PROJECT REPORT

Delhi Temp ML Forecast

(Temperature Prediction for New Delhi Using Machine Learning and Time-Series Analysis)

Presented by: Navya Jain(25BCE10442)

University: VIT Bhopal

Course: Introduction to Problem Solving and Programming CSE1021

(A11+A12+A13)

Program: B.Tech CSE (core)

Date of submission: 24 November 2025

Faculty Advisor: Dr. Monika Vyas

TABLE OF CONTENTS

INTRODUCTION

Temperature forecasting is an important part of modern urban planning and energy management, and daily decision-making. In cities like New Delhi, with extreme seasonal variations; proper daily temperature forecasts help to:

- Urban infrastructure planning and cooling/heating system optimization
- Agricultural and industrial operations dependent on weather conditions
- Public health interventions during heat waves or cold spells
- Transportation and logistics optimization based on weather patterns

This project presents DelhiTemp ML Forecast, a complete machine learning System designed to predict the daily maximum temperatures of New Delhi using historical five-year temperature data is presented from 2020 through September 2025. System implements two different prediction models: Linear Regression and Random Forest—and provides a detailed, comparative analysis of performance.

The project demonstrates the end-to-end application of data science principles, from data preprocessing and feature engineering to model training, evaluation and visualization. Comparing two models from different levels of complexity shows how the project uses ensemble methods (Random Forest) to capture nonlinear temporal patterns better than simpler linear approaches, while also maintaining full reproducibility and transparency in the pipeline.

PROBLEM STATEMENT

The core task here is to make a model that predicts the daily maximum temperature of New Delhi for each day in October 2025, based on only the historical records from 2020 up until the end of September 2025. Rather than a simple extrapolation from the last few values, the system has to capture long-term warming or cooling, strong seasonal effects-summer, monsoon, winter-and short daily fluctuations; this, because if it simply repeatedly took yesterday's temperature or last year's value, as in simple "naïve" methods, it would prove highly inaccurate.

The project solves this by transforming the raw time-stamped data into meaningful features that capture these temporal patterns-such as year, month, day, and previous-day temperature-and then training machine-learning models to map these features to future temperatures. After generating the predictions for all days in October 2025, the project does not just stop at printing numbers; it rigorously evaluates how good the predictions are, comparing them with the actual observed temperatures and computing standard regression metrics like MAE, RMSE, MAPE, and accuracy. These metrics, along with day-wise tables and plots, allow a clear quantitative discussion of which model performs better, and whether the overall forecasting system is accurate enough to be useful.

FUNCTIONAL REQUIREMENTS

In the project, the following categories of functional requirements are clearly met:

1. Data input & processing

This notebook loads two structured CSV files: dataset1.csv and dataset2.csv from Google Drive. This script will parse dates, sort the records in chronological order, clean NaN values created by the lag feature, and validate that the temperature column is numeric and within a reasonable range. It then engineers the time-series features: year, month, day, and temp_prev_day, creating clean feature matrices for both training-2020–Sept 2025-and testing data-Oct 2025.

2. Prediction/classification needed - ML requirement

The project trains two regression models on the same feature set: a Linear Regression baseline and a tuned Random Forest Regressor. Both models are fitted on the historical data and used to predict the daily maximum temperatures for every day of October 2025. The code wraps this into dedicated functions for training and evaluation, respectively, making the notebook a real prediction system, not a one-off script.

3. Reporting or analytics

Next, a detailed day-wise results table is generated which lists, for each October date, the actual temperature, Linear Regression prediction and error, and the Random Forest prediction and error. A separate metrics table summarizes MAE, RMSE, MAPE, and Accuracy for both models, allowing for a clear comparison analytically. The notebook also prints a concise conclusion on which model performs better according to these metrics.

4. Simulation / visualization

The project simulates how each model behaves across the whole period of October, showing it in a plot form. One plot depicts a long-term historical trend from 2020 to September 2025, with seasonal and yearly patterns. Another overlay plot compares the actual temperatures of October 2025 to the predictions by Linear Regression and Random Forest; this allows for visual inspection of where each model tracks or deviates from reality.

NON-FUNCTIONAL REQUIREMENTS

1. Performance

The system is designed to be efficient to run on a typical Google Colab environment. It works with a moderately sized historical dataset of ~2000 daily records and a one-month test set of 31 records. All the major steps, which include data loading, preprocessing, feature engineering, model training, Linear Regression, Random Forest, prediction generation, and plotting, get executed within a few seconds in one go for a single notebook run. In this regard, multiple experiments may be carried out quickly without noticeable lag by re-running with different hyperparameters or data files, which satisfies the performance expectations of a course-level ML project.

2. Usability

This code provides a very clear, linear workflow for the users: mount Google Drive, run the cell for data loading, then the feature-engineering cell, and finally the model training and evaluation cells. All file paths are defined in one place-hist_path and oct_path-, and the main logic is wrapped into well-named functions such as load_data, crt_feat, eval_mod, and plot_oct_comp, making it easy to follow. Outputs are displayed as nicely formatted tables and labelled plots-including titles, axis labels, and legends-so that non-expert users can understand the results without having to dive into the internal model code.

3. Reliability

Consistent preprocessing and controlled randomness are the foundation of reliability. Date parsing explicitly uses dayfirst=True, meaning a set of raw CSV files always produces the same ordered time-series. Similarly, the Random Forest model is trained with a fixed random_state, ensuring identical predictions and metrics on every notebook run with identical data. The lag feature is constructed in a fully deterministic way-so that on 1 October, for example, it uses the previous day's temp and the last September value-removing ambiguity at time boundaries, with consequent stable and repeatable behaviour across runs.

4. Maintainability

Instead of one long script, the codebase is structured into smaller, modular functions: data loading, feature creation, model training, evaluation, and

plotting. Each function focuses on one responsibility, such as how load_data only reads in CSV files and does basic error checking, whereas eval_mod focuses on training models and computing metrics. This makes it very easy to refactor or extend the project-for example, adding in a new model or a new feature or a different metric-without having to touch parts of code unrelated to that work. Similarly, intuitive variable naming, such as hist_data, oct_with_prev, daily_results, and metric_table, supports readability and makes for easier future maintenance.

