



Climate Change Impact Analyzer

By: Rachel Armstrong, David Echavarria, and Sahil Tyrewalla



What is our Project



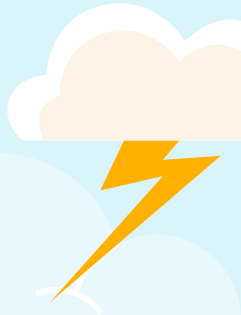

Our project, The Climate Change Impact Analyzer is a Python-based analytical tool designed to:

- Accurately preprocess diverse climate datasets.
- Predict key climate parameters (temperature, precipitation).
- Identify regional climate patterns through clustering.
- Detect anomalies indicating significant climate events.



What is Climate Change

Climate change refers to large, long-term changes in global weather patterns and temperatures. It occurs as rising temperatures, extreme weather occurrences, agricultural problems, and health effects. Addressing this situation requires strong analytical tools that can deliver accurate and useful information.





Requirements.txt

Requirements shows packages our project depends on and was generated using `pip freeze > requirements.txt`

This is our requirements.txt to the right

```
requirements.txt
1 blinker==1.9.0
2 click==8.1.8
3 contourpy==1.3.1
4 coverage==7.8.0
5 cycler==0.12.1
6 Flask==3.1.0
7 fonttools==4.56.0
8 itsdangerous==2.2.0
9 Jinja2==3.1.6
10 joblib==1.4.2
11 kiwisolver==1.4.8
12 MarkupSafe==3.0.2
13 matplotlib==3.10.1
14 numpy==2.2.4
15 packaging==24.2
16 pandas==2.2.3
17 pillow==11.1.0
18 pyparsing==3.2.3
19 python-dateutil==2.9.0.post0
20 pytz==2025.2
21 scikit-learn==1.6.1
22 scipy==1.15.2
23 seaborn==0.13.2
24 six==1.17.0
25 threadpoolctl==3.6.0
26 tzdata==2025.2
27 Werkzeug==3.1.3
28
```



README

Our README includes:

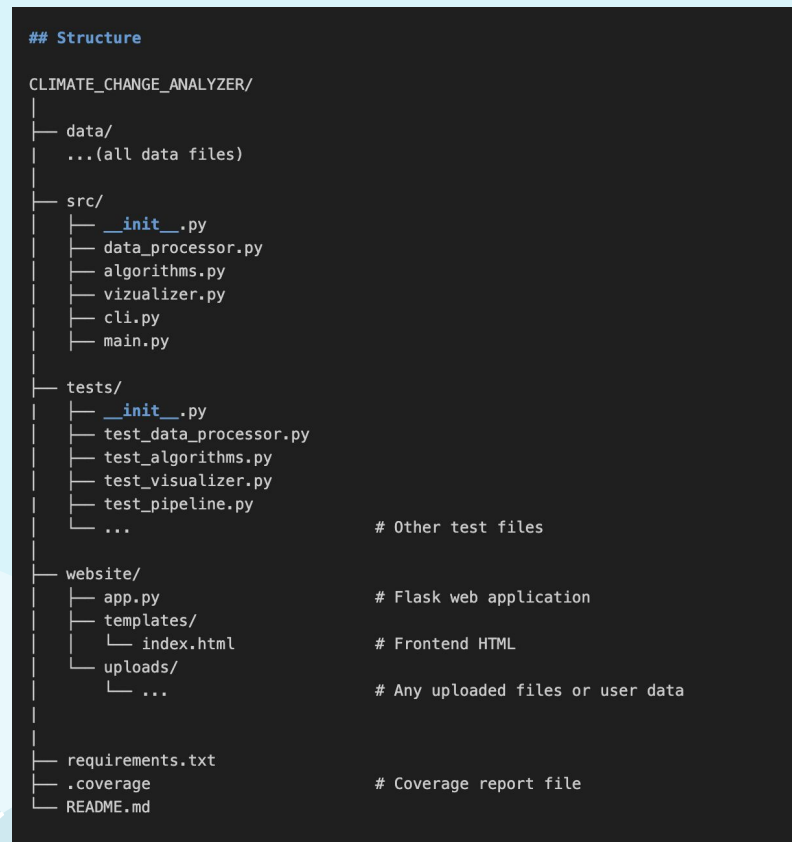
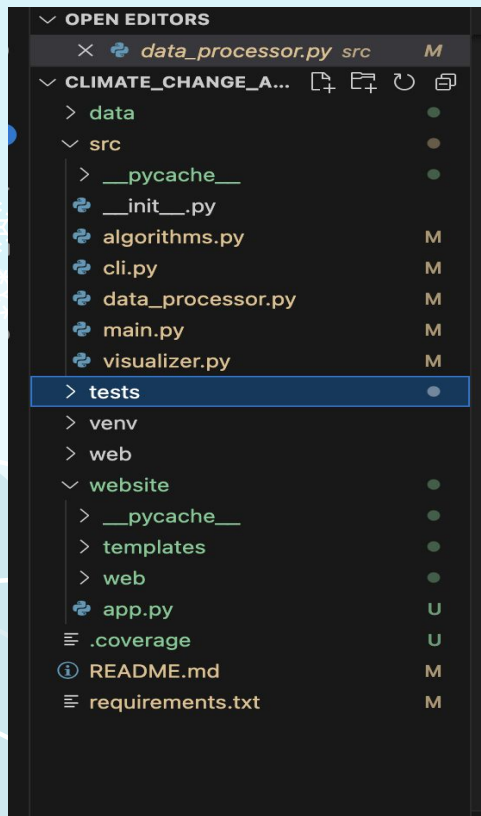
- Description
- Setup (how to set code up on your own by creating repository, virtual environment, etc.)
- Usage (How to run through main or CLI)
- Web Interface (We made an attempt at doing the bonus and includes website link)
- Running Test (test effectiveness of our code through unit test and pipeline test for extra credit)
- Project Structure
- Many Project Features

