Anomaly detection

Anomaly detection, also known as outlier detection, is the process of identifying patterns or instances that deviate significantly from the norm or expected behavior within a dataset. These anomalies are often indicative of unusual events, errors, or potential threats in various applications. Anomaly detection techniques are widely used across different domains, including finance, cybersecurity, healthcare, manufacturing, and more. Here are some key points about anomaly detection:

Objective: The primary goal of anomaly detection is to identify and flag instances that are significantly different from the majority of data points in a dataset. These anomalies may represent rare events, errors, or patterns that require further investigation.

Types of Anomalies:

Point Anomalies: Individual data points that are anomalous compared to the rest of the dataset.

Contextual Anomalies: Instances that are anomalous only in specific contexts or conditions.

Collective Anomalies: Groups of data points that together form an anomalous pattern, but each individual point may not be anomalous.

Approaches:

- Unsupervised Learning: Most anomaly detection techniques are unsupervised, meaning they don't require labeled data. They rely on learning the normal behavior of the dataset and flagging instances that deviate significantly from this norm.
- Supervised Learning: In some cases, anomaly detection can be framed as a supervised learning problem if labeled examples of anomalies are available.
- Semi-Supervised Learning: This approach combines elements of supervised and unsupervised learning, where the model is trained on a dataset containing both normal and anomalous instances.

Techniques:

- Statistical Methods: These include methods such as z-score, modified z-score, and Gaussian distribution modeling to identify outliers based on statistical properties of the data.
- Machine Learning Algorithms: Various machine learning algorithms, including clustering, density estimation, and neural networks, can be used for anomaly detection.
- Time-Series Analysis: Anomaly detection in time-series data involves identifying deviations from expected patterns over time, such as sudden spikes or drops.
- Deep Learning: Deep learning techniques, including autoencoders, generative adversarial networks (GANs), and recurrent neural networks (RNNs), have shown promising results for anomaly detection, especially in complex data domains such as images and sequences.

Applications:

1- Cybersecurity: Detecting malicious activities, intrusions, and network anomalies.

2- Fraud Detection: Identifying fraudulent transactions or activities in financial transactions.

3- Healthcare: Monitoring patient data for unusual patterns that may indicate health issues or medical errors.

4- Predictive Maintenance: Detecting anomalies in machinery or equipment sensor data to prevent breakdowns or failures.

5- Manufacturing: Identifying defects or anomalies in manufacturing processes to improve quality control.

In summary, anomaly detection plays a crucial role in identifying unusual patterns or events within data, enabling organizations to take proactive measures to address potential issues and improve overall efficiency and security.

Anomaly detection techniques aim to identify patterns in data that deviate significantly from the norm or expected behavior. Here's a more detailed overview of common anomaly detection techniques:

Statistical Methods:

- Z-Score: This method identifies anomalies based on the standard deviation from the mean of the dataset. Data points that fall outside a certain threshold (typically defined by a z-score cutoff) are considered anomalies.
- Modified Z-Score: Similar to the z-score method, but it is more robust to outliers by using the median and median absolute deviation (MAD) instead of the mean and standard deviation.
- Gaussian Distribution Modeling (GMM): Assumes that the data follows a Gaussian (normal) distribution. Anomalies are detected as data points with low probability under the fitted Gaussian distribution.

Machine Learning Algorithms:

- Clustering: Techniques like k-means clustering or DBSCAN (Density-Based Spatial Clustering of Applications with Noise) can be used to cluster data points, and outliers are identified as points that do not belong to any cluster or belong to small clusters.
- Density Estimation: Methods such as kernel density estimation (KDE) estimate the probability density function of the data, and anomalies are identified as points with low density.
- Isolation Forest: An ensemble learning method that constructs random decision trees to isolate anomalies by recursively partitioning the feature space.
- One-Class SVM (Support Vector Machine): Trains a model on the normal data and detects anomalies as data points lying far from the decision boundary.

Time-Series Analysis:

- Moving Average: Smoothens time-series data by averaging adjacent data points over a window. Anomalies are identified as data points that deviate significantly from the moving average.
- Exponential Smoothing: Assigns exponentially decreasing weights to past observations to estimate the current value. Anomalies are detected similarly to moving average methods.
- Seasonal Decomposition: Decomposes time-series data into seasonal, trend, and residual components. Anomalies are identified in the residual component.

Deep Learning:

- Autoencoders: Unsupervised neural networks that learn to reconstruct input data. Anomalies are detected by measuring the reconstruction error, with higher errors indicating anomalies.
- Variational Autoencoders (VAEs): Variants of autoencoders that learn a probabilistic latent space. Anomalies are detected based on the reconstruction error and the likelihood of data points in the latent space.
- Generative Adversarial Networks (GANs): Deep learning models that generate synthetic data samples. Anomalies are detected by measuring the difference between real and generated samples.

Ensemble Methods:

- Combining Multiple Algorithms: Ensemble methods combine the outputs of multiple anomaly detection algorithms to improve overall performance and robustness.

Each technique has its strengths and weaknesses, and the choice of method depends on factors such as the nature of the data, the type of anomalies expected, computational resources available, and the specific requirements of the application. Additionally, anomaly detection often involves a combination of techniques for better accuracy and reliability.

Double-click (or enter) to edit

```
!pip install shap
```

```
Collecting shap
  Downloading shap-0.45.0-cp310-cp310-manylinux_2_12_x86_64.manylinux2010_x86_64.manylinux_2_17_x86_64.manylinux2014_x86_64.whl (538 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 538.2/538.2 kB 9.4 MB/s eta 0:00:00
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-packages (from shap) (1.25.2)
Requirement already satisfied: scipy in /usr/local/lib/python3.10/dist-packages (from shap) (1.11.4)
Requirement already satisfied: scikit-learn in /usr/local/lib/python3.10/dist-packages (from shap) (1.2.2)
Requirement already satisfied: pandas in /usr/local/lib/python3.10/dist-packages (from shap) (1.5.3)
Requirement already satisfied: tqdm>=4.27.0 in /usr/local/lib/python3.10/dist-packages (from shap) (4.66.2)
Requirement already satisfied: packaging>20.9 in /usr/local/lib/python3.10/dist-packages (from shap) (24.0)
Collecting slicer==0.0.7 (from shap)
  Downloading slicer-0.0.7-py3-none-any.whl (14 kB)
Requirement already satisfied: numba in /usr/local/lib/python3.10/dist-packages (from shap) (0.58.1)
Requirement already satisfied: cloudpickle in /usr/local/lib/python3.10/dist-packages (from shap) (2.2.1)
Requirement already satisfied: llvmlite<0.42,>=0.41.0dev0 in /usr/local/lib/python3.10/dist-packages (from numba->shap) (0.41.1)
Requirement already satisfied: python-dateutil>=2.8.1 in /usr/local/lib/python3.10/dist-packages (from pandas->shap) (2.8.2)
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-packages (from pandas->shap) (2023.4)
Requirement already satisfied: joblib>=1.1.1 in /usr/local/lib/python3.10/dist-packages (from scikit-learn->shap) (1.3.2)
Requirement already satisfied: threadpoolctl>=2.0.0 in /usr/local/lib/python3.10/dist-packages (from scikit-learn->shap) (3.3.0)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-packages (from python-dateutil>=2.8.1->pandas->shap) (1.16.0)
Installing collected packages: slicer, shap
Successfully installed shap-0.45.0 slicer-0.0.7
```

```
import numpy as np
import pandas as pd
import holoviews as hv
from holoviews import opts
hv.extension('bokeh')
from matplotlib import pyplot as plt
import seaborn as sns
import os
import scipy.stats as stats
from sklearn.svm import OneClassSVM
from sklearn.ensemble import IsolationForest
from sklearn.neighbors import LocalOutlierFactor
!pip install changefinder
import changefinder
from sklearn.metrics import f1_score
import shap
shap.initjs()
from tabulate import tabulate
from IPython.display import HTML, display

# Load the dataset from Google Colab files
df = pd.read_csv("machine_temperature_system_failure.csv", low_memory=False)
print(f'machine_temperature_system_failure.csv : {df.shape}')
df.head(3)

#shap: Library for explaining the output of machine learning models.
#tabulate: Library for pretty-printing tabular data.
#IPython.display: Module for displaying HTML, images, and videos in the Jupyter Notebook.
```

```
Requirement already satisfied: changefinder in /usr/local/lib/python3.10/dist-packages (
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-packages (from ch
Requirement already satisfied: scipy in /usr/local/lib/python3.10/dist-packages (from ch
Requirement already satisfied: statsmodels in /usr/local/lib/python3.10/dist-packages (f
Requirement already satisfied: nose in /usr/local/lib/python3.10/dist-packages (from cha
Requirement already satisfied: pandas!=2.1.0,>=1.0 in /usr/local/lib/python3.10/dist-pac
Requirement already satisfied: patsy>=0.5.4 in /usr/local/lib/python3.10/dist-packages (
Requirement already satisfied: packaging>=21.3 in /usr/local/lib/python3.10/dist-package
Requirement already satisfied: python-dateutil>=2.8.1 in /usr/local/lib/python3.10/dist-
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-packages (
Requirement already satisfied: six in /usr/local/lib/python3.10/dist-packages (from pats
machine_temperature_system_failure.csv : (22695, 2)
```

	timestamp	value
0	2013-12-02 21:15:00	73.967322
1	2013-12-02 21:20:00	74.935882

preprocessing

```
# Convert timestamp column to datetime format
df['timestamp'] = pd.to_datetime(df['timestamp'])

# Create a binary column indicating whether each timestamp falls within the anomaly periods
anomaly_points = [
    ["2013-12-10 06:25:00.000000", "2013-12-12 05:35:00.000000"],
    ["2013-12-15 17:50:00.000000", "2013-12-17 17:00:00.000000"],
    ["2014-01-27 14:20:00.000000", "2014-01-29 13:30:00.000000"],
    ["2014-02-07 14:55:00.000000", "2014-02-09 14:05:00.000000"]
]
df['anomaly'] = 0
for start, end in anomaly_points:
    df.loc[((df['timestamp'] >= start) & (df['timestamp'] <= end)), 'anomaly'] = 1

# Extract datetime information into separate columns
df['year'] = df['timestamp'].dt.year
df['month'] = df['timestamp'].dt.month
df['day'] = df['timestamp'].dt.day
df['hour'] = df['timestamp'].dt.hour
df['minute'] = df['timestamp'].dt.minute

# Set timestamp column as the index and drop it
df.index = df['timestamp']
df.drop(['timestamp'], axis=1, inplace=True)

df.head(15)
```