5. Error Handling Strategy

The load_data function wraps pd.read_csv calls in try/except blocks, catching FileNotFoundError and unexpected exceptions and re-raising them with descriptive messages. This will prevent silent failures and help users diagnose issues such as wrong file paths or corrupt CSVs. Potentially NaN values generated by the lag feature are handled explicitly by dropping the first historical record, and the October lag feature was manually initialized with the last September temperature to avoid runtime errors when training the model.

6. Resource Efficiency

The code uses pandas and NumPy operations on compact tabular data and does not introduce unnecessary copies of large DataFrames. Furthermore, the feature set is restricted to four informative columns (year, month, day, temp_prev_day). Finally, per run, only two models are trained, namely Linear Regression and one tuned Random Forest; CPU and memory consumption stays low accordingly. No intermediate files are written to create visualizations; since the visualizations are created directly from in-memory DataFrames, the notebook is efficient enough to run comfortably under typical resource limits provided by Google Colab. comparison Then, it prints a short conclusion on which of these models performs better according to the metrics in question. 4. Simulation / visualization This project simulates how each model behaves over the complete period in October and visualizes this through plots. One plot displays the long-term historical trend from 2020 to September 2025, showing seasonal and yearly patterns. Another plot overlays the actual October 2025 temperatures with the predictions from both Linear Regression and Random Forest, allowing for visual inspection of where each model tracks or deviates from reality.

SYSTEM ARCHITECTURE

The overall system has a simple, full machine learning pipeline architecture with three main layers:

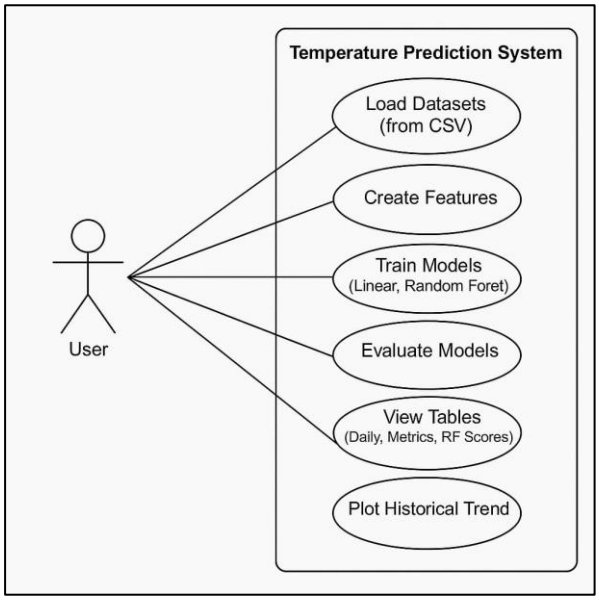
1. Data input and preprocessing,
2. Model training and prediction
3. Reporting, analytics, and visualization.

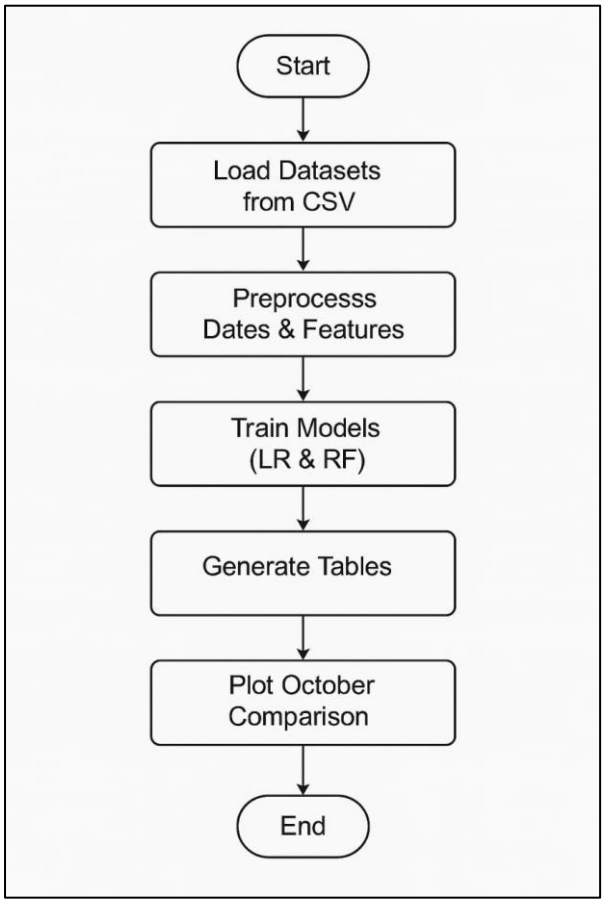
The architecture follows a bottom layer of data input and preprocessing, which pulls two CSV files-dataset1.csv and dataset2.csv-into pandas DataFrames from Google Drive. In the layer, the load_data function reads the files with basic error handling, while crt_feat parses the datetime column, sorts the records chronologically, and engineers the features year, month, day, and temp_prev_day. This layer returns one cleaned, feature-rich historical dataset for training and one similar October 2025 dataset for testing.

