TIME SERIES FORECASTING WITH MACHINE LEARNING

Ashley Honeycutt

May 2023

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

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.

Ashley Honeycutt

word count: 13,000

Abstract

In this project, I present my methods for using machine learning to predict temperature anomaly values three months in the future. I use the climate_dataset given to me for this task. I implement a standard data science workflow and explain my reasoning and methods at every step of the process. First, I explore and clean the data and select the most relevant features for input. Next, I apply the data to four different machine learning models: Multiple Linear Regression, Random Forest Regression, LSTM, and XGBoost. After each model I use standard evaluation metrics including mean absolute error, mean squared error, root mean squared error, and r squared to assess the performance of the models. The results of my modeling show multiple linear regression and LSTM performing the best, followed by XGBoost and lastly Random Forest Regression. In my conclusions I give a variety of reasons why this may be the case and provide recommendations for further improvements.

Table of Contents

1 Introduction

In this project, we are tasked with building a machine learning model to accurately predict a future value from a time series dataset.

A time series is recognizable as such because its datapoints are temporally ordered. In other words, the datapoints were collected at regular intervals and are presented in chronological order. Time series data present unique challenges when applying machine learning models. https://learn.microsoft.com/en-us/azure/architecture/data-guide/scenarios/time-series. Special care must be taken in the data preparation step not to disturb the temporal order of the data. Consequently, the first portion of this project is dedicated to exploring and cleaning the dataset I was provided.

The second portion of this project consists of building machine learning models, fitting the data to the models, evaluating the results, and tuning the models. I use Python machine learning libraries including scikit-learn, keras, and XGboost to construct the models. These are open-source libraries that streamline the machine learning process, as they contain pre-coded algorithms. These algorithms underscore the concept of machine learning, which is a subset of artificial intelligence. The algorithms accept our data as input, apply a variety of mathematical and statistical functions, and produce output that can be evaluated in a variety of ways. In a broad sense, the task of machine learning algorithms is to train the computer to recognize patterns in data. After training, when the computer is presented with new test data, a good machine learning model will be able to recognize patterns in the data and produce the desired output. https://reason.town/importanceof-statistics-in-machine-learning/. In this project, the machine learning models will take in certain variables from the dataset as input and produce a forecasted value as output.

Our dataset contains a time series of monthly weather-related data spanning from 1901 until 2020. The variables we are given include:

- 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

Task: Using this data, our goal is to construct a machine learning model that, given a sample of data, can accurately predict the anomaly value 3 months in the future. I expect that some of the other variables within the dataset can help us predict the anomaly value, so we will use mainly multivariate models to make our predictions.

To achieve the goal, I will implement a standard data science workflow taught in Harvard University's introductory Data Science course. https://github.com/cs109/2015/blob/master/Lectures/01-Introduction.pdf

• Step 1: Ask a Question

The question was prescribed for this project: Given a sample of data, predict the anomaly value 3 months in the future.

• Step 2: Get the Data

The data was given to us in the form of climate_dataset.csv. It was gathered from a variety of open sources including NASA and Kaggle.

• Step 3: Explore and Prepare the Data

In this step, I try to gain a deep understanding of the dataset by using commands from the Python Pandas library and plotting and visualizing the data. Additionally I search for missing values and determine the best

way to handle them. Each model requires unique preparation so I will further prepare the data before each model.

- Step 4: Model the Data

  I apply four types of models to the data including: Multiple Linear Regression, Random Forest Regressor, LSTM, and XGBoost. Each model possesses different advantages and drawbacks, and is implemented with varying degrees of success. I elaborate on the conceptual background of each model in the modelling section and describe and motivate each implementation.

- Step 5: Communicate and Visualize the Results

  For each model, I evaluate the predictions using standard metrics including mean absolute error, mean squared error, root mean squared error, and R2. I plot the predictions and the actual anomaly values to visualize the accuracy of the model. Finally, I compare and contrast the results of each model and reflect on ways in which the results could be improved.

2 Data Exploration and Preparation

In this section I describe the steps I took to prepare the data for modeling and the reasons motivating my decisions. Also included in this section is feature selection. I will describe the steps I took and the rationale behind the selection of features for the models.

Since our dataset is a time series, there are a few important concerns during data preparation. In the case of missing or null values, simply removing the entire row is often not appropriate as it disturbs the temporal order. Additionally, each datapoint in the time series cannot be considered independent of the other datapoints,because the data is sequential and each datapoint depends on previous ones. Across the time series there may be special trends such as seasonality or cycles that require special handling before inputting to a machine learning model. I keep all of these concerns in mind as I explore and prepare the data for modeling.
https://learn.microsoft.com/enus/azure/architecture/data-guide/scenarios/time-series

The goals of this section are: - to understand each variable in the dataset - what each variable represents - the patterns of each variable over time - how each variable correlates with the target variable - to handle datatypes (most machine learning models require numeric input) - to handle null values in a way that is appropriate to the particular variable and to time series data - either by imputation or removing rows where appropriate - to select the features most useful for predicting future anomaly values - I select them on the basis of the amount and quality of information they provide, their relationship with the target variable and each other

Import Libraries

[6523]:
```
import numpy as np import pandas as pd import seaborn as sns import
matplotlib.pyplot as plt from sklearn.model_selection import
train_test_split from sklearn.linear_model import LinearRegression from
sklearn.ensemble import RandomForestRegressor from sklearn.metrics import
mean_squared_error, mean_absolute_error, r2_score from
sklearn.preprocessing import MinMaxScaler from math import sqrt from
statsmodels.tsa.stattools import adfuller import tensorflow as tf from
tensorflow.keras.layers import LSTM, Dense from tensorflow.keras.models
import Sequential import os from tensorflow.keras.optimizers import Adam
import xgboost as xgb
color_pal =
sns.color_palette()
```

Read in dataset

The dataset was given to us in the form of a comma separated values (csv) file. I use pandas to read the csv file into a dataframe. Pandas is a Python library for data analysis and manipulation. Pandas dataframes possess a variety of parameters, attributes, and methods which make it easy to explore, understand, and make changes to the data. https://pandas.pydata.org/pandasdocs/stable/reference/api/pandas.DataFrame.html

[6524]:
```
df = pd.read_csv( 'climate_dataset.csv  ')
```

Display dataframe

This initial view of the dataframe gives us some important information. Firstly, our data was collected on a monthly basis from 1901 to 2020. We have 1,458 datapoints and 23 variables. Finally, there appear to be quite a few null values in the dataset that we need to handle.

[6525]:
```
df
```

[6525]:

	date	emission	anomaly_value	upper_95_ci	lower_95_ci
0	1/31/1901	4.482107e+10	-0.075	-0.161	-0.008
1	2/28/1901	4.482107e+10	-0.176	-0.259	-0.110
2	3/31/1901	4.482107e+10	-0.272	-0.358	-0.210
3	4/30/1901	4.482107e+10	-0.236	-0.317	-0.164
4	5/31/1901	4.482107e+10	-0.187	-0.259	-0.122
…	…	…	…	…	…
1453	9/30/2019	NaN	0.719	0.662	0.761
1454	10/31/2019	NaN	0.713	0.674	0.754
1455	11/30/2019	NaN	0.752	0.719	0.796
1456	12/31/2019	NaN	0.693	0.654	0.732
1457	1/31/2020	NaN	0.879	0.823	0.921

	AverageTemperature	AverageTemperatureUncertainty	City	Country	Latitude	…	Earthquake	Extreme temperature	Extreme weather	Flood
0	-4.087	0.422	Berlin	Germany						
1	-3.598	0.951	Berlin	Germany						
2	2.945	0.354	Berlin	Germany						
3	8.711	0.471	Berlin	Germany						
4	14.182	0.380	Berlin	Germany						
…	…	…	…	…						
1453	NaN	NaN	NaN	NaN						
1454	NaN	NaN	NaN	NaN						
1455	NaN	NaN	NaN	NaN						
1456	NaN	NaN	NaN	NaN						
1457	NaN	NaN	NaN	NaN						

```
0      52.24N …         NaN                  NaN              1.0    1.0
1      52.24N …         NaN                  NaN              1.0    1.0
2      52.24N …         NaN                  NaN              1.0    1.0
3      52.24N …         NaN                  NaN              1.0    1.0
4      52.24N …         NaN                  NaN              1.0    1.0
…       … …          …                    …                …     …
1453      NaN …         NaN                  NaN              NaN    NaN
1454      NaN …         NaN                  NaN              NaN    NaN
1455      NaN …         NaN                  NaN              NaN    NaN
1456      NaN …         NaN                  NaN              NaN    NaN
1457      NaN …         NaN                  NaN              NaN    NaN

        Impact Landslide Mass movement (dry) Volcanic activity Wildfire \
0        NaN        NaN                  NaN               1.0       NaN
1        NaN        NaN                  NaN               1.0       NaN
2        NaN        NaN                  NaN               1.0       NaN
3        NaN        NaN                  NaN               1.0       NaN
4        NaN        NaN                  NaN               1.0       NaN
…      …         …                    …                 …        …
1453     NaN        NaN                  NaN               NaN       NaN
1454     NaN        NaN                  NaN               NaN       NaN
1455     NaN        NaN                  NaN               NaN       NaN
1456     NaN        NaN                  NaN               NaN       NaN
1457     NaN        NaN                  NaN               NaN       NaN

      global_avg_temp
0               -0.28
1               -0.06
2               0.04 3  -0.06
 4               -0.17
…                …
1453              NaN
1454              NaN
1455              NaN
1456              NaN
1457              NaN
```

```
1458            rows x 23 columns]
```

Check datatypes

Most machine learning models require numeric data as input. Therefore, a good starting place is to check the datatype of each variable in the dataset. We can see that most variables are float64, which is a numeric type. There are a few object types, which I will explore further before deciding how to handle them.

[6526]:
```
df.dtypes
```

[6526]:
```
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
```

Change datatype of date column from object to datetime

First, we will handle the datatype of the date column. For time series analysis, it is important for the date to be in datetime format, so I converted it from object to datetime.
https://stackoverflow.com/questions/53669688/convert-date-column-from-object-data-typeto-date-time-data-type

[6527]:
```
df['date']=pd.to_datetime(df['date'], format='%m/%d/%Y')
```

Initial Visualization

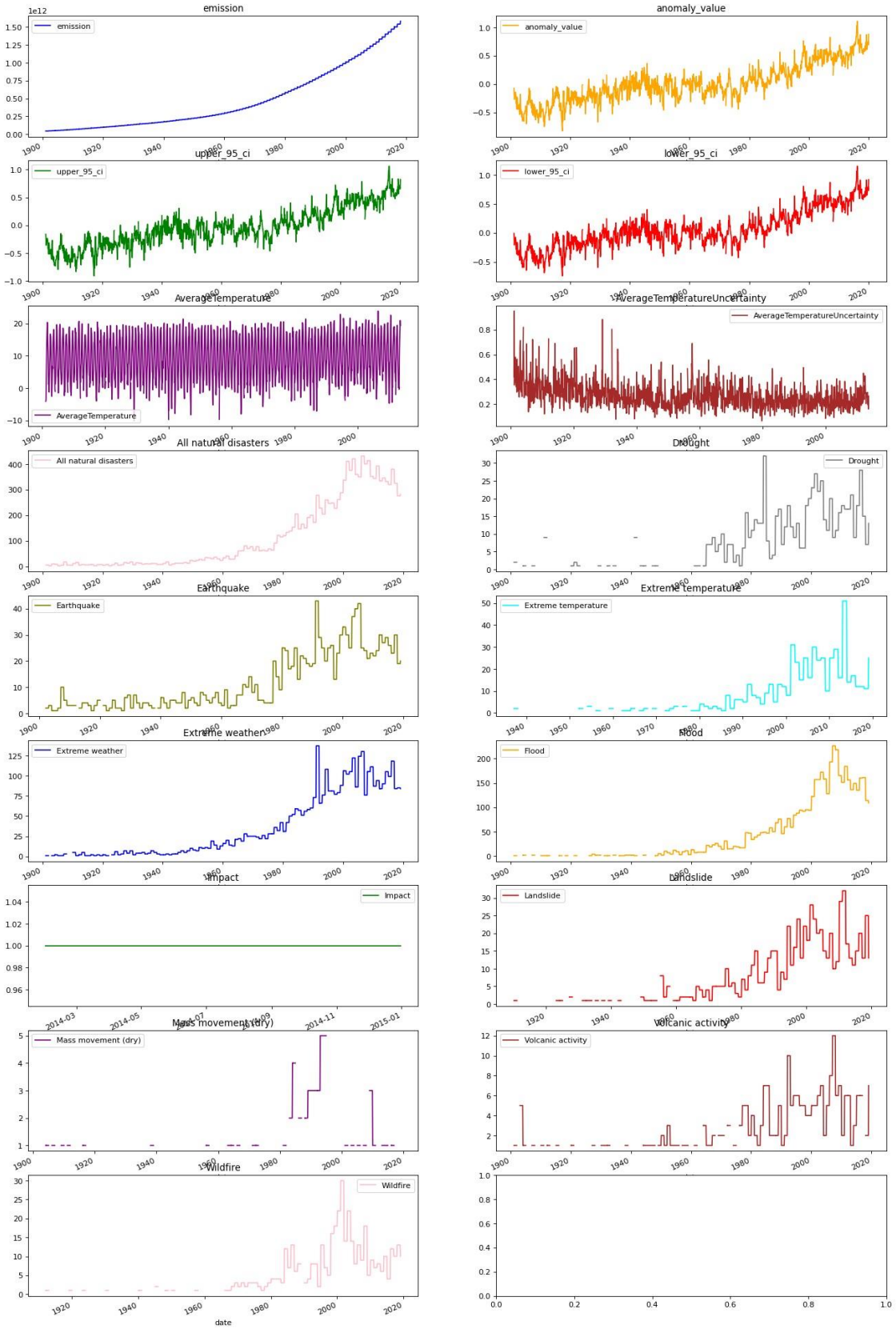
The next step of data exploration is to plot the data and look for trends. The code below creates a function 'Visualize' which builds a list of numeric features and contains a loop which iterates over the list creating a grid of subplots. Each resulting subplot displays a single variable plotted over time.

I observe the following by viewing the plots:

- **emission:** Shows a steady increase over time, as I would logically expect given global development over the past century
- **anomaly_value, upper_95_ci, lower_95_ci:** These appear to follow the same trend, which makes sense because the upper and lower 95 CI variables are measurements of anomaly value.
- **AverageTemperature, AverageTemperatureUncertainty:** Average temperature appears as I would expect it to. By general knowledge, I know that average temperatures vary by month over the course of a year, but stay relatively similar when comparing the same month across years. Average temperature uncertainty has decreased over time, which is in line with improved technology and more accurate temperature measurements.
- **All natural disasters, Drought, Earthquake, Extreme temperature, Extreme weather, Flood, Impact, Landslide, Mass movement (dry), Volcanic activity, Wildfire:** Most appear to follow a similar pattern, and contain quite a few missing values.
-

[6528]:
```
colors = ['blue','orange','green','red','purple','brown','pink','gray','olive','cyan']

df = df.set_index('date')  #set the index to date so it displays on the X axis

def Visualize(data):
    features = list(df.select_dtypes(include=[np.number]).columns.values)
    #selects only the numeric columns
    feature_size =len(features)
    fig, axes = plt.subplots(
        nrows=int(np.ceil(feature_size/2)), ncols=2,
        figsize=(20, feature_size*2),dpi=80,facecolor='w', edgecolor='k'
    )
    for i in range(feature_size):
        key=features[i]
        c=colors[i % (len(colors))]
        t_data = data[key]
        t_data.head()
        ax=t_data.plot(
            ax=axes[i//2, i%2],
            color=c,
            title='{}'.format(key),
            rot=25
        )
        ax.legend([key])
plt.tight_layout()

Visualize(df)

#code adapted from: https://www.youtube.com/watch?v=TpQtD7ONfxQ
```

```
<Figure size 640x480 with 0 Axes>
```


emission
1e12
anomaly_value
upper_95_ci
lower_95_ci
AverageTemperature
AverageTemperatureUncertainty
All natural disasters
Drought
Earthquake
Extreme temperature
Extreme weather
Flood
Impact
Landslide
Mass movement (dry)
Volcanic activity
Wildfire
date

Check for null values in each column

Below is the sum of null values in each column.

[6529]:
```
df.isna() .sum()
```

[6529]:
```
emission  53 anomaly_value  29
upper_95_ci     29
lower_95_ci                        29
AverageTemperature                106
AverageTemperatureUncertainty     106
City                              105
Country                           105
Latitude                          105
Longitude                         105
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     66
dtype: int64
```

Check for unique values in City column

We can see in the output that our data only applies to Berlin, Germany or 'nan'.

[6530]:
```
df.City .unique()
```

[6530]:
```
array(['Berlin', nan], dtype=object)
```

Drop location-related columns

We can drop City, Country, Latitude, and Longitude columns as these are all the same value or null. The models only accept numeric data, and in the case of categorical variables like city and country, we would need to numerically encode the values with a representative number, such as 1 for Berlin. Because they are all the same (or null), these columns are of no use in predicting future anomaly values.

[6531]:
```
df = df.drop(['City','Country','Latitude','Longitude'], axis=1)
```

Handling All Natural Disasters

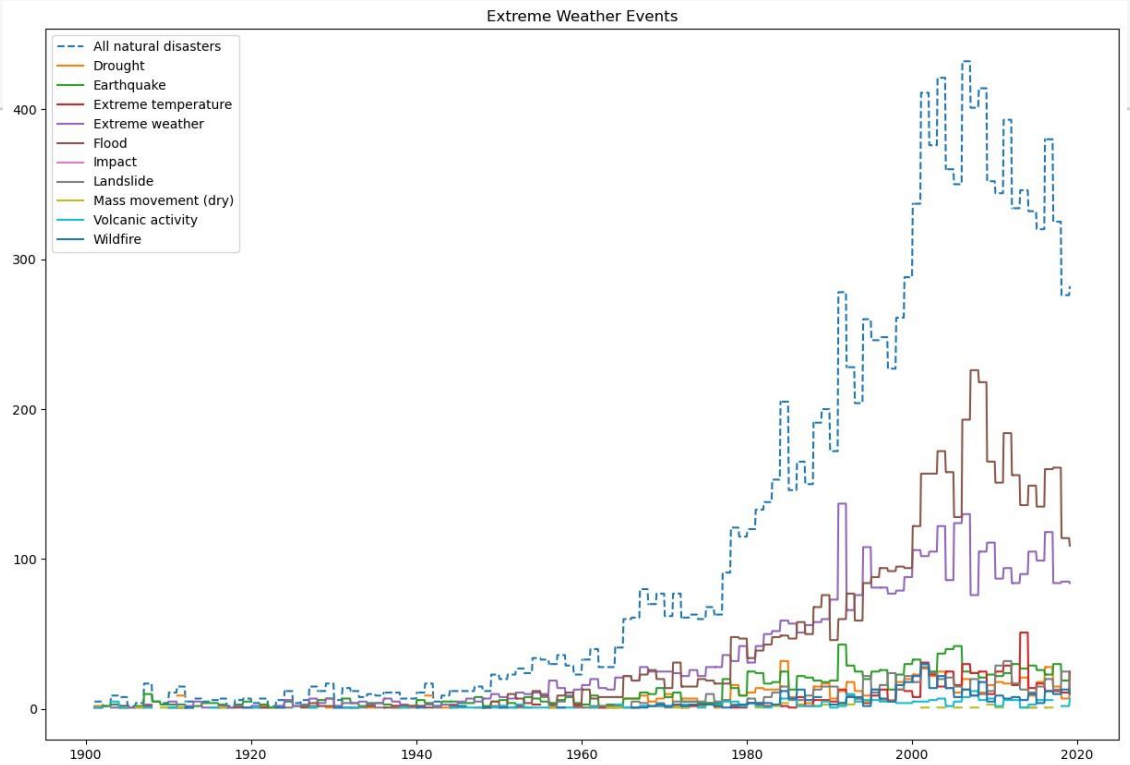
The columns: Drought, Earthquake, Extreme Temperature, Extreme Weather, Flood, Impact, Landslide, Mass movement (dry), Volcanic activity, and Wildfire contain counts of the respective events as values. The null values in those columns actually indicate zero events. Viewing our dataframe, we can see that the All Natural Disasters column contains the sum of events from each of those columns.

The graph below shows each of the weather event columns plotted together with All natural disasters represented by a dashed line. Visually, it appears that many of these columns follow a similar pattern and might be correlated.

For simplicity and to avoid multicollinearity, we will remove these columns and keep only All Natural Disasters.

Multicollinearity occurs when several input features are highly correlated which negatively impacts the regression results. The input features should all be independent variables while our target variable (anomaly value) is dependent. If the input features are dependent on each other, then this can make the regression coefficients less precise and harder to interpret. https://statisticsbyjim.com/regression/multicollinearity-in-regression-analysis/

[6532]:
```
import matplotlib.pyplot as plt

#plot each series
fig, ax = plt.subplots(figsize=(15, 10)) plt.plot(df['All natural
disasters'], label = 'All natural disasters', ls =␣
 ↪'--' ) plt.plot(df['Drought'], label = 'Drought')
plt.plot(df['Earthquake'], label = 'Earthquake')
plt.plot(df['Extreme temperature'], label = 'Extreme
temperature') plt.plot(df['Extreme weather'], label = 'Extreme
weather') plt.plot(df['Flood'], label = 'Flood')
plt.plot(df['Impact'], label = 'Impact')
plt.plot(df['Landslide'], label = 'Landslide')
plt.plot(df['Mass movement (dry)'], label = 'Mass movement
(dry)') plt.plot(df['Volcanic activity'], label = 'Volcanic
activity') plt.plot(df['Wildfire'], label = 'Wildfire')

#display plot
plt.title('Extreme Weather
Events') plt.legend() plt.show()
```



[6533]:
```
df = df.drop(['Drought', 'Earthquake', 'Extreme temperature', 'Extreme weather',
              'Flood', 'Impact', 'Landslide', 'Mass movement (dry)',
              'Volcanic activity', 'Wildfire'], axis=1)
```

Re-checking for null values

[6534]:
```
df.isna() .sum()
```

[6534]:
```
emission                        53
anomaly_value                   29
upper_95_ci                     29
lower_95_ci                     29
AverageTemperature             106
AverageTemperatureUncertainty  106
All natural disasters           41
global_avg_temp                 66
dtype: int64
```

Replacing null values with zeros

Since the All natural disasters column contains numeric values showing a count of the total number of disasters, I replace the nulls with 0's.

[6535]:
```
df['All natural disasters'] = df['All natural disasters'].fillna(0)
```

Viewing null values within the dataframe

Starting with emission, the first variable in the list above, I create a new dataframe to display all null values in the column. In the resulting dataframe, we can see that emission, anomaly_value, upper_95_ci, lower_95_ci, AverageTemperature, AverageTemperatureUncertainty, and global_avg_temp are missing 28-February data every four years from 1901 until 2016.

