e7-final

November 14, 2023

[387]:
```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import matplotlib.dates as mdates
import plotly.express as px
from scipy.signal import find_peaks
df=pd.read_csv("e7-htr-currernt.csv")
df.head()
```

[387]:

	Timestamp	HT R Phase Current
0	23-12-2018 05:30	0.0
1	23-12-2018 05:35	0.0
2	23-12-2018 05:40	0.0
3	23-12-2018 05:45	0.0
4	23-12-2018 05:50	0.0

[388]:
```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error, r2_score
```

[389]:
```
df['Timestamp'] = pd.to_datetime(df['Timestamp'], format='%d-%m-%Y %H:%M')

# Creating Date and Time columns
df['Date'] = df['Timestamp'].dt.strftime('%Y-%m-%d')
df['Time'] = df['Timestamp'].dt.strftime('%H:%M')

df.head()
```

[389]:

	Timestamp	HT R Phase Current	Date	Time
0	2018-12-23 05:30:00	0.0	2018-12-23	05:30
1	2018-12-23 05:35:00	0.0	2018-12-23	05:35
2	2018-12-23 05:40:00	0.0	2018-12-23	05:40
3	2018-12-23 05:45:00	0.0	2018-12-23	05:45
4	2018-12-23 05:50:00	0.0	2018-12-23	05:50

IDENTIFYING “MISSING DAYS”

[390]:
```
df.dropna(inplace=True)
df.head()
```

[390]:

	Timestamp	HT R Phase Current	Date	Time
0	2018-12-23 05:30:00	0.0	2018-12-23	05:30
1	2018-12-23 05:35:00	0.0	2018-12-23	05:35
2	2018-12-23 05:40:00	0.0	2018-12-23	05:40
3	2018-12-23 05:45:00	0.0	2018-12-23	05:45
4	2018-12-23 05:50:00	0.0	2018-12-23	05:50

[391]:
```
# Calculate sum of currents for every date
sum_currents_by_date = df.groupby(['Date', 'Time'])['HT R Phase Current'].sum().
 ↪reset_index()
# Create a new DataFrame
df_date = sum_currents_by_date.groupby('Date')['HT R Phase Current'].sum().
 ↪reset_index()
# Display the new DataFrame
df_date.head()
```

[391]:

	Date	HT R Phase Current
0	2018-12-23	4057.94
1	2018-12-24	3078.08
2	2018-12-25	6193.04
3	2018-12-26	5025.99
4	2018-12-27	4417.56

[392]:
```
# Identify missing days (dates with sum of currents less than 0.5)
missing_days = list(df_date[df_date['HT R Phase Current'] < 0.5]['Date'])
print(len(missing_days))
```

```
38
```

[393]:
```
#making a dataframe from missing_days list
df_missing=df[df['Date'].isin(missing_days)]
df_missing.head()
```

[393]:

	Timestamp	HT R Phase Current	Date	Time
3102	2019-01-03 00:00:00	0.0	2019-01-03	00:00
3103	2019-01-03 00:05:00	0.0	2019-01-03	00:05
3104	2019-01-03 00:10:00	0.0	2019-01-03	00:10
3105	2019-01-03 00:15:00	0.0	2019-01-03	00:15
3106	2019-01-03 00:20:00	0.0	2019-01-03	00:20

IDENTIFYING GOOD AND BAD DAYS #good can be in further 2 almost good and very good #bad is bad

[394]:
```
unique_dates = df_date['Date'].unique()
# New dataframe to store date, and stddev
df_data = pd.DataFrame(columns=['Date', 'Standard_Deviation'])
#calculating std dev for each date
for date_to_check in unique_dates:
    df_specific_date = df[df['Timestamp'].dt.strftime('%Y-%m-%d') ==␣
 ↪date_to_check]
    std_deviation = df_specific_date['HT R Phase Current'].std()

    # Appending to df_data
    df_data = pd.concat([df_data, pd.DataFrame({'Date': [date_to_check],␣
 ↪'Standard_Deviation': [std_deviation]})], ignore_index=True)

df_data.head()
```

