Climate Change Impact Analyzer Report

Climate change is one of humanity's most pressing problems, affecting our environments globally and influencing overall human welfare. Our project is a Python tool designed to provide insight into climate change data, focusing on temperature anomalies, precipitation patterns, and unusual climate change events across different regions. We utilized datasets from reliable sources like NOAA, which ensure extensive coverage and analysis.

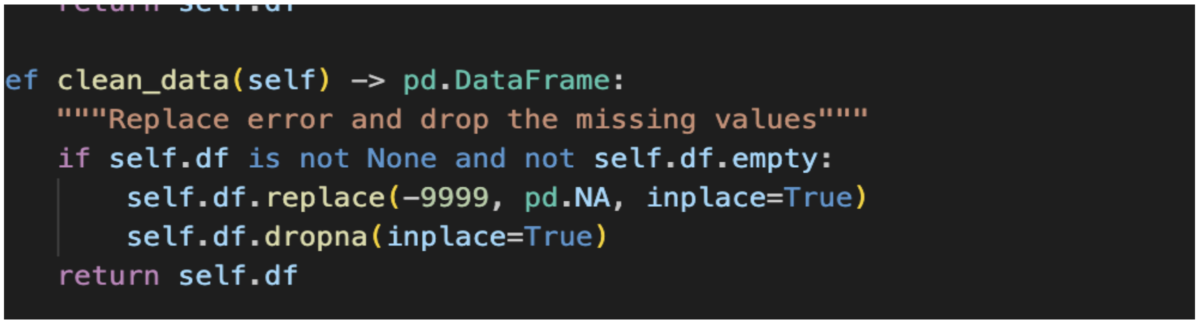
Our main objectives for the project were:

- Collect reliable data and Preprocess it to ensure accurate, clean, and reliable data
- Predict climate change parameters using machine learning methods.
- Identify natural clusters or patterns within climate data across different geographical regions.
- Detects anomalies in climate change trends that indicate data irregularities.
- Visualize our data in a simple and understandable way.

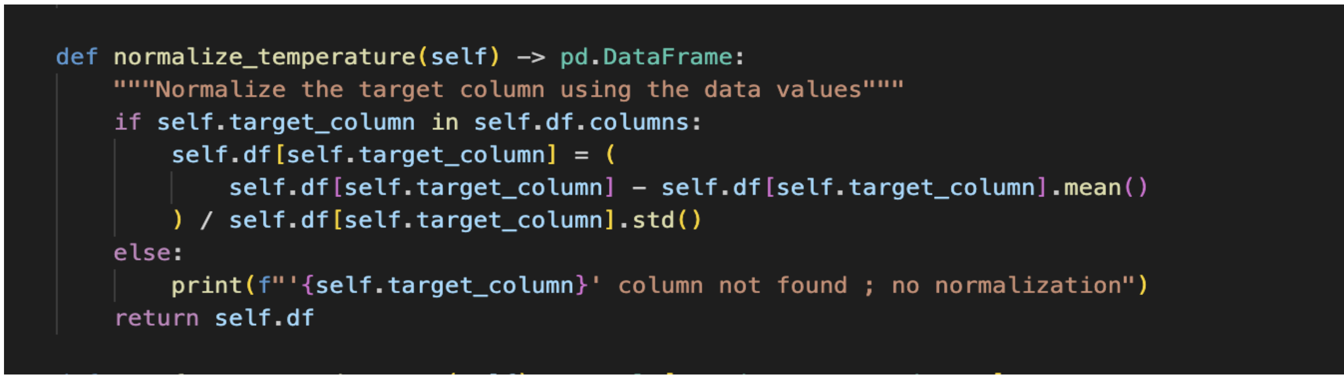
Throughout this report, we outline our methodology, discuss our rationale for specific choices/ algorithms, and highlight key findings.

Data Collection and Preprocessing

The accuracy of our analysis depends on our data quality, so we chose datasets from the top sources about climate change, such as the NOAA. They provide extensive data from regions across the world. Each dataset was put into a CSV file and structured with columns such as 'Year' and 'Month' and variables like 'Temperature' or 'Precipitation.' Our Data Collection and Preprocessing file cleans the data, normalizes it, and loads it.



```
ef 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
```



```python
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
```

These are examples of important functions in our Data Collection and Preprocessing file. As you can see, clean_data checks that the DataFrame is not empty or none. It also replaces -9999 with pd.NA (-9999 is a placeholder for missing data) and drops rows that contain missing data values.

