AI Weather Forecaster Report

Name: Tausif Ibne Iqbal

Project Summary:

I have built an AI Weather Forecaster which can collect data from the given API and automatically update its dataframe and upgrade the model for the latest forecast. The model is run Recurrent Neural Network (RNN) specifically its Long Short Term Memory (LSTM) variant. I have also built a backend server which can collect data from the Model and display it on the web and alongside it can display live weather from any place around the world. The model can display 24 hours ahead of data accurately. It shows the temperature, humidity, pressure, wind speed and the probability of rain.
I had also planned to build a frontend to display all things nicely on the web but didn't get enough time to do so. Though I plan to do it for the future.

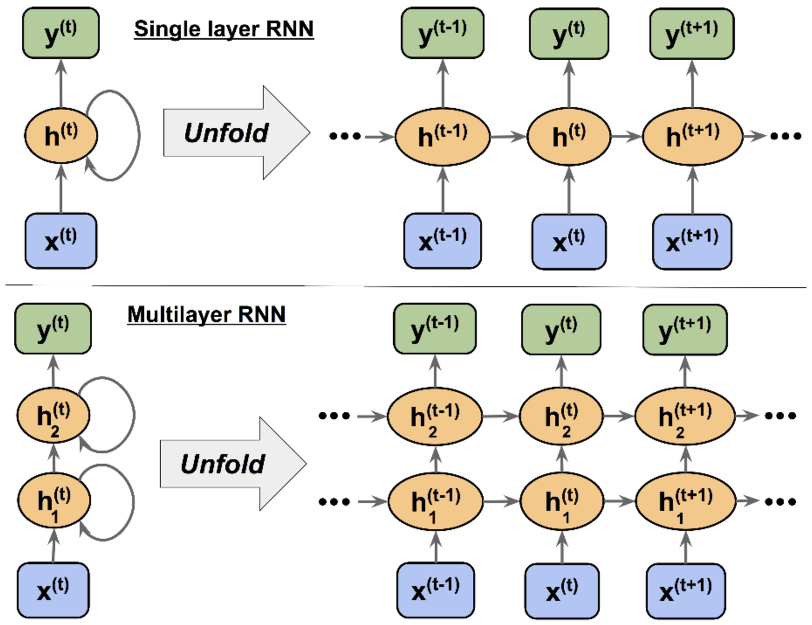
Brief Summary on RNN LSTM:

RNN is fundamentally different from MLP and CNN. The dataset is not random and independent of each other anymore. We are dealing with sequential data where the order of data matters, and so we need to build a Neural Network Memory.

RNN can be of 3 categories: Many-to-one, One-to-many and Many-to-many.

In CNN and MLP information flows from input layer to the hidden layer, and then from the hidden layer to the output layer. On the other hand, in a recurrent network, the hidden layer gets its input from both the input layer and the hidden layer from the previous time step.

Similarly for multilayers the next layers get information from the previous hidden layer's current time step and its own previous time step value.



Each directed edge (the connections between boxes) in the

representation of an
RNN that is shown in the picture is associated with a weight matrix. These weights do not depend on time *t*; therefore, they are shared across the time axis. The different weight matrices in a single layer RNN are as follows:

- $\boldsymbol{W_{xh}}$: The weight matrix between the input x(t) and the hidden layer h.
- $\boldsymbol{W_{hh}}$: The weight matrix between recurrent edges.
- $\boldsymbol{W_{hy}}$: The weight matrix between the hidden layer and the output layer.

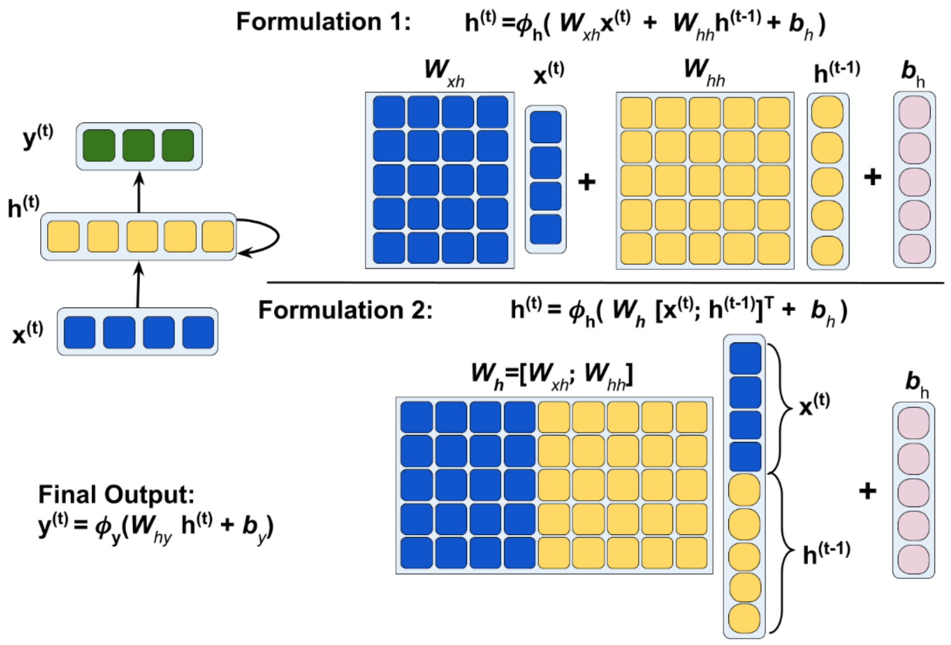
At the end we add a bias b and scaling to correct the values as needed.

$$W_h = [W_{xh} \quad W_{hh}] \; (Weight\ Matrix)$$

$$h^{(t)} = \varphi_h\left(z_h^{(t)}\right) = \varphi_h\left(W_{xh} * x^{(t)} + W_{hh} * h^{(t-1)} + b_h\right)$$

$$h^{(t)} = \varphi_h\left([W_{xh} \quad W_{hh}]\begin{bmatrix} x^{(t)} \\ h^{(t-1)} \end{bmatrix} + b_h\right)$$

These are the math behind the function of the hidden layers. φ_h is the activation function of hidden layers.


Formulation 1: $h^{(t)} = \phi_h(W_{xh}x^{(t)} + W_{hh}h^{(t-1)} + b_h)$
W_{xh}
$x^{(t)}$
W_{hh}
$h^{(t-1)}$
b_h
$y^{(t)}$
$h^{(t)}$
$x^{(t)}$
Formulation 2: $h^{(t)} = \phi_h(W_h\,[x^{(t)}; h^{(t-1)}]^T + b_h)$
$W_h = [W_{xh}; W_{hh}]$
$x^{(t)}$
$h^{(t-1)}$
b_h
Final Output: $y^{(t)} = \phi_y(W_{hy}\,h^{(t)} + b_y)$

My Code: My code has 6 files for the RNN LSTM Model and 1 file for the backend server.

Model Summary:

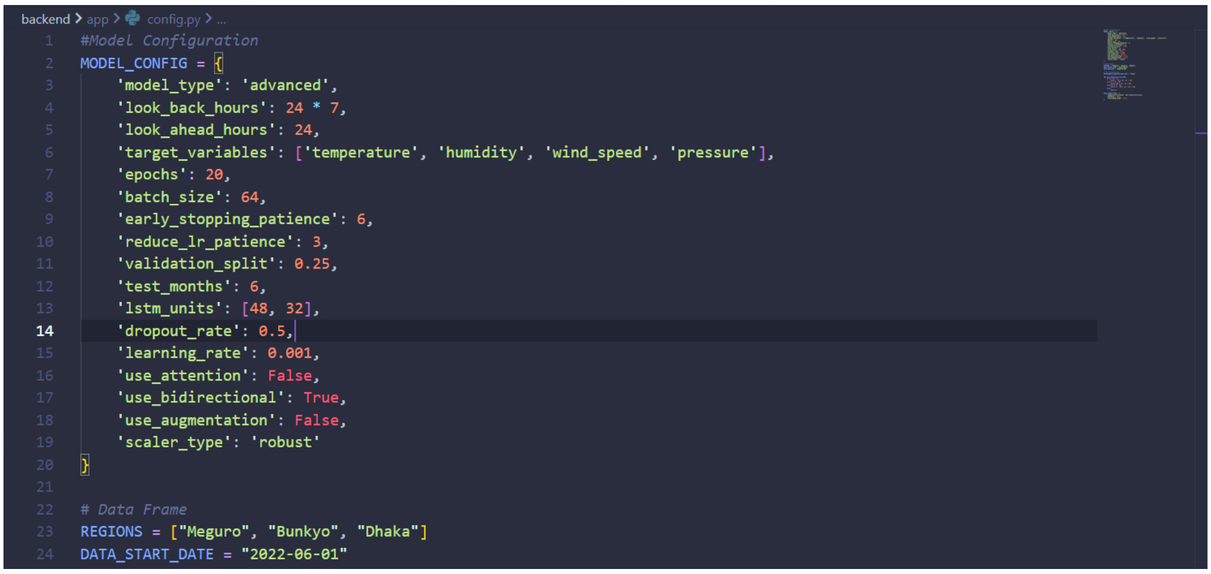
Layer (type)	Output Shape	Param #
Input Layer	(None, 168, 56)	0
Layer Normalization 1	(None, 168, 56)	112
Bidirectional	(None, 168, 96)	40320
Dropout 1	(None, 168, 96)	0
Layer Normalization 2	(None, 168, 96)	192
LSTM	(None, 32)	16512
Dropout 2	(None, 32)	0
Dense 1	(None, 64)	2112
Dropout 3	(None, 64)	0
Dense 2	(None, 32)	2080
Dropout 4	(None, 32)	0
Dense 3	(None, 96)	3168

RNN Model:

I have converted all the files into packages by using the __init__.py file, making them accessible to each other. So now instead of files they are all like packages.