[394]:

	Date	Standard_Deviation
0	2018-12-23	25.112546
1	2018-12-24	19.627740
2	2018-12-25	27.707181
3	2018-12-26	24.156746
4	2018-12-27	22.805630

[395]:
```
# A new dataframe named df_data is created to store date, stddev, mean
df_data = pd.DataFrame(columns=['Date', 'Standard_Deviation', 'Mean'])

unique_dates = df['Date'].unique()
#calculating std dev,mean for each date
for date_to_check in unique_dates:
    df_specific_date = df[df['Date']== date_to_check]

    # Checking if df is empty; proceeding if it has data
    if not df_specific_date.empty:
        std_deviation = df_specific_date['HT R Phase Current'].std()
        mean_value = df_specific_date['HT R Phase Current'].mean()

        # Appending to df_data
        df_data = pd.concat([df_data, pd.DataFrame({'Date': [date_to_check],␣
 ↪'Standard_Deviation': [std_deviation], 'Mean': [mean_value]})],␣
 ↪ignore_index=True)

#removing data whose stddev and mean is 0
df_data = df_data[(df_data['Standard_Deviation'] != 0) & (df_data['Mean'] != 0)]
df_data.head()
```

[395]:

	Date	Standard_Deviation	Mean
0	2018-12-23	25.112546	18.279009
1	2018-12-24	19.627740	10.687778

```
2  2018-12-25           27.707181  21.503611
3  2018-12-26           24.156746  17.451354
4  2018-12-27           22.805630  15.338750
```

[396]:
```
unique_dates = df_date['Date'].unique()
# creating df_noise_values to store noise
df_noise_values = pd.DataFrame(columns=['Date', 'Noise_Value'])#noise is ntng␣
 ↪but stddev of second derivative
for date_to_check in unique_dates:
    df_specific_date = df[df['Timestamp'].dt.strftime('%Y-%m-%d') ==␣
 ↪date_to_check].copy()
    df_specific_date['Second_Derivative'] = df_specific_date['HT R Phase␣
 ↪Current'].diff().diff()
    # Checking if the second derivative has at least two unique values to avoid␣
 ↪constant values
    if len(df_specific_date['Second_Derivative'].unique()) > 1:
        # Calculating noise
        noise_value = df_specific_date['Second_Derivative'].std()
        # Append to the noise values DataFrame
        df_noise_values = pd.concat([df_noise_values, pd.DataFrame({'Date':␣
 ↪[date_to_check], 'Noise_Value': [noise_value]})], ignore_index=True)
#removing columns whose noise is zero
df_noise_values = df_noise_values[df_noise_values['Noise_Value'] != 0]
df_noise_values.head()
```

[396]:
```
         Date  Noise_Value
0  2018-12-23     8.407470
1  2018-12-24     6.776399
2  2018-12-25     1.181446
3  2018-12-26     8.072541
4  2018-12-27     8.760386
```

[397]:
```
#merging df_noise_values to df_data
df_data = pd.merge(df_data, df_noise_values, on='Date', how='left',␣
 ↪suffixes=('', '_noise'))
df_data = df_data[df_data['Noise_Value'] != 0]
df_data.head()
```

[397]:
```
         Date  Standard_Deviation       Mean  Noise_Value
0  2018-12-23           25.112546  18.279009     8.407470
1  2018-12-24           19.627740  10.687778     6.776399
2  2018-12-25           27.707181  21.503611     1.181446
3  2018-12-26           24.156746  17.451354     8.072541
4  2018-12-27           22.805630  15.338750     8.760386
```

[398]:
```
#using sql ".where" function to classify into really good, good,bad days
df_data['Classification'] = np.where(df_data['Noise_Value'] < 1.3, 'Really Good␣
 ↪Day',
                                     np.where((df_data['Noise_Value'] >= 1.3) &␣
 ↪(df_data['Noise_Value'] < 4.79), 'Good Day', 'Bad Day'))

df_data.head()
#If 'Noise_Value'<1,then it is'Really Good'.
#If  1<='Noise_Value'<4.79  it is a 'Good Day'.
#Else a 'Bad Day'.
```