[6536]:
```
df_emission  = df[df[ 'emission '].isna()]
df_emission
```

[6536]:

date	emission	anomaly_value	upper_95_ci	lower_95_ci
1904-02-28	NaN	NaN	NaN	NaN
1908-02-28	NaN	NaN	NaN	NaN
1912-02-28	NaN	NaN	NaN	NaN
1916-02-28	NaN	NaN	NaN	NaN
1920-02-28	NaN	NaN	NaN	NaN
1924-02-28	NaN	NaN	NaN	NaN
1928-02-28	NaN	NaN	NaN	NaN

1932-02-28	NaN	NaN	NaN	NaN
1936-02-28	NaN	NaN	NaN	NaN
1940-02-28	NaN	NaN	NaN	NaN
1944-02-28	NaN	NaN	NaN	NaN
1948-02-28	NaN	NaN	NaN	NaN
1952-02-28	NaN	NaN	NaN	NaN
1956-02-28	NaN	NaN	NaN	NaN
1960-02-28	NaN	NaN	NaN	NaN
1964-02-28	NaN	NaN	NaN	NaN
1968-02-28	NaN	NaN	NaN	NaN
1972-02-28	NaN	NaN	NaN	NaN
1976-02-28	NaN	NaN	NaN	NaN
1980-02-28	NaN	NaN	NaN	NaN
1984-02-28	NaN	NaN	NaN	NaN
1988-02-28	NaN	NaN	NaN	NaN
1992-02-28	NaN	NaN	NaN	NaN
1996-02-28	NaN	NaN	NaN	NaN
2000-02-28	NaN	NaN	NaN	NaN
2004-02-28	NaN	NaN	NaN	NaN
2008-02-28	NaN	NaN	NaN	NaN
2012-02-28	NaN	NaN	NaN	NaN
2016-02-28	NaN	NaN	NaN	NaN
2018-02-28	NaN	0.554	0.507	0.603
2018-03-31	NaN	0.528	0.477	0.578
2018-04-30	NaN	0.615	0.575	0.655
2018-05-31	NaN	0.627	0.586	0.666
2018-06-30	NaN	0.587	0.544	0.626
2018-07-31	NaN	0.573	0.536	0.619
2018-08-31	NaN	0.594	0.553	0.632
2018-09-30	NaN	0.586	0.530	0.625
2018-10-31	NaN	0.598	0.563	0.642
2018-11-30	NaN	0.678	0.640	0.721
2018-12-31	NaN	0.590	0.549	0.630
2019-01-31	NaN	0.638	0.583	0.682
2019-02-28	NaN	0.738	0.690	0.785
2019-03-31	NaN	0.662	0.610	0.713
2019-04-30	NaN	0.874	0.833	0.915
2019-05-31	NaN	0.780	0.738	0.823
2019-06-30	NaN	0.610	0.566	0.652
2019-07-31	NaN	0.708	0.666	0.753
2019-08-31	NaN	0.706	0.662	0.743
2019-09-30	NaN	0.719	0.662	0.761
2019-10-31	NaN	0.713	0.674	0.754
2019-11-30	NaN	0.752	0.719	0.796
2019-12-31	NaN	0.693	0.654	0.732
2020-01-31	NaN	0.879	0.823	0.921

date	AverageTemperature	AverageTemperatureUncertainty	\

1904-02-28	NaN	NaN
1908-02-28	NaN	NaN
1912-02-28	NaN	NaN
1916-02-28	NaN	NaN
1920-02-28	NaN	NaN
1924-02-28	NaN	NaN
1928-02-28	NaN	NaN
1932-02-28	NaN	NaN
1936-02-28	NaN	NaN
1940-02-28	NaN	NaN
1944-02-28	NaN	NaN
1948-02-28	NaN	NaN
1952-02-28	NaN	NaN
1956-02-28	NaN	NaN
1960-02-28	NaN	NaN
1964-02-28	NaN	NaN
1968-02-28	NaN	NaN
1972-02-28	NaN	NaN
1976-02-28	NaN	NaN
1980-02-28	NaN	NaN
1984-02-28	NaN	NaN
1988-02-28	NaN	NaN
1992-02-28	NaN	NaN
1996-02-28	NaN	NaN
2000-02-28	NaN	NaN
2004-02-28	NaN	NaN
2008-02-28	NaN	NaN
2012-02-28	NaN	NaN
2016-02-28	NaN	NaN
2018-02-28	NaN	NaN
2018-03-31	NaN	NaN
2018-04-30	NaN	NaN
2018-05-31	NaN	NaN
2018-06-30	NaN	NaN
2018-07-31	NaN	NaN
2018-08-31	NaN	NaN
2018-09-30	NaN	NaN
2018-10-31	NaN	NaN
2018-11-30	NaN	NaN
2018-12-31	NaN	NaN
2019-01-31	NaN	NaN
2019-02-28	NaN	NaN
2019-03-31	NaN	NaN
2019-04-30	NaN	NaN
2019-05-31	NaN	NaN
2019-06-30	NaN	NaN
2019-07-31	NaN	NaN
2019-08-31	NaN	NaN

2019-09-30	NaN	NaN
2019-10-31	NaN	NaN
2019-11-30	NaN	NaN
2019-12-31	NaN	NaN
2020-01-31	NaN	NaN

date	All natural disasters	global_avg_temp
1904-02-28	8.0	NaN
1908-02-28	5.0	NaN
1912-02-28	5.0	NaN
1916-02-28	7.0	NaN
1920-02-28	4.0	NaN
1924-02-28	12.0	NaN
1928-02-28	12.0	NaN
1932-02-28	12.0	NaN
1936-02-28	11.0	NaN
1940-02-28	11.0	NaN
1944-02-28	12.0	NaN
1948-02-28	12.0	NaN
1952-02-28	27.0	NaN
1956-02-28	30.0	NaN
1960-02-28	33.0	NaN
1964-02-28	41.0	NaN
1968-02-28	70.0	NaN
1972-02-28	61.0	NaN
1976-02-28	63.0	NaN
1980-02-28	120.0	NaN
1984-02-28	205.0	NaN
1988-02-28	191.0	NaN
1992-02-28	228.0	NaN
1996-02-28	248.0	NaN
2000-02-28	337.0	NaN
2004-02-28	360.0	NaN
2008-02-28	414.0	NaN
2012-02-28	334.0	NaN
2016-02-28	380.0	NaN

```
2018-02-28                276.0                          NaN
2018-03-31                276.0                          NaN
2018-04-30                276.0                          NaN
2018-05-31                276.0                          NaN
2018-06-30                276.0                          NaN
2018-07-31                276.0                          NaN
2018-08-31                276.0                          NaN
2018-09-30                276.0                          NaN
2018-10-31                276.0                          NaN
2018-11-30                276.0                          NaN
2018-12-31                276.0                          NaN
2019-01-31                282.0                          NaN
2019-02-28                  0.0                          NaN
2019-03-31                  0.0                          NaN
2019-04-30                  0.0                          NaN
2019-05-31                  0.0                          NaN
2019-06-30                  0.0                          NaN
2019-07-31                  0.0                          NaN
2019-08-31                  0.0                          NaN
2019-09-30                  0.0                          NaN
2019-10-31                  0.0                          NaN
2019-11-30                  0.0                          NaN
2019-12-31                  0.0                          NaN
2020-01-31                  0.0                          NaN
```

Handling the missing February data

I suspect this issue might be due to a leap year sampling error. To explore further, I first create a month-year column from the date column so I can check for duplicates.

[6537]:
```
df = df.reset_index()
df['month_year '] = df['date '].dt.strftime( '%m/%Y')
df = df.set_index( 'date ')
df.head()
```

[6537]:
```
                emission anomaly_value upper_95_ci lower_95_ci \
date
1901-01-31 4.482107e+10        -0.075      -0.161      -
                                                       0.008
1901-02-28 4.482107e+10        -0.176      -0.259      -
                                                       0.110
1901-03-31 4.482107e+10        -0.272      -0.358      -
                                                       0.210
1901-04-30 4.482107e+10        -0.236      -0.317      -
                                                       0.164
1901-05-31 4.482107e+10        -0.187      -0.259      -
                                                       0.122
            AverageTemperature AverageTemperatureUncertainty \
```

```
date
1901-01-31             -4.087                          0.422
1901-02-28             -3.598                          0.951
1901-03-31              2.945                          0.354
1901-04-30              8.711                          0.471
1901-05-31             14.182                          0.380
           All natural disasters global_avg_temp month_year
date
1901-01-31                   5.0          -0.28    01/1901
1901-02-28                   5.0          -0.06    02/1901
1901-03-31                   5.0           0.04    03/1901
1901-04-30                   5.0          -0.06    04/1901
1901-05-31                   5.0          -0.17    05/1901
```

Checking for month-year duplicates, we can see that there is an error in the February data every four years. It appears that on each leap year, data was pulled for both February 28th and February 29th. Since February 29th is the correct data for those years, we will remove the February 28th rows.

[6538]:
```
dfduplicates  = df[df.duplicated([ 'month_year '], keep =False)]
dfduplicates
```

[6538]:
```
                emission anomaly_value upper_95_ci lower_95_ci \
date
1904-02-28           NaN           NaN          NaN          NaN
1904-02-29 5.116540e+10        -0.644       -0.716       -
                                                        0.586
1908-02-28           NaN           NaN          NaN          NaN
1908-02-29 6.131082e+10        -0.412       -0.486       -
                                                        0.354
1912-02-28           NaN           NaN          NaN          NaN
1912-02-29 7.307907e+10        -0.361       -0.432       -
                                                        0.295
1916-02-28           NaN           NaN          NaN          NaN
1916-02-29 8.609039e+10        -0.235       -0.308       -
                                                        0.164
1920-02-28           NaN           NaN          NaN          NaN
1920-02-29 9.948842e+10        -0.239       -0.321       -
                                                        0.161
1924-02-28           NaN           NaN          NaN          NaN
1924-02-29 1.130000e+11        -0.315       -0.395       -
                                                        0.240
1928-02-28           NaN           NaN          NaN          NaN
1928-02-29 1.280000e+11        -0.054       -0.131        0.022
1932-02-28           NaN           NaN          NaN          NaN
1932-02-29 1.440000e+11         0.157        0.062        0.244
1936-02-28           NaN           NaN          NaN          NaN
1936-02-29 1.580000e+11        -0.310       -0.397       -
                                                        0.238
1940-02-28           NaN           NaN          NaN          NaN
```

```
1940-02-29 1.750000e+11        -0.192      -0.246      -
                                                       0.133
1944-02-28          NaN           NaN         NaN         NaN
1944-02-29 1.960000e+11         0.281       0.229       0.328
1948-02-28          NaN           NaN         NaN         NaN
1948-02-29 2.150000e+11         0.089       0.037       0.153
1952-02-28          NaN           NaN         NaN         NaN
1952-02-29 2.380000e+11         0.184       0.146       0.226
1956-02-28          NaN           NaN         NaN         NaN
1956-02-29 2.640000e+11        -0.247      -0.280      -
                                                       0.216
1960-02-28          NaN           NaN         NaN         NaN
1960-02-29 2.970000e+11        -0.032      -0.060       0.003
1964-02-28          NaN           NaN         NaN         NaN
1964-02-29 3.360000e+11        -0.053      -0.083      -
                                                       0.021
1968-02-28          NaN           NaN         NaN         NaN
1968-02-29 3.830000e+11        -0.241      -0.263      -
                                                       0.224
1972-02-28          NaN           NaN         NaN         NaN
1972-02-29 4.400000e+11        -0.379      -0.398      -
                                                       0.357
1976-02-28          NaN           NaN         NaN         NaN
1976-02-29 5.070000e+11        -0.233      -0.247      -
                                                       0.220
1980-02-28          NaN           NaN         NaN         NaN
1980-02-29 5.810000e+11         0.134       0.115       0.149
1984-02-28          NaN           NaN         NaN         NaN
1984-02-29 6.570000e+11         0.118       0.097       0.139
1988-02-28          NaN           NaN         NaN         NaN
1988-02-29 7.370000e+11         0.388       0.358       0.418
1992-02-28          NaN           NaN         NaN         NaN
1992-02-29 8.260000e+11         0.367       0.328       0.410
1996-02-28          NaN           NaN         NaN         NaN
1996-02-29 9.150000e+11         0.116       0.077       0.160
2000-02-28          NaN           NaN         NaN         NaN
2000-02-29 1.010000e+12         0.224       0.184       0.268
2004-02-28          NaN           NaN         NaN         NaN
2004-02-29 1.110000e+12         0.507       0.463       0.551
2008-02-28          NaN           NaN         NaN         NaN
2008-02-29 1.230000e+12         0.158       0.113       0.202
2012-02-28          NaN           NaN         NaN         NaN
2012-02-29 1.360000e+12         0.310       0.263       0.354
2016-02-28          NaN           NaN         NaN         NaN
2016-02-29 1.500000e+12         0.934       0.889       0.981
           AverageTemperature AverageTemperatureUncertainty \
date
1904-02-28                NaN                           NaN
1904-02-29              1.365                         0.624
```

```
 1908-02-28                NaN                            NaN
1908-02-29 1.962 0.472 1912-02-28 NaN NaN
1912-02-29 1.856 0.361 1916-02-28 NaN NaN
1916-02-29 0.906 0.560 1920-02-28 NaN NaN
1920-02-29 3.435 0.355 1924-02-28 NaN NaN
1924-02-29 -2.872 0.364 1928-02-28 NaN NaN
1928-02-29 2.601 0.316 1932-02-28 NaN NaN
1932-02-29 -1.595 0.803 1936-02-28 NaN NaN
1936-02-29 0.125 0.208 1940-02-28 NaN NaN
1940-02-29 -6.861 0.376 1944-02-28 NaN NaN
1944-02-29 0.067 0.312 1948-02-28 NaN NaN
1948-02-29 0.263 0.235 1952-02-28 NaN NaN
1952-02-29 1.055 0.256 1956-02-28 NaN NaN
1956-02-29 -9.646 0.486 1960-02-28 NaN NaN
1960-02-29 -0.376 0.211 1964-02-28 NaN NaN
1964-02-29 0.032 0.271 1968-02-28 NaN NaN
1968-02-29 0.982 0.214 1972-02-28 NaN NaN
1972-02-29 2.189 0.303 1976-02-28 NaN NaN
1976-02-29 0.467 0.168 1980-02-28 NaN NaN
1980-02-29 1.772 0.182 1984-02-28 NaN NaN
1984-02-29 0.549 0.220 1988-02-28 NaN NaN
1988-02-29 3.285 0.252 1992-02-28 NaN NaN
1992-02-29 4.122 0.284 1996-02-28 NaN NaN
1996-02-29 -2.421 0.167 2000-02-28 NaN NaN
 2000-02-29              4.718                          0.167
 2004-02-28                NaN                            NaN
 2004-02-29              3.139                          0.149
 2008-02-28                NaN                            NaN
 2008-02-29              5.115                          0.273
 2012-02-28                NaN                            NaN
 2012-02-29             -1.944                          0.265
 2016-02-28                NaN                            NaN
 2016-02-29                NaN                            NaN
           All natural disasters global_avg_temp month_year
date
 1904-02-28                  8.0             NaN    02/1904
 1904-02-29                  0.0           -0.54    02/1904
 1908-02-28                  5.0             NaN    02/1908
 1908-02-29                  0.0           -0.36    02/1908
 1912-02-28                  5.0             NaN    02/1912
 1912-02-29                  0.0           -0.15    02/1912
 1916-02-28                  7.0             NaN    02/1916
 1916-02-29                  0.0           -0.21    02/1916
 1920-02-28                  4.0             NaN    02/1920
 1920-02-29                  0.0           -0.23    02/1920
 1924-02-28                 12.0             NaN    02/1924
 1924-02-29                  0.0           -0.26    02/1924
 1928-02-28                 12.0             NaN    02/1928
 1928-02-29                  0.0           -0.11    02/1928
```

```
1932-02-28                  12.0            NaN    02/1932
1932-02-29                   0.0          -0.17    02/1932
1936-02-28                  11.0            NaN    02/1936
1936-02-29                   0.0          -0.39    02/1936
1940-02-28                  11.0            NaN    02/1940
1940-02-29                   0.0           0.06    02/1940
1944-02-28                  12.0            NaN    02/1944
1944-02-29                   0.0           0.32    02/1944
1948-02-28                  12.0            NaN    02/1948
1948-02-29                   0.0          -0.12    02/1948
1952-02-28                  27.0            NaN    02/1952
1952-02-29                   0.0           0.13    02/1952
1956-02-28                  30.0            NaN    02/1956
1956-02-29                   0.0          -0.24    02/1956
1960-02-28                  33.0            NaN    02/1960
1960-02-29                   0.0           0.13    02/1960
1964-02-28                  41.0            NaN    02/1964
1964-02-29                   0.0          -0.11    02/1964
1968-02-28                  70.0            NaN    02/1968
1968-02-29                   0.0          -0.14    02/1968
1972-02-28                  61.0            NaN    02/1972
1972-02-29 0.0 -0.17 02/1972 1976-02-28 63.0 NaN 02/1976
1976-02-29 0.0 -0.07 02/1976 1980-02-28 120.0 NaN 02/1980
1980-02-29 0.0 0.42 02/1980 1984-02-28 205.0 NaN 02/1984
1984-02-29 0.0 0.18 02/1984 1988-02-28 191.0 NaN 02/1988
1988-02-29 0.0 0.42 02/1988 1992-02-28 228.0 NaN 02/1992
1992-02-29 0.0 0.42 02/1992 1996-02-28 248.0 NaN 02/1996
1996-02-29 0.0 0.49 02/1996 2000-02-28 337.0 NaN 02/2000
2000-02-29 0.0 0.59 02/2000 2004-02-28 360.0 NaN 02/2004
2004-02-29                   0.0            0.7    02/2004
2008-02-28 414.0 NaN 02/2008 2008-02-29 0.0 0.35 02/2008
2012-02-28 334.0 NaN 02/2012 2012-02-29 0.0 0.49 02/2012
2016-02-28                 380.0            NaN    02/2016
2016-02-29                   0.0           1.32    02/2016
```

I use the 'groupby' function to group the data by month-year. Then, I use aggregate (last) to keep only the last row of duplicate pairs, which in this case are February 29th datapoints from 1901-2016.

[6539]:
```
df = df.reset_index( 'date ')
df = df.groupby( 'month_year ').agg('last ').reset_index()
df = df.sort_values(by ='date ')
df= df.set_index( 'date ')
df = df.drop( 'month_year ', axis =1)
df.head()
```

[6539]:
```
                emission anomaly_value upper_95_ci lower_95_ci \
date
1901-01-31 4.482107e+10        -0.075      -0.161       -
                                                        0.008
```

```
1901-02-28 4.482107e+10        -0.176      -0.259      -
                                                       0.110
1901-03-31 4.482107e+10        -0.272      -0.358      -
                                                       0.210
1901-04-30 4.482107e+10        -0.236      -0.317      -
                                                       0.164
1901-05-31 4.482107e+10        -0.187      -0.259      -
                                                       0.122
            AverageTemperature AverageTemperatureUncertainty \
date
1901-01-31             -4.087                         0.422
1901-02-28             -3.598                         0.951
1901-03-31              2.945                         0.354
1901-04-30              8.711                         0.471
1901-05-31             14.182                         0.380
```

```
All natural disasters global_avg_temp
date
1901-01-31                  5.0          -0.28
1901-02-28                  5.0          -0.06
1901-03-31                  5.0           0.04
1901-04-30                  5.0          -0.06
1901-05-31                  5.0          -0.17
```

Below we can see that all null values are taken care of for anomaly value, upper 95 CI, lower 95 CI, and All natural disasters.

[6540]:
```
df.isna() .sum()
```

[6540]:
```
emission                        24
anomaly_value                    0
upper_95_ci                      0
lower_95_ci                      0
AverageTemperature              77
AverageTemperatureUncertainty   77
All natural disasters            0
global_avg_temp                 37
dtype: int64
```

Investigating global_avg_temp

All the remaining variables are of the float64 datatype except global_avg_temp, which is object. To explore why this is the case, I check for unique values in the global_avg_temp column.

The code below produces an array of unique values. Most are numbers, but at the end I can see a '***' and a 'None'. These values are not appropriate to input to the machine learning models. We will explore these further before deciding how to handle them.

[6541]:
```
df['global_avg_temp  '].unique()
```

[6541]:
```
array(['-0.28', '-0.06', '0.04', '-0.17', '-0.1', '-0.09', '-0.13',
       '-0.29', '-0.3', '-0.19', '-0.04', '-0.32', '-0.34', '-0.26',
       '-0.22', '-0.27', '-0.36', '-0.46', '-0.24', '-0.4', '-0.42',
       '-0.45', '-0.44', '-0.43', '-0.39', '-0.47', '-0.64', '-0.54',
       '-0.51', '-0.5', '-0.48', '-0.35', '-0.38', '-0.59', '-0.25',
       '-0.37', '-0.21', '-0.15', '-0.31', '-0.03', '-0.18', '-0.2',
       '-0.52', '-0.33', '-0.58', '-0.41', '-0.49', '-0.7', '-0.57', '-
      0.69', '-0.63', '-0.6', '-0.62', '-0.55', '-0.56', '0.01',
       '-0.14', '-0.23', '-0.05', '-0.01', '-0.11', '0.08', '-0.16',
       '-0.79', '-0.72', '-0.12', '0.03', '0.09', '0.2', '0.07', '0.12',
       '-0.07', '-0.02', '-0.08', '0.14', '0', '0.1', '0.02', '0.05',
       '0.13', '0.06', '0.41', '0.16', '0.11', '0.22', '0.25', '0.15',
       '0.28', '0.31', '0.26', '0.42', '0.32', '0.35', '0.27', '0.23',
       '0.24', '0.19', '0.38', '0.18', '0.21', '0.17', '0.29', '0.47',
       '0.3', '0.34', '0.55', '0.49', '0.39', '0.52', '0.4', '0.36',
       '0.46', '0.57', '0.44', '0.45', '0.33', '0.37', '0.76', '0.54',
       '0.43', '0.5', '0.51', '0.78', '0.48', '0.65', '0.59', '0.61',
       '0.89', '0.62', '0.63', '0.71', '0.7', '0.68', '0.67', '0.58',
       '0.56', '0.74', '0.91', '0.64', '0.72', '0.66', '0.75', '0.53',
       '0.69', '0.77', '0.8', '0.73', '0.96', '0.6', '0.79', '0.92',
       '0.87', '0.81', '0.82', '0.86', '1.07', '1.02', '1.1', '1.16',
       '1.32', '1.28', '1.08', '0.94', '0.84', '***', None], dtype=object)
```

Using the locate function, we are returned all instances of '***' within the dataframe. There are five instances and all appear to be in 2016.

[6542]:
```
df.loc[df[ 'global_avg_temp  '] == '***']
```

[6542]:

date	emission	anomaly_value	upper_95_ci	lower_95_ci
2016-08-31	1.500000e+12	0.744	0.707	0.783
2016-09-30	1.500000e+12	0.790	0.735	0.828
2016-10-31	1.500000e+12	0.729	0.693	0.772
2016-11-30	1.500000e+12	0.598	0.565	0.640
2016-12-31	1.500000e+12	0.553	0.516	0.593

```
            AverageTemperature AverageTemperatureUncertainty \
date
2016-08-31                NaN                           NaN
2016-09-30                NaN                           NaN
2016-10-31                NaN                           NaN
2016-11-30                NaN                           NaN
2016-12-31                NaN                           NaN

            All natural disasters global_avg_temp
date
2016-08-31                 380.0             ***
2016-09-30                 380.0             ***
2016-10-31                 380.0             ***
2016-11-30                 380.0             ***
2016-12-31                 380.0             ***
```

Next I search for all instances of 'None' within the global_avg_temp column. We are returned all of the rows at the end of the dataset, from 2017 until 2020.

In this result we can also see that AverageTemperature and AverageTemperatureUncertainty are missing values for those rows.

[6543]:
```
df_globavg_na  = df[df['global_avg_temp ' ].isna()]
df_globavg_na
```

[6543]:
```
                emission anomaly_value upper_95_ci lower_95_ci \
date
2017-01-31 1.540000e+12         0.620       0.568       0.655
2017-02-28 1.540000e+12         0.739       0.694       0.783
2017-03-31 1.540000e+12         0.845       0.793       0.891
2017-04-30 1.540000e+12         0.873       0.837       0.914
2017-05-31 1.540000e+12         0.737       0.698       0.777
2017-06-30 1.540000e+12         0.659       0.614       0.698
2017-07-31 1.540000e+12         0.641       0.596       0.686
2017-08-31 1.540000e+12         0.651       0.609       0.685
2017-09-30 1.540000e+12         0.714       0.659       0.756
2017-10-31 1.540000e+12         0.557       0.517       0.597
2017-11-30 1.540000e+12         0.571       0.531       0.612
2017-12-31 1.540000e+12         0.554       0.512       0.592
2018-01-31 1.580000e+12         0.600       0.542       0.640
2018-02-28          NaN         0.554       0.507       0.603
2018-03-31          NaN         0.528       0.477       0.578
2018-04-30          NaN         0.615       0.575       0.655
2018-05-31          NaN         0.627       0.586       0.666
2018-06-30          NaN         0.587       0.544       0.626
2018-07-31          NaN         0.573       0.536       0.619
2018-08-31          NaN         0.594       0.553       0.632
```

```
2018-09-30          NaN          0.586       0.530       0.625
2018-10-31          NaN          0.598       0.563       0.642
2018-11-30          NaN          0.678       0.640       0.721
2018-12-31          NaN          0.590       0.549       0.630
2019-01-31          NaN          0.638       0.583       0.682
2019-02-28          NaN          0.738       0.690       0.785
2019-03-31          NaN          0.662       0.610       0.713
2019-04-30          NaN          0.874       0.833       0.915
2019-05-31          NaN          0.780       0.738       0.823
2019-06-30          NaN          0.610       0.566       0.652
2019-07-31          NaN          0.708       0.666       0.753
2019-08-31          NaN          0.706       0.662       0.743
2019-09-30          NaN          0.719       0.662       0.761
2019-10-31          NaN          0.713       0.674       0.754
2019-11-30          NaN          0.752       0.719       0.796
2019-12-31          NaN          0.693       0.654       0.732
2020-01-31          NaN          0.879       0.823       0.921
           AverageTemperature AverageTemperatureUncertainty \
date
2017-01-31                NaN                           NaN
2017-02-28                NaN                           NaN
2017-03-31                NaN                           NaN
2017-04-30                NaN                           NaN
2017-05-31                NaN                           NaN

2017-06-30                NaN                           NaN
2017-07-31                NaN                           NaN
2017-08-31                NaN                           NaN
2017-09-30                NaN                           NaN
2017-10-31                NaN                           NaN
2017-11-30                NaN                           NaN
2017-12-31                NaN                           NaN
2018-01-31                NaN                           NaN
2018-02-28                NaN                           NaN
2018-03-31                NaN                           NaN
2018-04-30                NaN                           NaN
2018-05-31                NaN                           NaN
2018-06-30                NaN                           NaN
2018-07-31                NaN                           NaN
2018-08-31                NaN                           NaN
2018-09-30                NaN                           NaN
2018-10-31                NaN                           NaN
2018-11-30                NaN                           NaN
2018-12-31                NaN                           NaN
2019-01-31                NaN                           NaN
2019-02-28                NaN                           NaN
2019-03-31                NaN                           NaN
2019-04-30                NaN                           NaN
```

2019-05-31	NaN	NaN
2019-06-30	NaN	NaN
2019-07-31	NaN	NaN
2019-08-31	NaN	NaN
2019-09-30	NaN	NaN
2019-10-31	NaN	NaN
2019-11-30	NaN	NaN
2019-12-31	NaN	NaN
2020-01-31	NaN	NaN

date	All natural disasters	global_avg_temp
2017-01-31	325.0	None
2017-02-28	325.0	None
2017-03-31	325.0	None
2017-04-30	325.0	None
2017-05-31	325.0	None
2017-06-30	325.0	None
2017-07-31	325.0	None
2017-08-31	325.0	None
2017-09-30	325.0	None
2017-10-31	325.0	None
2017-11-30	325.0	None
2017-12-31	325.0	None
2018-01-31	276.0	None
2018-02-28	276.0	None
2018-03-31	276.0	None
2018-04-30	276.0	None
2018-05-31	276.0	None
2018-06-30	276.0	None
2018-07-31	276.0	None
2018-08-31	276.0	None
2018-09-30	276.0	None
2018-10-31	276.0	None
2018-11-30	276.0	None
2018-12-31	276.0	None
2019-01-31	282.0	None
2019-02-28	0.0	None
2019-03-31	0.0	None
2019-04-30	0.0	None
2019-05-31	0.0	None

```
2019-06-30                    0.0           None
2019-07-31                    0.0           None
2019-08-31                    0.0           None
2019-09-30                    0.0           None
2019-10-31                    0.0           None
2019-11-30                    0.0           None
2019-12-31                    0.0           None
2020-01-31                    0.0           None
```

For now, we will change the datatype of the global_avg_temp column from object to numeric so that it can be plotted and we can view the overall trends to determine the next steps.