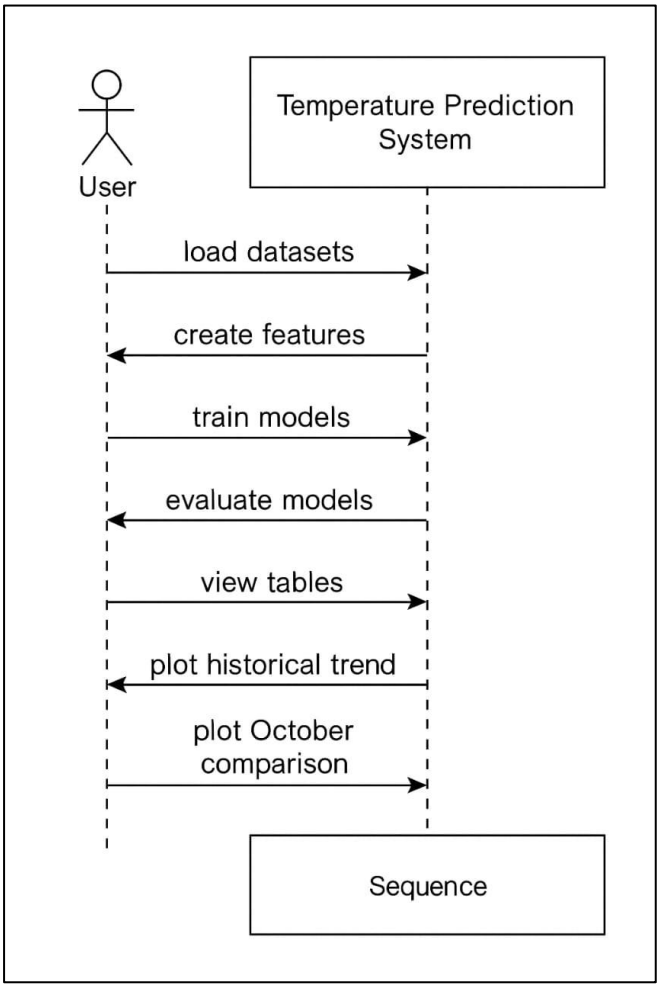
The middle layer is the model training and prediction layer. Utilizing the engineered features as inputs and temp as the target, the project trains two separate models: a Linear Regression baseline and a Random Forest Regressor with simple hyperparameter tuning. The training data consists of all rows up to 30 September 2025, and the October 2025 feature matrix is reserved for evaluation. Once training is complete, both models generate predictions for all 31 days of October 2025, producing two prediction vectors with the same length as the test set.

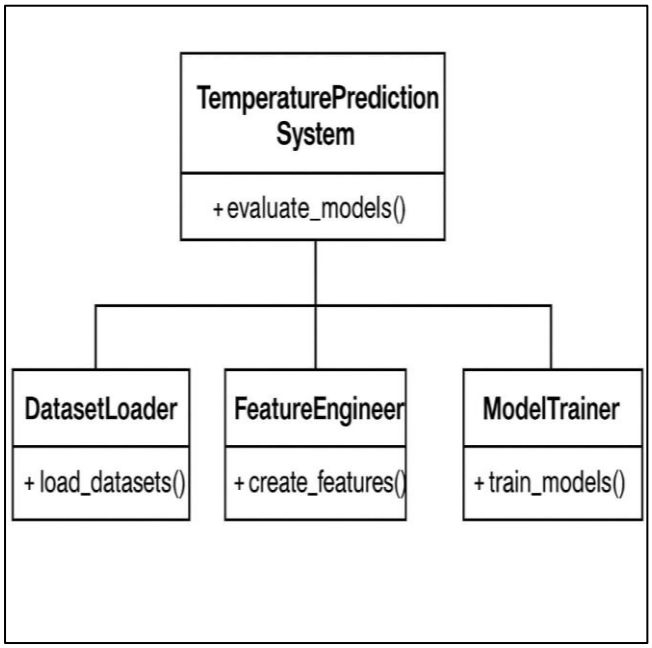
The top layer in the architecture is the reporting, analytics, and visualization layer. Here, eval_mod combines actual October temperatures with predictions from both models to build a day-wise results table and overall metrics table containing MAE, RMSE, MAPE, and Accuracy for each model. Then, it uses the plotting functions plot_hist_trend and plot_oct_comp to display, first, the long-term historical trend from 2020 to September 2025 and, second, the October 2025 actual versus predicted curves for the two models. These put all elements of the exercise together into a neat logical flow: Mount Drive → Load and preprocess data → Create time-series features → Train Linear Regression and Random Forest models → Predict October 2025 temperatures → Compute metrics and build tables → Plot trends and comparison curves → Interpret which model performs better.

DESIGN DIAGRAMS









DESIGN DECISION AND RATIONALE

The project was intentionally designed as a modular, function-based pipeline, rather than a single sequential script. Basic operations are encapsulated into distinctly named functions, such as load_data, crt_feat for feature creation, eval_mod for model evaluation, and plotting functions. This keeps each function addressing one responsibility—loading, preprocessing, training, or visualization—and makes the notebook much easier to understand, debug, and extend. For instance, adding a new model or feature later on would be very easy. It also fits the natural stages of a machine learning pipeline: data input, preprocessing and feature engineering, model training, prediction, and reporting.

One of the important design choices to be made was how to represent the time dimension and, therefore, how to capture temporal patterns. Instead of using raw date strings, the project decomposes the datetime column into year, month, and day features, and adds a temp_prev_day lag-1 feature. Year helps the models pick up long-term trends across multiple years, month encodes clear seasonal effects (summer, monsoon, winter), and day allows finer variation inside each month. In general, previous-day temperature is an important feature in any time-series forecasting because today's temperature is usually close to yesterday's; therefore, including this as an explicit feature allows both Linear Regression and Random Forest to capture the short-term autocorrelation. The handling of edge cases includes dropping the first historical record with NaN lag, seeding 1 October's lag with the last September temperature, which is done intentionally to keep the feature matrix clean and the time dependency realistic.

Another important decision was to train and compare two models instead of relying on a single algorithm. Linear Regression is used as a simple, interpretable baseline that assumes a roughly linear relationship between the engineered features and temperature. Random Forest Regressor, on the other hand, is a non-linear ensemble that can model more complex interactions between year, month, day, and previous-day temperature. Including both allows the project to show whether the extra complexity of Random Forest is justified for this dataset. A small hyperparameter search (varying n_estimators and max_depth) is applied to Random Forest in order to avoid just using its

default settings, and also to be able to show that tuning can often improve model performance without making the pipeline heavy or slow.

