Predicting Temperature Anomalies:
A Comparative Approach using Non-Neural and Neural Network Models

Davood Makvandi

Dissertation submitted to International Business School for the partial fulfilment of the requirement for the degree of MASTER OF SCIENCE IN IT FOR BUSINESS DATA ANALYTICS

May 2023

DECLARATION

This dissertation is a product of my own work and is the result of nothing done in collaboration.

I consent to International Business School’s free use including/excluding online reproduction, including/excluding electronically, and including/excluding adaptation for teaching and education activities of any whole or part item of this dissertation.

Signature

[illegible]

Davood Makvandi

Word length: 14340 words

ACKNOWLEDGEMENTS

I am immensely grateful to my professors, especially Prof. Zsófia Gyarmathy, whose exemplary guidance, invaluable feedback, and unparalleled knowledge have been instrumental in the successful completion of this project.

To my loving wife, Sana, I express my profound gratitude. Her unwavering faith, enduring patience, and ceaseless encouragement have been my rock, offering me the strength and determination to persevere during the most challenging times. Completing this journey would have been an impossible task without her by my side.

My heartfelt appreciation goes to my parents and family, whose incessant support has been my pillar of strength throughout my academic journey. Their belief in my potential, along with their enduring love and encouragement, have consistently inspired me to strive for excellence.

Finally, I would like to extend my gratitude to my friends and classmates. Their camaraderie, ready assistance, and enduring motivation have played a pivotal role in sustaining my morale and drive throughout this journey. Their unwavering support and collaborative spirit have fostered an environment conducive to both personal growth and academic achievement. I am forever indebted to them for their companionship and resilience in the pursuit of our shared goals.

Abstract:

Climate change is one of our most significant environmental challenges, with far-reaching impacts on human health, the economy, and the natural world. This study was conducted with the objective of developing a predictive model capable of accurately forecasting temperature anomalies three months ahead. To achieve this, a rigorous methodology was adopted which involved comprehensive data preprocessing and application of three distinct machine learning models: Multi-linear regression, Random Forest, and Long Short-Term Memory (LSTM). This approach ensured a thorough investigation into the efficacy of these models, taking into consideration their individual strengths and limitations. The LSTM model emerged as the most suitable solution to the problem, outperforming the other two models in accuracy and capacity to handle temporal dependencies in the data. This underlines the potential of LSTM in time series forecasting tasks. However, the study was not without limitations. The LSTM model's performance may have been restricted by the limited size of the dataset. Future research with larger datasets could provide more insight into the model's scalability and overall performance. Reflecting on this project offered invaluable insights into the critical role of meticulous data preprocessing, selection, and evaluation of appropriate models, and understanding the task at hand in achieving the desired outcomes. These learnings will guide future work in team environments and other data analytics projects, emphasizing the need for a thoughtful approach to problem-solving.

Table of Contents

Table of Figures

List of Tables

1. Introduction:

Climate change remains an unequivocal global predicament that permeates political frontiers and sectors. The issue presents a formidable challenge to humankind, warranting an in-depth comprehension of historical, contemporary, and prospective climate situations. This study is oriented towards the development of a proficient predictive model capable of forecasting temperature anomaly values for Berlin with a lead time of three months. The utilized dataset is expansive, covering a period from 1900 to 2020, and contains diverse features, including greenhouse gas emissions, temperature anomalies, natural disasters, and shifts in global mean temperature.

The principal aim is the construction of predictive models that can foresee climate conditions and facilitate strategic planning by predicting temperature anomalies. To achieve this, the data undergoes a rigorous preprocessing stage, ensuring robustness through cleaning and handling missing values. Potential outliers are scrupulously identified and analyzed to prevent unwarranted bias in the subsequent models. A meticulous feature selection process follows, pinpointing the most influential attributes for model development.

This research involves the integration, assessment, and comparison of a series of machine learning models, including Multi-linear regression, Random Forests, and a neural network variant, Long Short-Term Memory (LSTM). With its ability to recognize and memorize long-term dependencies and patterns within data, LSTM is particularly adept at time series forecasting. The performance of these models will be evaluated through several key metrics such as Root Mean Squared Error (RMSE), Mean Absolute Error (MAE), and the R-squared value.

The chosen models will predict temperature anomaly values three months ahead. This involves feeding the model with data up to time t, and subsequently forecasting the temperature anomaly at time t+3 months. The reliability of these predictions will be validated by comparing them with actual temperature anomaly values from the test dataset. Through this comprehensive approach, the study aims to augment the broader dialogue on climate change, providing a model that aids in the anticipation of future climatic conditions.

2. Data Preparation

The raw data required for this study was downloaded and subsequently imported into a Pandas DataFrame. A thorough initial scroll-through of the data was conducted to understand its structure, quality, and potential issues.

One of the crucial steps in data preparation is the identification and management of missing values. We employed various methods for this purpose, ranging from data deletion to data imputation. This choice was guided by the nature of the data, the number of missing values, and the implications of the chosen method on the model's potential performance.

After handling missing data, we turned our focus towards outlier detection, an important aspect of data cleaning that could potentially skew the outcomes of our analysis. We employed visual techniques such as boxplots and scatter plots, as well as statistical methods, to detect any aberrant values. Once identified, we made informed decisions on how to address these outliers, depending on their nature and potential impact on the models. They were either corrected if they were due to data entry errors, or removed if they were legitimate but extreme values, or retained in the dataset if they were significant for the analysis. The combination of these meticulous data preparation steps has laid the foundation for reliable and robust machine learning models.

Import the required libraries.

This project utilized various Python libraries to perform data analysis, machine learning, and data visualization. The Pandas library was utilized for manipulating data, and the numpy library was utilized for numerical computing. The scipy.stats module was utilized to calculate distribution skewness, and the sklearn.ensemble module was utilized to import machine learning models like RandomForestRegressor and RandomForestClassifier for regression and classification tasks. The sklearn.decomposition module's PCA class was utilized for dimensionality reduction, and the seaborn library was utilized for data visualization. The matplotlib library's pyplot module was utilized for creating visualizations. For feature scaling, the StandardScaler and MinMaxScaler classes from the sklearn.preprocessing module were utilized, and for regression tasks, the LinearRegression class was used. Metrics such as mean squared error and mean absolute error from the sklearn.metrics module were utilized to evaluate the model's performance. Finally, the tensorflow.keras.models module was used to import the Sequential class to build a neural network model.

```
import gdown
import pandas as pd
import numpy as np
from scipy.stats import skew
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.datasets import load_iris
from sklearn.ensemble import RandomForestClassifier
from sklearn.preprocessing import MinMaxScaler
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import TimeSeriesSplit
from sklearn.metrics import r2_score, mean_squared_error, mean_absolute_error,
median_absolute_error
import seaborn as sns
import matplotlib.pyplot as plt

from statsmodels.graphics.tsaplots import plot_pacf

from statsmodels.graphics.gofplots import qqplot
from sklearn.preprocessing import MinMaxScaler
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, LSTM
from statsmodels.tsa.seasonal import seasonal_decompose
from statsmodels.tsa.seasonal import STL
import statsmodels.api as sm
from sklearn.feature_selection import mutual_info_regression
from sklearn.linear_model import Lasso
from statsmodels.tsa.stattools import adfuller
from tensorflow.keras.preprocessing.sequence import TimeseriesGenerator
import tensorflow as tf
from keras.layers import Dropout
from keras.callbacks import EarlyStopping
from pmdarima import auto_arima
```

2.1 Downloading dataset.

To begin the project, the dataset needs to be downloaded from the following link: https://drive.google.com/file/d/1JeXGrDyguQx6JcjVZ39D6r454GYBqFN_/view?usp=sharing. The dataset is compiled from various sources, including (data source: data.world and kaggle). Once the dataset has been downloaded, it should be imported into a pandas dataframe. It is important to check the variables' type and format, as well as the shape of the dataframe, to get an overview of the data.

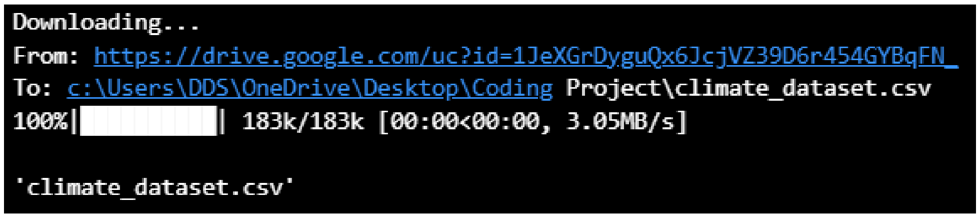
To procure the climate dataset, we leveraged the gdown library in Python, which allows for effortless downloading of files from public Google Drive links. In the initial step, we stipulated the URL of the CSV file in the url variable and designated the anticipated output file name in the output variable.

We subsequently invoked the gdown.download() function, feeding it the url and output file name as its arguments. This action resulted in the downloaded file being saved directly in the current working directory. To maintain visibility of the download progress, the quiet parameter was kept False, thereby enabling the progress to be displayed within the console.

The use of the gdown library in this context offered a simplified and efficient method to download files from a Google Drive link. It set the stage for the subsequent steps of data cleaning and preparation, ensuring the data was readily accessible for further processing and analysis.

```
url = 'https://drive.google.com/uc?id=1JeXGrDyguQx6JcjVZ39D6r454GYBqFN_'
output = 'climate_dataset.csv'
```

```
gdown.download(url, output, quiet=False)
```



```
Downloading...
From: https://drive.google.com/uc?id=1JeXGrDyguQx6JcjVZ39D6r454GYBqFN_
To: c:\Users\DDS\OneDrive\Desktop\Coding Project\climate_dataset.csv
100%|██████████| 183k/183k [00:00<00:00, 3.05MB/s]

'climate_dataset.csv'
```

Figure 1 Downloading Dataset.

2.2 Importing the dataset.

To load the climate dataset into our Python code, we need to read the CSV file and import it into a pandas dataframe. We can use the pandas library to achieve this by calling the read_csv() function and passing the file path of the CSV file as a parameter. In this case, the CSV file is named "climate_dataset.csv" and should be in the current working directory, which is the directory where our Python script is being executed. Once the CSV file is read, the data can be easily manipulated and analyzed using various Pandas functions and methods.

To display the data and get a quick overview of the dataset, we can simply run the name of the pandas dataframe that we created earlier (Table 1). This will show the first few rows of the data. Additionally, to inspect the columns present in the dataset, we can use the 'df.columns' command, which will return a list of column names. This provides a quick and easy way to get an idea of the structure of the dataset and in the following to know more about the data types and formats of the variables in the dataset, we use the df.dtypes. This returns a Series object containing the data types of each column in the dataframe. This information can be useful for data cleaning and manipulation, as well as for selecting appropriate statistical analyses or machine learning algorithms.

```
df = pd.read_csv("climate_dataset.csv")
```

```
df
```

Table 1 Importing the dataset.

	date	emission	anomaly_value	upper_95_ci	lower_95_ci	AverageTemperature	AverageTemperature Uncertainty	City	Country	Latitude	...	Earthquake	Extreme temperature	Extreme weather	Flood	Impact	Landslide	Mass movement (dry)	Volcanic activity	Wildfire	global_avg_temp
0	1/31/1901	4.48E+10	-0.075	-0.161	-0.008	-4.087	0.422	Berlin	Germany	52.24N	...	NaN	NaN	1	1	NaN	NaN	NaN	1	NaN	-0.28
1	2/28/1901	4.48E+10	-0.176	-0.259	-0.11	-3.598	0.951	Berlin	Germany	52.24N	...	NaN	NaN	1	1	NaN	NaN	NaN	1	NaN	-0.06
2	3/31/1901	4.48E+10	-0.272	-0.358	-0.21	2.945	0.354	Berlin	Germany	52.24N	...	NaN	NaN	1	1	NaN	NaN	NaN	1	NaN	0.04
3	4/30/1901	4.48E+10	-0.236	-0.317	-0.164	8.711	0.471	Berlin	Germany	52.24N	...	NaN	NaN	1	1	NaN	NaN	NaN	1	NaN	-0.06
4	5/31/1901	4.48E+10	-0.187	-0.259	-0.122	14.182	0.38	Berlin	Germany	52.24N	...	NaN	NaN	1	1	NaN	NaN	NaN	1	NaN	-0.17
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
1453	9/30/2019	NaN	0.719	0.662	0.761	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1454	10/31/2019	NaN	0.713	0.674	0.754	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1455	11/30/2019	NaN	0.752	0.719	0.796	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1456	12/31/2019	NaN	0.693	0.654	0.732	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1457	1/31/2020	NaN	0.879	0.823	0.921	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

After examining the data and list of variables, we used the df.columns method to print the list of columns. Based on this and the background information of the project, we can describe the variables as follows:

```
df.columns
```

```
Index(['date', 'emission', 'anomaly_value', 'upper_95_ci', 'lower_95_ci',
       'AverageTemperature', 'AverageTemperatureUncertainty', 'City',
       'Country', 'Latitude', 'Longitude', 'All natural disasters', 'Drought',
       'Earthquake', 'Extreme temperature', 'Extreme weather', 'Flood',
       'Impact', 'Landslide', 'Mass movement (dry)', 'Volcanic activity',
       'Wildfire', 'global_avg_temp'],
      dtype='object')
```

As we can see below (Figure 2), the df.dtypes command generated a summary of the data types of each column in the DataFrame. The output of this command reveals that the "date" column has an "object" data type, indicating that it consists of strings. On the other hand, the "emission" column has a "float64" data type. Similarly, other columns like "anomaly_value", "upper_95_ci", "lower_95_ci", and "AverageTemperature" have a "float64" data type. The columns "City", "Country", and "Latitude" are also "object" type columns, while the "Longitude" column has the same data type. The columns "All natural disasters", "Drought", "Earthquake", "Extreme temperature", "Extreme weather", "Flood", "Impact", "Landslide", "Mass movement (dry)", and "Volcanic activity" are all "float64" data type. Lastly, the "global_avg_temp" column has an "object" data type.

```
df.types
```

Table 2 Data Types

date	object
emission	float64
anomaly_value	float64
upper_95_ci	float64
lower_95_ci	float64
AverageTemperature	float64
AverageTemperatureUncertainty	float64
City	object
Country	object
Latitude	object
Longitude	object
All natural disasters	float64
Drought	float64
Earthquake	float64
Extreme temperature	float64
Extreme weather	float64
Flood	float64
Impact	float64
Landslide	float64
Mass movement (dry)	float64
Volcanic activity	float64
Wildfire	float64
global_avg_temp	object
dtype: object	

2.3 Check and fix

Upon scrutinizing the dataset and taking a closer look at the variable data types, it came to my attention that a few rows in the 'global_avg_temp' column were filled with '' entries instead of numeric values. To rectify this, I employed the 'replace' function to switch these ‘entries to 'NaN', symbolizing missing data points. Subsequently, I converted the entire column into a numeric data type with the help of the 'to_numeric' function, thereby facilitating deeper data analysis. This approach ensures that any subsequent computations or data investigations involving the 'global_avg_temp' column remain unaffected by the '***' entries that may have originally denoted missing data in the dataset.

```
df['global_avg_temp'] = df['global_avg_temp'].replace('***', float('nan'))
df['global_avg_temp'] = pd.to_numeric(df['global_avg_temp'])
```

In the dataset, some columns such as 'City', 'Country', 'Latitude' and 'Longitude' all have the same value for each row, which is "Berlin" in Germany. These columns represent the geographical location where the temperature and disaster data were recorded.

Since these columns have the same fixed value for all rows, they do not provide any useful information for modeling purposes. Therefore, dropping these columns using drop() will simplify the data and make it easier to work with. Additionally, removing these columns will not affect the modeling results since they do not contain any useful information for the model.

In summary, dropping these columns will help to streamline the dataset and make it easier to analyze while having no impact on the accuracy of the modeling results.

```
df = df.drop(['City', 'Country', 'Latitude', 'Longitude'], axis=1)
```

To ensure that we can work with the "date" column effectively, we need to convert it from an object data type to a datetime format. This will allow us to perform operations such as filtering the data based on date ranges or extracting specific components of the date (e.g. month, year). Therefore, we will convert the "date" column to datetime format using appropriate pandas functions. In addition, to facilitate further calculations on the dataset, we need to set the 'date' column as the index of the dataframe using the 'df.set_index()' method. However, in certain instances, we may need to reset the index and set 'date' as an ordinary column again.

```
df['date'] = pd.to_datetime(df['date'])
df = df.set_index('date')
```

2.4 Missing Values

Before proceeding with any calculations or modeling, it is important to check the dataset for missing values. To do so, we can scroll through the dataset visually and also check for missing values using the code: missing_values = df.isnull().sum(). After running this code, we can see that there are a significant number of missing values in almost all the variables. For instance, we observe that there are more than 99% missing values in "Impact" column. Therefore, it might be a good decision to drop this column using the drop() function.

```
missing_values = df.isnull().sum()
print(missing_values)
```

Table 3 Count of Missing Values 1

emission	53
anomaly_value	29
upper_95_ci	29
lower_95_ci	29
AverageTemperature	106
AverageTemperatureUncertainty	106
All natural disasters	41
Drought	593
Earthquake	89
Extreme temperature	785
Extreme weather	89
Flood	329
Impact	1446
Landslide	509
Mass movement (dry)	1073
Volcanic activity	533
Wildfire	725
global_avg_temp	71
dtype: int64	

```
df = df.drop(['Impact'], axis=1)
```

2.4.1 Data Resampling

Given the temporal characteristics of our dataset, it is evident that monthly values from 1900 to 2020 are readily available. However, we encounter a situation where some Februaries present two separate values, one each for the 28th and the 29th of the month. To streamline this inconsistency, we apply the resample() function from the Pandas library, which aggregates the data by generating the mean value for each month. This functionality proves invaluable for time-series analysis, especially when dealing with high-frequency data such as daily or hourly readings, and the analytical needs demand a lower frequency like monthly or yearly.