[6544]:
```
df['global_avg_temp'] = df['global_avg_temp'].apply(pd.to_numeric, errors =␣
↪'coerce')
```

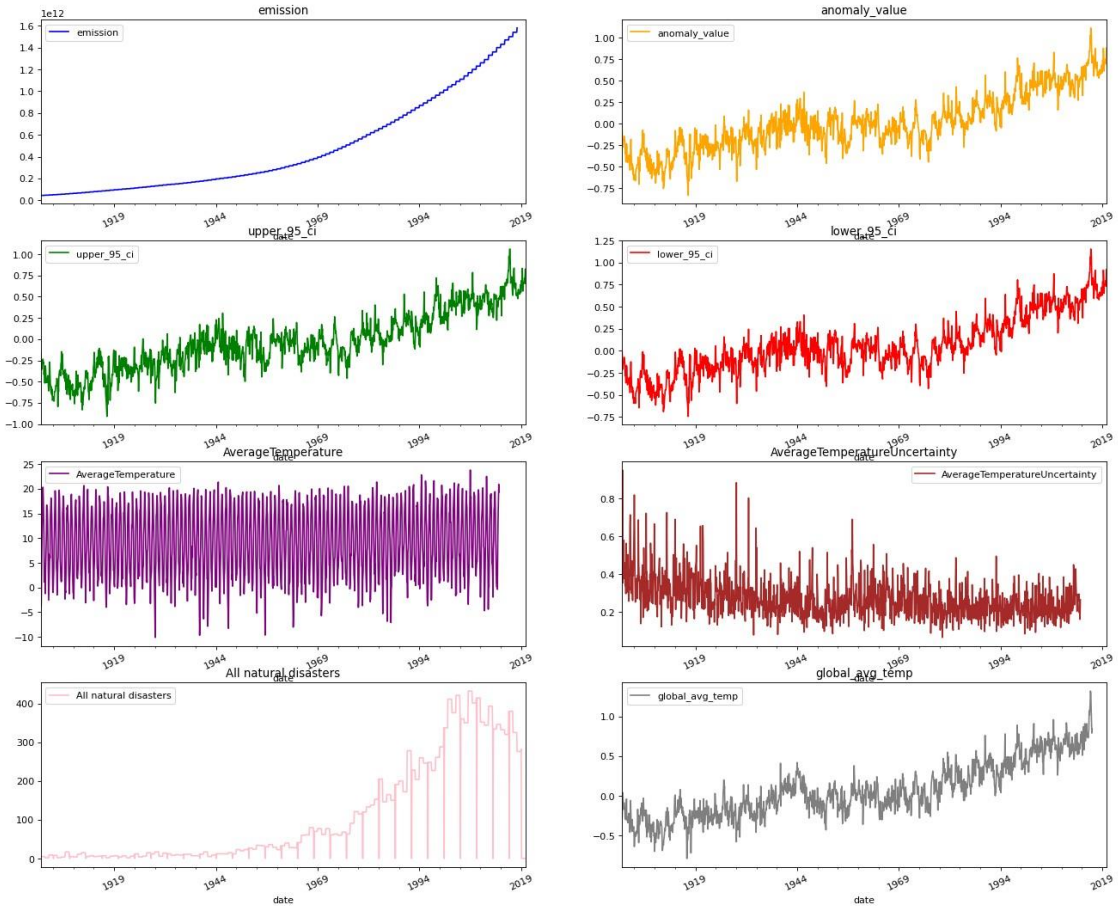
Second Visualization

The plots below display all remaining variables over time. Global_avg_temp appears to correlate closely with the target variable, anomaly_value.

[6545]:
```
colors = ['blue','orange','green','red','purple','brown','pink','gray']

def Visualize2(data):
    features = list(df.columns.values)
    feature_size =len(features)
    fig, axes = plt.subplots(
        nrows=int(np.ceil(feature_size /2)), ncols=2,
        figsize=(20, feature_size *2),dpi=80,facecolor='w', edgecolor='k'
    )
    for i in range(feature_size):
        key=features[i]
        c=colors[i % (len(colors))]
```

```
        t_data  = data[key]
        t_data .head()
        ax=t_data .plot(
             ax=axes[i //2, i%2],
             color =c,
             title ='{}'.format(key),
             rot =25
        )
        ax.legend([key])
plt.tight_layout()

Visualize2(df)

#code adapted from: https://www.youtube.com/watch?v=TpQtD7ONfxQ
```

```
<Figure size 640x480 with 0 Axes>
```



[6546]:
```
print('Correlation coefficient of anomaly_value and upper_95_ci:␣
↪',df['anomaly_value'].corr(df['upper_95_ci']).round(3)) print('Correlation
coefficient of anomaly_value and lower_95_ci:␣
```

```
↪',df['anomaly_value'].corr(df['lower_95_ci']).round(3))
```

```
Correlation coefficient of anomaly_value and upper_95_ci: 0.998
Correlation coefficient of anomaly_value and lower_95_ci: 0.998
```

[6547]:
```
df_new  = df.copy()
```

Remove Upper and Lower 95 CI columns

Above we can see that the correlation between these columns is nearly perfect.

Since these columns are also measurements of anomaly value, inputting them to the models may introduce a multicollinearity problem, artificially enhance the performance of the models, and cloud our ability to interpret the results. Therefore, I will drop both of them from the dataframe.

[6548]:
```
df = df.drop(['upper_95_ci','lower_95_ci'], axis=1)
```

3.1 Feature Selection and Further Data Preparation

The features I select should optimize the performance of the model while minimizing the computational power required. Selecting only a few features for the multivariate models will help minimize the computer resources required while also making the results easier to interpret. https://machinelearningmastery.com/an-introduction-to-feature-selection/

My goal for this section is to narrow the number of features down to three or four. I continue to explore and handle missing values, and I investigate the relationship between the variables and the target variable in order to select the most powerful predictors.

[6549]:
```
df.isna() .sum()
```

[6549]:
```
emission  24 anomaly_value  0
AverageTemperature              77
AverageTemperatureUncertainty   77
All natural disasters  0 global_avg_temp 42
dtype: int64
```

Since AverageTemperature and AverageTemperatureUncertainty have the most remaining null values (77), I create a dataframe to observe where they are within the dataset.

I can see that both of these columns are missing all values from 2013 until 2020 - which is nearly seven years of missing data. Our dataset spans 120 years, and so these columns are missing 5.8% of total data.

There are a few options for handling these missing values. One option is to trim off all rows in which there are missing values. I would prefer not to do so since it would mean losing data from other columns. Another option is to impute values into the missing rows. For AverageTemperature, it would make the most sense to perform monthy mean imputation, since monthly AverageTemperature values are similar from year to year. However, imputing seven years of data, 5.8% of the dataset, would certainly affect the final results. Finally, we could remove the columns entirely. In order to decide what to do, I create a correlation heatmap to observe how the variables relate to one another.

[6550]:
```
df_avgtemp  = df[df[ 'AverageTemperature  '].isna()]
df_avgtemp
```

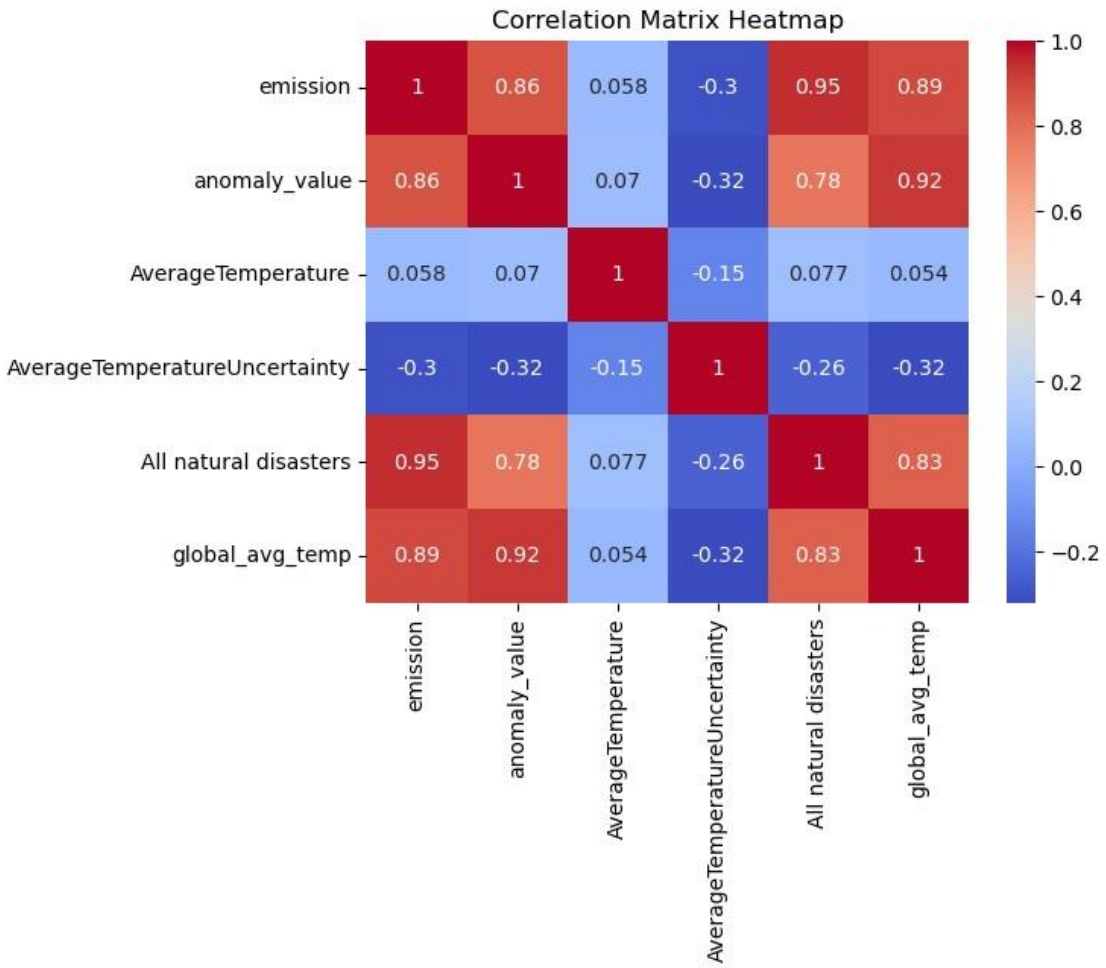
[6550]:
```
                emission anomaly_value AverageTemperature \
date
```

```
2013-09-30 1.400000e+12          0.541                 NaN
2013-10-31 1.400000e+12          0.546                 NaN
2013-11-30 1.400000e+12          0.516                 NaN
2013-12-31 1.400000e+12          0.660                 NaN
2014-01-31 1.430000e+12          0.532                 NaN
…                   …            …                     …
2019-09-30          NaN          0.719                 NaN
2019-10-31          NaN          0.713                 NaN
2019-11-30          NaN          0.752                 NaN
2019-12-31          NaN          0.693                 NaN
2020-01-31          NaN          0.879                 NaN

            AverageTemperatureUncertainty All natural disasters \
date
2013-09-30                            NaN                 346.0
2013-10-31                            NaN                 346.0
2013-11-30                            NaN                 346.0
2013-12-31                            NaN                 346.0
2014-01-31                            NaN                 332.0
…                                     …                   …
2019-09-30                            NaN                   0.0
2019-10-31                            NaN                   0.0
2019-11-30                            NaN                   0.0
2019-12-31                            NaN                   0.0
2020-01-31                            NaN                   0.0

            global_avg_temp
date
2013-09-30             0.77
2013-10-31             0.70
2013-11-30             0.81
2013-12-31             0.67
2014-01-31             0.73
…                      …
2019-09-30              NaN
2019-10-31              NaN
2019-11-30              NaN
2019-12-31              NaN
2020-01-31              NaN
[77 rows x 6 columns]
```

Correlation Matrix Heatmap

This heatmap shows a matrix of correlation coefficients for all variables colored according to the strength of the correlation. The closer the value to 1 (absolute value), the stronger the correlation.

[6551]:
```
corr = df.corr()

# Generate heatmap
ax = sns.heatmap(corr, cmap ='coolwarm ', annot =True)
ax.set_title( 'Correlation Matrix Heatmap  ')
plt.show()
```



The heatmap shows strong correlation between global_avg_temp (92%), All natural disasters (78%), emission (86%), and the target variable, anomaly_value.

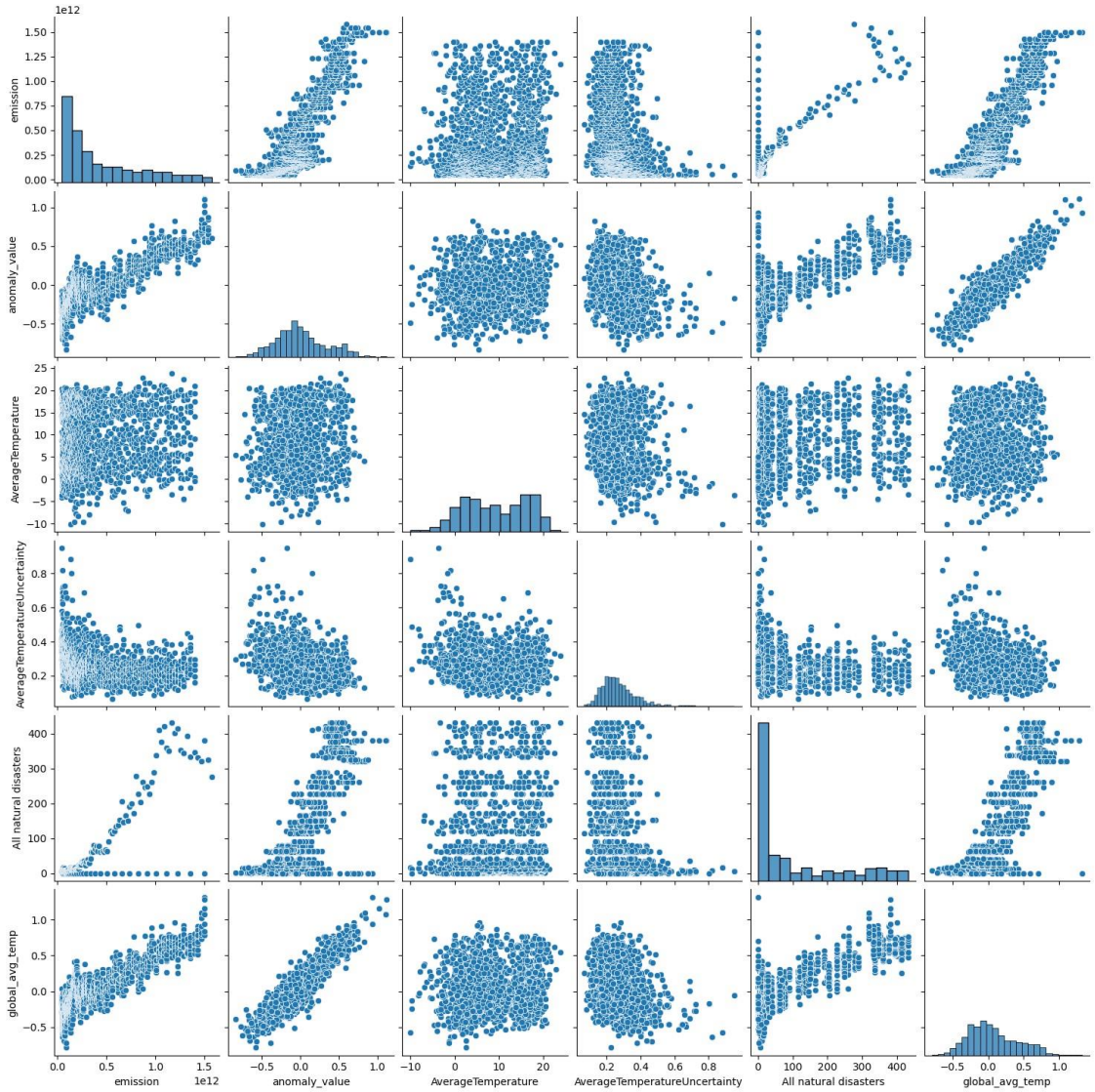
AverageTemperature and AverageTemperatureUncertainty show a very weak correlation to anomaly_value (7% and -32% respectively).

Pairplots

Below is a matrix of scatterplots showing the correlation between all variables. A strong correlation between two variables is represented by a relatively straight line. We can see that anomaly value shows a relatively strong correlation with emission, All natural disasters, and global_avg_temp. Meanwhile, the plots with AverageTemperature and AverageTemperatureUncertainty appear pretty random, indicating weak to no correlation.

One issue with these measures of correlation is that they are between variables at the same point in time. Ultimately, our models will be predicting anomaly value 3 months in the future. Therefore, I will next investigate auto-correlation, which checks for correlation between a time series and a lagged time series.

[6552]:
```
pairplots  = sns.pairplot(df)
```



3.1.1 Auto-Correlation

Autocorrelation produces a coefficient which indicates the strength of correlation between two variables at different points in a time series. The stronger the correlation, the better a predictor the lagged variable is of the other variable. This is a useful measurement in selecting features for our models, as the purpose of our models is to predict future anomaly values based on past values of features. https://statisticsbyjim.com/time-series/autocorrelation-partial-autocorrelation/

Below I calculated the autocorrelations of each feature with a lag of 3. We can see that emission, all natural disasters, and global_avg_temp have the strongest correlations, while AverageTemperature and AverageTemperatureUncertainty are very weak.

[6553]:
```
#autocorrelations with lag-3 print('Autocorrelation
Coefficient of Emission and Anomaly Value:␣
 ↪',df['emission'].shift(3).corr(df['anomaly_value']).round(3))
print('Autocorrelation Coefficient of All natural disasters and Anomaly Value:␣
 ↪',df['All natural disasters'].shift(3).corr(df['anomaly_value']).round(3))
print('Autocorrelation Coefficient of global_avg_temp and Anomaly Value:␣
 ↪',df['global_avg_temp'].shift(3).corr(df['anomaly_value']).round(3))
print('Autocorrelation Coefficient of AverageTemperature and Anomaly Value:␣
 ↪',df['AverageTemperature'].shift(3).corr(df['anomaly_value']).round(3))
print('Autocorrelation Coefficient of AverageTemperatureUncertainty and Anomaly␣
↪Value: ',df['AverageTemperatureUncertainty'].shift(3).
 ↪corr(df['anomaly_value']).round(3))
```

```
Autocorrelation Coefficient of Emission and Anomaly Value: 0.864
Autocorrelation Coefficient of All natural disasters and Anomaly Value: 0.785
Autocorrelation Coefficient of global_avg_temp and Anomaly Value: 0.908
Autocorrelation Coefficient of AverageTemperature and Anomaly Value: 0.092
Autocorrelation Coefficient of AverageTemperatureUncertainty and Anomaly Value:
-0.32
```

Drop AverageTemperature and AverageTemperatureUncertainty

Because they do not appear to be strong predictors of anomaly value, I drop both columns.

[6554]:
```
df = df.drop(['AverageTemperature', 'AverageTemperatureUncertainty'], axis=1)
```

Handling final null values

Now our feature with the most missing values is global_avg_temp. All of the missing values are at the end of the dataset, from 8-31-2016 until 1-31-2020 (including the '***' instances mentioned above). The emission column is also missing 24 values in that date range.

I am faced with a choice of performing some type of imputation on the missing values, or trimming the last few years of the dataset.

Since the missing values are consecutive, and constitute multiple years of data, it seems likely that any method of interpolating values - either by linear or forward fill - might lead to inaccurate predictions. So I decide that in lieu of imputation, it is best to remove all rows in that date range.

Losing only 3.33% of the total dataset is favorable to making predictions based on 4 consecutive years of imputed data.

[6555]:
```
df = df.reset_index( 'date ')
df = df.drop(df[df[ 'date '] >= '2016-08-31 '].index)
```

We can see below that the last rows were successfully trimmed.

[6556]:
```
df.set_index( 'date ', inplace =True )
df.tail()
```

[6556]:
```
                emission anomaly_value All natural disasters \
date
 2016-03-31 1.500000e+12         1.111                 380.0
 2016-04-30 1.500000e+12         1.106                 380.0
 2016-05-31 1.500000e+12         0.937                 380.0
 2016-06-30 1.500000e+12         0.707                 380.0
 2016-07-31 1.500000e+12         0.744                 380.0
             global_avg_temp
date
 2016-03-31             1.28
 2016-04-30             1.08
 2016-05-31             0.94
 2016-06-30             0.79
 2016-07-31             0.84
```

Final Feature Selection

I have achieved zero null values in all columns and narrowed the dataset to the following features:

- emission
- All natural disasters
- global_avg_temp • anomaly_value

And our target variable:

- anomaly_value (we will add a lag of 3)

[6557]:
```
df.isna() .sum()
```

[6557]:
```
emission  0 anomaly_value   0
All natural disasters  0
global_avg_temp  0
dtype: int64
```

Our dataframe now has 1,387 rows and 4 columns.

[6558]:
```
df.shape
```

[6558]:
```
(1387, 4)
```

[6559]:
```
df_LSTM  = df.copy()  #creating copies for the LSTM models
df_LSTM2  = df.copy()
```

3.2 Feature Engineering

To prepare the target feature, I created a new column of anomaly values shifted forward by three steps, as we will be forecasting values three steps in the future. I want the independent features to be three time steps behind the

target. This way, our input with which we will train the models uses X,y pairs of X = emission, anomaly value, All natural disasters, and global avg temp, and y = anomaly value 3 time steps ahead.

[6560]:
```
df['anomaly_value_target  '] = df['anomaly_value '].shift( -3)
df.head()
```

[6560]:
```
emission anomaly_value All natural disasters \ date
1901-01-31 4.482107e+10        -0.075                   5.0
1901-02-28 4.482107e+10        -0.176                   5.0
1901-03-31 4.482107e+10        -0.272                   5.0
1901-04-30 4.482107e+10        -0.236                   5.0
1901-05-31 4.482107e+10        -0.187                   5.0

            global_avg_temp anomaly_value_target
date
1901-01-31           -0.28               -0.236
1901-02-28           -0.06               -0.187
1901-03-31            0.04               -0.196
1901-04-30           -0.06               -0.146
1901-05-31           -0.17               -0.188
```

[6561]:
```
df.tail( 5)
```

[6561]:
```
                emission anomaly_value All natural disasters \
date
2016-03-31 1.500000e+12         1.111                 380.0
2016-04-30 1.500000e+12         1.106                 380.0
2016-05-31 1.500000e+12         0.937                 380.0
2016-06-30 1.500000e+12         0.707                 380.0
2016-07-31 1.500000e+12         0.744                 380.0

            global_avg_temp anomaly_value_target
date
2016-03-31            1.28                0.707
2016-04-30            1.08                0.744
2016-05-31            0.94                  NaN
2016-06-30            0.79                  NaN
2016-07-31            0.84                  NaN
```

Since shifting created three null values at the top of the dataset, I remove all three top rows from the dataset.