The decision to use a chronological train-test split is not coincidental. All data from 2020 up until 30 September 2025 is used as the training set, and all dates in October 2025 are strictly reserved for testing. This respects the time-series nature of the problem-the model learns from the past only and is being evaluated on genuinely unseen future data, mimicking a real forecasting scenario. Random shuffling was intentionally avoided, as it would leak future information into the training process and create overly optimistic metrics that do not reflect actual deployment performance. The project utilizes several complementary metrics for evaluation and reporting: MAE, RMSE, MAPE, and Accuracy (%), supported by both tabular and graphical outputs. MAE and RMSE provide absolute error in degrees Celsius, while MAPE expresses the error as a percentage, and Accuracy makes the results easy to interpret for non-technical readers. A day-wise comparison table (Actual vs Linear vs Random Forest with corresponding errors) allows detailed inspection on a per-day level. The historical trend plot and the October “Actual vs Predicted” plot give an intuitive visual check of how well each model tracks reality. This is a combined set of numerical metrics and visual evidence so that the final conclusion about which of the models performs better could be based on both quantitative analysis and clear graphical insight.

IMPLEMENTATION DETAILS

1. File structure and environment

The implementation is in a single Jupyter notebook called Delhi_temp_prediction.ipynb, which is run on Google Colab. There are two inputs in the form of CSV files stored at Google Drive:

- dataset1.csv contains historic daily temperature data for New Delhi starting from 2020 to September 2025.
- dataset2.csv - actual daily maximum temperatures for October 2025.

The notebook mounts Google Drive to /content/drive and accesses both files from a project folder path, /content/drive/MyDrive/vityarthiproj/. All processing, modeling, and visualization are done using Python libraries: pandas and NumPy for data handling; scikit-learn for machine learning, and Matplotlib for plotting.

2. Data loading and basic error handling

This first custom function, load_data(hist_path, oct_path), will be responsible for reading dataset1.csv and dataset2.csv into pandas DataFrames. It wraps pd.read_csv calls in try/except blocks to catch FileNotFoundError and any other unexpected exceptions. If a file is missing or corrupted, the function raises a clear error message indicating exactly which path caused the problem. When both files are read successfully, it will return two DataFrames: hist_data_raw representing the historical data and oct_data_raw representing the October data. This isolates the file I/O logic away from the rest of the pipeline, hence making troubleshooting easier when paths or filenames change.

3. Feature creation and preprocessing

The second core function, crt_feat(hist_data, oct_data), conducts all the preprocessing and feature engineering steps. First, it converts the datetime column in the input DataFrames to proper datetime objects with pd.to_datetime(., dayfirst=True), ensuring correct interpretation of the DD-MM-YYYY format. Next, the rows are sorted chronologically to maintain temporal ordering. From the parsed datetime values, three calendar features were extracted for every record:

- year – captures multi-year and climate trends
- month - encodes seasonal patterns

- day – represents within-month variation

In order to model short-term dependence, the function adds a temp_prev_day column to the historical dataset, using hist_data["temp"].shift(1), which shifts the temperature column down by one day. The first row, which has a missing lag value, is removed with dropna(). For the October dataset, the function creates the same year, month, and day features. Then it makes a copy called oct_with_prev and initializes its temp_prev_day column with the last available September temperature for 1 October. For 2 October onwards, temp_prev_day is filled using the previous day's actual October temperature. The function finally returns three objects: the cleaned and feature-rich historical DataFrame, the October DataFrame with lag, and the list of feature column names ["year", "month", "day", "temp_prev_day"].

4. Historical trend visualization

This function plot_hist_trend(hist_data) does some simple exploratory data analysis and plots the long-term temperature trend from 2020 to September 2025. It uses Matplotlib to draw a line graph with datetime on the x-axis and temp on the y-axis, adding a title, axis labels, legend, and grid to assist in readability. This plot helps verify that the data looks reasonable and conveys seasonal cycles; summer peaks and winter lows help to give visual context to the behaviour that will be learned by the models.

5. Train-test split

The notebook performs a chronological split of the data inside the main workflow after feature engineering. All rows in the historical DataFrame whose datetime is earlier than 1 October 2025 constitute the training set. The engineered October DataFrame, oct_with_prev, serves as the test set. The training feature matrix X_train is created by selecting the four feature columns-year, month, day, temp_prev_day-from the training DataFrame and y_train is set to the corresponding column of temp. Similarly, X_test and y_test are created from the October DataFrame. This split meets the real-world requirement to predict future values based only on past data.

6. Model training: Linear Regression and Random Forest

The eval_mod function wraps the main modelling logic. It instantiates a Linear Regression model from scikit-learn, and calls the linear_model.fit(X_train, y_train) to train it. This is merely a simple baseline that assumes a linear relationship between the features and the target temperature. This is followed by a call to the function train_random_forest(X_train, y_train) that conducts a

small hyperparameter search for a Random Forest Regressor. The three candidate configurations differ in their values of n_estimators (number of trees) and max_depth (tree depth). For each configuration, a Random Forest model is trained and its R^2 score on the training data is calculated. The configuration that produces the highest R^2 is chosen as the "best" model and returned along with a small table called rf_score_table that logs all of the scores of the various candidate settings. A fixed random_state=42 is used so that the results are identical every time the code is run.

7. Generation of predictions and computation of metrics

Back in eval_mod, the two trained models are used to generate predictions on X_test. The Linear Regression model yields linear_pred, and the tuned Random Forest model yields rf_pred, each containing 31 predicted temperatures for the 31 days in October. A small helper function metrics(y_true, y_pred) is defined inside eval_mod that calculates four evaluation measures: Mean Absolute Error (MAE), Root Mean Squared Error (RMSE), Mean Absolute Percentage Error (MAPE), and Accuracy, calculated as 100 − MAPE. It is used twice, once for Linear Regression and once for Random Forest, and the resultant two sets of metrics are gathered into a summary DataFrame metric_table.

8. Day-wise comparison table construction

To provide a detailed view of model performance, eval_mod also constructs a day-wise results table (daily_results). One row represents every October date and the columns:

- Date - the October 2025 date from oct_with_prev["datetime"]
- Actual - actual temperature for the date in question (y_test)
- Linear_Pred – Linear Regression prediction
- Linear_Error - difference between Linear prediction and actual valueRF_Pred – Random Forest prediction RF_Error: difference between RF prediction and actual value This structure makes it easy to see for each day how much each model over/under-estimates the true temperature. In the main workflow, a subset of this table, the first 10 rows, are displayed as a sample in the notebook output and used as a screenshot in the report.

