```
# Resource used from here along with fragments of code
# https://www.kaggle.com/code/victorambonati/unsupervised-anomaly-detection
# dataset link: https://www.kaggle.com/datasets/boltzmannbrain/nab

import pandas as pd
import numpy as np

import matplotlib
import seaborn
import matplotlib.dates as md
from matplotlib import pyplot as plt

from sklearn import preprocessing
from sklearn.decomposition import PCA
from sklearn.cluster import KMeans

from sklearn.covariance import EllipticEnvelope
#from pyemma import msm # not available on Kaggle Kernel
from sklearn.ensemble import IsolationForest
from sklearn.svm import OneClassSVM
from sklearn.preprocessing import StandardScaler



df = pd.read_csv("machine_temperature_system_failure.csv")

print(df.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 22695 entries, 0 to 22694
Data columns (total 2 columns):
 #   Column     Non-Null Count  Dtype  
---  ------     --------------  -----  
 0   timestamp  22695 non-null  object 
 1   value      22695 non-null  float64
dtypes: float64(1), object(1)
memory usage: 354.7+ KB
None
```

```
print(df['timestamp'].head(10)) # checking the time stamp frequency and format
```

```
0    2013-12-02 21:15:00
1    2013-12-02 21:20:00
2    2013-12-02 21:25:00
3    2013-12-02 21:30:00
4    2013-12-02 21:35:00
5    2013-12-02 21:40:00
6    2013-12-02 21:45:00
7    2013-12-02 21:50:00
8    2013-12-02 21:55:00
```

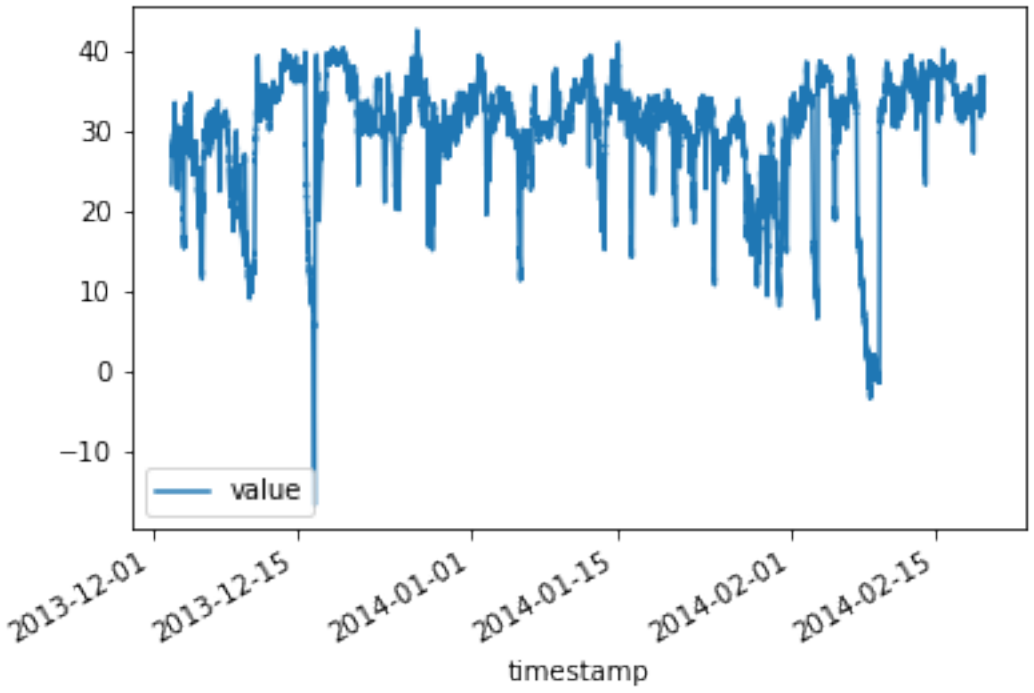
```
9    2013-12-02 22:00:00
Name: timestamp, dtype: object
```

```
# printing the machine temperature mean, median and range
print(df['value'].mean())
print(df['value'].median())
print(df['value'].max()- df['value'].min())
```

```
85.92649821067992
89.40824624
106.425821594
```