[6562]:
```
df = df.reset_index( 'date ')
df = df.drop(df .tail( 3).index)
df = df.set_index( 'date ')
df.head()
```

[6562]:
```
                emission anomaly_value All natural disasters \
date
1901-01-31 4.482107e+10        -0.075                   5.0
1901-02-28 4.482107e+10        -0.176                   5.0
1901-03-31 4.482107e+10        -0.272                   5.0
```

```
1901-04-30 4.482107e+10        -0.236                   5.0
1901-05-31 4.482107e+10        -0.187                   5.0

           global_avg_temp anomaly_value_target
date
1901-01-31           -0.28               -0.236
1901-02-28           -0.06               -0.187
1901-03-31            0.04               -0.196
1901-04-30           -0.06               -0.146
1901-05-31           -0.17               -0.188
```

[6563]:
```
df.describe()
```

[6563]:
```
           emission anomaly_value All natural disasters global_avg_temp \
count 1.384000e+03    1384.000000           1384.000000     1384.000000
mean   4.541142e+11      -0.011314            102.351879        0.065484
std    4.023181e+11       0.311308            129.874091        0.345016
min    4.482107e+10      -0.832000              0.000000       -0.790000
25%    1.320000e+11      -0.229250              9.000000       -0.190000
50%    2.800000e+11      -0.043000             29.000000        0.000000
75%    7.160000e+11       0.161250            165.000000        0.292500
max    1.500000e+12       1.111000            432.000000        1.320000

       anomaly_value_target
count           1384.000000

mean              -0.009210

std                0.313484

min               -0.832000

25%               -0.229000

50%               -0.042000

75%                0.167250

max                1.111000
```

Finally, I create copies of the dataframe to apply to each model. Each model has different preprocessing requirements such as scaling or differencing, and so more data preparation will take place before fitting each model.

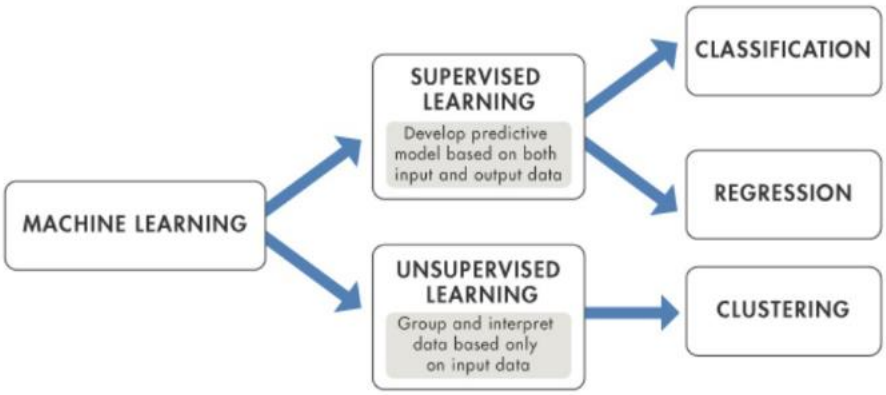
[6564]:
```
df1 = df.copy() #for Multiple Linear regression
model df2 = df.copy() #for random forest model df5
= df.copy() #for XGboost model
```

3 Modeling

In this project, I will be implementing a sub-type of machine learning called supervised learning. Supervised learning models are trained using input-output pairs, and then make predictions on new input data. By contrast, unsupervised models use only input data for training, and produce results without having any knowledge of actual output values.

To be specific, the input-output pairs for our models will consist of an X_train and a y_train set.
The X_train will be the independent features, and the y_train will be the lagged target feature (anomaly_value).
After the model is trained, I will make predictions of future y values based on an X_test set. Finally, the y predictions will be compared to a y_test set (actual anomaly values) and I will calculate several metrics to assess how accurately the model was able to predict.

Supervised models can be either classification or regression models. Classification models are used to predict categorical variables in tasks such as image classification or email spam identification. In this project, we will be implementing regression models. Regression models are used to predict continuous output - continuous meaning a quantitative value with an infinite range of possibilities.



https://www.mathworks.com/help/stats/machine-learning-in-matlab.html

What is a regression model?

Regression is a statistical concept which can describe the relationship between independent variables (one or more) and a dependent variable. Through a correlation coefficient, it shows how much the variation in the dependent variable can be explained by variations in the independent variable(s). The closer the correlation coefficient is to 1, the stronger the correlation. This is determined by finding the line of best fit through a scatterplot of (X,y) datapoints. In its simplest form, the regression equation for line of best fit is:

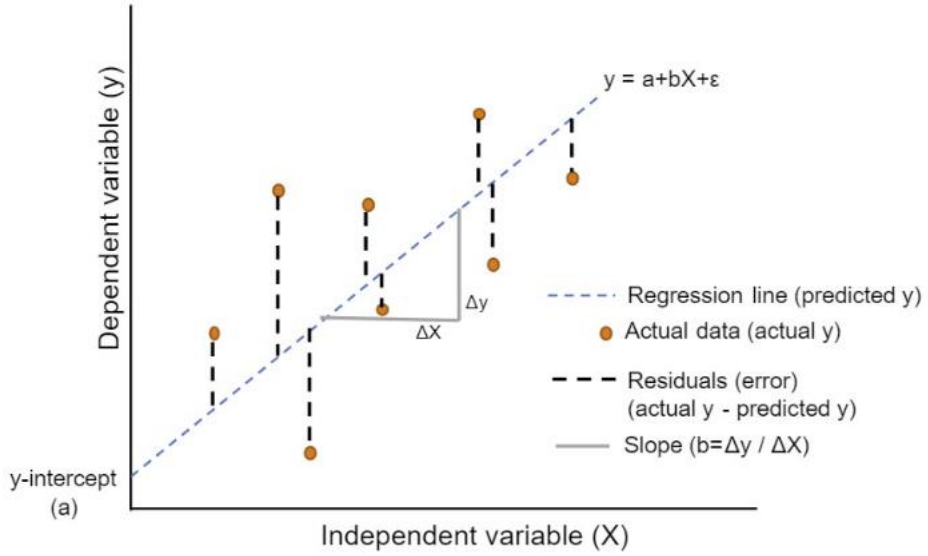
Simple linear regression:

$$Y = a + bX + u$$

https://www.investopedia.com/terms/r/regression.asp

To get the residuals - or errors, you calculate the distance from the actual values to the line of best fit. The line with the lowest sum of squares (residuals) is fitted to the data and used to predict y values for new X inputs.

$$\text{residuals} = \text{actual } y(y_i) - \text{predicted } y\ (\hat{y}_i)$$



https://www.reneshbedre.com/blog/learn-to-calculate-residuals-regression.html?utm_content=cmptrue

Models used for this project

I will use the following models to try to predict future anomaly values:

- Multiple Linear Regression
- Random Forest Regression
- LSTM (recurrent neural network)
- XGBoost Regressor

4.1 Steps of the Modeling Process

- Step 1: Preparing input data
  - This step includes the data preparation and feature engineering I performed above
  - Data pre-processing is model-specific and can include things like scaling and removing time-dependent trends:
    - Scaling is used to either normalize or standardize the data for machine learning models. Normalization includes things like min-max scaling - which scales the data to fit within the range of the minimum and maximum of the series. This helps to remove the possibility of bias if the machine assigns greater importance to features of higher magnitude. Standardization is performed to change the shape of the data to a normal distribution. You can use the standard

scaler function to do this. Whether or not to scale depends on the specific dataset and the model being used, but trial and error can be helpful to compare performance with and without scaling.

* Whether or not to remove time-dependent trends or seasonality is also dependent on the dataset and model being used. Time series basically inherently have time-dependent trends. The technical term for time-trended data is non-stationary (meaning the mean and variance are not constant over time). To check for stationarity there are a variety of statistical tests including the Augmented Dickey-Fuller test, or you could just visualize your data plotted over time. Some models are timesensitive and others are not. Random Forest, for example, is not programmed to handle time series. That doesn't necessarily mean it isn't a good idea to use Random Forest for time series forecasting, however. It is possible to make your data stationary by using techniques like differencing. Once the data is stationary, it will likely be better understood by the model. Trial and error is also useful to determine if stationarity affects the performance of your model.

- Step 2: train - test - split
  - This step involves splitting the data into training and testing sets. Sklearn has a helpful train-test-split function which takes in an X and y and returns X_train, y_train, X_test, and y_test. The X_train and y_train data form the input data (the supervised inputoutput training pairs) with which to train the models. The X_test set is used to predict on, and the results are compared against the y_test set, which is actual output values. It is important to choose a good split to adequately train the model while avoiding overfitting.
  - Sometimes a third split is included called validation data. This is better to use if you have a large dataset. The validation sets evaluate the performance of the model and help to prevent overfitting, while the test set is used to evaluate the model's performance on new data post-training.
- Step 3: Constructing the model
  - For simpler models it is necessary only to instantiate an object of the model from whichever library (sklearn, keras, etc.). For neural networks, you must specify the layers of the model which recieve input, apply functions, and produce output. In this step you can set the hyperparameters to the optimum settings for your data. Hyperparameters are different for each model.
- Step 4: Compiling the model
  - This step configures the model with an optimizer and loss function. There are many choices for these functions but an optimizer with wide appeal is 'Adam' and a widely used loss function is 'mean squared error'. These are set by you, and so can be trial and errored. However, some models work best with certain functions.
- Step 5: Fitting the model
  - In this step you fit the model with your input data and validation data if applicable, and for certain models there are other hyperparameters that can be set. These include epochs (the number of times the data is passed through the model), batch size (the size of data passed through the model), verbose (which specifies how detailed the feedback appears as the model is training), and shuffle (which specifies whether or not to shuffle the data as it is fed to the model for training), and others.
- Step 6: Predicting
  - The models we use have a built in predict function in which we pass the X_test set. It produces an array of output values based on the knowledge the machine gained during training.
- Step 7: Evaluating the Predictions against the test set/Cross Validation
  - Now we compare the predictions against the y_test set to see how closely they match up. In this project I will use visualizations and four metrics outlined below.

– The four evaluation methods are:

Mean Absolute Error

Mean Absolute Error is the sum of the absolute value of all residuals divided by the total number of datapoints. It basically indicates the average error of the distance from the predicted values to the actual values. The result is relative to the units of data being predicted. In this case, we are predicting temperature anomaly values.

$$\mathbf{MAE} = \frac{1}{n}\sum_{i=1}^{n}|\boldsymbol{x}_i - \boldsymbol{x}|$$

https://www.statisticshowto.com/absolute-error/

Mean Squared Error

Mean Squared Error is the sum of all squared residuals divided by the total number of datapoints. By squaring instead of taking the absolute value of the residuals, MSE imposes a greater penalty on larger residuals. MSE is therefore larger than mean absolute error. It is also more difficult to interpret, since by squaring the residuals, it is no longer in the same units as the y value.

$$\mathbf{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \tilde{y}_i)^2$$

https://www.freecodecamp.org/news/machine-learning-mean-squared-error-regressionline-c7dde9a26b93/

Root Mean Squared Error

Root Mean Squared Error is simply the square root of the mean squared error. This returns the error to the units of the y values, which makes it easily interpretable.

$$RMSE = \sqrt{\sum_{i=1}^{n}\frac{(\hat{y}_i - y_i)^2}{n}}$$

https://towardsdatascience.com/what-does-rmse-really-mean-806b65f2e48e

R2

R2 is equal to 1 minus the sum of squared residuals over the total sum of squares. The total sum of squares is equal to the sum of squares of each actual value minus the mean of actual values. The smaller the sum of squared residuals with respect to the sum of squares, the higher the resulting R2. This means that the data is better fitted to the line of best fit than just the average y value. R2 returns

a value between 0 and 1 and can be positive or negative. The closer the value is to |1|, the stronger the correlation between the two variables.

$$R^2 = 1 - \frac{RSS}{TSS}$$

R^2 = coefficient of determination
RSS = sum of squares of residuals
TSS = total sum of squares

https://vitalflux.com/r-squared-explained-machine-learning/

https://machinelearningmastery.com/training-validation-test-split-and-cross-validation-doneright/

https://www.youtube.com/watch?v=i_LwzRVP7bg

The function below takes in test values and predicted values and performs four evaluation measures. It will be applied to each model.

[6565]:
```
def Evaluate(y_test, y_pred):
    try:
        print(f'Mean Absolute Error: {mean_absolute_error(y_test,y_pred)}')
        print(f'Mean Squared Error: {mean_squared_error(y_test,y_pred)}')
        print(f'Root Mean Squared Error:␣
↪{sqrt(mean_absolute_error(y_test,y_pred))}')
        print(f'R Squared:
        {r2_score(y_test,y_pred)}')
    except :
        pass
```

4 Multiple Linear Regression

Overview

Multiple linear regression models are simply linear regression models with one dependent variable and two or more independent variables. Information about the independent variables is used to make predictions about the dependent variable. This type of model makes a couple assumptions about the data:

- there is a linear relationship between the independent and dependent variables
- the independent variables are not too strongly correlated with one another (no perfect multicollinearity)
- the observations are independent of each other

The formula for the line of best fit in a multiple linear regression model is:

Formula and Calculation of Multiple Linear Regression

$$y_i = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + ... + \beta_p x_{ip} + \epsilon$$

where, for $i = n$ observations:

y_i = dependent variable

x_i = explanatory variables

β_0 = y-intercept (constant term)

β_p = slope coefficients for each explanatory variable

ϵ = the model's error term (also known as the residuals)

https://www.investopedia.com/terms/m/mlr.asp

Why I chose Multiple Linear Regression

Our data fits two assumptions for this model. Firstly, the pairplots and correlation coefficients in the feature selection section show a linear relationship between each feature and the target. Secondly, in the data preparation step I took care to try to avoid perfect multicollinearity among the features. Also, as a multiple regression model, it is able to accept multiple features and make predictions about the dependent variable, which are the basic requirements of solving the task at hand. However, the nature of a time series is that the observations are not independent of one another. I have to assume that each observation relies on past observations and thus there is a dependence. Nonetheless, this is a simple machine learning model that is easy to interpret, and so I am using it to establish baseline scores for the other models.

[6566]:
```
df1.head()
```

[6566]:
```
                emission anomaly_value All natural disasters \
date
1901-01-31 4.482107e+10        -0.075                    5.0
1901-02-28 4.482107e+10        -0.176                    5.0
1901-03-31 4.482107e+10        -0.272                    5.0
1901-04-30 4.482107e+10        -0.236                    5.0
1901-05-31 4.482107e+10        -0.187                    5.0

            global_avg_temp anomaly_value_target
date
1901-01-31           -0.28               -0.236
1901-02-28           -0.06               -0.187
1901-03-31            0.04               -0.196
1901-04-30           -0.06               -0.146
1901-05-31           -0.17               -0.188
```

Test Train Split

I assign the variables emission, anomaly_value, all natural disasters, and global avg temp to X, our independent variables. I assign the lagged anomaly values to y, our target - or dependent variable.

Then, I use sklearn's train_test_split function to divide the data into training and testing sets. I specify the test set as 20% of the data. This seems to be the best split for ensuring proper training while avoiding overfitting.

random_state is a hyperparameter of this function which sets the seed for the random number generator which splits the data into sets. Keeping this number the same is important for generating the same split with each pass.

By specifying shuffle = False, the data will not be shuffled before creating the split. I have kept this the same for each model so we can compare results across models more easily.

https://www.bitdegree.org/learn/train-test-split https://stackoverflow.com/questions/42191717/scikitlearn-random-state-in-splitting-dataset

[6567]:
```
X = df1.drop('anomaly_value_target  ',axis=1)
y = df1['anomaly_value_target  ']  #target feature

X_train, X_test, y_train, y_test     = train_test_split(X, y, test_size     = 0.2,␣
 ↪random_state  = 0, shuffle  = False)

print (X_train.shape)
print (X_test.shape)
print (y_train.shape)
print (y_test.shape)
```

```
(1107, 4)
(277, 4)
(1107,)
(277,)
```

Fitting the model and predicting

Next, I instantiate an object of a linear regression model. Then, I fit it with the X_train and y_train sets. Finally, I generate predictions on the basis of the X_test set.

[6568]:
```
model1  = LinearRegression()

model1.fit(X_train, y_train)

y_pred1  = model1.predict(X_test)
```

Evaluation

In the dataframe below you can see the predicted values are somewhat close to the actual values, but not exactly on target.

[6569]:
```
results_model1_df   = pd.DataFrame({ 'Actual values  ':y_test,  'Predicted values  ':
 ↪y_pred1})
results_model1_df
```

[6569]:

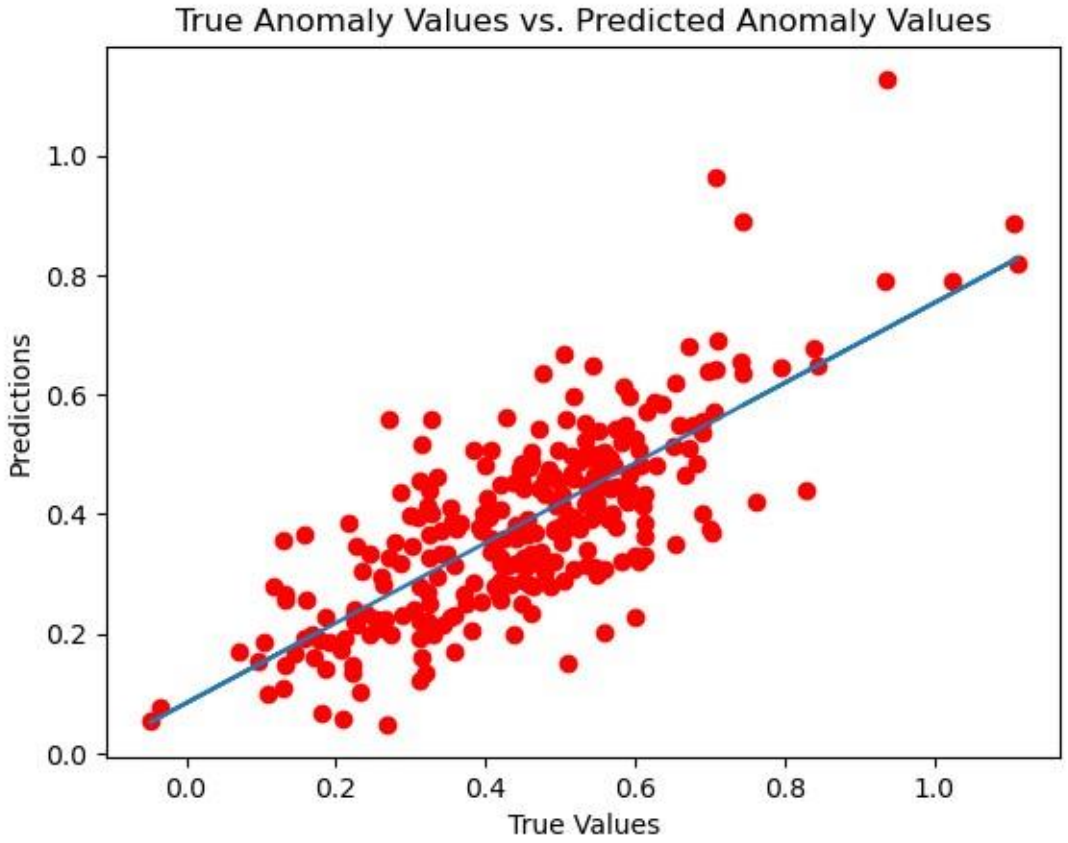
date	Actual values	Predicted values
1993-04-30	0.179	0.190364
1993-05-31	0.131	0.148249
1993-06-30	0.097	0.155507
1993-07-31	0.070	0.170657
1993-08-31	0.109	0.100278
…	…	…
2015-12-31	1.111	0.818984
2016-01-31	1.106	0.885992
2016-02-29	0.937	1.127348

```
2016-03-31          0.707          0.964199
2016-04-30          0.744          0.888447
[277 rows x 2 columns]
```

Visualizations

This plot shows the true anomaly values plotted against the predicted anomaly values with the line of best fit added. There does appear to be a reasonable linear relationship, although a tighter spread around the line would be much better.

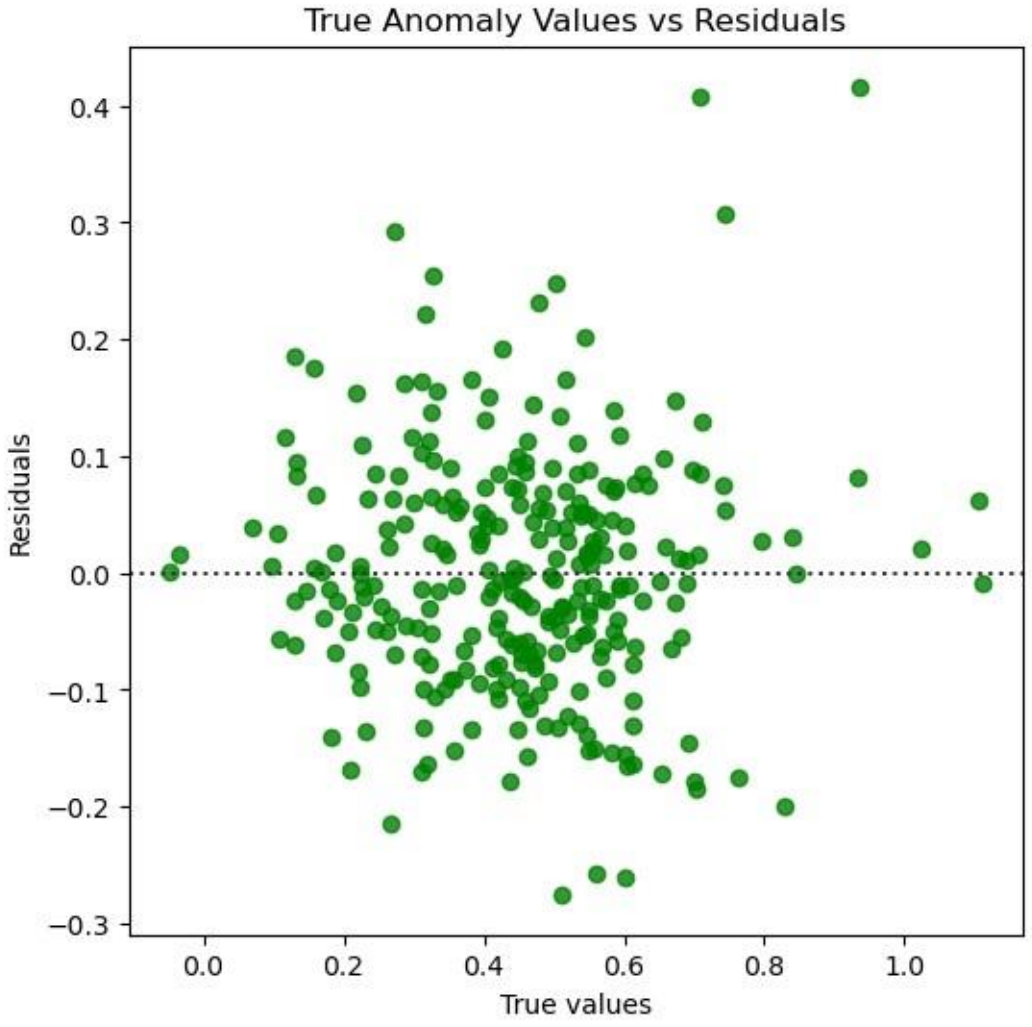
[6570]:
```
slope, intercept   = np.polyfit(y_test, y_pred1,    1)

plt.scatter(y_test, y_pred1, color     = ['red'])
plt.plot(y_test, slope    * y_test  + intercept)
plt.xlabel( 'True Values  ')
plt.ylabel( 'Predictions  ')
plt.title( 'True Anomaly Values vs. Predicted Anomaly Values      ')
plt.show()
```



A good residual plot should show a random scatter above and below the 0 line with no obvious patterns. Ours is a decent residual plot, as no clear patterns are visible and it appears random. https://www.statology.org/good-vs-bad-residual-plot/

[6571]:
```
plt.figure(figsize =(6,6))
sns.residplot(y_test, y_pred1, color    ='green')
plt.xlabel( 'True values ')
plt.ylabel( 'Residuals ')
plt.title( 'True Anomaly Values vs Residuals    ')
plt.show()
```

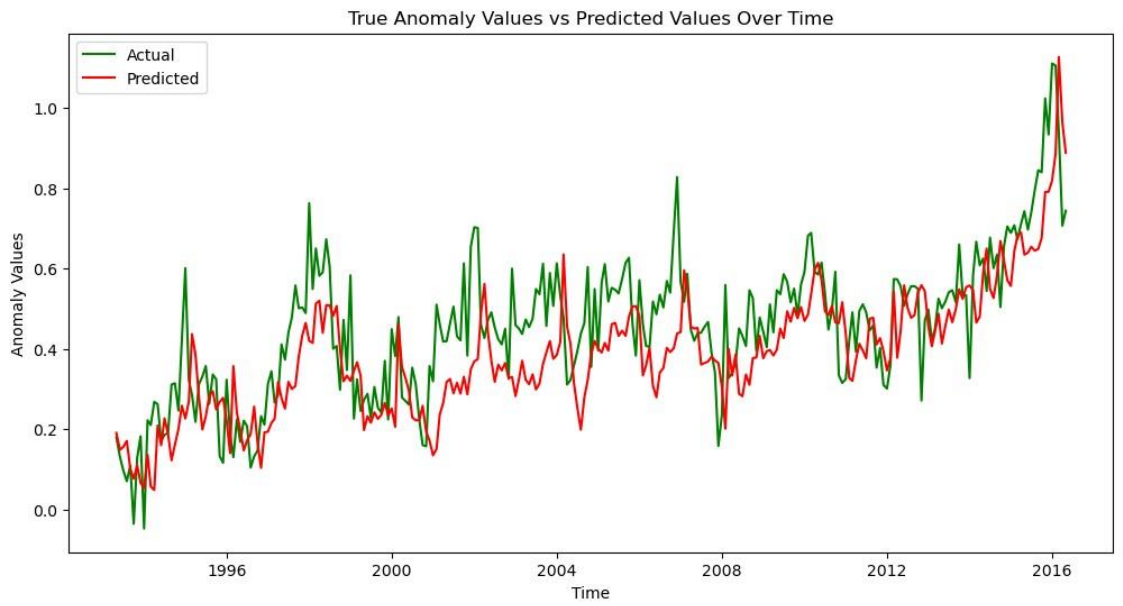
```
c:\Users\hashl\anaconda3\lib\site-packages\seaborn\_decorators.py:36:
FutureWarning:

Pass the following variables as keyword args: x, y. From version 0.12, the only
valid positional argument will be `data`, and passing other arguments without
an explicit keyword will result in an error or misinterpretation.
```



True vs Predicted Values over time

The predicted values appear to have captured some of the trends of the actual values, but there is some obvious misalignment between the two lines.

