcn/daily/all/"

# Directory to save downloaded files
original_noaa_cache = "data/original"

# Ensure the directory exists
os.makedirs(original_noaa_cache, exist_ok=True)

# URL to download data from
data_path_url = "https://www.ncei.noaa.gov/pub/data/ghcn/daily/all/"

# Setup logging
logging.basicConfig(level=logging.INFO)

# Loop through the station codes and download the corresponding files
for station_code, file_name in station_code_dict.items():
    url = f"{data_path_url}{file_name}.dly"
    try:
        # Download the file and save it
        urllib.request.urlretrieve(url, os.path.join(original_noaa_cache, f"{station_code}.dly"))
        logging.info(f"Successfully scraped data for: {station_code}")
    except Exception as e:
        logging.error(f"Failed to download data for {station_code}: {e}")
```

```

# Converting NOAA dataset in .csv format

# Define the column names and their respective column positions
columns = ['ID', 'YEAR', 'MONTH', 'ELEMENT'] + [f'VALUE{i}' for i in range(1, 32)] + [f'MFLAG{i}' for i

# Define the fixed column positions for each variable
column_positions = [
    (0, 11),  # ID (1-11)
    (11, 15),  # YEAR (12-15)
    (15, 17),  # MONTH (16-17)
    (17, 21)  # ELEMENT (18-21)
] + [(i*5+21, i*5+26) for i in range(31)] * 3  # VALUE1 to VALUE31, MFLAG1 to MFLAG31, QFLAG1 to QFLAG3


# Function to parse a single line of the file
def parse_line(line):
    data = {}

    # Extract the values for each column based on their positions
    data['ID'] = line[0:11].strip()
    data['YEAR'] = int(line[11:15].strip())
    data['MONTH'] = int(line[15:17].strip())
    data['ELEMENT'] = line[17:21].strip()

    # Extract VALUE, MFLAG, QFLAG, SFLAG columns
    for i in range(31):
        start = 21 + i * 8
        value_str = line[start:start + 5].strip()
        try:
            data[f'VALUE{i + 1}'] = int(value_str) if value_str else None
        except ValueError:
            data[f'VALUE{i + 1}'] = None
        data[f'MFLAG{i + 1}'] = line[start + 5:start + 6].strip()
        data[f'QFLAG{i + 1}'] = line[start + 6:start + 7].strip()
        data[f'SFLAG{i + 1}'] = line[start + 7:start + 8].strip()

    return data


def convert_dly_to_dataframe(input_dir, output_dir, parse_line, file_extension="csv"):
    """
    Converts all .dly files from the input directory to DataFrames and saves them in the output directo
    """
    os.makedirs(output_dir, exist_ok=True)  # Create output directory if it doesn't exist
    for filename in os.listdir(input_dir):
        if filename.endswith(".dly"):
            file_path = os.path.join(input_dir, filename)

            # Read and parse the file into a list of dictionaries
            records = []
            with open(file_path, 'r') as f:
                for line in f:
                    record = parse_line(line)
                    records.append(record)

            # Convert to DataFrame
            df = pd.DataFrame(records)

            # Save the DataFrame
            output_file_path = os.path.join(output_dir, f"{os.path.splitext(filename)[0]}.{file_extensi
            if file_extension == "csv":
                df.to_csv(output_file_path, index=False)
            elif file_extension == "parquet":
                df.to_parquet(output_file_path, index=False)

            print(f"Saved {output_file_path}")


input_dir = 'data/original'
output_dir = 'data/processed'

convert_dly_to_dataframe(input_dir, output_dir, parse_line)
```

```
# NOAA data preprocessing

def process_weather_data(data_file, hourly_data_file):
    data = pd.read_csv(data_file)
    hourly_data = pd.read_csv(hourly_data_file)

    # Convert the 'Datetime' column in 'hourly_data' to a datetime object and extract Year, Month, Day
    hourly_data['Datetime'] = pd.to_datetime(hourly_data[['Year', 'Month', 'Day']])
    hourly_data['Year'] = hourly_data['Datetime'].dt.year
    hourly_data['Month'] = hourly_data['Datetime'].dt.month
    hourly_data['Day'] = hourly_data['Datetime'].dt.day

    data.loc[
        (data['YEAR'] == 2024) &
        ((data['MONTH'] == 11) & (data['DAY'] >= 12)),
        ['TAVG', 'TMIN', 'TMAX', 'PRCP']
    ] = None  # Replace values of these columns to NA for this date range

    hourly_aggregated = hourly_data.groupby(['Year', 'Month', 'Day']).agg(
        TAVG=('Temperature (F)', 'mean'),  # Average Temperature to TAVG
        TMIN=('Temp Min (F)', 'min'),  # Min Temp to TMIN
        TMAX=('Temp Max (F)', 'max'),  # Max Temp to TMAX
        PRCP=('Rain (1h)', 'sum'),  # Pressure to PRCP
        SNOW=('Snow (1h)', 'sum')
    ).reset_index()

    # Rename the columns in hourly_aggregated to avoid conflict during merge
    hourly_aggregated = hourly_aggregated.rename(columns={
        'TAVG': 'TAVG_new',
        'TMIN': 'TMIN_new',
        'TMAX': 'TMAX_new',
        'PRCP': 'PRCP_new',
        'SNOW': 'SNOW_new'
    })

    # Merge the aggregated hourly data with 'data', replacing NA values with hourly aggregated values
    data = pd.merge(data, hourly_aggregated, how='outer', left_on=['YEAR', 'MONTH', 'DAY'],
                    right_on=['Year', 'Month', 'Day'])