[398]:

	Date	Standard_Deviation	Mean	Noise_Value	Classification
0	2018-12-23	25.112546	18.279009	8.407470	Bad Day
1	2018-12-24	19.627740	10.687778	6.776399	Bad Day
2	2018-12-25	27.707181	21.503611	1.181446	Really Good Day
3	2018-12-26	24.156746	17.451354	8.072541	Bad Day
4	2018-12-27	22.805630	15.338750	8.760386	Bad Day

[399]:
```
#making a list of really good days
really_good_days_list = list(df_data[df_data['Classification'] == 'Really Good␣
 ↪Day']['Date'])
len(really_good_days_list)
```

[399]: 22

[400]:
```
#making individual lists of good days and bad days
good_days_list = list(df_data[df_data['Classification'] == 'Good Day']['Date'])
bad_days_list = list(df_data[df_data['Classification'] == 'Bad Day']['Date'])
```

[401]:
```
#(Manual Inspection) Looking into some graphs
#can be done automatedly by using rolling stddev but chose to do manually
additional_bad_days = ['2019-01-23', '2019-01-27', '2019-01-28', '2019-02-09',␣
 ↪'2019-03-11', '2019-06-19', '2019-09-26', '2019-09-29', '2019-09-30',]
for date in additional_bad_days:
    if date in good_days_list:
        good_days_list.remove(date)
        bad_days_list.append(date)
print("Updated Good Days List:")
print(len(good_days_list))
```

```
Updated Good Days List:
40
```

[402]:
```
additional_good_days = ['2019-02-10', '2019-04-29']
for date in additional_good_days:
    if date in bad_days_list:
        bad_days_list.remove(date)
```

```
        good_days_list.append(date)
print("Updated Good Days List Length:", len(good_days_list))
```

```
Updated Good Days List Length: 42
```

[403]:
```
additional_bad_days = ['2019-09-26','2019-01-23']
for date in additional_bad_days:
    if date in really_good_days_list:
        really_good_days_list.remove(date)
        bad_days_list.append(date)
print("Updated Good Days List:")
print(len(really_good_days_list))
```

```
Updated Good Days List:
20
```

#REALLY GOOD DAYS, GOOD DAYS , BAD DAYS ARE IDENTIFIED I NOW NEED TO BETTER THEM

[404]:
```
#from df variate into good and bad days
df_really_good=df[df['Date'].isin(really_good_days_list)]
df_really_good.head()
```

[404]:

	Timestamp	HT R Phase Current	Date	Time
510	2018-12-25 00:00:00	0.1	2018-12-25	00:00
511	2018-12-25 00:05:00	0.1	2018-12-25	00:05
512	2018-12-25 00:10:00	0.1	2018-12-25	00:10
513	2018-12-25 00:15:00	0.1	2018-12-25	00:15
514	2018-12-25 00:20:00	0.1	2018-12-25	00:20

[405]:
```
#from df variate into good and bad days
df_good=df[df['Date'].isin(good_days_list)]
df_bad=df[df['Date'].isin(bad_days_list)]
```

[406]:
```
# A function for rolling std dev
def rolling_std_dev(series, window_size):
    return series.rolling(window=window_size, min_periods=1).std()
#rolling std dev for a window size of 5
window_size = 5
df_good['Rolling_Std_Dev'] = rolling_std_dev(df_good['HT R Phase Current'],␣
 ↪window_size)
#printing graphs for first 5 days
for date in df_good['Date'].unique()[:5]:
    # Filter data for the current date
    date_data = df_good[df_good['Date'] == date]

   #graph using plotly
```

```
    fig = px.line(date_data, x=date_data.index, y=['HT R Phase Current',␣
 ↪'Rolling_Std_Dev'],
                  labels={'value': 'HT R Phase Current'},
                  title=f'Original Data and Rolling Std Dev for Date {date}',
                  line_shape='linear', render_mode='svg')
    fig.show()
```

```
/var/folders/_g/fk82m8554cb11z2y23b6pz3m0000gn/T/ipykernel_1509/130960304.py:6:
SettingWithCopyWarning:


A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
```