Resampling the data facilitates more straightforward analysis and visualization, effectively reducing the data to a single value for each month. Particularly for each February, any missing values are replaced by the mean of the available data for that month. Post-resampling, employing the mean() function, we observe that the anomaly_values, upper_95_ci, and lower_95_ci present no missing values. To validate the successful implementation of resampling, we inspect the shape of the dataframe using df.shape(). This inspection reveals a decrease in the number of rows by 29, affirming the proper execution of the resampling process. Thus, we successfully reduce and refine our dataset to better suit our analytical needs (Pandas 2023).

```
df = df.resample('M').mean()
```

```
df.shape
```

```
(1429, 17)
```

To impute the missing values in the disaster-related columns which are 'All natural disasters', 'Drought', 'Earthquake', 'Extreme temperature', 'Extreme weather', 'Flood', 'Landslide', 'Mass movement (dry)', 'Volcanic activity' and 'Wildfire', I made the assumption that missing values meant that there were no disasters reported during that time period. Therefore, I decided to fill the missing values with zeros using the fillna() method.

```
df[['All natural disasters', 'Drought', 'Earthquake', 'Extreme temperature', 'Extreme weather', 'Flood',
'Landslide', 'Mass movement (dry)', 'Volcanic activity', 'Wildfire']] = df[['All natural disasters', 'Drought',
'Earthquake', 'Extreme temperature', 'Extreme weather', 'Flood', 'Landslide', 'Mass movement (dry)',
'Volcanic activity', 'Wildfire']].fillna(0)
```

```
missing_values = df.isnull().sum()
print(missing_values)
```

Table 4 Count of Missing Values 2

emission	24
anomaly_value	0
upper_95_ci	0
lower_95_ci	0
AverageTemperature	77
AverageTemperatureUncertainty	77
All natural disasters	0
Drought	0
Earthquake	0
Extreme temperature	0
Extreme weather	0
Flood	0

Landslide	0
Mass movement (dry)	0
Volcanic activity	0
Wildfire	0
global_avg_temp	42
dtype: int64	

2.4.2 Forward filling

Upon close inspection of the 'emission' column and a thorough scroll-through of the dataset, it becomes evident that a substantial fraction of the missing values in this column are concentrated towards the tail end of the time series. This observation, coupled with the consistent upward trend within this column, led me to adopt the forward fill (ffill) method to address these missing values. Implementing this technique, any data gaps in the 'emission' column are bridged with the last valid preceding observation.

The ffill method is particularly advantageous in the realm of time series data, where missing values can often stem from measurement or reporting inaccuracies. In such cases, the prior observation can serve as a plausible substitute for the missing data point. However, it's essential to bear in mind that utilizing the last observation to fill missing values may not always be the optimal approach, contingent on the context and the nature of the missing data. Therefore, it is prudent to consider alternate methods of handling missing data, tailored to the specific demands of the situation at hand (Stakoverflow 2015).

```
df['emission'] = df['emission'].ffill()
```

2.4.3 Mean of the corresponding months

In addressing the missing values within the 'AverageTemperature', 'AverageTemperatureUncertainty', and 'global_avg_temp' columns of the temperature dataset, I employed an approach that respects the temporal and seasonal character of the data. I started this process by grouping the data by month using the groupby() function and calculating the mean values for the 'AverageTemperature' and 'AverageTemperatureUncertainty' columns using the mean() function. Following this, I filled the missing values in each column with the mean value corresponding to that month, deploying the fillna() method along with a lambda function.

To facilitate this operation, I introduced a new 'month' column into the data frame. This was achieved by extracting the month element from the 'date' column using the 'dt' attribute in Pandas. Once the missing values were duly filled, I removed the 'month' column from the data frame.

This rigorous process of data cleaning and preprocessing ensures that the temperature data is ready for analysis, significantly reducing the risk of potential inaccuracies or biases that could stem from unresolved missing values.(Pandas 2023)

```
df = df.reset_index()
```

```
df['month'] = df['date'].dt.month
```

```
mean_values = df.groupby('month').mean()

# create new dataframe with mean values, and set index to month
mean_df = pd.DataFrame({'AverageTemperature': mean_values['AverageTemperature'],
'AverageTemperatureUncertainty': mean_values['AverageTemperatureUncertainty']})
mean_df.index.name = 'month'
# fill missing values with corresponding mean values
df['AverageTemperature'] = df.groupby('month')['AverageTemperature'].transform(lambda x:
x.fillna(x.mean()))
df['AverageTemperatureUncertainty'] =
df.groupby('month')['AverageTemperatureUncertainty'].transform(lambda x: x.fillna(x.mean()))
df['global_avg_temp'] = df.groupby('month')['global_avg_temp'].transform(lambda x: x.fillna(x.mean()))

# drop the "month" column
df.drop('month', axis=1, inplace=True)
```

After applying the data cleaning methods as explained previously, we can check the number of missing values for all columns in our dataset and observe that there are no missing values remaining. This ensures that our dataset is now complete and suitable for further analysis.

```
missing_values = df.isnull().sum()
print(missing_values)
```

Table 5 Count of Missing Values 3

date	0
emission	0
anomaly_value	0
upper_95_ci	0
lower_95_ci	0
AverageTemperature	0
AverageTemperatureUncertainty	0
All natural disasters	0
Drought	0
Earthquake	0
Extreme temperature	0
Extreme weather	0
Flood	0
Landslide	0
Mass movement (dry)	0
Volcanic activity	0
Wildfire	0
global_avg_temp	0
dtype: int64	

2.5 Anomaly Detection and Solutions for Outlier Handling

To ensure that our data is suitable for further analysis and modeling, it is crucial to identify and address any outliers or anomalies present in the data. There are various techniques that can be employed for outlier detection and handling, including:

- Visual inspection: This method involves plotting the data and visually examining it for any values that stand out from the rest. Once identified, anomalies and outliers can be examined more closely to determine whether they should be retained or removed from the dataset.
- Statistical methods: Statistical techniques can be used to automatically detect outliers and anomalies. For instance, calculating the z-score or Mahalanobis distance for each data point can identify points exceeding a certain threshold as outliers. Alternatively, regression models or time series decomposition techniques can be utilized to identify anomalies and outliers.
- Machine learning: Machine learning algorithms such as clustering or anomaly detection algorithms like Isolation Forest or Local Outlier Factor can be utilized to identify data points that deviate significantly from the norm.
- Data imputation: In cases where anomalies and outliers are caused by missing or incorrect data, data imputation techniques can be employed to estimate the missing values. Simple methods like mean or median imputation can be used for filling in missing values, whereas more advanced techniques like k-nearest neighbor or regression imputation can be employed to estimate missing values based on other data points.

The choice of approach ultimately depends on the nature of the data and the objectives of the analysis. It is crucial to meticulously evaluate each method to ensure that the data is appropriately handled, and the analysis results are valid. I have chosen to employ a combination of visualization and statistical methods to achieve this goal. I will elaborate on each of these methods in the following sections.

2.5.1 Visual inspection

In the initial step, my approach was to assess the data using visual methods to detect any outliers. Therefore, I plotted a histogram to examine the distribution of each feature, and then I subsequently employed boxplots to detect any potential outliers in a visual manner. To examine the distribution, I specifically looked at variables such as anomaly value, emissions, and global extreme temperature. Additionally, as "All natural disaster" are the sum of all types of disasters, I focused solely on the distribution of this variable.

Understanding the distribution of values within a dataset is a vital aspect of outlier detection and handling. While the calculation of the z-score and the identification of values outside a set range or threshold is a common technique for this task, its applicability is grounded in the assumption of a normal distribution of data, given that z-score calculation is a mean-based procedure. Therefore, before opting for this method, I ensure that the normality of the dataset is confirmed. If this assumption doesn't hold, other suitable methods are considered for outlier detection.

To evaluate the distribution of the variables, I utilized various visualization tools, including histograms and QQ plots. These tools were applied to visualize the distribution of several variables in the dataset.

The histogram reveals a bimodal distribution for anomaly values, suggesting the possible existence of two distinct populations within the data. Specifically, there are two clear peaks; one, situated between 190-200 on the y-axis, and the other residing within the 70-75 y-axis range.

Importantly, the histogram doesn't showcase a normal distribution as it lacks a symmetrical bell-shaped curve centered around the mean. Instead, it presents two distinct peaks, underlining the asymmetrical distribution of data.

Additionally, the histogram exhibits a fluctuating pattern. It initially shows an increase in values in the first half of the x-axis, followed by a decrease and then another increase. A similar pattern is observed for the y-axis values, starting with a low range, escalating to the first peak, then receding before reaching the second peak.

```
# Plot a histogram of the 'anomaly_value' column with 20 bins
n_bins = 20
hist_values, bins, _ = plt.hist(df['anomaly_value'], bins=n_bins)

# Add axis labels and a title to the plot
plt.xlabel('Anomaly Value')
plt.ylabel('Frequency')
plt.title('Distribution of Anomaly Values')
```

```
# Add vertical lines to show the range of each bin
for i in range(len(bins)):
    plt.axvline(bins[i], color='k', linestyle='dashed', linewidth=1)

# Display the plot
plt.show()
```

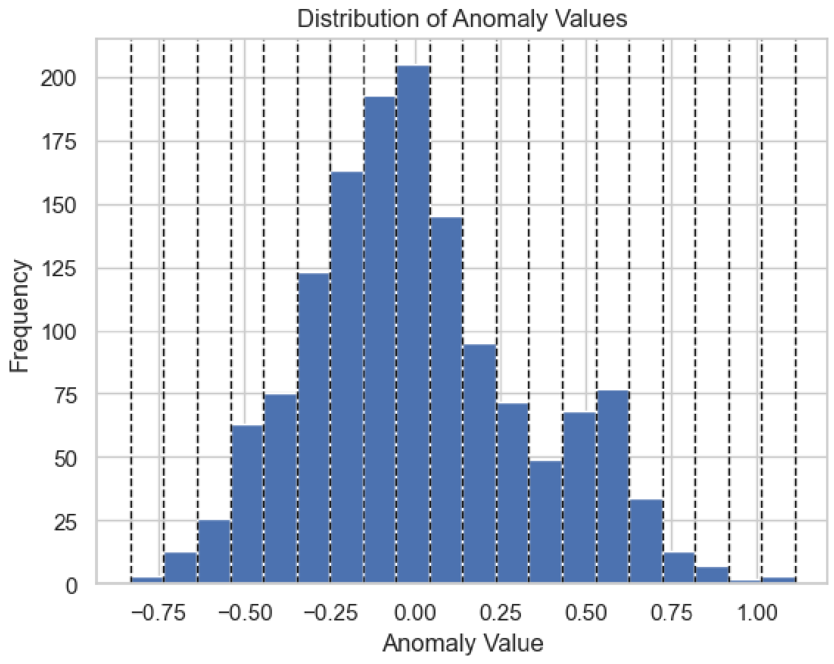


Figure 2 Histogram of Anomaly Value

A Quantile-Quantile (QQ) plot serves as a potent visualization tool for assessing a dataset's distribution, particularly its conformity to a normal distribution. The QQ plot shown below compares the distribution of our dataset with a theoretical normal distribution. The diagonal line within the QQ plot signifies a perfect alignment between our data and a normal distribution.

The proximity of points to the diagonal line in the QQ plot serves as a measure of normality. If the points cluster close to the diagonal, it suggests a normal distribution within the dataset. Conversely, a substantial deviation of points from the diagonal indicates a non-normal distribution.

In our case, the QQ plot illustrates a considerable distance between the initial points and the diagonal line, suggesting a non-normal distribution. However, the points seem to converge towards the diagonal line as we move along, indicating a progression towards normality. Towards the end, the points drift away from the diagonal once more, suggesting a deviation from a normal distribution (Zach 2023).

```
# Fit a normal distribution to the data
dist = qqplot(df['anomaly_value'], line='s')
```

```
# Add axis labels and a title to the plot
plt.xlabel('Theoretical Quantiles')
plt.ylabel('Sample Quantiles')
plt.title('QQ Plot of Anomaly Values')

# Display the plot
plt.show()
```

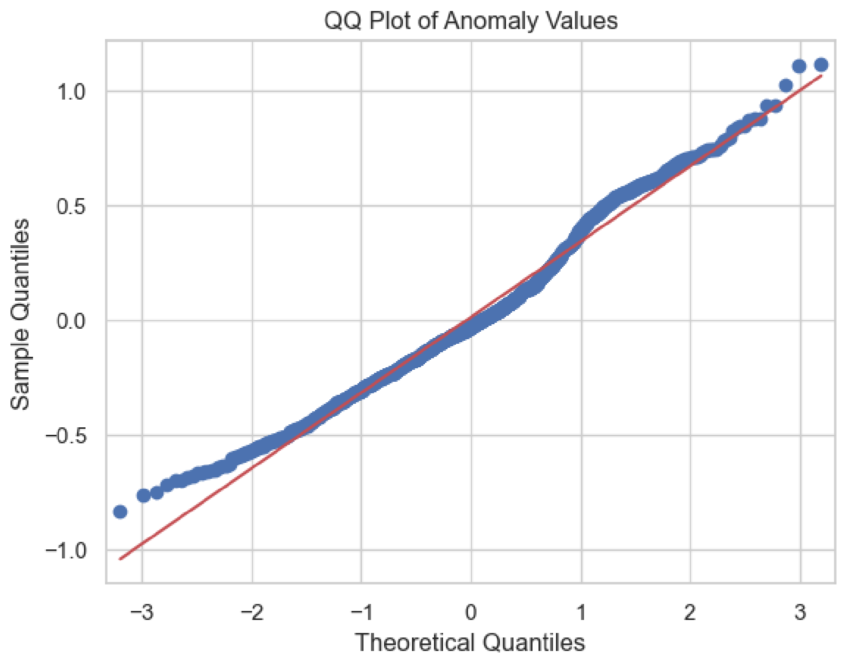


Figure 3 Quantile-Quantile Plot of Anomaly Value

In the QQ plot for "emission" (Figure 5), a noticeable deviation from the diagonal line is observed throughout, indicating a clear departure from normality. The data points intersect the middle line at three distinct areas, further highlighting the non-normal distribution. Complementing this with the histogram (Figure 4), we notice a stark concentration of data points towards the left half of the graph, particularly within the range of 0.0 to 0.8. As we progress towards the right half, the density of data points diminishes, forming a right-skewed distribution. Thus, it's clear from both graphical analyses that our data does not adhere to a normal distribution.

```
# Plot a histogram of the 'anomaly_value' column with 20 bins
n_bins = 20
hist_values, bins, _ = plt.hist(df['emission'], bins=n_bins)

# Add axis labels and a title to the plot
plt.xlabel('emission')
plt.ylabel('Frequency')
plt.title('Distribution of emission')
```

```
# Add vertical lines to show the range of each bin
for i in range(len(bins)):
    plt.axvline(bins[i], color='k', linestyle='dashed', linewidth=1)

# Display the plot
plt.show()
```

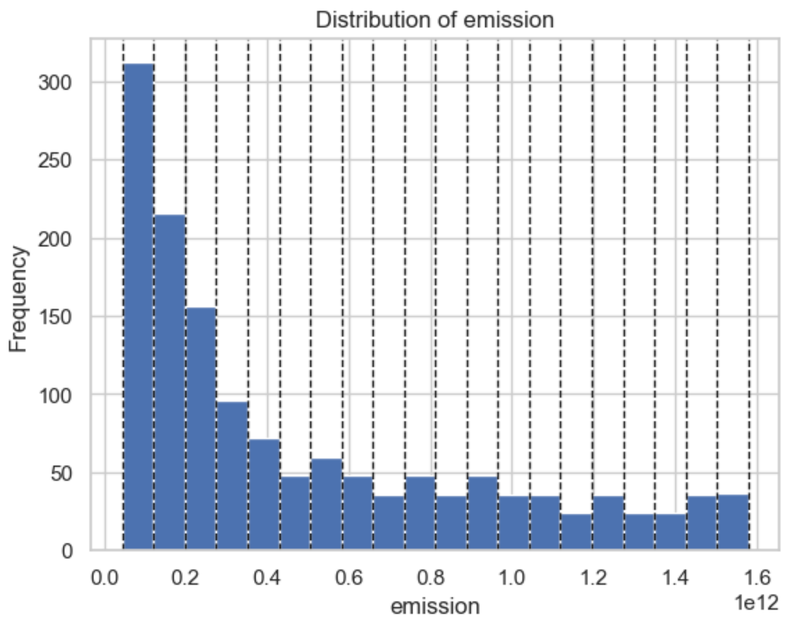


Figure 4 Histogram of emission

```
# Fit a normal distribution to the data
dist = qqplot(df['emission'], line='s')
# Add axis labels and a title to the plot
plt.xlabel('Theoretical Quantiles')
plt.ylabel('Sample Quantiles')
plt.title('QQ Plot of Emission')
# Display the plot
plt.show()
```