    # Replace the NA values in the columns with the values from the aggregated hourly
    data['YEAR'] = data['YEAR'].combine_first(data['Year']).astype(int)
    data['MONTH'] = data['MONTH'].combine_first(data['Month']).astype(int)
    data['DAY'] = data['DAY'].combine_first(data['Day']).astype(int)

    data['TAVG'] = data['TAVG_new'].combine_first(data['TAVG'])
    data['TAVG'] = data['TAVG_new'].combine_first(data['TAVG'])  # Replace NA in 'TAVG' with hourly agg
    data['TMIN'] = data['TMIN_new'].combine_first(data['TMIN'])  # Replace NA in 'TMIN' with hourly agg
    data['TMAX'] = data['TMAX_new'].combine_first(data['TMAX'])  # Replace NA in 'TMAX' with hourly agg
    data['PRCP'] = data['PRCP_new'].combine_first(data['PRCP'])  # Replace NA in 'PRCP' with hourly agg
    data['SNOW'] = data['SNOW_new'].combine_first(data['SNOW'])
    # Drop the extra columns created during the merge (e.g., columns with '_new' suffix)
    data['SNWD'] = data['SNWD'].fillna(0)
    data = data.drop(columns=[col for col in data.columns if col.endswith('_new') or col in ['Year', 'Mo

    return data

def process_all_weather_data(data_directory, hourly_data_directory, locations):
    """
    Process all weather data files in the given directories using the provided location mapping.
    """

    # Ensure the 'combined' directory exists within data_directory
    combined_directory = os.path.join(data_directory, 'combined')
    os.makedirs(combined_directory, exist_ok=True)  # Create the combined directory if it doesn't exist

    # Iterate through each location in the locations dictionary
    for city, details in locations.items():
        airport_id = details['id']
        hourly_data_file = f"{hourly_data_directory}/{city}_hourly_data.csv"
        data_file = f"{data_directory}/{airport_id}.csv"

        if os.path.exists(data_file) and os.path.exists(hourly_data_file):
            #print(f"Processing data for {city}...")

            # Process the data using the process_weather_data function
            final_data = process_weather_data(data_file, hourly_data_file)

            # Save the processed data to the 'combined' directory
            output_file = os.path.join(combined_directory, f"{airport_id}.csv")
            final_data.to_csv(output_file, index=False)
            #print(f"Processed data saved to {output_file}")

        else:
            print(f"Files not found for {city}. Skipping...")
```

```

# Downloading data for OpenWeather

# Coordinates for each location
locations = {
    "Anchorage": {"lat": 61.2181, "lon": -149.9003, "id": "PANC"},
    "Boise": {"lat": 43.6150, "lon": -116.2023, "id": "KBOI"},
    "Chicago": {"lat": 41.8781, "lon": -87.6298, "id": "KORD"},
    "Denver": {"lat": 39.7392, "lon": -104.9903, "id": "KDEN"},
    "Detroit": {"lat": 42.3314, "lon": -83.0458, "id": "KDTW"},
    "Honolulu": {"lat": 21.3069, "lon": -157.8583, "id": "PHNL"},
    "Houston": {"lat": 29.7604, "lon": -95.3698, "id": "KIAH"},
    "Miami": {"lat": 25.7617, "lon": -80.1918, "id": "KMIA"},
    "Minneapolis": {"lat": 44.9778, "lon": -93.2650, "id": "KMSP"},
    "Oklahoma City": {"lat": 35.4676, "lon": -97.5164, "id": "KOKC"},
    "Nashville": {"lat": 36.1627, "lon": -86.7816, "id": "KBNA"},
    "New York": {"lat": 40.7128, "lon": -74.0060, "id": "KJFK"},
    "Phoenix": {"lat": 33.4484, "lon": -112.0740, "id": "KPHX"},
    "Portland ME": {"lat": 43.6591, "lon": -70.2568, "id": "KPWM"},
    "Portland OR": {"lat": 45.5051, "lon": -122.6750, "id": "KPDX"},
    "Salt Lake City": {"lat": 40.7608, "lon": -111.8910, "id": "KSLC"},
    "San Diego": {"lat": 32.7157, "lon": -117.1611, "id": "KSAN"},
    "San Francisco": {"lat": 37.7749, "lon": -122.4194, "id": "KSFO"},
    "Seattle": {"lat": 47.6062, "lon": -122.3321, "id": "KSEA"},
    "Washington DC": {"lat": 38.9072, "lon": -77.0369, "id": "KDCA"}
}

# Directory where data will be saved
repo_root = os.path.abspath(os.path.join(os.path.dirname(__file__), '..'))
directory = os.path.join(repo_root, 'data/original/openweather_hourly/')

# Ensure the directory exists
os.makedirs(directory, exist_ok=True)

# Define the time range: past 1 year (in Unix timestamps)
end_time = int(time.time())  # Current time in Unix timestamp
start_time = end_time - (22 * 24 * 3600)  # One year ago in Unix timestamp


# Function to retrieve data for a specific time range
def get_data_for_range(city, lat, lon, start_time, end_time):
    # Construct the URL for the API call with units=imperial for Fahrenheit
    url = f'https://history.openweathermap.org/data/2.5/history/city?lat={lat}&lon={lon}&type=hour&star
    # Send the API request
    response = requests.get(url)

    if response.status_code == 200:
        data_json = response.json()  # If request is successful, return data_json
        return data_json
    else:
        print(f"Failed to retrieve data: {response.status_code}, {response.text}")
        return None  # Return None if the request failed


# Loop through each location and retrieve the historical weather data in chunks
for city, coords in locations.items():
    lat = coords["lat"]
    lon = coords["lon"]

    # Initialize an empty list to store the full data for this city
    full_data = []

    # Define the time range in chunks (1 week at a time)
    current_start_time = start_time
    while current_start_time < end_time:
        current_end_time = min(current_start_time + (7 * 24 * 3600), end_time)  # One week ahead, or un

        # Get the data for this chunk
        data_json = get_data_for_range(city, lat, lon, current_start_time, current_end_time)

        if data_json and 'list' in data_json:
            # print(
            #     f"Data retrieval successful for {city} from {time.strftime('%Y-%m-%d', time.gmtime(cur
            full_data.append(data_json['list'])  # Append the data chunk to the full data list
        else:
            print(
                f"Failed to retrieve data for {city} from {time.strftime('%Y-%m-%d', time.gmtime(curren

        # Move the start time forward by one week
        current_start_time = current_end_time

    # Save the full data as a JSON file for later use
    if full_data:
        file_name = f'{city}_hourly_data.json'
        file_path = os.path.join(directory, file_name)

        # Flatten the list of data chunks and save to file
        with open(file_path, 'w') as json_file:
            json.dump(full_data, json_file, indent=4)
        # print(f"Full data saved to '{file_path}'.")
    else:
        print(f"No data saved for {city}.")