[407]:
```
#writing same algorithm for bad days
def rolling_std_dev(series, window_size):
    return series.rolling(window=window_size, min_periods=1).std()
window_size = 5
df_bad['Rolling_Std_Dev'] = rolling_std_dev(df_bad['HT R Phase Current'],␣
 ↪window_size)
for date in df_bad['Date'].unique()[:5]:
    # Filter data for the current date
    date_data = df_bad[df_bad['Date'] == date]
    fig = px.line(date_data, x=date_data.index, y=['HT R Phase Current',␣
 ↪'Rolling_Std_Dev'],
                  labels={'value': 'HT R Phase Current'},
                  title=f'Original Data and Rolling Std Dev for Date {date}',
                  line_shape='linear', render_mode='svg')
    fig.show()
```

```
/var/folders/_g/fk82m8554cb11z2y23b6pz3m0000gn/T/ipykernel_1509/1991420341.py:5:
SettingWithCopyWarning:


A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
```

[408]:
```
#(Manual Inspection) can be obtained with a condition that it can be less than␣
 ↪a threshold but for more accuracy and precision chose to do manually
```

```
additional_good_days = ['2018-12-26', '2018-12-27', '2019-01-01', '2019-01-06',␣
 ↪'2019-02-02', '2019-02-03', '2019-02-14', '2019-02-15',␣
 ↪'2019-04-08','2019-04-26','2019-05-06','2019-05-11']
for date in additional_good_days:
    if date in bad_days_list:
        bad_days_list.remove(date)
        good_days_list.append(date)
print("Updated Good Days List Length:", len(good_days_list))
```

```
Updated Good Days List Length: 54
```

#HANDLING REALLY GOOD DAYS AND GOOD DAYS

BY ROLLING WINDOW(MEAN)

[409]:
```
df_good=df_good.copy()
#creating  a new column for improved current
def smooth_data(series, window_size=25):
    return series.rolling(window=window_size, center=True).mean()

df_good['Improved_Current'] = smooth_data(df_good['HT R Phase Current'])
```

[410]:
```
#removing na's
df_good = df_good.dropna(subset=['Improved_Current'])
df_good.head()
```

[410]:

	Timestamp	HT R Phase Current	Date	Time	Rolling_Std_Dev	Improved_Current
13482	2019-02-10 01:00:00	0.09	2019-02-10	01:00	0.0	0.0612
13483	2019-02-10 01:05:00	0.09	2019-02-10	01:05	0.0	0.0648
13484	2019-02-10 01:10:00	0.09	2019-02-10	01:10	0.0	0.0684
13485	2019-02-10 01:15:00	0.09	2019-02-10	01:15	0.0	0.0720
13486	2019-02-10 01:20:00	0.09	2019-02-10	01:20	0.0	0.0756

[411]:
```
#from dt.time object convertinng to string and creating hour,minute
df_good['Time'] = df_good['Time'].astype(str)
df_good[['Hour', 'Minute']] = df_good['Time'].str.split(':', expand=True)
df_good.head()
```

[411]:

	Timestamp	HT R Phase Current	Date	Time
13482	2019-02-10 01:00:00	0.09	2019-02-10	01:00
13483	2019-02-10 01:05:00	0.09	2019-02-10	01:05

```
13484 2019-02-10 01:10:00                0.09  2019-02-10  01:10
13485 2019-02-10 01:15:00                0.09  2019-02-10  01:15
13486 2019-02-10 01:20:00                0.09  2019-02-10  01:20

       Rolling_Std_Dev  Improved_Current Hour Minute
13482              0.0            0.0612   01     00
13483              0.0            0.0648   01     05
13484              0.0            0.0684   01     10
13485              0.0            0.0720   01     15
13486              0.0            0.0756   01     20
```

