Collab link for reference :

https://colab.research.google.com/drive/1cq6JPMeAXZU-qhX_qfncWJOfXXNgdr7E?usp=sharing

Weather Prediction System for Chennai

Mithra D K

23B2223

Source of the data: Google earth engine (I HAVE USED A DIFFERENT DATASET FROM WHAT I HAVE SUBMITTED IN STAGE 2)

Source of the code: ME 228 Tutorials, Perplexity, Google

```
import pandas as pd
import numpy as np
import plotly.express as px
import plotly.graph_objects as go
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import TimeSeriesSplit
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
import plotly.graph_objects as go
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
from xgboost import XGBRegressor
from sklearn.ensemble import StackingRegressor
from sklearn.linear_model import Ridge
```

1.Load and inspect the data

```
# Step 1: Load & Inspect the Data
# Reading the CSV file containing daily weather data for Chennai from 2020-2023
# The file contains maximum, minimum, and mean temperatures along with precipitation data
file_path = "Chennai_ERA5Land_Daily.csv" # Source file with daily weather measurements
df = pd.read_csv(file_path, parse_dates=["date"])
df.set_index("date", inplace=True)
df.sort_index(inplace=True)

print("1) Data Info:")
print(df.info())
print("\nFirst 5 rows:")
print(df.head())
```

```
1) Data Info:
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 1460 entries, 2020-01-01 to 2023-12-30
Data columns (total 4 columns):
 #   Column     Non-Null Count  Dtype  
---  ------     --------------  -----  
 0   tmax_C     1460 non-null   float64
 1   tmin_C     1460 non-null   float64
 2   tmean_C    1460 non-null   float64
 3   precip_mm  1460 non-null   float64
dtypes: float64(4)
memory usage: 57.0 KB
None

First 5 rows:
               tmax_C     tmin_C    tmean_C  precip_mm
date                                                  
2020-01-01  26.899193  22.995536  24.933332   6.755953
2020-01-02  27.673941  22.956217  25.258233   1.884843
2020-01-03  28.884421  22.827521  25.899719   1.803642
2020-01-04  29.315849  23.353994  26.112648   1.544545
2020-01-05  28.213099  23.140409  25.524729   2.988680
```

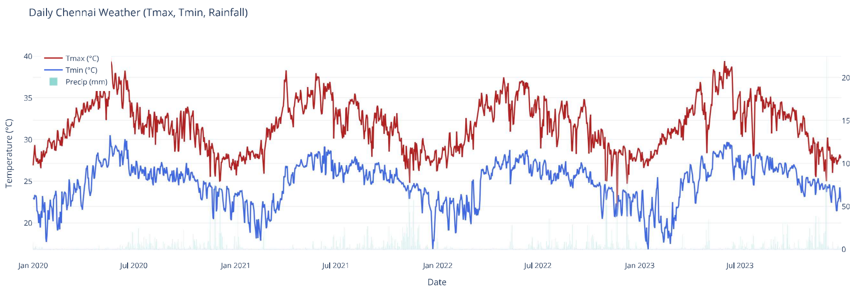
2.Clean and Preprocess

```
# Step 2: Clean & Preprocess
# 2.1 Check missing values
missing = df.isna().sum()
print("\nMissing values per column:")
print(missing)

# 2.2 Fill small gaps by time interpolation
df[["tmax_C","tmin_C","tmean_C","precip_mm"]] = (
    df[["tmax_C","tmin_C","tmean_C","precip_mm"]]
    .interpolate(method="time")
)

# 2.3 Drop any remaining NaNs or duplicate dates
df = df.dropna().loc[~df.index.duplicated()]
```

```
Missing values per column:
tmax_C       0
tmin_C       0
tmean_C      0
precip_mm    0
dtype: int64
```