API_KEY = os.getenv('OPENWEATHER_API_KEY')

# Dictionary containing city names and their coordinates (latitude, longitude)
```

```python
city_coordinates = {
    'Anchorage': (61.2181, -149.9003),
    'Boise': (43.615, -116.2023),
    'Chicago': (41.8781, -87.6298),
    'Denver': (39.7392, -104.9903),
    'Detroit': (42.3314, -83.0458),
    'Honolulu': (21.3069, -157.8583),
    'Houston': (29.7604, -95.3698),
    'Miami': (25.7617, -80.1918),
    'Minneapolis': (44.9778, -93.265),
    'Oklahoma City': (35.4676, -97.5164),
    'Nashville': (36.1627, -86.7816),
    'New York': (40.7128, -74.006),
    'Phoenix': (33.4484, -112.074),
    'Portland ME': (43.6591, -70.2568),
    'Portland OR': (45.5051, -122.675),
    'Salt Lake City': (40.7608, -111.891),
    'San Diego': (32.7157, -117.1611),
    'San Francisco': (37.7749, -122.4194),
    'Seattle': (47.6062, -122.3321),
    'Washington DC': (38.9072, -77.0369)
}

BASE_URL = 'https://api.openweathermap.org/data/2.5/weather'

# Directory to save the combined weather data file
output_dir = '../data/original'
os.makedirs(output_dir, exist_ok=True)

# CSV file path
csv_file_path = os.path.join(output_dir, "current_weather_data.csv")

# Fetch weather data and save to CSV
with open(csv_file_path, mode='w', newline='') as file:
    writer = csv.writer(file)
    # Write the header row
    writer.writerow([
        "City", "Latitude", "Longitude", "Weather Description", "Temperature (F)", "Feels Like (F)",
        "Temp Min (F)", "Temp Max (F)", "Pressure", "Humidity", "Sea Level", "Ground Level",
        "Wind Speed", "Wind Deg", "Wind Gust", "Clouds All", "Datetime", "Country",
        "Sunrise", "Sunset", "Timezone", "City ID", "City Name"
    ])

    # Function to fetch weather data for a city
    def fetch_weather(city, lat, lon):
        params = {
            'lat': lat,
            'lon': lon,
            'appid': API_KEY,
            'units': 'imperial'  # Use 'metric' for Celsius
        }
        response = requests.get(BASE_URL, params=params)
        if response.status_code == 200:
            weather_data = response.json()
            # Extract relevant data
            coord = weather_data.get('coord', {})
            weather = weather_data.get('weather', [{}])[0]
            main = weather_data.get('main', {})
            wind = weather_data.get('wind', {})
            clouds = weather_data.get('clouds', {})
            sys = weather_data.get('sys', {})

            # Create a row with all requested data, handling missing values
            return [
                city,
                coord.get('lat', 'N/A'), coord.get('lon', 'N/A'),
                weather.get('description', 'N/A'),
                main.get('temp', 'N/A'), main.get('feels_like', 'N/A'),
                main.get('temp_min', 'N/A'), main.get('temp_max', 'N/A'),
                main.get('pressure', 'N/A'), main.get('humidity', 'N/A'),
                main.get('sea_level', 'N/A'), main.get('grnd_level', 'N/A'),
                wind.get('speed', 'N/A'), wind.get('deg', 'N/A'), wind.get('gust', 'N/A'),
                clouds.get('all', 'N/A'),
                weather_data.get('dt', 'N/A'), sys.get('country', 'N/A'),
                sys.get('sunrise', 'N/A'), sys.get('sunset', 'N/A'),
                weather_data.get('timezone', 'N/A'), weather_data.get('id', 'N/A'),
                weather_data.get('name', 'N/A')
            ]
        else:
            print(f"Error fetching data for {city}. Status code: {response.status_code}")
            return [city] + ["Error"] * 21

    # Fetch and write weather data for each city
    for city, (lat, lon) in city_coordinates.items():
        city_weather = fetch_weather(city, lat, lon)
        writer.writerow(city_weather)

print(f"Detailed weather data saved to {csv_file_path}")
```

Data Downloading and Preprocessing

Rgression Dataset

```
# In the following functions we create a regression database to train predictors on.
# Each row of the database will represent one "training example" which consists of:
# - The current date
# - The station string
# - The previous 30 days data of TMIN, TAVG, TMAX, SNOW, PRCP, flattened into a 1D array
# - The labels, which next 5 days data of TMIN, TAVG, TMAX, flattened into a 1D array

def is_valid_date(year, month, day):
  """
  Checks if a given date is valid.
  Args:
    year (int): The year.
    month (int): The month (1-12).
    day (int): The day.