[412]:
```
#plotting graphs for good days between HT R Phase current and Imporved Current
unique_dates = df_good['Date'].unique()[:5]

for date in unique_dates:
    df_other_good = df_good[df_good['Date'] == date]
    fig = px.line(df_other_good, x='Time', y=['HT R Phase Current','Improved_Current'],
                  labels={'value': 'Current'},
                  title=f'Current Comparison for {date}',
                  line_shape='linear', render_mode='svg')
    fig.update_layout(xaxis_title='Timestamp', yaxis_title='Current',legend_title='Current Type')

    # Change x-axis ticks to display every 1 hour
    fig.update_xaxes(tickmode='array', tickvals=df_other_good['Time'][::12],ticktext=df_other_good['Time'][::12], tickangle=45)
    fig.show()
```

[413]:
```
#replacing zero values (iff) replacing them with predicted current(:FIXING VALUES)
df_good = df_good.copy()#creating a copy for ceavet's issue
df_good['Time'] = pd.to_datetime(df_good['Time'])
df_good.set_index('Time', inplace=True)
zero_mask = (df_good['HT R Phase Current'] == 0)
df_good.loc[zero_mask, 'HT R Phase Current'] = df_good.loc[zero_mask,'Improved_Current']
df_good.reset_index(drop=True, inplace=True)  # Use drop=True to avoid adding an additional index column
df_good.head()
```

[413]:
```
             Timestamp  HT R Phase Current        Date  Rolling_Std_Dev  \
0 2019-02-10 01:00:00                0.09  2019-02-10              0.0
1 2019-02-10 01:05:00                0.09  2019-02-10              0.0
2 2019-02-10 01:10:00                0.09  2019-02-10              0.0
3 2019-02-10 01:15:00                0.09  2019-02-10              0.0
```

```
4 2019-02-10 01:20:00                0.09  2019-02-10              0.0

   Improved_Current Hour Minute
0            0.0612   01     00
1            0.0648   01     05
2            0.0684   01     10
3            0.0720   01     15
4            0.0756   01     20
```

#GOOD DAYS(ALMOST) ARE HANDLED NOW HANDLING BAD DAYS

[414]:
```
df_bad=df_bad.copy()
df_bad['Time'] = pd.to_datetime(df_bad['Time'])

# Making new columns for hour, minutes as time is a string in dt.time object
df_bad['Hour'] = df_bad['Time'].dt.hour
df_bad['Minute'] = df_bad['Time'].dt.minute
df_bad.head()
```

[414]:
```
             Timestamp  HT R Phase Current        Date                 Time  \
0 2018-12-23 05:30:00                 0.0  2018-12-23 2023-11-14 05:30:00
1 2018-12-23 05:35:00                 0.0  2018-12-23 2023-11-14 05:35:00
2 2018-12-23 05:40:00                 0.0  2018-12-23 2023-11-14 05:40:00
3 2018-12-23 05:45:00                 0.0  2018-12-23 2023-11-14 05:45:00
4 2018-12-23 05:50:00                 0.0  2018-12-23 2023-11-14 05:50:00

   Rolling_Std_Dev  Hour  Minute
0              NaN     5      30
1              0.0     5      35
2              0.0     5      40
3              0.0     5      45
4              0.0     5      50
```

[415]:
```
df_bad = df_bad.copy()
#applying the same logic for bad days also
def smooth_data(series, window_size=25):
    return series.rolling(window=window_size, center=True, min_periods=1).mean()
df_bad['Improved_Current'] = smooth_data(df_bad['HT R Phase Current'])
#drop nan values(or) replace them with HT R Phase current
nan_mask = df_bad['HT R Phase Current'].isna()
df_bad.loc[nan_mask, 'Improved_Current'] = df_bad.loc[nan_mask, 'HT R Phase␣
 ↪Current']

df_bad.head()
```