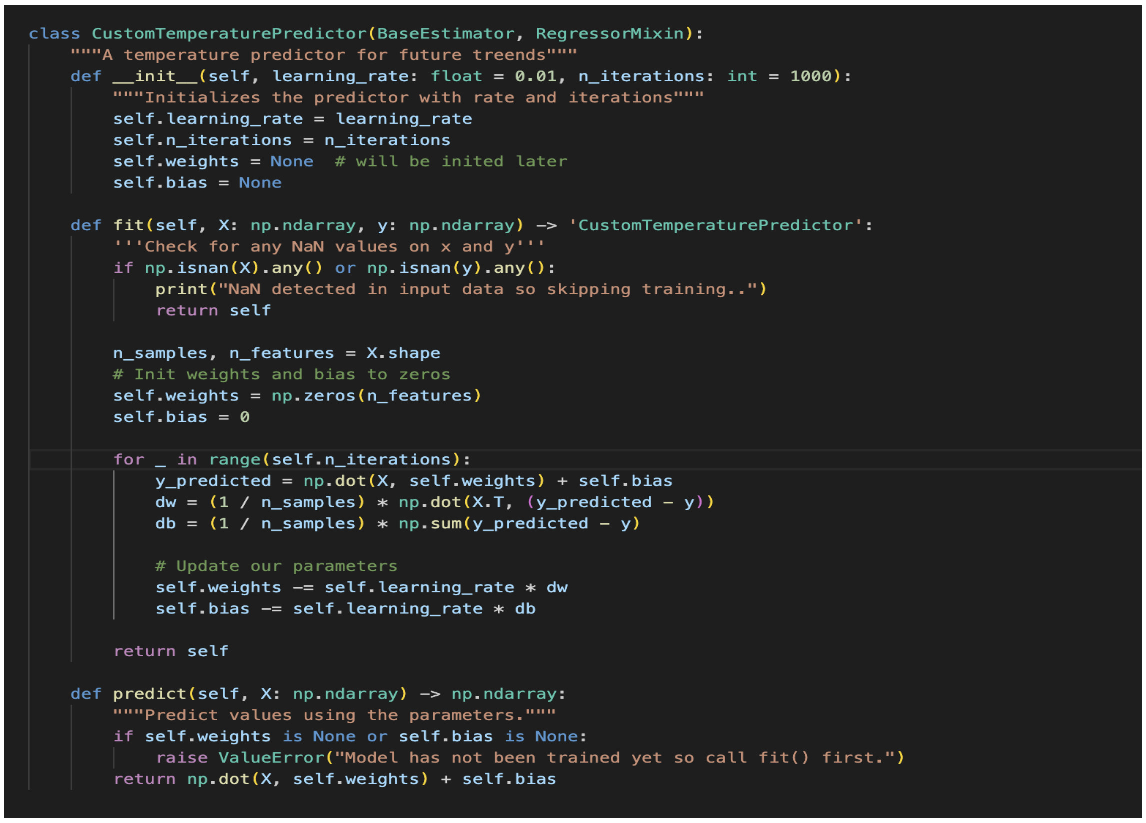
The function normalize_temperature processes the data on a similar scale. Here, we implemented the Z-score method with the formula subtracting the column's mean from each value and dividing it by the standard deviation.

Innovative Algorithms

Throughout this portion of the project, we developed:

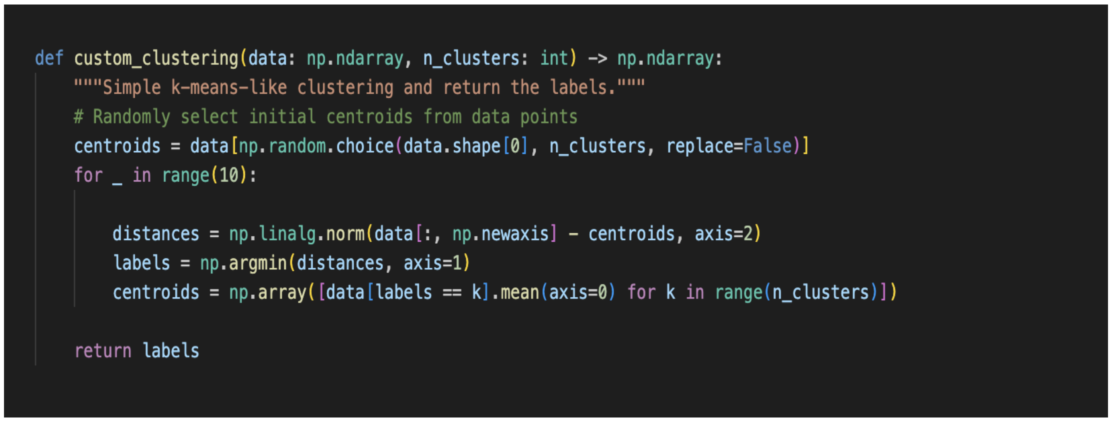
- Machine learning algorithms to predict future trends
- Cluster algorithm to group similar data
- Time series analysis algorithm to detect anomalies

In developing these functions, we used custom linear regression to predict climate change values.



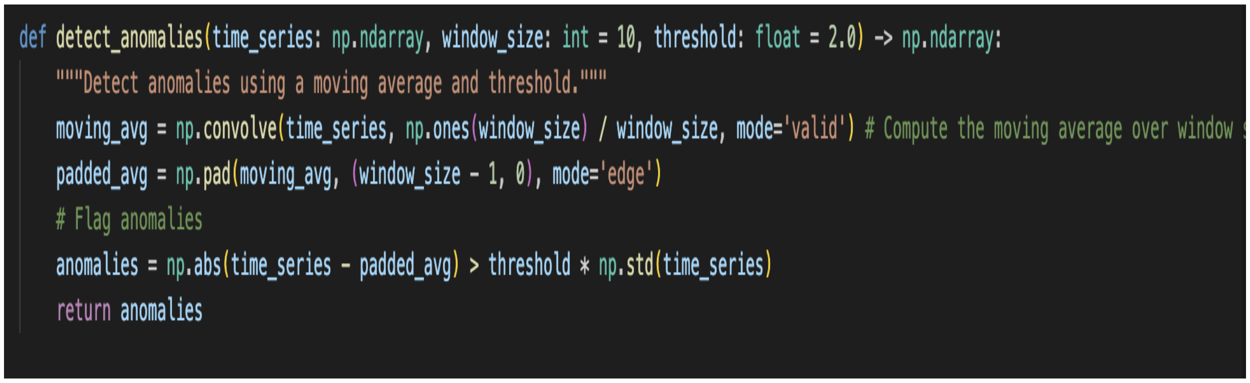
```python
class CustomTemperaturePredictor(BaseEstimator, RegressorMixin):
    """A temperature predictor for future treends"""
    def __init__(self, learning_rate: float = 0.01, n_iterations: int = 1000):
        """Initializes the predictor with rate and iterations"""
        self.learning_rate =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

We built this custom algorithm, CustomTemperaturePredictor, to predict temperature or precipitation based on historical data. It uses gradient descent to update its weight and bias to minimize the error between predicted and real values. The model checks for missing values, initializes the parameters, and runs multiple training interrelations to come up with the best prediction.



```python
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 custom_clustering function implements a simple k-means clustering algorithm to group similar data points. It randomly selects centroids from the dataset and then refines them over 10 iterations. Each iteration calculates the distance between them and assigns each data point to the nearest centroid. This method groups similar climate change data patterns and helps identify similar climate change characteristics for our data.