  Returns:
    bool: True if the date is valid, False otherwise.
  """

  try:
    datetime.date(year, month, day)
    return True
  except ValueError:
    return False

stations_list = [
    "PANC", # Anchorage
    "KBOI", # Boise
    "KORD", # Chicago
    "KDEN", # Denver
    "KDTW", # Detroit
    "PHNL", # Honolulu
    "KIAH", # Houston
    "KMIA", # Miami
    "KMSP", # Minneapolis
    "KOKC", # Oklahoma City
    "KBNA", # Nashville
    "KJFK", # New York
    "KPHX", # Phoenix
    "KPWM", # Portland ME
    "KPDX", # Portland OR
    "KSLC", # Salt Lake City
    "KSAN", # San Diego
    "KSFO", # San Francisco
    "KSEA", # Seattle
    "KDCA", # Washington DC
]

# Write function that takes in a given day and station, and returns a single row of the regression data

def create_regression_example(year, month, day, station):
    """
    Create a single row of the regression dataset for a given day and station.
    """
    # Get the data for the specified station and year
    repo_root = os.path.abspath(os.path.join(os.path.dirname(__file__), '..'))
    file_path = os.path.join(repo_root, f"data/restructured_simple/{station}.csv")
    df = pd.read_csv(file_path)

    # Find current day index
    current_day_index = df[(df['YEAR'] == year) & (df['MONTH'] == month) & (df['DAY'] == day)].index[0]

    # Get the previous 30 days data of TMIN, TAVG, TMAX, SNOW, PRCP, flattened into a 1D array
```

```python
    previous_data = df.loc[current_day_index-30:current_day_index-1, ['TMIN', 'TAVG', 'TMAX', 'SNOW', '
    previous_data = previous_data.flatten()

    # Get the labels, which are the next 5 days data of TMIN, TAVG, TMAX, flattened into a 1D array
    labels = df.loc[current_day_index:current_day_index+4, ['TMIN', 'TAVG', 'TMAX']].values
    labels = labels.flatten()

    # Combine all the data into a single row
    example = np.concatenate([np.array([year, month, day]), previous_data, labels])

    # reshape the array to be (x,1)
    #example = example.reshape(-1, 1)

    return example




# Create the regression dataset for a given station
# Rules for which current days to include:
# - Only use data from 2014 to 2023
# - Use all days not including November 15 to December 15
# Input: station (string)
# Output: dataframe of regression dataset

def create_regression_dataset(station):

    """
    Create the regression dataset for a given station.

    Args:
        station (str): The station for which to create the regression dataset.

    Returns:
        pd.DataFrame: The regression dataset for the specified station.
    """
    # Initialize an empty list to store the regression examples
    regression_examples = []
    # Loop through all days from 2014 to 2023
    for year in range(2014, 2025):
        for month in range(1, 13):
            for day in range(1, 32):

                if year == 2024 and month >= 11:
                    continue
                # Skip days from November 15 to December 15
                if month == 11 and day >= 15 and day <= 30:
                    continue
                if month == 12 and day >= 1 and day <= 15:
                    continue
                # Skip invalid dates
                if not is_valid_date(year, month, day):
                    continue

                # Create a regression example for the current day
                example = create_regression_example(year, month, day, station)
                regression_examples.append(example)

    # Save the regression examples in a CSV
    regression_df = pd.DataFrame(regression_examples)
    repo_root = os.path.abspath(os.path.join(os.path.dirname(__file__), '..'))
    path_dir = os.path.join(repo_root, f'analysis/regression_data/{station}.csv')
    regression_df.to_csv(path_dir, index=False)



def create_regression_dataset_full(station):

    """
    Create the regression dataset for a given station.

    Args:
        station (str): The station for which to create the regression dataset.

    Returns:
        pd.DataFrame: The regression dataset for the specified station.
    """
    # Initialize an empty list to store the regression examples
    regression_examples = []
    # Loop through all days from 2014 to 2023
    for year in range(2014, 2025):
        for month in range(1, 13):
            for day in range(1, 32):
                # Skip invalid dates
                if year == 2024 and month >= 11:
                    continue

                if not is_valid_date(year, month, day):
                    continue

                # Create a regression example for the current day
                example = create_regression_example(year, month, day, station)
                regression_examples.append(example)

    # Save the regression examples in a CSV
    regression_df = pd.DataFrame(regression_examples)
    repo_root = os.path.abspath(os.path.join(os.path.dirname(__file__), '..'))
    path_dir = os.path.join(repo_root, f'analysis/regression_data_full/{station}.csv')
    regression_df.to_csv(path_dir, index=False)

# do the same for all stations
for station in stations_list:
    create_regression_dataset(station)
    create_regression_dataset_full(station)
    print(f"Created regression dataset for station {station}")
```

Evaluation

```
# The following functions contain code to evaluate the performance of the model.
# The model is evaluated using the mean squared error (MSE)


# Helper function to get data for a specific station at a specific year.
# Given the year X, we want data from Oct 1, X to Dec. 31, X.
# Input: station (string), year (int), directory (string)

def get_data_station_year(station, year, directory = "data/restructured_simple/"):
    # Get the data for the specified station and year
    filename = f"{station}.csv"
    file_path = os.path.join(directory, filename)
    df = pd.read_csv(file_path)

    # The first column of the dataframe is the year, second is the month, third is the day
    df = df[((df['YEAR'] == year) & (df['MONTH'] >= 10)) ]

    return df

# Helper function to predict t_min, t_avg, t_max for the next five days, given the current day of a spe
# Input: station (string), year (int), month (int), day (int), predictor (Predictor object), data (data
# Output: list of predicted temperatures for the next five days

def predict_station_day(station, predictor, data):

    predictions = predictor.predict(data, station)
    #print(predictions)
    return predictions

#predict_station_day("KBNA", 2023, 11, 19, predictor.test_predictor.TestPredictor(), None)

# Helper function to get MSE for a specific station and current day for predictions of the next five da
# Input: station (string), year (int), month (int), day (int), predictor (Predictor object), data (data
# Output: MSE (float)

def get_mse_station_day(station, year, month, day, predictor, data):
    # get index of current day
    current_day_index = data[(data['YEAR'] == year) & (data['MONTH'] == month) & (data['DAY'] == day)].i
    # grab the following five rows using index
    actual_temps = data.loc[current_day_index:current_day_index+4, ['TMIN', 'TAVG', 'TMAX']].values
    # make into a 1D list
    actual_temps = actual_temps.flatten()
    actual_temps = actual_temps.tolist()