Much of this will be shown throughout the presentation

README



As you can see from Project Structure; it stands true to what is shown in photos below





Projects Data

Collected our data from NOAA and focused on temperature, precipitation and anomalies

Source: NOAA (National Oceanic and Atmospheric Administration) datasets.

Focus: Regional climate data (e.g., precipitation) from 1980 to 2025.

Format: CSV files with **Year** and **Value** columns.

Regions analyzed: Caribbean, Oceania, Gulf of America, Northern Hemisphere, etc.

```
# Title: Southern Hemisphere Land and Ocean February - January Precipitation
# Units: Millimeters
# Missing: -9999
Year,Value
1980,946.49
1981,943.51
1982,951.37
1983,961.68
1984,983.67
1985,991.52
1986,974.38
1987,948.19
1988,950.90
1989,994.58
1990,958.68
1991,945.66
1992,935.45
1993,967.90
1994,954.13
1995,938.58
1996,958.43
1997,969.65
1998,949.97
1999,1009.86
2000,955.17
2001,982.00
2002,976.39
2003,968.46
2004,969.82
2005,968.24
2006,966.18
2007,969.08
2008,975.69
2009,953.88
2010,982.38
2011,968.73
2012,946.46
2013,965.77
2014,964.91
2015,945.29
2016,956.68
2017,979.34
2018,972.37
2019,939.97
2020,949.63
```

```
# Title: Europe February - January Precipitation
# Units: Millimeters
# Missing: -9999
Year,Value
1980,899.64
1981,903.45
1982,922.23
1983,819.80
1984,848.85
1985,829.45
1986,856.64
1987,811.08
1988,888.97
1989,851.52
1990,811.59
1991,805.34
1992,791.47
1993,857.76
1994,861.40
1995,843.79
1996,850.81
1997,834.60
1998,885.32
1999,899.36
2000,891.92
2001,884.63
2002,882.91
2003,919.26
2004,800.37
2005,869.43
2006,835.58
2007,877.52
2008,869.34
2009,879.11
2010,909.90
2011,941.27
2012,826.86
2013,901.09
2014,864.67
2015,899.88
2016,823.37
2017,848.59
2018,854.01
2019,821.31
2020,824.67
```

Data Collection and Preprocessing



Data cleaning is the processing of fixing or removing inaccurate data before analyzed.