[6572]:
```
plt.figure(figsize=(12, 6)) plt.plot(y_test.index, y_test,
label='Actual', color='green') plt.plot(y_test.index,
y_pred1, label='Predicted', color='red') plt.title('True
Anomaly Values vs Predicted Values Over Time')
plt.xlabel('Time') plt.ylabel('Anomaly Values') plt.legend()
plt.show()
```



Evaluation Scores

The Evaluate function returns our four metrics below. MAE shows that the average residual is 0.1 degrees. In other words the average difference from the actual anomaly value to the predicted anomaly value is 0.1 degrees. As expected since squaring imposes a higher penalty on residuals, the RMSE is greater than MAE at 0.33 degrees. The r2 can be interpreted as: 43% of the variation in predicted anomaly values can be explained by variations in the independent variables. The closer r2 is to 1 the better the fit, so ours isn't ideal.

Since this is the first model and the simplest, I will use these scores as the baseline to assess the performance of the other models. My goal is to decrease the errors and increase the r2, ideally to above 70%.

[6573]:
```
Evaluate(y_test,y_pred1)
```

```
Mean Absolute Error: 0.10933566406200707
Mean Squared Error: 0.018475538316648214
Root Mean Squared Error: 0.3306594381867952
R Squared: 0.4315165447918714
```

Conclusions and Reflections

Although it is usually recommended to scale data before applying it to a multiple linear regression model, the model surprisingly performed much worse when I did. Also, removing trends and seasonality by differencing

negatively impacted the results of the model. Perhaps the algorithm in the linear regression model is equipped well enough to handle the original scale of the data in this particular dataset and non-stationary data. When I used the describe function to observe the descriptive statistics of the dataset, it appeared like anomaly value and global avg temperature are on a similar scale, but emission and all natural disasters are not. Since using the min-max-scaler on the dataset did not improve the performance, it makes me wonder if the r2 in this run is artificially inflated by the difference in scale.

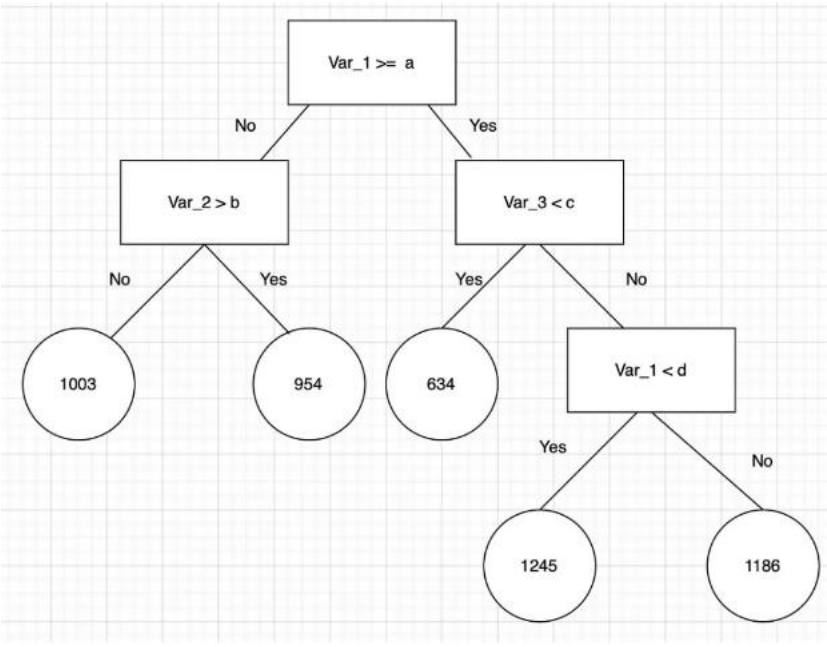
This was the simplest model I chose and I did not expect it to achieve great results, especially since all assumptions for the model were not met by our data. The next models are likely able to handle more complex data and so I expect to achieve better results than the linear regression model.

5 Random Forest Regression Model

Random Forest Regression models are ensemble learning models, meaning they use multiple models which make their own predictions and then average those predictions to get a final prediction. The model type used is decision tree. A random forest is essentially a forest of decision trees, generated independently from random samples of data and averaged together to produce a robust prediction. It is known as a bagging method which means that the random samples selected from the dataset can be sampled again.

In a similar way that crowd-sourcing opinions usually leads to better problem solving than a single person's efforts, a random forest model is more effective than a simple decision tree model. Aggregating the results of many different decision trees reduces the variance produced by any single tree.
https://www.kaggle.com/code/carlmcbrideellis/an-introduction-to-xgboost-regression

Below is an example of a decision tree. It starts at the root and branches off depending on variable conditions and ends up with a final decision at the 'leaf node'.



https://towardsdatascience.com/random-forest-regression-5f605132d19d

The primary hyperparameters in the random forest regression model are:

- n_estimators = the number of decision trees in the model

- criterion = the loss function used to make decisions. Default is mean squared error
- max_depth = the longest length of steps from root to leaf node of the decision trees. Too great a depth can lead to overfitting.
- bootstrap = accepts a boolean, True or False. True is the default and means that random samples are used to build the trees. False means that the whole dataset will be used to build the trees.
- max_samples = the largest size of each sample for each decision tree. bootstrap must be set to True in order to use this hyperparameter

I will tune these parameters to try to achieve the best result possible.

Since random forest isn't sensitive to time series, the data must be converted into a supervised learning problem first. We have already done that by shifting the anomaly values ahead by three steps. This created input(X's) - output(y+3) pairs to feed to the model for training. Hence, the resulting predictions should reflect anomaly values 3 months in the future.

https://machinelearningmastery.com/convert-time-series-supervised-learning-problem-python/

https://towardsdatascience.com/multivariate-time-series-forecasting-using-random-forest-

2372f3ecbad1

https://www.kaggle.com/code/pbizil/random-forest-regression-for-time-series-predict/notebook

https://towardsdatascience.com/random-forest-regression-5f605132d19d

[6574]:
```
df2.head()
```

[6574]:
```
                emission anomaly_value All natural disasters \
date
1901-01-31 4.482107e+10        -0.075                    5.0
1901-02-28 4.482107e+10        -0.176                    5.0
1901-03-31 4.482107e+10        -0.272                    5.0
1901-04-30 4.482107e+10        -0.236                    5.0
1901-05-31 4.482107e+10        -0.187                    5.0

            global_avg_temp anomaly_value_target
date
1901-01-31           -0.28               -0.236
1901-02-28           -0.06               -0.187
1901-03-31            0.04               -0.196
1901-04-30           -0.06               -0.146
1901-05-31           -0.17               -0.188
```

Stationarity

A time series is stationary if mean, variance, and standard deviation are constant over time. Some models require data to be stationary. Random forest does not, but after some experimentation, I found that the results of stationary data were slightly more interpretable than the results of the original, non-stationary data. The original data produced very high errors and an r2 above 200%, which can sometimes be caused by time-trends in data https://statisticsbyjim.com/regression/rsquared-too-high/.

Below I explore the stationarity of the data and handle it using differencing.

Augmented Dickey-Fuller Test

We can apply this test to the data to check for stationarity. With our eyes we can simply observe that there are time-dependent trends in our data, but applying this test gives an official answer. The null hypothesis of this test is that the data is non-stationary. We can reject the null if the returned p-value is less than the 95% significance level at 0.05.

[6575]:
```
columns = df2.columns
df_stats = pd.DataFrame(columns =['P-value '], index =columns)
for col in columns:
  dftest = adfuller(df2[col], autolag ='AIC') #applies the augmented dickey
 ↪fuller test to the data
  df_stats.loc[col] =dftest[1]

import plotly.graph_objects as go
fig = go.Figure(data =[go.Bar(
            x=df_stats.index, y =df_stats['P-value '],
            text =df_stats['P-value '],
            textposition ='auto',)])

fig.show() #plots the p-values
#code adapted from https://www.kaggle.com/code/pbizil/
 ↪random-forest-regression-for-time-series-predict/notebook
```

Differencing

The P-value for all variables is greater than 0.05, meaning we cannot reject the null hypothesis that our data is non-stationary. Next we perform differencing to make the data stationary. Differencing means to calculate the difference of each consecutive observation in a time series to remove trends.

[6576]:
```
data_diff = pd.DataFrame(columns =columns)
data_diff['date']=df2.index
for col in columns:
  data_diff[col] = pd.DataFrame(np.diff(df2[col])) #differencing

data_diff =data_diff.dropna()
for col in columns:
  dftest = adfuller(data_diff[col], autolag ='AIC') #applying ADF test again
  df_stats.loc[col] =dftest[1]

import plotly.graph_objects as go
fig = go.Figure(data =[go.Bar(
            x=df_stats.index, y =df_stats['P-value '], #graphing the new p-values
            text =df_stats['P-value '],
            textposition ='auto',)])

fig.show()

#code adapted from https://www.kaggle.com/code/pbizil/
 ↪random-forest-regression-for-time-series-predict/notebook
```

We can see that all variables except emission have become stationary, so we will difference emission again.

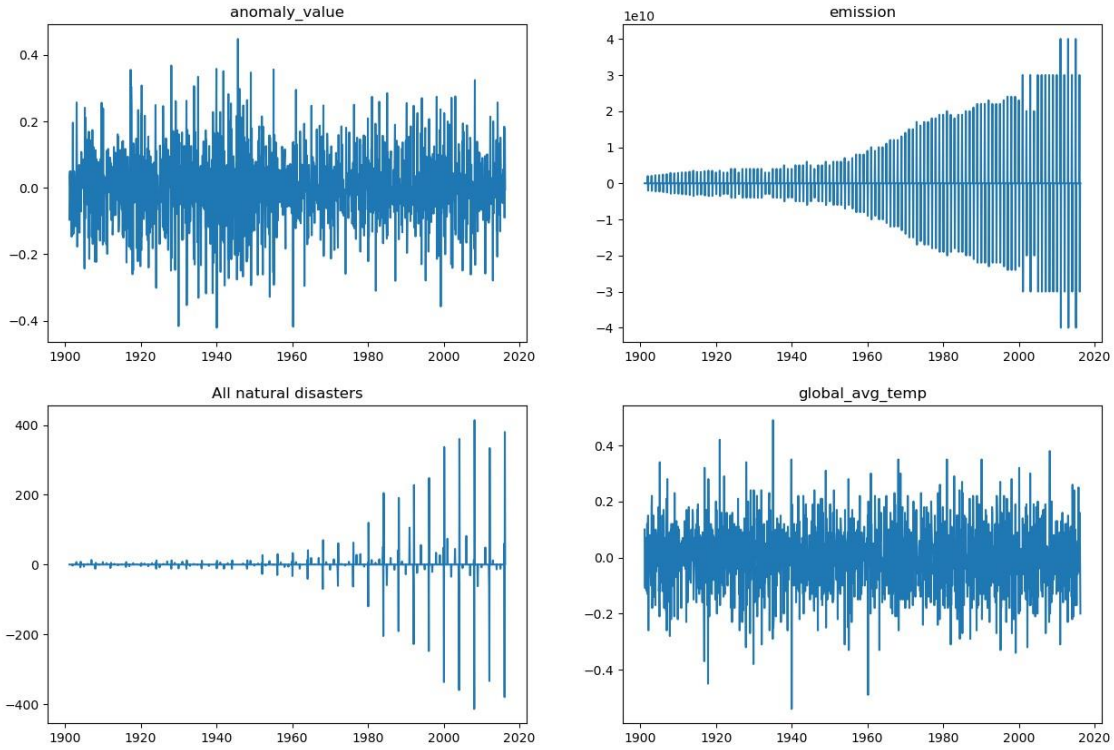
[6577]:
```
data_diff[ 'emission '] = data_diff[ 'emission '].diff() .dropna()
data_diff =data_diff .dropna()

for col in columns:
  dftest = adfuller(data_diff[col], autolag   ='AIC ')
  df_stats .loc[col] =dftest[ 1]


import plotly .graph_objects as go
fig = go.Figure(data =[go.Bar(
             x=df_stats .index, y =df_stats[ 'P-value '],
             text =df_stats[ 'P-value '],
             textposition ='auto ',)])

fig .show()
#code adapted from https://www.kaggle.com/code/pbizil/
 ↪random-forest-regression-for-time-series-predict/notebook
```

All p-values are now below 0, so our data is now stationary.

Visualizing Stationary Data

Below we can see that our data do appear to have less time-dependent trends than before differencing.

[6578]:
```
data_diff  = data_diff .set_index( 'date ') #resetting the index

fig, axs  = plt.subplots( 2, 2, figsize  = (15,10))
axs[ 0, 0].plot(data_diff[ 'anomaly_value ])
axs[ 0, 0].set_title( 'anomaly_value ')
axs[ 0, 1].plot(data_diff[ 'emission '])
axs[ 0, 1].set_title( 'emission ')
axs[ 1, 0].plot(data_diff[ 'All natural disasters  '])
axs[ 1, 0].set_title( 'All natural disasters  ')
axs[ 1, 1].plot(data_diff[ 'global_avg_temp '])
axs[ 1, 1].set_title( 'global_avg_temp ')
plt.show()
```



Train Test Split

We implement the same split as in the multiple linear regression model.

[6579]:
```
X = data_diff.drop('anomaly_value_target ', axis=1)
y = data_diff['anomaly_value_target ']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2,␣
↪random_state = 0, shuffle = False)

print(X_train.shape)
print(y_train.shape)
print(X_test.shape)
print(y_test.shape)
```

```
(1105, 4)
(1105,)
(277, 4)
(277,)
```

Training the model

We are using sklearn's random forest regressor. I set it to generate 100 decision trees with a max depth of 3 steps.

I tried a variety of configurations of the hyperparameters n_estimators, max_depth, and max_samples, and all produced unsatisfactory results. I will reflect on this at the end of the section.

[6580]:
```
model2 = RandomForestRegressor(n_estimators=100, max_depth = 3, random_state=0)
model2.fit(X_train, y_train)
```

[6580]:
```
RandomForestRegressor(max_depth=3, random_state=0)
```

Making Predictions

[6581]:
```
y_pred2 = model2.predict(X_test)
```

Actuals vs Predicted Values

In the results dataframe, the difference between the actual and predicted values isn't looking good. I can see that the differences seem quite large for each datapoint.

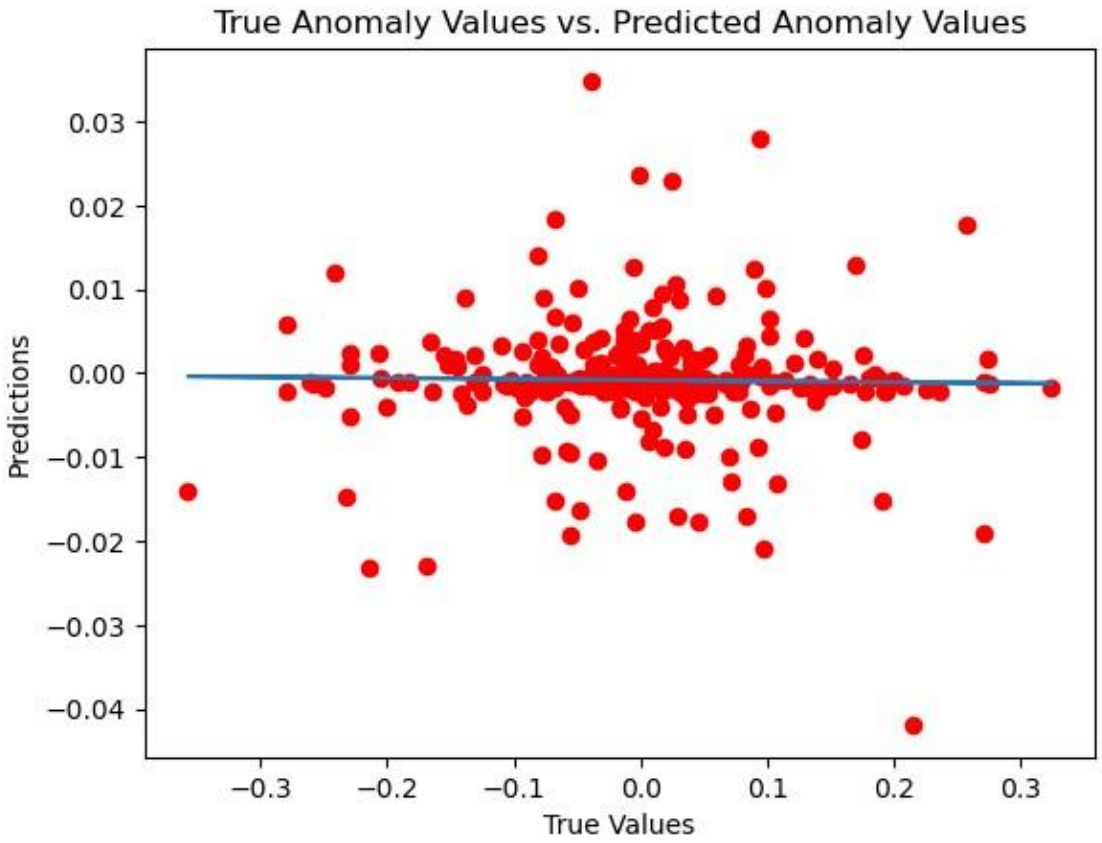
[6582]:
```
results_model2_df = pd.DataFrame({'Actual values ':y_test, 'Predicted values ':
↪y_pred2})
results_model2_df
```

[6582]:
```
            Actual values Predicted values
date
1993-03-31         0.001          0.000194
1993-04-30        -0.048         -
                                  0.000658
1993-05-31        -0.034         -
                                  0.001090
1993-06-30        -0.027         -
                                  0.002097
1993-07-31         0.039          0.000944
…                  …                …
2015-11-30         0.177         -
                                  0.002097
2015-12-31        -0.005         -
                                  0.017681
2016-01-31        -0.169         -
                                  0.023019
2016-02-29        -0.230          0.002382
2016-03-31         0.037         -
                                  0.003159
[277 rows x 2 columns]
```

Visualizations

Compared to the plot from the multiple linear regression model in which the points were clearly arranged in a linear fashion, this scatterplot appears very random. The line of best fit is nearly horizontal, pointing downward. There is a apparently a very poor fit of the data to the model.

[6583]:
```
slope, intercept   = np.polyfit(y_test, y_pred2,    1)

plt.scatter(y_test, y_pred2, color     = ['red'])
plt.plot(y_test, slope    * y_test  + intercept)
plt.xlabel( 'True Values ')
plt.ylabel( 'Predictions ')
plt.title( 'True Anomaly Values vs. Predicted Anomaly Values      ')
plt.show()
```

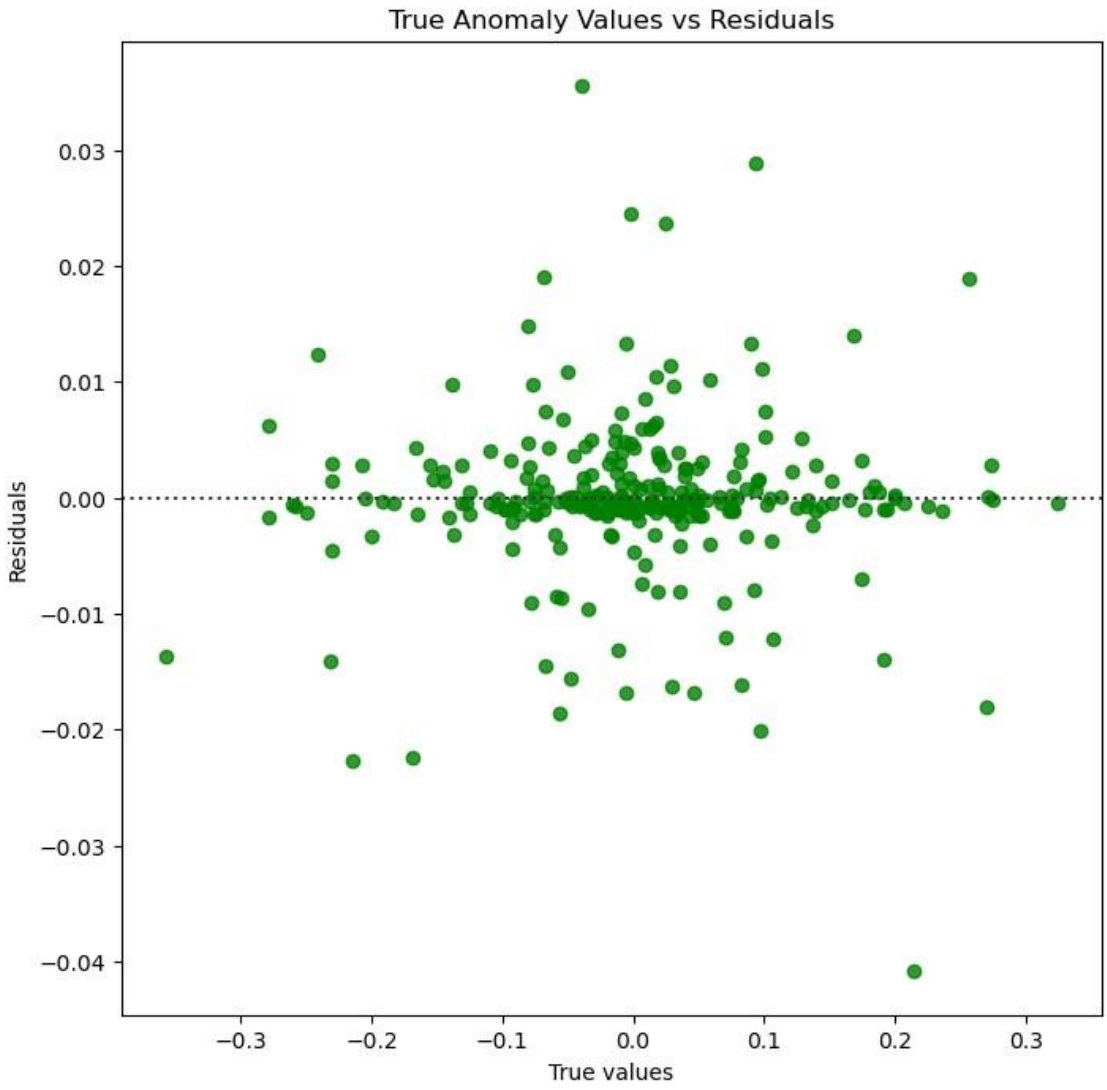


Residual Plot

Having seen the calculations and the plot of true vs predicted anomaly values, I expected the residual plot to appear worse than it does. A well-fitting model should show a random dispersion of residuals on both sides of the horizontal axis in the residuals plot. Ours appears to do just that, but based on the other information I've gathered above I don't believe I have achieved a good result with random forest.