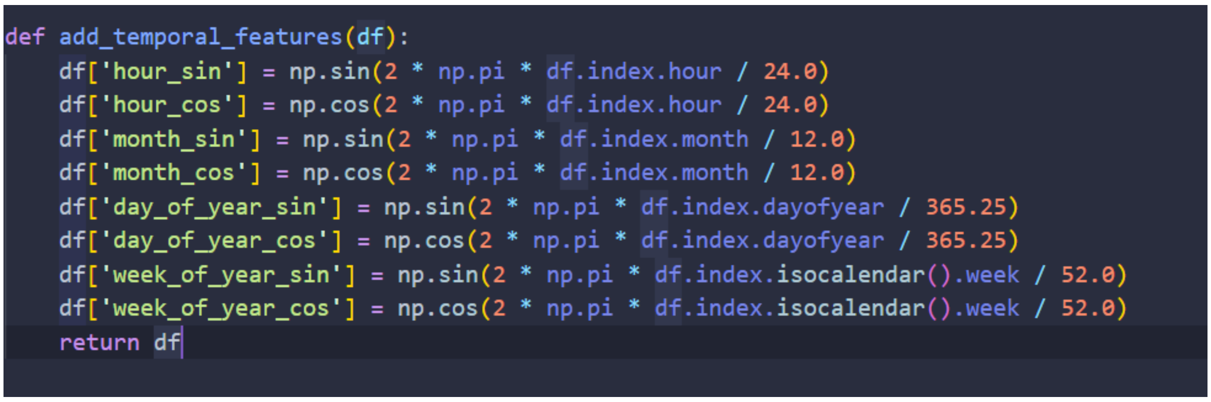
Below are all the packages and how they work.

Config.py: This package stores all the hyperparameters, LSTM Model Architecture biases scaling and my API information.



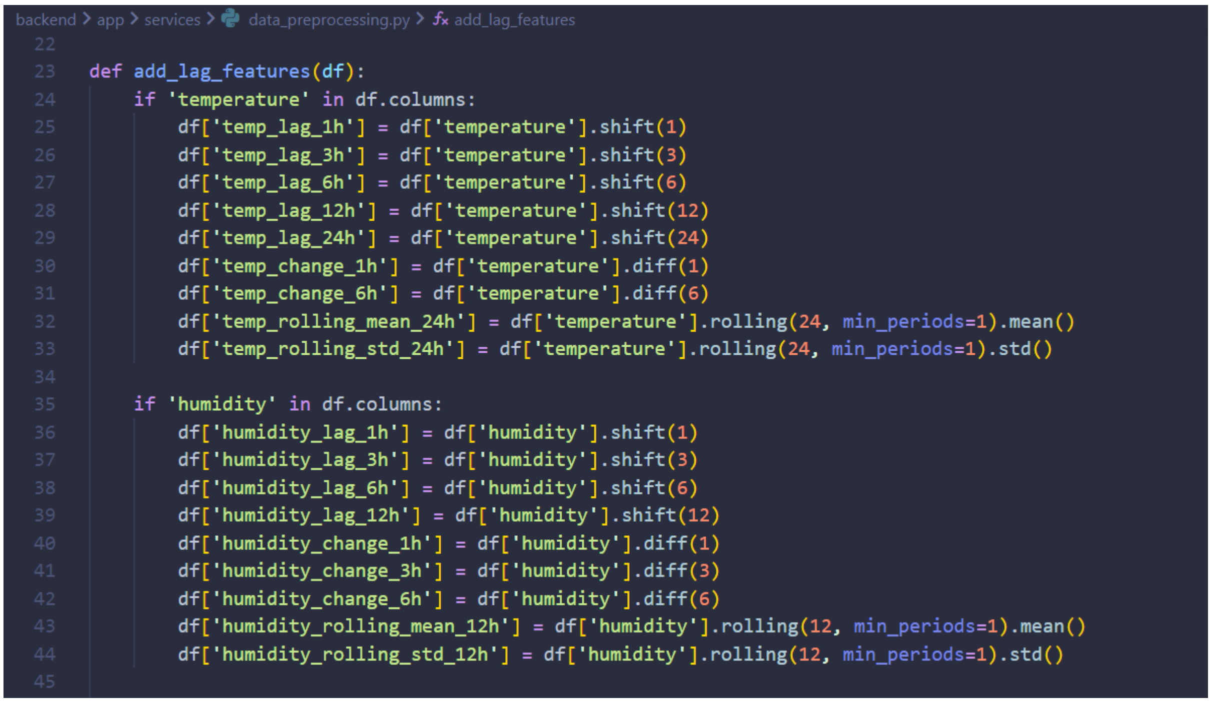
```
#Model Configuration
MODEL_CONFIG = {
    'model_type': 'advanced',
    'look_back_hours': 24 * 7,
    'look_ahead_hours': 24,
    'target_variables': ['temperature', 'humidity', 'wind_speed', 'pressure'],
    'epochs': 20,
    'batch_size': 64,
    'early_stopping_patience': 6,
    'reduce_lr_patience': 3,
    'validation_split': 0.25,
    'test_months': 6,
    'lstm_units': [48, 32],
    'dropout_rate': 0.5,
    'learning_rate': 0.001,
    'use_attention': False,
    'use_bidirectional': True,
    'use_augmentation': False,
    'scaler_type': 'robust'
}

# Data Frame
REGIONS = ["Meguro", "Bunkyo", "Dhaka"]
DATA_START_DATE = "2022-06-01"
```

Data_preprocessing.py: This package transforms the raw time-series data into the structured, into normalized format required by the LSTM model.



```python
def add_temporal_features(df):
    df['hour_sin'] = np.sin(2 * np.pi * df.index.hour / 24.0)
    df['hour_cos'] = np.cos(2 * np.pi * df.index.hour / 24.0)
    df['month_sin'] = np.sin(2 * np.pi * df.index.month / 12.0)
    df['month_cos'] = np.cos(2 * np.pi * df.index.month / 12.0)
    df['day_of_year_sin'] = np.sin(2 * np.pi * df.index.dayofyear / 365.25)
    df['day_of_year_cos'] = np.cos(2 * np.pi * df.index.dayofyear / 365.25)
    df['week_of_year_sin'] = np.sin(2 * np.pi * df.index.isocalendar().week / 52.0)
    df['week_of_year_cos'] = np.cos(2 * np.pi * df.index.isocalendar().week / 52.0)
    return df
```

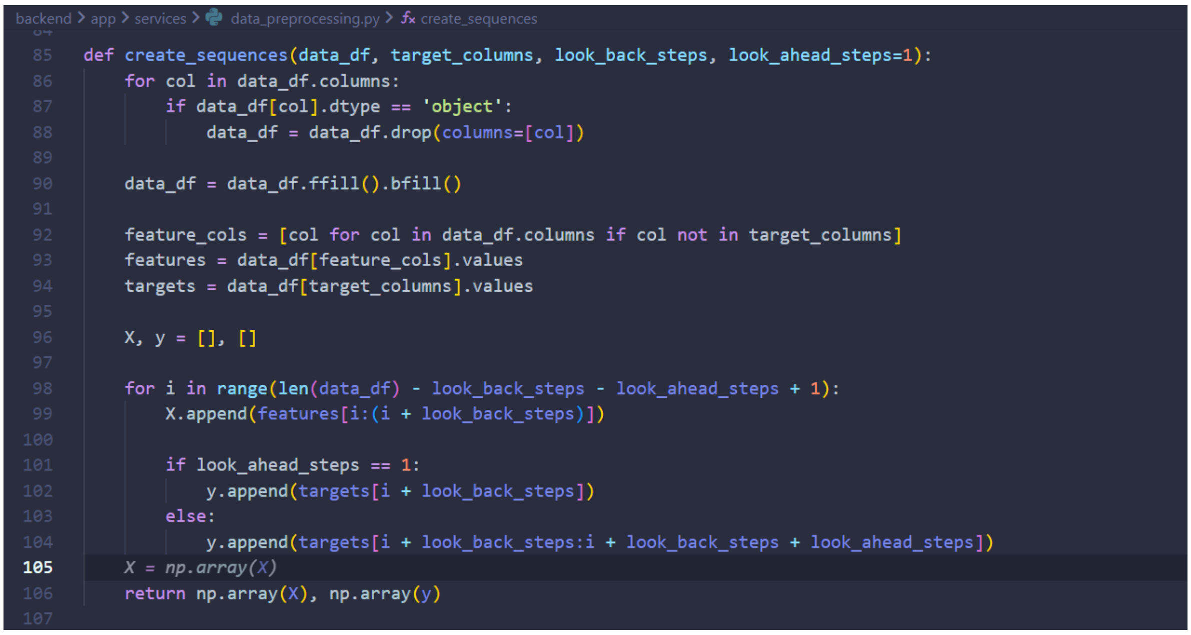
This function extracts the information from the datetime index and represents them using sine and cosine transformations.