Implemented Data cleaning

```
def clean_data(self) -> pd.DataFrame:
    """Replace error and drop the missing values"""
    if self.df is not None and not self.df.empty:
        self.df.replace(-9999, pd.NA, inplace=True)
        self.df.dropna(inplace=True)
    return self.df
```

Checks that DataFrame is not empty or none, replaces all -9999 with pd.NA (-9999 is a placeholder for missing data), and drops the rows that contain missing values.

Data Collection and Preprocessing



Implemented normalization techniques - **Normalization** is the process of transforming data so that it's on a similar scale, this helps models train more effectively and ensures fair comparisons between values.

```
def normalize_temperature(self) -> pd.DataFrame:
    """Normalize the target column using the data values"""
    if self.target_column in self.df.columns:
        self.df[self.target_column] = (
            self.df[self.target_column] - self.df[self.target_column].mean()
        ) / self.df[self.target_column].std()
    else:
        print(f"'{self.target_column}' column not found ; no normalization")
    return self.df
```

We implemented normalization using the Z-score method. The formula subtracts the column's mean from each value and divides it by the standard deviation.

`self.df[self.target_column].mean()` gets the **average**.

`self.df[self.target_column].std()` gets the **standard deviation**.

The subtraction and division normalize the column.

Data Collection and Preprocessing



Data loading is the process of reading external data files (like CSVs) into your program so they can be analyzed or used by algorithms.

```
3
4 def load_data(self) -> pd.DataFrame:
5     """Load data from CSV, this takes the normal data name and renames the target column -potential"""
6     try:
7         print(f" Loading data from: {self.file_path}")
8         self.df = pd.read_csv(
9             self.file_path,
10            engine="python",
11            comment="#",
12        )
13
14        # Normalize column names
15        self.df.columns = [col.strip().lower() for col in self.df.columns]
16
17        # Rename standard data columns to match the wanted target column
18        if "value" in self.df.columns:
19            (parameter) self: Self@DataProcessor |target_column}, inplace=True)
20        elif
21            self.df.rename(columns={"anomaly": self.target_column}, inplace=True)
22
23        print(f" Loaded, Columns: {self.df.columns.tolist()}")
24    except Exception as e:
25        print(f"Error loading data from {self.file_path}: {e}")
26        self.df = pd.DataFrame()
27    return self.df
28
```

`pd.read_csv(self.file_path, engine="python", comment="#")`

- Loads the raw CSV file using the provided file path.
- Ignores comment lines (starting with #), which often contain metadata.

`self.df.columns = [col.strip().lower() for col in self.df.columns]`

- Standardizes all column names: removes leading/trailing spaces and converts everything to lowercase (e.g., " Value " → "value").

Automatically detects whether the column is labeled "value" or "anomaly" and renames it to match the desired target (e.g., "precipitation" or "temperature").



Innovative Algorithms

We developed:

- **Machine learning algorithm to predict future trends.**
- **Clustering algorithm to group similar data**
- **Time series analysis algorithm to detect anomalies**

We use custom linear regression in order to predict climate values

We use custom K-means like clustering to group climate data into points

We have an anomaly detection for the average

Innovate Algorithms

Custom Linear Regression:

It is coded to predict climate values such as temperature

```
class CustomTemperaturePredictor(BaseEstimator, RegressorMixin):
    """A temperature predictor for future treends"""
    def __init__(self, learning_rate: float = 0.01, n_iterations: int = 1000):
        """Initializes the predictor with rate and iterations"""
        self.learning_rate = learning_rate
        self.n_iterations = n_iterations
        self.weights = None # will be initied later
        self.bias = None

    def fit(self, X: np.ndarray, y: np.ndarray) -> 'CustomTemperaturePredictor':
        """Check for any NaN values on x and y"""
        if np.isnan(X).any() or np.isnan(y).any():
            print("NaN detected in input data so skipping training..")
            return self

        n_samples, n_features = X.shape
        # Init weights and bias to zeros
        self.weights = np.zeros(n_features)
        self.bias = 0

        for _ in range(self.n_iterations):
            y_predicted = np.dot(X, self.weights) + self.bias
            dw = (1 / n_samples) * np.dot(X.T, (y_predicted - y))
            db = (1 / n_samples) * np.sum(y_predicted - y)

            # Update our parameters
            self.weights -= self.learning_rate * dw
            self.bias -= self.learning_rate * db

        return self

    def predict(self, X: np.ndarray) -> np.ndarray:
        """Predict values using the parameters."""
        if self.weights is None or self.bias is None:
            raise ValueError("Model has not been trained yet so call fit() first.")
        return np.dot(X, self.weights) + self.bias
```

This is the CustomTemperaturePredictor class that is created in algorithms.py but it is used in CLI, Main, and in the website as well.