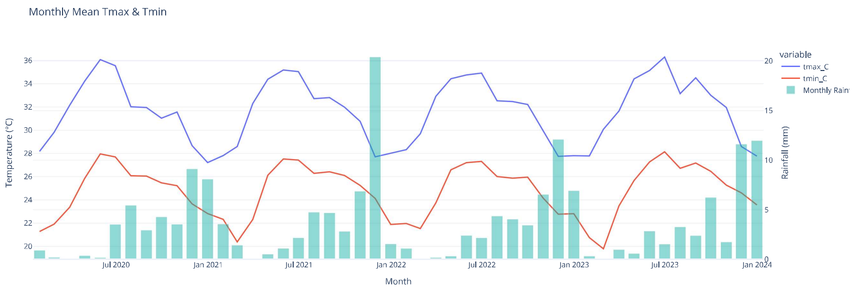
3.Exploratory Data Analysis (EDA) using Plotly

```
#Step 3:: 3.1 Daily time series: Max/Min Temp & Rainfall
fig = go.Figure()
fig.add_trace(go.Scatter(
    x=df.index, y=df["tmax_C"],
    mode="lines", name="Tmax (°C)",
    line=dict(color="firebrick")
))
fig.add_trace(go.Scatter(
    x=df.index, y=df["tmin_C"],
    mode="lines", name="Tmin (°C)",
    line=dict(color="royalblue")
))
fig.add_trace(go.Bar(
    x=df.index, y=df["precip_mm"],
    name="Precip (mm)", yaxis="y2",
    marker_color="lightseagreen", opacity=0.5
))
fig.update_layout(
    title="Daily Chennai Weather (Tmax, Tmin, Rainfall)",
    xaxis_title="Date",
    yaxis=dict(title="Temperature (°C)"),
    yaxis2=dict(
        title="Rainfall (mm)",
        overlaying="y", side="right"
    ),
    legend=dict(x=0.01, y=0.99),
    hovermode="x unified",
    template="plotly_white"
)
fig.show()
```

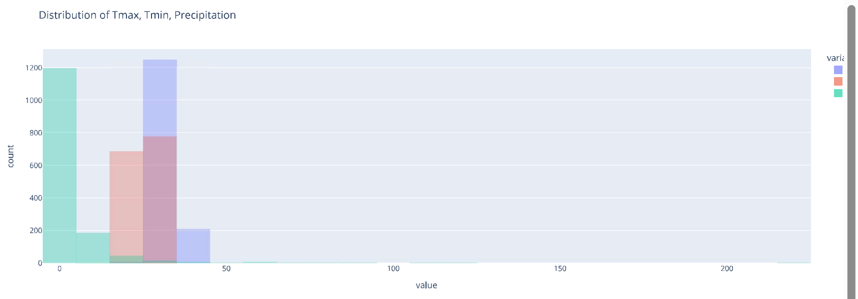


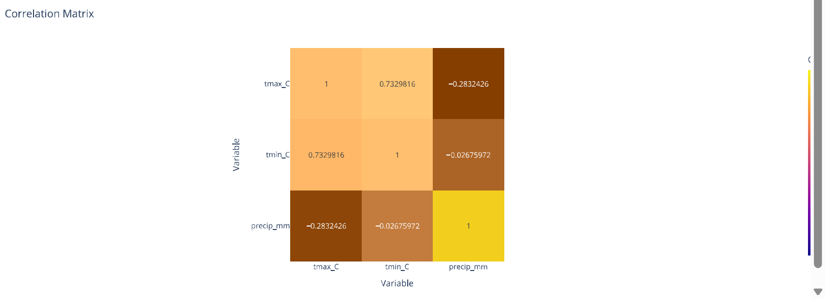
```
# 3.2 Monthly averages
monthly = df.resample("M").mean().reset_index()
fig2 = px.line(
    monthly, x="date", y=["tmax_C","tmin_C"],
    labels={"value":"Temperature (°C)","date":"Month"},
    title="Monthly Mean Tmax & Tmin"
)
fig2.add_bar(
    x=monthly["date"], y=monthly["precip_mm"],
    name="Monthly Rainfall (mm)", yaxis="y2",
    marker_color="lightseagreen", opacity=0.5
)
fig2.update_layout(
    yaxis2=dict(
        title="Rainfall (mm)", overlaying="y", side="right"
    ),
    hovermode="x unified",
    template="plotly_white"
)
fig2.show()
```

```
<ipython-input-29-1708a56bdeb2>:2: FutureWarning:

'M' is deprecated and will be removed in a future version, please use 'ME' instead.
```



```
# 3.3 Distributions & Correlation
# Histogram
fig3 = px.histogram(
    df, x=["tmax_C","tmin_C","precip_mm"],
    nbins=30, barmode="overlay",
    title="Distribution of Tmax, Tmin, Precipitation"
)
fig3.update_traces(opacity=0.6)
fig3.show()

# Correlation heatmap
corr = df[["tmax_C","tmin_C","precip_mm"]].corr()
fig4 = px.imshow(
    corr, text_auto=True, zmin=-1, zmax=1,
    labels=dict(x="Variable", y="Variable", color="Correlation"),
    title="Correlation Matrix"
)
fig4.show()
```