[6584]:
```
plt.figure(figsize =(8,8))
sns.residplot(y_test, y_pred2, color    ='green')
plt.xlabel( 'True values ')
plt.ylabel( 'Residuals ')
plt.title( 'True Anomaly Values vs Residuals    ')
plt.show()
```

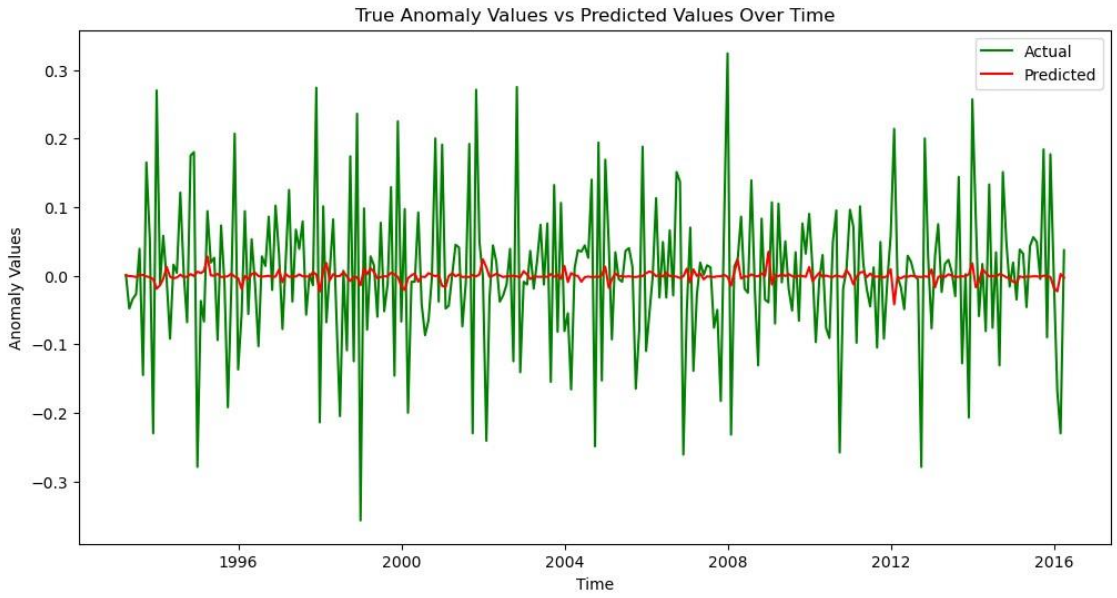
```
c:\Users\hashl\anaconda3\lib\site-packages\seaborn\_decorators.py:36:
FutureWarning:

Pass the following variables as keyword args: x, y. From version 0.12, the only
valid positional argument will be `data`, and passing other arguments without
an explicit keyword will result in an error or misinterpretation.
```



True vs Predicted Values over time

We can easily see in this graph that the model failed to capture the trends in anomaly values.

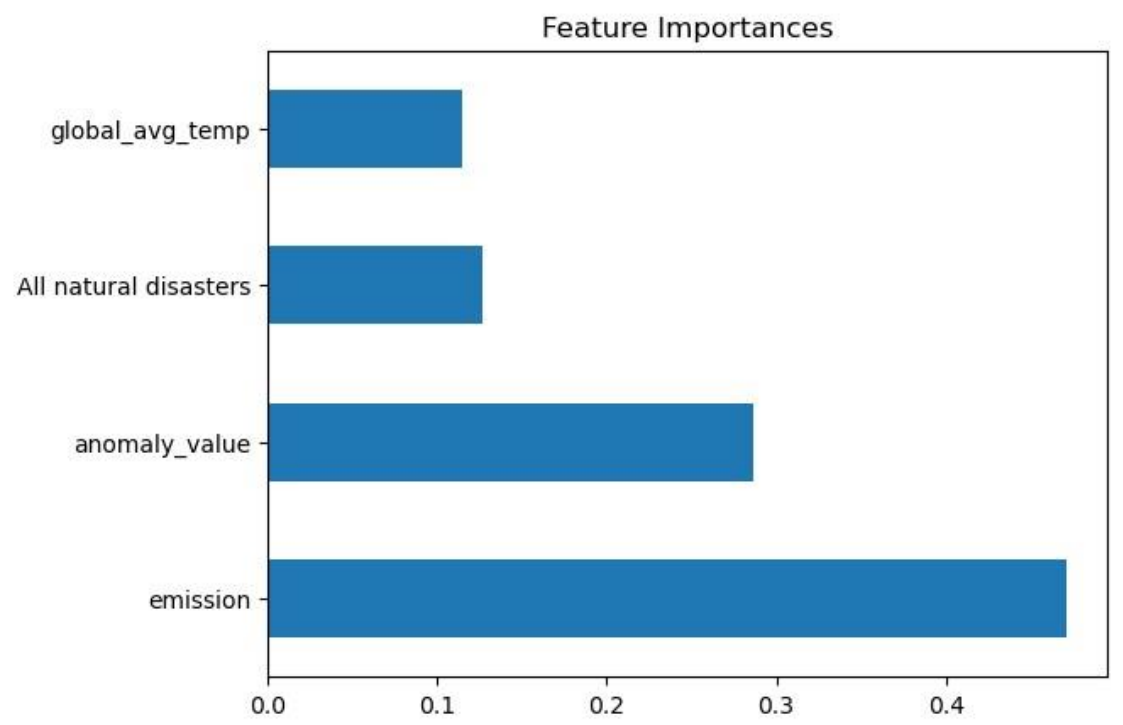
[6585]:
```
plt.figure(figsize=(12, 6)) plt.plot(y_test.index, y_test,
label='Actual', color='green') plt.plot(y_test.index,
y_pred2, label='Predicted', color='red') plt.title('True
Anomaly Values vs Predicted Values Over Time')
plt.xlabel('Time') plt.ylabel('Anomaly Values') plt.legend()
plt.show()
```



Feature Importances

Surprisingly, emission is ranked as most important. This must be because it underwent two stages of differencing before being input to the model. Now being the most stationary of all variables, it has become most useful to the model.

[6586]:
```
feat_importances = pd.Series(model2.feature_importances_,
index=X.columns) feat_importances.nlargest(4).plot(kind='barh')
plt.title('Feature Importances') plt.show()
```



Evaluation Scores

Because we have differenced our data (I was not able to reverse the differencing), the values we are comparing are differenced values, and thus are on a smaller scale than the original values which we evaluated in the multiple linear regression model. It doesn't bode well then that the MAE, MSE, and RMSE scores are roughly the same as for multiple linear regression.

The r2 is very disappointing. Being a negative value, it shows a very poor fit of the data to the model.

[6587]:
```
Evaluate(y_test,y_pred2)
```

```
Mean Absolute Error: 0.08013945611567982
Mean Squared Error: 0.011781059938520718
Root Mean Squared Error: 0.28308913104476446
R Squared: -0.007625419784970111
```

Conclusions and Reflections

Unfortunately this model isn't producing good results. I repeatedly tuned the hyperparameters of n_estimators, max_depth, and max_samples by increasing or decreasing them in various configurations, but I could not achieve better results. The negative r2 indicates that the model is better-fitted to a horizontal line across the mean y value.

I was not able to reverse the differencing, which probably had some effect on the evaluation scores, however, since all data was differenced the predictions should still be comparable with the y_test set. I can see clearly in the plot of true vs. predicted anomaly values over time that the data is stationary.

In a previous iteration I did not difference the data to make it stationary, and the r2 was -200%, which is terrible and not interpretable. It does appear that the model was able to handle stationary data slightly better, but I will conclude that random forest regression isn't well suited to time series forecasting t+3 in the future, at least in this particular dataset. It is true that random forest regressor models are not designed to be sensitive to temporally-dependent data, but with proper data preparation this can sometimes be overcome. For the future I would not choose random forest as my model for a time series forecasting problem, particularly if I am forecasting more than one step into the future. I expect models with built-in abilities to handle long-term dependencies in data such as LSTM will perform better on our problem.

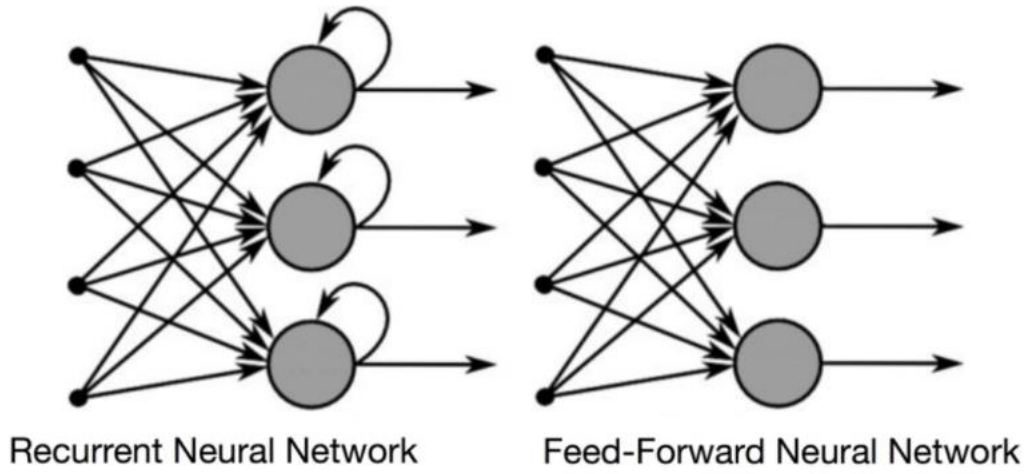
6 LSTM Model

LSTM is the acronym for "long short-term memory". It is a type of recurrent neural network (RNN). It is ideal for the task at hand, as it is capable of learning temporal or sequence dependencies in prediction problems.

"Long short term memory" can be compared to human memory. Humans are able to remember things long-term, but we do not keep a memory record of every insignificant thing that has happened to us. We tend to only remember the more significant things in the long-term. LSTM models work in a similar way. They do not 'remember' everything long term, but instead are able to filter out noise by assigning weights to important bits of information and storing only those long-term. This is a unique capability that is not afforded by many other types of models.

To explain further, I will explore what a recurrent neural network is. RNNs are algorithms designed for sequential data as they are the only type of algorithm with an internal memory. They were derived from feed-forward neural networks, which work by sending information in only a forward direction from the input layers to the hidden layers and out. Because feed-forward networks only move in one direction, they don't have a mechanism for memory as they only recieve information from new inputs. Recurrent neural networks on the other hand send new inputs forward from the input layer through the hidden layers, and then the outputs are cycled back through the hidden layers. In this way, RNNs learn simultaneously from both new inputs and from the outputs of previous inputs.

Below is an illustration of a recurrent neural network and a feed-forward neural network showing the difference in the flow of information.



https://builtin.com/data-science/recurrent-neural-networks-and-lstm

This function of cycling inputs back through the hidden layers afford RNNs a short term memory. LSTMs add an additional function which gives them a long term memory.

Backpropagation through time is an important concept in understanding the advantage of LSTM models. It is an algorithm that runs backwards through the neural network finding partial derivatives of error with respect to weight, which are then used by a gradient descent algorithm to adjust the loss function applied to the weights - effectively finding the optimal weights which minimize the total loss of the neural network.

This process can lead to two major issues called the vanishing gradient problem and the exploding gradient problem. The vanishing gradient problem occurs when backpropagation produces a gradient that is 'vanishingly' small, which in turn is used to adjust the weights by such a small amount that the machine isn't actually learning as it should. The exploding gradient problem is the opposite situation, in which the gradients are so large that the weights are adjusted much higher than they deserve. The result is the same - the total loss is not minimized and the machine is not actually learning as it should. Since these problems usually occur in the early layers of the network (because backpropagation works from back to front), the machine is unable to 'remember' long-term dependencies.

The way that LSTM tackles these problems is through a specialized memory cell. The memory cell has the ability to accept new inputs, allow its stored information to affect outputs at the current timestep, or to discard (forget) some information through three different 'gates', which are opened or closed on the basis of weights assigned by a sigmoid function (ranging 0 to 1). The gates update at every time step, which allows for greater 'control' and prevents the vanishing or exploding gradient problem from occuring.

Given LSTM models' unique capabilities in recognizing temporal patterns and their dynamic ability to assign importance throughout information flow and reduce noise, they seem like an ideal choice for our task of forescasting on a time series.

https://www.youtube.com/watch?v=qO_NLVjD6zE

https://machinelearningmastery.com/gentle-introduction-long-short-term-memory-networksexperts/

[6588]:
```
df_LSTM .head()
```

[6588]:
```
                emission anomaly_value All natural disasters \
date
 1901-01-31 4.482107e+10         -0.075                    5.0
 1901-02-28 4.482107e+10         -0.176                    5.0
 1901-03-31 4.482107e+10         -0.272                    5.0
 1901-04-30 4.482107e+10         -0.236                    5.0
 1901-05-31 4.482107e+10         -0.187                    5.0
            global_avg_temp
date
 1901-01-31           -0.28
 1901-02-28           -0.06
1901-03-31 0.04 1901-04-30 -0.06
 1901-05-31           -0.17
```

Below I use a for loop to generate a special dataframe with new time-based features. These are copies of the independent features at t-1, t-2, and t-3, while the target feature is at present. This is a special way of framing a time series for multi-step forecasting, as I read in https://machinelearningmastery.com/convert-time-series-supervised-learningproblem-python/. I was unsure if this is the appropriate way to frame the data. I worried about introducing multicollinearity. However, I learned that LSTM models and neural networks in general are able to handle multicollinearity since the data is fitted using backpropagation https://datascience.stackexchange.com/questions/28328/how-does-multicollinearity-affectneural-networks. This style of data framing would likely introduce a multicollinearity problem to the multiple linear regression model.

[6589]:
```
features  = df_LSTM[[ 'emission ','All natural disasters  ','global_avg_temp  ',␣
 ↪'anomaly_value ']]
window_size  = 3

for feature  in features:
    for i in range (1, window_size +1):
        df_LSTM[ f'{feature }_lag-{i}'] = df_LSTM[feature]  .shift(i)

df_LSTM  = df_LSTM .dropna()

df_LSTM .head()
```

[6589]:
```
                emission anomaly_value All natural disasters \
date
 1901-04-30 4.482107e+10         -0.236                    5.0
 1901-05-31 4.482107e+10         -0.187                    5.0
 1901-06-30 4.482107e+10         -0.196                    5.0
 1901-07-31 4.482107e+10         -0.146                    5.0
 1901-08-31 4.482107e+10         -0.188                    5.0
             global_avg_temp emission_lag-1 emission_lag-2 emission_lag-3 \
date
 1901-04-30           -0.06   4.482107e+10   4.482107e+10   4.482107e+10
 1901-05-31           -0.17   4.482107e+10   4.482107e+10   4.482107e+10
 1901-06-30           -0.10   4.482107e+10   4.482107e+10   4.482107e+10
 1901-07-31           -0.09   4.482107e+10   4.482107e+10   4.482107e+10
 1901-08-31           -0.13   4.482107e+10   4.482107e+10   4.482107e+10
```

```
            All natural disasters_lag-1 All natural disasters_lag-2 \
date
1901-04-30                          5.0                          5.0
1901-05-31                          5.0                          5.0
1901-06-30                          5.0                          5.0
1901-07-31                          5.0                          5.0
1901-08-31                          5.0                          5.0

            All natural disasters_lag-3 global_avg_temp_lag-1 \
date
1901-04-30                          5.0                  0.04
1901-05-31                          5.0                 -0.06
1901-06-30                          5.0                 -0.17
1901-07-31                          5.0                 -0.10
1901-08-31                          5.0                 -0.09

            global_avg_temp_lag-2 global_avg_temp_lag-3 anomaly_value_lag-1 \
date
1901-04-30                 -0.06                 -0.28              -0.272
1901-05-31                  0.04                 -0.06              -
                                                                    0.236
1901-06-30                 -0.06                  0.04              -
                                                                    0.187
1901-07-31                 -0.17                 -0.06              -
                                                                    0.196
1901-08-31                 -0.10                 -0.17              -
                                                                    0.146

            anomaly_value_lag-2 anomaly_value_lag-3
date
1901-04-30              -0.176              -0.075

1901-05-31              -0.272              -0.176

1901-06-30              -0.236              -0.272

1901-07-31              -0.187              -0.236

1901-08-31              -0.196              -0.187
```

Below I assign the independent features to X and the target feature to y.

```
[6590]: features = ['emission_lag-1', 'emission_lag-2', 'emission_lag-3',
                    'All natural disasters_lag-1', 'All natural disasters_lag-2',
                    'All natural disasters_lag-3', 'global_avg_temp_lag-1',
                    'global_avg_temp_lag-2', 'global_avg_temp_lag-3', 'anomaly_value_lag-1',
                    'anomaly_value_lag-2', 'anomaly_value_lag-3']
```

```
df_LSTM['anomaly_value_target'] = df_LSTM['anomaly_value'] X =

df_LSTM[features] #assigning independent features to X y =

df_LSTM['anomaly_value_target'] #assigning the target column to y

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

```
(1384, 12)
(1384,)
```

Next, I scale the data using the min max scaling function. This adjusts all data to be within the minimum and maximum values. For LSTM and other neural networks, scaling the data is helpful because it helps the model to converge faster and uses less resources. I scale both the X and y data. For scaling, y must be 2 dimensional instead of 1 dimensional and so I reshape it using the reshape function.

[6591]:
```
X_scaler  = MinMaxScaler(feature_range  =(0,1))
y_scaler  = MinMaxScaler(feature_range  =(0,1))

X = X_scaler .fit_transform(X)
y = y_scaler .fit_transform(y  .values .reshape( -1,1))
```

For input to the LSTM model, input must be in the shape of a 3D tensor with dimensions (batch size, time steps, and number of features). Hence, I reshape the X data to comply.

[6592]:
```
X = X.reshape(X .shape[ 0],1,X.shape[ 1]) #(batch size = 1384, time steps = 1,
  ↪number of features = 12)

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

```
(1384, 1, 12)
(1384, 1)
```

Train Valid Test Split

Here we use the train test split function to divide data into different sets. I am using an 80/10/10 split for train/valid/and test sets.

[6593]:
```
VALID_AND_TEST_SIZE   = 0.1

X_train, X_else, y_train, y_else     = train_test_split(X, y,
  ↪test_size =VALID_AND_TEST_SIZE  *2, shuffle =False )
X_valid, X_test, y_valid, y_test     = train_test_split(X_else, y_else, test_size     =0.
  ↪5, shuffle =False )

print (X_train .shape)
print (X_valid .shape)
print (X_test .shape)
print (y_train .shape)
print (y_valid .shape)
print (y_test .shape)
```

```
(1107, 1, 12)
(138, 1, 12)
(139, 1, 12)
(1107, 1)
(138, 1)
(139, 1)
```

Constructing and compiling the LSTM Model

First, we instantiate the model. The Sequential() LSTM algorithm we are using comes from the Keras library.

Next, we add layers to the model. First, we must specify the input shape as input_shape = (time steps, number of features) in the LSTM layer. Return_sequences must be set to True if it is before a dense layer, since it produces a 3D output and Dense layers only accepts 3D input. If it is set to false, it will return only the last item in the sequence of outputs from each time step input. The number in the LSTM layer is the number of units, which means the size of the dimensionality of the output of the layer. This, as all hyperparameters, can be adjusted through trial and error.

A dense layer is called a fully connected layer, because every neuron recieves input from every neuron in the prior layer. The dense layers' purpose is to learn the more complex, non-linear relationships from the input data.

For good measure, I added another LSTM layer and another dense layer. I want to add some complexity, but not too much.

Finally, I compile the model using a loss function and an optimizer. I used mean squared error because it is a reliable and widely used loss function due to the ease with which it is computed and differentiated by the algorithm. I chose the optimizer Adam because of its reliably high performance and efficiency.

[6594]:
```
model3  = Sequential()

model3 .add(LSTM( 150, return_sequences  =True, input_shape  =(1,12)))
model3 .add(Dense( 1))
model3 .add(LSTM( 75))
model3 .add(Dense( 1))

model3 .compile(loss ='mean_squared_error  ', optimizer ='adam ')

model3 .summary()
```

```
Model: "sequential_120"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm_240 (LSTM)             (None, 1, 150)            97800

 dense_239 (Dense)           (None, 1, 1)              151

 lstm_241 (LSTM)             (None, 75)                23100

 dense_240 (Dense)           (None, 1)                 76
=================================================================
Total params: 121,127
Trainable params: 121,127
Non-trainable params: 0
_________________________________________________________________
```

Here, I fit the model with the training and validation data.

There are a few hyperparameters which can be tuned in this stage, including batch size, number of epochs, verbosity, and shuffle. I experimented with different configurations to achieve the best results I could.

[6595]:
```
history = model3.fit(X_train,y_train, batch_size = 100,␣
↪validation_data=(X_valid,y_valid), epochs=75, verbose=1, shuffle=False)
```

```
Epoch 1/75
12/12 [==============================] - 4s 61ms/step - loss: 0.1262 -
val_loss:
0.2982
Epoch 2/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0816 - val_loss:
0.1769
Epoch 3/75
12/12 [==============================] - 0s 8ms/step - loss: 0.0340 - val_loss:
0.0434
Epoch 4/75
12/12 [==============================] - 0s 7ms/step - loss: 0.0061 - val_loss:
0.0053
Epoch 5/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0066 - val_loss:
0.0042
Epoch 6/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0057 - val_loss:
0.0056
Epoch 7/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0053 - val_loss:
0.0040
Epoch 8/75
12/12 [==============================] - 0s 7ms/step - loss: 0.0049 - val_loss:
0.0031
Epoch 9/75
```

```
12/12 [==============================] - 0s 6ms/step - loss: 0.0050 - val_loss:
0.0032
Epoch 10/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0049 - val_loss:
0.0032
Epoch 11/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0048 - val_loss:
0.0030
Epoch 12/75
12/12 [==============================] - 0s 7ms/step - loss: 0.0047 - val_loss:
0.0029
Epoch 13/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0046 - val_loss:
0.0029
Epoch 14/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0045 - val_loss:
0.0028
Epoch 15/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0044 - val_loss:
0.0027
Epoch 16/75
12/12 [==============================] - 0s 7ms/step - loss: 0.0043 - val_loss:
0.0027
Epoch 17/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0042 - val_loss:
0.0026
Epoch 18/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0041 - val_loss:
0.0026
Epoch 19/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0040 - val_loss:
0.0026
Epoch 20/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0039 - val_loss:
0.0025
Epoch 21/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0038 - val_loss:
0.0025
Epoch 22/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0037 - val_loss:
0.0025
Epoch 23/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0036 - val_loss:
0.0024
Epoch 24/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0035 - val_loss:
0.0024
Epoch 25/75
12/12 [==============================] - 0s 7ms/step - loss: 0.0035 - val_loss:
0.0024
```

```
Epoch 26/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0034 - val_loss:
0.0024
Epoch 27/75
12/12 [==============================] - 0s 7ms/step - loss: 0.0033 - val_loss:
0.0024
Epoch 28/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0032 - val_loss:
0.0024
Epoch 29/75
12/12 [==============================] - 0s 7ms/step - loss: 0.0031 - val_loss:
0.0024
Epoch 30/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0030 - val_loss:
0.0024
Epoch 31/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0029 - val_loss:
0.0025
Epoch 32/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0028 - val_loss:
0.0025
Epoch 33/75
12/12 [==============================] - 0s 7ms/step - loss: 0.0028 - val_loss:
0.0025
Epoch 34/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0027 - val_loss:
0.0025
Epoch 35/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0026 - val_loss:
0.0025
Epoch 36/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0026 - val_loss:
0.0025
Epoch 37/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0025 - val_loss:
0.0025
Epoch 38/75
12/12 [==============================] - 0s 8ms/step - loss: 0.0025 - val_loss:
0.0026
Epoch 39/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0024 - val_loss:
0.0026
Epoch 40/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0024 - val_loss:
0.0026
Epoch 41/75
12/12 [==============================] - 0s 7ms/step - loss: 0.0023 - val_loss:
0.0026
Epoch 42/75
12/12 [==============================] - 0s 7ms/step - loss: 0.0023 - val_loss:
```

```
0.0026
Epoch 43/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0023 - val_loss:
0.0026
Epoch 44/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0022 - val_loss:
0.0026
Epoch 45/75
12/12 [==============================] - 0s 9ms/step - loss: 0.0022 - val_loss:
0.0026
Epoch 46/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0022 - val_loss:
0.0026
Epoch 47/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0021 - val_loss:
0.0026
Epoch 48/75
12/12 [==============================] - 0s 7ms/step - loss: 0.0021 - val_loss:
0.0026
Epoch 49/75
12/12 [==============================] - 0s 7ms/step - loss: 0.0021 - val_loss:
0.0026
Epoch 50/75
12/12 [==============================] - 0s 7ms/step - loss: 0.0021 - val_loss:
0.0026
Epoch 51/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0020 - val_loss:
0.0026
Epoch 52/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0020 - val_loss:
0.0026
Epoch 53/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0020 - val_loss:
0.0026
Epoch 54/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0020 - val_loss:
0.0025
Epoch 55/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0020 - val_loss:
0.0025
Epoch 56/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0020 - val_loss:
0.0025
Epoch 57/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0019 - val_loss:
0.0025
Epoch 58/75
12/12 [==============================] - 0s 7ms/step - loss: 0.0019 - val_loss:
0.0025
Epoch 59/75
```