What it does:

This is a custom algorithm we built to temperature or precipitation based on historical data.

Gradient Descent:

Runs for **n_iterations** times:

- Predicts values using:
 $y = X \cdot \text{weights} + \text{bias}$
- Calculates error between predicted and actual values.
- Computes gradients and updates to reduce that error.



Innovate Algorithms

K-Means like Clustering:

A simplified k-means algorithm to group climate data points into clusters.

```
def custom_clustering(data: np.ndarray, n_clusters: int) -> np.ndarray:
    """Simple k-means-like clustering and return the labels."""
    # Randomly select initial centroids from data points
    centroids = data[np.random.choice(data.shape[0], n_clusters, replace=False)]
    for _ in range(10):
        distances = np.linalg.norm(data[:, np.newaxis] - centroids, axis=2)
        labels = np.argmin(distances, axis=1)
        centroids = np.array([data[labels == k].mean(axis=0) for k in range(n_clusters)])
    return labels
```

This is the custom_clustering that is created in algorithms.py but it is used in CLI, Main, and in the website as well for visualizing grouped climate behavior.

What it does:

This randomly initializes centroids, assigns labels based on closest centroid, then updates it based off of the information gathered.

Picks random points from the dataset to act as initial cluster centers (centroids).

Calculates the distance from every point to every centroid using Euclidean distance.

Helps figure out which centroid each point is closest to.



Innovate Algorithms

Anomaly Detection with Moving Average:

This is a method used to identify unusual climate data points.

```
def detect_anomalies(time_series: np.ndarray, window_size: int = 10, threshold: float = 2.0) -> np.ndarray:
    """Detect anomalies using a moving average and threshold."""
    moving_avg = np.convolve(time_series, np.ones(window_size) / window_size, mode='valid') # Compute the moving average over window_size
    padded_avg = np.pad(moving_avg, (window_size - 1, 0), mode='edge')
    # Flag anomalies
    anomalies = np.abs(time_series - padded_avg) > threshold * np.std(time_series)
    return anomalies
```

This is the `detect_anomalies` function. It compares each value to a rolling average and flags it if the difference is too large using standard deviation. Used in Main, CLI, and website.

What it does:

Establishes baseline of normal in the time series using a rolling average over `window_size`

Then, it flags data points that deviate too far from the moving average.

The threshold is $2 * \text{standard deviation}$.

Returns an array like `[False, False, True, False...]`

Each `True` means that it point is an anomaly.



Data Visualization

For Data Visualization, we have plots that were made using the Matplotlib for creating informative plots

- Custom technique for displaying multidimensional climate data

Animated visualization showing climate change over time.

Did the bonus and innovated the data visually.

When it is run, 2 line plots are made, including the actual and the predicted trends over time.

There is a scatter plot that is made that depicts the clustered data

Highlighting anomalies using markers to draw attention to significant things that are different.

Interactive Animations:


This provides an engaging way to interpret complex data, facilitating clearer understanding and effective communication.



Data Visualization

We built our own visualizations using Matplotlib and Seaborn so users can actually see the climate data story and not just the numbers.