    # grab the data up to the current day
    data = data.loc[:current_day_index - 1]

    # Get the predicted temperatures for the next five days
    predicted_temps = predict_station_day(station, predictor, data)

    mse = np.mean((actual_temps - predicted_temps) ** 2)
    #print(mse)
    return mse



# Evaluate the model on a given station and year
# Input: station (string), year (int), predictor (Predictor object)
# Output: MSE (float)

def evaluate_model_station_year(station, year, predictor):
    # Get the data for the specified station and year
    df = get_data_station_year(station, year)
    # Initialize a list to store the MSE for each day
    mses = []
    # Iterate over each day in November and December of the given year
    for month in range(11, 13):
        if month == 11:
            day_range = range(25, 31)
        else:
            day_range = range(1, 11)
        for day in day_range:
            # Calculate the MSE for the current day
            mse = get_mse_station_day(station, year, month, day, predictor, df)
            #print(mse)
            mses.append(mse)
    # Calculate the average MSE for the year
    avg_mse = np.mean(mses)
    return avg_mse


# Function to evaluate for all stations in a given year
# Input: year (int), predictor (Predictor object)
# Output: Mean MSE (float)

def evaluate_model_year(year, predictor):
    # Initialize a list to store the MSE for each station
    mse_list = []
    # Iterate over each station
    for station in stations_list:
        # Calculate the MSE for the station and year
        mse = evaluate_model_station_year(station, year, predictor)
        #print(station)
        #print(mse)
        mse_list.append(mse)
    # print(mse_list)
    # Calculate the mean MSE for all stations
    mean_mse = np.mean(mse_list)
    return mean_mse

# evaluate_model_year(2019, predictor.test_predictor.LinearRegressionPredictor())

# Function to evaluate the model for a given amount of years
# Input: start_year (int), end_year (int), predictor (Predictor object)
```

```python
# Output: List of mean MSE for each year

def evaluate_model_years(start_year, end_year, predictor):
    mse_list = []
    for year in range(start_year, end_year + 1):
        mse = evaluate_model_year(year, predictor)
        mse_list.append(mse)
        # print(mse)
    # make mse_list into python list
    # mse_list = mse_list.tolist()

    # round to 2 decimal places
    mse_list = np.round(mse_list, 2)
    return mse_list

# function to evaluate all models for a given amount of years
def eval_all_models_years(models, start_year, end_year):
    # create a list of all models in predictor.test_predictor


    # print the model name and the mean mse for each year
    for model in models:
        print(str(model))
        print(evaluate_model_years(start_year, end_year, model).round(2))

# define a function that takes in a list of predictors and a year range, and returns the mean mse acros
# for different weights of the models with WeightedPredictor

def testing_weight_predictor(predictor_list, start_year, end_year, iters):
    # get number of predictors
    num_predictors = len(predictor_list)
    # create a list of a list of weights to iterate through. Each row is a different set of weights,
    # must have num_predictor floats, and sum to 1.
    weights_list = np.random.dirichlet(np.ones(num_predictors), size=iters)

    # create a list to store the mean mse for each set of weights
    mse_list = []

    # iterate through each set of weights
    for weights in weights_list:
        print("once")

        # create a WeightedPredictor object with the given weights
        model = predictor.test_predictor.WeightedPredictor(predictor_list, weights)
        # get the mean mse for the given set of weights
        mse = evaluate_model_years(start_year, end_year, model)
        # find mean of mse
        mse = np.mean(mse)
        # add to mse_list
        mse_list.append(mse)

    # combine the weights list with the mse list, and sort by mse lowest to highest
    mse_list = np.array(mse_list)
    weights_list = np.round(weights_list, 2)
    mse_list = np.round(mse_list, 2)
    combined = np.column_stack((weights_list, mse_list))
    combined = combined[combined[:,num_predictors].argsort()]

    # save combined to a csv
    np.savetxt("predictor/weights/weight_mse.csv", combined, delimiter=",", fmt='%s')

    return combined



predictor_list = [ #predictor.test_predictor.PreviousDayPredictor(),
                  predictor.test_predictor.LinearRegressionPredictor(),
                  predictor.test_predictor.RidgeRegressionPredictor(),
                  predictor.test_predictor.LassoPredictor(),
                  predictor.test_predictor.RandomForestPredictor()
                  ]
```

Prediction Models

```
# arima

def arima_forecast(series, order=(1, 1, 1), steps=5, start_date=None):
    model = ARIMA(series, order=order)
    model_fit = model.fit()
    forecast = model_fit.forecast(steps=steps)
    future_dates = [start_date + pd.Timedelta(days=i) for i in range(1, steps + 1)]
    forecast_df = pd.DataFrame({'DATE': future_dates, 'FORECAST': forecast})
    return forecast_df

def evaluate_predictions(actual, predicted):
    mse = mean_squared_error(actual, predicted)
    mae = mean_absolute_error(actual, predicted)
    return mse, mae

directory = '.'
output_file = 'output/AllCities_ARIMA.txt'
os.makedirs('output', exist_ok=True)

def combine_future_predictions(tmin_forecast_df, tmax_forecast_df, tavg_forecast_df):
    """Combine TMIN, TMAX, and TAVG forecasts into a single DataFrame."""
    combined_forecast = pd.DataFrame({
        'DATE': tmin_forecast_df['DATE'],
        'TMIN': tmin_forecast_df['FORECAST'],
        'TMAX': tmax_forecast_df['FORECAST'],
        'TAVG': tavg_forecast_df['FORECAST']
    })
    return combined_forecast

for filename in os.listdir(directory):
    if filename.endswith('.csv'):
        city_name = os.path.splitext(filename)[0]
        file_path = os.path.join(directory, filename)
        df = pd.read_csv(file_path)

        df['DATE'] = pd.to_datetime(df[['YEAR', 'MONTH', 'DAY']], errors='coerce')