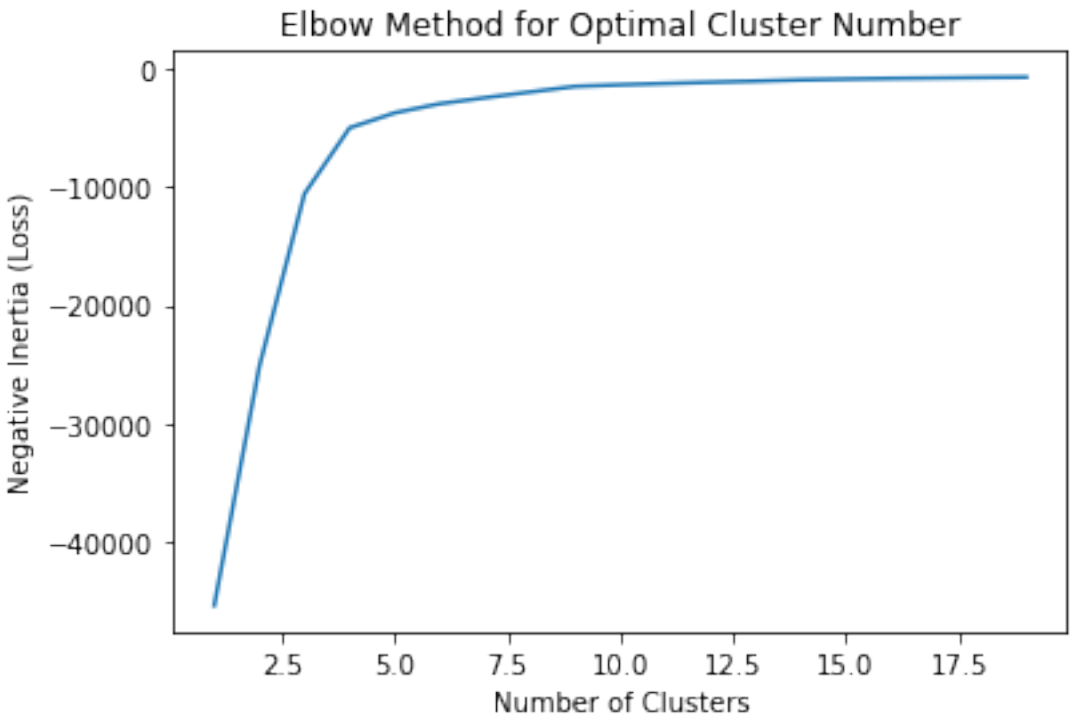
```
df['timestamp'] = pd.to_datetime(df['timestamp'])
# Convert F to °C (temperature mean= 71 -> fahrenheit)
df['value'] = (df['value'] - 32) * 5/9
# plot the data
df.plot(x='timestamp', y='value')
```

```
<AxesSubplot:xlabel='timestamp'>
```




```
# the hours and if it's night or day (7:00-22:00)
df['hours'] = df['timestamp'].dt.hour
df['daylight'] = ((df['hours'] >= 7) & (df['hours'] <=
22)).astype(int)
# the day of the week (Monday=0, Sunday=6) and if it's a week end day
or week day.
df['DayOfTheWeek'] = df['timestamp'].dt.dayofweek
df['WeekDay'] = (df['DayOfTheWeek'] < 5).astype(int)
```