backend > app > services > data_preprocessing.py > add_lag_features

```python

def add_lag_features(df):
    if 'temperature' in df.columns:
        df['temp_lag_1h'] = df['temperature'].shift(1)
        df['temp_lag_3h'] = df['temperature'].shift(3)
        df['temp_lag_6h'] = df['temperature'].shift(6)
        df['temp_lag_12h'] = df['temperature'].shift(12)
        df['temp_lag_24h'] = df['temperature'].shift(24)
        df['temp_change_1h'] = df['temperature'].diff(1)
        df['temp_change_6h'] = df['temperature'].diff(6)
        df['temp_rolling_mean_24h'] = df['temperature'].rolling(24, min_periods=1).mean()
        df['temp_rolling_std_24h'] = df['temperature'].rolling(24, min_periods=1).std()

    if 'humidity' in df.columns:
        df['humidity_lag_1h'] = df['humidity'].shift(1)
        df['humidity_lag_3h'] = df['humidity'].shift(3)
        df['humidity_lag_6h'] = df['humidity'].shift(6)
        df['humidity_lag_12h'] = df['humidity'].shift(12)
        df['humidity_change_1h'] = df['humidity'].diff(1)
        df['humidity_change_3h'] = df['humidity'].diff(3)
        df['humidity_change_6h'] = df['humidity'].diff(6)
        df['humidity_rolling_mean_12h'] = df['humidity'].rolling(12, min_periods=1).mean()
        df['humidity_rolling_std_12h'] = df['humidity'].rolling(12, min_periods=1).std()

```

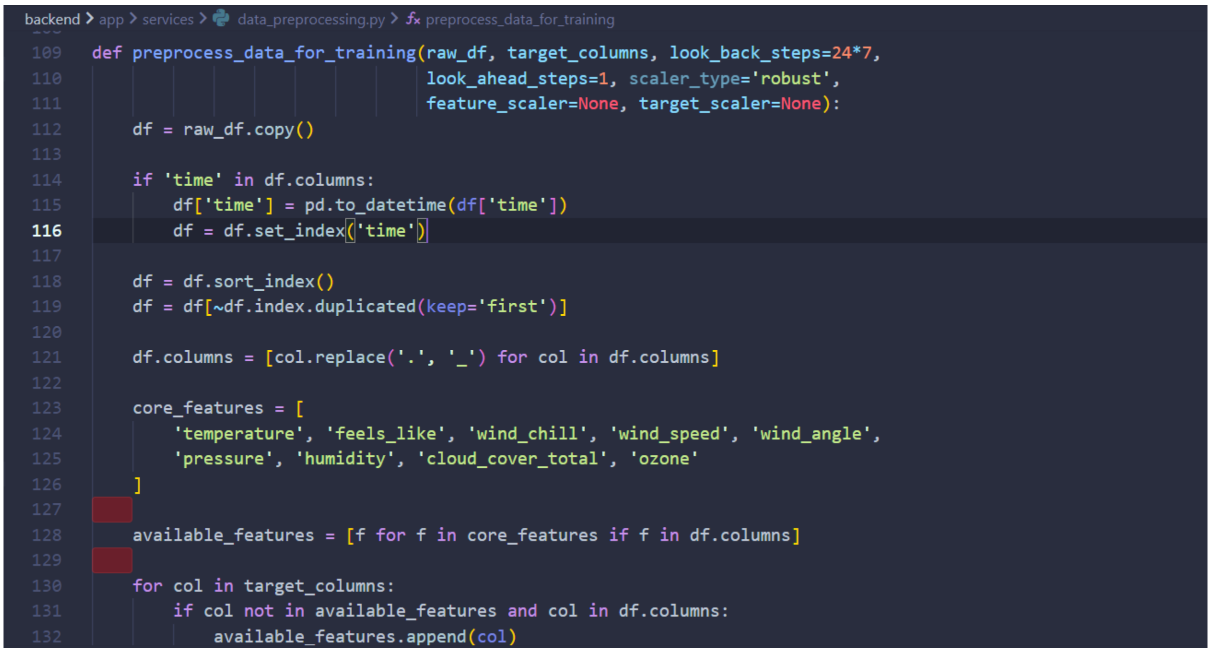
This function introduces new features by shifting existing weather variables by various time lags which are 1,3,6,12 and 24 hours. It also calculates differences and rolling statistics.



backend > app > services > data_preprocessing.py > create_sequences

```python
def create_sequences(data_df, target_columns, look_back_steps, look_ahead_steps=1):
    for col in data_df.columns:
        if data_df[col].dtype == 'object':
            data_df = data_df.drop(columns=[col])

    data_df = data_df.ffill().bfill()

    feature_cols = [col for col in data_df.columns if col not in target_columns]
    features = data_df[feature_cols].values
    targets = data_df[target_columns].values

    X, y = [], []

    for i in range(len(data_df) - look_back_steps - look_ahead_steps + 1):
        X.append(features[i:(i + look_back_steps)])

        if look_ahead_steps == 1:
            y.append(targets[i + look_back_steps])
        else:
            y.append(targets[i + look_back_steps:i + look_back_steps + look_ahead_steps])
    X = np.array(X)
    return np.array(X), np.array(y)

```

This function transforms the previously processed Dataframe into the 3D input format (X) and 2D/3D output format (y) required by LSTM networks.
It explicitly creates sequential input-output pairs that the LSTM will learn from, allowing it to understand how a sequence of past observations leads to a sequence of future observations.

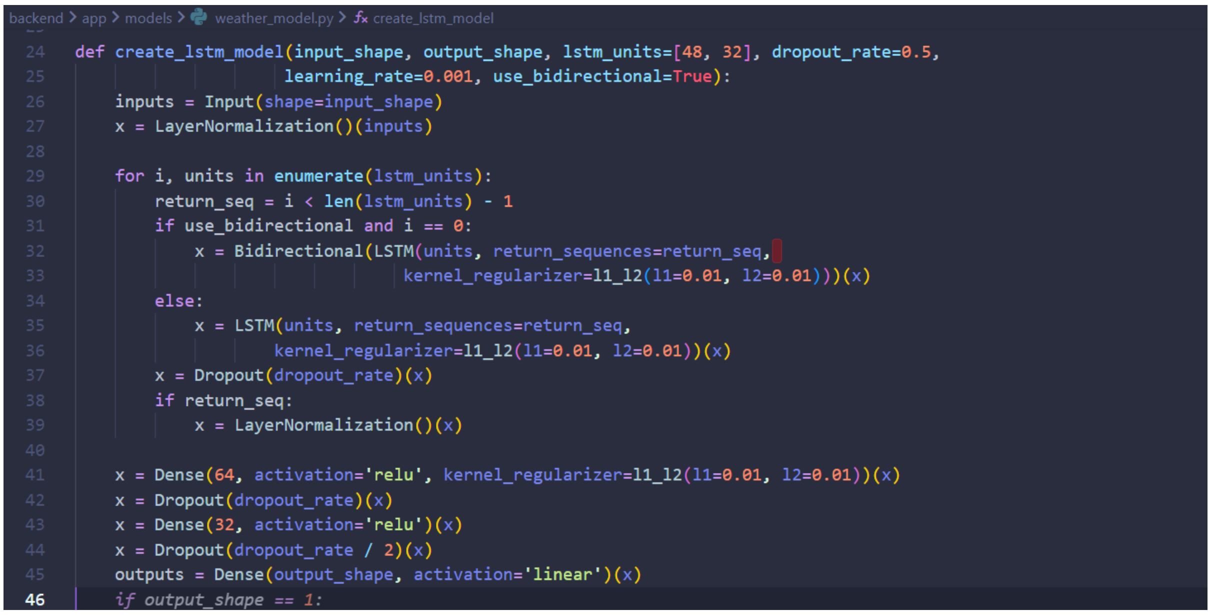


backend > app > services > data_preprocessing.py > preprocess_data_for_training

```python
def preprocess_data_for_training(raw_df, target_columns, look_back_steps=24*7,
                                  look_ahead_steps=1, scaler_type='robust',
                                  feature_scaler=None, target_scaler=None):
    df = raw_df.copy()

    if 'time' in df.columns:
        df['time'] = pd.to_datetime(df['time'])
        df = df.set_index('time')

    df = df.sort_index()
    df = df[~df.index.duplicated(keep='first')]

    df.columns = [col.replace('.', '_') for col in df.columns]

    core_features = [
        'temperature', 'feels_like', 'wind_chill', 'wind_speed', 'wind_angle',
        'pressure', 'humidity', 'cloud_cover_total', 'ozone'
    ]

    available_features = [f for f in core_features if f in df.columns]

    for col in target_columns:
        if col not in available_features and col in df.columns:
            available_features.append(col)
```

This function sets the time column as index, selects the relevant features needed for my model. Handles missing values by forward/backward fill, missing values and adds lag and temporal features.
It also initializes scaling using the RobustScaler. This scaler is robust to outliers. Its scaling features use the interquartile range (IQR) and the median, rather than the mean and standard deviation.
Finally it calls sequences to get X and y.