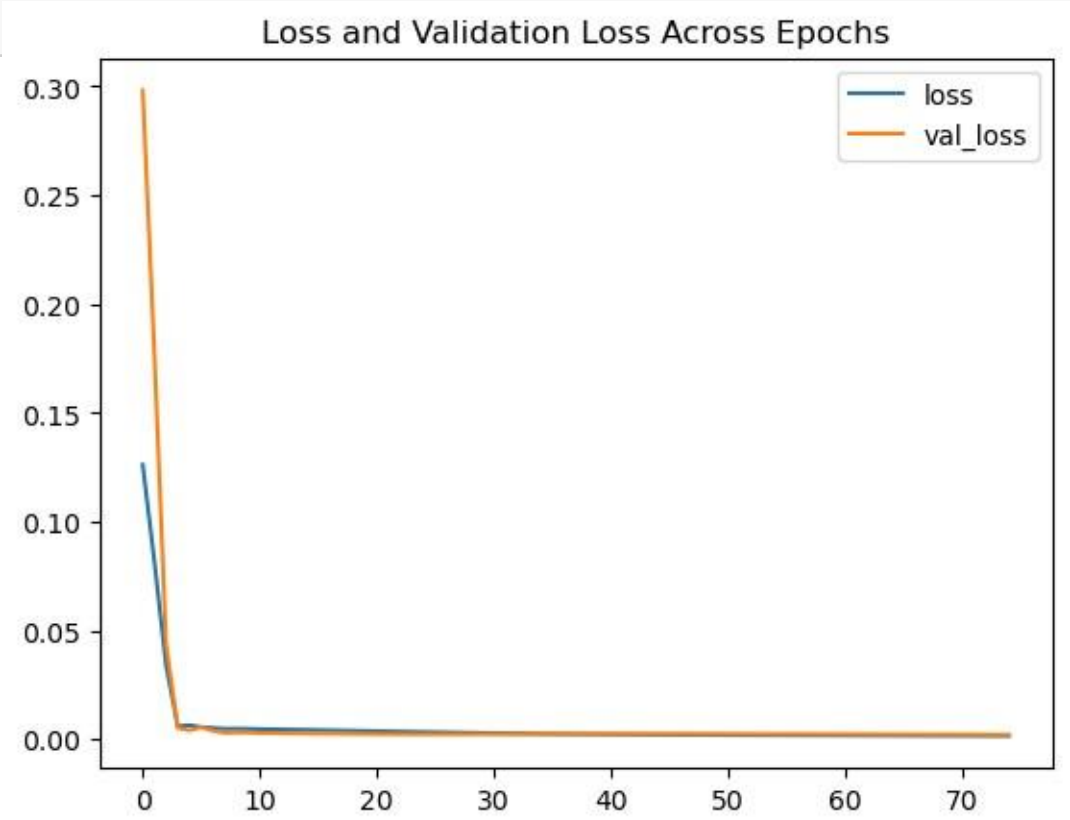
```
12/12 [==============================] - 0s 6ms/step - loss: 0.0019 - val_loss:
0.0025
Epoch 60/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0019 - val_loss:
0.0025
Epoch 61/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0019 - val_loss:
0.0024
Epoch 62/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0018 - val_loss:
0.0024
Epoch 63/75
12/12 [==============================] - 0s 7ms/step - loss: 0.0018 - val_loss:
0.0024
Epoch 64/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0018 - val_loss:
0.0024
Epoch 65/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0018 - val_loss:
0.0024
Epoch 66/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0018 - val_loss:
0.0024
Epoch 67/75
12/12 [==============================] - 0s 7ms/step - loss: 0.0018 - val_loss:
0.0023
Epoch 68/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0017 - val_loss:
0.0023
Epoch 69/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0017 - val_loss:
0.0023
Epoch 70/75
12/12 [==============================] - 0s 8ms/step - loss: 0.0017 - val_loss:
0.0023
Epoch 71/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0017 - val_loss:
0.0023
Epoch 72/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0017 - val_loss:
0.0023
Epoch 73/75
12/12 [==============================] - 0s 7ms/step - loss: 0.0017 - val_loss:
0.0022
Epoch 74/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0017 - val_loss:
0.0022
Epoch 75/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0016 - val_loss:
0.0022
```

Visualizations

Below we can see a plot of the loss and validation loss over the epochs, or the 'learning curve'. Ours shows great results, since the loss drops quickly from the first epoch and reaches stability near zero. There is little to no gap with the validation loss. This likely means our model was a good fit.

https://machinelearningmastery.com/learning-curves-for-diagnosing-machine-learning-modelperformance/

[6596]:
```
import matplotlib.pyplot as plt
plt.plot(history.history["loss"], label="loss")
plt.plot(history.history["val_loss"],
label="val_loss") plt.title('Loss and Validation
Loss Across Epochs') plt.legend() plt.show()
```



Predicting and Reversing Scaling

Below I generate predictions and use the min max scaler's inverse transform function to reverse the scaling of the predicted values and the y_test set. This way we can better understand and evaluate the results.

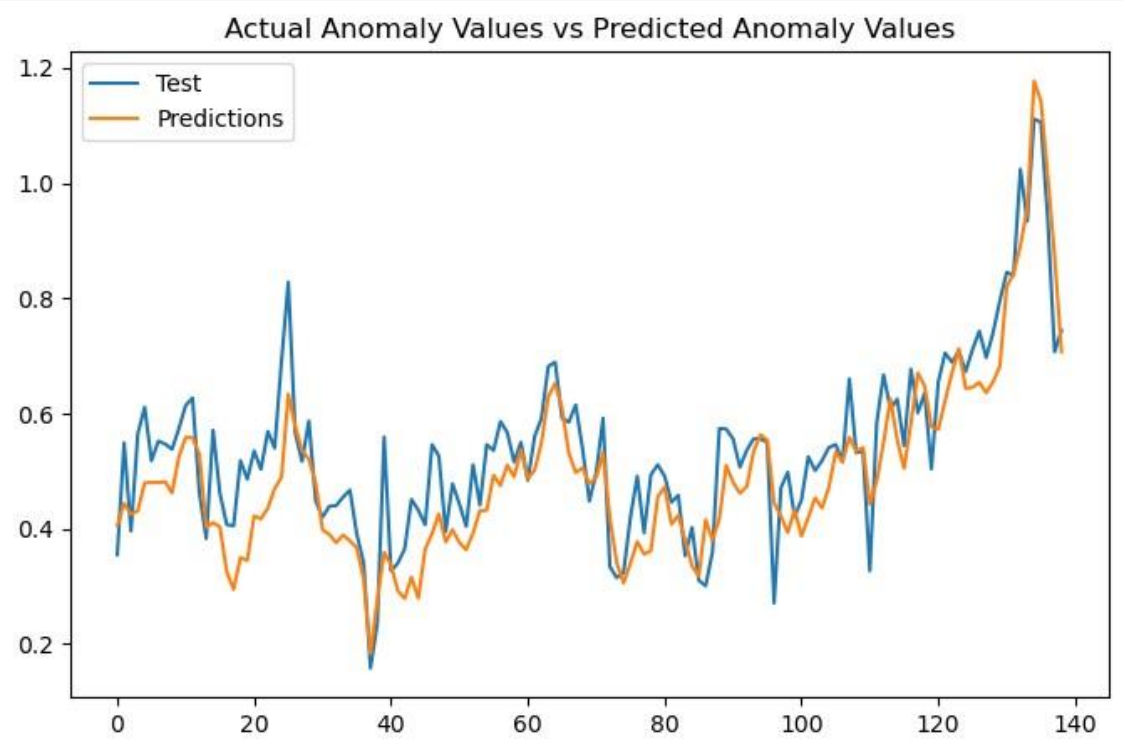
[6597]:
```
y_pred3 = model3.predict(X_test)
```

```
5/5 [==============================] - 1s 3ms/step
```

[6598]:
```
unscaled_predictions = y_scaler.inverse_transform(y_pred3)
unscaled_y_test = y_scaler.inverse_transform(y_test)
```

In this plot we can see the test values and the predicted values together. It isn't perfect, but there is a fairly close alignment to the patterns of the test data.

[6599]:
```
plt.figure(figsize=(8,5)) plt.plot(list(unscaled_y_test),
label="Test") plt.plot(list(unscaled_predictions),
label="Predictions") plt.title('Actual Anomaly Values vs
Predicted Anomaly Values') plt.legend() plt.show()
```



[6600]:
```
Evaluate(unscaled_y_test,unscaled_predictions)
```

```
Mean Absolute Error: 0.06618145640891233
Mean Squared Error: 0.006568072283636186
Root Mean Squared Error: 0.2572575682247508
R Squared: 0.7229236466459645
```

Conclusions and Reflections

I am happy with the performance of the model. The r2 value is above 70%, indicating a good fit, and the errors are much smaller than the baseline scores of the multiple linear regression model. This means that our model was able to predict anomaly values with good accuracy.

I tuned the hyperparameters over and over until I achieved this result. I began by adjusting the LSTM units of the LSTM layers. I began with a low number, and found that when I increased the number of units the results improved dramatically. However, once I set the first layer to 200 units and the second to 100, I found that the performance declined quite a bit. This is likely due to overfitting. While adding units enhances the ability of the model to identify complex relationships between data, adding too many units trains the machine too well on the particular dataset

and decreases its performance when it is presented with new data (overfitting). In the end, I settled on 150 units in the first layer and 75 in the second as I found this to give the best results.

I also tuned the batch size and the number of epochs. A smaller batch size seemed to perform better, but smaller than 100 was detrimental. Also, the bigger the batch size the slower the training.

The number of epochs at 75 seemed to produce the best results. Anything above 75 seemed to lead

to overfitting and a decrease in performance. Under 75 epochs did not perform as well, so I settled at 75.

To follow up on my concerns about the data framing for input to this model, I am unsure if the model is solving the assigned task. The task requires a prediction of anomaly value t+3 in the future from an input sample with the latest datapoint at present, t. With the data structure I used for this model, there are input features at only 1 or 2 lags from the target. Because of this, I am not sure I can interpret the predicted values as being forecasted for 3 months ahead. Because of my doubts, I ran another LSTM with the target set at t+3 the closest input feature to compare results.

7 LSTM with a Modified Input Data Structure

Since the point of this project is to forecast anomaly value 3 months in the future from the latest input value, I adjusted the input data structure of this model to try to reflect that goal. Besides that, I have kept all else the same including the scaling, the structure of the model, and the settings of the hyperparameters. I expect by introducing this time gap, this model will produce worse results than the other LSTM.

[6601]:
```
df_LSTM2 .head()
```

[6601]:
```
                emission anomaly_value All natural disasters \
date
 1901-01-31 4.482107e+10         -0.075                    5.0
 1901-02-28 4.482107e+10         -0.176                    5.0
 1901-03-31 4.482107e+10         -0.272                    5.0
 1901-04-30 4.482107e+10         -0.236                    5.0
 1901-05-31 4.482107e+10         -0.187                    5.0
             global_avg_temp
date
 1901-01-31            -0.28
 1901-02-28            -0.06
1901-03-31 0.04 1901-04-30 -0.06
 1901-05-31            -0.17
```

Creating the input data

With the code below I created new columns for each feature with t-1 and t-2 lags. I will also keep the feature columns at present t as input values. The target will be shifted t+3 to the future, so that the latest input value is 3 lags behind the target.

[6602]:
```
features  = df_LSTM2[[ 'emission ','All natural disasters  ','global_avg_temp  ',␣
  ↪'anomaly_value ']]
window_size  = 3

for feature  in features:
    for i in range (1, window_size):
```

```
        df_LSTM2[ f'{feature }_lag-{i}'] = df_LSTM2[feature]  .shift(i)

df_LSTM2  = df_LSTM2 .dropna()

df_LSTM2 .head()
```

[6602]:
```
                emission anomaly_value All natural disasters \
date
1901-03-31 4.482107e+10        -0.272                    5.0
1901-04-30 4.482107e+10        -0.236                    5.0
1901-05-31 4.482107e+10        -0.187                    5.0
1901-06-30 4.482107e+10        -0.196                    5.0
1901-07-31 4.482107e+10        -0.146                    5.0
            global_avg_temp emission_lag-1 emission_lag-2 \
date
1901-03-31 0.04  4.482107e+10      4.482107e+10 1901-04-30 -0.06
     4.482107e+10      4.482107e+10
1901-05-31           -0.17    4.482107e+10    4.482107e+10
1901-06-30           -0.10    4.482107e+10    4.482107e+10
1901-07-31           -0.09    4.482107e+10    4.482107e+10

            All natural disasters_lag-1 All natural disasters_lag-2 \
date
1901-03-31                         5.0                         5.0
1901-04-30                         5.0                         5.0
1901-05-31                         5.0                         5.0
1901-06-30                         5.0                         5.0
1901-07-31                         5.0                         5.0

            global_avg_temp_lag-1 global_avg_temp_lag-2 anomaly_value_lag-1 \
date
      1901-03-31     -0.06 -0.28 -0.176 1901-04-30 0.04  -0.06 -0.272
1901-05-31                 -0.06                  0.04              -0.236
1901-06-30                 -0.17                 -0.06              -0.187
1901-07-31                 -0.10                 -0.17              -0.196

            anomaly_value_lag-2
date
1901-03-31              -0.075
1901-04-30              -0.176
1901-05-31              -0.272
1901-06-30              -0.236
1901-07-31              -0.187
```

[6603]:
```
features = ['emission','emission_lag-1',␣
↪'emission_lag-2','anomaly_value','anomaly_value_lag-1',
            'anomaly_value_lag-2', 'All natural disasters','All natural␣
↪disasters_lag-1',
            'All natural␣
↪disasters_lag-2','global_avg_temp','global_avg_temp_lag-1','global_avg_temp_lag-2']
```

```
df_LSTM2['anomaly_value_target'] = df_LSTM2['anomaly_value'].shift(-3)␣
 ↪#creating a target column shifted ahead by 3 df_LSTM2 =

df_LSTM2.dropna() #dropping the resulting null values X =

df_LSTM2[features] #assigning independent features to X y =

df_LSTM2['anomaly_value_target'] #assigning the target column to y

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

```
(1382, 12)
(1382,)
```

Scaling the data

[6604]:
```
X_scaler  = MinMaxScaler(feature_range  =(0,1))
y_scaler  = MinMaxScaler(feature_range  =(0,1))

X = X_scaler .fit_transform(X)
y = y_scaler .fit_transform(y  .values .reshape( -1,1))
```

[6605]:
```
X = X.reshape(X .shape[ 0],1,X.shape[ 1])

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

```
(1382, 1, 12)
(1382, 1)
```

Train - Valid - Test Split

[6606]:
```
VALID_AND_TEST_SIZE   = 0.1

X_train, X_else, y_train, y_else     = train_test_split(X, y,  ␣
 ↪test_size =VALID_AND_TEST_SIZE  *2, shuffle =False )
X_valid, X_test, y_valid, y_test     = train_test_split(X_else, y_else, test_size    =0.
 ↪5, shuffle =False )

print (X_train .shape)
print (X_valid .shape)
```

```
print (X_test .shape)
print (y_train .shape)
print (y_valid .shape)
print (y_test .shape)
```

```
(1105, 1, 12)
(138, 1, 12)
(139, 1, 12)
(1105, 1)
```

```
(138, 1)
(139, 1)
```

Constructing and Compiling the Model

[6607]:
```
model3  = Sequential()

model3 .add(LSTM( 150, return_sequences  =True , input_shape =(1,12)))
model3 .add(Dense( 1))
model3 .add(LSTM( 75))
model3 .add(Dense( 1))

model3 .compile(loss ='mean_squared_error  ', optimizer ='adam ')

model3 .summary()
```

```
Model: "sequential_121"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm_242 (LSTM)             (None, 1, 150)            97800

 dense_241 (Dense)           (None, 1, 1)              151

 lstm_243 (LSTM)             (None, 75)                23100

 dense_242 (Dense)           (None, 1)                 76
=================================================================
Total params: 121,127
Trainable params: 121,127
Non-trainable params: 0
_________________________________________________________________
```

Fitting the Model

[6608]:
```
history = model3.fit(X_train,y_train, batch_size = 100,␣
 ↪validation_data=(X_valid,y_valid), epochs=75, verbose=1, shuffle=False)
```

```
Epoch 1/75
12/12 [==============================] - 4s 64ms/step - loss: 0.1278 -
val_loss:
0.3079
Epoch 2/75
12/12 [==============================] - 0s 8ms/step - loss: 0.0844 - val_loss:
0.1908
Epoch 3/75
12/12 [==============================] - 0s 7ms/step - loss: 0.0364 - val_loss:
0.0525
Epoch 4/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0065 - val_loss:
0.0049
Epoch 5/75
```

```
12/12 [==============================] - 0s 6ms/step - loss: 0.0079 - val_loss:
0.0052
Epoch 6/75
12/12 [==============================] - 0s 8ms/step - loss: 0.0064 - val_loss:
0.0056
Epoch 7/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0058 - val_loss:
0.0039
Epoch 8/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0056 - val_loss:
0.0035
Epoch 9/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0057 - val_loss:
0.0034
Epoch 10/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0056 - val_loss:
0.0034
Epoch 11/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0055 - val_loss:
0.0035
Epoch 12/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0055 - val_loss:
0.0035
Epoch 13/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0054 - val_loss:
0.0035
Epoch 14/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0054 - val_loss:
0.0036
Epoch 15/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0054 - val_loss:
0.0036
Epoch 16/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0053 - val_loss:
0.0036
Epoch 17/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0053 - val_loss:
0.0036
Epoch 18/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0052 - val_loss:
0.0036
Epoch 19/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0052 - val_loss:
0.0036
Epoch 20/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0052 - val_loss:
0.0036
Epoch 21/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0051 - val_loss:
0.0036
```

```
Epoch 22/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0051 - val_loss:
0.0036
Epoch 23/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0051 - val_loss:
0.0036
Epoch 24/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0050 - val_loss:
0.0035
Epoch 25/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0050 - val_loss:
0.0035
Epoch 26/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0050 - val_loss:
0.0035
Epoch 27/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0049 - val_loss:
0.0035
Epoch 28/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0049 - val_loss:
0.0035
Epoch 29/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0049 - val_loss:
0.0035
Epoch 30/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0048 - val_loss:
0.0035
Epoch 31/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0048 - val_loss:
0.0034
Epoch 32/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0048 - val_loss:
0.0034
Epoch 33/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0047 - val_loss:
```

```
0.0034
Epoch 34/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0047 -
val_loss:
0.0034
Epoch 35/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0047 -
val_loss:
0.0034
Epoch 36/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0047 -
val_loss:
0.0034
Epoch 37/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0046 -
val_loss:
0.0034
Epoch 38/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0046 -
val_loss:
0.0034
Epoch 39/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0046 -
val_loss:
0.0034
Epoch 40/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0046 -
val_loss:
0.0034
Epoch 41/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0046 -
val_loss:
0.0034
Epoch 42/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0045 -
val_loss:
0.0034
Epoch 43/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0045 -
val_loss:
0.0034
Epoch 44/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0045 -
val_loss:
0.0034
Epoch 45/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0045 -
val_loss:
0.0034
Epoch 46/75
```

```
0.0034
12/12 [==============================] - 0s 6ms/step - loss: 0.0045 -
val_loss:
0.0034
Epoch 47/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0045 -
val_loss:
0.0034
Epoch 48/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0044 -
val_loss:
0.0034
Epoch 49/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0044 -
val_loss:
Epoch 50/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0044 -
val_loss:
0.0034
Epoch 51/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0044 -
val_loss:
0.0034
Epoch 52/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0044 -
val_loss:
0.0034
Epoch 53/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0044 -
val_loss:
0.0034
Epoch 54/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0044 -
val_loss:
0.0034
Epoch 55/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0043 -
val_loss:
0.0034
Epoch 56/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0043 -
val_loss:
0.0034
Epoch 57/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0043 -
val_loss:
0.0034
Epoch 58/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0043 -
val_loss:
```

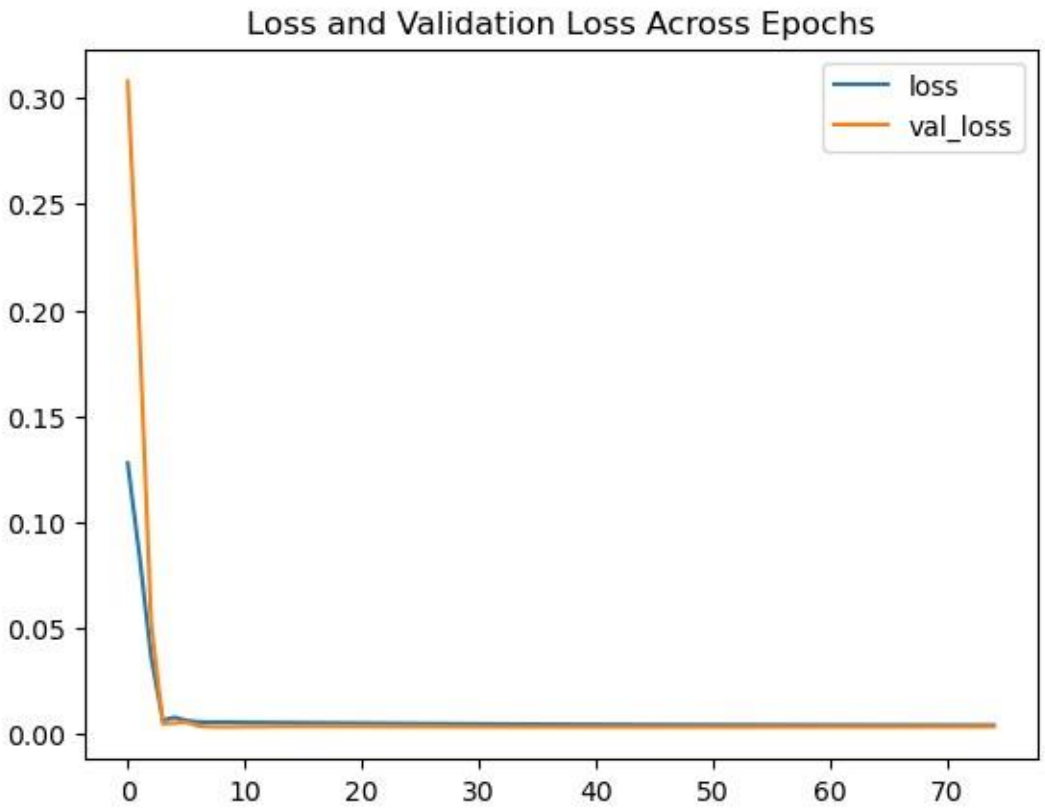
```
0.0034
0.0034
Epoch 59/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0043 -
val_loss:
0.0034
Epoch 60/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0043 -
val_loss:
0.0034
Epoch 61/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0043 -
val_loss:
0.0034
Epoch 62/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0043 -
val_loss:
0.0034
Epoch 63/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0043 -
val_loss:
0.0034
Epoch 64/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0043 -
val_loss:
0.0034
Epoch 65/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0043 -
val_loss:
Epoch 66/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0043 -
val_loss:
0.0034
Epoch 67/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0042 -
val_loss:
0.0034
Epoch 68/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0042 -
val_loss:
0.0034
Epoch 69/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0042 -
val_loss:
0.0035
Epoch 70/75
12/12 [==============================] - 0s 11ms/step - loss: 0.0042 -
val_loss:
0.0035
Epoch 71/75
```

```
0.0034
12/12 [==============================] - 0s 6ms/step - loss: 0.0042 -
val_loss:
0.0035
Epoch 72/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0042 -
val_loss:
0.0035
Epoch 73/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0042 -
val_loss:
0.0035
Epoch 74/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0042 -
val_loss:
0.0035
Epoch 75/75
12/12 [==============================] - 0s 6ms/step - loss: 0.0042 -
val_loss:
0.0035
```

Plotting Loss and Validation Loss across Epochs

The learning curve looks promising. It seems as if the model has successfully learned. The loss curves drop steeply and stabilize near zero, and there is little to no gap between the loss and validation loss lines.