[415]:
```
             Timestamp  HT R Phase Current        Date                 Time  \
0 2018-12-23 05:30:00                 0.0  2018-12-23 2023-11-14 05:30:00
```

```
1 2018-12-23 05:35:00                0.0  2018-12-23 2023-11-14 05:35:00
2 2018-12-23 05:40:00                0.0  2018-12-23 2023-11-14 05:40:00
3 2018-12-23 05:45:00                0.0  2018-12-23 2023-11-14 05:45:00
4 2018-12-23 05:50:00                0.0  2018-12-23 2023-11-14 05:50:00

   Rolling_Std_Dev  Hour  Minute  Improved_Current
0              NaN     5      30          0.000000
1              0.0     5      35          0.000000
2              0.0     5      40          0.048667
3              0.0     5      45          0.045625
4              0.0     5      50          0.042941
```

[417]:
```
#plotting predicted vs original current graphs as did for df_good
unique_dates_bad = df_bad['Date'].unique()
unique_dates_bad = unique_dates_bad[:5]
for date in unique_dates_bad:
    df_other_bad = df_bad[df_bad['Date'] == date]
    fig = px.line(df_other_bad, x='Time', y=['HT R Phase Current',␣
 ↪'Improved_Current'],
                  labels={'value': 'Current'},
                  title=f'Current Comparison for {date}',
                  line_shape='linear', render_mode='svg')
    fig.update_layout(xaxis_title='Timestamp', yaxis_title='Current',␣
 ↪legend_title='Current Type')
    fig.update_xaxes(tickmode='array', tickvals=df_other_bad['Time'][::12],␣
 ↪ticktext=df_other_bad['Time'][::12], tickangle=45)
    fig.show()
```

[418]:
```
#Replacing the missing values
df_bad['Time'] = pd.to_datetime(df_bad['Time'])
df_bad.set_index('Time', inplace=True)
# Create a mask for the time range 7:00 to 18:00
time_mask = (df_bad.index.hour >= 7) & (df_bad.index.hour <= 18)
zero_mask = (df_bad['HT R Phase Current'] == 0) & time_mask
df_bad.loc[zero_mask, 'HT R Phase Current'] = df_bad.loc[zero_mask,␣
 ↪'Improved_Current']
df_bad.reset_index(inplace=True,drop=True)
df_bad.head()
```

[418]:
```
             Timestamp  HT R Phase Current        Date  Rolling_Std_Dev  Hour  \
0 2018-12-23 05:30:00                 0.0  2018-12-23              NaN     5
1 2018-12-23 05:35:00                 0.0  2018-12-23              0.0     5
2 2018-12-23 05:40:00                 0.0  2018-12-23              0.0     5
3 2018-12-23 05:45:00                 0.0  2018-12-23              0.0     5
4 2018-12-23 05:50:00                 0.0  2018-12-23              0.0     5

   Minute  Improved_Current
```

```
0       30          0.000000
1       35          0.000000
2       40          0.048667
3       45          0.045625
4       50          0.042941
```

NOW ALL THE DAYS ARE MODIFIED CONFIGURING MODEL(RANDOM FOREST REGRESSION BEGINS)

[419]:
```
#if any column exists as a string then converting into datetime object
df_good['Timestamp'] = pd.to_datetime(df_good['Timestamp'])
df_good['Hour'] = df_good['Timestamp'].dt.hour
df_good['Minute'] = df_good['Timestamp'].dt.minute

X = df_good[['Hour', 'Minute']]
y = df_good['HT R Phase Current']
#splting in 0.8 train size
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=40)

# Creating and training a RandomForestRegressor
model = RandomForestRegressor(n_estimators=100, random_state=42)
model.fit(X_train, y_train)


y_pred = model.predict(X_test)

# Evaluate the model
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

print(f'Mean Squared Error: {mse}')
print(f'R-squared (R2) Score: {r2}')
```

```
Mean Squared Error: 179.6119594501162
R-squared (R2) Score: 0.7755025296931534
```

[420]:
```
from sklearn.linear_model import LinearRegression
from sklearn.svm import SVR
from sklearn.ensemble import GradientBoostingRegressor
```