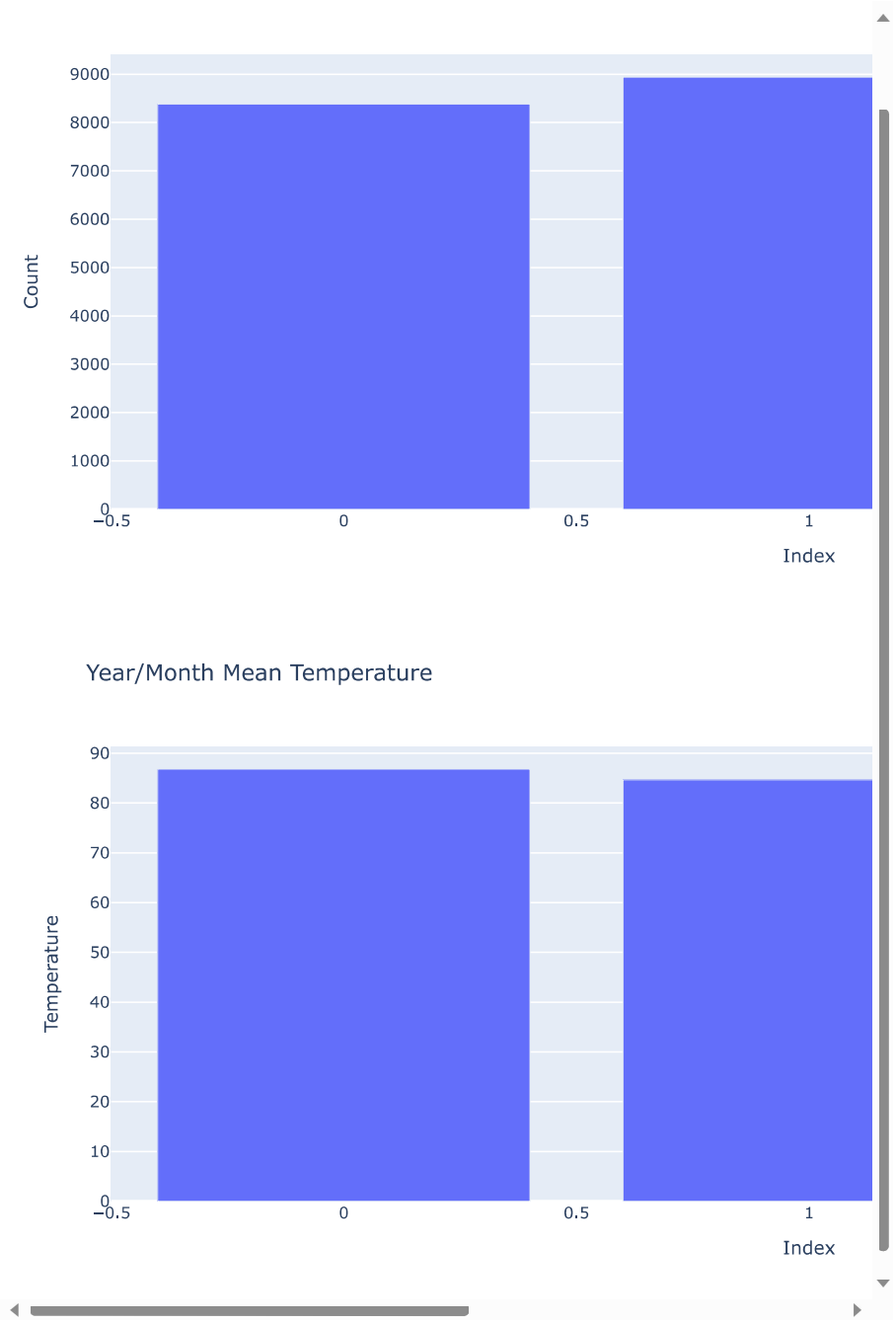
timestamp	value	anomaly	year	month	day	hour	minute
2013-12-02 21:15:00	73.967322	0	2013	12	2	21	15
2013-12-02 21:20:00	74.935882	0	2013	12	2	21	20
2013-12-02 21:25:00	76.124162	0	2013	12	2	21	25
2013-12-02 21:30:00	78.140707	0	2013	12	2	21	30
2013-12-02 21:35:00	79.329836	0	2013	12	2	21	35
2013-12-02 21:40:00	78.710418	0	2013	12	2	21	40
2013-12-02 21:45:00	80.269784	0	2013	12	2	21	45
2013-12-02 21:50:00	80.272828	0	2013	12	2	21	50
2013-12-02 21:55:00	80.353425	0	2013	12	2	21	55
2013-12-02 22:00:00	79.486523	0	2013	12	2	22	0
2013-12-02 22:05:00	80.783277	0	2013	12	2	22	5
2013-12-02 22:10:00	79.508159	0	2013	12	2	22	10
2013-12-02 22:15:00	79.302033	0	2013	12	2	22	15
2013-12-02 22:20:00	80.802624	0	2013	12	2	22	20
2013-12-02 22:25:00	80.377789	0	2013	12	2	22	25

Next steps: Generate code with df View recommended plots

EDA

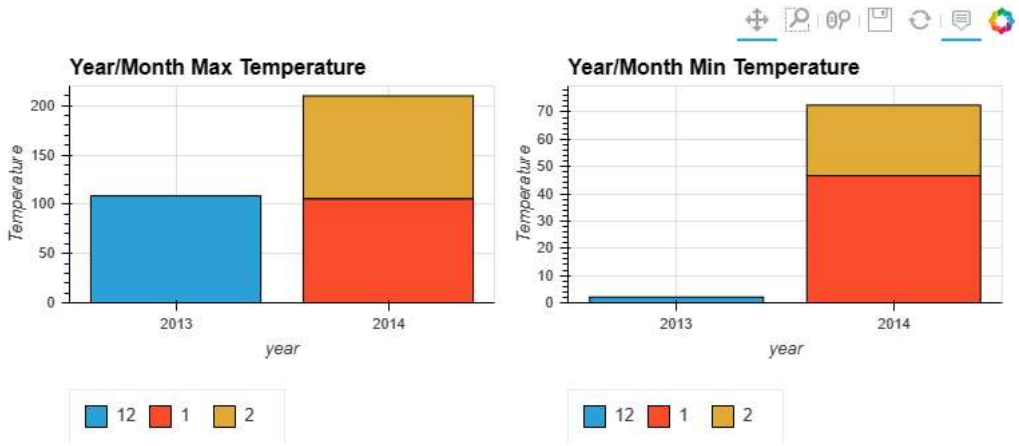
EDA stands for Exploratory Data Analysis. It's an approach to analyzing data sets to summarize their main characteristics, often with visual methods. The main purpose of EDA is to understand the data, discover patterns, detect anomalies, and formulate hypotheses that can be tested further. EDA involves various techniques such as data visualization, statistical analysis, and data mining to gain insights into the underlying structure of the data.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import os
import scipy.stats as stats
from sklearn.svm import OneClassSVM
from sklearn.ensemble import IsolationForest
from sklearn.neighbors import LocalOutlierFactor
from changefinder import ChangeFinder
from sklearn.metrics import f1_score


# Load the dataset from Google Colab files
df = pd.read_csv("machine_temperature_system_failure.csv", low_memory=False)
# Convert timestamp to datetime
df['timestamp'] = pd.to_datetime(df['timestamp'])

# Define anomaly points
anomaly_points = [
    ["2013-12-10 06:25:00.000000", "2013-12-12 05:35:00.000000"],
    ["2013-12-15 17:50:00.000000", "2013-12-17 17:00:00.000000"],
    ["2014-01-27 14:20:00.000000", "2014-01-29 13:30:00.000000"],
    ["2014-02-07 14:55:00.000000", "2014-02-09 14:05:00.000000"]
]

# Set anomaly flag
df['anomaly'] = 0
for start, end in anomaly_points:
    df.loc[((df['timestamp'] >= start) & (df['timestamp'] <= end)), 'anomaly'] = 1

# Extract datetime information
df['year'] = df['timestamp'].dt.year
df['month'] = df['timestamp'].dt.month
df['day'] = df['timestamp'].dt.day
df['hour'] = df['timestamp'].dt.hour
df['minute'] = df['timestamp'].dt.minute