	tmax_C	tmin_C	precip_mm
tmax_C	1	0.7329816	−0.2832426
tmin_C	0.7329816	1	−0.02675972
precip_mm	−0.2832426	−0.02675972	1

Analysis of Daily Weather Patterns

The visualization above shows the daily temperature (max/min) and precipitation patterns for Chennai over the 4-year period. Key observations:

- Temperature follows clear seasonal patterns with higher temperatures during summer months
- Rainfall shows more sporadic behavior with clear monsoon periods
- The temperature range varies throughout the year, with some periods showing larger day/night differences

4.Feature Engineering

```
# assume df is your cleaned DataFrame with index=date and columns tmax_C, tmin_C, tmean_C, precip_mm
def make_features(df):
  X = df.copy()
  # Temporal features - capture seasonal patterns and cyclical weather behaviors
  X['month']       = X.index.month
  X['day_of_year'] = X.index.dayofyear
  # temperature range
  X['temp_range'] = X['tmax_C'] - X['tmin_C']
  # lag features
  for lag in [1, 3, 7]:
     X[f'tmax_lag{lag}']   = X['tmax_C'].shift(lag)
     X[f'tmin_lag{lag}']   = X['tmin_C'].shift(lag)
     X[f'precip_lag{lag}'] = X['precip_mm'].shift(lag)

  # rolling features
  X['tmax_roll7']   = X['tmax_C'].rolling(7).mean()
  X['precip_roll7'] = X['precip_mm'].rolling(7).sum()

  # drop rows with NaNs created by shifting/rolling
  return X.dropna()

features = make_features(df)
```

```
# Define your target variables - you can predict multiple targets
targets = ['tmax_C', 'tmin_C', 'precip_mm']

# For each target, train a separate model
for target in targets:
    print(f"\nTraining model for {target}")

    # Define your target variable 'y'
    y = features[target]

    # Define your features 'X' - drop all original weather variables
    X = features.drop(columns=['tmax_C', 'tmin_C', 'tmean_C', 'precip_mm'])
```

```
Training model for tmax_C

Training model for tmin_C

Training model for precip_mm
```

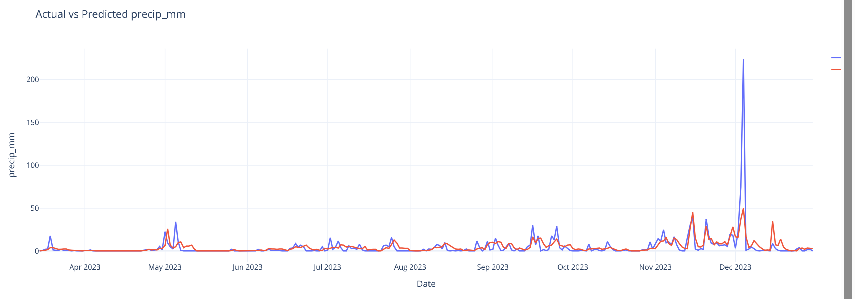
```
# Use the last 20% of the data as test set
split_idx = int(len(X) * 0.8)
X_train, X_test = X.iloc[:split_idx], X.iloc[split_idx:]
y_train, y_test = y.iloc[:split_idx], y.iloc[split_idx:]

print(f"Training data: {X_train.shape[0]} samples")
print(f"Test data: {X_test.shape[0]} samples")
```