[421]:
```
# Linear Regression
linear_model = LinearRegression()
linear_model.fit(X_train, y_train)
linear_predictions = linear_model.predict(X_test)
linear_r2 = r2_score(y_test, linear_predictions)
linear_mse = mean_squared_error(y_test, linear_predictions)
```

```
print(f'Linear Regression:')
print(f'Mean Squared Error: {linear_mse}')
print(f'R-squared (R2) Score: {linear_r2}')
print()
```

```
Linear Regression:
Mean Squared Error: 800.4826541366461
R-squared (R2) Score: -0.0005254183984197969
```

[422]:
```
# Support Vector Regression (SVR)
svr_model = SVR()
svr_model.fit(X_train, y_train)
svr_predictions = svr_model.predict(X_test)
svr_mse = mean_squared_error(y_test, svr_predictions)
svr_r2 = r2_score(y_test, svr_predictions)
print(f'Support Vector Regression (SVR):')
print(f'Mean Squared Error: {svr_mse}')
print(f'R-squared (R2) Score: {svr_r2}')
```

```
Support Vector Regression (SVR):
Mean Squared Error: 541.2730838820947
R-squared (R2) Score: 0.3234613193423559
```

[423]:
```
# Gradient Boosting Regressor
gb_model = GradientBoostingRegressor(n_estimators=100, random_state=42)
gb_model.fit(X_train, y_train)
gb_predictions = gb_model.predict(X_test)
gb_mse = mean_squared_error(y_test, gb_predictions)
gb_r2 = r2_score(y_test, gb_predictions)
print(f'Gradient Boosting Regressor:')
print(f'Mean Squared Error: {gb_mse}')
print(f'R-squared (R2) Score: {gb_r2}')
```

```
Gradient Boosting Regressor:
Mean Squared Error: 173.23738485407728
R-squared (R2) Score: 0.7834701275940631
```

#RANDOM FOREST REGRESSOR IS USED AND FINALIZED BASED ON MSE AND R2 value

USING THE MODEL TO FIX FOR BAD DAYS

[424]:
```
#using model to fix for bad days
df_bad['Hour'] = df_bad['Timestamp'].dt.hour
df_bad['Minute'] = df_bad['Timestamp'].dt.minute
features_bad = df_bad[['Hour', 'Minute']]

#adding it as a new column
```

```
df_bad['Predicted_Current'] = model.predict(features_bad)
df_bad.head()
```

[424]:

	Timestamp	HT R Phase Current	Date	Rolling_Std_Dev	Hour	\
0	2018-12-23 05:30:00	0.0	2018-12-23	NaN	5	
1	2018-12-23 05:35:00	0.0	2018-12-23	0.0	5	
2	2018-12-23 05:40:00	0.0	2018-12-23	0.0	5	
3	2018-12-23 05:45:00	0.0	2018-12-23	0.0	5	
4	2018-12-23 05:50:00	0.0	2018-12-23	0.0	5	

	Minute	Improved_Current	Predicted_Current
0	30	0.000000	0.106018
1	35	0.000000	0.141261
2	40	0.048667	0.250353
3	45	0.045625	0.380660
4	50	0.042941	0.501280

#HIGHLIGHTING CHANGES

[425]:
```
import plotly.express as px

# Creating a line plot for Original vs Improved Data
fig_original_vs_improved = px.line(df_bad, x='Timestamp', y=['HT R Phase␣
 ↪Current', 'Improved_Current'],
                                   labels={'value': 'HT R Phase Current'},
                                   title='Original vs Improved Data')

# Creating a line plot for Original vs Predicted Data
fig_original_vs_predicted = px.line(df_bad, x='Timestamp', y=['HT R Phase␣
 ↪Current', 'Predicted_Current'],
                                    labels={'value': 'HT R Phase Current'},
                                    title='Original vs Predicted Data')

fig_original_vs_improved.show()
fig_original_vs_predicted.show()
```

[426]:
```
import plotly.express as px
# Creating a line plot for Good Data
fig_good = px.line(df_good, x='Timestamp', y=['HT R Phase Current',␣
 ↪'Improved_Current'],
                   labels={'value': 'HT R Phase Current'},
                   title='Good Data')

fig_good.show()
```

[ ]:
```

```