# Set timestamp as index and drop the timestamp column
df.index = df['timestamp']
df.drop(['timestamp'], axis=1, inplace=True)



import plotly.graph_objects as go

# Basic analysis: Count the number of values per year and month
count_data = df.groupby(['year', 'month'])['value'].count().reset_index()
count_fig = go.Figure(data=[go.Bar(x=count_data.index, y=count_data['value'])])
count_fig.update_layout(title='Year/Month Count', xaxis_title='Index', yaxis_title='Count')

# Calculate the mean temperature per year and month
mean_data = df.groupby(['year', 'month'])['value'].mean().reset_index()
mean_fig = go.Figure(data=[go.Bar(x=mean_data.index, y=mean_data['value'])])
mean_fig.update_layout(title='Year/Month Mean Temperature', xaxis_title='Index', yaxis_title='Temperature')

# Display the count and mean temperature plots
count_fig.show()
mean_fig.show()
```

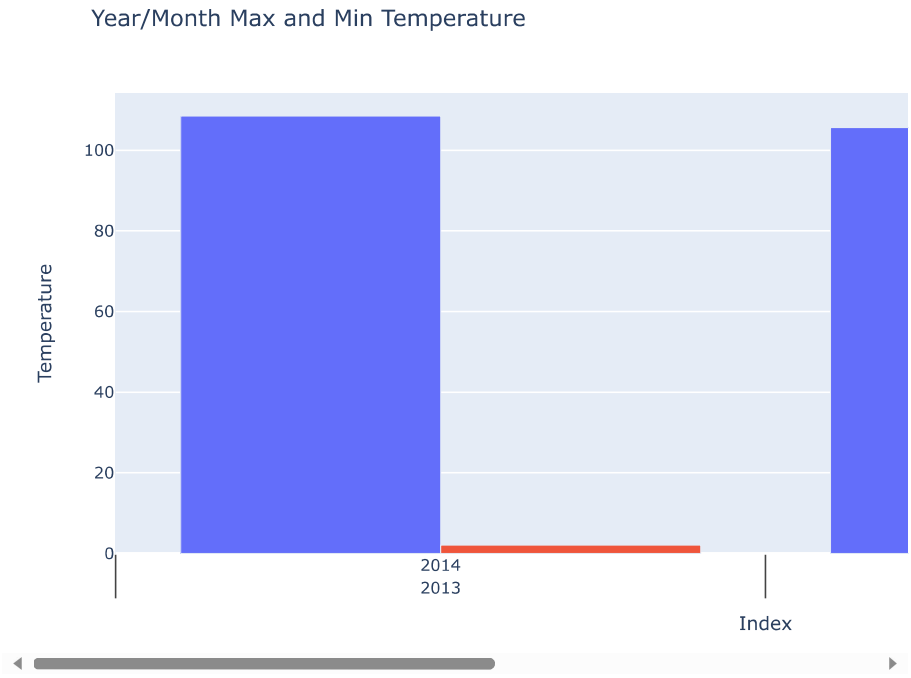



```
# This code generates two plots: one showing the count of values per year and month, and another showing the mean temperature per year and m

# Basic analysis: Count the number of values per year and month
count = hv.Bars(df.groupby(['year','month'])['value'].count()).opts(ylabel="Count", title='Year/Month Count')

# Calculate the mean temperature per year and month
mean = hv.Bars(df.groupby(['year','month']).agg({'value': ['mean']})['value']).opts(ylabel="Temperature", title='Year/Month Mean Temperature

# Display the count and mean temperature plots
(count + mean).opts(opts.Bars(width=380, height=300,tools=['hover'],show_grid=True, stacked=True, legend_position='bottom'))
```

```
/usr/local/lib/python3.10/dist-packages/holoviews/plotting/bokeh/plot.py:987: UserWarnir
  layout_plot = gridplot(
/usr/local/lib/python3.10/dist-packages/holoviews/plotting/bokeh/plot.py:987: UserWarnir
  layout_plot = gridplot(
```

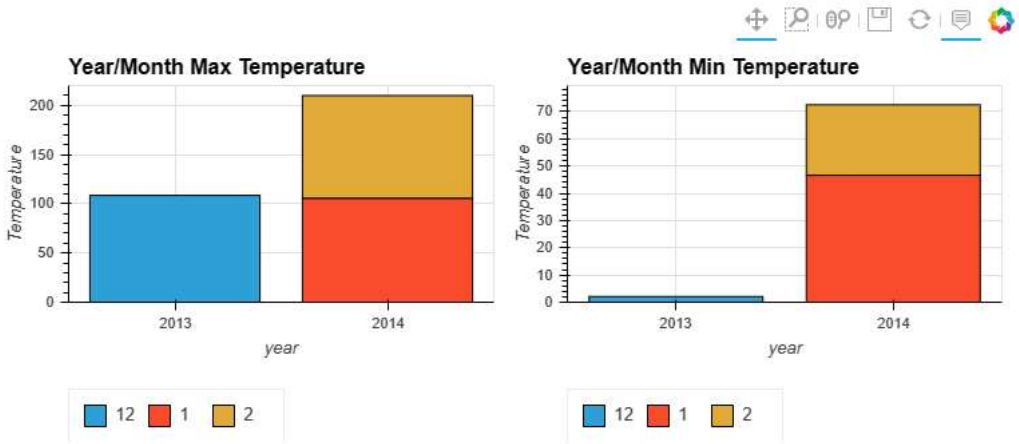



```
import plotly.graph_objects as go

# Calculate the maximum and minimum temperatures per year and month
year_maxmin = df.groupby(['year','month']).agg({'value': ['min', 'max']})

# Create traces for maximum and minimum temperature
max_trace = go.Bar(x=year_maxmin.index, y=year_maxmin['value']['max'], name='Max Temperature')
min_trace = go.Bar(x=year_maxmin.index, y=year_maxmin['value']['min'], name='Min Temperature')

# Create layout
layout = go.Layout(title='Year/Month Max and Min Temperature', xaxis_title='Index', yaxis_title='Temperature')

# Create figure
fig = go.Figure(data=[max_trace, min_trace], layout=layout)

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

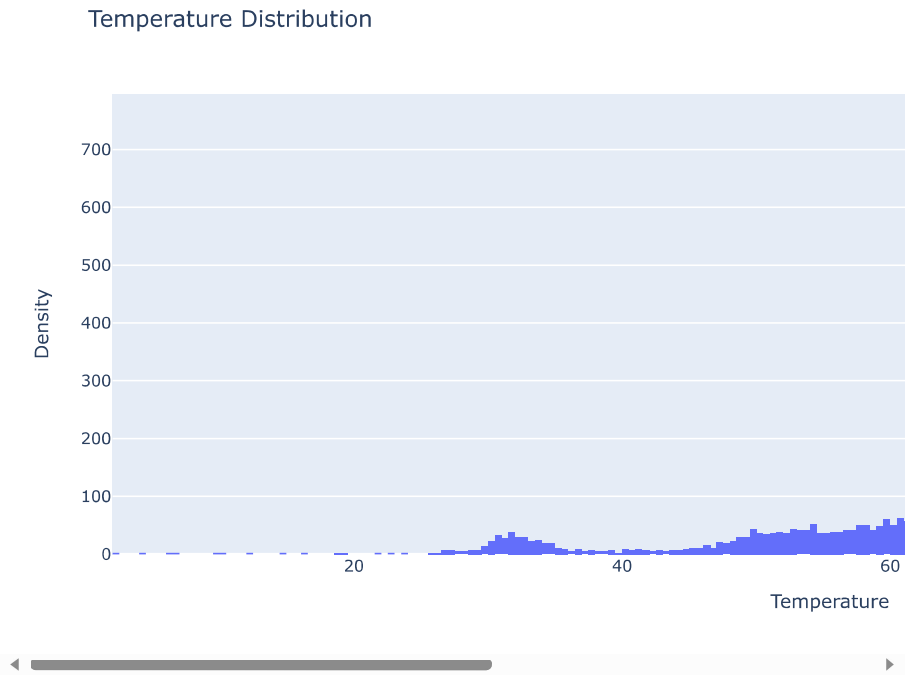


```
# This code generates two plots: one showing the maximum temperature and another showing the minimum temperature per year and month.

# Calculate the maximum and minimum temperatures per year and month
year_maxmin = df.groupby(['year','month']).agg({'value': ['min', 'max']})

# Display the maximum and minimum temperature plots
(hv.Bars(year_maxmin['value']['max']).opts(ylabel="Temperature", title='Year/Month Max Temperature') \
+ hv.Bars(year_maxmin['value']['min']).opts(ylabel="Temperature", title='Year/Month Min Temperature')) \
.opts(opts.Bars(width=380, height=300,tools=['hover'],show_grid=True, stacked=True, legend_position='bottom'))
```

```
/usr/local/lib/python3.10/dist-packages/holoviews/plotting/bokeh/plot.py:987: UserWarnir
  layout_plot = gridplot(
/usr/local/lib/python3.10/dist-packages/holoviews/plotting/bokeh/plot.py:987: UserWarnir
  layout_plot = gridplot(
```