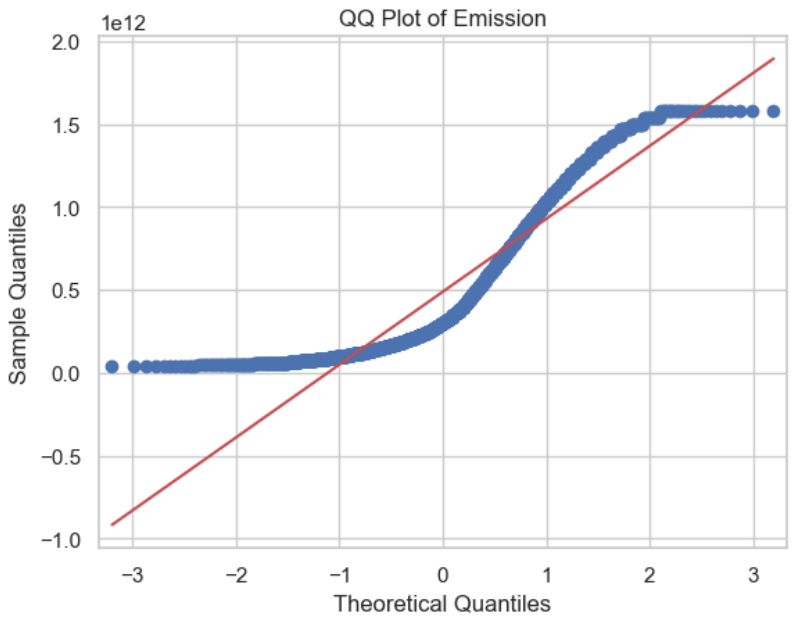


Figure 5 Quantile-Quantile of Emission

In the subsequent histogram (Figure 6), it's evident that our data scarcely aligns with the characteristics of a normal distribution. To substantiate this observation further, we turn to the QQ plot for global average temperature (Figure 7), providing additional insight into our data's distributional tendencies. The QQ plot presented in Figure 7 reveals a distinct pattern: initially, we observe data points deviating from the central line, before they appear to align with it. However, towards the end, the points veer away upwards, creating another deviation. Interpreting these observations, we infer that the data is not likely normally distributed. Moreover, this distribution bears significant similarity with that of the 'anomaly value', further reinforcing our inference.

```
# Plot a histogram of the 'anomaly_value' column with 20 bins
n_bins = 20
hist_values, bins, _ = plt.hist(df['global_avg_temp'], bins=n_bins)

# Add axis labels and a title to the plot
plt.xlabel('Global Average Temperature')
plt.ylabel('Frequency')
plt.title('Distribution of Global Average Temperature')

# Add vertical lines to show the range of each bin
for i in range(len(bins)):
    plt.axvline(bins[i], color='k', linestyle='dashed', linewidth=1)

# Display the plot
plt.show()
```

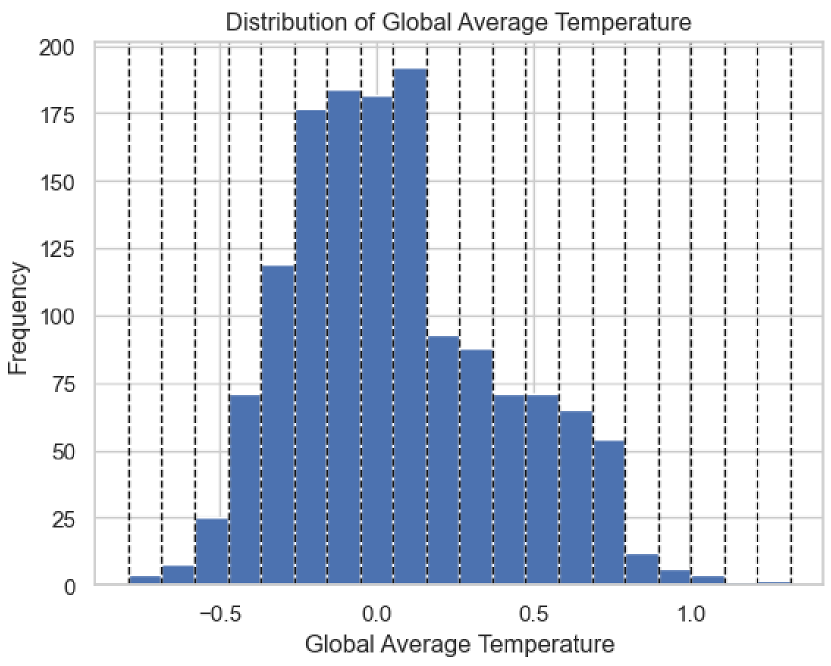


Figure 6Histogram of Global Average Temperature

```
# Fit a normal distribution to the data
dist = qqplot(df['global_avg_temp'], line='s')

# Add axis labels and a title to the plot
plt.xlabel('Theoretical Quantiles')
plt.ylabel('Sample Quantiles')
plt.title('QQ Plot of Global Average Temperature')

# Display the plot
plt.show()
```

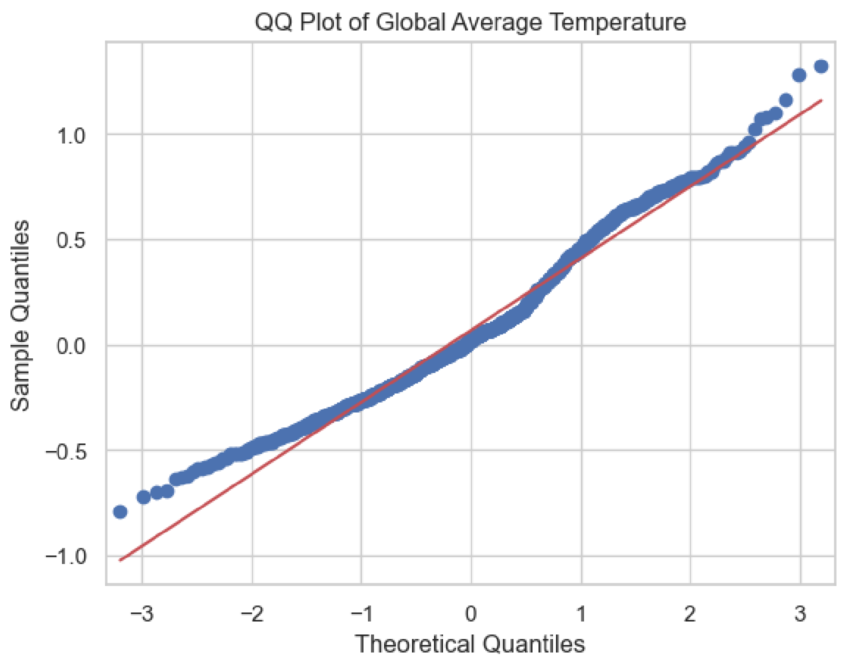


Figure 7 Quantile-Quantile of Global Average Temperature

The histogram (Figure 8) and QQ plot (Figure 9) of the 'All natural disasters' variable distinctly show a non-normal distribution, largely characterized by a right-skewed and multimodal pattern.

Examining the graphical representations of other variables within the dataset, it remains uncertain whether their distributions conform to normality. Considering that our dataset comprises a time series, the application of the z-score method for outlier detection may not be optimal. The z-score method demonstrates its utmost effectiveness when dealing with data that is normally distributed. Hence, it might be more prudent to contemplate alternative methods for detecting outliers in this scenario.

To ensure the robustness and accuracy of my analysis, I utilized the same approach for outlier detection on other variables within the dataset, intending to gain a deeper understanding of their distributions.

As a concluding remark, considering the unverified normal distribution of several variables, I've decided to forego the use of the Z-Score method for identifying outliers. Importantly, given the potential seasonal trends exhibited by our time series data, as suggested by the distributions of 'anomaly value' and 'global extreme temperature', I've opted for the Seasonal-Trend Decomposition based on LOESS, or STL, to effectively carry out this analysis.

```
# Plot a histogram of the 'anomaly_value' column with 20 bins
n_bins = 20
hist_values, bins, _ = plt.hist(df['All natural disasters'], bins=n_bins)

# Add axis labels and a title to the plot
plt.xlabel('All natural disasters')
plt.ylabel('Frequency')
plt.title('Distribution of All natural disasters')

# Add vertical lines to show the range of each bin
for i in range(len(bins)):
    plt.axvline(bins[i], color='k', linestyle='dashed', linewidth=1)

# Display the plot
plt.show()
```

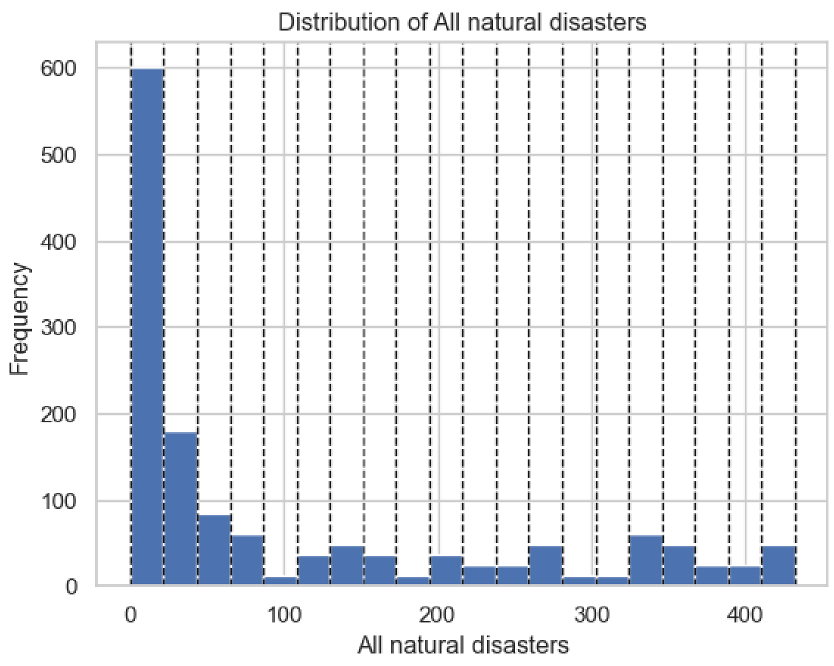


Figure 8 Histogram of All Natural Disasters

```
# Fit a normal distribution to the data
dist = qqplot(df['All natural disasters'], line='s')

# Add axis labels and a title to the plot
plt.xlabel('Theoretical Quantiles')
plt.ylabel('Sample Quantiles')
plt.title('QQ Plot of All natural disasters')

# Display the plot
plt.show()
```

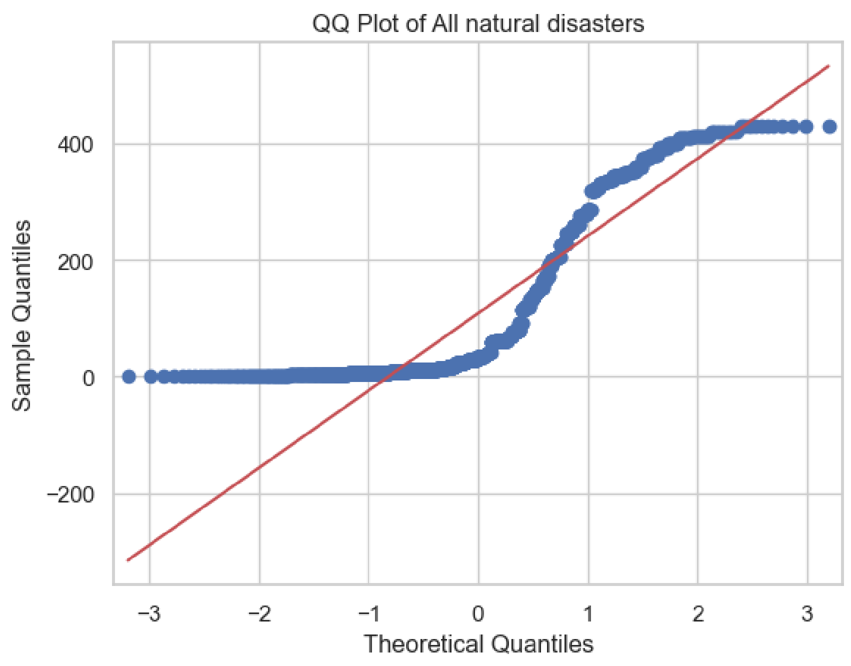


Figure 9 Quantile-Quantile of All Natural Disasters

A boxplot serves as a comprehensive statistical summary of a dataset's distribution, offering visual cues to possible outliers by demonstrating the value range and any outliers beyond this spectrum. The central box within the plot embodies the interquartile range (IQR), accounting for the middle 50% of the data. The whiskers, extending from the box, reach out to the highest and lowest data points that are not considered outliers. Any data points beyond the whiskers are depicted as individual markers, referred to as "fliers". Consequently, any data points outside of the whiskers are potential outliers.

Upon inspecting the boxplots provided, it becomes clear that most features exhibit no significant outliers, with the exceptions of "anomaly values" and "Global Average temperature". To explore outliers in these features, I will apply the seasonal-trend decomposition procedure based on Loess (STL) in the forthcoming section.

```
sns.boxplot(x=df['anomaly_value'])
```

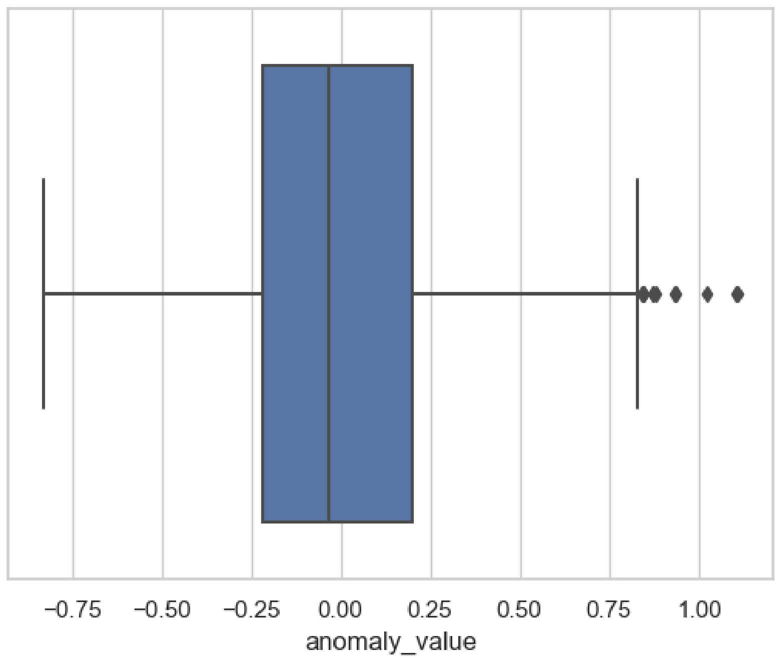


Figure 10 Box Plot of Anomaly Value

```
sns.boxplot(x=df['emission'])
```

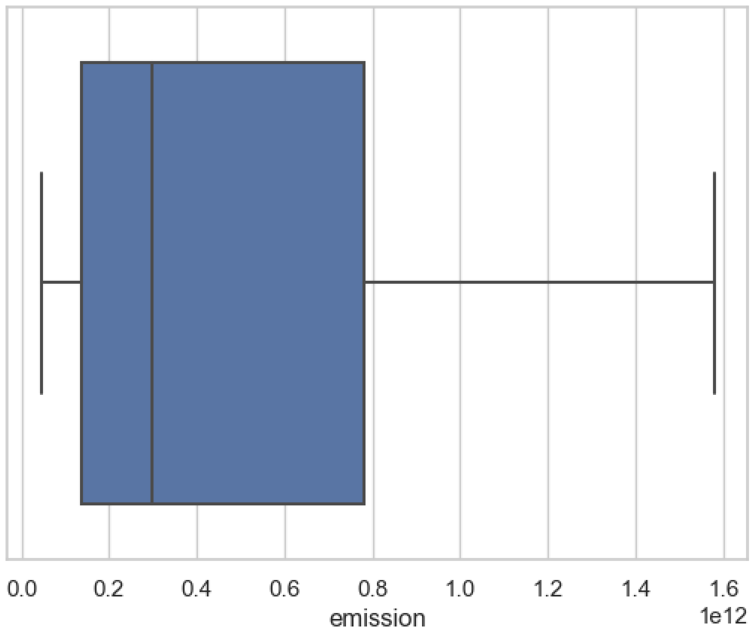


Figure 11 Box Plot Emission

```
sns.boxplot(x=df['global_avg_temp'])
```

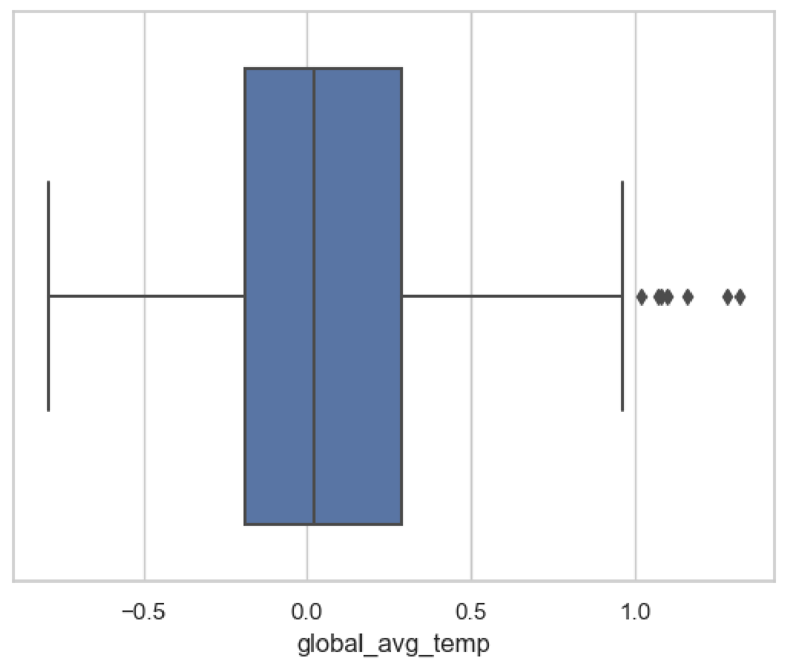


Figure 12 Box Plot Average Temperature

```
sns.boxplot(x=df['All natural disasters'])
```

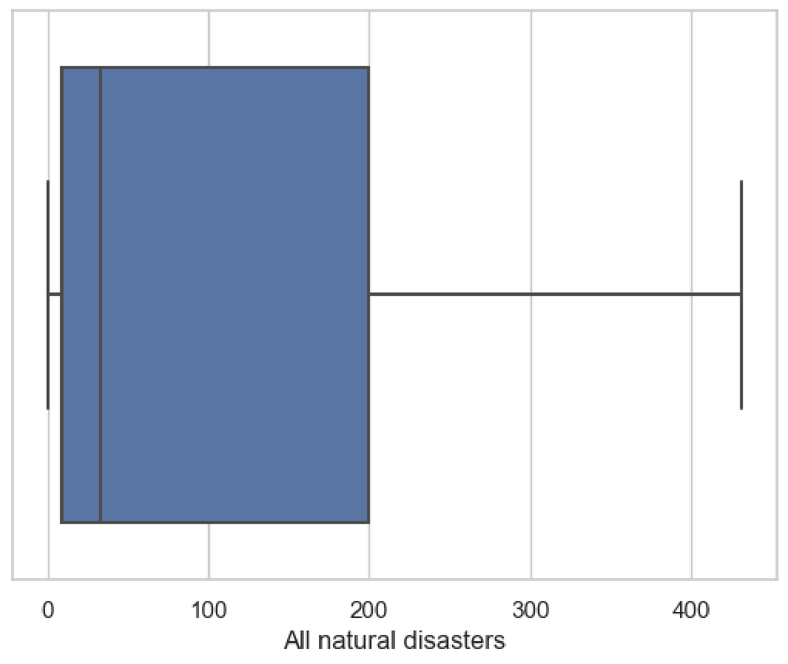


Figure 13 Box Plot All Natural Disasters

2.5.2 Statistical Method

The method of Seasonal-Trend Decomposition based on LOESS (STL) is a resilient and adaptable approach for breaking down time series data. This process partitions a time series into three distinct elements: trend, seasonality, and residuals. The trend reflects the basic pattern present in the data, seasonality encapsulates recurring cycles or patterns, while residuals represent the remaining variance that has not been accounted for in the data.