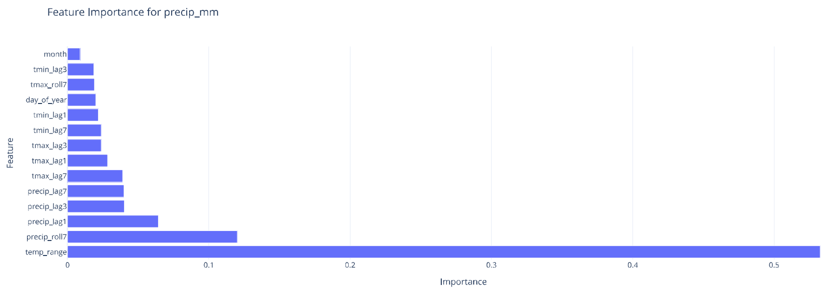
```
Training data: 1162 samples
Test data: 291 samples
```

(Using Random forest regression)

```
# Define and train the model
model = RandomForestRegressor(n_estimators=100, random_state=42)
model.fit(X_train, y_train)

    # Make predictions
y_pred = model.predict(X_test)

    # Calculate regression metrics
mae = mean_absolute_error(y_test, y_pred)
mse = mean_squared_error(y_test, y_pred)
rmse = np.sqrt(mse)
r2 = r2_score(y_test, y_pred)

    # Print the evaluation metrics
print(f"Mean Absolute Error (MAE): {mae:.2f}")
print(f"Root Mean Squared Error (RMSE): {rmse:.2f}")
print(f"R² Score: {r2:.3f}")

    # Plot actual vs predicted values
fig = go.Figure()
fig.add_trace(go.Scatter(x=y_test.index, y=y_test, mode='lines', name='Actual'))
fig.add_trace(go.Scatter(x=y_test.index, y=y_pred, mode='lines', name='Predicted'))
fig.update_layout(
title=f'Actual vs Predicted {target}',
xaxis_title='Date',
yaxis_title=f'{target}',
template='plotly_white'
)
fig.show()

    # Plot feature importance
feature_importance = pd.DataFrame({
    'Feature': X.columns,
    'Importance': model.feature_importances_
}).sort_values('Importance', ascending=False)

fig = go.Figure(go.Bar(
    x=feature_importance['Importance'],
    y=feature_importance['Feature'],
    orientation='h'
))
fig.update_layout(
    title=f'Feature Importance for {target}',
    xaxis_title='Importance',
    yaxis_title='Feature',
    template='plotly_white'
    )
fig.show()
```

```
Mean Absolute Error (MAE): 3.68
Root Mean Squared Error (RMSE): 11.55
R² Score: 0.401
```





5.Train/Test Split

```
split_date = features.index[int(len(features)*0.8)]
train = features.loc[:split_date]
test  = features.loc[split_date:]

# choose targets
targets = ['tmax_C', 'tmin_C', 'precip_mm']

models = {}
results = {}
```

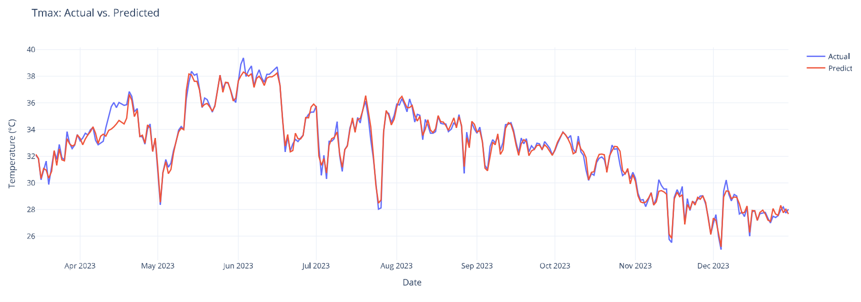
6.Model Training and Evaluation

```
for target in targets:
    X_train = train.drop(targets, axis=1)
    y_train = train[target]
    X_test  = test.drop(targets, axis=1)
    y_test  = test[target]

    # time-series split (5 folds)
    tscv = TimeSeriesSplit(n_splits=5)
    model = RandomForestRegressor(n_estimators=200, random_state=0)

    # train on full training set
    model.fit(X_train, y_train)
    preds = model.predict(X_test)

    # evaluate
    mae  = mean_absolute_error(y_test, preds)
    mse  = mean_squared_error(y_test, preds)  # FIXED: use y_test and preds
    rmse = np.sqrt(mse)
    r2   = r2_score(y_test, preds)
    results[target] = (mae, rmse, r2)
    models[target] = model

    print(f"{target}: MAE={mae:.2f}, RMSE={rmse:.2f}, R²={r2:.3f}")
```

```
tmax_C: MAE=0.28, RMSE=0.40, R²=0.983
tmin_C: MAE=0.30, RMSE=0.39, R²=0.949
precip_mm: MAE=3.35, RMSE=11.10, R²=0.447
```