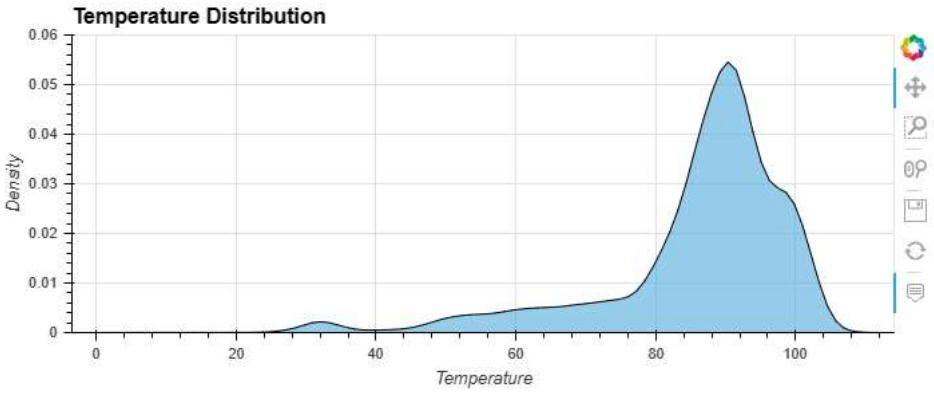



```
import plotly.express as px

# Create a histogram for temperature distribution
fig = px.histogram(df, x='value', title='Temperature Distribution',
                   labels={'value': 'Temperature', 'density': 'Density'})

# Set the layout
fig.update_layout(xaxis_title='Temperature', yaxis_title='Density', showlegend=False)

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

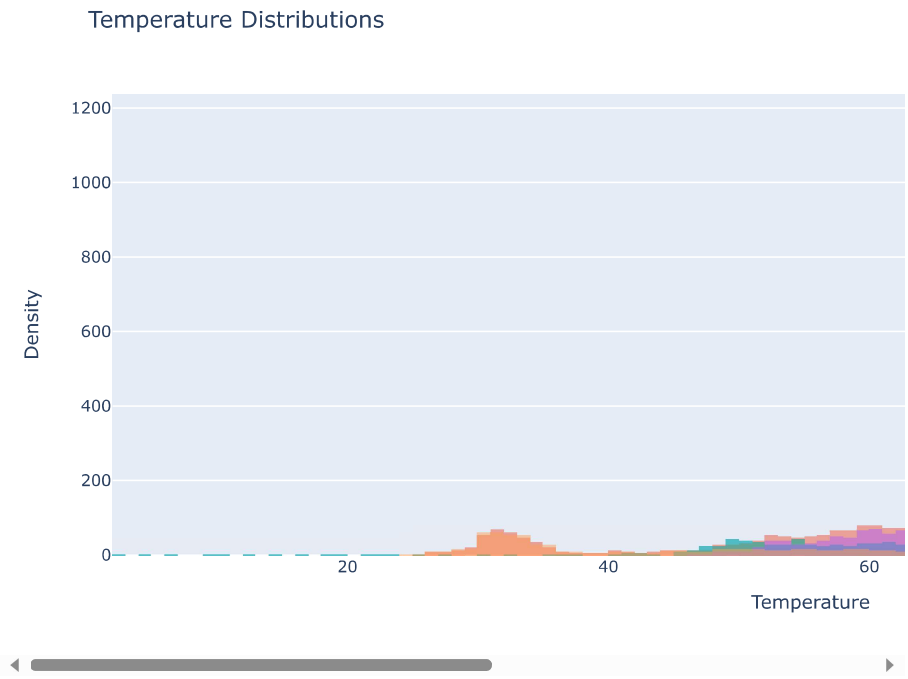



```
# Display the distribution of temperatures
hv.Distribution(df['value']).opts(opts.Distribution(title="Temperature Distribution", xlabel="Temperature", ylabel="Density", width=700, hei
```

```
/usr/local/lib/python3.10/dist-packages/holoviews/core/util.py:1585: PanelDeprecationWar
  value = param_value_if_widget(value)
```

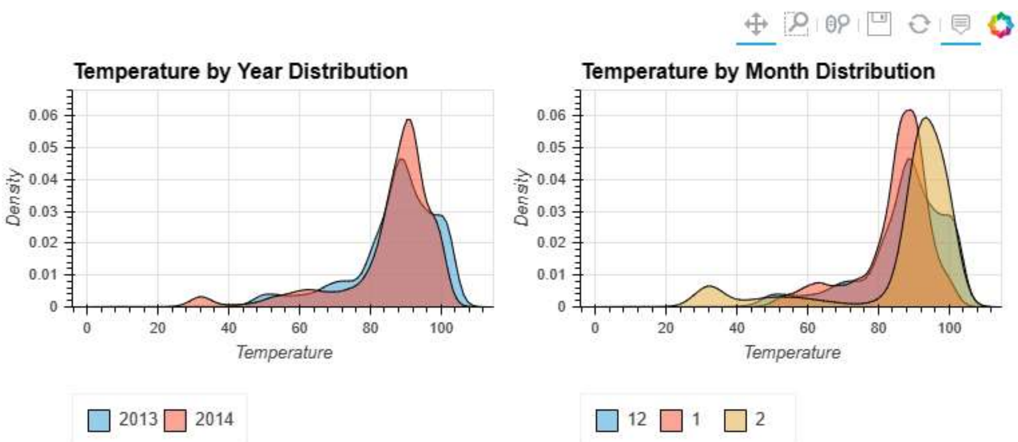


```
import plotly.graph_objects as go

# Create subplots for temperature distributions by year and by month
fig = go.Figure()

# Add traces for temperature distributions by year
for year in [2013, 2014]:
    fig.add_trace(go.Histogram(x=df.loc[df['year']==year, 'value'],
                               name=str(year),
                               histnorm='density',
                               opacity=0.5))

# Add traces for temperature distributions by month
for month in [12, 1, 2]:
    fig.add_trace(go.Histogram(x=df.loc[df['month']==month, 'value'],
                               name=str(month),
                               histnorm='density',
                               opacity=0.5))

# Update layout
fig.update_layout(title="Temperature Distributions",
                  xaxis_title="Temperature",
                  yaxis_title="Density",
                  barmode='overlay',
                  showlegend=True,
                  legend_title="Category")

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



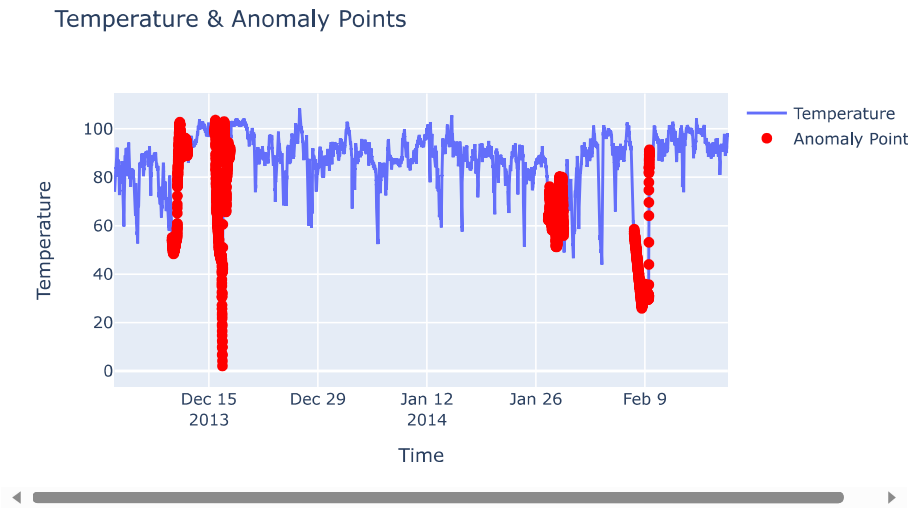

```
# Display the temperature distributions by year and by month
((hv.Distribution(df.loc[df['year']==2013,'value'], label='2013') * hv.Distribution(df.loc[df['year']==2014,'value'], label='2014')).opts(ti
(hv.Distribution(df.loc[df['month']==12,'value'], label='12') * hv.Distribution(df.loc[df['month']==1,'value'], label='1') \
     * hv.Distribution(df.loc[df['month']==2,'value'], label='2')).opts(title="Temperature by Month Distribution", legend_position='bottom')
     .opts(opts.Distribution(xlabel="Temperature", ylabel="Density", width=380, height=300,tools=['hover'],show_grid=True))
```

```
/usr/local/lib/python3.10/dist-packages/holoviews/core/util.py:1585: PanelDeprecationWar
  value = param_value_if_widget(value)
/usr/local/lib/python3.10/dist-packages/holoviews/core/util.py:1585: PanelDeprecationWar
  value = param_value_if_widget(value)
/usr/local/lib/python3.10/dist-packages/holoviews/core/util.py:1585: PanelDeprecationWar
  value = param_value_if_widget(value)
/usr/local/lib/python3.10/dist-packages/holoviews/core/util.py:1585: PanelDeprecationWar
  value = param_value_if_widget(value)
/usr/local/lib/python3.10/dist-packages/holoviews/core/util.py:1585: PanelDeprecationWar
  value = param_value_if_widget(value)
/usr/local/lib/python3.10/dist-packages/holoviews/plotting/bokeh/plot.py:987: UserWarnir
  layout_plot = gridplot(
/usr/local/lib/python3.10/dist-packages/holoviews/plotting/bokeh/plot.py:987: UserWarnir
  layout_plot = gridplot(
```