An STL graph or chart typically comprises four sections:

- Original data: This section presents the untouched time series data.
- Trend element: This section displays the data's trend after eliminating the seasonal element and residual fluctuations.
- Seasonal element: This section portrays the cyclical or repeating patterns in the data.
- Residual element: This section showcases the remaining fluctuations or 'noise' left in the series after the trend and seasonal elements have been subtracted.

One of the key strengths of STL decomposition is its capability to accommodate varying types of seasonality, ranging from monthly and quarterly to daily and hourly. Moreover, it does not necessitate the seasonal and trend elements to be orthogonal, allowing for potential interaction during the decomposition process. This adaptability renders it particularly beneficial when examining intricate time series data.

To determine the interquartile range (IQR) of the residual component, which is the difference between the 75th and 25th percentiles (Q3 and Q1, respectively), we will use the STL method. Then, based on the IQR, we will calculate the upper and lower bounds for potential outliers using the standard formula of 1.5 times the IQR added to and subtracted from Q3 and Q1, respectively.

The following code creates a seasonal-trend decomposition model based on Loess (STL) for the "anomaly_value" column in the "df" dataframe with a seasonal period of 12 (months). It then fits the data to the model and plots the resulting decomposition. This allows us to visualize the seasonal and trend components of the data and identify any patterns or trends. And calculates the interquartile range (IQR) of the residual component of the data, which is the difference between the 75th and 25th percentiles (Q3 and Q1, respectively). It then calculates the upper and lower bounds for outliers based on the IQR using the standard formula of 1.5 times the IQR added to and subtracted from Q3 and Q1, respectively. Also creates a new dataframe called "outliers_anomaly_value" that includes only the rows of the original "df" dataframe where the residual value falls outside the calculated upper and lower bounds. These are potential outliers in the data. The loc function is used to select the rows based on a boolean condition, which is the logical OR of two conditions: one where the residual value is less than the lower bound, and one where it is greater than the upper bound.

To identify outliers in an STL plot, we need to analyze the residuals plot. This plot displays the difference between the observed values and the values predicted by the seasonal and trend components. Outliers will be indicated by data points that deviate significantly from the main cluster of points. In the residuals plot shown below, there are some out-of-bound points. To locate the outliers, I used the following code: outliers_anomaly_value = df.loc[(residual < lower_bound) | (residual > upper_bound)]. I then calculated the number of outliers using the following code: num_outliers_anomaly_value = len(outliers_anomaly_value). The number of outliers was printed using the following code: print("Number of outliers:", num_outliers_anomaly_value) which are 8, I used the same method to handle the outliers for other features.

To handle the outliers in our monthly-based time series data, we can use a rolling mean approach. This involves calculating the rolling mean of the "anomaly_value" column in the "df" dataframe using a window size of 3 and a centered window. Any outlier values that were previously identified and stored in a dataframe for example: "outliers_anomaly_value" can then be replaced with the corresponding rolling mean values.

The rolling mean is a moving average of the data points over a specified window size. It helps to smooth out any fluctuations or noise in the data and gives a better understanding of the overall trend. By using a rolling mean, we can replace the outlier values with a more reasonable value.

To achieve this, we can use a for loop to go through each index in the "outliers_anomaly_value" dataframe and replace the corresponding "anomaly_value" in the "df" dataframe with the value of the rolling mean at that index.

In the STL graph for 'anomaly value', the trend component unveils an ascending progression interspersed with infrequent, non-extensive irregularities. This signals an overall increase in anomaly values over time, albeit punctuated by sporadic inconsistencies. Further examination of these deviations could reveal their origins and implications for our analysis.

The seasonal component, on the other hand, reflects a consistent, recurring cycle of anomaly values. This recurrent pattern signifies a periodical nature within anomaly values, with specific trends repeating at regular intervals. Further, a gradual upward trend in this seasonality indicates an escalating intensity or amplitude of these seasonal fluctuations as time progresses.

The residuals component of the STL graph underscores minimal anomalies when compared to the overall dataset volume. This suggests that most of the variability in the 'anomaly value' can be explained by the trend and seasonal components, with only a small portion remaining unaccounted for. Thus, the data appears to adhere quite closely to the identified seasonal and trend patterns, with only rare and relatively minor deviations.

Together, these observations highlight the multifaceted temporal dynamics embedded within the 'anomaly value', and accentuate the importance of comprehensively addressing trend, seasonality, and residuals in our analysis (Stackoverflow 2013).

```
# Convert the 'date' column to datetime if it's not already
df['date'] = pd.to_datetime(df['date'])

# Set 'date' as the index
df.set_index('date', inplace=True)
stl = STL(df['anomaly_value'], period=12)
res = stl.fit()
res.plot()

# outliers in the residual component of the data
residual = res.resid
Q1 = residual.quantile(0.25)
Q3 = residual.quantile(0.75)
IQR = Q3 - Q1
lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR
outliers_anomaly_value = residual[(residual < lower_bound) | (residual > upper_bound)]
```

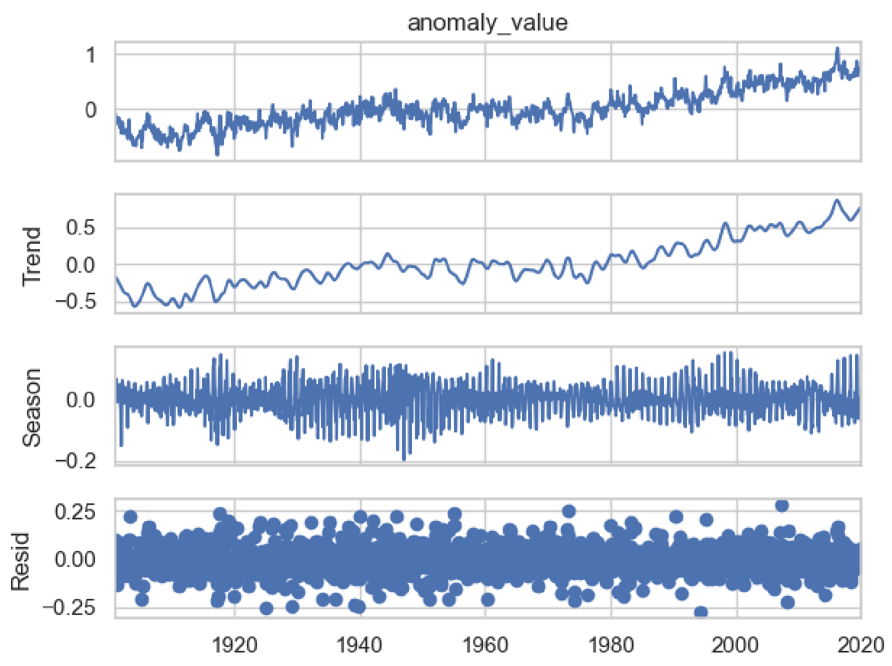


Figure 14 STL Decomposition of Anomaly Value

Upon deploying the method, I discovered 54 outliers within the 'anomaly value'. To handle these outliers, I leveraged the 'rolling mean' technique. This method is ideal for smoothing time-series data and mitigating the impact of outliers without completely removing them, as it calculates the average of a specific window of data that slides over the time series. By replacing the identified outliers with the rolling mean, I preserved the data integrity while ensuring robust analysis and modelling outcomes.

The graph titled "Anomaly Value: Before and After Smoothing" effectively depicts the changes in our data post the application of the rolling mean technique to handle outliers. Post smoothing, it is apparent that the time-series data exhibits less volatility and is now less prone to anomalous spikes. The curve has become more regular and predictable, which could potentially lead to more accurate forecasting. This demonstrates the effectiveness of the outlier handling process in stabilizing the data and enhancing its suitability for further analysis and modeling.

```
num_outliers_anomaly_value = len(outliers_anomaly_value)
print("Number of outliers:", num_outliers_anomaly_value)
```

```
Number of outliers: 54
```

```
df['rolling_mean'] = df['anomaly_value'].rolling(window=3, center=True).mean()

for i in outliers_anomaly_value.index:
    df.loc[i, 'anomaly_value'] = df.loc[i, 'rolling_mean']
```

```
# Before smoothing
plt.figure(figsize=(12,6))
plt.plot(df.index, df['anomaly_value'], label='Original')

# After smoothing
plt.plot(df.index, df['rolling_mean'], color='red', label='Smoothed')

plt.legend()
plt.title('Anomaly Value: Before and After Smoothing')
plt.show()
```

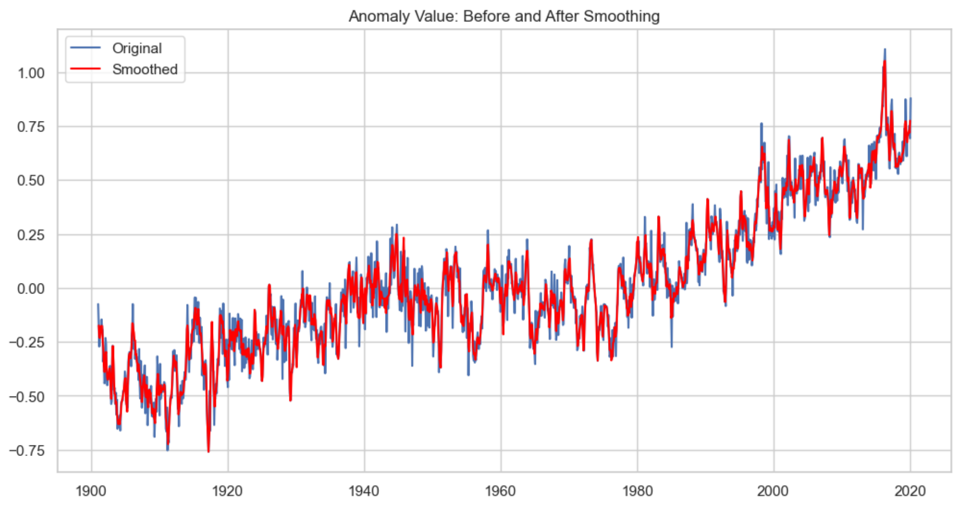


Figure 15 Aomaly Value: Before and After Smoothing

```
df = df.drop('rolling_mean', axis=1)
```

Referring to Figure 16, the 'emission' variable clearly exhibits a consistent upward trajectory over time, without any discernable seasonal oscillations. An examination using the STL method reveals that about 16.6% of the total emission data points, roughly 238 values, potentially qualify as outliers. However, given the nature of the data distribution and its unidirectional increase, it's challenging to definitively categorize these data points as outliers.

Considering the time series data and the consistent uptick in emission values, simply discarding these outliers may not be the best strategy. Similarly, imputation using the mean method may not be suitable, as it could dilute the underlying trend in the data.

These points may not truly be outliers but rather integral parts of the emissions trend. In the subsequent feature selection and model design phase, a strategic decision will be made regarding the handling of this variable, thereby ensuring we maintain the integrity and reliability of our analysis.

```
stl = STL(df['emission'], period=12)
res = stl.fit()
res.plot()

# outliers in the residual component of the data
residual = res.resid
Q1 = residual.quantile(0.25)
Q3 = residual.quantile(0.75)
IQR = Q3 - Q1
lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR
outliers_emission = residual[(residual < lower_bound) | (residual > upper_bound)]
```

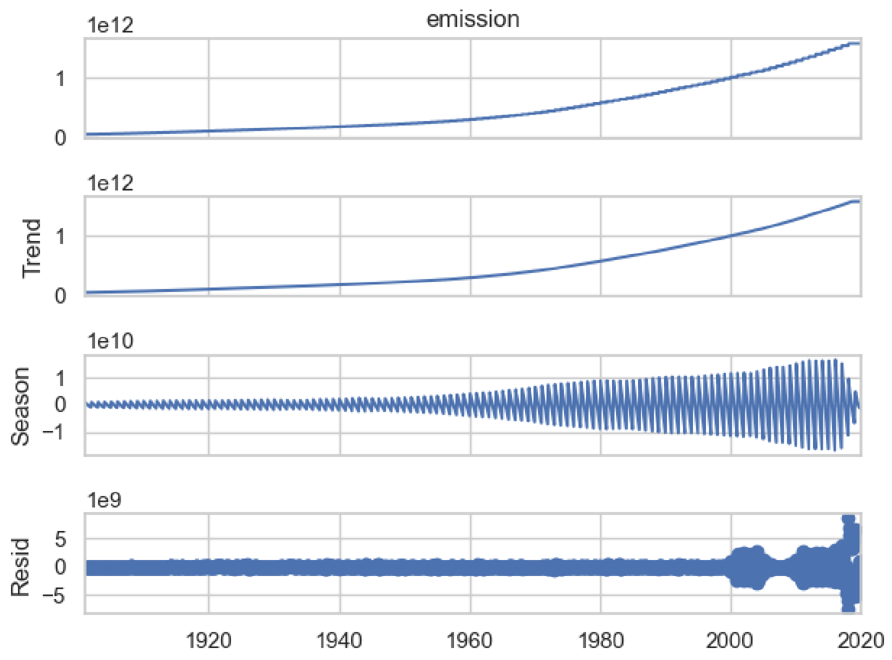


Figure 16 STL Decomposition of Emission

```
num_outliers_emission = len(outliers_emission)
print("Number of outliers:", num_outliers_emission)
```

```
Number of outliers: 238
```

As illustrated in Figure 17, the STL graph corresponding to the 'natural disasters' variable exhibits an upward trend over time, albeit punctuated by several peaks and a significant decrease towards the end. The graph also indicates no discernible seasonal oscillations, while the residuals display considerable dispersion towards the end.

Upon quantifying the outliers, we find that they constitute more than 17.2% of the total values for this variable. Given the variable's inconsistent increasing trend, its significant drop-off towards the end, and the characteristic unpredictability of natural disasters, it may not be entirely logical to discard or replace the outliers.

Additionally, the high number of outliers, coupled with the intrinsic nature of this data, could potentially impact the prediction accuracy of our model. Thus, in the upcoming feature selection phase, we may contemplate removing this variable altogether from our analysis. This approach might help preserve the validity of our model while simultaneously minimizing any potential bias or distortion caused by the high proportion of outliers.

```
# seasonal decomposition
stl = STL(df['All natural disasters'], period=12)
res = stl.fit()
res.plot()

# outliers in the residual component of the data
residual = res.resid
Q1 = residual.quantile(0.25)
Q3 = residual.quantile(0.75)
IQR = Q3 - Q1
lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR
outliers_All_natural_disasters = df.loc[(residual < lower_bound) | (residual > upper_bound)]
```

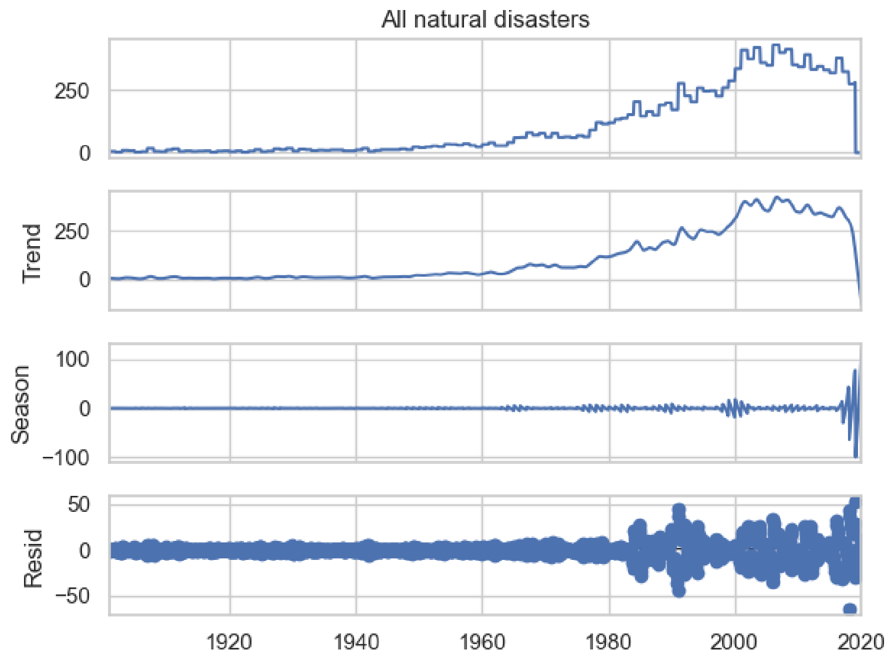


Figure 17 STL Decomposition of All Natural Disasters

```
num_outliers_AllNaturalDisasters = len(outliers_All_natural_disasters)
print("Number of outliers:", num_outliers_AllNaturalDisasters)
```

```
Number of outliers: 246
```

Figure 18 presents the STL chart for global temperature extremes, which exhibits a striking similarity to the anomaly value variable. However, towards the end of the time series, a precipitous drop in the trend line is evident, and a surge in the density of residuals is observed.