Weather_model.py: This is the LSTM model, which does all the training, evaluation and prediction logics.


backend > app > models > weather_model.py > create_lstm_model

```python
def create_lstm_model(input_shape, output_shape, lstm_units=[48, 32], dropout_rate=0.5,
                      learning_rate=0.001, use_bidirectional=True):
    inputs = Input(shape=input_shape)
    x = LayerNormalization()(inputs)

    for i, units in enumerate(lstm_units):
        return_seq = i < len(lstm_units) - 1
        if use_bidirectional and i == 0:
            x = Bidirectional(LSTM(units, return_sequences=return_seq,
                                   kernel_regularizer=l1_l2(l1=0.01, l2=0.01)))(x)
        else:
            x = LSTM(units, return_sequences=return_seq,
                     kernel_regularizer=l1_l2(l1=0.01, l2=0.01))(x)
        x = Dropout(dropout_rate)(x)
        if return_seq:
            x = LayerNormalization()(x)

    x = Dense(64, activation='relu', kernel_regularizer=l1_l2(l1=0.01, l2=0.01))(x)
    x = Dropout(dropout_rate)(x)
    x = Dense(32, activation='relu')(x)
    x = Dropout(dropout_rate / 2)(x)
    outputs = Dense(output_shape, activation='linear')(x)
    if output_shape == 1:
```

This defines my LSTM model
It begins with an Input layer, followed by LayerNormalization to stabilize activations by normalizing inputs across features for each sample, which creates faster and stable training. The core of the network consists of stacked Long Short-Term Memory (LSTM) layers, where the first layer is Bidirectional. This means it processes the input sequence both forwards and backward, allowing it to capture dependencies from both past and future contexts. There are 2 layers with the 1st one having 48 neurons and the 2nd one with 32 neurons. The kernel_regularizer=l1_l2(l1=0.01, l2=0.01) penalizes large weights to prevent overfitting.

The LSTM layers internally leverage a sophisticated gating mechanism—comprising of

- **Forget gate**

$$f^{(t)} = \sigma\left(W_f \begin{bmatrix} x^{(t)} \\ h^{(t-1)} \end{bmatrix} + b_f\right)$$

- **Input Gate**

$$i^{(t)} = \sigma\left(W_i \begin{bmatrix} x^{(t)} \\ h^{(t-1)} \end{bmatrix} + b_i\right)$$

- **Candidate Cell State:** Cell with the highest value.

$$C_c^{(t)} = tanh\left(W_C \begin{bmatrix} x^{(t)} \\ h^{(t-1)} \end{bmatrix} + b_C\right)$$

- **Output Gate**

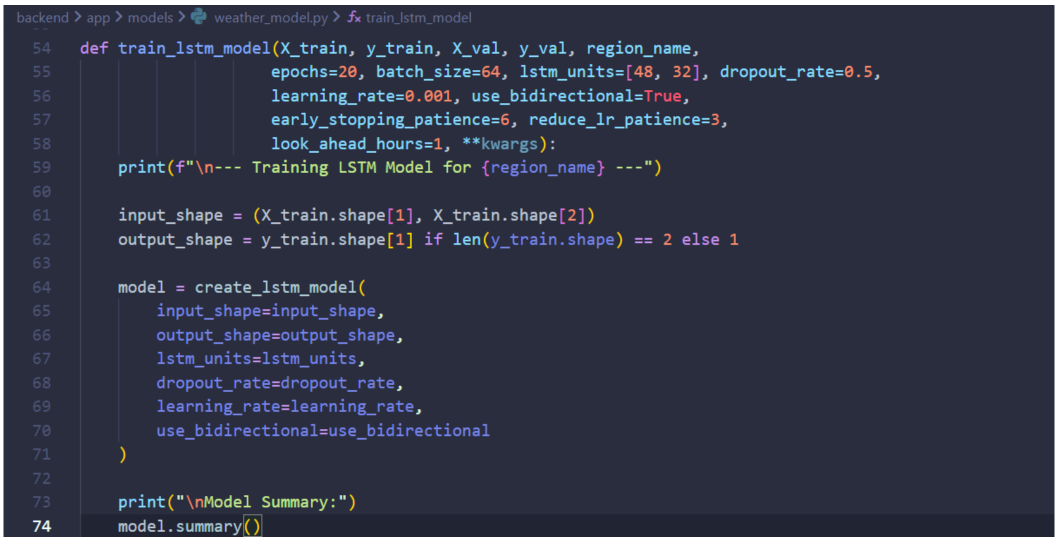
$$O^{(t)} = \sigma\left(W_O \begin{bmatrix} x^{(t)} \\ h^{(t-1)} \end{bmatrix} + b_O\right)$$

These controls in and out of the cell state which follows the equation.

$$C^{(t)} = f^{(t)}C^{(t-1)} + i^{(t)}C_c^{(t)}$$

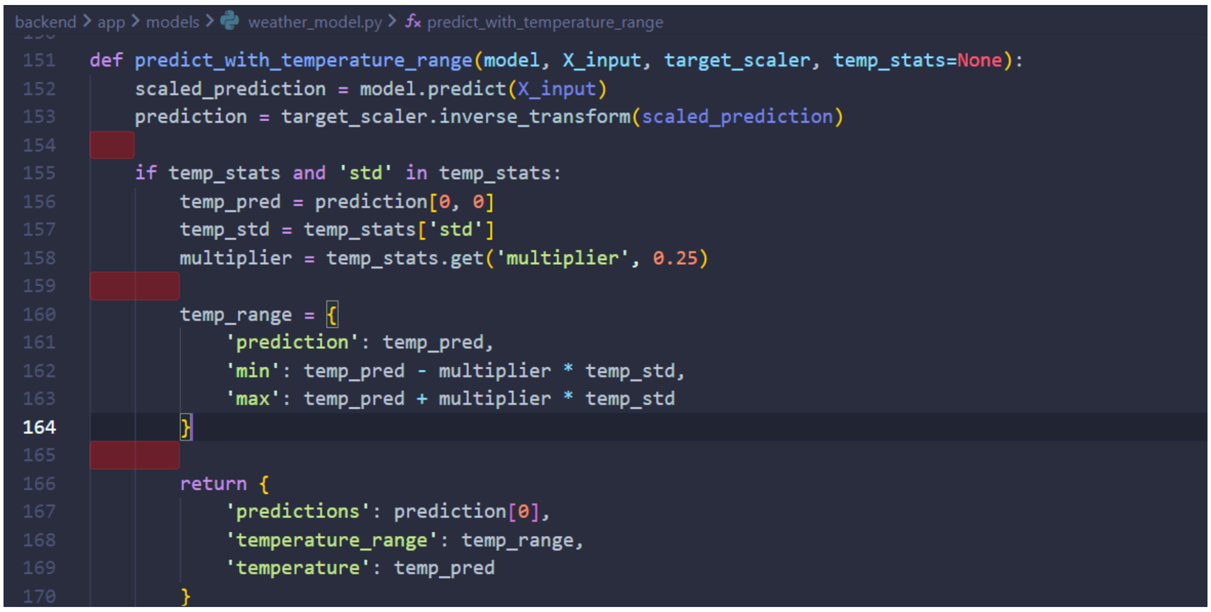
The hyperbolic tangent (tanh) activation functions within the gates, is fundamental for LSTM's vanishing gradient problem. It works by allowing gradients to flow more directly through time, preserving long-term memory.

The return_sequences=True is used for intermediate LSTM layers to make sure 3D output (batch, timesteps, features) is passed to the next recurrent layer, while the final LSTM layer returns only the last output (2D output) to feed into the dense layers.
Dropout layers are placed after each LSTM and dense layer to randomly deactivate a fraction of neurons during training, further preventing overfitting by forcing the network to learn more robust features.
The model includes two Dense layers with relu activation (64 and 32 units respectively), which perform non-linear transformations, before a final Dense layer with output_shape units and a linear activation function outputs the raw numerical predictions for the required weather variable. The model is compiled with the Adam optimizer, which can adjust learning rates for each parameter, and includes clipnorm=1.0 to prevent exploding gradients by reducing the gradient magnitudes during backpropagation.


backend > app > models > weather_model.py > train_lstm_model

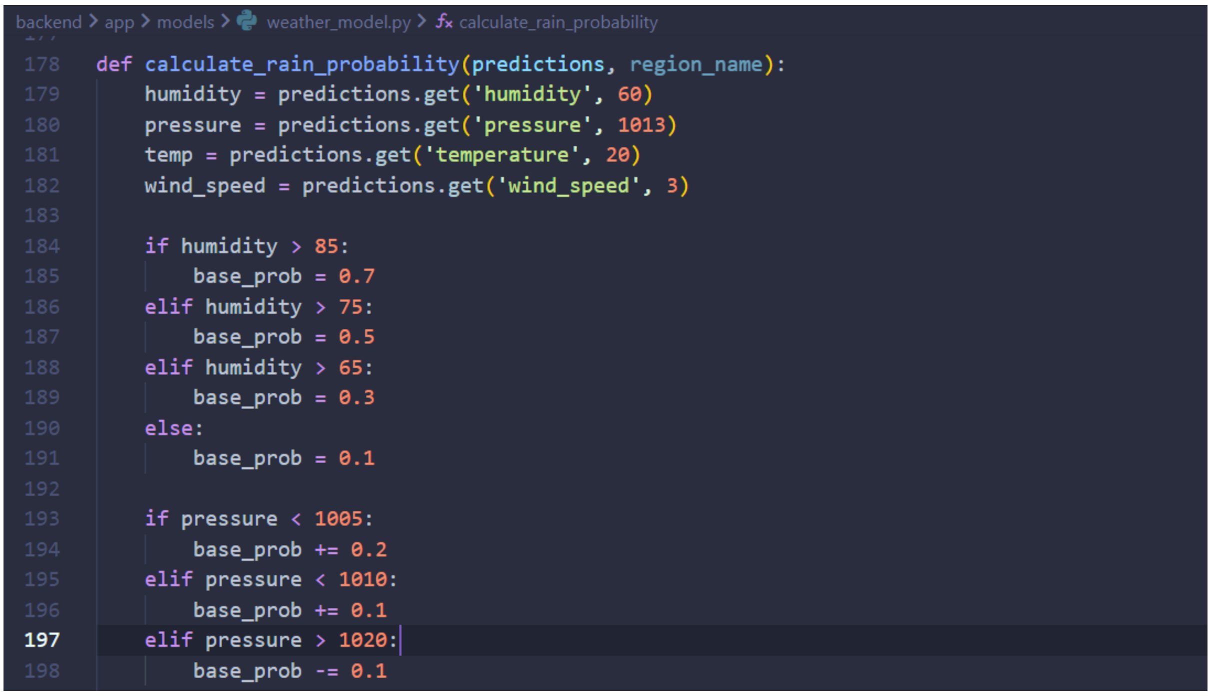
```python
def train_lstm_model(X_train, y_train, X_val, y_val, region_name,
                     epochs=20, batch_size=64, lstm_units=[48, 32], dropout_rate=0.5,
                     learning_rate=0.001, use_bidirectional=True,
                     early_stopping_patience=6, reduce_lr_patience=3,
                     look_ahead_hours=1, **kwargs):
    print(f"\n--- Training LSTM Model for {region_name} ---")

    input_shape = (X_train.shape[1], X_train.shape[2])
    output_shape = y_train.shape[1] if len(y_train.shape) == 2 else 1

    model = create_lstm_model(
        input_shape=input_shape,
        output_shape=output_shape,
        lstm_units=lstm_units,
        dropout_rate=dropout_rate,
        learning_rate=learning_rate,
        use_bidirectional=use_bidirectional
    )

    print("\nModel Summary:")
    model.summary()
```