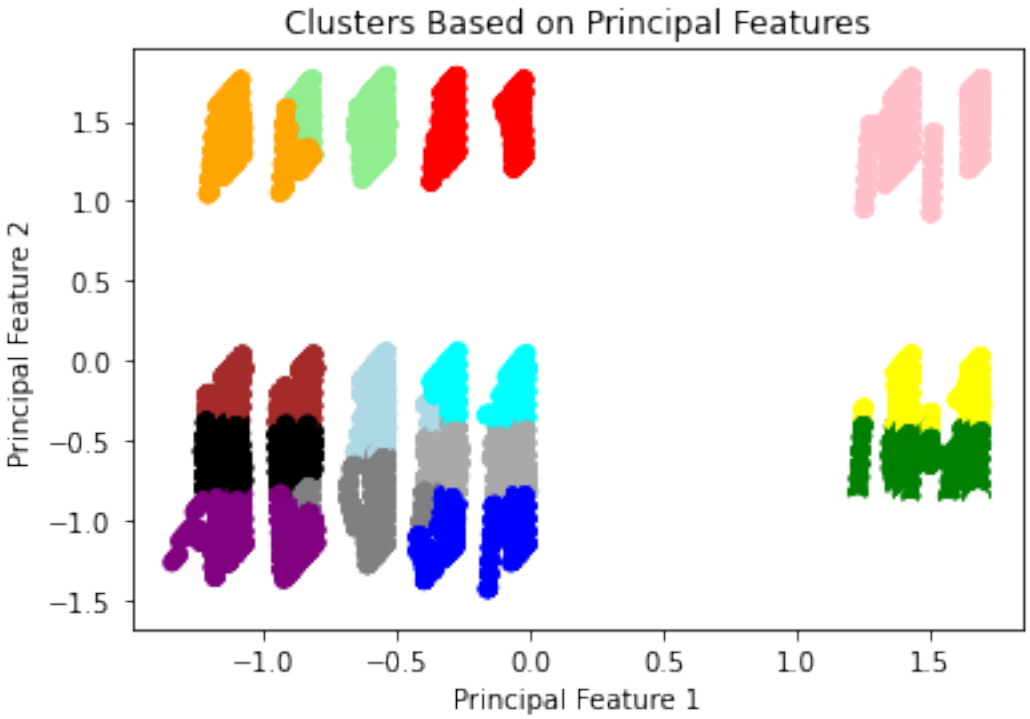
```
# An estimation of anomly population of the dataset (necessary for
several algorithm)
outliers_fraction = 0.01

# Select the columns of interest
selected_columns = ['value', 'hours', 'daylight', 'DayOfTheWeek',
'WeekDay']
data = df[selected_columns]

# Standardize the selected columns
scaler = StandardScaler()
data_scaled = scaler.fit_transform(data)

# Apply PCA to reduce to 2 important features
pca = PCA(n_components=2)
data_pca = pca.fit_transform(data_scaled)

# Standardize the 2 new features
pca_scaler = StandardScaler()
data_pca_scaled = pca_scaler.fit_transform(data_pca)

# Convert the result back to a DataFrame
data = pd.DataFrame(data_pca_scaled, columns=['PCA_1', 'PCA_2'])

# Now, 'data' contains the two standardized principal components as
'PCA_1' and 'PCA_2'

import matplotlib.pyplot as plt
from sklearn.cluster import KMeans

# Define the range of cluster numbers to try
n_clusters = range(1, 20)

# Initialize KMeans models for different cluster numbers
kmeans_models = [KMeans(n_clusters=i).fit(data) for i in n_clusters]

# Calculate the negative inertia (opposite of score) for each model
inertia_scores = [-kmeans.inertia_ for kmeans in kmeans_models]

# Create a plot to visualize the elbow method
plt.plot(n_clusters, inertia_scores)
plt.xlabel('Number of Clusters')
plt.ylabel('Negative Inertia (Loss)')
plt.title('Elbow Method for Optimal Cluster Number')
plt.show()
# used logistic regression
```




```
# Define the range of cluster numbers to try
n_clusters = range(1, 20)

# Initialize KMeans models for different cluster numbers
kmeans_models = [KMeans(n_clusters=i).fit(data) for i in n_clusters]

df['cluster'] = kmeans_models[14].predict(data)
df['principal_feature1'] = data['PCA_1']
df['principal_feature2'] = data['PCA_2']
df['cluster'].value_counts()
```

```
0     1848
3     1848
13    1587
7     1575
9     1538
8     1522
2     1501
6     1486
14    1482
4     1472
10    1412
11    1385
5     1365
1     1346
12    1328
Name: cluster, dtype: int64
```

```
# assigned cluster labels to your data using K-means clustering
data['cluster'] = kmeans_models[14].labels_  # Adjust the index as
needed

# Define colors for clusters
cluster_colors = {
    0: 'red', 1: 'blue', 2: 'green', 3: 'pink', 4: 'black', 5:
'orange',
    6: 'cyan', 7: 'yellow', 8: 'brown', 9: 'purple', 10: 'white',
    11: 'grey', 12: 'lightblue', 13: 'lightgreen', 14: 'darkgrey'
}

# a scatter plot of the different clusters using the two main features
plt.scatter(data['PCA_1'], data['PCA_2'],
c=data["cluster"].apply(lambda x: cluster_colors[x]))

# create axis labels and title
plt.xlabel('Principal Feature 1')
plt.ylabel('Principal Feature 2')
plt.title('Clusters Based on Principal Features')

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

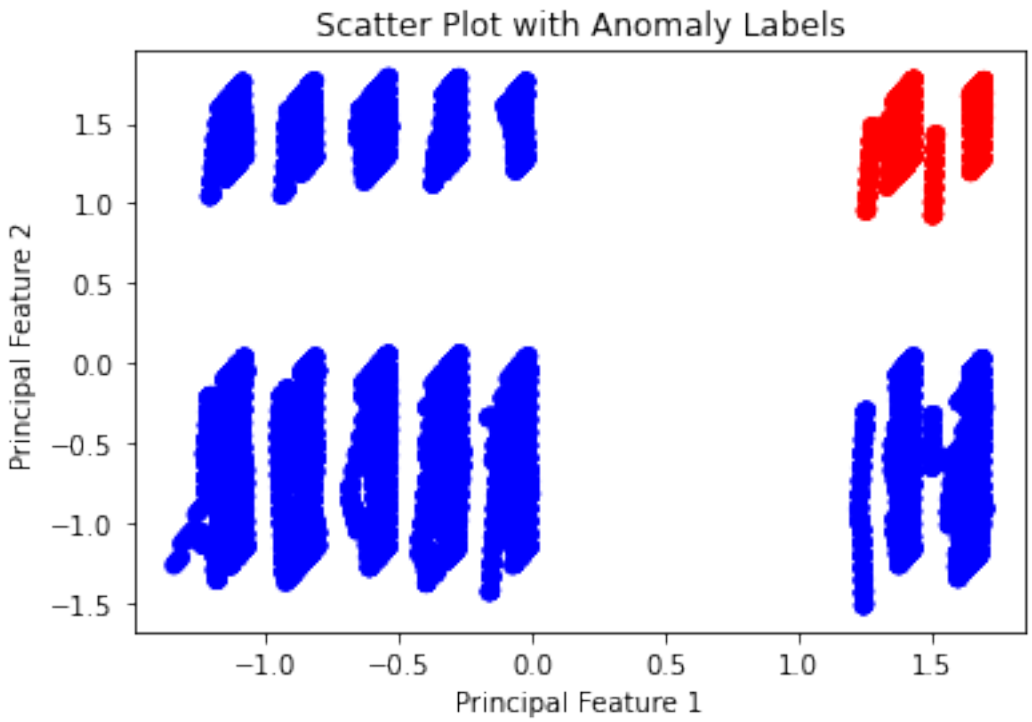