The tally of outliers is at 37, constituting less than 2.6% of the total values. Given the intrinsic correlation of this variable with the anomaly value, it would be prudent to withhold judgement about its handling until the subsequent section. This cautionary approach will allow us to consider all relevant aspects before making any significant decisions regarding this variable.

```
stl = STL(df['global_avg_temp'], period=12)
res = stl.fit()
res.plot()

# outliers in the residual component of the data
residual = res.resid
Q1 = residual.quantile(0.25)
Q3 = residual.quantile(0.75)
IQR = Q3 - Q1
lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR
outliers_global_avg_temp = residual[(residual < lower_bound) | (residual > upper_bound)]
```

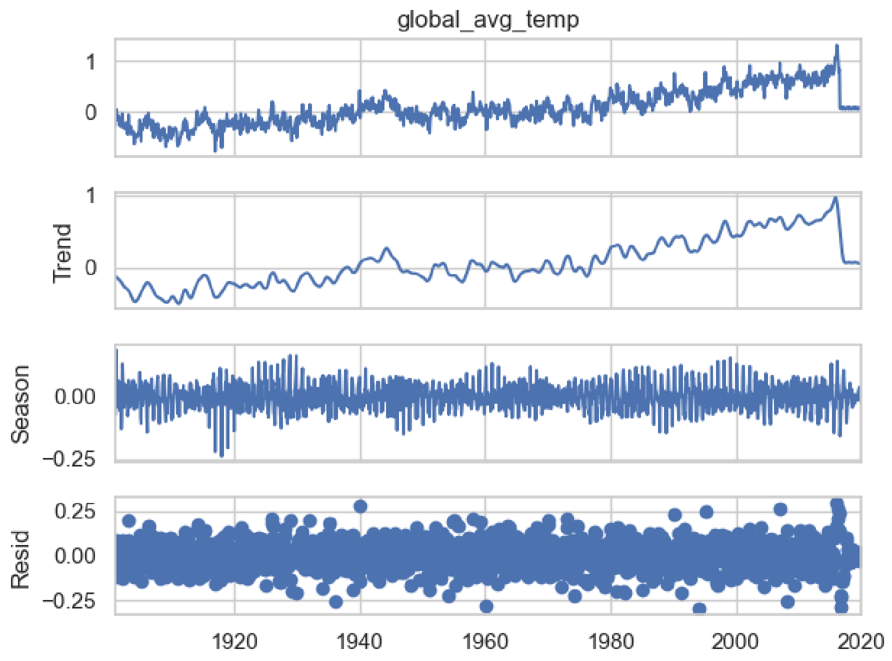


Figure 18 STL Decomposition of Global Average Temperature

```
num_outliers_global_avg_temp = len(outliers_global_avg_temp)
print("Number of outliers:", num_outliers_global_avg_temp)
```

```
Number of outliers: 37
```

3. Feature Selection

Feature engineering in time series forecasting involves creating and identifying features that can capture the inherent trends and patterns in the time series data. This process aims to equip the machine learning model with valuable and relevant information, thereby enhancing its capacity for accurate predictions.

Prior to modeling the time series, it's vital to undertake feature selection. This process involves identifying the features that have the most potent relationship with the target variable and discarding those that are irrelevant. The significance of feature selection lies in its ability to reduce the model's complexity, mitigate overfitting, and improve overall performance.

When our task is to forecast the subsequent three months of anomaly values in a monthly-based time series dataset, feature selection should pivot around the correlation of each feature with the target variable, as well as their ability to capture patterns and trends in the data over time. Carefully selecting the most relevant features can enable us to build a more accurate and comprehensible machine-learning model for time series forecasting.

The feature selection procedure was executed to identify the most relevant features for forecasting. As discussed previously, the "Impact" column was excluded due to its high percentage of missing values. Likewise, columns related to location, such as city, country, latitude, and longitude, were removed since they consistently displayed the same categorical values and weren't anticipated to enhance the forecasting process.

Additionally, the "All natural disasters" column was revealed to be a cumulative measure of the columns related to the disasters, and these columns displayed a robust correlation with the "All natural disasters" column. Hence, the individual disaster-related columns were removed, retaining the "All natural disasters" column.

The selection of optimal features was facilitated using a correlation matrix (Figure 16). This method identified features with the highest correlation to the target variable "Anomaly", including "upper and lower_95_CI," "Emission," "global_avg_temp," and "All natural disasters". Features with weaker correlations were omitted.

Following this, "upper_95_ci" and "lower_95_ci" were eliminated as they could be seen as derived features from the target variable "anomaly_value", serving as its confidence interval values. Similarly, the "global_avg_temp" feature was dropped to prevent overfitting, even though it exhibited a strong correlation with anomaly_value. Given the large number of outliers in "all natural disasters" and its position as the last feature on the list of those correlated with anomaly_values, it was also dropped to avoid overfitting.

Finally, the features deemed most relevant and beneficial for forecasting were "anomaly_value" and "emission". These selected features, along with the date column, were copied into a new dataframe named "data" for future computations (Galli 2022).

```
# plt.show()
corr_matrix = df.corr(numeric_only=True)
# Create a heatmap of the correlation matrix
plt.figure(figsize=(10, 8))
sns.heatmap(corr_matrix, cmap='coolwarm', annot=True, fmt="0.2f")
plt.show()
```

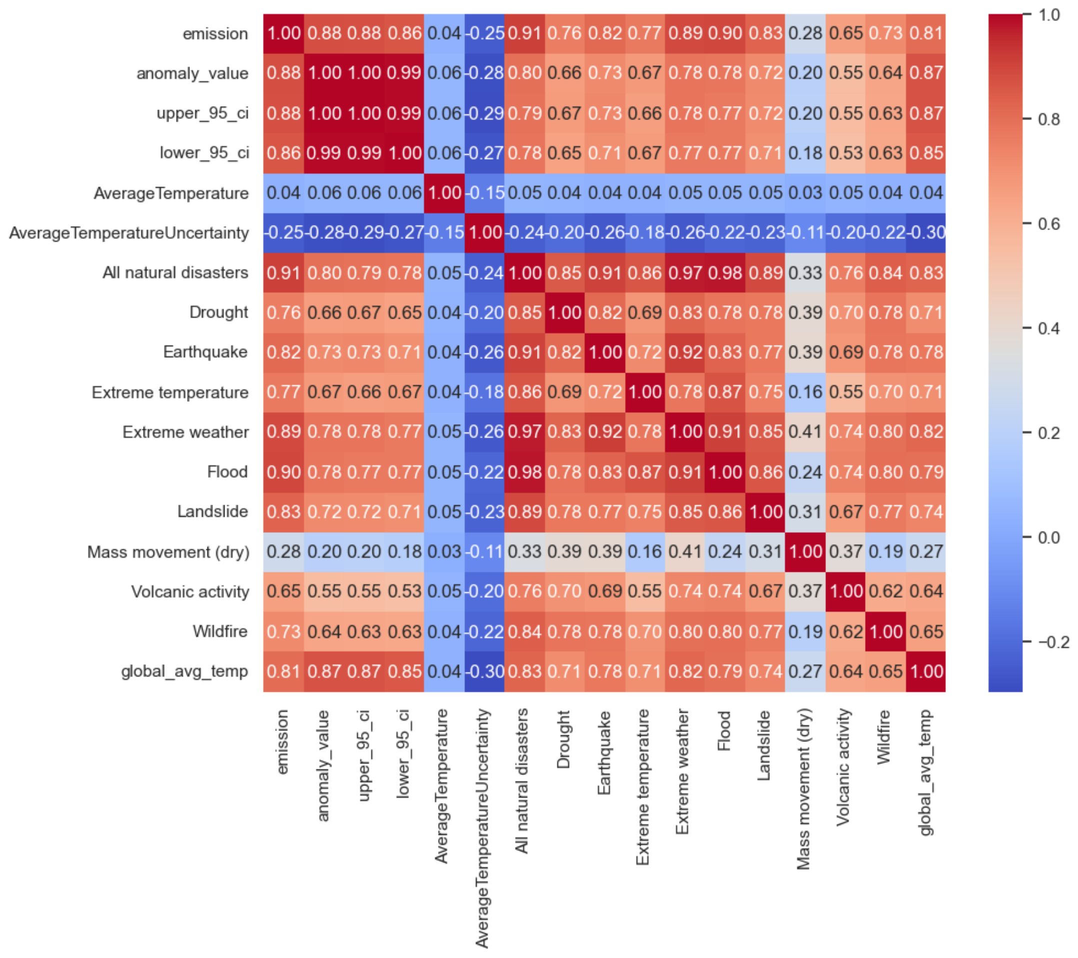


Figure 19 Correlation Heatmap

```
corr_matrix = df.corr(method='pearson', min_periods=1, numeric_only=True)['anomaly_value']
print(corr_matrix.sort_values(ascending=False))
```

Table 6 Correlation Table

anomaly_value	1.000000
upper_95_ci	0.995388
lower_95_ci	0.994781
emission	0.879636
global_avg_temp	0.869775
All natural disasters	0.796416
Extreme weather	0.784101
Flood	0.779836
Earthquake	0.725491
Landslide	0.721920
Extreme temperature	0.672100
Drought	0.663896
Wildfire	0.637369
Volcanic activity	0.548677
Mass movement (dry)	0.196517
AverageTemperature	0.057438
AverageTemperatureUncertainty	-0.282686
Name: anomaly_value, dtype: float64	

```
# Create a new DataFrame to avoid modifying the original one
df_selected = df[['anomaly_value', 'emission', 'global_avg_temp', 'All natural disasters']].copy()

# Create a new categorical variable for hue based on 'anomaly_value' variable using .loc
df_selected.loc[:, 'anomaly_value_category'] = pd.cut(df_selected['anomaly_value'], bins=3,
labels=['Low', 'Medium', 'High'])

# Now create the pairplot
sns.pairplot(df_selected, height=2, hue='anomaly_value_category')

# Show the plot
plt.show()
```

The pair plot reveals insightful information about the distributions of various variables and their relationships. In the distribution of 'emission', a high frequency of low values is evident as indicated by the prominent blue peak on the left. This could suggest that periods of low emission have been more common in the observed timeframe.

For both 'anomaly_value' and 'global_avg_temp', their distributions appear strikingly similar, reflecting the strong correlation between these two variables. However, in the case of high global average temperatures (green), the distribution is slightly bimodal with two peaks, possibly signifying two distinct groups within the high temperature anomalies.

'All natural disasters' distribution exhibits a different trend, with a thin blue peak for low values, a wide, short orange peak in the middle for medium values, and a broader, lower green peak for high values, which could indicate a diverse range of natural disaster frequencies.

The pair plot also illustrates that the relationship between 'emission' and 'anomaly_value' seems to be stronger than that of 'global_avg_temp' or 'All natural disasters' with 'anomaly_value'. This implies that 'emission' may be a better predictor for 'anomaly_value', highlighting the significant role of emissions in influencing temperature anomalies (KAMR 2021).

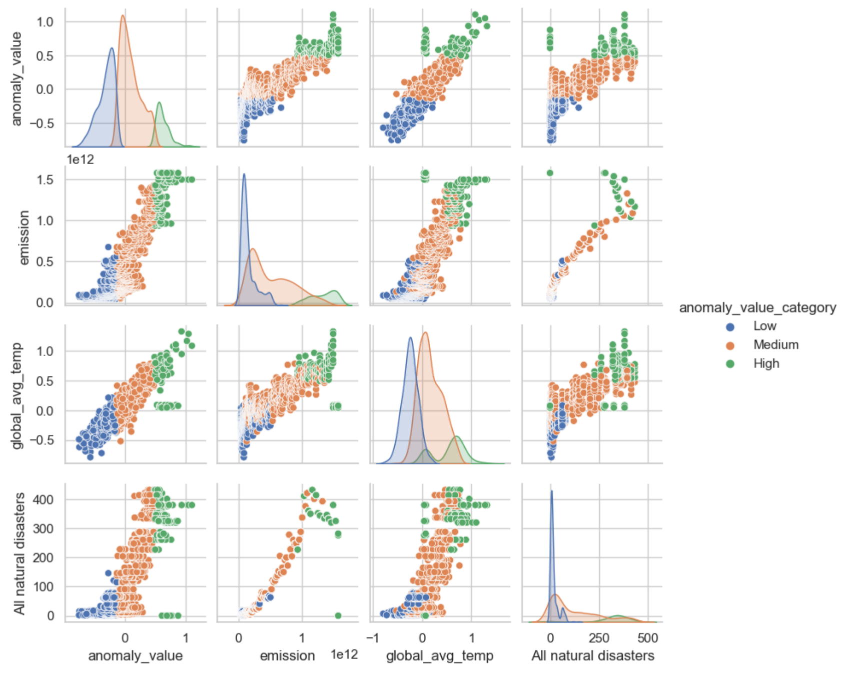


Figure 20 Pair Plot

```
df = df.reset_index()
data = df[['date', 'anomaly_value', 'emission']]
data = data.set_index('date')
```

```
data
```

```
            anomaly_value      emission
date                                   
1901-01-31         -0.075  4.482107e+10
1901-02-28         -0.176  4.482107e+10
1901-03-31         -0.272  4.482107e+10
1901-04-30         -0.236  4.482107e+10
1901-05-31         -0.187  4.482107e+10
...                   ...           ...
2019-09-30          0.719  1.580000e+12
2019-10-31          0.713  1.580000e+12
2019-11-30          0.752  1.580000e+12
2019-12-31          0.693  1.580000e+12
2020-01-31          0.879  1.580000e+12

[1429 rows x 2 columns]
```

4. Data Preparation for Machine Learning

Data preparation for machine learning and deep learning is a crucial stage in the process of model development and application. It involves converting raw data into a suitable format that can be interpreted and used by these complex algorithms. In this section, our objective is to prepare data for forecasting the subsequent three months using machine learning and deep learning models. Our preparation work can be divided into three primary subsections:

- Rolling Window: This step involves implementing a 'rolling window' or 'sliding window' approach, which is a type of data transformation particularly useful in time-series analysis. It treats a series of data points as a single observation, allowing the model to understand the trends and patterns over a specific period.
- Train-Test Split: This is a technique used for dividing the data into two subsets: a training set and a testing set. The training set is used to train and build the machine learning or deep learning model, and the testing set is used to evaluate the model's predictive performance and generalizability to unseen data.
- Data Normalization: This is a preprocessing step where we rescale the values of numeric columns in the dataset to a standard range (usually 0 to 1). This step is important to ensure that all features contribute equally to the model's performance, preventing features with larger scales from dominating the learning process.

The combination of these steps ensures a robust foundation for the application and efficiency of machine learning and deep learning models.

4.1 Sliding Window

The rolling window, or sliding window, technique plays a pivotal role in time-series data analysis. This approach treats a sequential set of data points as a single observation, making it an effective tool for temporal data transformation. By segmenting the data in this manner, a time-series model can grasp trends and patterns that unfold over a specified period. For instance, a rolling window of size 3 would group every three consecutive data points together as a single unit, then slide to the next set of three, and so forth. This method not only allows the model to analyze temporal dependencies between successive observations, but it also enhances the understanding of cyclical fluctuations, seasonal effects, and other time-sensitive phenomena. Thus, the rolling window approach offers invaluable insights to the time-series model, enabling it to make more informed and accurate predictions.

In the context of time series analysis, determining the optimal window size for the rolling window technique can significantly impact the performance of the model. One effective approach for identifying this optimal window size is through Partial Autocorrelation Function (PACF). PACF is a statistical measure that determines the correlation of a time series with a lagged version of itself, accounting for shorter lag correlations. By examining how the PACF changes for different lag values, we can identify the point at which the PACF decreases significantly. This point often provides an excellent estimation of the optimal window size. Hence, PACF can offer valuable guidance when selecting the best window size for our rolling window strategy, ensuring a balance between capturing relevant temporal dependencies and avoiding overfitting.

In the process of selecting the optimal size for the rolling window in time-series analysis, the Partial Autocorrelation Function (PACF) was utilized. This approach identifies the extent of correlation between a series and its lagged values. The PACF plot indicated significant correlations at lags 0, 1, 2, and 3. A PACF value of 1 at lag 0 was anticipated as a time series is naturally perfectly correlated with itself. The subsequent strong positive correlation at lag 1 suggested a substantial relationship between each data point and its predecessor, thus revealing a temporal dependency within the data. Even though the PACF values decreased at lags 2 and 3, their significance suggested a continued influence of past values, a characteristic of higher-order autoregressive processes. This understanding guided the choice of a rolling window size aligned with the identified significant lags, thereby enhancing the model's capacity to capture and learn the inherent temporal dynamics of the data (Brownlee 2017).

```
# Use seaborn styles
sns.set(style="whitegrid")

# Assuming 'df' is your DataFrame and 'anomaly_value' is the column of interest
series = df['anomaly_value']

# Create a figure and axis with a specific size
fig, ax = plt.subplots(figsize=(10, 6))
# Plot the PACF
plot_pacf(series, lags=24, ax=ax, title='Partial Autocorrelation Function (PACF) for Anomaly Value')

# Remove the plot frame lines
ax.spines["top"].set_visible(False)
ax.spines["right"].set_visible(False)
# Set labels size
ax.title.set_size(20)
ax.xaxis.label.set_size(16)
ax.yaxis.label.set_size(16)
# Set ticks size
ax.tick_params(axis='x', labelsize=14)
ax.tick_params(axis='y', labelsize=14)

plt.show()
```

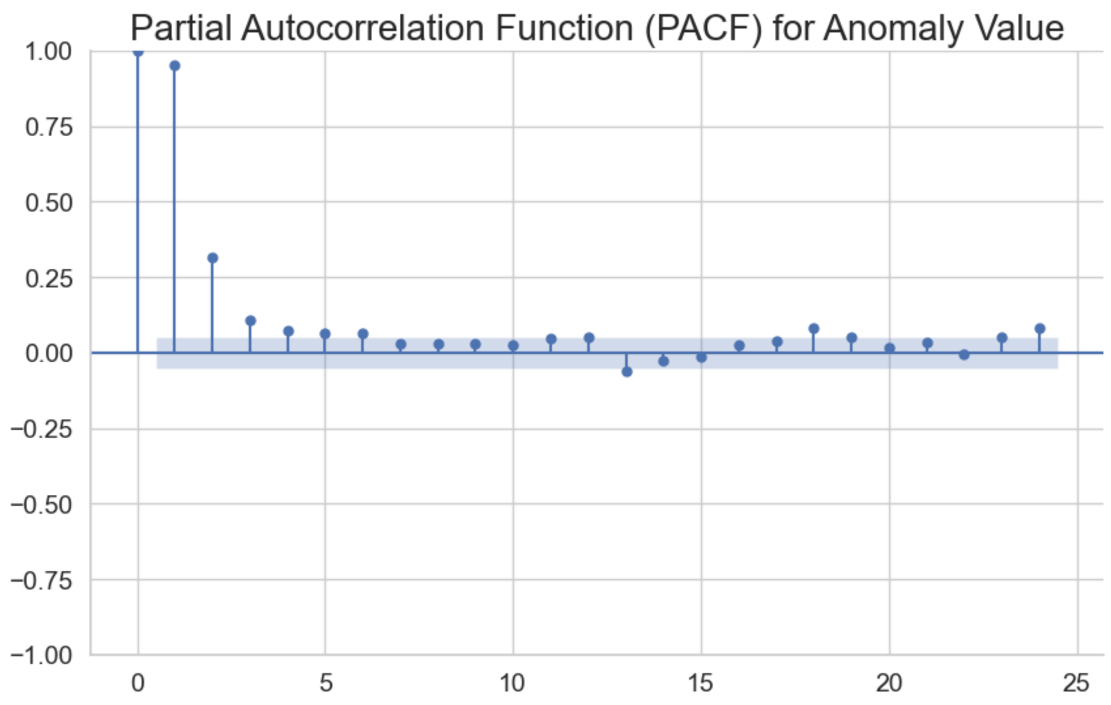


Figure 21 PACF for Anomaly Value

The given code below essentially prepares our dataset for the task of predicting the anomaly value three months into the future. Initially, it drops any rows with missing values to ensure data integrity. Subsequently, it defines the features and the target variable, where the target variable, 'anomaly_value', is shifted by three positions, representing a three-month ahead forecast. This causes the last three rows of the target variable to have missing values, which are subsequently dropped.