Trains the LSTM model using the prepared data and saves the trained model and associated parameters.


backend > app > models > weather_model.py > predict_with_temperature_range

```python
def predict_with_temperature_range(model, X_input, target_scaler, temp_stats=None):
    scaled_prediction = model.predict(X_input)
    prediction = target_scaler.inverse_transform(scaled_prediction)

    if temp_stats and 'std' in temp_stats:
        temp_pred = prediction[0, 0]
        temp_std = temp_stats['std']
        multiplier = temp_stats.get('multiplier', 0.25)

        temp_range = {
            'prediction': temp_pred,
            'min': temp_pred - multiplier * temp_std,
            'max': temp_pred + multiplier * temp_std
        }

        return {
            'predictions': prediction[0],
            'temperature_range': temp_range,
            'temperature': temp_pred
        }
```

For the maximum and minimum temperature predictions I have used a manual technique to use the data from the last 6 months to find the standard deviation and then use that for the possible max and min temperature. The max will be my_predicted _emperature + 0.25*std and the min will be my_predicted _emperature - 0.25*std.


backend > app > models > weather_model.py > calculate_rain_probability

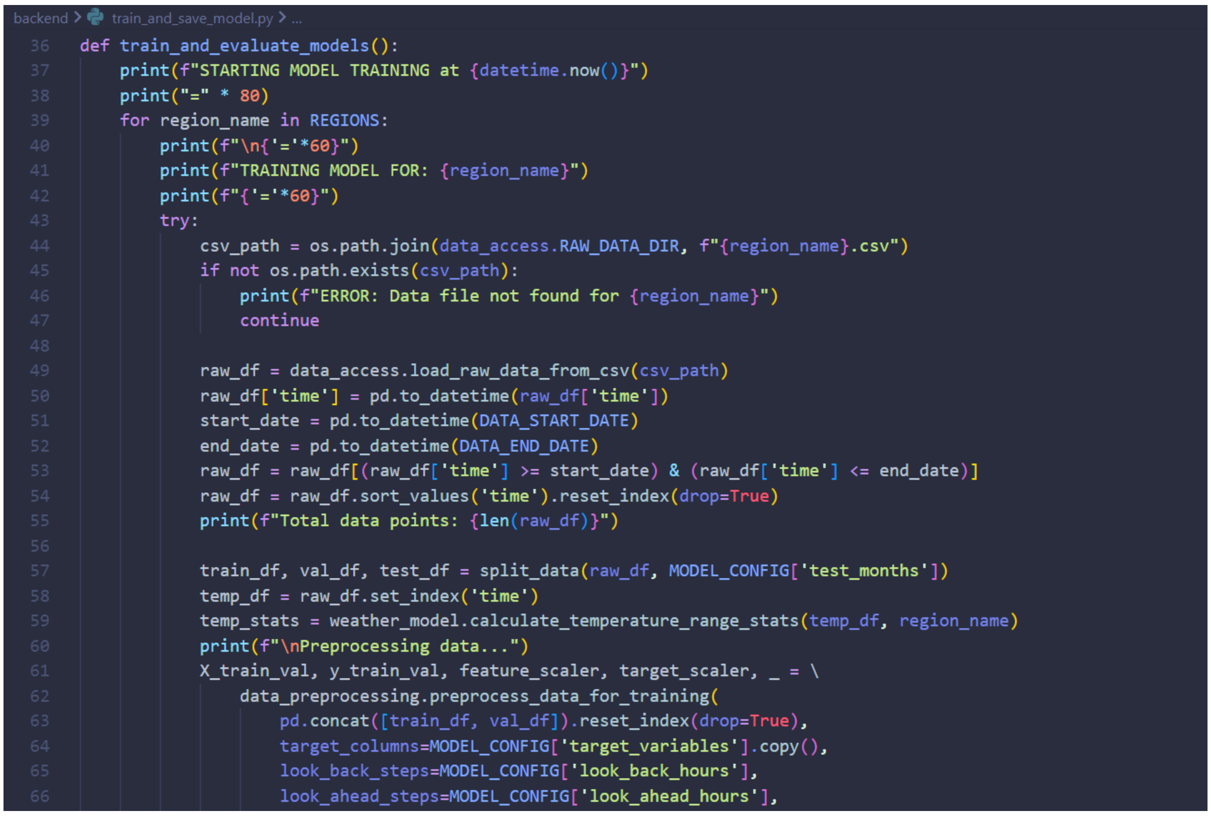
```python
def calculate_rain_probability(predictions, region_name):
    humidity = predictions.get('humidity', 60)
    pressure = predictions.get('pressure', 1013)
    temp = predictions.get('temperature', 20)
    wind_speed = predictions.get('wind_speed', 3)

    if humidity > 85:
        base_prob = 0.7
    elif humidity > 75:
        base_prob = 0.5
    elif humidity > 65:
        base_prob = 0.3
    else:
        base_prob = 0.1

    if pressure < 1005:
        base_prob += 0.2
    elif pressure < 1010:
        base_prob += 0.1
    elif pressure > 1020:
        base_prob -= 0.1
```

I have also made a manual rain probability function which gives the probability for rain in a certain day. It takes predicted humidity, pressure, temperature, and wind_speed as input. It then applies a series of if/elif conditions to adjust a base_prob based on these values. For example, higher humidity increases base_prob, lower pressure increases it, very high temperature decreases it, and high wind speed slightly increases it. The final probability is clipped between 0 and 1.

The rest of the functions are just bias corrections, starting the evaluation and making the plots.

Train_and_save_model.py: This script is the entry point for the training and evaluation. This is where all the packages are used together to train my model.

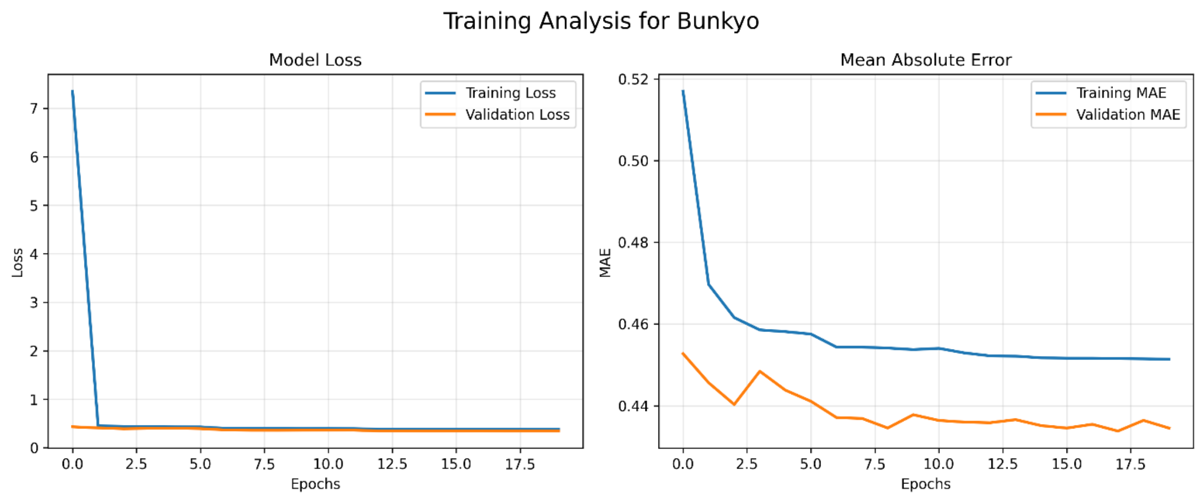
It splits the raw historical Datafame into training, validation, and test sets based on time.