```
def getDistanceByPoint(data, model):
    distance = pd.Series()
    for i in range(0, len(data)):
```

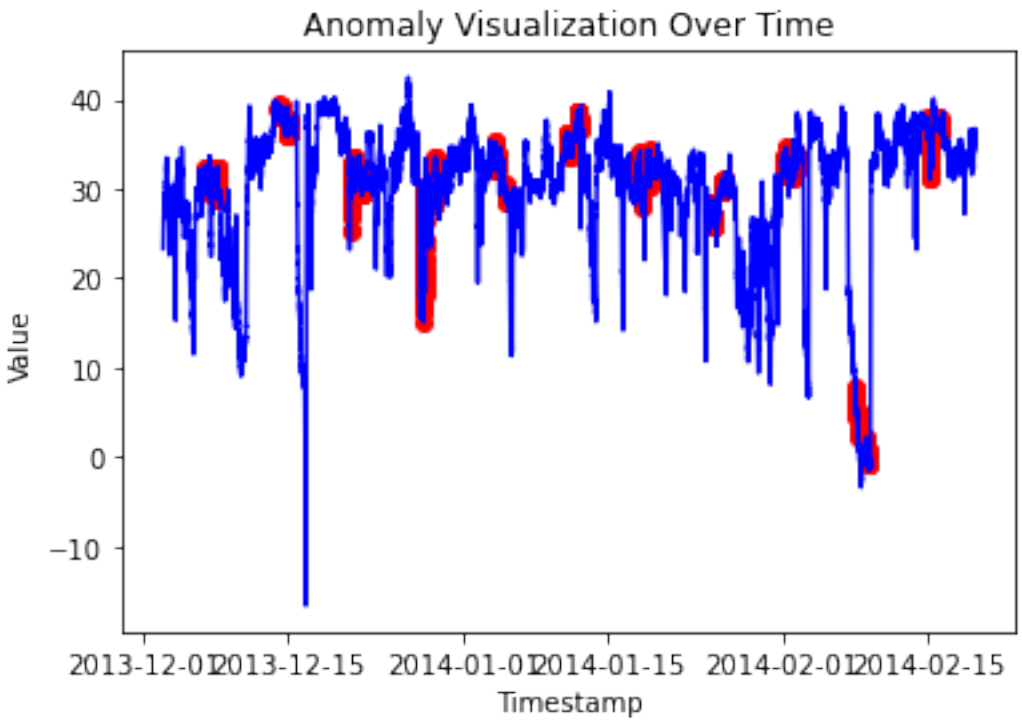
```
        Xa = np.array(data.loc[i])
        cluster_label = model.labels_[i]

        # Check if the cluster label is valid
        if cluster_label >= 0 and cluster_label <
len(model.cluster_centers_):
            Xb = model.cluster_centers_[cluster_label]

            # Perform any necessary data transformation or alignment
here
            # For example, you can reshape Xa and Xb to have the same
shape

            # Calculate the Euclidean distance between Xa and Xb
            dist = np.linalg.norm(Xb)

            # Assign the distance to the Series
            distance.at[i] = dist
        else:
            # Handle the case where the cluster label is invalid
            distance.at[i] = np.nan  # You can choose how to handle
this case

    return distance
# In this modified function, you should perform any necessary data
transformation or alignment before calculating the Euclidean distance.
How you reshape or preprocess the data depends on the specifics of
your dataset and what you are trying to achieve.

# Train Markov model to get the transition matrix
def getTransitionMatrix(df):
    df = np.array(df)
    model = msm.estimate_markov_model(df, 1)
    return model.transition_matrix

# Calculate anomalies using a Markov model and sliding window
def markovAnomaly(df, window_size, threshold):
    transition_matrix = getTransitionMatrix(df)
    real_threshold = threshold ** window_size
    df_anomaly = []
    for j in range(0, len(df)):
        if j < window_size:
            df_anomaly.append(0)
        else:
            sequence = df[j - window_size : j]
            sequence = sequence.reset_index(drop=True)
            df_anomaly.append(anomalyElement(sequence, real_threshold,
transition_matrix))
    return df_anomaly
```

```
# Define or import the anomalyElement function for your specific use
case
def anomalyElement(sequence, threshold, transition_matrix):
    return 0  # Replace with your logic

distance = getDistanceByPoint(data, kmeans_models[14])

# Determine the number of outliers based on a given fraction
outliers_fraction = 0.01  # You can adjust this value
number_of_outliers = int(outliers_fraction * len(distance))

# Calculate the threshold for anomalies
threshold = distance.nlargest(number_of_outliers).min()

# Create a new column 'anomaly21' to indicate anomalies (0:normal,
1:anomaly)
df['anomaly21'] = (distance >= threshold).astype(int)

# Define colors for anomalies (0:normal, 1:anomaly) and NaN values
anomaly_colors = {0: 'blue', 1: 'red'}

# Create a scatter plot to visualize anomalies, color-coded based on
the 'anomaly21' column
plt.scatter(
    df['principal_feature1'], df['principal_feature2'],
    c=df['anomaly21'].map(anomaly_colors).fillna('gray')
)