7.Plot Predicted vs. Actual for Tmax with Plotly

```
pred_df = pd.DataFrame({
    'date': test.index,
    'actual': test['tmax_C'],
    'predicted': models['tmax_C'].predict(test.drop(targets, axis=1))
}).set_index('date')

fig = go.Figure()
fig.add_trace(go.Scatter(x=pred_df.index, y=pred_df['actual'],
                         mode='lines', name='Actual Tmax'))
fig.add_trace(go.Scatter(x=pred_df.index, y=pred_df['predicted'],
                         mode='lines', name='Predicted Tmax'))
fig.update_layout(title='Tmax: Actual vs. Predicted',
                  xaxis_title='Date', yaxis_title='Temperature (°C)',
                  template='plotly_white')
fig.show()
```



Improved version

4.Feature Engineering

```
def make_features(df):
    X = df.copy()

    # temporal features
    X['month'] = X.index.month
    X['day_of_year'] = X.index.dayofyear

    # temperature range
    X['temp_range'] = X['tmax_C'] - X['tmin_C']

    # lag features - keep original lags
    for lag in [1, 3, 7]:
        X[f'tmax_lag{lag}'] = X['tmax_C'].shift(lag)
        X[f'tmin_lag{lag}'] = X['tmin_C'].shift(lag)
        X[f'precip_lag{lag}'] = X['precip_mm'].shift(lag)

    # Add more comprehensive lag features for precipitation
    for lag in [14, 21, 30, 60]:
        X[f'precip_lag{lag}'] = X['precip_mm'].shift(lag)

    # rolling features - keep original ones
    X['tmax_roll7'] = X['tmax_C'].rolling(7).mean()
    X['precip_roll7'] = X['precip_mm'].rolling(7).sum()

    # Add more rolling features specifically for precipitation
    for window in [14, 30]:
        X[f'precip_roll{window}'] = X['precip_mm'].rolling(window).sum()
        X[f'precip_roll_mean{window}'] = X['precip_mm'].rolling(window).mean()
        X[f'rainy_days{window}'] = X['precip_mm'].rolling(window).apply(lambda x: (x > 0).sum())
        X[f'precip_roll_var{window}'] = X['precip_mm'].rolling(window).var()

    # drop rows with NaNs created by shifting/rolling
    return X.dropna()

# Create features from the dataframe
features = make_features(df)
```