We use data visualization to make complex climate data easy to understand, the user is able to see::

- 
- How accurate our predictions are
 - How clusters are formed in the data
 - Where anomalies/spikes occur

What it does and where it is used:

Prediction VS Actual - Line Plot with Markers

Clustering - Scatter Plot with Color

Anomaly Detection - Line Plot with the OUTLIERS (anomalies)

(Bonus) - Animation - Matplotlib animation

All made on visualizer.py and called on CLI, Main, and even Website



Data Visualization

Prediction VS Actual - Line Plot with Marker:

```
class Visualizer:
    """The class that plots the trends"""
    @staticmethod
    def plot_trend(years, actual, predicted, show: bool = True):
        """Plot actual vs predicted value over time"""
        # Convert to list if they ain't already (avoid NumPy truth value issues)
        years = list(years)
        actual = list(actual)
        predicted = list(predicted)

        # If there's no data quits and tell user
        if not years or not actual or not predicted:
            print("Not enough data for trend.")
            return

        # Create a figure and plot both actual and predicted data
        plt.figure(figsize=(10, 5))
        plt.plot(years, actual, label="Actual", marker='o')
        plt.plot(years, predicted, label="Predicted", linestyle="--", marker='x')
        plt.xlabel("Year")
        plt.ylabel("Normalized value")
        plt.title("Climate Trend over time")
        plt.legend()
        plt.grid(True)
        plt.tight_layout()
        plt.show()
```

This function visually compares the actual climate values to the predicted values over time.

Converts input to Python lists

Plots actual values using circles

Plots predicted values using dashed lines and x markers

Adds labels, title, legend, and grid for clarity

Allowing this to be visual helps by showing whether the prediction model is accurate
Makes it easy for non-technical users to understand the model's performance

Data Visualization

Clustering - Scatter Plot with Color

```
@staticmethod
def plot_clusters(data, labels, show: bool = True):
    """Plot clustered data with different color for each."""
    if not data or not labels:
        raise ValueError("Input lists cannot be empty")

    # Check if data is 1D or 2D and set up x and y
    if len(data[0]) == 1: # 1D data
        x_data = [d[0] for d in data]
        y_data = [0] * len(x_data)
    else: # 2D
        x_data = [d[0] for d in data]
        y_data = [d[1] for d in data]

    #scatter plot if clusters
    plt.figure(figsize=(10, 6))
    plt.scatter(x_data, y_data, c=labels, cmap='viridis', label="Clusters")
    plt.xlabel('Feature 1')
    plt.ylabel('Feature 2')
    plt.title('Clustered data')
    plt.legend()
    plt.show()
```

This function visually displays clusters of climate data, showing how different data points group together based on similarity.

Started off first with error handling

Checks if the data is one-dimensional (e.g., only "year") - IF NOT IT SPLITS IT UP TO X AND Y

```
plt.scatter(x_data, y_data, c=labels,
            cmap='viridis')
```

Plots a scatterplot where each point is color-coded based on its cluster label.

viridis colormap for nice visual distinction.

Data Visualization

Anomaly Detection - Line Plot with the OUTLIERS (anomalies)

```
@staticmethod
def plot_anomalies(time_series: List[float], anomalies: List[bool], show: bool = True) -> None:
    """Plot time series with detected anomalies"""
    if not time_series or not anomalies:
        raise ValueError("Input lists are not allowed to be empty")

    plt.figure(figsize=(10, 6))
    plt.plot(time_series, label='Time Series')
    # Mark anomalies (non-anomalies shown as NaN)
    plt.plot(np.where(anomalies, time_series, np.nan), 'ro', label='Anomalies')
    plt.xlabel('Time')
    plt.ylabel('Normalized Value')
    plt.title('Anomaly Detection in Time Series')
    plt.legend()
    plt.grid(True)
    plt.tight_layout()
    plt.show()
```

This function highlights unusual or extreme data points, useful in climate analysis for identifying outliers in trends.

```
plt.plot(time_series, label='Time Series')
```

Plots the entire timeline of climate data

```
plt.plot(np.where(anomalies, time_series,
np.nan), 'ro', label='Anomalies')
```

Uses np.where() to only plot red dots (ro) where anomalies are True.