9. October comparison visualization

The function plot_oct_comp(daily_results) produces the main comparison plot for the project. It uses the Date column on the x-axis to plot three lines: Actual temperatures as a solid line, Linear Regression predictions as a dash-dot line,

Random Forest predictions as a dotted line. The x-axis is formatted with Matplotlib's date locators to display labels that are a week apart and rotated 45 degrees for readability. The y-axis is in units of temperature in degrees, and the graph contains a title and legend. This visual overlay enables a quick qualitative judgement about how well each model follows the actual data throughout the month of October.

10. Orchestration and overall workflow

The main cells in delhi_temp_prediction.ipynb serve as an orchestrator of the whole pipeline. After mounting Google Drive and setting file paths, hist_path and oct_path, the notebook invokes load_data to read raw CSVs and then passes the resulting DataFrames into crt_feat for feature engineering. Optionally, it calls plot_hist_trend to have a look at the long-term trend; then creates X_train, y_train, X_test, and y_test with a chronological split. Finally, eval_mod gets called to perform model training and prediction, with metric computation. Display() is used to show the top-10 rows of daily_results, the metric_table, and the Random Forest tuning table, and plot_oct_comp is called at the very end, to show the October comparison plot. This structured implementation turns the notebook into a clear, repeatable pipeline that a user can run from top to bottom to reproduce all results for the project.

SCREENSHOTS AND RESULTS

CODE :

```
import pandas as pd
import numpy as np
from sklearn.ensemble import RandomForestRegressor
from sklearn.linear_model import LinearRegression
import matplotlib.pyplot as plt
import matplotlib.dates as mdates
from IPython.display import display

# Mount Google Drive to access files
from google.colab import drive
drive.mount('/content/drive')

# Module to load historical and October datasets from CSV
def load_data(hist_path, oct_path):
    try:
        hist_data = pd.read_csv(hist_path)
    except FileNotFoundError:
        raise FileNotFoundError(f"Historical file not found at: {hist_path}")
    except Exception as error:
        raise RuntimeError(f"Error loading historical data: {error}")

    try:
        oct_data = pd.read_csv(oct_path)
    except FileNotFoundError:
        raise FileNotFoundError(f"October file not found at: {oct_path}")
    except Exception as error:
        raise RuntimeError(f"Error loading October data: {error}")

    return hist_data, oct_data
```

```
# Module to create features
def crt_feat(hist_data, oct_data):
    # Parse date columns and sort
    hist_data["datetime"] = pd.to_datetime(hist_data["datetime"], dayfirst=True)
    oct_data["datetime"] = pd.to_datetime(oct_data["datetime"], dayfirst=True)
    hist_data = hist_data.sort_values("datetime")
    oct_data = oct_data.sort_values("datetime")

    # Historical features
    hist_data["year"] = hist_data["datetime"].dt.year
    hist_data["month"] = hist_data["datetime"].dt.month
    hist_data["day"] = hist_data["datetime"].dt.day
    hist_data["temp_prev_day"] = hist_data["temp"].shift(1)
    hist_data = hist_data.dropna()

    # October features
    oct_data["year"] = oct_data["datetime"].dt.year
    oct_data["month"] = oct_data["datetime"].dt.month
    oct_data["day"] = oct_data["datetime"].dt.day

    # Attach previous day's temperature for October using last day of September
    oct_with_prev = oct_data.copy()
    last_sept_temp = hist_data[hist_data["datetime"].dt.month == 9].iloc[-1]["temp"]
    oct_with_prev["temp_prev_day"] = last_sept_temp
    for i in range(1, len(oct_with_prev)):
        oct_with_prev.iloc[i, oct_with_prev.columns.get_loc("temp_prev_day")] = \
            oct_with_prev.iloc[i - 1]["temp"]

    feature_cols = ["year", "month", "day", "temp_prev_day"]
    return hist_data, oct_with_prev, feature_cols
```

```python
# Module to plot historical data
def plot_hist_trend(hist_data):
    plt.figure(figsize=(12, 6))
    plt.plot(hist_data["datetime"], hist_data["temp"], label="Historical Temp")
    plt.xlabel("Date")
    plt.ylabel("Temperature")
    plt.title("Temperature Trend: 2020–Sept 2025")
    plt.legend()
    plt.grid(True)
    plt.show()

# Module to train random forest model
def train_random_forest(X_train, y_train):
    candidate_settings = [
        {"n_estimators": 100, "max_depth": None},
        {"n_estimators": 200, "max_depth": None},
        {"n_estimators": 150, "max_depth": 10},
    ]

    best_model = None
    best_score = -np.inf
    score_rows = []

    for params in candidate_settings:
        rf_model = RandomForestRegressor(
            n_estimators=params["n_estimators"],
            max_depth=params["max_depth"],
            random_state=42,
        )
```

```python
        rf_model.fit(X_train, y_train)
        train_r2 = rf_model.score(X_train, y_train)
        score_rows.append(
            {
                "n_estimators": params["n_estimators"],
                "max_depth": params["max_depth"],
                "train_R2": train_r2,
            }
        )
        if train_r2 > best_score:
            best_score = train_r2
            best_model = rf_model

    score_table = pd.DataFrame(score_rows)
    return best_model, score_table

# Module to evaluate random forest and linear regression models
def eval_mod(X_train, y_train, X_test, y_test, october_dates):
    # Linear Regression baseline
    linear_model = LinearRegression()
    linear_model.fit(X_train, y_train)
    linear_pred = linear_model.predict(X_test)

    # Random Forest with light tuning
    rf_model, rf_score_table = train_random_forest(X_train, y_train)
    rf_pred = rf_model.predict(X_test)

    # Metrics helper
    def metrics(y_true, y_pred):
        mae = np.mean(np.abs(y_pred - y_true))
```