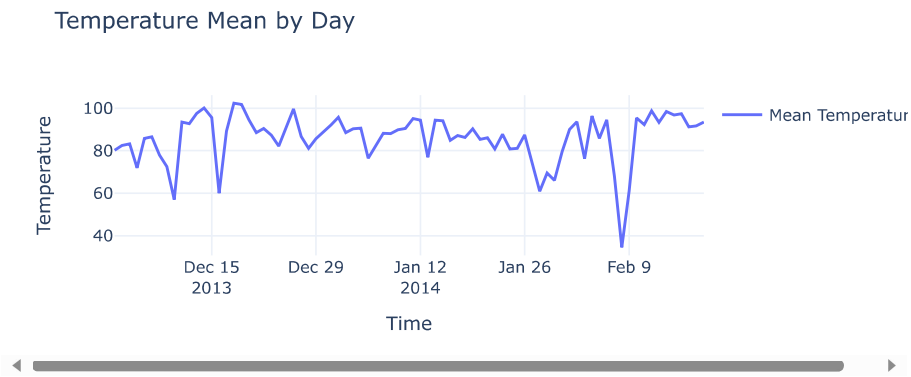


Time Series Analysis

```
import plotly.graph_objects as go

# Extract anomaly points
anomalies = df[df['anomaly']==1]

# Create the plot
fig = go.Figure()

# Add temperature curve
fig.add_trace(go.Scatter(x=df.index, y=df['value'], mode='lines', name='Temperature'))

# Add anomaly points
fig.add_trace(go.Scatter(x=anomalies.index, y=anomalies['value'], mode='markers', name='Anomaly Points', marker=dict(color='red', size=8)))

# Update layout
fig.update_layout(title="Temperature & Anomaly Points",
                  xaxis_title="Time",
                  yaxis_title="Temperature",
                  width=700,
                  height=400,
                  showlegend=True)

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




```
import plotly.graph_objects as go

# Resample data to calculate mean temperature by day
mean_temperature_by_day = df['value'].resample('D').mean()

# Create the plot
fig = go.Figure()

# Add mean temperature curve
fig.add_trace(go.Scatter(x=mean_temperature_by_day.index, y=mean_temperature_by_day, mode='lines', name='Mean Temperature'))

# Update layout
fig.update_layout(title="Temperature Mean by Day",
                  xaxis_title="Time",
                  yaxis_title="Temperature",
                  width=700,
                  height=300,
                  showlegend=True,
                  hovermode='x unified',
                  template='plotly_white')

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



Model1. Hotelling's T2

Hotelling's T^2 (T-squared) statistic is a multivariate statistical method used for detecting outliers or anomalies in a dataset. It is an extension of the univariate t-test to multiple dimensions.

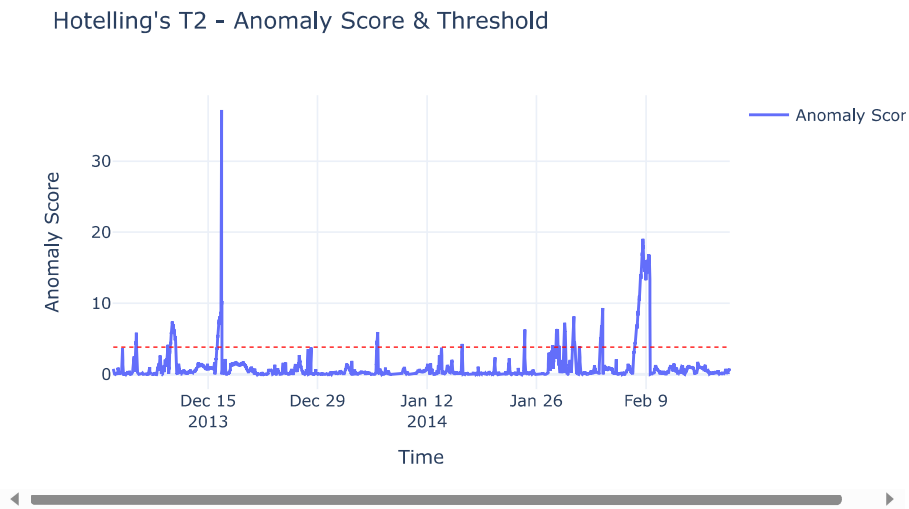
In the context of anomaly detection, Hotelling's T^2 statistic measures how far each data point is from the mean of the dataset in a multivariate space. It calculates the distance of each data point from the centroid of the dataset, considering the covariance between different dimensions.

The anomaly score is computed using Hotelling's T^2 statistic, and it represents the degree of deviation of each data point from the centroid of the dataset. Higher anomaly scores indicate greater deviation from the norm and are suggestive of potential anomalies.

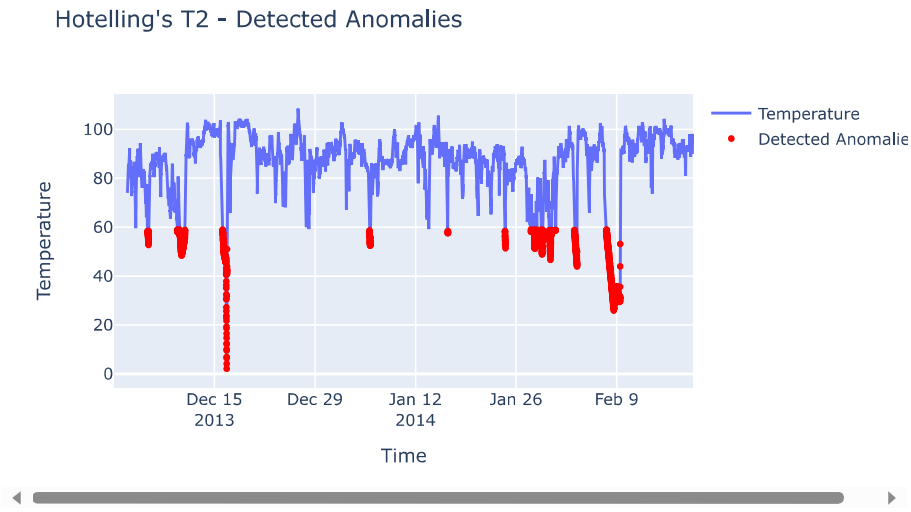
The threshold is a predetermined value that serves as a cutoff point for determining whether a data point is considered an anomaly. It is often based on statistical properties of the data distribution, such as percentiles of the chi-square distribution.

In anomaly detection applications, if the computed anomaly score for a data point exceeds the threshold, that data point is flagged as an anomaly. Otherwise, it is considered normal. This threshold provides a criterion for distinguishing between normal and abnormal behavior in the dataset.

```
import plotly.graph_objects as go

# Calculate anomaly score
hotelling_df = pd.DataFrame()
hotelling_df['value'] = df['value']
mean = hotelling_df['value'].mean()
std = hotelling_df['value'].std()
hotelling_df['anomaly_score'] = ((hotelling_df['value'] - mean) / std) ** 2

# Calculate anomaly threshold using chi-square distribution
hotelling_df['anomaly_threshold'] = stats.chi2.ppf(q=0.95, df=1)

# Identify anomalies based on the anomaly score and threshold
hotelling_df['anomaly'] = np.where(hotelling_df['anomaly_score'] > hotelling_df['anomaly_threshold'], 1, 0)

# Create the plot
fig = go.Figure()

# Add anomaly score curve
fig.add_trace(go.Scatter(x=hotelling_df.index, y=hotelling_df['anomaly_score'], mode='lines', name='Anomaly Score'))

# Add threshold line
fig.add_shape(type='line', x0=hotelling_df.index[0], y0=hotelling_df['anomaly_threshold'].iloc[0], x1=hotelling_df.index[-1], y1=hotelling_d
              line=dict(color='red', width=1, dash='dot'), name='Threshold')

# Update layout
fig.update_layout(title="Hotelling's T2 - Anomaly Score & Threshold",
                  xaxis_title="Time",
                  yaxis_title="Anomaly Score",
                  width=700,
                  height=400,
                  showlegend=True,
                  hovermode='x unified',
                  template='plotly_white')

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



```
import plotly.graph_objs as go

# Extract anomalies
anomalies = hotelling_df[hotelling_df['anomaly'] == 1]

# Create plot traces for temperature and anomalies
trace_temp = go.Scatter(x=hotelling_df.index, y=hotelling_df['value'], mode='lines', name='Temperature')
trace_anomalies = go.Scatter(x=anomalies.index, y=anomalies['value'], mode='markers', marker=dict(color='red', size=5), name='Detected Anoma

# Create layout
layout = go.Layout(title="Hotelling's T2 - Detected Anomalies", xaxis=dict(title='Time'), yaxis=dict(title='Temperature'), width=700, height

# Create figure
fig = go.Figure(data=[trace_temp, trace_anomalies], layout=layout)

# Show figure
fig.show()
```




```
from sklearn.metrics import f1_score

# Calculate F1 score
hotelling_f1 = f1_score(df['anomaly'], hotelling_df['anomaly'])

# Print F1 score
print(f'Hotelling\'s T2 F1 Score: {hotelling_f1}')
```

```
Hotelling's T2 F1 Score: 0.5440778799351
```

```
from sklearn.metrics import classification_report

# Generate classification report
report = classification_report(df['anomaly'], hotelling_df['anomaly'])

# Print classification report
print(report)
```

```
                precision    recall  f1-score   support

             0       0.94      0.98      0.96     20427
             1       0.70      0.44      0.54      2268