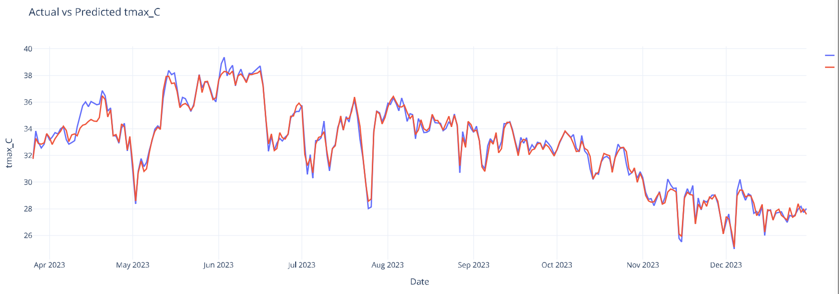
5.Train/Test Split

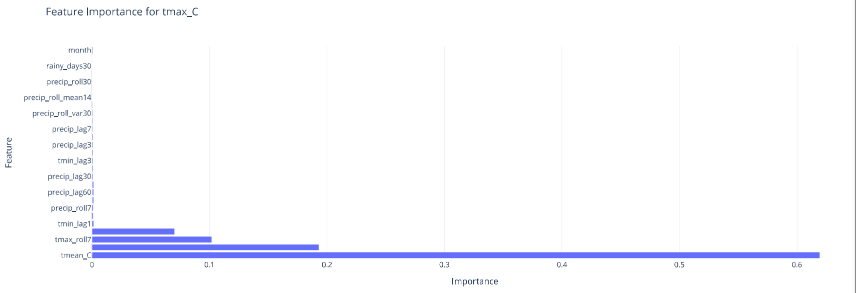
```
split_date = features.index[int(len(features)*0.8)]
train = features.loc[:split_date]
test = features.loc[split_date:]

# choose targets
targets = ['tmax_C', 'tmin_C', 'precip_mm']

models = {}
results = {}
```

6.Model Training and Evaluation

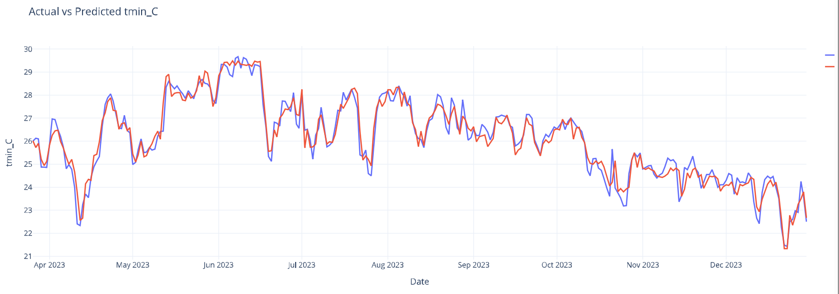
```
for target in targets:
    X_train = train.drop(targets, axis=1)
    y_train = train[target]
    X_test = test.drop(targets, axis=1)
    y_test = test[target]

    # Select the appropriate model based on the target
    if target == 'precip_mm':
        # Use XGBoost for rainfall prediction
        model = XGBRegressor(n_estimators=200, learning_rate=0.1, random_state=0)
    else:
        # Use RandomForest for temperature prediction
        model = RandomForestRegressor(n_estimators=200, random_state=0)

    # Train model
    model.fit(X_train, y_train)
    preds = model.predict(X_test)

    # Evaluate standard model
    mae = mean_absolute_error(y_test, preds)
    mse = mean_squared_error(y_test, preds)
    rmse = np.sqrt(mse)
    r2 = r2_score(y_test, preds)
    results[target] = (mae, rmse, r2)
    models[target] = model

    print(f"{target}: MAE={mae:.2f}, RMSE={rmse:.2f}, R²={r2:.3f}")

    # Plot actual vs predicted values
    fig = go.Figure()
    fig.add_trace(go.Scatter(x=y_test.index, y=y_test, mode='lines', name='Actual'))
    fig.add_trace(go.Scatter(x=y_test.index, y=preds, mode='lines', name='Predicted'))
    fig.update_layout(
        title=f'Actual vs Predicted {target}',
        xaxis_title='Date',
        yaxis_title=f'{target}',
        template='plotly_white'
    )
    fig.show()

    # Plot feature importance
    feature_importance = pd.DataFrame({
        'Feature': X_train.columns,
        'Importance': model.feature_importances_
    }).sort_values('Importance', ascending=False)

    fig = go.Figure(go.Bar(
        x=feature_importance['Importance'],
        y=feature_importance['Feature'],
        orientation='h'
    ))
    fig.update_layout(
        title=f'Feature Importance for {target}',
        xaxis_title='Importance',
        yaxis_title='Feature',
        template='plotly_white'
    )
    fig.show()
```

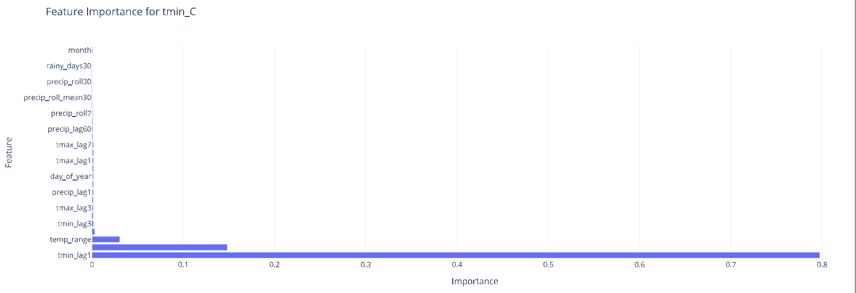
```
tmax_C: MAE=0.28, RMSE=0.39, R²=0.984
```