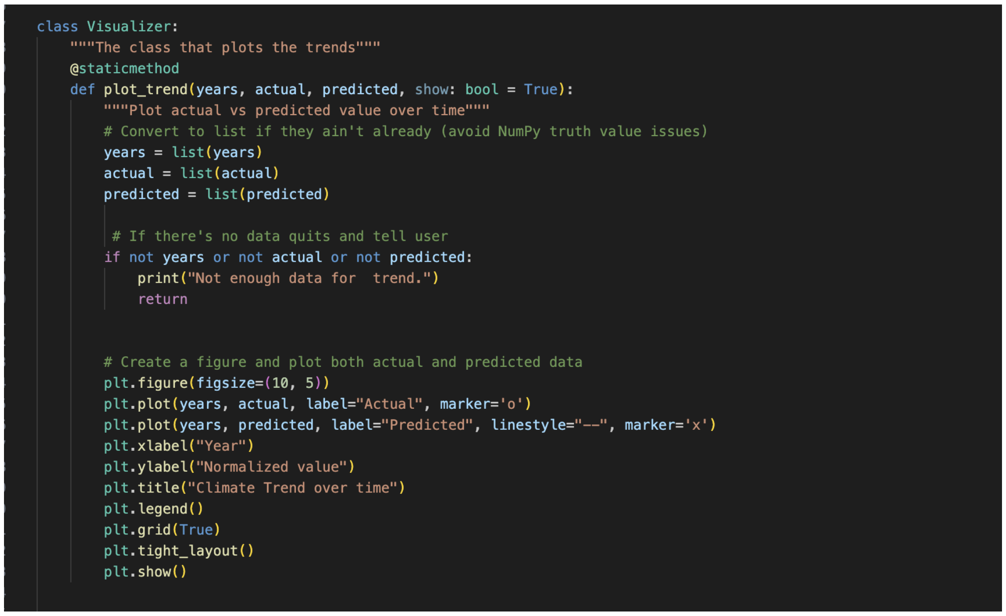


```python
def detect_anomalies(time_series: np.ndarray, window_size: int = 10, threshold: float = 2.0) -> np.ndarray:
    """Detect anomalies using a moving average and threshold."""
    moving_avg = np.convolve(time_series, np.ones(window_size) / window_size, mode='valid') # Compute the moving average over window
    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
```

Another important part of our innovative algorithms is anomaly detection, which compares values to the average and flags if the difference is large using standard deviation. It returns an array with each element being true or false; with true meaning, it points to an anomaly.

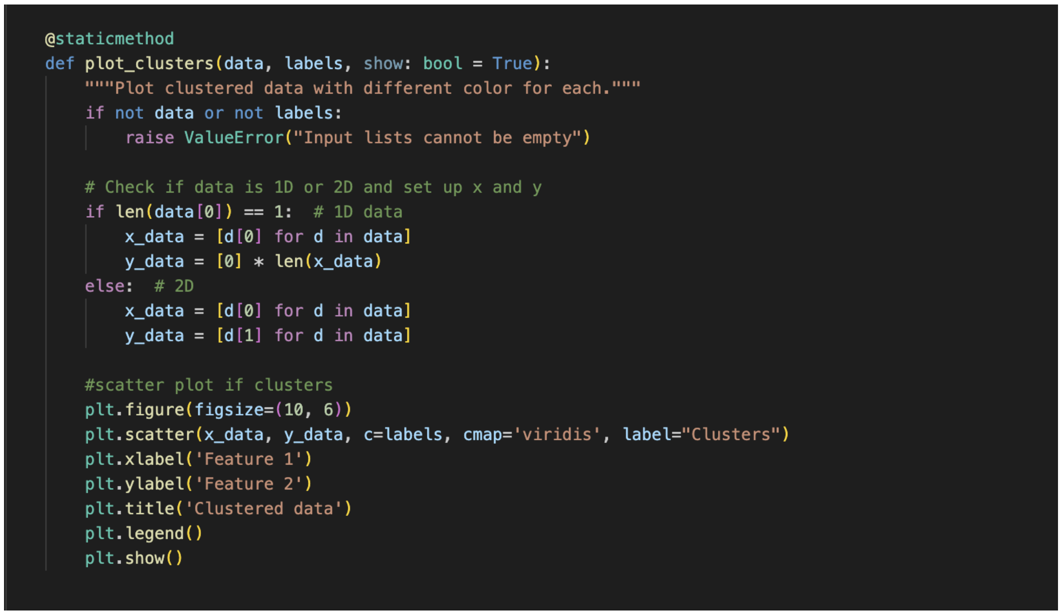
Data Visualizations

Throughout this portion of the project, we used Matplotlib to create interactive visualizations and implemented a visualization technique to display our data animatedly to show the changes over time.