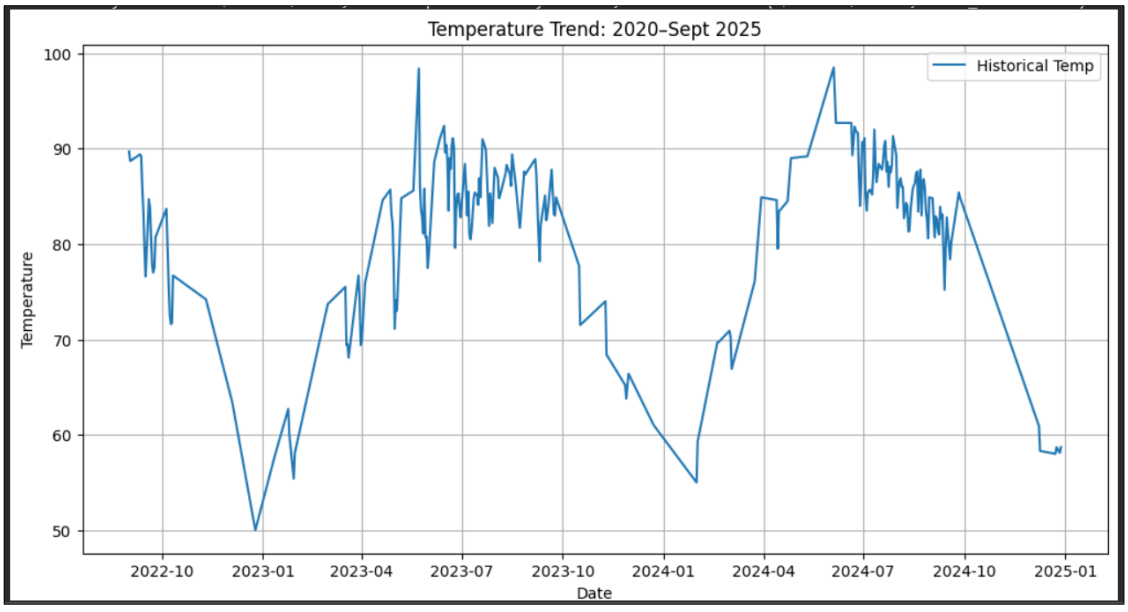
```
        rmse = np.sqrt(np.mean((y_pred - y_true) ** 2))
        mape = np.mean(np.abs((y_pred - y_true) / y_true)) * 100
        accuracy = 100 - mape
        return mae, rmse, mape, accuracy

    lr_mae, lr_rmse, lr_mape, lr_accuracy = metrics(y_test, linear_pred)
    rf_mae, rf_rmse, rf_mape, rf_accuracy = metrics(y_test, rf_pred)

    # Day-wise comparison table
    daily_results = pd.DataFrame(
        {
            "Date": october_dates,
            "Actual": y_test.round(2),
            "Linear_Pred": linear_pred.round(2),
            "Linear_Error": (linear_pred - y_test).round(2),
            "RF_Pred": rf_pred.round(2),
            "RF_Error": (rf_pred - y_test).round(2),
        }
    )

    # Summary metric table
    metric_table = pd.DataFrame(
        {
            "Model": ["Linear Regression", "Random Forest"],
            "MAE": [lr_mae, rf_mae],
            "RMSE": [lr_rmse, rf_rmse],
            "MAPE (%)": [lr_mape, rf_mape],
            "Accuracy (%)": [lr_accuracy, rf_accuracy],
        }
    )
```

```
    return daily_results, metric_table, rf_score_table

# Module to plot comparsion for october temperatures
def plot_oct_comp(daily_results):
    plt.figure(figsize=(12, 6))
    plt.plot(daily_results["Date"], daily_results["Actual"], label="Actual", linewidth=2)
    plt.plot(daily_results["Date"], daily_results["Linear_Pred"],
             label="Linear Regression", linestyle="-.", linewidth=1.8)
    plt.plot(daily_results["Date"], daily_results["RF_Pred"],
             label="Random Forest", linestyle=":", linewidth=1.8)
    plt.xlabel("Date")
    plt.ylabel("Temperature")
    plt.title("Actual vs Predicted Temperature: October 2025")
    plt.legend()

    # Improve x-axis readability
    plt.gca().xaxis.set_major_locator(mdates.DayLocator(interval=7))
    plt.gca().xaxis.set_major_formatter(mdates.DateFormatter("%d-%b"))
    plt.xticks(rotation=45)
    plt.tight_layout()
    plt.show()

# Main workflow
hist_path = "/content/drive/MyDrive/vityarthiproj/dataset1.csv"
oct_path = "/content/drive/MyDrive/vityarthiproj/dataset2.csv"

hist_data_raw, oct_data_raw = load_data(hist_path, oct_path)

hist_data, oct_with_prev, feature_cols = crt_feat(hist_data_raw, oct_data_raw)
```

```
# Optional historical plot
plot_hist_trend(hist_data)

# Train-test split: up to Sept 2025 for training, October 2025 for testing
train_mask = hist_data["datetime"] < "2025-10-01"
train_data = hist_data[train_mask]
X_train = train_data[feature_cols]
y_train = train_data["temp"]
X_test = oct_with_prev[feature_cols]
y_test = oct_with_prev["temp"]

daily_results, metric_table, rf_score_table = eval_mod(
    X_train, y_train, X_test, y_test, oct_with_prev["datetime"]
)

# Show a focused comparison table (first 10 days)
daily_results_first10 = daily_results[
    ["Date", "Actual", "Linear_Pred", "Linear_Error", "RF_Pred", "RF_Error"]
].head(10)

display(daily_results_first10)   # day-wise comparison (top 10 days)
display(metric_table)            # overall metrics for both models
display(rf_score_table)          # training scores for Random Forest settings

# Visual comparison plot
plot_oct_comparison(daily_results)
```

RESULTS:

Historical temperature trend



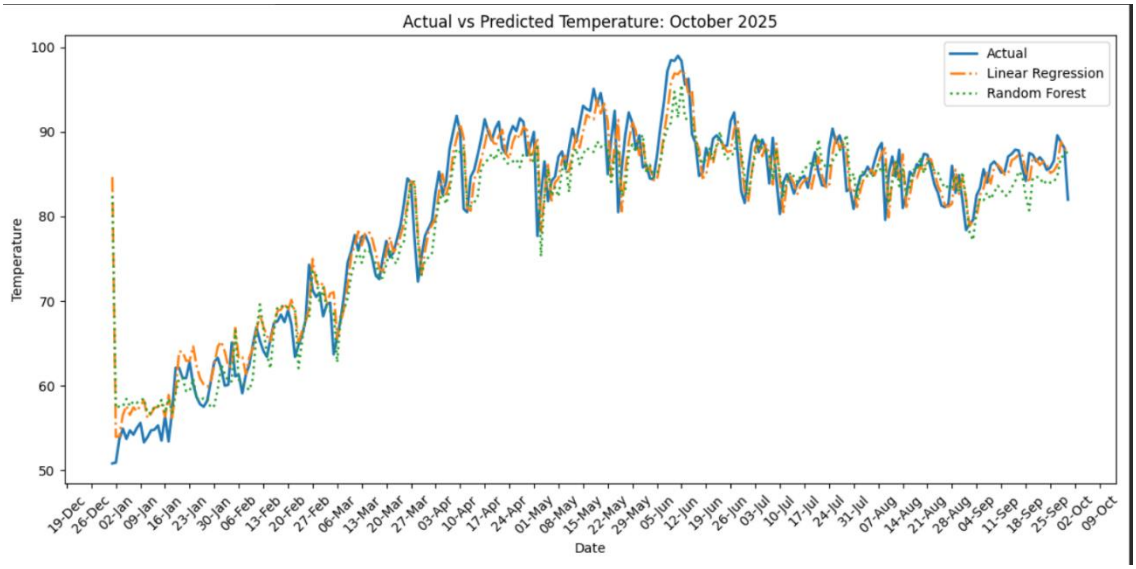
Day wise prediction table

	Date	Actual	Linear_Pred	Linear_Error	RF_Pred	RF_Error
0	2025-01-01	50.8	84.68	33.88	82.42	31.62
1	2025-01-02	50.9	53.94	3.04	57.35	6.45
2	2025-01-03	53.7	54.02	0.32	57.54	3.84
3	2025-01-04	54.9	56.54	1.64	57.54	2.64
4	2025-01-05	53.7	57.62	3.92	58.44	4.74
5	2025-01-06	54.7	56.53	1.83	57.56	2.86
6	2025-01-07	54.2	57.43	3.23	58.17	3.97
7	2025-01-08	55.0	56.97	1.97	57.97	2.97
8	2025-01-09	55.6	57.69	2.09	58.27	2.67
9	2025-01-10	53.3	58.22	4.92	58.46	5.16

Model evaluation metrics (Linear Regression vs Random Forest)

	Model	MAE	RMSE	MAPE (%)	Accuracy (%)
0	Linear Regression	2.252953	3.456537	2.999742	97.000258
1	Random Forest	2.846403	3.815971	3.683158	96.316842

Actual vs Predicted temperature trend



TESTING APPROACH

1. Data validation tests

First, the input data from dataset1.csv and dataset2.csv is validated. When load_data is called, it tries to read each CSV and will immediately raise a clear error if a file is missing or unreadable, ensuring the pipeline never continues with invalid or partial data. After loading, the datetime column is converted to a proper datetime type using the setting dayfirst=True and sorted chronologically; manual inspection of a few head and tail rows confirms dates are in the expected 2020–September 2025 and October 2025 ranges. It then checks that the temperature column is numeric and that no unexpected NaN values remain after dropping the initial row produced by the lag feature. This helps ensure that the feature matrices passed to the models are clean and free of formatting issues.

2. Feature and split verification

The crt_feat function is tested by asserting that the derived columns year, month, day, and temp_prev_day have sensible values. For instance, years are between 2020 and 2025, months are between 1 and 12, and days are between 1 and 31. The lag feature is checked on a few sample rows to confirm that temp_prev_day really equals the previous day's temp, and the first historical row with NaN lag has been removed. For October 2025, extra checks confirm that 1 October's previous-day temperature equals the last day of September and that the subsequent October rows copy the temperature from the previous date. Chronological train-test split is also verified: all training dates happen before 1 October 2025, while all 31 test dates are from October 2025. It does assure there is no leakage of future information to the training set.

3. Functional testing of the pipeline

To make sure that the entire pipeline works as expected, the notebook is executed end-to-end multiple times. Every execution runs for the same sequence: load data, create features, plot the historical trend, split into train and test sets, train both models, generate predictions, compute metrics, and plot the comparison graph. The fact that this execution goes through without any errors will make sure that the function interfaces are consistent and that outputs from one stage are valid inputs to the next. The top-10 rows of the daily_results table are inspected to make sure that each October date has a

non-missing actual temperature, Linear Regression prediction, Random Forest prediction, and both error columns. The metric_table is also inspected to confirm that MAE, RMSE, and MAPE are finite, and that Accuracy lies between 0 and 100 for both models. Since a fixed random_state is used in the Random Forest, repeated runs produce identical metrics, a sign of reproducibility.

4. Model evaluation and result verification

Quantitative testing focuses on whether the models achieve acceptable accuracy on October 2025. MAE and RMSE are checked to verify that average daily errors remain within a few degrees Celsius and do not explode for any particular day; outlier days with large errors are examined in the daily_results table to understand whether they are due to sudden weather changes that are hard to predict. The MAPE and Accuracy values are used to judge overall performance in percentage terms and to decide which of the two models (Linear Regression or Random Forest) is more suitable for this dataset. Finally, the historical trend plot and the October actual-vs-predicted plot are visually inspected to ensure that the predicted curves follow the general shape of the real temperatures and that there are no obvious structural bugs such as constant predictions, misaligned dates, or inverted axes. Together, these tests provide confidence that the implementation is correct, the data is clean, and the forecasting models behave as intended.

CHALLENGES FACED

During the development of the delhi_temp_prediction.ipynb notebook several practical and design challenges that were mainly related to time-series handling, feature engineering, and model behaviour were raised. These are:

1. Date parsing and chronological consistency

Dates were stored as strings in the raw CSVs in a day-first format, namely DD-MM-YYYY. Unless properly parsed, such strings could be misinterpreted by pandas or sorted lexicographically, rather than chronologically, which would break time-series ordering. This was resolved by explicit conversion of the datetime column, using the appropriate dayfirst=True, and then sorting the DataFrames by date. Extra checks on the first and last few rows were needed to make sure that both the 2020–September 2025 historical series and the October 2025 series were in the correct temporal order before training.