Non-anomalies are replaced with np.nan which does not plot them

Helps scientists focus attention on specific years or regions that cause problems

Data Visualization

(Bonus) - Animation - Matplotlib animation

```
@staticmethod
def animate_temperature_comparison(actual, predicted, show: bool = True):
    """Animate the comparison between actual and predicted values"""
    import matplotlib.pyplot as plt
    from matplotlib.animation import FuncAnimation
    import numpy as np

    if len(actual) != len(predicted):
        print("Mismatched lengths so cant animate")
        return

    print("Generating animation")

    #sets up the figure for animation
    fig, ax = plt.subplots()
    x = list(range(len(actual)))
    ax.set_xlim(0, len(actual))
    ax.set_ylim(min(min(actual), min(predicted)), max(max(actual), max(predicted)))
    ax.set_title("Actual vs Predicted Over Time")
    ax.set_xlabel("Time")
    ax.set_ylabel("Value")

    line_actual, = ax.plot([], [], label="Actual", color="blue")
    line_predicted, = ax.plot([], [], label="Predicted", color="orange")
    ax.legend()

    def init():
        """initializer for animation """
        line_actual.set_data([], [])
        line_predicted.set_data([], [])
        return line_actual, line_predicted

    def update(frame):
        """Updates the function for the frame """
        line_actual.set_data(x[:frame], actual[:frame])
        line_predicted.set_data(x[:frame], predicted[:frame])
        return line_actual, line_predicted
```

This function creates an animated line chart showing how actual and predicted climate values evolve over time

Initializes the animation figure and sets the plot boundaries based on data range.

```
line_actual, = ax.plot([], [], label="Actual",
color="blue")
line_predicted, = ax.plot([], [], label="Predicted",
color="orange")
```

Prepares two lines: one for actual values, one for predictions, drawn frame by frame.

```
ani = FuncAnimation(... interval=100,
repeat=False)
```

Tells Matplotlib to draw 1 new frame every 100ms until all points are plotted.



Main

The purpose of main.py is to visualize the project as something the user can easily use to navigate through everything.

1. `files = list_data_files()`
 - a. Recursively finds all .csv files in the data directory
 - b. Displays them with a number so the user can select one.
2. `target_column = auto_detect_target_column(...)`
 - a. Looks at both the file name and its headers.
 - b. Automatically detects whether it should use "temperature", "precipitation", or "anomaly" to analyze.
3. Preprocess the Data
 - a. Cleans the data (removes -9999 and NaNs).
 - b. Normalizes values so the models work better.
 - c. Extracts features like **year** and **month** as inputs.
4. Runs the prediction
 - a. Shows the first 5 predictions vs actual values.
 - b. Visualizes it as a line plot + animation.



Main

5. `labels = custom_clustering(X, n_clusters=3)`

`Visualizer.plot_clusters`

- a. Groups the data into 3 clusters.
- b. Helps identify regional or seasonal patterns in the data.
- c. Colored Scatter Plot

6. Anomaly Detection:

`anomalies = detect_anomalies(y, window_size=3, threshold=1.5)`

`Visualizer.plot_anomalies`

- a. Flags data points that significantly deviate from the moving average

7. `print("\n Done, all steps completed.")`

- a. Everything is plotted and visualized



CLI – Command Line Interface

The CLI is designed for users who prefer working directly in the terminal.

- Load specific climate datasets
- Choose a desired action: predict, cluster, or anomalies
- Run everything via command-line flags, where there is no user interface

An example command:

```
python3 -m src.cli --folder precipitation --file precipAsiaNOAA.csv --action predict --target_column precipitation
```

Our CLI tool lets users skip the interface entirely and work straight from the command line. That's really useful in scientific workflows or automation scripts, especially if you're analyzing multiple datasets in a loop or doing large-scale modeling.



Unit Testing

Automated tests using unittest framework:

We do the unit tests for data processing and each algorithm, and the bonus entire data Pipeline tests to ensure overall functionality.

Demonstrate Unit Testing Live:

```
FAILED (failures=2)
(venv) sahil_tyre@Sahils-MacBook-Air climate_change_analyzer % coverage report
Name                               Stmts  Miss  Cover
-----
src/__init__.py                     0      0   100%
src/algorithms.py                   39      3    92%
src/data_processor.py               43      1    98%
src/visualizer.py                   82     29    65%
tests/test_algorithms.py            22      2    91%
tests/test_data_processor.py        62      2    97%
tests/test_pipeline.py              42      0   100%
tests/test_visualizer.py            27      2    93%
TOTAL                               317     39    88%
```

In our case, we wrote automated tests to validate:

- Data loading and preprocessing
- Each algorithm (prediction, clustering, anomaly detection)
- Visualizer logic
- End-to-end pipeline functionality

We used Python's built-in unittest module and tracked coverage using the coverage tool.

coverage run -m unittest discover tests
coverage report



Website

As a bonus, we built a Flask-based web application so users can interact with climate data without needing to touch any code.