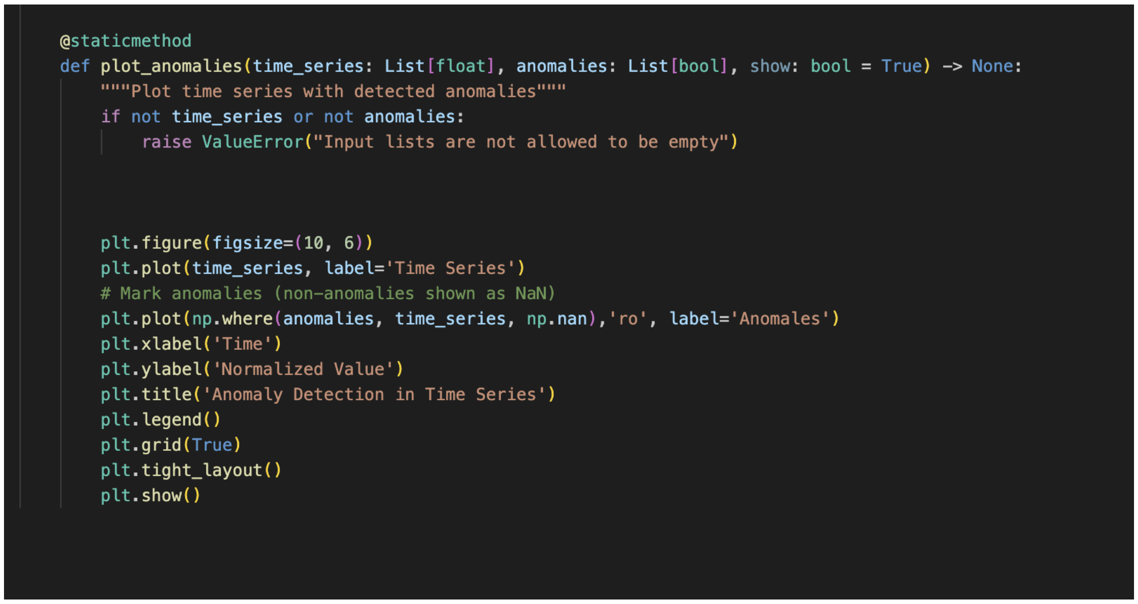
```python
class Visualizer:
    """The class that plots the trends"""
    @staticmethod
    def plot_trend(years, actual, predicted, show: bool = True):
        """Plot actual vs predicted value over time"""
        # Convert to list if they ain't already (avoid NumPy truth value issues)
        years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

The plot_trend function compares the actual climate change values to the predicted values and converts the input to a Python list. It plots the real values using circles and predicts using x. It also adds labels, titles, and more, as you can see.



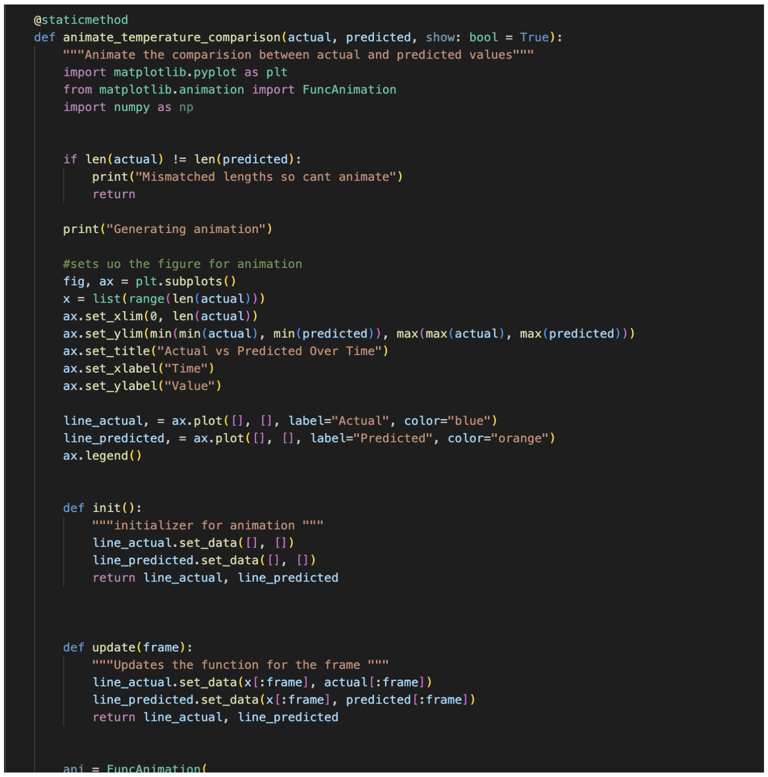
```python
@staticmethod
def plot_clusters(data, labels, show: bool = True):
    """Plot clustered data with different color for each."""
    if not data or not labels:
        raise ValueError("Input lists cannot be empty")