        # need to be corrected
        # just for test
        df = df[df['DATE'] <= '2024-11-19']  # Filter data before or on 2024-11-19

        df['TMAX'] = df['TMAX'].replace(0, np.nan)
        df['TMIN'] = df['TMIN'].replace(0, np.nan)
        df['TAVG'] = (df['TMAX'] + df['TMIN']) / 2
        df['TMAX'] = df['TMAX'].interpolate(method='linear')
        df['TMIN'] = df['TMIN'].interpolate(method='linear')
        df['TAVG'] = df['TAVG'].interpolate(method='linear')

        # Train-Test Split
        train_tmax = df['TMAX'][-600:-5]
        test_tmax = df['TMAX'][-5:]
        train_tmin = df['TMIN'][-600:-5]
        test_tmin = df['TMIN'][-5:]
        train_tavg = df['TAVG'][-600:-5]
        test_tavg = df['TAVG'][-5:]

        start_date = df['DATE'].max()
        tmax_arima_forecast_df = arima_forecast(train_tmax, order=(5, 1, 3), steps=5, start_date=start_
        tmin_arima_forecast_df = arima_forecast(train_tmin, order=(5, 1, 1), steps=5, start_date=start_
        tavg_arima_forecast_df = arima_forecast(train_tavg, order=(5, 1, 3), steps=5, start_date=start_

        combined_forecast_df = combine_future_predictions(tmin_arima_forecast_df, tmax_arima_forecast_d

        # Evaluate ARIMA predictions
        tmax_arima_mse, tmax_arima_mae = evaluate_predictions(test_tmax, tmax_arima_forecast_df['FORECA
        tmin_arima_mse, tmin_arima_mae = evaluate_predictions(test_tmin, tmin_arima_forecast_df['FORECA
        tavg_arima_mse, tavg_arima_mae = evaluate_predictions(test_tavg, tavg_arima_forecast_df['FORECA

        output_text = f"City: {city_name}\n"
        output_text += f"TMAX - MSE: {tmax_arima_mse:.2f}, MAE: {tmax_arima_mae:.2f}\n"
        output_text += f"TMIN - MSE: {tmin_arima_mse:.2f}, MAE: {tmin_arima_mae:.2f}\n"
        output_text += f"TAVG - MSE: {tavg_arima_mse:.2f}, MAE: {tavg_arima_mae:.2f}\n\n"
        output_text += "Future 5-Day Predictions (Combined):\n"
        output_text += combined_forecast_df.to_string(index=False)
        output_text += "\n\n" + "-" * 50 + "\n\n"

        with open(output_file, 'a', encoding='utf-8') as f:
            f.write(output_text)

        print(f"Processed {city_name}, results appended to {output_file}")

# historical mean

def evaluate_predictions(actual, predicted, num_features=1):
```

```python
    mse = mean_squared_error(actual, predicted)
    mae = mean_absolute_error(actual, predicted)
    r2 = r2_score(actual, predicted)  # R² calculation
    n = len(actual)  # Number of data points
    adj_r2 = 1 - (1 - r2) * (n - 1) / (n - num_features - 1)  # Adjusted R²
    return mse, mae, r2, adj_r2

# Define Historical Mean prediction function
def historical_mean_forecast(series, steps=5, start_date=None):
    mean_value = series.mean()
    forecast_values = [mean_value] * steps
    future_dates = [start_date + pd.Timedelta(days=i) for i in range(1, steps + 1)]
    forecast_df = pd.DataFrame({'DATE': future_dates, 'FORECAST': forecast_values})
    return forecast_df


directory = '.'
output_file = 'output/AllCities_HistoricalMean.txt'
os.makedirs('output', exist_ok=True)

with open(output_file, 'w', encoding='utf-8') as f:
    f.write("Historical Mean Results for All Cities:\n\n")

for filename in os.listdir(directory):
    if filename.endswith('.csv'):
        city_name = os.path.splitext(filename)[0]
        file_path = os.path.join(directory, filename)
        df = pd.read_csv(file_path)

        df['DATE'] = pd.to_datetime(df[['YEAR', 'MONTH', 'DAY']], errors='coerce')
        # just for test
        # df = df[df['DATE'] <= '2024-11-19']  # Filter data before or on 2024-11-19
        df['TAVG'] = (df['TMAX'] + df['TMIN']) / 2
        df['TMAX'] = df['TMAX'].interpolate(method='linear')
        df['TMIN'] = df['TMIN'].interpolate(method='linear')
        df['TAVG'] = df['TAVG'].interpolate(method='linear')

        # Train-Test Split
        train_tmax = df['TMAX'][:-5]
        test_tmax = df['TMAX'][-5:]
        train_tmin = df['TMIN'][:-5]
        test_tmin = df['TMIN'][-5:]
        train_tavg = df['TAVG'][:-5]
        test_tavg = df['TAVG'][-5:]

        start_date = df['DATE'].max()  # Start date for predictions
        tmax_historical_forecast_df = historical_mean_forecast(train_tmax, steps=5, start_date=start_da
        tmin_historical_forecast_df = historical_mean_forecast(train_tmin, steps=5, start_date=start_da
        tavg_historical_forecast_df = historical_mean_forecast(train_tavg, steps=5, start_date=start_da

        tmax_mse, tmax_mae, tmax_r2, tmax_adj_r2 = evaluate_predictions(test_tmax, tmax_historical_fore
        tmin_mse, tmin_mae, tmin_r2, tmin_adj_r2 = evaluate_predictions(test_tmin, tmin_historical_fore
        tavg_mse, tavg_mae, tavg_r2, tavg_adj_r2 = evaluate_predictions(test_tavg, tavg_historical_fore
        combined_forecast_df = pd.DataFrame({
            'DATE': tmax_historical_forecast_df['DATE'],
            'TMIN': tmin_historical_forecast_df['FORECAST'],
            'TMAX': tmax_historical_forecast_df['FORECAST'],
            'TAVG': tavg_historical_forecast_df['FORECAST']
        })

        output_text = f"City: {city_name}\n"
        output_text += f"TMAX - MSE: {tmax_mse:.2f}, MAE: {tmax_mae:.2f}, R²: {tmax_r2:.4f}, Adjusted R
        output_text += f"TMIN - MSE: {tmin_mse:.2f}, MAE: {tmin_mae:.2f}, R²: {tmin_r2:.4f}, Adjusted R
        output_text += f"TAVG - MSE: {tavg_mse:.2f}, MAE: {tavg_mae:.2f}, R²: {tavg_r2:.4f}, Adjusted R
        output_text += "Future 5-Day Predictions (Combined):\n"
        output_text += combined_forecast_df.to_string(index=False)
        output_text += "\n\n" + "-" * 50 + "\n\n"

        with open(output_file, 'a', encoding='utf-8') as f:
            f.write(output_text)

        print(f"Processed {city_name}, results appended to {output_file}")