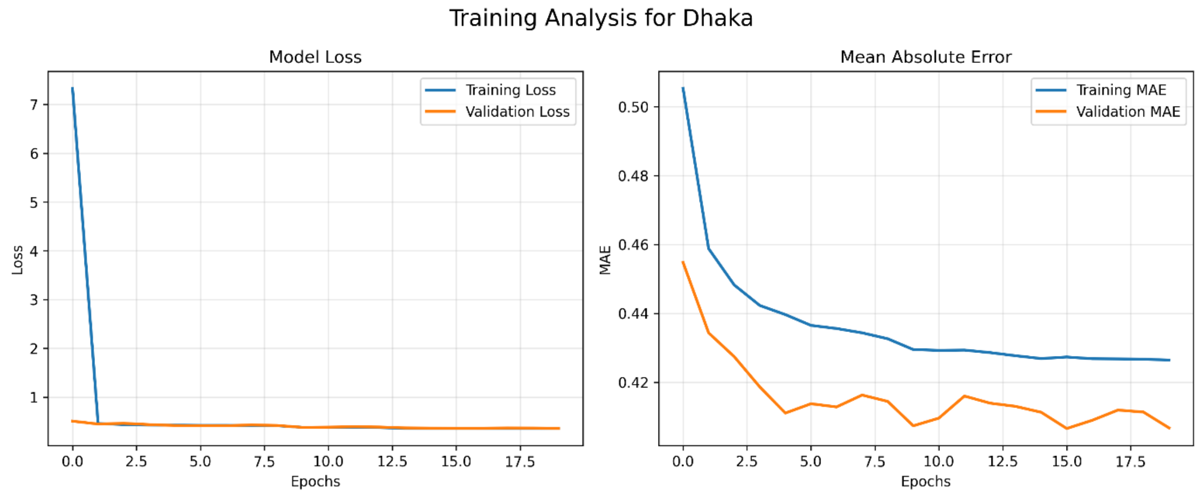


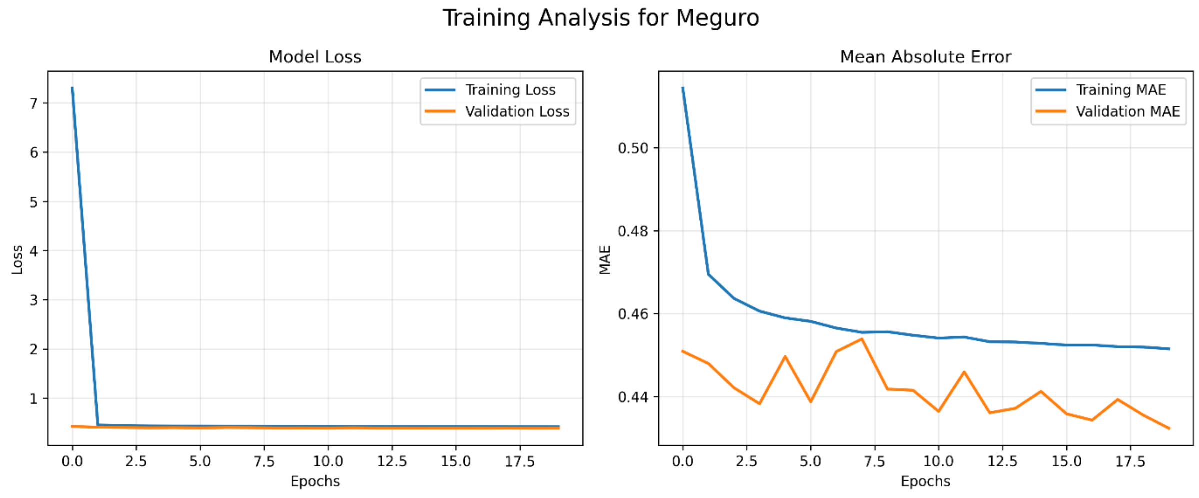
```python
def train_and_evaluate_models():
    print(f"STARTING MODEL TRAINING at {datetime.now()}")
    print("=" * 80)
    for region_name in REGIONS:
        print(f"\n{'='*60}")
        print(f"TRAINING MODEL FOR: {region_name}")
        print(f"{'='*60}")
        try:
            csv_path = os.path.join(data_access.RAW_DATA_DIR, f"{region_name}.csv")
            if not os.path.exists(csv_path):
                print(f"ERROR: Data file not found for {region_name}")
                continue

            raw_df = data_access.load_raw_data_from_csv(csv_path)
            raw_df['time'] = pd.to_datetime(raw_df['time'])
            start_date = pd.to_datetime(DATA_START_DATE)
            end_date = pd.to_datetime(DATA_END_DATE)
            raw_df = raw_df[(raw_df['time'] >= start_date) & (raw_df['time'] <= end_date)]
            raw_df = raw_df.sort_values('time').reset_index(drop=True)
            print(f"Total data points: {len(raw_df)}")

            train_df, val_df, test_df = split_data(raw_df, MODEL_CONFIG['test_months'])
            temp_df = raw_df.set_index('time')
            temp_stats = weather_model.calculate_temperature_range_stats(temp_df, region_name)
            print(f"\nPreprocessing data...")
            X_train_val, y_train_val, feature_scaler, target_scaler, _ = \
                data_preprocessing.preprocess_data_for_training(
                    pd.concat([train_df, val_df]).reset_index(drop=True),
                    target_columns=MODEL_CONFIG['target_variables'].copy(),
                    look_back_steps=MODEL_CONFIG['look_back_hours'],
                    look_ahead_steps=MODEL_CONFIG['look_ahead_hours'],
```

At first it iterates through each regions, then splits them into training, validation and test data. Then it calls the previously explained data_preprocessing.preprocess_data_for_training function to prepare the dataset. Reshapes the dataset to match the expected output shape of the Dense layer for multi-output prediction. Then does a second split, specifically for the training process's internal validation. Then calls the weather_model.train_lstm_model function to train the model for the current region, passing all relevant MODEL_CONFIG parameters from the config.py package.

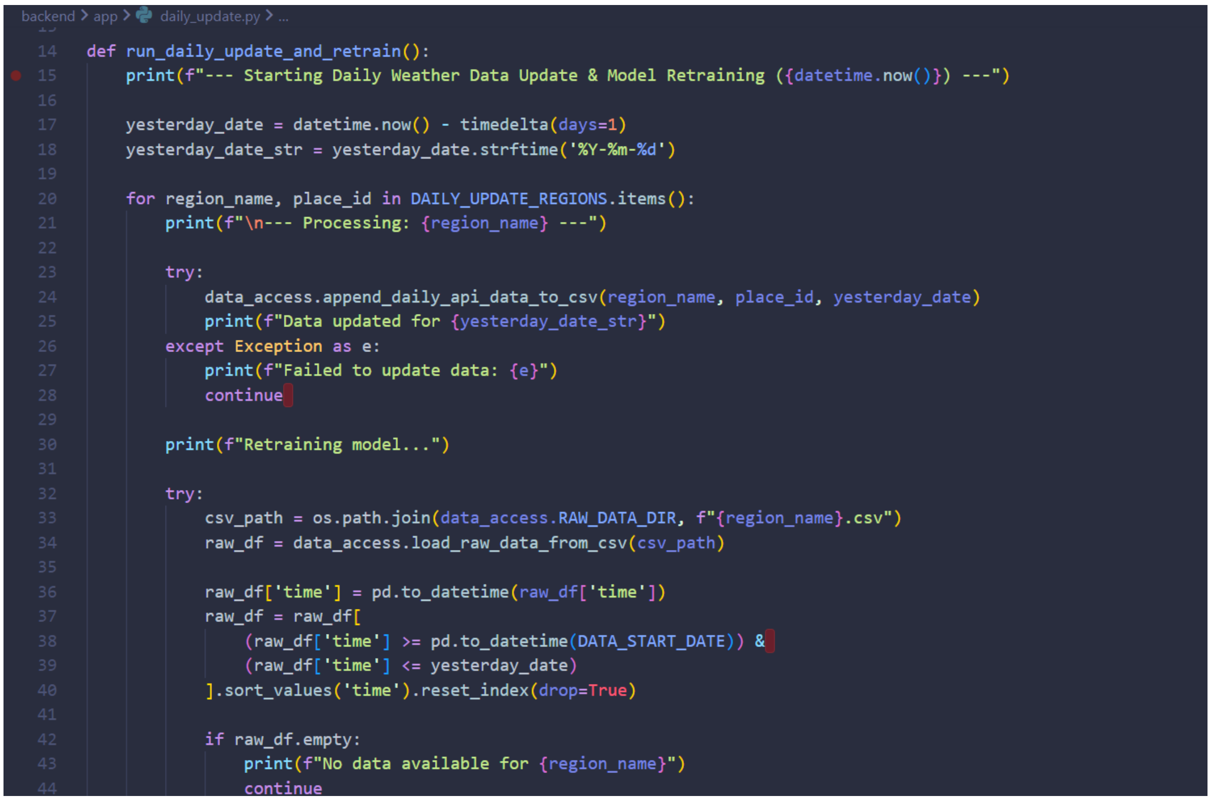
Then it starts testing of the datatest. And does the evaluation. Below are my evaluation plots.


Training Analysis for Bunkyo
Model Loss
Training Loss
Validation Loss
Loss
Epochs
Mean Absolute Error
Training MAE
Validation MAE
MAE
Epochs


Training Analysis for Dhaka
Model Loss
Training Loss
Validation Loss
Loss
Epochs
Mean Absolute Error
Training MAE
Validation MAE
MAE
Epochs


Training Analysis for Meguro
Model Loss
Training Loss
Validation Loss
Loss
Epochs
Mean Absolute Error
Training MAE
Validation MAE
MAE
Epochs

Daily_update.py: This package is designed to be run periodically (e.g., daily) to fetch the latest weather data and keep models up-to-date. I have set my OS's Task Scheduler to run this package every night at 3am.