How it works:

- Simple file selection
- Dropdown actions
- One-click results
- Automatically generated graphs

We use **python3 -m website.app** on the terminal to start the website.

Launches a local web server at <http://127.0.0.1:5000>

Select a dataset from the dropdown (CSV files from the /data)

Choose an action (predict, clustering, and anomaly detection)

Click Run Analysis to view results

Website

The site uses the same backend logic as the CLI

- Loads and cleans the data
- Runs the chosen algorithm
- Generates and displays plots using `matplotlib`

Climate Change Impact Analyzer

Select a data file:

precipitation/precipAfricaNOAA.csv

Choose Action:

- ✓ Predict
- Cluster
- Anomalies

Climate Change Impact Analyzer

Select a data file:

precipitation/precipAfricaNOAA.csv

Choose Action:

Predict

Run Analysis

Climate Change Impact Analyzer

Select a data file:

✓ precipitation/precipAfricaNOAA.csv
precipitation/precipAntarcticNOAA.csv
precipitation/precipArcticNOAA.csv
precipitation/precipAsiaNOAA.csv
precipitation/precipAtlanticNOAA.csv
precipitation/precipCaribbeanNOAA.csv
precipitation/precipEuropeNOAA.csv
precipitation/precipGlobalNOAA.csv
precipitation/precipGulfNOAA.csv
precipitation/precipHawaiiNOAA.csv
precipitation/precipJapanNOAA.csv
precipitation/precipNorthNOAA.csv
precipitation/precipOceanicNOAA.csv
precipitation/precipPacificNOAA.csv
precipitation/precipSANDIA.csv
precipitation/precipSouthNOAA.csv
temperature_anomaly/tempAfricaNOAA.csv
temperature_anomaly/tempAntarcticNOAA.csv
temperature_anomaly/tempArcticNOAA.csv
temperature_anomaly/tempAsiaNOAA.csv
temperature_anomaly/tempAtlanticNOAA.csv
temperature_anomaly/tempCaribbeanNOAA.csv
temperature_anomaly/tempEuropeNOAA.csv
temperature_anomaly/tempGlobalNOAA.csv
temperature_anomaly/tempGulfNOAA.csv
temperature_anomaly/tempHawaiiNOAA.csv
temperature_anomaly/tempJapanNOAA.csv
temperature_anomaly/tempNorthNOAA.csv
temperature_anomaly/tempOceanicNOAA.csv
temperature_anomaly/tempPacificNOAA.csv
temperature_anomaly/tempSANDIA.csv
temperature_anomaly/tempSouthNOAA.csv



User Interface

The user interface was made in 2 purposes, the CLI and the Website.

1. CLI
 - a. For users who want direct control
 - b. Uses flags like `--action`, `--file`, `--target_column`
 - c. Output shown in the terminal and charts open automatically
 - d. Great for automation, scripting, or batch testing
2. Web Interface on Flask
 - a. User-friendly graphical interface
 - b. Select a file
 - c. Choose action from a dropdown to predict, cluster, anomaly detection
 - d. Generates plots and displays results
 - e. `http://127.0.0.1:5000`

Climate Change Impact Analyzer

Select a data file:
precipitation/precipAfricaNOAA.csv

Choose Action:
Predict

Run Analysis

```
e.app.py : [Errno 2] No such file or directory
(venv) sahil_tyre@wc-dhcp58d056 climate_change_analyzer % python3 -m src.cli --folder temperature_anomaly --file tempHawaiiNOAA.csv --action predict --target_column anomaly

Loading data from: data/temperature_anomaly/tempHawaiiNOAA.csv
Loaded, Columns: ['year', 'anomaly']

Predictions: [-0.40340599 -0.3858666 -0.36832721 -0.35078782 -0.33324843]
Actual:      [-0.82125981 -0.58607802  0.03127417  0.14886507 -0.056919 ]
Generating animation
```