      accuracy                           0.93     22695
     macro avg       0.82      0.71      0.75     22695
  weighted avg       0.92      0.93      0.92     22695
```

Model2. One-Class SVM

One-Class SVM (Support Vector Machine) is a machine learning algorithm used for anomaly detection. Unlike traditional SVMs, which are typically used for binary classification tasks, One-Class SVM is trained on only one class of data, often referred to as the "normal" or "inlier" class. The algorithm learns the structure of this class and then identifies anomalies as instances that deviate significantly from this learned structure.

Here's how One-Class SVM works:

Training Phase: During training, One-Class SVM learns a decision boundary that encloses the normal data points in a high-dimensional space. The goal is to find the smallest possible region that contains the majority of the data points.

Testing Phase: During testing or inference, the algorithm evaluates new data points and determines whether they fall within the learned boundary. Points outside this boundary are classified as anomalies or outliers.

- Key characteristics of One-Class SVM include:

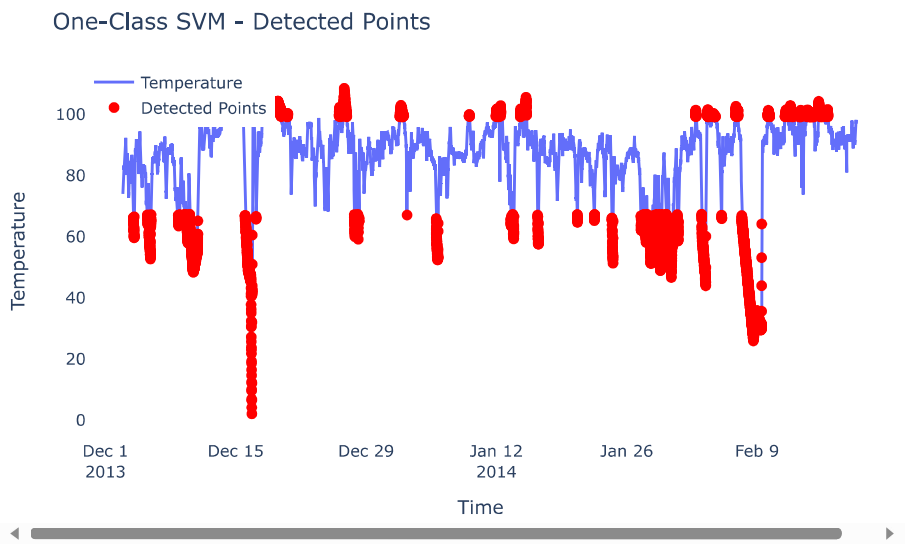
Kernel Trick: Like traditional SVMs, One-Class SVM can use kernel functions to map input data into a higher-dimensional space, allowing it to capture complex nonlinear relationships between features.

Outlier Sensitivity: The performance of One-Class SVM depends on the choice of parameters and the representation of normal data during training. It may struggle with highly imbalanced datasets where anomalies are rare compared to normal instances.

Scalability: One-Class SVM can be computationally expensive, especially when dealing with large datasets, as it involves solving a quadratic optimization problem. However, approximation techniques and optimizations exist to improve scalability.

Overall, One-Class SVM is a powerful technique for detecting anomalies in scenarios where only normal data is available for training. It is commonly used in various applications such as fraud detection, intrusion detection, and fault detection in industrial systems.

```
import plotly.graph_objects as go

# Fit One-Class SVM model
ocsvm_model = OneClassSVM(nu=0.2, gamma=0.001, kernel='rbf')
ocsvm_ret = ocsvm_model.fit_predict(df['value'].values.reshape(-1, 1))

# Create DataFrame to store results
ocsvm_df = pd.DataFrame()
ocsvm_df['value'] = df['value']
ocsvm_df['anomaly'] = [1 if i == -1 else 0 for i in ocsvm_ret]

# Find anomalies
anomalies = ocsvm_df[ocsvm_df['anomaly'] == 1]

# Plot temperature data and detected anomalies
fig = go.Figure()

# Plot temperature data
fig.add_trace(go.Scatter(x=ocsvm_df.index, y=ocsvm_df['value'], mode='lines', name='Temperature'))

# Plot detected anomalies
fig.add_trace(go.Scatter(x=anomalies.index, y=anomalies['value'], mode='markers', marker=dict(color='red', size=8), name='Detected Points'))

# Customize layout
fig.update_layout(title="One-Class SVM - Detected Points",
                  xaxis_title="Time",
                  yaxis_title="Temperature",
                  width=700,
                  height=400,
                  hovermode='closest',
                  showlegend=True,
                  legend=dict(x=0, y=1),
                  margin=dict(l=50, r=50, t=50, b=50),
                  plot_bgcolor='rgba(0,0,0,0)')
fig.show()
```




```
from sklearn.metrics import f1_score

# Calculate F1 score
ocsvm_f1 = f1_score(df['anomaly'], ocsvm_df['anomaly'])

# Print F1 score
print(f'One-Class SVM F1 Score: {ocsvm_f1}')
```

```
One-Class SVM F1 Score: 0.4224441833137485
```

```
from sklearn.metrics import classification_report

# Generate classification report
report = classification_report(df['anomaly'], ocsvm_df['anomaly'])

# Print classification report
print(report)
```

```
                  precision    recall  f1-score   support

               0       0.95      0.85      0.90     20427
               1       0.32      0.63      0.42      2268

        accuracy                           0.83     22695
       macro avg       0.64      0.74      0.66     22695
    weighted avg       0.89      0.83      0.85     22695
```

Model3. Isolation Forest

Isolation Forest is an anomaly detection algorithm that works by isolating anomalies in the dataset. It builds an ensemble of decision trees, where each tree is trained on a random subset of the data. During training, the algorithm tries to isolate anomalies by recursively partitioning the data into subsets, such that anomalies end up in smaller partitions and are isolated faster than normal data points.

The key idea behind Isolation Forest is that anomalies are typically few in number and are different from normal instances in the dataset. As a result, they require fewer splits to be isolated from the rest of the data. By measuring the average path length needed to isolate a data point across all trees in the forest, Isolation Forest can identify anomalies as instances with shorter average path lengths.

Here are some key points about Isolation Forest:

Random Partitioning: Each decision tree in the ensemble is built by randomly selecting a feature and then choosing a random split value within the range of that feature.

Isolation of Anomalies: Anomalies are expected to have shorter average path lengths to the root of the tree compared to normal instances. This is because they are typically isolated more quickly due to their different nature.

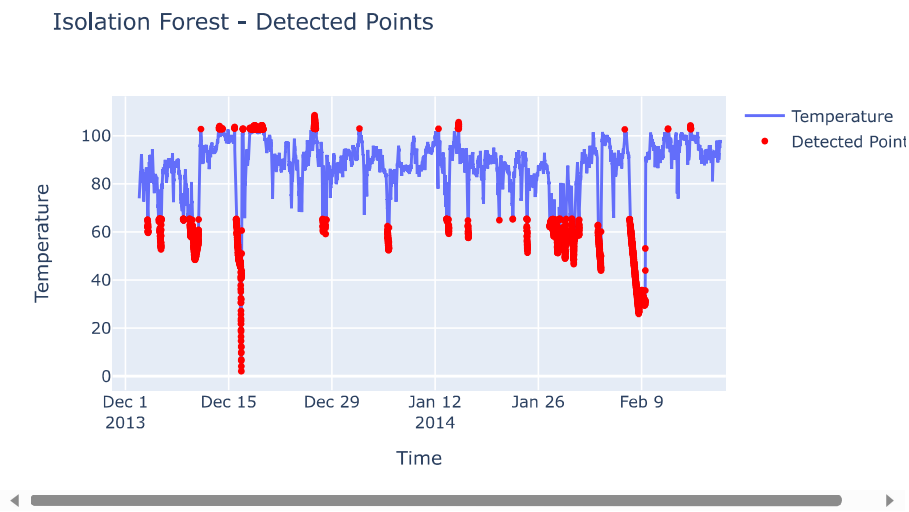
Scalability: Isolation Forest is particularly suitable for large datasets as it can efficiently handle high-dimensional data and is relatively insensitive to the size of the dataset.

Unsupervised Learning: Isolation Forest is an unsupervised learning algorithm, meaning it does not require labeled data to train. It automatically identifies anomalies based on the structure of the data.

Robustness: Isolation Forest is robust to outliers and noise in the data. It does not rely on assumptions about the distribution of the data and can handle skewed or imbalanced datasets effectively.

Overall, Isolation Forest is a powerful and efficient algorithm for anomaly detection, especially in scenarios where anomalies are rare and distinct from normal instances.

```
import plotly.graph_objects as go

# Define Isolation Forest model
iforest_model = IsolationForest(n_estimators=300, contamination=0.1, max_samples=700)

# Fit model and predict anomalies
iforest_ret = iforest_model.fit_predict(df['value'].values.reshape(-1, 1))

# Create DataFrame to store values and anomalies
iforest_df = pd.DataFrame()
iforest_df['value'] = df['value']
iforest_df['anomaly']  = [1 if i==-1 else 0 for i in iforest_ret]

# Filter anomalies
anomalies = [[ind, value] for ind, value in zip(iforest_df[iforest_df['anomaly']==1].index, iforest_df.loc[iforest_df['anomaly']==1,'value']

# Create Plotly figure
fig = go.Figure()

# Add curve for temperature values
fig.add_trace(go.Scatter(x=iforest_df.index, y=iforest_df['value'], mode='lines', name='Temperature'))

# Add points for detected anomalies
fig.add_trace(go.Scatter(x=[point[0] for point in anomalies], y=[point[1] for point in anomalies], mode='markers', marker=dict(color='red',

# Update layout
fig.update_layout(title="Isolation Forest - Detected Points", xaxis_title="Time", yaxis_title="Temperature", width=700, height=400, showlege

# Show plot
fig.show()
```




```
# Calculate F1 score for Isolation Forest
iforest_f1 = f1_score(df['anomaly'], iforest_df['anomaly'])

# Print the F1 score
print(f'Isolation Forest F1 Score : {iforest_f1}')
```

```
Isolation Forest F1 Score : 0.5315116791538123
```

```
from sklearn.metrics import classification_report
```

```
# Generate classification report
report = classification_report(df['anomaly'], iforest_df['anomaly'])