# Set axis labels and title
plt.xlabel('Principal Feature 1')
plt.ylabel('Principal Feature 2')
plt.title('Scatter Plot with Anomaly Labels')

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

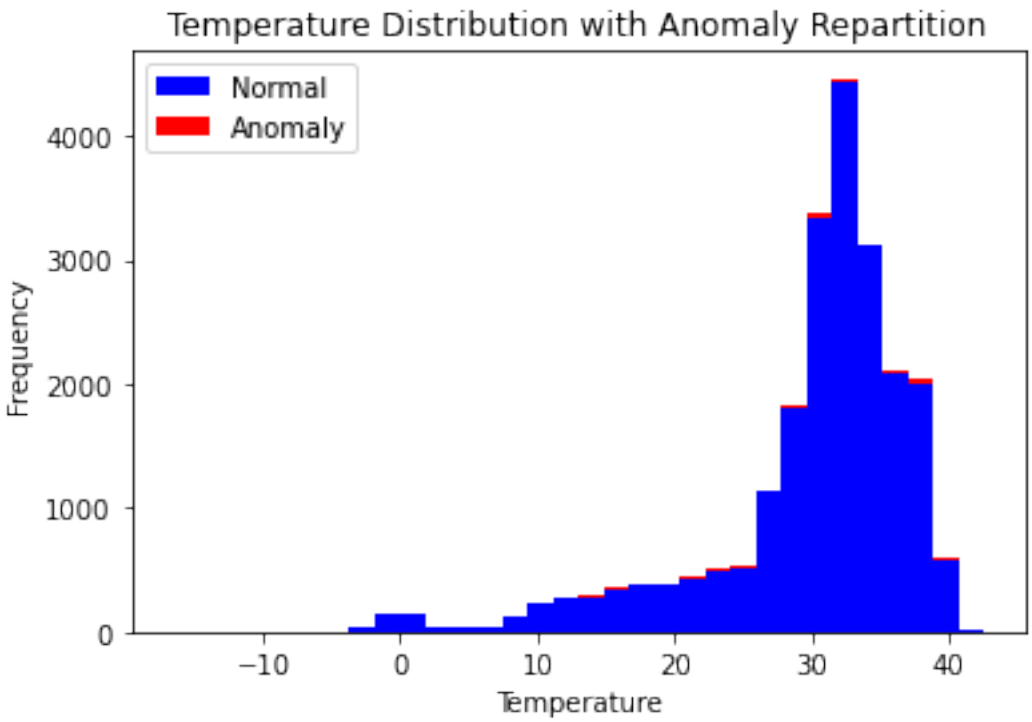
```
/tmp/ipykernel_306/350868234.py:2: FutureWarning: The default dtype
for empty Series will be 'object' instead of 'float64' in a future
version. Specify a dtype explicitly to silence this warning.
  distance = pd.Series()
```



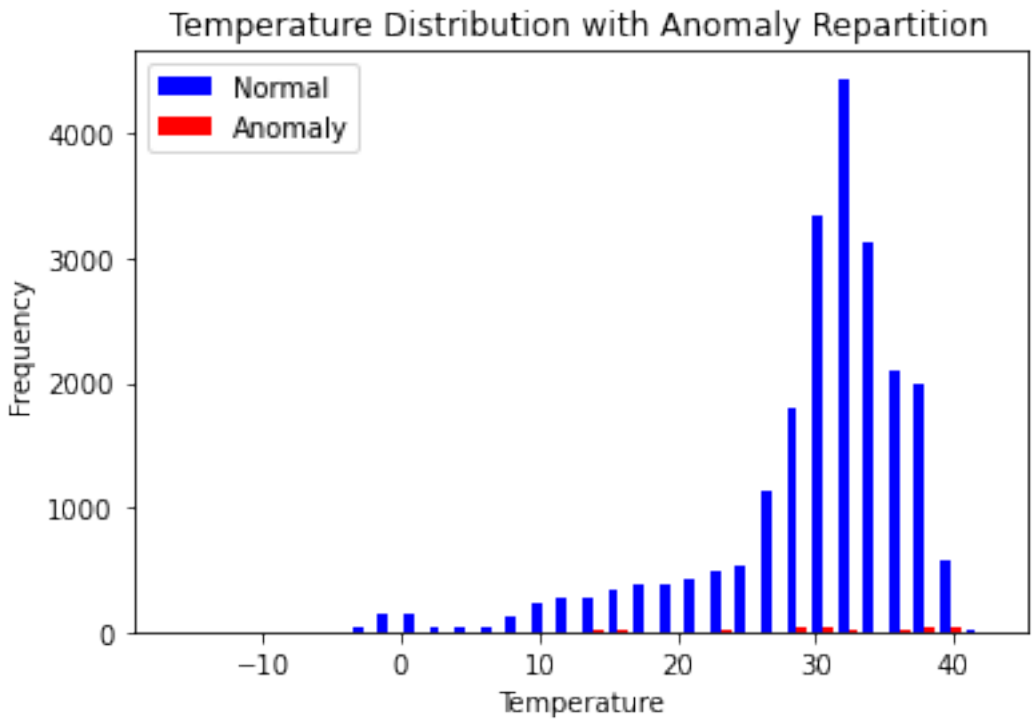

```
# Filter anomalies
anomalies = df[df['anomaly21'] == 1]

# Create a line plot of the entire time series in blue
plt.plot(df['timestamp'], df['value'], color='blue')

# Scatter plot anomalies in red
plt.scatter(anomalies['timestamp'], anomalies['value'], color='red')

# Set axis labels and title
plt.xlabel('Timestamp')
plt.ylabel('Value')
plt.title('Anomaly Visualization Over Time')

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