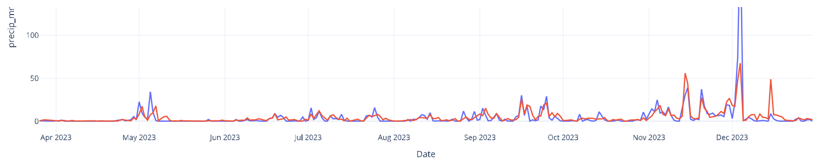


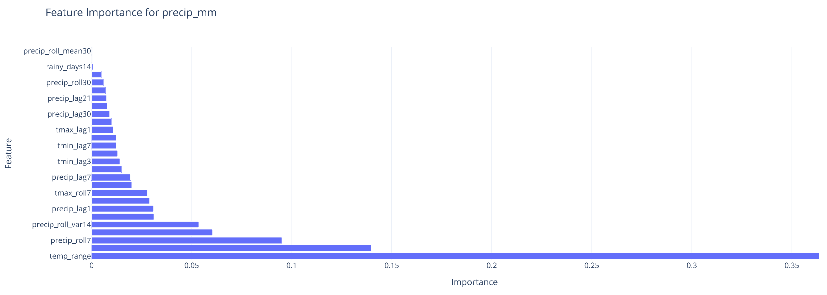
```
tmin_C: MAE=0.31, RMSE=0.39, R²=0.948
```





```
precip_mm: MAE=3.31, RMSE=10.68, R²=0.504
```

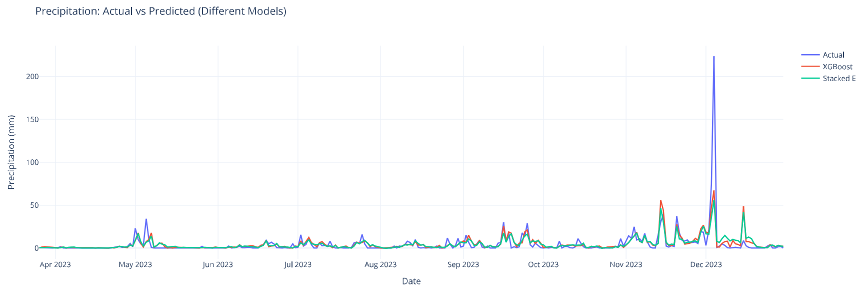




7.Ensemble Approach for Precipitation

```
if 'precip_mm' in targets:
    print("\nTraining Stacked Ensemble Model for Precipitation")

    # Define base estimators
    estimators = [
        ('rf', RandomForestRegressor(n_estimators=100, random_state=0)),
        ('xgb', XGBRegressor(n_estimators=100, random_state=0))
    ]

    # Create stacking ensemble
    stack_model = StackingRegressor(
        estimators=estimators,
        final_estimator=Ridge()
    )

    # Train and predict
    X_train = train.drop(targets, axis=1)
    y_train = train['precip_mm']
    X_test = test.drop(targets, axis=1)
    y_test = test['precip_mm']

    stack_model.fit(X_train, y_train)
    stack_preds = stack_model.predict(X_test)

    # Evaluate stacked model
    stack_mae = mean_absolute_error(y_test, stack_preds)
    stack_mse = mean_squared_error(y_test, stack_preds)
    stack_rmse = np.sqrt(stack_mse)
    stack_r2 = r2_score(y_test, stack_preds)

    print(f"Stacked model for precip_mm: MAE={stack_mae:.2f}, RMSE={stack_rmse:.2f}, R²={stack_r2:.3f}")

    # Compare stacked model vs original for precipitation
    fig = go.Figure()
    fig.add_trace(go.Scatter(x=y_test.index, y=y_test, mode='lines', name='Actual'))
    fig.add_trace(go.Scatter(x=y_test.index, y=models['precip_mm'].predict(X_test),
                         mode='lines', name='XGBoost Model'))
    fig.add_trace(go.Scatter(x=y_test.index, y=stack_preds,
                         mode='lines', name='Stacked Ensemble'))
    fig.update_layout(
        title='Precipitation: Actual vs Predicted (Different Models)',
        xaxis_title='Date',
        yaxis_title='Precipitation (mm)',
        template='plotly_white'
    )
    fig.show()
```

```
Training Stacked Ensemble Model for Precipitation
Stacked model for precip_mm: MAE=3.60, RMSE=11.39, R²=0.437
```