# Print classification report
print(report)
```

```
                   precision    recall  f1-score   support

               0       0.95      0.95      0.95     20427
               1       0.53      0.53      0.53      2268

        accuracy                           0.91     22695
       macro avg       0.74      0.74      0.74     22695
    weighted avg       0.91      0.91      0.91     22695
```

Model4. LOF

LOF, or Local Outlier Factor, is a popular algorithm used in anomaly detection to identify outliers in datasets. It assesses the local density deviation of a data point with respect to its neighbors. The basic idea is that outliers tend to have significantly lower density than their neighbors.

Here's how LOF works:

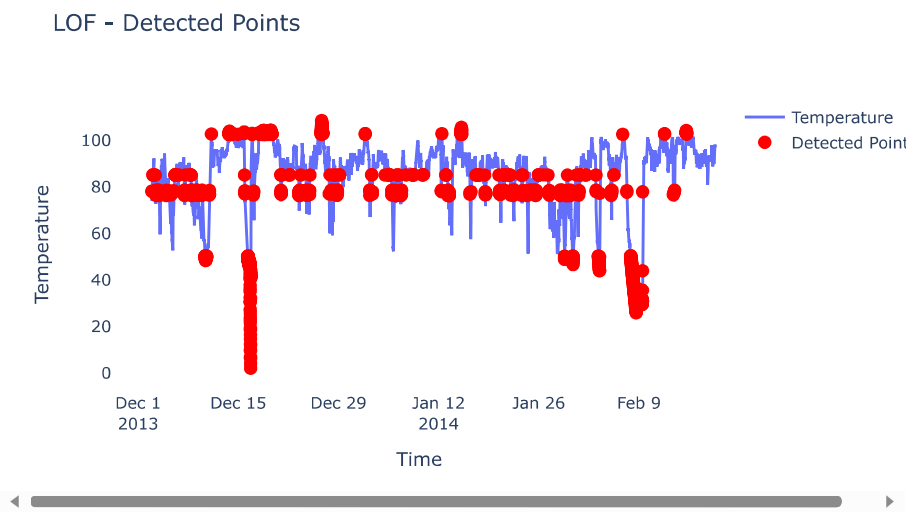
Calculate Local Density: For each data point, LOF computes its local density by estimating the distance to its k-nearest neighbors. This distance is typically measured using Euclidean distance, but other distance metrics can also be used.

Compute Reachability Distance: LOF computes the reachability distance for each point, which represents how far a data point is from its neighbors in terms of density. It is computed as the maximum of the distance to the k-nearest neighbor and the local density of the neighbor itself.

Calculate Local Outlier Factor: Finally, LOF calculates the local outlier factor for each data point. This factor represents how much the density of a point differs from the densities of its neighbors. Points with significantly lower densities compared to their neighbors are considered outliers and have higher LOF values.

By analyzing the LOF values, analysts can identify data points that deviate significantly from their local neighborhoods, indicating potential anomalies or outliers. LOF is effective in detecting outliers in high-dimensional datasets and is robust to variations in data distribution and density.

```
import plotly.graph_objects as go

# Create LOF model
lof_model = LocalOutlierFactor(n_neighbors=500, contamination=0.07)

# Fit the model and predict anomalies
lof_ret = lof_model.fit_predict(df['value'].values.reshape(-1, 1))

# Create DataFrame to store values and anomalies
lof_df = pd.DataFrame()
lof_df['value'] = df['value']
lof_df['anomaly'] = [1 if i == -1 else 0 for i in lof_ret]

# Filter anomalies
anomalies = lof_df[lof_df['anomaly'] == 1]

# Create scatter plot for temperature values and anomalies
fig = go.Figure()
fig.add_trace(go.Scatter(x=lof_df.index, y=lof_df['value'], mode='lines', name='Temperature'))
fig.add_trace(go.Scatter(x=anomalies.index, y=anomalies['value'], mode='markers', marker=dict(color='red', size=10), name='Detected Points'))
fig.update_layout(title='LOF - Detected Points', xaxis_title='Time', yaxis_title='Temperature', showlegend=True, width=700, height=400, hove
fig.show()
```




```
from sklearn.metrics import f1_score

# Calculate F1 score for LOF
lof_f1 = f1_score(df['anomaly'], lof_df['anomaly'])

# Print F1 score
print(f'LOF F1 Score : {lof_f1}')
```

```
LOF F1 Score : 0.3577910292973813
```

```
from sklearn.metrics import classification_report

# Generate classification report
report = classification_report(df['anomaly'], lof_df['anomaly'])

# Print classification report
print(report)
```

```
                  precision    recall  f1-score   support

               0       0.93      0.96      0.94     20427
               1       0.43      0.30      0.36      2268

        accuracy                           0.89     22695
       macro avg       0.68      0.63      0.65     22695
    weighted avg       0.88      0.89      0.88     22695
```

Model5. ChangeFinder

ChangeFinder is a method used in anomaly detection to identify abrupt changes or shifts in the underlying distribution of a time series dataset. It is particularly effective for detecting sudden changes in data patterns, which may indicate anomalies or significant events. Here's how it works:

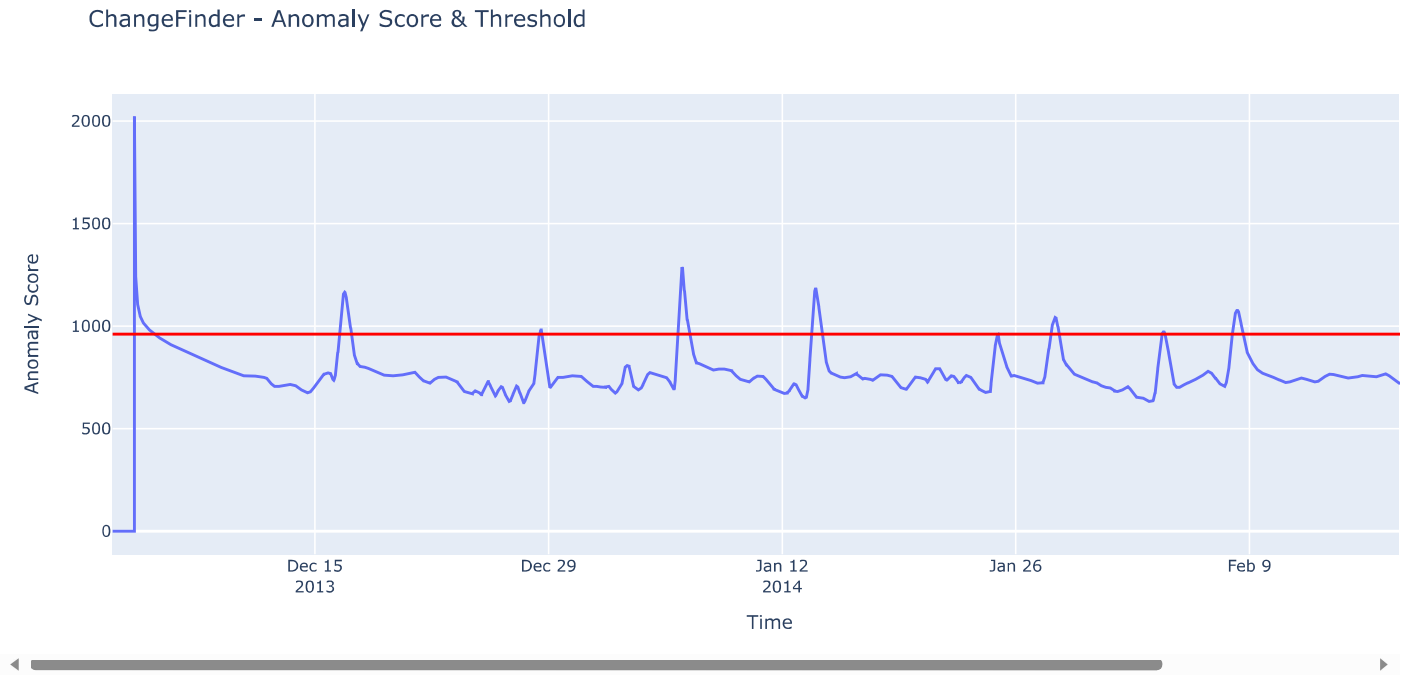
- Calculation of Change Score: ChangeFinder computes a change score for each data point in the time series. This change score quantifies the deviation of the current data point from the expected or typical pattern observed in the data history.
- Detection of Change Points: By analyzing the change scores over time, ChangeFinder identifies change points where significant deviations occur. These change points represent instances of sudden changes or anomalies in the data.
- Thresholding: ChangeFinder often employs a thresholding mechanism to distinguish between normal variations and anomalous changes. When the change score exceeds a certain threshold, it indicates the presence of an anomaly.
- Adaptation to Dynamic Environments: ChangeFinder is capable of adapting to dynamic environments where the underlying data distribution may change over time. It continuously updates its model based on incoming data, ensuring robustness to evolving patterns.

Applications of ChangeFinder in industry include:

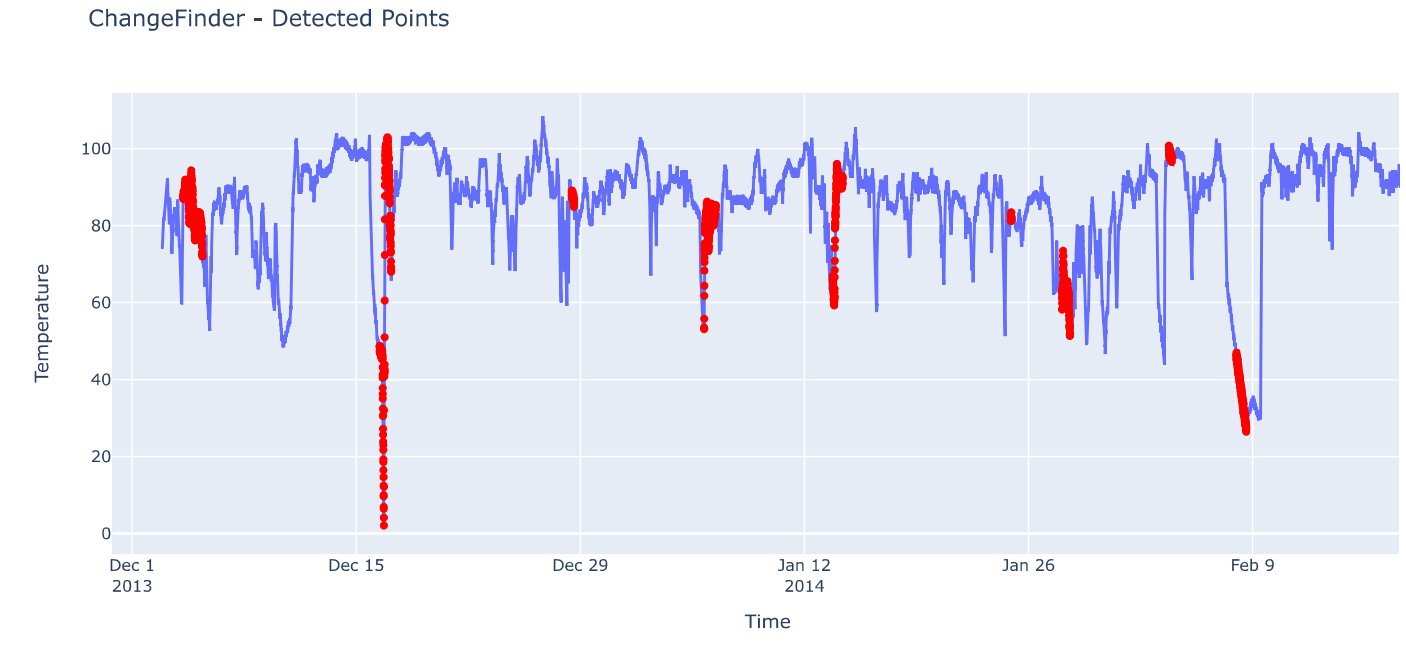
- Fault Detection in Machinery: ChangeFinder can be used to monitor sensor data from industrial machinery such as turbines, pumps, or motors. Sudden deviations in sensor readings may indicate equipment malfunctions or failures, enabling timely maintenance or intervention to prevent downtime.
- Anomaly Detection in Network Traffic: In cybersecurity applications, ChangeFinder can analyze network traffic data to detect unusual patterns or anomalies that may indicate security breaches, network intrusions, or denial-of-service attacks.
- Quality Control in Manufacturing: ChangeFinder can help identify deviations from expected quality standards in manufacturing processes. By monitoring sensor data from production lines, it can detect anomalies such as defects or variations in product quality.
- Environmental Monitoring: ChangeFinder is used in environmental monitoring systems to detect sudden changes in air quality, water quality, or weather conditions. It can help identify pollution events, natural disasters, or abnormal environmental phenomena.

Overall, ChangeFinder provides a powerful tool for real-time anomaly detection in various industrial applications, allowing organizations to detect and respond to unexpected events efficiently.