```
# Filter normal and anomaly values
normal_values = df[df['anomaly21'] == 0]['value']
anomaly_values = df[df['anomaly21'] == 1]['value']

# Create a histogram with stacked bars
plt.hist([normal_values, anomaly_values], bins=32, stacked=True,
color=['blue', 'red'], label=['Normal', 'Anomaly'])

# Add a legend
plt.legend()

# Set axis labels and title
plt.xlabel('Temperature')
plt.ylabel('Frequency')
plt.title('Temperature Distribution with Anomaly Repartition')

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



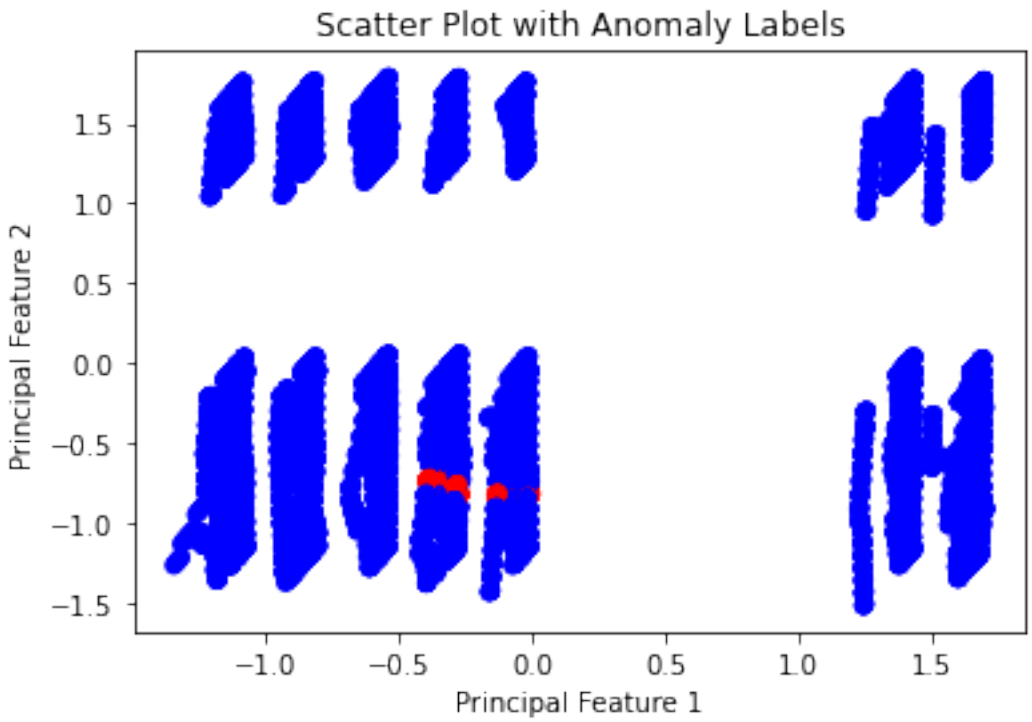

```
# Filter normal and anomaly values
normal_values = df[df['anomaly21'] == 0]['value']
anomaly_values = df[df['anomaly21'] == 1]['value']

# Create a histogram with stacked bars
plt.hist([normal_values, anomaly_values], bins=32, stacked=False,
color=['blue', 'red'], label=['Normal', 'Anomaly'])

# Add a legend
plt.legend()

# Set axis labels and title
plt.xlabel('Temperature')
plt.ylabel('Frequency')
plt.title('Temperature Distribution with Anomaly Repartition')

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



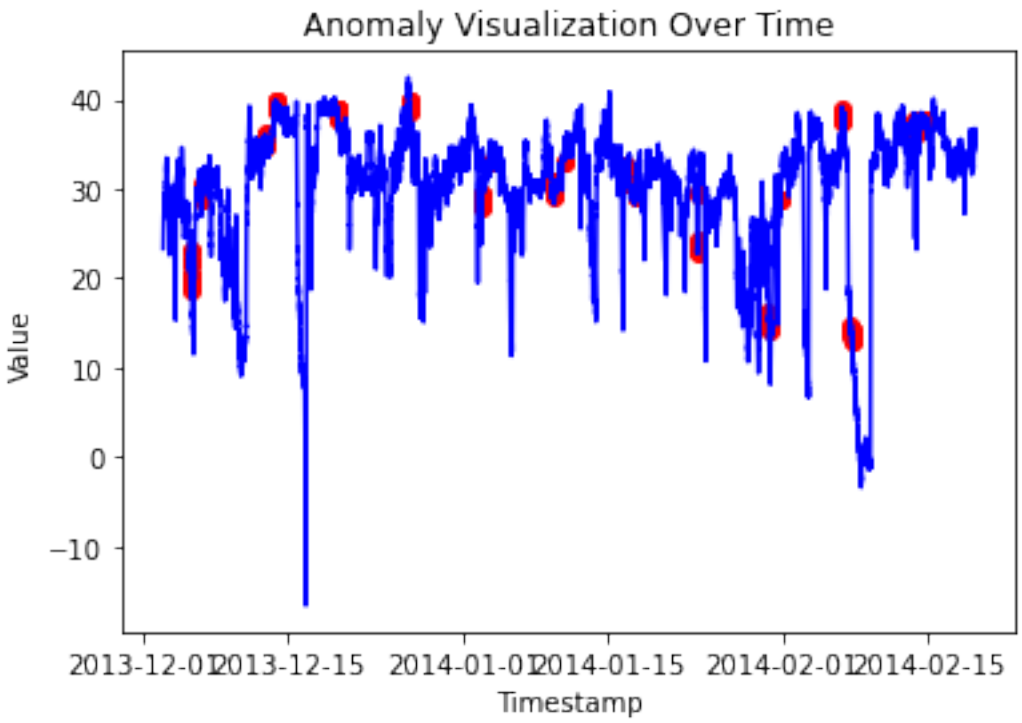