    # 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()
```

The plot_clusters visually shows us the clusters of climate change and how different data is grouped together based on similarity. It starts off with error handling and checks if the data is one-dimensional. If it's not, it splits it into X and Y, and then it plots a scatterplot where each point is color-coded.



```python
@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='Anomales')
    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 data and helps identify outliers. Plt.plot(time_series, label=' TimeSeries') plots the entire timeline of climate change, and the second plt.plot uses np.where() to only plot red dots where anomalies are true.

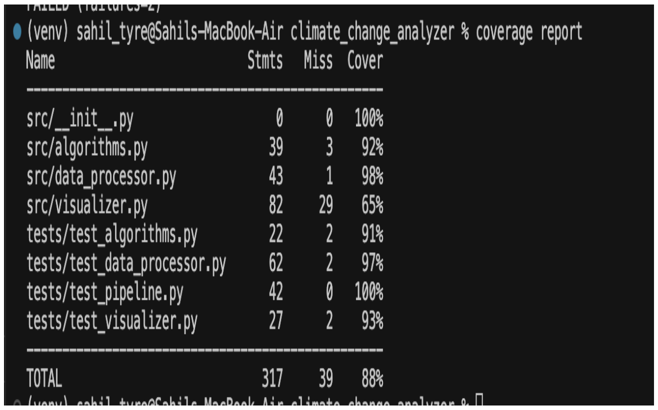


```python
@staticmethod
def animate_temperature_comparison(actual, predicted, show: bool = True):
    """Animate the comparision between actual and predicted values"""
    import matplotlib.pyplot as plt
    from matplotlib.animation import FuncAnimation
    import numpy as np


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

    print("Generating animation")

    #sets uo 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


    ani = FuncAnimation(
```

This function attempted the extra credit by creating an animated line chart showing the actual vs. predicted climate value. As you can see here, this function animates the figure and sets plot boundaries based on the range of data. For example, line_actual and line_predicted prepare two lines for the actual and predicted predictions to be drawn.

Unit Testing

We did unit tests for data processing and each algorithm and included the Pipeline test to ensure our overall functionality. We used pythons built-in unittest module to track coverage using: *coverage run -m unittest discover tests*
coverage report
<u>Example:</u>



```
(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%
```

Command Line Interface

The CLI is for users who prefer working in the terminal. It loads specific datasets, gives options for desired actions like predicting clusters or anomalies, and runs everything via command-line flags. This is useful in scientific workflows, especially when analyzing multiple datasets.

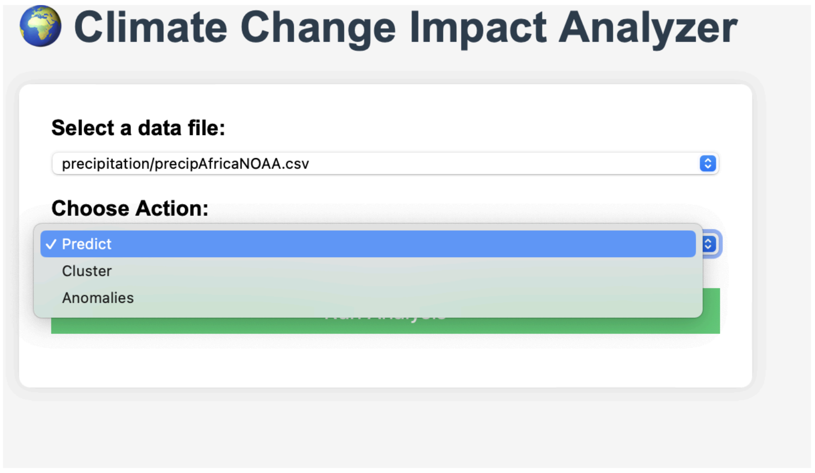
An example command:
python3 -m src.cli --folder precipitation --file precipAsiaNOAA.csv --action predict --target_column precipitation

Website

As a bonus, we built a Flask-based website so users can interact with the climate change data without needing to touch code. This helps simplify the user experience. It works with simple file selection, dropdown actions, one-click results, and automatically generated graphs. We start the website with
python3 -m website.app on the terminal and launch http://127.0.0.1:5000

The website uses backend logic as the CLI, loads/cleans data, runs the chosen algorithm, and displays the plots using matplotlib.