```
import plotly.graph_objects as go

# Create ChangeFinder model
cf_model = changefinder.ChangeFinder(r=0.002, order=1, smooth=250)

# Create DataFrame for ChangeFinder
ch_df = pd.DataFrame()
ch_df['value'] = df['value']

# Update anomaly scores using ChangeFinder
ch_df['anomaly_score'] = [cf_model.update(i) for i in ch_df['value']]

# Calculate anomaly score percentiles
ch_score_q1 = np.percentile(ch_df['anomaly_score'], 25)
ch_score_q3 = np.percentile(ch_df['anomaly_score'], 75)

# Calculate anomaly threshold
ch_df['anomaly_threshold'] = ch_score_q3 + (ch_score_q3 - ch_score_q1) * 3

# Define anomalies
ch_df['anomaly'] = ch_df.apply(lambda x: 1 if x['anomaly_score'] > x['anomaly_threshold'] else 0, axis=1)

# Create traces for anomaly score and threshold
anomaly_score_trace = go.Scatter(x=ch_df.index, y=ch_df['anomaly_score'], mode='lines', name='Anomaly Score')
threshold_trace = go.Scatter(x=ch_df.index, y=ch_df['anomaly_threshold'], mode='lines', name='Threshold', line=dict(color='red'))

# Create layout
layout = go.Layout(title="ChangeFinder - Anomaly Score & Threshold", xaxis=dict(title="Time"), yaxis=dict(title="Anomaly Score"))

# Create figure
fig = go.Figure(data=[anomaly_score_trace, threshold_trace], layout=layout)

# Show figure
fig.show()
```




```
import plotly.graph_objects as go

# Define anomalies
anomalies = [[ind, value] for ind, value in zip(ch_df[ch_df['anomaly']==1].index, ch_df.loc[ch_df['anomaly']==1,'value'])]

# Create traces for temperature and detected points
temperature_trace = go.Scatter(x=ch_df.index, y=ch_df['value'], mode='lines', name='Temperature')
anomalies_trace = go.Scatter(x=[anomaly[0] for anomaly in anomalies], y=[anomaly[1] for anomaly in anomalies], mode='markers', name='Detecte

# Create layout
layout = go.Layout(title="ChangeFinder - Detected Points", xaxis=dict(title="Time"), yaxis=dict(title="Temperature"))

# Create figure
fig = go.Figure(data=[temperature_trace, anomalies_trace], layout=layout)

# Show figure
fig.show()
```



```
from sklearn.metrics import f1_score

# Calculate F1 score
ch_f1 = f1_score(df['anomaly'], ch_df['anomaly'])

# Print F1 score
print(f'ChangeFinder F1 Score : {ch_f1}')
```

```
ChangeFinder F1 Score : 0.2928434329585961
```

```
from sklearn.metrics import classification_report

# Generate classification report
report = classification_report(df['anomaly'], ch_df['anomaly'])

# Print classification report
print(report)
```

```
                   precision    recall  f1-score   support

                0       0.92      0.96      0.94     20427
                1       0.39      0.24      0.29      2268

         accuracy                           0.89     22695
        macro avg       0.65      0.60      0.62     22695
     weighted avg       0.87      0.89      0.87     22695
```

Model6. Variance Based Method

In anomaly detection, the Variance Based Method involves using the variance of a dataset to identify anomalies. Here's how it typically works:

- Calculate Variance: First, the variance of the dataset is computed. Variance measures the dispersion of values in the dataset around the mean. Higher variance indicates greater spread of data points.
- Set Thresholds: Based on the calculated variance, thresholds are determined. These thresholds define the range within which most of the data points are expected to lie.
- Detect Anomalies: Data points that fall outside the defined threshold range are considered anomalies. These points are unusual compared to the majority of the data and may represent outliers or abnormalities.
- Flag Anomalies: Anomalies are then flagged or labeled for further investigation or action. This could involve alerting system administrators, triggering corrective actions, or performing more in-depth analysis.

Here's an example of how the Variance Based Method might be applied:

Suppose you have a dataset of temperature readings from a sensor. You calculate the variance of these readings over a certain period. Based on the variance, you set thresholds to identify temperatures that are significantly different from the norm. Any temperature readings outside these thresholds are flagged as anomalies, indicating potential issues such as equipment malfunction or environmental changes.

The key advantage of the Variance Based Method is its simplicity and ease of implementation. However, it may not be suitable for all types of data or scenarios, especially when data distribution is complex or non-Gaussian. Additionally, choosing appropriate threshold values can be challenging and may require domain knowledge or experimentation.

```
import plotly.graph_objects as go

# Calculate anomaly thresholds
sigma_df = pd.DataFrame()
sigma_df['value'] = df['value']
std = sigma_df['value'].std()
sigma_df['anomaly_threshold_3r'] = mean + 1.5 * std
sigma_df['anomaly_threshold_3l'] = mean - 1.5 * std
sigma_df['anomaly'] = sigma_df.apply(lambda x: 1 if (x['value'] > x['anomaly_threshold_3r']) or (x['value'] < x['anomaly_threshold_3l']) els

# Identify anomalies
anomalies = [[ind, value] for ind, value in zip(sigma_df[sigma_df['anomaly'] == 1].index, sigma_df.loc[sigma_df['anomaly'] == 1, 'value'])]

# Create plot
fig = go.Figure()
fig.add_trace(go.Scatter(x=sigma_df.index, y=sigma_df['value'], mode='lines', name='Temperature'))
fig.add_trace(go.Scatter(x=[x[0] for x in anomalies], y=[x[1] for x in anomalies], mode='markers', marker=dict(color='red', size=2), name='D
fig.update_layout(title="Variance Based Method - Detected Points", xaxis_title="Time", yaxis_title="Temperature", width=700, height=400, shc
fig.show()
```


```
from sklearn.metrics import f1_score

# Calculate F1 score
sigma_f1 = f1_score(df['anomaly'], sigma_df['anomaly'])

# Print F1 score
print(f'Variance (sigma) Based Method F1 Score : {sigma_f1}')
```

```
Variance (sigma) Based Method F1 Score : 0.585277463193658
```

```
from sklearn.metrics import classification_report
```