```
# change dist with Xa to find different anomaly pattern
def getDistanceByPoint1(data, model):
    distance = pd.Series()
    for i in range(0, len(data)):
        Xa = np.array(data.loc[i])
        cluster_label = model.labels_[i]

        # Check if the cluster label is valid
        if cluster_label >= 0 and cluster_label <
len(model.cluster_centers_):
            Xb = model.cluster_centers_[cluster_label]

            # Perform any necessary data transformation or alignment
here
            # For example, you can reshape Xa and Xb to have the same
shape

            # Calculate the Euclidean distance between Xa and Xb
            dist = np.linalg.norm(Xa)

            # Assign the distance to the Series
            distance.at[i] = dist
        else:
            # Handle the case where the cluster label is invalid
            distance.at[i] = np.nan  # You can choose how to handle
this case
```

```
    return distance

distance = getDistanceByPoint1(data, kmeans_models[14])

# Determine the number of outliers based on a given fraction
outliers_fraction = 0.01  # You can adjust this value
number_of_outliers = int(outliers_fraction * len(distance))

# Calculate the threshold for anomalies
threshold = distance.nlargest(number_of_outliers).min()

# Create a new column 'anomaly21' to indicate anomalies (0:normal,
1:anomaly)
df['anomaly21'] = (distance >= threshold).astype(int)

# Define colors for anomalies (0:normal, 1:anomaly) and NaN values
anomaly_colors = {0: 'blue', 1: 'red'}

# Create a scatter plot to visualize anomalies, color-coded based on
the 'anomaly21' column
plt.scatter(
    df['principal_feature1'], df['principal_feature2'],
    c=df['anomaly21'].map(anomaly_colors).fillna('gray')
)

# Set axis labels and title
plt.xlabel('Principal Feature 1')
plt.ylabel('Principal Feature 2')
plt.title('Scatter Plot with Anomaly Labels')

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

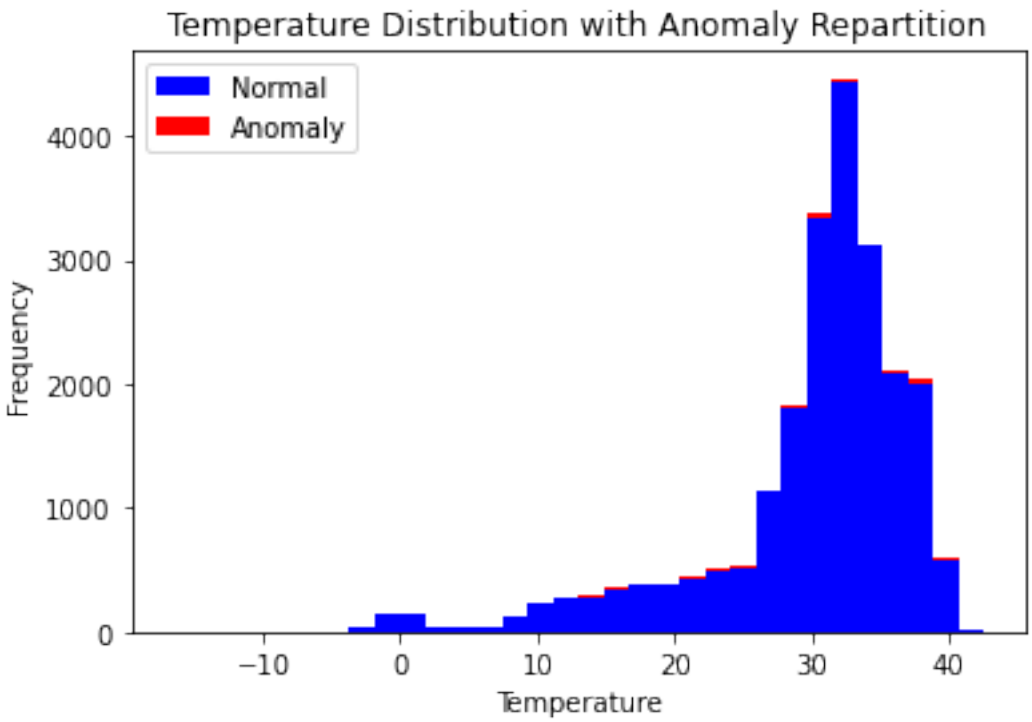
```
/tmp/ipykernel_306/1207856372.py:3: FutureWarning: The default dtype
for empty Series will be 'object' instead of 'float64' in a future
version. Specify a dtype explicitly to silence this warning.
  distance = pd.Series()
```