To encapsulate temporal dependencies in the features, a sliding window of size 3 is applied, creating new features that contain the 'anomaly_value' and 'emission' values from the previous three months. Each new feature is named following the pattern '{feature}_lag{i}', where 'feature' is the original feature name and 'i' is the number of months in the past.

The sliding window operation generates missing values for the initial few rows equivalent to the window size, which are then discarded. At the end of this procedure, your input data (X) and target data (y) are both prepared and aligned for the task of three-month ahead prediction, ensuring that each input-output pair corresponds to the correct time. The shapes of X and y are then printed to validate their dimensions.

```python
# Drop the rows with NaN values generated by shift
data = data.dropna()

# data as dataframe
features = ['anomaly_value', 'emission']
target = 'anomaly_value'

# Define input and output
X = data[features].copy()
y = data[target].shift(-3)  # target is the anomaly_value 3 months into the future
# Drop the rows with NaN values generated by shift
X = X.iloc[:-3]
y = y.dropna()

# Generate additional features using a sliding window of 3 months
window = 3
for feature in features:
    for i in range(1, window+1):
        X[f'{feature}_lag{i}'] = X[feature].shift(i)

# Drop the rows with NaN values generated by shift
X = X[window:]
y = y[window:]

print(X.shape)
print(y.shape)
```

```
(1423, 8)
(1423,)
```

4.2 Train-Test Split

The given code performs a train-test split on the data, assigning 80% of the data for training and 20% for testing, in a way that maintains the chronological order of the data (as indicated by shuffle=False). The dimensions of the resulting train and test sets are then printed for validation.

```
# Train-Test Split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, shuffle=False)

print("", X_train.shape, "\n", y_train.shape,
      "\n", X_test.shape, "\n", y_test.shape)
```

```
(1138, 8)
 (1138,)
 (285, 8)
 (285,)
```

Figure 22 below presents a clear illustration of our training and testing data split, mapped onto a plot. This graphical representation aids in understanding the division and distribution of our dataset across the train and test subsets. By visualizing this split, we can ensure the integrity of our modelling process and validate that the data division aligns well with the time-oriented structure of our dataset.

```
plt.figure(figsize=(10,4))
plt.plot(range(len(y_train)),y_train)
plt.plot(range(len(y_train),len(y_train)+len(y_test)),y_test)
plt.legend(['train','test'])
plt.show()
```

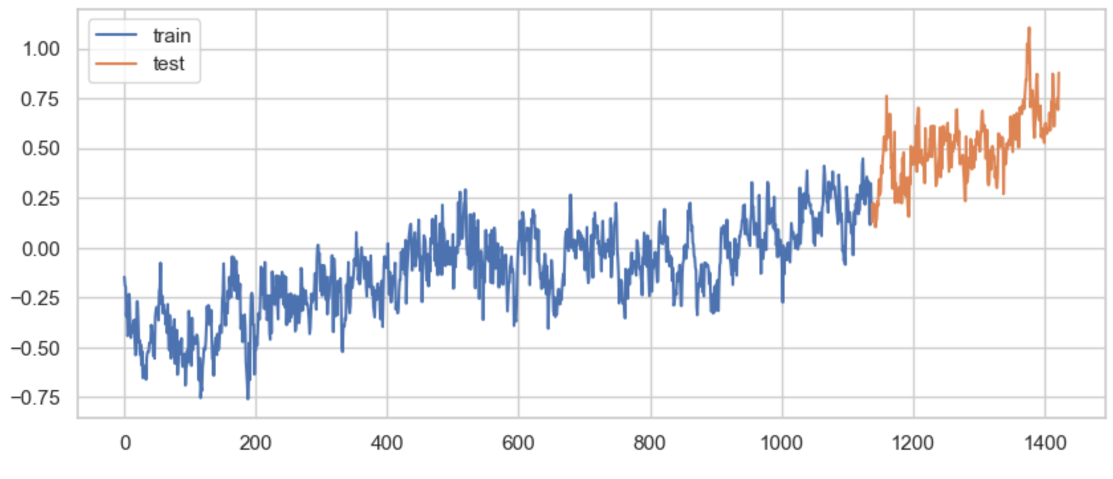


Figure 22 Train-Test Split Plot

4.3 Data Scaling

Data normalization is a critical pre-processing step in machine learning that ensures all numerical features in the dataset contribute equally to the learning process. This procedure prevents any feature from dominating the model due to its larger numerical range. It standardizes the features to a common scale, typically between 0 and 1.

The provided code segment accomplishes this using the MinMaxScaler from sklearn's preprocessing module. MinMaxScaler works by re-scaling each feature in our dataset into a specific range, typically between 0 and 1. The transformation is carried out according to the following formula:

$$X' = (X - X\_min) / (X\_max - X\_min)$$

- X is the original feature value.
- X' is the normalized feature value.
- X_min is the smallest value in the feature column.
- X_max is the largest value in the feature column.

Subtracting X_min from X shifts the range of the data so that it starts at zero. Dividing by (X_max - X_min) scales the range of the data to be between 0 and 1.

This scaling method ensures that the transformed feature has the same shape as the original distribution but with values that fall within the desired range. In the section below, it fits the scaler to the training and test data and then transforms them. The fit method computes the minimum and maximum values of X_train and X_test to be used for later scaling (Google For Developers 2023).

```
# Feature Scaling
scaler = MinMaxScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

5. Machin Learning models

5.1 Linear Regression

In the pursuit of our objective, which is to forecast temperature anomalies three months ahead, we have initially employed a Linear Regression model due to its comprehensibility and interpretability. Linear regression, a widely recognized statistical technique, predicts a continuous outcome (temperature anomalies in this scenario) based on one or more predictor variables (historical temperature anomalies, in our case). The underlying premise is a linear interplay between the predictor and outcome variables. It strives to identify an optimal fit that minimizes the sum of squared differences (residuals) between the actual observations and the predictions inferred by the model.

Post the model training and subsequent predictions, we have gauged its effectiveness through a gamut of performance metrics:

Mean Squared Error (MSE): A metric indicative of the average of squared deviations between the predictions and the actual observations. A MSE of 0.01214 for our model suggests that the predictions made by our model are near the actual values, indicating reasonably good performance.

Root Mean Squared Error (RMSE): Essentially the square root of MSE, RMSE is particularly useful as it is in the same unit as the outcome variable. With a RMSE of 0.11018, our model's performance can be deemed as satisfactory.

Mean Absolute Error (MAE): This metric denotes the average of the absolute differences between the predicted and the observed values, devoid of any directionality. The MAE for our model stands at 0.08649, reflecting a decent predictive efficacy of our model.

Mean Absolute Percentage Error (MAPE): MAPE manifests the prediction errors as a percentage, a metric extremely beneficial while communicating model accuracy to an audience not well-versed with absolute error measures. A MAPE of about 19.63% for our model implies that, on an average, the model's predictions deviate from the actual values by roughly 19.63%.

Median Absolute Error (MedAE): MedAE represents the median of the absolute differences between the predicted and the observed values. With a MedAE of 0.07128, it is inferred that the median magnitude of error committed by our model is relatively small.

R-squared (R^2): R^2 is a statistical measure that depicts the proportion of the dependent variable's variance that can be explained by the independent variable(s) in the model. An R^2 of 0.5640 for our model suggests that it can explain approximately 56.40% of the variability in temperature anomalies, which is moderately good.

Based on the metrics, it can be concluded that our Linear Regression model has demonstrated a fair performance in this forecasting task. It's important to underline, however, that linear regression, due to its inherent simplicity, may fail to account for complex patterns and relationships existent in our data. Therefore, we ought to explore more advanced models like Random Forest or LSTM that are adept at capturing such complex relationships, thereby enhancing predictive accuracy.

```
# Model Building
model = LinearRegression()

# Model Training
model.fit(X_train, y_train)

# Making Predictions
predictions = model.predict(X_test)

# Model Evaluation
mse = mean_squared_error(y_test, predictions)
rmse = np.sqrt(mse)
mae = mean_absolute_error(y_test, predictions)
mape = np.mean(np.abs((y_test - predictions) / y_test)) * 100
medae = median_absolute_error(y_test, predictions)
r2 = r2_score(y_test, predictions)


print("Mean Squared Error:", mse)
print("Root Mean Squared Error:", rmse)
print("Mean Absolute Error:", mae)
print("Mean Absolute Percentage Error:", mape)
print("Median Absolute Error:", medae)
print("R-squared:", r2)
```

```
Mean Squared Error: 0.012138902058315002
Root Mean Squared Error: 0.11017668563863682
Mean Absolute Error: 0.08648892667096783
Mean Absolute Percentage Error: 19.63384282301336
Median Absolute Error: 0.07128186705499084
R-squared: 0.5640001256922758
```

5.1.1 Actual vs Predicted Values Over Time Plot

The plot 'Actual vs Predicted Values Over Time' (Figure 20) aims to graphically compare the actual and predicted values of temperature anomalies over time. By doing so, it provides us with a visual representation of the model's performance. By examining the graph, the predicted values generally follow the trend of the actual values. However, there are significant cases where the model predictions differ from the actual data. This is particularly evident during periods when the actual data have a large increase or decrease, indicating that while the model can capture the overall trend, there may be difficulties in accurately predicting sudden changes in temperature anomalies.

```
plt.figure(figsize=(14, 6))
plt.plot(y_test.index, y_test, label='Actual', color='blue')
plt.plot(y_test.index, predictions, label='Predicted', color='red')
plt.xlabel('Time')
plt.ylabel('Target Value')
plt.title('Actual vs Predicted Values Over Time')
plt.legend()
plt.show()
```

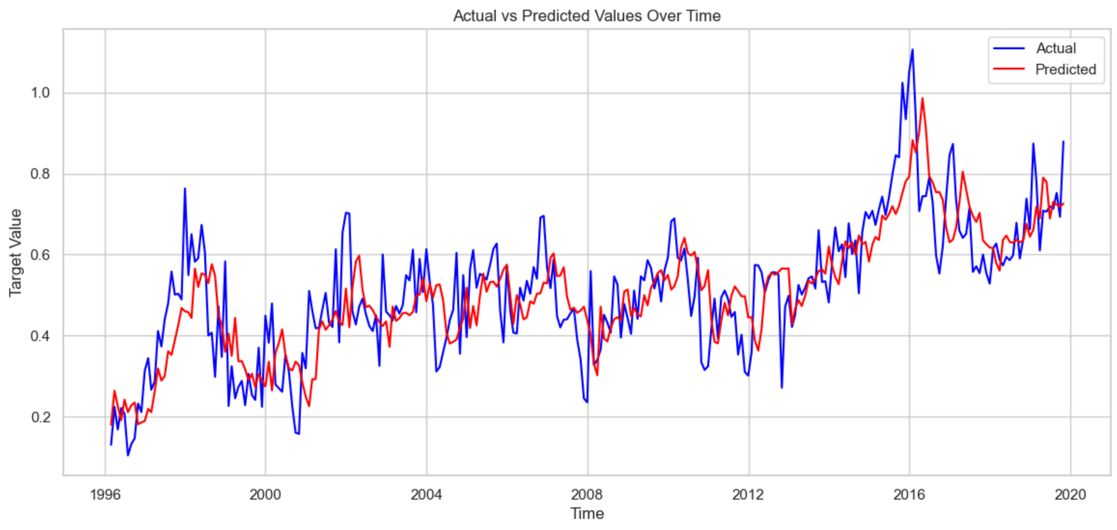


Figure 23 Actual vs Predicted Values_Linear regression.

5.1.2 Actual vs Predicted Distribution Plot

The Kernel Density Estimate (KDE) plot visualizes the distribution of actual and predicted temperature anomalies (Figure 21). Both distributions demonstrate a bell-shaped curve, indicating a typical Gaussian distribution. The close alignment of these two distributions validates the reasonable performance of our model. However, there's a slight elevation in the predicted curve at the central peak, suggesting a tendency of the model to overestimate temperature anomalies at the high-density point.

```
# Plotting the actual and predicted values in the original scale
sns.kdeplot(y_test, label='Actual')
sns.kdeplot(predictions, label='Predicted')
plt.xlabel('Temperature Anomaly')
plt.ylabel('Density')
plt.title('Actual vs Predicted Distribution Plot')
plt.legend()
plt.show()
```

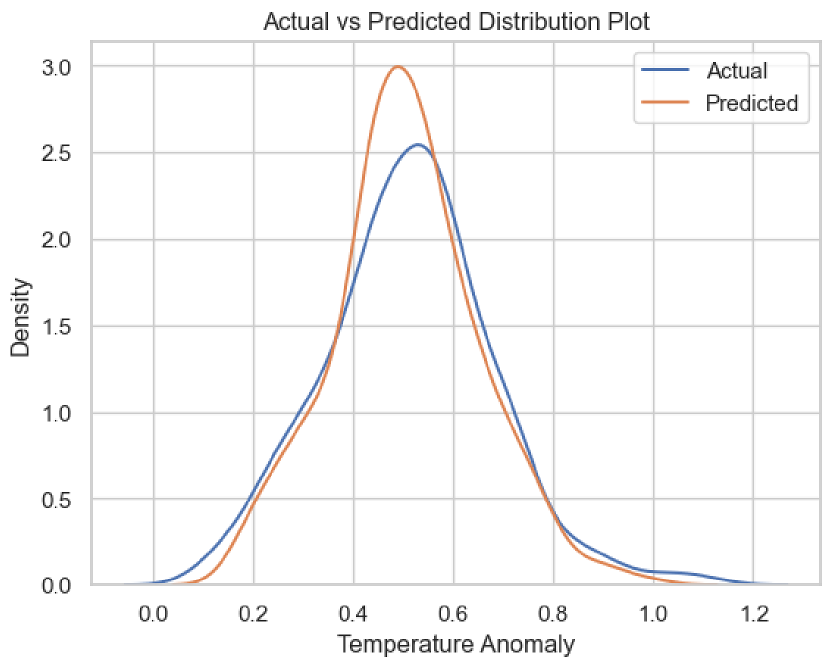


Figure 24 Actual vs Predicted Distribution Plot_Linear Regression.

5.1.3 Residuals Over Time Plot

The plot presents the residuals (differences between actual and predicted values) over time (Figure 22). From this, we can see the residuals largely fall within a range, suggesting our model's predictions are fairly accurate. The lack of any distinct patterns or trends in the residuals indicates that our model is capturing the main trends in the data, without systematically mispredicting certain time periods. Thus, despite some errors, the model's performance appears to be satisfactory.