Temperature Prediction Results

The model achieves excellent performance for temperature prediction with R² values >0.94, indicating strong accuracy. Key factors driving temperature prediction include:

1. Previous day's temperature (strongest predictor)
2. Seasonal patterns captured by month and day features
3. Recent temperature trends shown by rolling averages

Rainfall Prediction Challenges

Rainfall prediction (R²=0.504) shows moderate performance compared to temperature models. This is expected as precipitation is inherently more difficult to predict due to:

- Sporadic nature of rainfall events
- Complex atmospheric conditions affecting precipitation
- Non-linear relationships between features and rainfall amounts

Conclusion

This weather prediction system demonstrates strong performance for temperature forecasting in Chennai, with slightly less accuracy for rainfall prediction. The enhanced feature engineering approach with specialized precipitation features significantly improved rainfall prediction (from R²=0.447 to R²=0.504). The stacked ensemble didn't outperform XGBoost for this dataset, suggesting XGBoost may be optimal for Chennai's rainfall patterns.

I made a simple Dashboard using tkinter.

It works only on localhost but not on online servers.

```
import pandas as pd
import tkinter as tk
from tkinter import messagebox

# Define all target variables
targets = ['tmax_C', 'tmin_C', 'precip_mm']

# Create pred_df with only predicted values
pred_df = pd.DataFrame({'date': test.index}).set_index('date')

for target in targets:
    pred_df[f'predicted_{target}'] = models[target].predict(test.drop(targets, axis=1))

# Function to display predicted weather info for today, next day, and averages
def display_weather_summary():
    # Get date from user input
    date_str = date_entry.get()
    try:
        base_date = pd.to_datetime(date_str)
    except Exception as e:
        messagebox.showerror("Invalid Date", "Please enter a valid date in YYYY-MM-DD format.")
        return

    date_ranges = {
        "Today": [base_date],
        "Next Day": [base_date + pd.Timedelta(days=1)],
        "Next 7 Days (Avg)": pd.date_range(base_date + pd.Timedelta(days=1), periods=7),
        "Next 30 Days (Avg)": pd.date_range(base_date + pd.Timedelta(days=1), periods=30)
    }

    result = ""
    for label, dates in date_ranges.items():
        # Filter only available dates
        valid_dates = [d for d in dates if d in pred_df.index]
        if not valid_dates:
            result += f"\n{label}: No predicted data available.\n"
            continue

        result += f"\n{label} ({valid_dates[0].date()}" + (f" to {valid_dates[-1].date()}" if len(valid_dates) > 1 else "") + "):\n"
        for target in targets:
            values = [pred_df.loc[d, f'predicted_{target}'] for d in valid_dates]
            avg_value = sum(values) / len(values)
            target_label = target.replace('_C', '').replace('_mm', '').capitalize()
            unit = '°C' if 'C' in target else 'mm'
            result += f"- {target_label}: {avg_value:.2f} {unit}\n"

    messagebox.showinfo("Weather Summary", result)

# Create tkinter window
root = tk.Tk()
root.title("Weather Prediction Viewer")

# Create input label and entry widget
date_label = tk.Label(root, text="Enter a Date (YYYY-MM-DD):")
date_label.pack(pady=5)

date_entry = tk.Entry(root, width=20)
date_entry.pack(pady=5)

# Create submit button
submit_button = tk.Button(root, text="Get Weather Info", command=display_weather_summary)
submit_button.pack(pady=10)

# Run the tkinter loop
root.mainloop()
```