Explain Data

So as we know, All data was collected from NOAA (National Oceanic and Atmospheric Administration)
All the CSV files were organized into folders like:

- `precipitation/`
- `temperature_anomaly/`

Data:

- `Year` column (always present)
- `Month` (sometimes(not in this case, but have it just in case))
- `Value` column (renamed to match: temperature, precipitation, or anomaly)

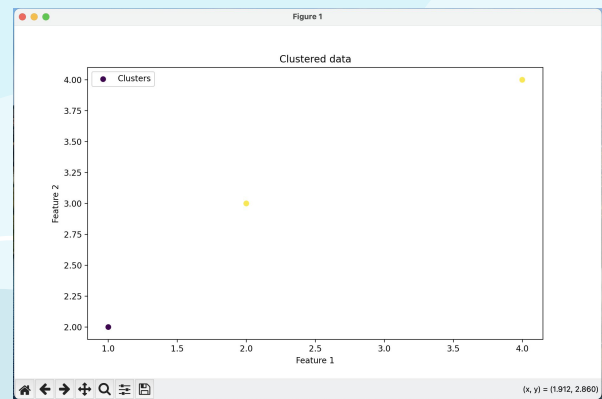
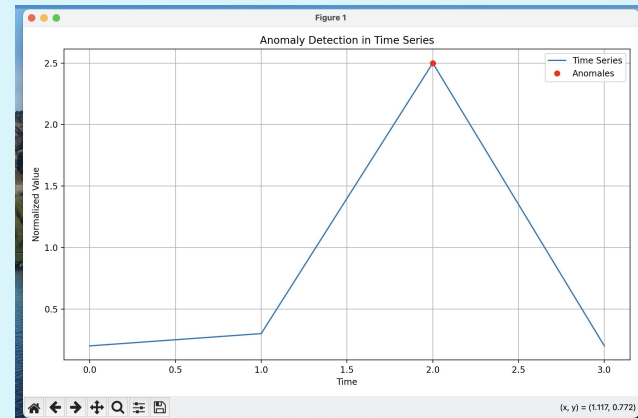
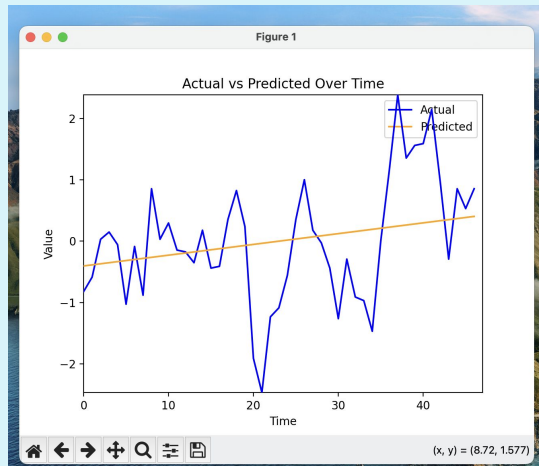
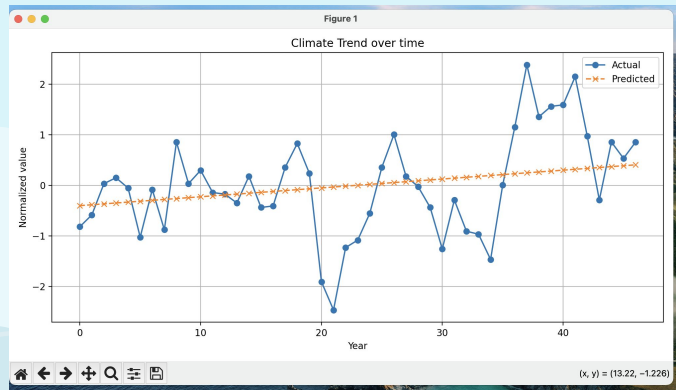
We normalized all values to standardize across regions

This allowed our real world testing of our prediction and anomaly models, which gave us clear patterns to plot trends and clusters



Explain Data

Examples:





Live Demonstration

Now, we will show you the data we came up with and our smooth user interaction, live! !