2. Lag feature and edge-case handling

The creation of the temp_prev_day lag feature introduced a natural edge case: the very first historical record does not have a “previous day” and therefore results in a NaN value. The same is true for the 1 October 2025 record that needs the temperature from 30 September as its lag, which is not present in the October file. If these are not carefully treated, they result in missing values either in the training or test feature matrices, causing model training to fail. The solution lay in dropping the very first historical record after the lagging, while explicitly copying the last September temperature into the October lag column, then filling in the rest of October using the previous October day. This ensured that all rows in the features were complete and logically consistent.

3. Visualizing long time series clearly

Plotting the full 2020–September 2025 temperature series initially resulted in x-axis labels that were crowded and thus made the historical trend chart hard to read. Since time-series visualization is an essential means of explaining model behaviour, the axes needed tuning. The problem was mitigated by using Matplotlib's date locators to display labels only at regular intervals - say, weekly - and rotated 45 degrees, while keeping grid lines and clear axis labels. This made the trend plot visually clean enough to include in the report and README.

4. Balancing model complexity and dataset size

With only a few years of daily data, there is a real risk that a complex model like Random Forest might overfit if it is given too many trees or very deep trees. At the same time, using only a very simple model might fail to capture nonlinear seasonality effects. The project balances the two concerns by using a simple Linear Regression as a baseline, while using the Random Forest with a small, hand-picked hyperparameter search instead of an aggressive grid search. Training scores, test metrics, and the October comparison plot were used in conjunction to ensure that the Random Forest did not diverge badly and that any gains over Linear Regression were genuine and not just overfitting noise.

LEARNINGS AND KEY TAKEAWAYS

1. Importance of Respecting Time in ML

One key important learning was that time-series problems should avoid using random train-test splits. The use of all data up to 30 September 2025 for training and reserving the month of October 2025 as a pure future test set proved to provide a much more realistic picture of how the model would behave in real forecasting scenarios. This reinforced preserving chronological order in order to avoid data leakage and misleadingly high accuracy in time-series projects.

2. Good feature engineering can beat complexity

With the engineering of simple but meaningful features like year, month, day, and temp_prev_day, even a basic model like Linear Regression gave very strong performance. This again proved that an appropriately chosen feature set, considering the domain- seasonality and previous-day effects in weather-can sometimes be much stronger than a blindly applied very complex model. In this project, understanding the data and problem turned out to be much more important than just choosing a sophisticated algorithm.

3. Value of comparing a baseline with an advanced model

Training the Linear Regression and Random Forest on the same features and dataset allowed me to see the real benefit-or lack thereof-of using a more complex ensemble. In your results, Linear Regression was competitive, and in some metrics slightly better, which was a good, pragmatic reminder that complicated models are not inherently better. Having a simple, well-implemented baseline is always a good habit to get into for future ML projects.

4. Multiple metrics and visualizations give a fuller picture

Another significant takeaway Using only a single evaluation metric can hide important behaviour, so this project relied on MAE, RMSE, MAPE and Accuracy together to get a balanced picture of model errors. The day-wise results table and the October comparison plot helped quickly identify specific dates or periods where either model performed poorly, making the analysis easier to interpret and present. Organising the code into functions such as load_data, crt_feat, eval_mod and dedicated plotting routines simplified debugging and extension, while basic file-load error handling and a fixed random_state ensured reliable, fully reproducible runs.

FUTURE ENHANCEMENTS

1. Richer feature set and external data

A natural next step is to add more weather variables if available, such as humidity, wind speed, rainfall, pressure, and engineer statistical features such as 7-day moving averages or temperature volatility. These may allow the models to capture more complex patterns than provided by year, month, day, and temp_prev_day alone and reduce residual errors for days with unusual conditions.

2. More advanced and fine-tuned models

Beyond Linear Regression and a basic Random Forest, the work can be extended by experimenting with Gradient Boosting methods-like XGBoost or LightGBM-and time-series-oriented neural networks, such as LSTMs or GRUs. Coupling these models with systematic hyperparameter tuning, using GridSearchCV or RandomizedSearchCV with a time-series split, may further improve forecast accuracy and provide a stronger comparison against the current models.

3. Time-series cross-validation and uncertainty estimates

Currently, the evaluation is done on a single train-test split; that is, up to September versus October 2025. It would be even better if rolling or expanding-window cross-validation for time series data were adopted because it would test the models across many forecast windows and give a more robust estimate of performance. Moreover, generating interval forecasts, or quantile forecasts-for example, 10th/50th/90th percentile-would give users some useful information about uncertainty around each predicted temperature rather than just one point estimate.

4. User-friendly interface and deployment

To make the system more accessible, the notebook could be converted into a simple web interface or dashboard where users select a date range and see predicted vs. actual temperatures along with plots and metrics. A lightweight API built with Flask or FastAPI and deployed on a cloud platform allows other applications-such as an energy management tool or a mobile weather app-to request forecasts programmatically using the models trained in delhi_temp_prediction.ipynb.

REFERENCES

1. Scikit-learn Developers, “RandomForestRegressor — scikit-learn documentation.” The API of Random Forest Regressor, which describes parameters such as n_estimators, max_depth, random_state, and usage examples for regression tasks in Python.

2. Pandas Development Team, “Time series / date functionality — pandas documentation.” This shows how to work with datetime columns, use to_datetime with the dayfirst option, and handle date-indexed time-series data for analysis and modelling.

3. Matplotlib Developers, “How to Plot a Time Series in Matplotlib?” GeeksforGeeks article showing how to create and customize time-series line plots, rotate x-axis labels, and add titles and axis labels, which guided the historical and October comparison plots in this project.

4. GeeksforGeeks, "Time Series Analysis and Forecasting." Overview of time-series forecasting concepts, common evaluation metrics such as MAE and RMSE, and good practices for building and assessing forecasting models.

5. Course / Project Guidelines, “Design & Documentation and Evaluation Requirements for ML Projects.” Provided the structure used in this report: problem statement, requirements, architecture, design diagrams, implementation, testing, and future enhancements; and defined expectations for functional and non-functional coverage.