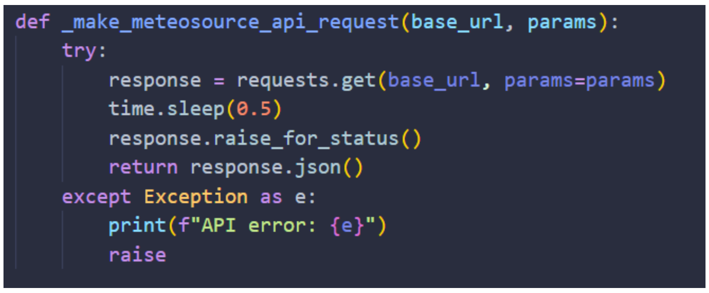


```
backend > app > daily_update.py > ...
def run_daily_update_and_retrain():
    print(f"--- Starting Daily Weather Data Update & Model Retraining ({datetime.now()}) ---")

    yesterday_date = datetime.now() - timedelta(days=1)
    yesterday_date_str = yesterday_date.strftime('%Y-%m-%d')

    for region_name, place_id in DAILY_UPDATE_REGIONS.items():
        print(f"\n--- Processing: {region_name} ---")

        try:
            data_access.append_daily_api_data_to_csv(region_name, place_id, yesterday_date)
            print(f"Data updated for {yesterday_date_str}")
        except Exception as e:
            print(f"Failed to update data: {e}")
            continue

        print(f"Retraining model...")

        try:
            csv_path = os.path.join(data_access.RAW_DATA_DIR, f"{region_name}.csv")
            raw_df = data_access.load_raw_data_from_csv(csv_path)

            raw_df['time'] = pd.to_datetime(raw_df['time'])
            raw_df = raw_df[
                (raw_df['time'] >= pd.to_datetime(DATA_START_DATE)) &
                (raw_df['time'] <= yesterday_date)
            ].sort_values('time').reset_index(drop=True)

            if raw_df.empty:
                print(f"No data available for {region_name}")
                continue
```

At first it find yesterday's date to fetch the most recent full day's data. Calls data_access.append_daily_api_data_to_csv function to fetch and append new data for the region. Then calls data_preprocessing.preprocess_data_for_training function on the entire available dataset (up to yesterday) to prepare X and y for retraining. Then Splits the preprocessed X and y into training and validation sets (e.g., 80/10 split) for the retraining process.

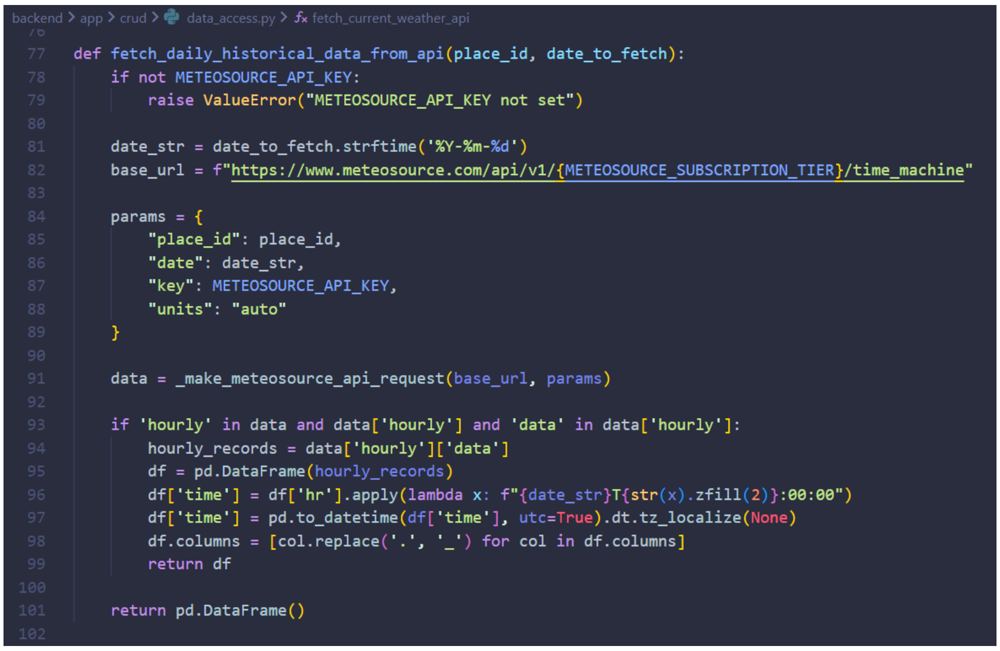
Data_access.py: This package handles all interactions with the raw data, whether it's loading from local CSVs or fetching from external APIs.

In the first 2 function it simply just reads and loads data for a specific region with a specific time.



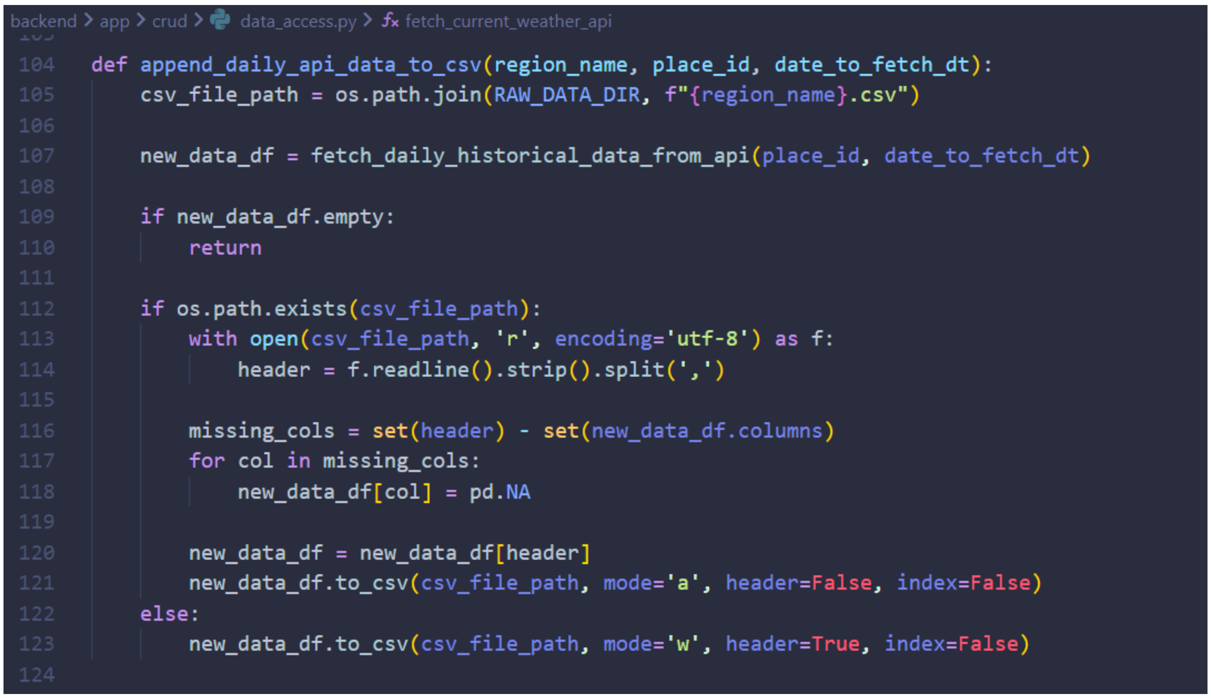
```python
def _make_meteosource_api_request(base_url, params):
    try:
        response = requests.get(base_url, params=params)
        time.sleep(0.5)
        response.raise_for_status()
        return response.json()
    except Exception as e:
        print(f"API error: {e}")
        raise
```

This is a helper function to make HTTP GET requests to the Meteosource API which I am using for my dataset. I have also added delays so not to overload the API.


backend > app > crud > data_access.py > fetch_current_weather_api

```python
def fetch_daily_historical_data_from_api(place_id, date_to_fetch):
    if not METEOSOURCE_API_KEY:
        raise ValueError("METEOSOURCE_API_KEY not set")

    date_str = date_to_fetch.strftime('%Y-%m-%d')
    base_url = f"https://www.meteosource.com/api/v1/{METEOSOURCE_SUBSCRIPTION_TIER}/time_machine"

    params = {
        "place_id": place_id,
        "date": date_str,
        "key": METEOSOURCE_API_KEY,
        "units": "auto"
    }

    data = _make_meteosource_api_request(base_url, params)

    if 'hourly' in data and data['hourly'] and 'data' in data['hourly']:
        hourly_records = data['hourly']['data']
        df = pd.DataFrame(hourly_records)
        df['time'] = df['hr'].apply(lambda x: f"{date_str}T{str(x).zfill(2)}:00:00")
        df['time'] = pd.to_datetime(df['time'], utc=True).dt.tz_localize(None)
        df.columns = [col.replace('.', '_') for col in df.columns]
        return df

    return pd.DataFrame()
```