# OLS

def evaluate_model(y_test, y_pred, X_test):
    n = len(y_test)
    p = X_test.shape[1]
    r2 = r2_score(y_test, y_pred)  # R² score
    adj_r2 = 1 - (1 - r2) * (n - 1) / (n - p - 1)  # Adjusted R²
    mse = mean_squared_error(y_test, y_pred)
    rmse = np.sqrt(mse)
    mae = mean_absolute_error(y_test, y_pred)
    return r2, adj_r2, mse, rmse, mae


def generate_future_predictions(data, model_min, model_max, model_avg, days=5, start_date=None, lag_fea
    future_predictions = []
    current_data = data.iloc[-1][lag_features].values.reshape(1, -1)
    current_date = pd.to_datetime(start_date)


    for day in range(1, days + 1):
        pred_min = model_min.predict(current_data)[0]
        pred_max = model_max.predict(current_data)[0]
        pred_avg = model_avg.predict(current_data)[0]

        future_predictions.append({
            'DATE': current_date + pd.Timedelta(days=day),
            'TMIN': pred_min,
            'TMAX': pred_max,
            'TAVG': pred_avg
        })
        current_data = np.roll(current_data, -3)
        current_data[0, :3] = [pred_max, pred_min, pred_avg]  # Insert new predictions


    return pd.DataFrame(future_predictions)


directory = '.'
output_file = 'output/AllCities_LinearRegression.txt'
os.makedirs('output', exist_ok=True)


with open(output_file, 'w', encoding='utf-8') as f:
    f.write("Linear Regression Model Results for All Cities:\n\n")


for filename in os.listdir(directory):
    if filename.endswith('.csv'):
        city_name = os.path.splitext(filename)[0]
        file_path = os.path.join(directory, filename)
        df = pd.read_csv(file_path)

        df['DATE'] = pd.to_datetime(df[['YEAR', 'MONTH', 'DAY']], errors='coerce')  # Handle invalid da
        df = df[df['DATE'].notna()]

        # Filter data before or on 2024-11-19
        # need to be corrected
        # just for test
        # df = df[df['DATE'] <= '2024-11-19']
        df['TMAX'] = df['TMAX'].interpolate(method='linear')
        df['TMIN'] = df['TMIN'].interpolate(method='linear')
        df['PRCP'] = df['PRCP'].interpolate(method='linear')
        df['SNOW'] = df['SNOW'].interpolate(method='linear')
        df['SNWD'] = df['SNWD'].interpolate(method='linear')
        df['TAVG'] = (df['TMAX'] + df['TMIN']) / 2

        for i in range(1, 10):
            df[f'TMAX_lag{i}'] = df['TMAX'].shift(i)
            df[f'TMIN_lag{i}'] = df['TMIN'].shift(i)
            df[f'TAVG_lag{i}'] = df['TAVG'].shift(i)
        df.dropna(inplace=True)

        lag_features = [f'TMAX_lag{i}' for i in range(1, 10)] + \
                       [f'TMIN_lag{i}' for i in range(1, 10)] + \
                       [f'TAVG_lag{i}' for i in range(1, 10)] + ['PRCP', 'SNOW', 'SNWD']


        X = df[lag_features]
        y_min = df['TMIN']
        y_max = df['TMAX']
        y_avg = df['TAVG']
        # Train-Test Split
        # test_size=0.2
        X_train, X_test, y_min_train, y_min_test, y_max_train, y_max_test, y_avg_train, y_avg_test = tra
            X, y_min, y_max, y_avg, test_size=0.2, random_state=42
        )

        model_min = LinearRegression()
        model_min.fit(X_train, y_min_train)

        model_max = LinearRegression()
        model_max.fit(X_train, y_max_train)

        model_avg = LinearRegression()
        model_avg.fit(X_train, y_avg_train)


        y_min_pred = model_min.predict(X_test)
        y_max_pred = model_max.predict(X_test)
        y_avg_pred = model_avg.predict(X_test)

        r2_min, adj_r2_min, mse_min, rmse_min, mae_min = evaluate_model(y_min_test, y_min_pred, X_test)
        r2_max, adj_r2_max, mse_max, rmse_max, mae_max = evaluate_model(y_max_test, y_max_pred, X_test)
        r2_avg, adj_r2_avg, mse_avg, rmse_avg, mae_avg = evaluate_model(y_avg_test, y_avg_pred, X_test)

        future_predictions = generate_future_predictions(df,
                                                         model_min, model_max,
                                                         model_avg, days=5,
                                                         #need to be corrected
                                                         start_date='2024-11-19',
                                                         lag_features=lag_features)

        output_text = f"City: {city_name}\n"

        output_text += f"TMIN Evaluation: R²: {r2_min:.4f}, Adjusted R²: {adj_r2_min:.4f}, MSE: {mse_mi

        output_text += f"TMAX Evaluation: R²: {r2_max:.4f}, Adjusted R²: {adj_r2_max:.4f}, MSE: {mse_ma

        output_text += f"TAVG Evaluation: R²: {r2_avg:.4f}, Adjusted R²: {adj_r2_avg:.4f}, MSE: {mse_av

        output_text += "Future 5-Day Predictions (Combined):\n"

        output_text += future_predictions.to_string(index=False)

        output_text += "\n\n" + "-" * 50 + "\n\n"


        with open(output_file, 'a', encoding='utf-8') as f:
            f.write(output_text)

        print(f"Processed {city_name}, results appended to {output_file}")