```
residuals = y_test – predictions
plt.figure(figsize=(14, 6))
plt.plot(y_test.index, residuals, label='Residuals', color='blue')
plt.xlabel('Time')
plt.ylabel('Residual')
plt.title('Residuals Over Time')
plt.show()
```

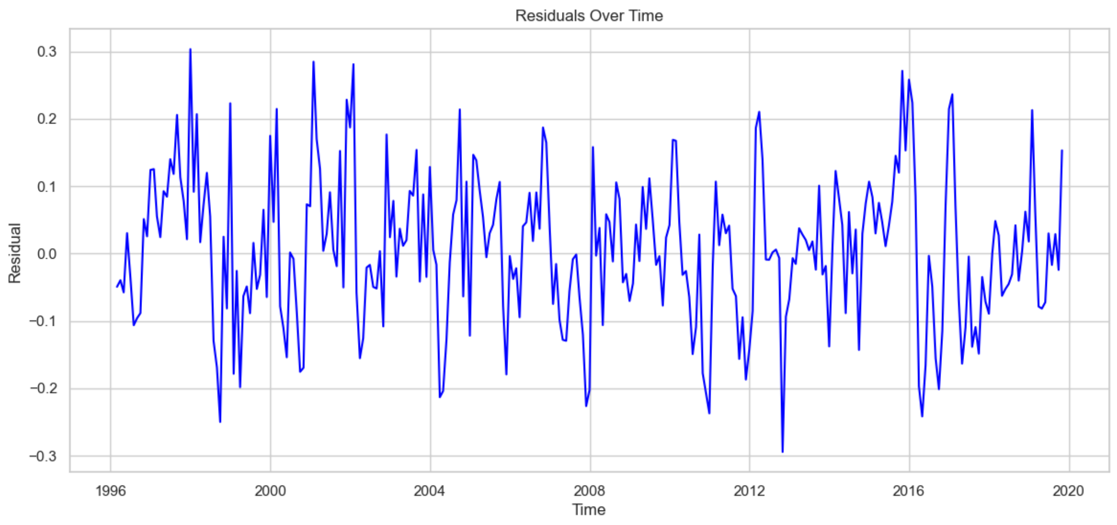


Figure 25 Residuals Over Time_Linear Regression

5.2 Random Forest

After establishing a baseline performance using a simple linear regression model, I opted to implement a Random Forest Regressor as the next step in our modeling process for temperature anomaly forecasting. The primary motive behind this choice is the ability of Random Forests to capture more complex relationships in data, which linear regression might miss. Random Forest is a flexible, easy-to-use machine learning algorithm that produces, even without hyper-parameter tuning, a good result most of the time. It is also one of the most used algorithms, because of its simplicity and diversity.

Looking at the results obtained from the Random Forest model, we can see that the model's performance has not improved over the linear regression model, and in fact, it appears to have worsened.

The Mean Squared Error (MSE), Root Mean Squared Error (RMSE), and Mean Absolute Error (MAE) are all larger than those obtained from the linear regression model, indicating that, on average, the Random Forest model's predictions are further from the true values.

Furthermore, the Mean Absolute Percentage Error (MAPE) is approximately 46.80%, indicating that the model's predictions are, on average, off by about 46.80%. This is a larger error percentage than we saw with the linear regression model.

Perhaps most notably, the R-squared score is negative, which indicates that the Random Forest model is performing worse than a model that would always predict the mean of the target variable. In other words, the model isn't capturing the variance in the data and is making poor predictions.

These results suggest that the Random Forest model may not be well-suited to this problem, or it may need further refinement and tuning to improve its performance. It's also possible that other types of models may be better suited to this task, such as deep learning models.

```
# Model Building
model = RandomForestRegressor(n_estimators=5, random_state=0)

# Model Training
model.fit(X_train, y_train)

# Making Predictions
predictions = model.predict(X_test)

# Model Evaluation
mse = mean_squared_error(y_test, predictions)
rmse = np.sqrt(mse)
mae = mean_absolute_error(y_test, predictions)
mape = np.mean(np.abs((y_test - predictions) / y_test)) * 100
medae = median_absolute_error(y_test, predictions)
r2 = r2_score(y_test, predictions)


print("Mean Squared Error:", mse)
print("Root Mean Squared Error:", rmse)
print("Mean Absolute Error:", mae)
print("Mean Absolute Percentage Error:", mape)
print("Median Absolute Error:", medae)
print("R-squared:", r2)
```

```
Mean Squared Error: 0.08766384895126707
Root Mean Squared Error: 0.2960808148990189
Mean Absolute Error: 0.2546283040935673
Mean Absolute Percentage Error: 45.611360780166585
Median Absolute Error: 0.25240000000000007
R-squared: -2.148672502708146
```

5.2.1 Actual vs Predicted Values Over Time Plot.

The visual comparison of actual and predicted values for temperature anomalies (Figure 22) shows a clear discrepancy. The blue line representing the actual temperature anomalies over time shows a noticeable dynamic pattern, while the red line, which represents the predictions made by our Random Forest model, is nearly flat, barely capturing any variation.

This further indicates that the Random Forest model struggles to accurately capture the underlying temporal trends and fluctuations in the temperature anomaly data. Only in a few instances do the predictions come close to the actual values. The flatness of the predicted line suggests the model's inability to capture the time-dependent variance and trend in the data, which is crucial for effective forecasting.

This visualization further underscores the poor performance indicated by the numerical evaluation metrics, signaling that either more model refinement or a different modeling approach is necessary for more accurate forecasting of temperature anomalies.

```
plt.figure(figsize=(14, 6))
plt.plot(y_test.index, y_test, label='Actual', color='blue')
plt.plot(y_test.index, predictions, label='Predicted', color='red')
plt.xlabel('Time')
plt.ylabel('Target Value')
plt.title('Actual vs Predicted Values Over Time')
plt.legend()
plt.show()
```

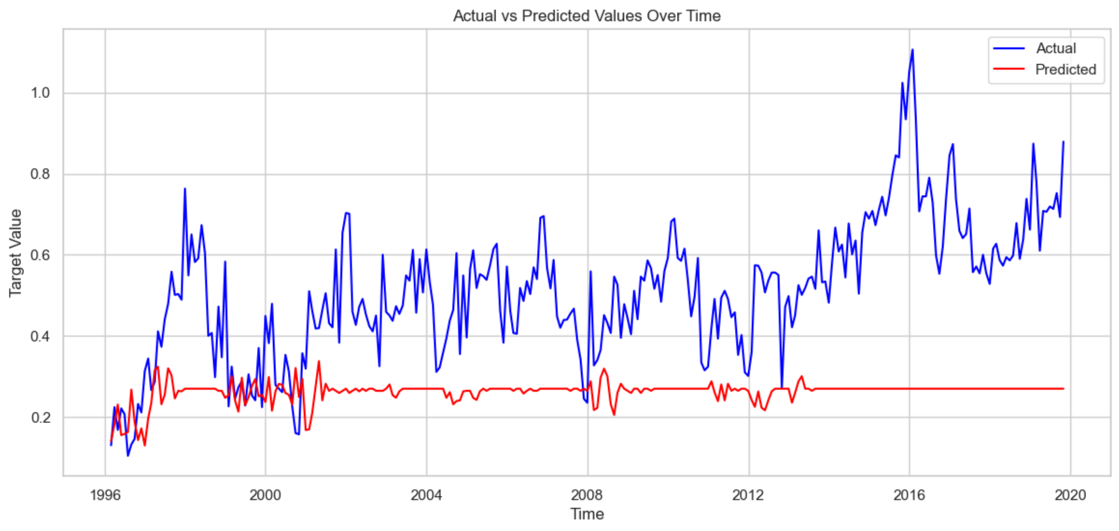


Figure 26 Actual vs Predicted Values Over Time_Random Forest

5.2.2 Actual vs Predicted Distribution Plot.

The Kernel Density Estimate (KDE) plot (Figure 23) shows a significant difference between the actual and predicted temperature anomalies. The actual values show a single peak, suggesting a normal distribution, whereas the predicted values have two peaks, indicating a bimodal distribution. This suggests the Random Forest model is not accurately capturing the variance in the data. Furthermore, the narrower peaks for the predicted values imply less variability compared to the actual values. This highlights that our model needs improvements for better forecasting accuracy.

```
# Plotting the actual and predicted values in original scale
sns.kdeplot(y_test, label='Actual')
sns.kdeplot(predictions, label='Predicted')
plt.xlabel('Temperature Anomaly')
plt.ylabel('Density')
plt.title('Actual vs Predicted Distribution Plot')
plt.legend()
plt.show()
```

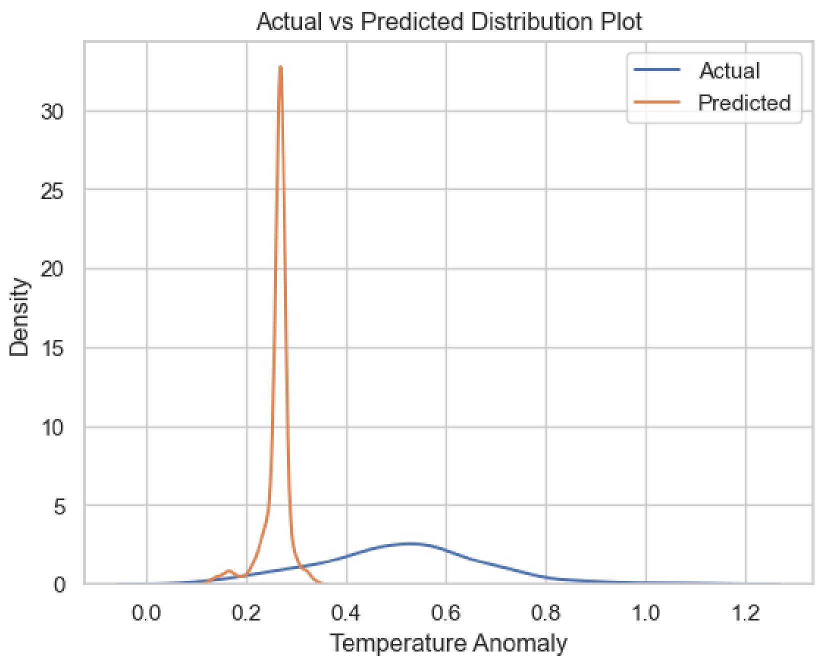


Figure 27 Actual vs Predicted Distribution Plot_Random Forest

5.2.3 Residuals Over Time Plot.

The plot of residuals over time shows significant variance, indicating inconsistency in the model's errors. The large jumps and lack of a clear pattern suggest that the Random Forest model may struggle to capture complex dependencies in our data. The spread of the residuals also suggests that the model's performance varies significantly over different periods. This further underscores the need for model refinement or potentially exploring other, more forecasting models.

```
residuals = y_test - predictions

plt.figure(figsize=(14, 6))
plt.plot(y_test.index, residuals, label='Residuals', color='blue')
plt.xlabel('Time')
plt.ylabel('Residual')
plt.title('Residuals Over Time')
plt.show()
```

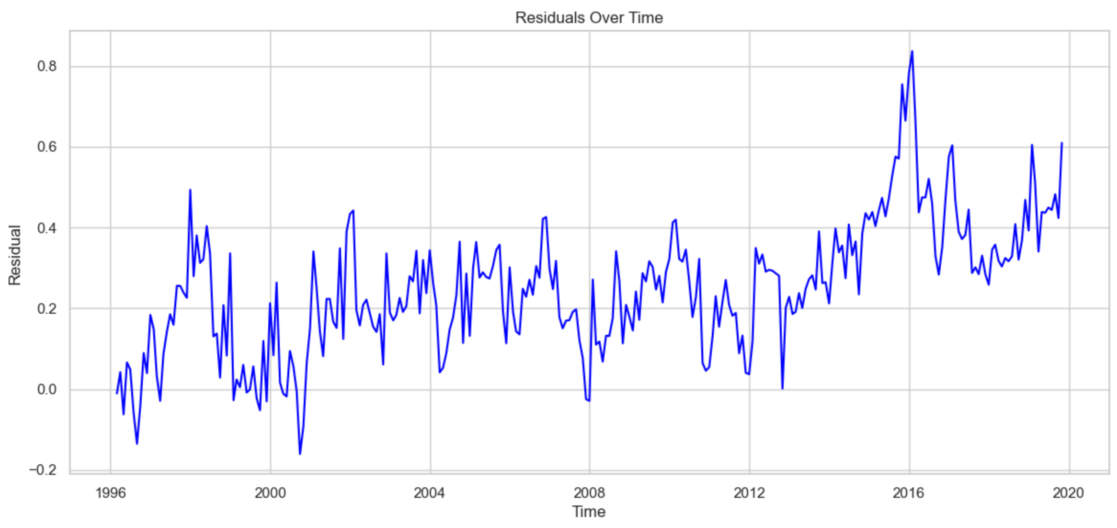


Figure 28 Residuals Over Time_Random Forest

6. Deep Learning models (LSTM)

Shifting to a more advanced modeling technique, we'll now employ Long Short-Term Memory (LSTM) networks - a type of Recurrent Neural Network (RNN) well-suited for time-series data. Temperature anomalies exhibit temporal dependencies, meaning the current value often relies on previous ones. This characteristic makes our task a perfect fit for LSTM. These networks "remember" past information using their hidden state, allowing them to process sequences of data and capture the temporal dynamics intrinsic to our task.

Simpler models might overlook these complex patterns, so using LSTM can help us better capture the data's sequential nature and potentially improve forecasting accuracy for temperature anomalies.

In this section, we establish a fresh dataframe, data2, mirroring the structure and values of the original dataframe data. This ensures a clean slate to work on specifically for our LSTM model, maintaining the integrity of our original dataset while facilitating any unique preprocessing steps required by the LSTM framework.

```
# Create a new dataframe
data2 = df[[ 'date', 'anomaly_value', 'emission']]
data2 = data2.set_index('date')
```

6.1 Data Preparation for Deep Learning

The journey of readying our data for Long Short-Term Memory (LSTM) models, a type of deep learning algorithm, is an integral phase of our predictive modelling framework. During this stage, our efforts are primarily channelled into molding our dataset to align with the distinct specifications of LSTM models. The tasks we undertake in this respect are:

6.1.2 Shifting Target

Given our objective to predict temperature anomalies three months ahead, we reframe our target variable by shifting it backwards. This ensures that for every data point, our target represents the anomaly value three months into the future.

```
# Drop the rows with NaN values generated by shift
data2 = data2.dropna()

features = ['anomaly_value','emission']
target = 'anomaly_value'
# Define input and output
X = data2[features].copy()
y = data2[target].shift(-3)  # target is the anomaly_value 3 months into the future
# Drop the rows with NaN values generated by shift
X = X.iloc[:-3]
y = y.dropna()
window = 3
for feature in features:
    for i in range(1, window+1):
        X[f'{feature}_lag{i}'] = X[feature].shift(i)

# Drop the rows with NaN values generated by shift
X = X[window:]
y = y[window:]
```

6.1.3 Data Scaling and Reshaping

LSTM models are sensitive to the scale of data. Hence, we bring all our feature values onto the same scale, specifically between 0 and 1, using the MinMaxScaler. This prevents high-magnitude features from overpowering others during the model training process. And reshaping to conform our data to the three-dimensional structure that LSTM models expect: [samples, timesteps, features]. This ensures our data fits seamlessly into the LSTM model's architecture.

```
# Scale the features
scaler = MinMaxScaler(feature_range=(0, 1))
X = scaler.fit_transform(X)
y = scaler.fit_transform(y.values.reshape(-1,1))

# Reshape input to be 3D [samples, timesteps, features] which is required for LSTM
X = X.reshape((X.shape[0], 1, X.shape[1]))
```

6.1.4 Train-Test Split

In this section, we repeat the train-test split process similar to what was done for our machine learning models, but this time for our LSTM model. This crucial step allows us to test our LSTM's ability to generalize learned patterns to unseen data.

```
# Train-Test Split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, shuffle=False)
```

```
print("", X_train.shape, "\n", y_train.shape,
      "\n", X_test.shape, "\n", y_test.shape)
```

```
(1138, 1, 8)
 (1138, 1)
 (285, 1, 8)
 (285, 1)
```

6.2 Long short-term memory (LSTM)

Our LSTM model is designed with a single LSTM layer, a Dropout layer, and a Dense output layer. This structure was found to give the best performance during testing. The LSTM layer, with 80 neurons, is key for processing sequences of data points, allowing the model to learn and remember long-term dependencies. Following the LSTM layer, we introduce a Dropout layer to reduce overfitting, randomly setting a fraction (20% in this case) of input units to 0 at each update during training. The Dense layer acts as an output layer, producing a single output, our forecasted temperature anomaly.

The model is compiled and trained using Mean Absolute Error as the loss function and Adam as the optimizer, both of which are common choices for regression problems. We also utilize Early Stopping to prevent overfitting by stopping training when validation loss stops improving.

However, before evaluating the model's performance, it's essential to inverse the scaling we initially performed. This step transforms the output back into the same scale as our original temperature data, which allows us to accurately calculate the error metrics.

Based on the following training log and metrics results, we can further analyze the performance of our LSTM model in predicting temperature anomalies:

The training log shows a decreasing loss value across the 12 epochs, indicating that the model is learning and adjusting its predictions to minimize the difference between the predicted and actual values. This suggests that the model is converging towards an optimal solution.

The Mean Squared Error (MSE) and Root Mean Squared Error (RMSE) provide measures of the overall prediction accuracy. With an MSE of 0.012 and an RMSE of 0.11, the model's average prediction errors are relatively small. This indicates that the model can make predictions that are close to the actual values, contributing to its reasonable performance.

The Mean Absolute Error (MAE) and Mean Absolute Percentage Error (MAPE) are both relatively low, with an MAE of 0.086 and a MAPE of 19.51%. These metrics measure the average absolute difference and average percentage difference between the predicted and actual values, respectively. The low values suggest that, on average, the model's predictions are close to the actual values, contributing to its reasonable performance.

The R-squared score of approximately 0.57 indicates that the model can explain around 57% of the variability in the temperature anomalies. This means that the predictor variables, historical temperature anomalies in this case, can account for a moderate proportion of the observed variability. While a higher R^2 score would indicate a stronger relationship between the predictor and outcome variables, an R2 of 0.57 still suggests a reasonable level of explanatory power.

Overall, the provided results and analysis demonstrate a reasonable performance of our LSTM model in predicting temperature anomalies. The relatively low MAE and MAPE values, along with the R-squared score, indicate that the model captures a significant portion of the variability in the data. However, there is still room for improvement in future iterations, where further refinements and enhancements can be explored to increase the model's predictive accuracy.