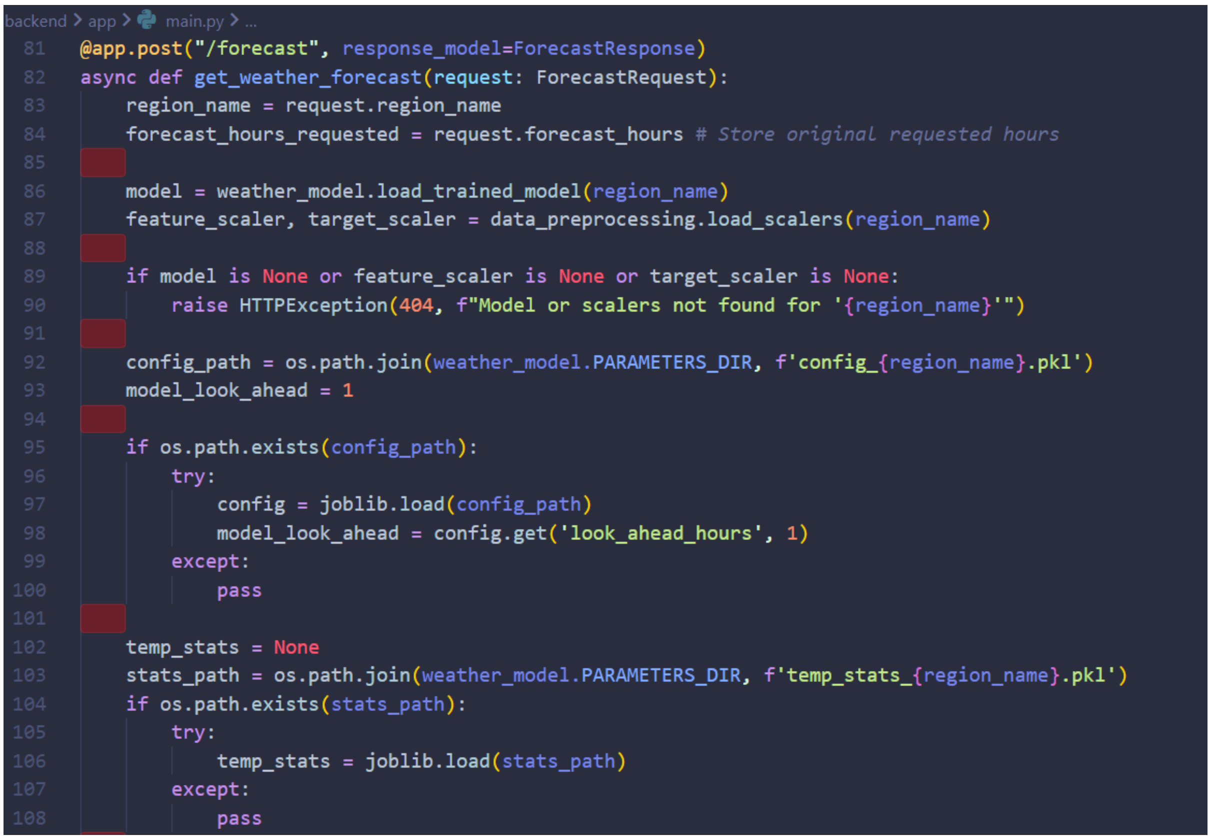
This function fetches historical hourly weather data for a specific place_id and date from the Meteosource API. It constructs the API URL using the required parameters and makes the call.


backend > app > crud > data_access.py > fetch_current_weather_api

```python
def append_daily_api_data_to_csv(region_name, place_id, date_to_fetch_dt):
    csv_file_path = os.path.join(RAW_DATA_DIR, f"{region_name}.csv")

    new_data_df = fetch_daily_historical_data_from_api(place_id, date_to_fetch_dt)

    if new_data_df.empty:
        return

    if os.path.exists(csv_file_path):
        with open(csv_file_path, 'r', encoding='utf-8') as f:
            header = f.readline().strip().split(',')

        missing_cols = set(header) - set(new_data_df.columns)
        for col in missing_cols:
            new_data_df[col] = pd.NA

        new_data_df = new_data_df[header]
        new_data_df.to_csv(csv_file_path, mode='a', header=False, index=False)
    else:
        new_data_df.to_csv(csv_file_path, mode='w', header=True, index=False)
```

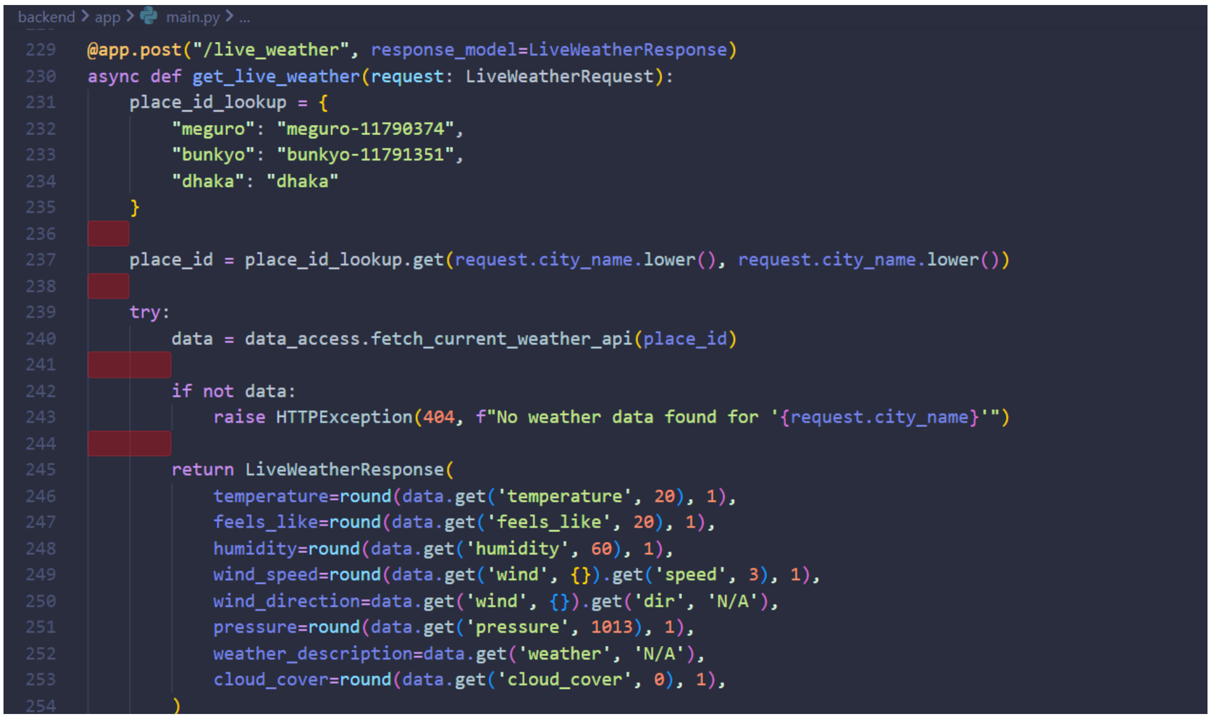
This function appends newly fetched daily historical data to the existing CSV file for a given region. It reads the header to ensure new data columns match the existing ones, filling missing columns with pd.NA. Then, it appends the new data to the CSV without writing the header again.

Backend Server: This file sets up a web API using FastAPI, allowing others to request weather forecasts and live weather data. It can request live weather requests for anywhere in the world and also request data from my Model.


backend > app > main.py > ...

```python
@app.post("/forecast", response_model=ForecastResponse)
async def get_weather_forecast(request: ForecastRequest):
    region_name = request.region_name
    forecast_hours_requested = request.forecast_hours # Store original requested hours

    model = weather_model.load_trained_model(region_name)
    feature_scaler, target_scaler = data_preprocessing.load_scalers(region_name)

    if model is None or feature_scaler is None or target_scaler is None:
        raise HTTPException(404, f"Model or scalers not found for '{region_name}'")

    config_path = os.path.join(weather_model.PARAMETERS_DIR, f'config_{region_name}.pkl')
    model_look_ahead = 1

    if os.path.exists(config_path):
        try:
            config = joblib.load(config_path)
            model_look_ahead = config.get('look_ahead_hours', 1)
        except:
            pass

    temp_stats = None
    stats_path = os.path.join(weather_model.PARAMETERS_DIR, f'temp_stats_{region_name}.pkl')
    if os.path.exists(stats_path):
        try:
            temp_stats = joblib.load(stats_path)
        except:
            pass
```

This is the main endpoint for requesting weather forecasts. At first it receives a Forecast Request with the region and forecast hour. It then loads the config_{region_name}.pkl and temp_stats_{region_name}.pkl. It then applies the bias corrections to them. Then collects the temperature range and the rain probability. Finally it gathers all predicted values, ranges, rain info, and metadata into a Response object.



```python
@app.post("/live_weather", response_model=LiveWeatherResponse)
async def get_live_weather(request: LiveWeatherRequest):
    place_id_lookup = {
        "meguro": "meguro-11790374",
        "bunkyo": "bunkyo-11791351",
        "dhaka": "dhaka"
    }

    place_id = place_id_lookup.get(request.city_name.lower(), request.city_name.lower())

    try:
        data = data_access.fetch_current_weather_api(place_id)

        if not data:
            raise HTTPException(404, f"No weather data found for '{request.city_name}'")

        return LiveWeatherResponse(
            temperature=round(data.get('temperature', 20), 1),
            feels_like=round(data.get('feels_like', 20), 1),
            humidity=round(data.get('humidity', 60), 1),
            wind_speed=round(data.get('wind', {}).get('speed', 3), 1),
            wind_direction=data.get('wind', {}).get('dir', 'N/A'),
            pressure=round(data.get('pressure', 1013), 1),
            weather_description=data.get('weather', 'N/A'),
            cloud_cover=round(data.get('cloud_cover', 0), 1),
        )
```

This is the endpoint to fetch live weather from the Meteosource API.
It maps the city_name from the request to a place_id. Then calls data_access.fetch_current_weather_api to get live data from Meteosource, and then finally constructs a Response object with the fetched data.