[6609]:
```
import matplotlib.pyplot as plt
plt.plot(history.history["loss"], label="loss")
plt.plot(history.history["val_loss"],
label="val_loss") plt.title('Loss and Validation
Loss Across Epochs') plt.legend() plt.show()
```



Predicting and Reversing Scaling

[6610]:
```
y_pred3  = model3 .predict(X_test)
```

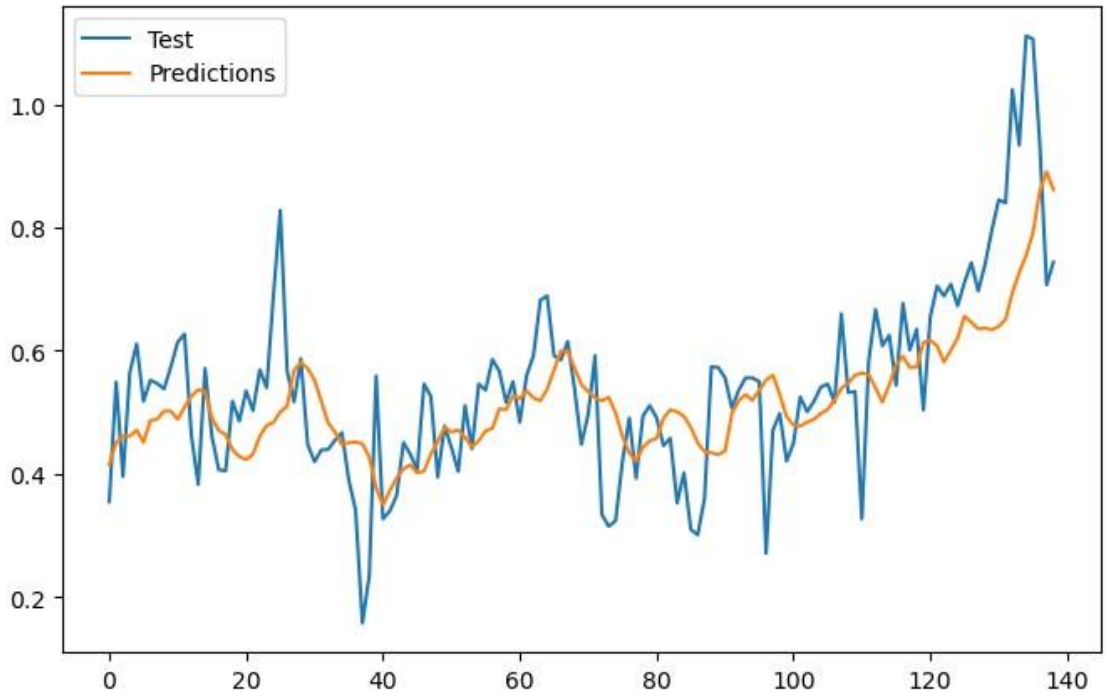
```
5/5 [==============================] - 1s 2ms/step
```

[6611]:
```
unscaled_predictions = y_scaler.inverse_transform(y_pred3)
unscaled_y_test = y_scaler.inverse_transform(y_test)
```

Plotting Actual Anomaly Values vs Predicted

This graph looks better than I expected given the change in our data structure and the increase in time gap between the features and the target. The predictions follow the general trend of the test values, but seem to be missing the details.

[6612]:
```
plt.figure(figsize =(8,5))
plt.plot( list (unscaled_y_test), label  ="Test ")
plt.plot( list (unscaled_predictions), label   ="Predictions ")
plt.legend()
plt.show()
```



Evaluation Scores

[6613]:
```
Evaluate(unscaled_y_test,unscaled_predictions)
```

```
Mean Absolute Error: 0.08731903786967986
Mean Squared Error: 0.012864114774034433
Root Mean Squared Error: 0.2954979490109531
R Squared: 0.4573229622339118
```

Conclusions and Reflections

I expected a worse performance from this model because of the time gap between the input features and the target, and that is what happened. However, the results are not terrible. The r2 is 46%, which outperforms the random forest regressor model and the baseline multiple linear regression model. Considering the complexity and power of the LSTM RNN algorithm, as well as its applicability to the task at hand, I thought it would perform *much* better than multiple linear regression. I suppose simplicity is best sometimes.

I kept the configuration of the model structure and the hyperparamters the same as the first iteration of LSTM so I could directly compare. This model experienced a loss of roughly .30 in r2 score, and an increase in MAE by about 0.024 degrees. The MAE, MSE, and RMSE are all lower than the baseline scores. Most importantly, I can confidently say that this model is predicting anomaly values t+3 in the future.

To make the LSTM work better, I think one improvement would be to have a larger dataset. Since LSTMs learn long term dependencies, they need a lot of data to fulfill their purpose. Perhaps our dataset is too small for such a complex algorithm to produce the best results. Also, adding new features which are better predictors of anomaly value would likely help to boost the predictive power.

8 XGBoost Model

XGBoost, also known as Extreme Gradient Booster, belongs to a class of gradient boosting models. It uses ensemble learning based on decision trees, just like random forest. The difference is that random forest uses bagging, while XGBoost uses boosting. The gradient boosting technique of XGBoost involves building decision trees sequentially (rather than simultaneously as in bagging) and each successive tree learns from the previous. A gradient-boosting algorithm minimizes the gradient loss as the model is training, similar to the fitting process of neural networks. It is not a deep learning algorithm, but rather a highly efficient decision tree - ensemble model.

XGBoost is a highly capable model. In fact, I did not need to do as much data preparation as I did because XGBoost is able to handle null values, categorical variables, and multicollinearity. It is known for being highly efficient and accurate.

It will be interesting to compare the results of XGBoost againt the random forest regressor model, since both are decision-tree-based ensemble models. I expect XGBoost to outperform random forest due to its gradient boosting method, greater complexity, and the presence of more hyperparameters to tune to optimize the performance.

https://xgboost.readthedocs.io/en/stable/parameter.html https://machinelearningmastery.com/xgboost-for-regression/

https://www.kaggle.com/code/robikscube/time-series-forecasting-with-machine-learning-yt

[6614]:
```
df5.head()
```

[6614]:

date	emission	anomaly_value	All natural disasters	global_avg_temp	anomaly_value_target
1901-01-31	4.482107e+10	-0.075	5.0	-0.28	-0.236
1901-02-28	4.482107e+10	-0.176	5.0	-0.06	-0.187
1901-03-31	4.482107e+10	-0.272	5.0	0.04	-0.196
1901-04-30	4.482107e+10	-0.236	5.0	-0.06	-0.146
1901-05-31	4.482107e+10	-0.187	5.0	-0.17	-0.188

Train Valid Test Split

Including a validation set vastly improved the performance of the model. I split the data 80/10/10 train/valid/test.

[6615]:
```
X = df5[['emission ','All natural disasters  ','global_avg_temp ', 'anomaly_value ']]
y = df5['anomaly_value_target  ']


X_train, X_else, y_train, y_else    = train_test_split(X, y, test_size   =.2,␣
 ↪shuffle =False)
X_valid, X_test, y_valid, y_test    = train_test_split(X_else, y_else, test_size    =.
 ↪5, shuffle =False)


print(X_train.shape)
print(X_valid.shape)
print(X_test.shape)
print(y_train.shape)
print(y_valid.shape)
print(y_test.shape)
```

```
(1107, 4)
(138, 4)
(139, 4)
(1107,)
(138,)
(139,)
```

Constructing and Fitting the Model

XGBoostRegressor has lots of hyperparameters to tune. I experimented with a variety of configurations and settled on this one.

- base_score is the global bias assigned to all initial instances. If a number is not set, it is automatically estimated. This did not have much effect on the results.
- booster - here we specify a booster function. For this model, since we are working with a regressor, we choose gblinear.
- n_estimators is the number of rounds. Increasing this improved the performance
- early_stopping_rounds is set to 10. It is the number of rounds that the model is allowed to run after validation score stops improving to prevent overfitting.
- objective is set to reg:linear since we are performing a regression task

Next, we fit the training data to the model and specify our evaluation data as the valid and test sets.

[6616]:
```
model5  = xgb.XGBRegressor(base_score  =0.5, booster ='gblinear ',
                           n_estimators =2000,
                           freq  = 1,
                           early_stopping_rounds  =10,
                           objective ='reg:linear ')

model5.fit(X_train, y_train,
        eval_set =[(X_valid, y_valid),(X_test, y_test)],
        verbose =20)
```

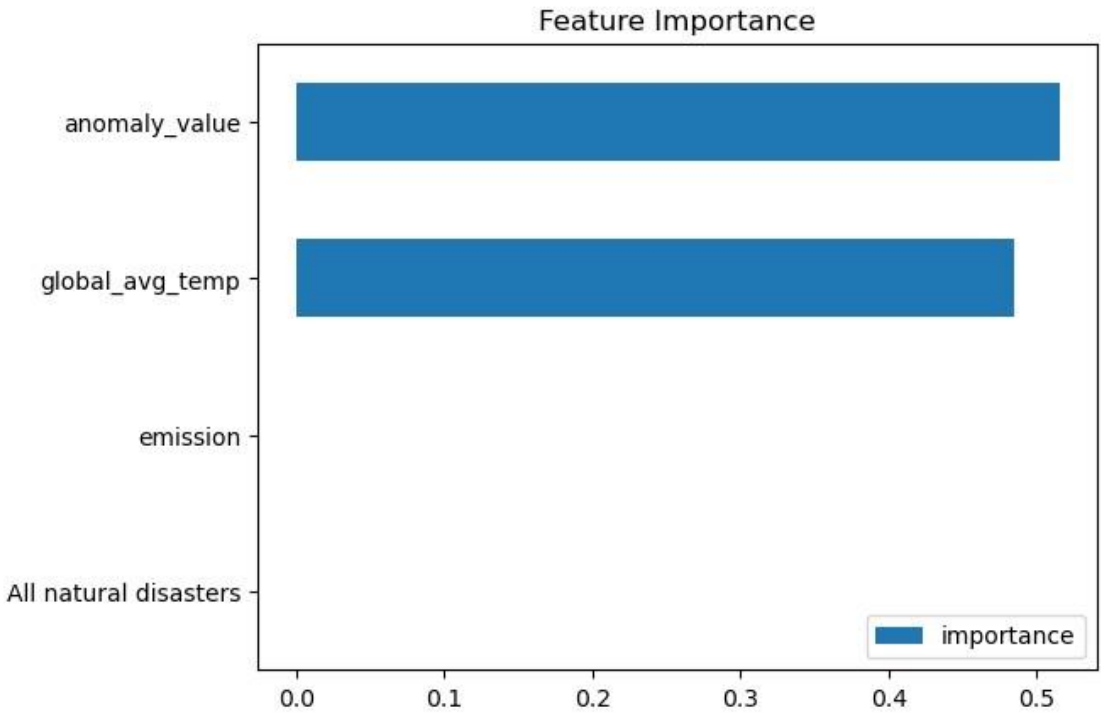
```
[19:51:10] WARNING: C:\buildkite-agent\builds\buildkite-windows-cpu-
autoscalinggroup-i-07593ffd91cd9da33-1\xgboost\xgboost-
ciwindows\src\objective\regression_obj.cu:213: reg:linear is now deprecated
in favor of reg:squarederror.
[19:51:10] WARNING: C:\buildkite-agent\builds\buildkite-windows-cpu-
autoscalinggroup-i-07593ffd91cd9da33-1\xgboost\xgboost-ci-
windows\src\learner.cc:767: Parameters: { "freq" } are not used.
[0]     validation_0-rmse:0.16142     validation_1-
                                      rmse:0.19643
[20]    validation_0-rmse:0.11970     validation_1-
                                      rmse:0.13589
[40]    validation_0-rmse:0.11786     validation_1-
                                      rmse:0.12665
[60]    validation_0-rmse:0.12032     validation_1-
                                      rmse:0.12313
[80]    validation_0-rmse:0.12387     validation_1-
                                      rmse:0.12189
[100]   validation_0-rmse:0.12738     validation_1-
                                      rmse:0.12173
[102]   validation_0-rmse:0.12769     validation_1-
                                      rmse:0.12171
```

[6616]:
```
XGBRegressor(base_score=0.5, booster='gblinear', callbacks=None,
             colsample_bylevel=None, colsample_bynode=None,
             colsample_bytree=None, early_stopping_rounds=10,
             enable_categorical=False, eval_metric=None, feature_types=None,
             freq=1, gamma=None, gpu_id=None, grow_policy=None,
             importance_type=None, interaction_constraints=None,
             learning_rate=None, max_bin=None, max_cat_threshold=None,
             max_cat_to_onehot=None, max_delta_step=None, max_depth=None,
             max_leaves=None, min_child_weight=None, missing=nan,
             monotone_constraints=None, n_estimators=2000, n_jobs=None,
             num_parallel_tree=None, objective='reg:linear', …)
```

Feature Importance

A useful tool of XGBoost is the ability to know the feature importances. The model assigned importance to only two features, anomaly value and global avg temp which are closely correlated with lagged anomaly value.

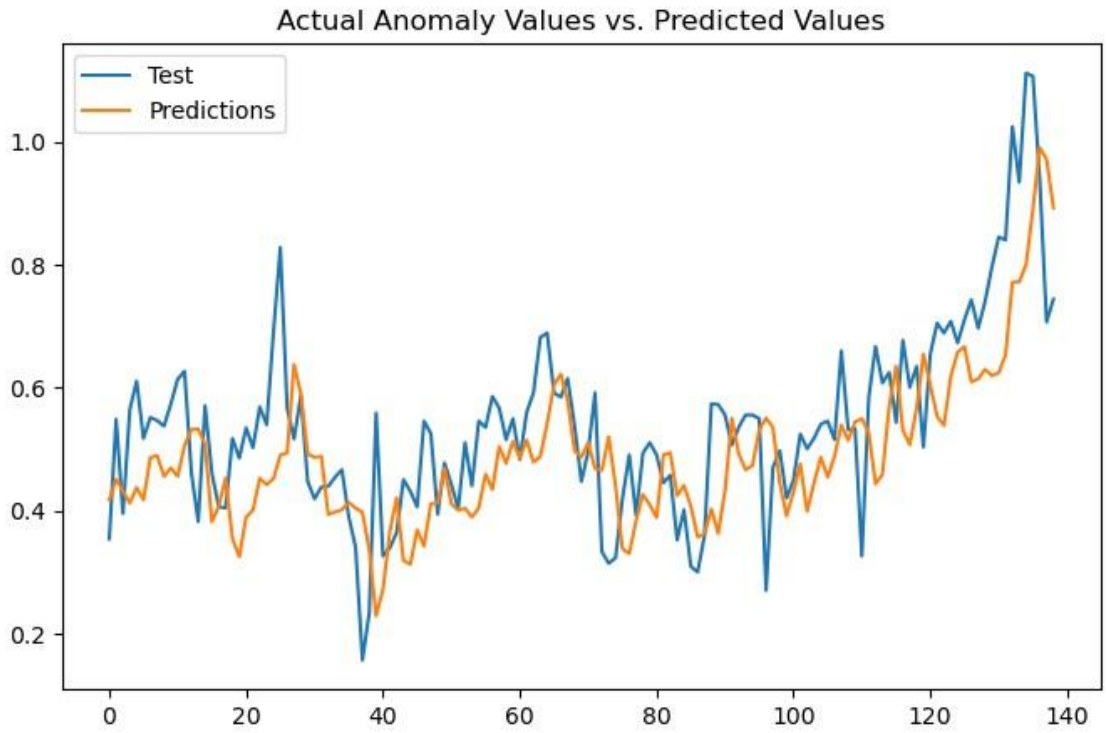
[6617]:
```
feature_importance =
            pd.DataFrame(data=model5.feature_importances_,
            index=model5.feature_names_in_,
            columns=['importance'])
feature_importance.sort_values('importance').plot(kind='barh',
title='Feature␣
 ↪Importance')
plt.show()
```



Predicting and Plotting True Values vs. Predicted

We can see in the plot that the model was able to learn general trends in anomaly value but did not exactly hit the target with its predictions.

[6618]:
```
y_pred5  = model5 .predict(X_test)

plt .figure(figsize =(8,5))
plt .plot( list (y_test), label  ="Test ")
plt .plot( list (y_pred5), label  ="Predictions ")
plt .title( 'Actual Anomaly Values vs. Predicted Values     ')
plt .legend()
plt .show()
```



Evaluation Scores

[6619]:
```
Evaluate(y_test,y_pred5)
```

```
Mean Absolute Error: 0.09849871318974941
Mean Squared Error: 0.014817091642383535
Root Mean Squared Error: 0.3138450464636162
R Squared: 0.37493597172908444
```

Conclusions and Reflections

XGBoostRegressor performed decently, with an r2 of 37% which is close to the baseline. The MAE and RMSE are very similar to the baseline scores. For a model as highly praised as XGBoost I would have expected better results, but multiple linear regression was able to outperform it. However as I expected, it vastly outperformed the random forest model. I must mention that I differenced the data for the random forest, and did not reverse the differencing for evaluations which affected the scores.

I experimented with tuning the hyperparameters to try to improve the performance. Increasing the n_estimators made the biggest difference by far. Additionally, adding a validation set to the train-test split enhanced the performance greatly.

To further improve the model, adding more data would be beneficial. Without time limits, I would have liked to try feature engineering as I did for the LSTM models by adding copies of shifted features to the dataset. This would likely help the model learn patterns better, and since XGBoost is not sensitive to multicollinearity, this wouldn't be a problem as it would be for multiple linear regression. Since XGBoost is not sensitive to messy data, scaling or differencing may not help much, but it would be worth a try in further exploration.

9 Final Conclusions and Reflections

First of all, this project proved to me how vast and complex the world of machine learning is. I could spend years devoted to the study of each model and still not fully understand their mechanisms and complexities. The algorithms used by the learning models operate on highly advanced mathematical and statistical functions which are mostly beyond my current understanding. However, the process of putting together this project enhanced my knowledge of machine learning exponentially, and opened my eyes to a world of resources from websites and blogs to youtube channels. I now understand much more about the data science workflow for machine learning, including data preparation techniques, feature selection and engineering, data pre-processing for different types of models, modeling, predicting, and cross validating the predictions. If I had more time to deepen my understanding and continue my exploration of the climate dataset, experiment with different ways of feature engineering, more models, and different settings of hyperparameters, I am sure I could reach a better result. And so, the results I got from my models are not definitive for this dataset, but could be improved upon in hundreds of ways. I have done my best to explain my reasoning which motivated every step of this project and have interpreted the results to the best of my ability.

The final ranking of model performance based on r2 is as follows:

- 1) LSTM
- 2) Multiple Linear Regression
- 3) LSTM with Modified Data structure (3 time steps between features and target)
- 4) XGBoostRegressor
- 5) Random Forest Regressor

My first consideration regarding the top performer (LSTM) is the data preparation. While I read online that this was the proper way to engineer the features for LSTM, I had doubts that it applied to a t+3 forecasting problem like this one, since the gap between the closest independent feature sample and the target was only one lag. If my doubts are warranted, then the top performer is actually the LSTM with the corrected lag data structure followed closely by the Multiple Linear Regression model. The second LSTM perfomed worse because naturally it is harder for the model to learn trends across a three month lag than a one month lag.

One reason that a simple model like multiple linear regression performed so well might be that our dataset is relatively small. Complex models like LSTM which can remember long-term dependencies do best on large datasets. XGBoost also performs better on large datasets. Perhaps a dataset of 10,000+ datapoints would perform better with these models. Data from the same date range but sampled weekly or daily instead of monthly could be an option for increasing the size of the dataset and enhancing the models' ability to detect trends.

Besides the limitations of the small dataset, LSTM is a great model for this type of problem. Random Forest performed terribly, but I must take some blame since I wasn't able to successfully reverse the differencing before evaluating the results. Reversing the differencing might have improved its scores, although I still think random forest is not the ideal model for this type of problem and I would not select it in the future.

Another important thing to consider is feature selection. I did not have enough time to do so, but experimenting with different independent features would be beneficial. I tried to be as thorough as possible, selecting the variables which are the best predictors of anomaly value by calculating the autocorrelations and observing pair plots. However, there may be better methods for choosing the best features. One option that is outside the scope of this project would be to source new data online and add more features to the dataset. I am sure there are many interesting variables besides the ones included in this dataset that could be good predictors of anomaly value. Particularly with the current climate crisis, there is plentiful free data available on this subject. More features could

significantly enhance the performance of the models. Different ways of cleaning data could also have been implemented. I deleted several rows of datapoints that contained null values for some variables. Interpolation or imputation could have been performed instead, although I think I made the right choice.

There are endless configurations of hyperparameters to try for LSTM and XGBoostRegressor. With time, these can be experimented with at length and the optimum performance could be achieved. I did quite a bit of experimentation with the hyperparameters, but was limited by time.

The train-test split could also be adjusted to allocate more or less data to the test set. I stuck with an 80/20 split or 80/10/10 if including a validation set for every model for consistency. 80/20 is a reasonable split, but in the future I would try 70/30 as well. The goal is to provide enough data for satisfactory training but not too much that it causes overfitting to the data.

For the future, I would likely select LSTM if I had a time series forecasting problem. I think LSTM is best suited for this type of problem and as a neural network has the best capability to capture complexities in the data. With proper data preparation and feature engineering, I believe LSTM is the model that has the most potential.

It would be useful to have expertise on the subject of climatology to better understand the variables in our dataset and the possible effects of each variable on anomaly values. In general, it is helpful to have subject matter knowledge when working with a dataset in a particular subject.

During the process of putting together this project, I uncovered far more questions than answers. With each short line of code, an entire world of complex ideas was opened and at times I felt lost in the vastness of available information. I did my best to understand the concepts behind each step of the project and enjoyed gaining new knowledge. For the future, I hope to continue to deepen my understanding of this exciting and growing field.

10 References

https://learn.microsoft.com/en-us/azure/architecture/data-guide/scenarios/time-series

https://reason.town/importance-of-statistics-in-machine-learning/

https://github.com/cs109/2015/blob/master/Lectures/01-Introduction.pdf https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html

https://www.stackoverflow.com/questions/53669688/convert-date-column-from-object-data-typeto-date-time-data-type

https://www.youtube.com/watch?v=TpQtD7ONfxQ

https://statisticsbyjim.com/regression/multicollinearity-in-regression-analysis/

https://machinelearningmastery.com/an-introduction-to-feature-selection/ https://statisticsbyjim.com/time-series/autocorrelation-partial-autocorrelation/ https://www.mathworks.com/help/stats/machine-learning-in-matlab.html https://www.investopedia.com/terms/r/regression.asp

https://www.reneshbedre.com/blog/learn-to-calculate-residuals-regression.html?utm_content=cmptrue

https://www.statisticshowto.com/absolute-error/

https://www.freecodecamp.org/news/machine-learning-mean-squared-error-regression-linec7dde9a26b93/

https://towardsdatascience.com/what-does-rmse-really-mean-806b65f2e48e https://vitalflux.com/r-squared-explained-machine-learning/

https://machinelearningmastery.com/training-validation-test-split-and-cross-validation-doneright/

https://www.youtube.com/watch?v=i_LwzRVP7bg https://www.investopedia.com/terms/m/mlr.asp

https://www.bitdegree.org/learn/train-test-split

https://stackoverflow.com/questions/42191717/scikit-learn-random-state-in-splitting-dataset

https://www.statology.org/good-vs-bad-residual-plot/

https://towardsdatascience.com/multivariate-time-series-forecasting-using-random-forest-2372f3ecbad1

https://www.kaggle.com/code/pbizil/random-forest-regression-for-time-series-predict/notebook

https://towardsdatascience.com/random-forest-regression-5f605132d19d

https://towardsdatascience.com/random-forest-regression-5f605132d19d

https://machinelearningmastery.com/convert-time-series-supervised-learning-problem-python/

https://statisticsbyjim.com/regression/r-squared-too-high/

https://www.kaggle.com/code/pbizil/random-forest-regression-for-time-series-predict/notebook

https://builtin.com/data-science/recurrent-neural-networks-and-lstm

https://www.youtube.com/watch?v=qO_NLVjD6zE

https://machinelearningmastery.com/gentle-introduction-long-short-term-memory-networksexperts/

https://machinelearningmastery.com/convert-time-series-supervised-learning-problem-python/
https://datascience.stackexchange.com/questions/28328/how-does-multicollinearity-affect-neuralnetworks

https://machinelearningmastery.com/learning-curves-for-diagnosing-machine-learning-modelperformance/

https://xgboost.readthedocs.io/en/stable/parameter.html https://machinelearningmastery.com/xgboost-for-regression/

https://www.kaggle.com/code/robikscube/time-series-forecasting-with-machine-learning-yt

[6620]:
```
%%javascript
$.getScript( 'https://kmahelona.github.io/ipython_notebook_goodies/
  ↪ipython_notebook_toc.js'  )
```

```
<IPython.core.display.Javascript object>
```