# random forest
```

```python
def evaluate_model(y_test, y_pred, X_test):
    """Evaluate a model using multiple metrics, including adjusted R²."""
    n = len(y_test)  # Number of observations
    p = X_test.shape[1]  # Number of predictors
    r2 = r2_score(y_test, y_pred)  # R² score
    adj_r2 = 1 - (1 - r2) * (n - 1) / (n - p - 1)  # Adjusted R²
    mse = mean_squared_error(y_test, y_pred)  # Mean Squared Error
    rmse = np.sqrt(mse)  # Root Mean Squared Error
    mae = mean_absolute_error(y_test, y_pred)  # Mean Absolute Error
    return r2, adj_r2, mse, rmse, mae

def generate_future_predictions(data, model_min, model_max, model_avg, days=5, start_date=None, lag_fea
    future_predictions = []
    current_data = data.iloc[-1][lag_features].values.reshape(1, -1)
    current_date = pd.to_datetime(start_date)

    for day in range(1, days + 1):
        pred_min = model_min.predict(current_data)[0]
        pred_max = model_max.predict(current_data)[0]
        pred_avg = model_avg.predict(current_data)[0]

        future_predictions.append({
            'DATE': current_date + pd.Timedelta(days=day),
            'TMIN': pred_min,
            'TMAX': pred_max,
            'TAVG': pred_avg
        })

        current_data = np.roll(current_data, -3)
        current_data[0, :3] = [pred_max, pred_min, pred_avg]  # Insert new predictions

    return pd.DataFrame(future_predictions)

directory = '.'
output_file = 'output/AllCities_RandomForest.txt'
os.makedirs('output', exist_ok=True)

with open(output_file, 'w', encoding='utf-8') as f:
    f.write("Random Forest Regressor Results for All Cities:\n\n")

for filename in os.listdir(directory):
    if filename.endswith('.csv'):
        city_name = os.path.splitext(filename)[0]
        file_path = os.path.join(directory, filename)
        df = pd.read_csv(file_path)

        df['DATE'] = pd.to_datetime(df[['YEAR', 'MONTH', 'DAY']], errors='coerce')  # Handle invalid da
        df = df[df['DATE'].notna()]

        # just for test
        # need to be corrected
        # df = df[df['DATE'] <= '2024-11-19']  # Filter data before or on 2024-11-19
        df['TMAX'] = df['TMAX'].interpolate(method='linear')
        df['TMIN'] = df['TMIN'].interpolate(method='linear')
        df['PRCP'] = df['PRCP'].interpolate(method='linear')
        df['SNOW'] = df['SNOW'].interpolate(method='linear')
        df['SNWD'] = df['SNWD'].interpolate(method='linear')
        df['TAVG'] = (df['TMAX'] + df['TMIN']) / 2

        for i in range(1, 6):
            df[f'TMAX_lag{i}'] = df['TMAX'].shift(i)
            df[f'TMIN_lag{i}'] = df['TMIN'].shift(i)
            df[f'TAVG_lag{i}'] = df['TAVG'].shift(i)
        df.dropna(inplace=True)
        lag_features = [f'TMAX_lag{i}' for i in range(1, 6)] + \
                       [f'TMIN_lag{i}' for i in range(1, 6)] + \
                       [f'TAVG_lag{i}' for i in range(1, 6)] + ['PRCP', 'SNOW', 'SNWD']

        X = df[lag_features]
        y_min = df['TMIN']
        y_max = df['TMAX']
        y_avg = df['TAVG']

        # Train-Test Split
        X_train, X_test, y_min_train, y_min_test, y_max_train, y_max_test, y_avg_train, y_avg_test = tra
            X, y_min, y_max, y_avg, test_size=0.2, random_state=42
        )

        model_min = RandomForestRegressor(n_estimators=30, max_depth=10,random_state=42)
        model_max = RandomForestRegressor(n_estimators=30, max_depth=10,random_state=42)
        model_avg = RandomForestRegressor(n_estimators=30, max_depth=10,random_state=42)

        model_min.fit(X_train, y_min_train)
        model_max.fit(X_train, y_max_train)
        model_avg.fit(X_train, y_avg_train)

        y_min_pred = model_min.predict(X_test)
        y_max_pred = model_max.predict(X_test)
        y_avg_pred = model_avg.predict(X_test)

        r2_min, adj_r2_min, mse_min, rmse_min, mae_min = evaluate_model(y_min_test, y_min_pred, X_test)
        r2_max, adj_r2_max, mse_max, rmse_max, mae_max = evaluate_model(y_max_test, y_max_pred, X_test)
        r2_avg, adj_r2_avg, mse_avg, rmse_avg, mae_avg = evaluate_model(y_avg_test, y_avg_pred, X_test)

        future_predictions = generate_future_predictions(df, model_min, model_max, model_avg, days=5, s

        output_text = f"City: {city_name}\n"
        output_text += f"TMIN Evaluation: R²: {r2_min:.4f}, Adjusted R²: {adj_r2_min:.4f}, MSE: {mse_mi
        output_text += f"TMAX Evaluation: R²: {r2_max:.4f}, Adjusted R²: {adj_r2_max:.4f}, MSE: {mse_ma
        output_text += f"TAVG Evaluation: R²: {r2_avg:.4f}, Adjusted R²: {adj_r2_avg:.4f}, MSE: {mse_av
        output_text += "Future 5-Day Predictions:\n"
        output_text += future_predictions.to_string(index=False)
        output_text += "\n\n" + "-" * 50 + "\n\n"

        with open(output_file, 'a', encoding='utf-8') as f:
            f.write(output_text)

        print(f"Processed {city_name}, results appended to {output_file}")
```

```
```