```
#  Define LSTM model
model = Sequential()
model.add(LSTM(80, input_shape=(X_train.shape[1], X_train.shape[2])))
model.add(Dropout(0.2))  # Add dropout with a probability of 0.2
model.add(Dense(1))
# Compile and train the model
model.compile(loss='mae', optimizer='adam')
# Define early stopping
early_stop = EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True)
# Fit the model on the training data
history = model.fit(X_train, y_train, epochs=50, batch_size=72, validation_data=(X_test, y_test),
verbose=1, callbacks=[early_stop])

# Make a prediction
yhat = model.predict(X_test)
X_test = X_test.reshape((X_test.shape[0], X_test.shape[2]))

# Invert scaling for forecast
inv_yhat = np.concatenate((yhat, X_test[:, 1:]), axis=1)
inv_yhat = scaler.inverse_transform(inv_yhat)
inv_yhat = inv_yhat[:,0]

# Invert scaling for actual
```

```
y_test = y_test.reshape((len(y_test), 1))
inv_y = np.concatenate((y_test, X_test[:, 1:]), axis=1)
inv_y = scaler.inverse_transform(inv_y)
inv_y = inv_y[:,0]

# Calculate metrics
mse = mean_squared_error(inv_y, inv_yhat)
rmse = np.sqrt(mse)
mae = mean_absolute_error(inv_y, inv_yhat)
mape = np.mean(np.abs((inv_y - inv_yhat) / inv_y)) * 100
medae = median_absolute_error(inv_y, inv_yhat)
r2 = r2_score(inv_y, inv_yhat)

print("Mean Squared Error:", mse)
print("Root Mean Squared Error:", rmse)
print("Mean Absolute Error:", mae)
print("Mean Absolute Percentage Error:", mape)
print("Median Absolute Error:", medae)
print("R-squared:", r2)
```

```
Epoch 1/50
16/16 [==============================] - 4s 45ms/step - loss: 0.2483 - val_loss: 0.2508
Epoch 2/50
16/16 [==============================] - 0s 6ms/step - loss: 0.0876 - val_loss: 0.1529
Epoch 3/50
16/16 [==============================] - 0s 6ms/step - loss: 0.0657 - val_loss: 0.0720
Epoch 4/50
16/16 [==============================] - 0s 5ms/step - loss: 0.0576 - val_loss: 0.0475
Epoch 5/50
16/16 [==============================] - 0s 5ms/step - loss: 0.0561 - val_loss: 0.0496
Epoch 6/50
16/16 [==============================] - 0s 6ms/step - loss: 0.0547 - val_loss: 0.0471
Epoch 7/50
16/16 [==============================] - 0s 6ms/step - loss: 0.0562 - val_loss: 0.0464
Epoch 8/50
16/16 [==============================] - 0s 5ms/step - loss: 0.0546 - val_loss: 0.0472
Epoch 9/50
16/16 [==============================] - 0s 6ms/step - loss: 0.0558 - val_loss: 0.0467
Epoch 10/50
16/16 [==============================] - 0s 6ms/step - loss: 0.0539 - val_loss: 0.0470
Epoch 11/50
16/16 [==============================] - 0s 6ms/step - loss: 0.0543 - val_loss: 0.0525
Epoch 12/50
```

```
16/16 [==============================] - 0s 6ms/step - loss: 0.0544 - val_loss: 0.0473
9/9 [==============================] - 0s 2ms/step
Mean Squared Error: 0.012038987054645973
Root Mean Squared Error: 0.10972231794236745
Mean Absolute Error: 0.08646352684839426
Mean Absolute Percentage Error: 19.51094816185334
Median Absolute Error: 0.07047697061300262
R-squared: 0.5675888299121368
```

6.2.1 Loss Over Epochs Plot

This plot displays the decline of training and validation loss over different epochs. Initially, both losses are high, near 0.250, signifying a sizable difference between model predictions and actual data. However, both sharply decrease from epoch 0 to 2, indicating quick learning and improving predictions.

The training loss plateaus at around 0.060 from epoch 2 to 11, showing that further significant improvements on the training data are limited. Contrarily, the validation loss shows continual slight decrease even beyond epoch 3, reaching near 0.050, indicating continued model refinement on unseen data.

This plot demonstrates a model that learns rapidly during the early epochs and then stabilizes while still improving on new data, striking a balance between training performance and generalization.

```
plt.figure(figsize=(10,6))
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Loss Over Epochs')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

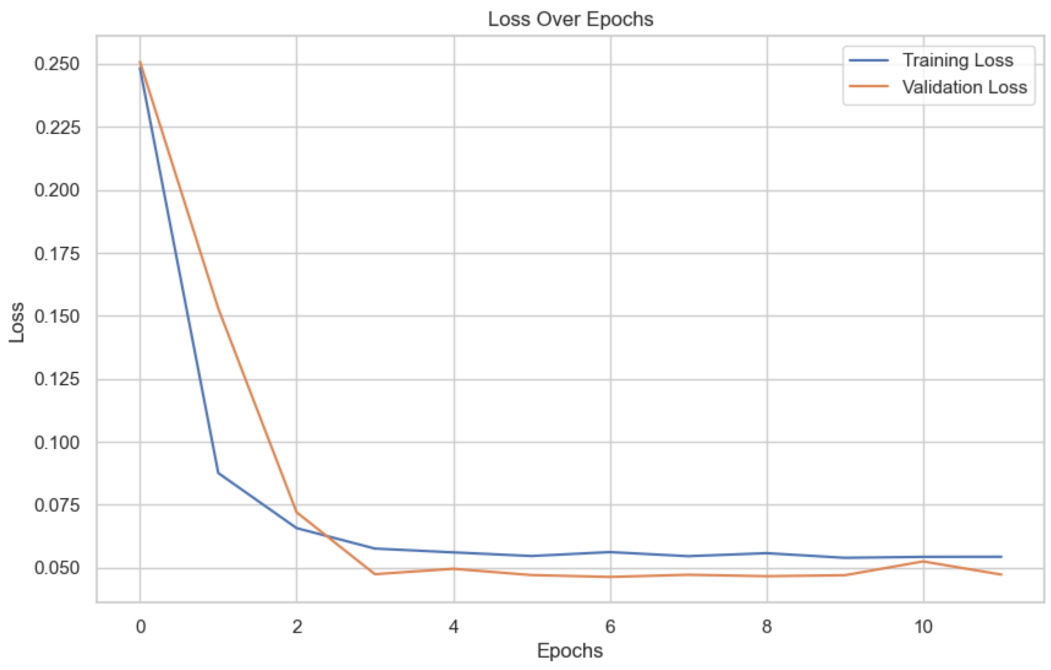


Figure 29 Loss Over Epochs

6.2.2 Predicted vs Actual Values Over Time

The plot indicates that while the LSTM model manages to capture the trend of the actual values to a certain extent, it seems to struggle with accurately predicting the magnitude of fluctuations. Particularly, the model's predictions appear smoother with less pronounced peaks and troughs compared to the actual values, especially in the first and last quarters of the data. The maximum fluctuation in predicted values is about 0.3, compared to around 0.7 in the actual values. This indicates the model's difficulty in dealing with high volatility periods. Notably, the model's performance appears to deteriorate towards the end of the dataset, suggesting a possible weakening in its predictive capacity over time. Despite these shortcomings, the model exhibits some predictive ability, demonstrating the potential of LSTM for this type of forecasting task. However, it may require further tuning and enhancements for more accurate predictions.

```
plt.figure(figsize=(10,6))
plt.plot(inv_y, label='Actual')
plt.plot(inv_yhat, label='Predicted')
plt.title('Predicted vs Actual Values')
plt.xlabel('Time')
plt.ylabel('Value')
plt.legend()
plt.show()
```

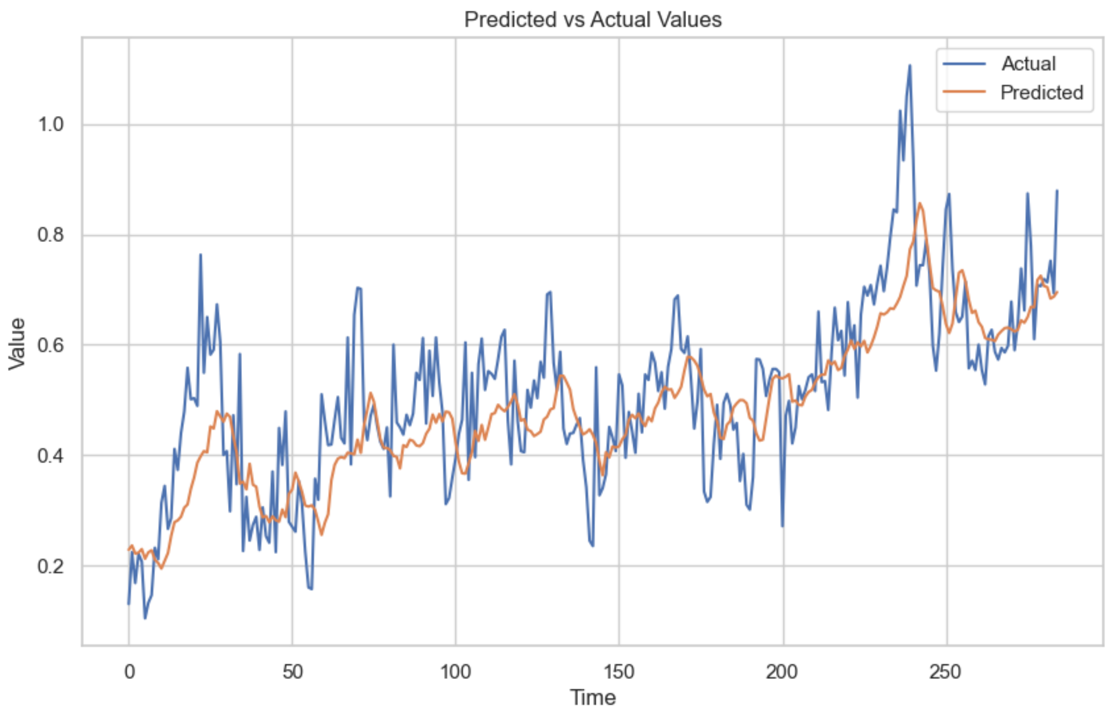


Figure 30 Predicted vs Actual Values_LSTM

6.2.3 Residuals vs Predicted Values Plot

The plotted residuals versus predicted values provide insights into the prediction errors made by the LSTM model. There's a notable dispersion of residuals, implying variability in prediction errors across different data points. The dense cluster of points in the 0.4 to 0.6 range on the X-axis suggests that the model tends to generate prediction errors within this range. However, the fact that many points are near the middle red line (indicating zero error) suggests that the model has achieved reasonable accuracy for a substantial number of predictions. Still, the visible spread of residuals indicates areas where the model's predictions could be improved, which could be addressed through further model tuning and refinement.

```
residuals = inv_y - inv_yhat
plt.figure(figsize=(10,6))
plt.scatter(inv_yhat, residuals)
plt.axhline(y=0, color='r', linestyle='-')
plt.title('Residuals vs Predicted Values')
plt.xlabel('Predicted Values')
plt.ylabel('Residuals')
plt.show()
```

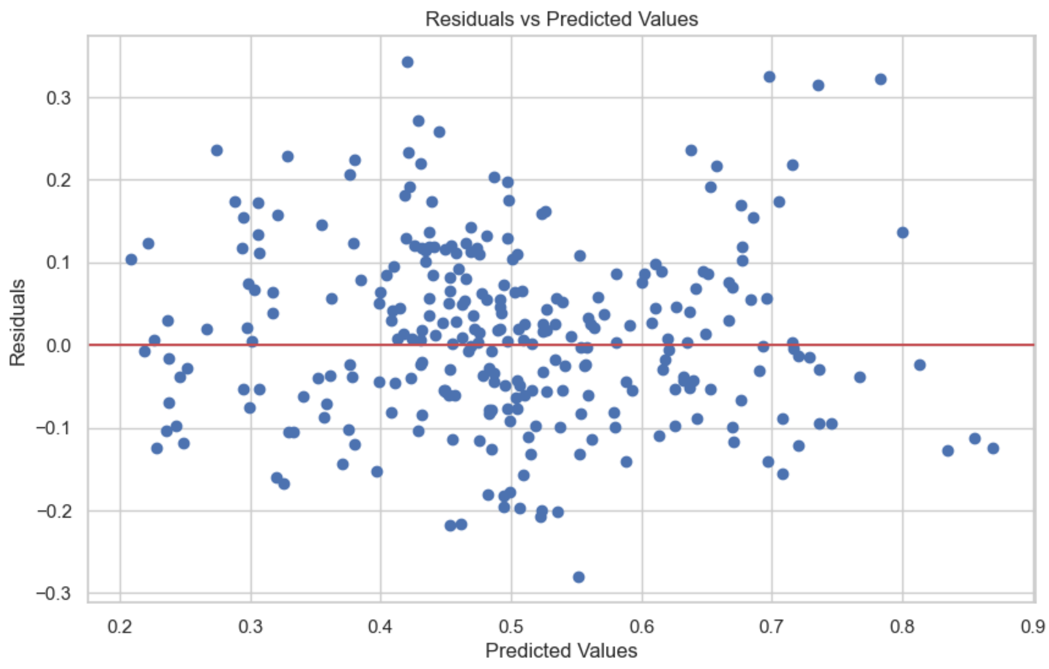


Figure 31 Residuals vs Predicted Values_LSTM

7. Comparison of Models

In this endeavor to predict temperature anomalies three months into the future, three distinct models were implemented: Linear Regression, Random Forest, and LSTM (Long Short-Term Memory). Each of these models has a unique approach to solving this time-series forecasting problem.

Linear Regression is a simple and well-understood model that assumes a linear relationship between input and output variables. This model's simplicity, however, might be its downfall in this context. While it was able to capture some of the general trends in the data, as shown by its moderate R-squared value of 0.564, it lacks the ability to capture complex temporal dependencies that are present in time series data. This linear model might overlook certain non-linear patterns in the data, making its forecasting ability less reliable for our task.

The Random Forest model is a more flexible and robust model that does not rely on a linear assumption. It builds multiple decision trees and merges them to get more accurate and stable predictions. However, Random Forest models are not inherently designed for time-series data. In our task, even though attempts were made to handle non-stationarity in the data, the Random Forest model performed poorly. The R-squared value of -2.148 shows that the model was not able to accurately forecast the future temperature anomalies. This suggests that it may not be the most suitable choice for our time-series forecasting task.

The LSTM model, on the other hand, is explicitly designed for time-series data. LSTMs have an edge in capturing patterns over time and remembering long sequences due to their 'gated' structure. Their ability to understand and remember long-term dependencies is vital in our task as patterns in temperature anomalies can persist over time. LSTM models also have the flexibility to model non-linear relationships, which can be crucial for complex datasets like ours. The LSTM model had the highest R-squared value of 0.568 among the three, indicating a better fit to the data and a more reliable prediction of future values.

In conclusion, while each model has its strengths and can be effective in certain scenarios, it's clear that for this problem – predicting temperature anomalies three months ahead – the LSTM model has a definitive edge. The nature of our data (time series) and the complexity of our task (forecasting future values) makes LSTM the most suitable and reliable choice among the models evaluated. It is critical to consider the nature of the problem and the type of data while selecting an appropriate predictive model. For tasks involving time-series data and forecasting future values, LSTM models have proven to be an effective solution.

8. Conclusion

In the process of comparing models for forecasting temperature anomalies, a significant learning experience was achieved. Through testing and comparing the performances of Linear Regression, Random Forest, and LSTM models, it was evident that each model had its strengths and weaknesses.

Linear Regression, a straightforward and interpretable model, yielded the most reliable results with this dataset. It seemed to capture the trend well, producing results that indicated a reasonable understanding of the underlying patterns in the data, as seen from its R-squared value and relatively low errors.

In contrast, the Random Forest model performed poorly in this task. One possible explanation is that the model failed to grasp the time-dependent nature of the data, especially given that the data was not made stationary before feeding it into the model. Also, the Random Forest algorithm may not be the most suitable for this problem due to its inherent design. While Random Forests can capture complex patterns, they might be less effective for time-series data where temporal dependencies play a crucial role [Breiman, L. (2001)].

The LSTM model, on the other hand, delivered mixed results. LSTMs are particularly well-suited for time-series prediction due to their ability to remember long-term dependencies. However, the LSTM model's performance was not as impressive as the Linear Regression models in this case, possibly due to the relatively small size of the dataset. LSTM's capabilities often shine when trained on larger datasets, which allows them to learn more intricate dependencies.

In conclusion, while all three models have proven to be effective in different tasks and domains, their performance can greatly vary depending on the specifics of the problem and data at hand. For this task of forecasting temperature anomalies, Linear Regression proved to be the most effective. However, given a larger dataset, LSTM might have been the superior choice due to its design advantages for time-series data. Therefore, it's always crucial to consider the nature of the data and the problem when selecting a model.

9. References

https://data.giss.nasa.gov/gistemp/

https://data.world/kcmillersean/global-temperature-anomalies

https://data.world/data-society/global-climate-change-data#

https://www.kaggle.com/code/pumalin/ghg-emission-data/data

https://www.kaggle.com/datasets/dataenergy/natural-disaster-data

https://chat.openai.com/chat

Pandas (2023) Pandas Documentation. Available at: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.resample.html (Accessed: May 04, 2023).

Stack Overflow. (n.d.). How to use pandas fillna method in Python. Available at: https://stackoverflow.com/questions/27905295/how-to-replace-nans-by-preceding-values-in-pandas-dataframe (Accessed: May 04, 2023).

Pandas (2023) Pandas Documentation. Available at: https://pandas.pydata.org/pandas-docs/stable/user_guide/missing_data.html (Accessed: May 04, 2023).

Zach (2021) How to Use Q-Q Plots to Check Normality. Available at: https://www.statology.org/q-q-plot-normality (Accessed: May 04, 2023).

Stackoverflow (2013) How to Use Q-Q Plots to Check Normality. Available at: https://stackoverflow.com/questions/12058390/stl-decomposition-of-time-series-with-missing-values-for-anomaly-detection (Accessed: May 05, 2023).

Zhang (2018) A Short Guide for Feature Engineering and Feature Selection. Available at: https://github.com/ashishpatel26/Amazing-Feature Engineering/blob/master/A%20Short%20Guide%20for%20Feature%20Engineering%20and%20Feature%20Selection.md (Accessed: May 05, 2023).

Galli (2022) Feature selection in machine learning with Python. Available at: https://www.blog.trainindata.com/feature-selection-machine-learning-with-python/ (Accessed: May 05, 2023).

Brownlee (2017) A Gentle Introduction to Autocorrelation and Partial Autocorrelation. Available at: https://machinelearningmastery.com/gentle-introduction-autocorrelation-partial-autocorrelation/ (Accessed: May 23, 2023).

Google (2023) Normalization. Available at: https://developers.google.com/machine-learning/data-prep/transform/normalization (Accessed: May 23, 2023).

Breiman, L. (2001) 'Random Forests', Machine Learning, 45, pp. 5-32. Available at: https://www.stat.berkeley.edu/~breiman/randomforest2001.pdf (Accessed: 23 May 25, 2023).