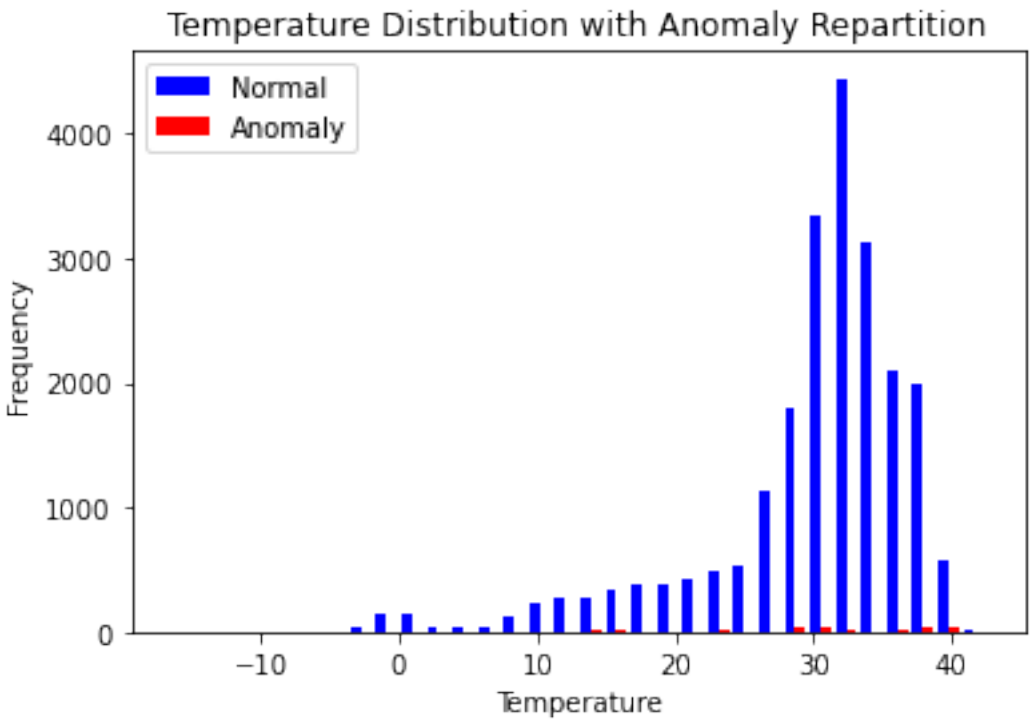

```
# Filter anomalies
anomalies = df[df['anomaly21'] == 1]

# Create a line plot of the entire time series in blue
plt.plot(df['timestamp'], df['value'], color='blue')

# Scatter plot anomalies in red
plt.scatter(anomalies['timestamp'], anomalies['value'], color='red')

# Set axis labels and title
plt.xlabel('Timestamp')
plt.ylabel('Value')
plt.title('Anomaly Visualization Over Time')

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




```
# Filter normal and anomaly values
normal_values = df[df['anomaly21'] == 0]['value']
anomaly_values = df[df['anomaly21'] == 1]['value']

# Create a histogram with stacked bars
plt.hist([normal_values, anomaly_values], bins=32, stacked=True,
color=['blue', 'red'], label=['Normal', 'Anomaly'])

# Add a legend
plt.legend()

# Set axis labels and title
plt.xlabel('Temperature')
plt.ylabel('Frequency')
plt.title('Temperature Distribution with Anomaly Repartition')

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




```
# Filter normal and anomaly values
normal_values = df[df['anomaly21'] == 0]['value']
anomaly_values = df[df['anomaly21'] == 1]['value']

# Create a histogram with stacked bars
plt.hist([normal_values, anomaly_values], bins=32, stacked=False,
color=['blue', 'red'], label=['Normal', 'Anomaly'])

# Add a legend
plt.legend()

# Set axis labels and title
plt.xlabel('Temperature')
plt.ylabel('Frequency')
plt.title('Temperature Distribution with Anomaly Repartition')

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


Temperature Distribution with Anomaly Repartition
Normal
Anomaly
Frequency
4000
3000
2000
1000
0
−10
0
10
20
30
40
Temperature