Shown below is an example of the user choosing an action:



Climate Change Impact Analyzer

Select a data file: precipitation/precipAfricaNOAA.csv

Choose Action: ✓ Predict

Main

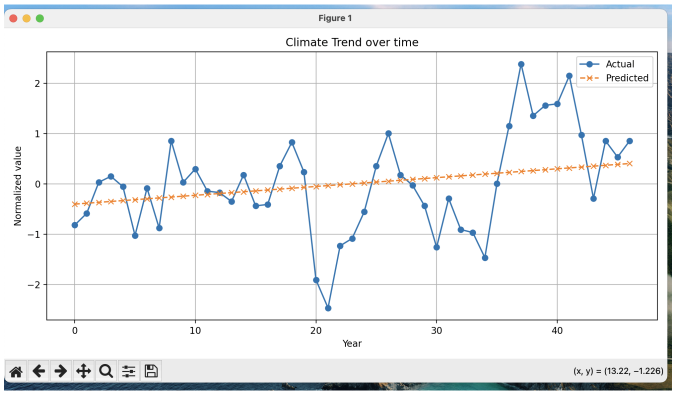
The purpose of main is to visualize the project and make it so the user as a smooth interaction.
We have many sections in the main, like files=list_data_files(), which displays the .csv files so the user can select one. Then there is target_colum = auto_detect_target_colum(..), which detects what variable we are analyzing (temperature, precipitation, anomaly). Then, we preprocess the data as explained earlier by cleaning, normalizing, and loading. It runs the prediction by showing five predictions vs actual values and visualizes it as a plot and an animated version. Then there is labels = custom_clustering(X, n_clusters=3). Visualizer.plot_clusters, which groups the data into 3 clusters, helps identify data patterns and does the color scatter plot. The Main also includes anomaly detection, which flags data points that are far different from the average and then prints everything and visualizes it.

Why These Algorithms?

We chose simple and interpretable algorithms for this project to focus on clarity and ease of visualization. For climate prediction, we did a custom linear regression model that provides a relationship between variables like a year, month, and temperature, making it easy to explain and easier for the models to learn and make predictions. We used the k-means approach for clustering because it is straightforward and works well for grouping similar patterns without required data. We also used moving average anomaly because it effectively identifies spikes or drops in variables while being easy to use and interpret. All our algorithms were balanced between accuracy and simplicity, making it ideal for completing this project and demonstrating machine learning and data analysis.

Explain Data

As explained earlier, the Data was collected from NOAA, put into CSV files, and organized into Columns like Year and Month. We normalized the values across the regions, which helped our algorithms generate proper anomalies and predictions, giving us clear patterns to plot.



This visualization is an example of how our data is presented. It shows the overall climate change trend, with the orange line showing the predicted trends using our linear regression model. These results help visualize and interpret climate change patterns.

Throughout our analysis we found several key insights from the climate change data processed. Our linear regression model captured the overall increase in temperature over time which aligns with real world climate change patterns. The clustering algorithm grouped similar data and helped us identify seasons, regions, and/or patterns with similar temperature behavior. Our anomaly detection algorithm flagged extreme temperature spikes that deviated from the moving average and could represent potential extreme weather events or data errors. Overall, our results show that even with simple data, meaningful patterns and trends could be identified, visualized and explained.

To sum it up, the Climate Change Impact Analyzer successfully shows how data and machine learning can be combined to analyze climate change data and get important insight. Through data preprocessing, algorithm development, and clear visuals, we were able to predict trends, detect anomalies, and identify natural clusters within climate patterns. This tool could provide valuable insight for researchers, policymakers, and people concerned about climate change.