In [571…
```
import os
import requests
from dotenv import load_dotenv
import datetime as d
import re
import pandas as pd
import json
import numpy as np
from sklearn.preprocessing import FunctionTransformer
from sklearn.multioutput import MultiOutputRegressor
from sklearn.svm import LinearSVR
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedKFold
from sklearn.metrics import mean_absolute_error
import plotly.express as px
import plotly.io as pio
import psycopg2
from io import StringIO
```

In [366…
```
#Load API key and host
load_dotenv(dotenv_path="api.env")

api_key = os.getenv("API_KEY")
```

In [367…
```
#Gets start date and end date for API nad the calendar
def get_dates()->list:
    dates = []
    while True:
        try:
            start_date = input("Input start date (YYYY-MM-DD): \n")
            start_date = d.datetime.strptime(start_date, "%Y-%m-%d")
            start_date.strftime("%Y-%m-%d")
            end_date = input("Input end date (YYYY-MM-DD): \n")
            end_date = d.datetime.strptime(end_date, "%Y-%m-%d")
            end_date.strftime("%Y-%m-%d")
            if (start_date > end_date):
                print("Start date cannot be after the end date. Please try again
                continue
            break #If both dates are valid and in order, exit the loop
        except ValueError:
            print("Sorry, that is in the incorrect format. Please try again.")

    dates.append(start_date)
    dates.append(end_date)

    return dates
```

In [368…
```
def access_api()->list:

    dates = get_dates()

    start_date = dates[0]
    end_date = dates[1]

    start_date = start_date.date()
    end_date = end_date.date()
    solar_flare = f"https://api.nasa.gov/DONKI/FLR?startDate={start_date}&endDat
    print(solar_flare)

    solar_energetic_particle = f"https://api.nasa.gov/DONKI/SEP?startDate={start
    print(solar_energetic_particle)

    try:
        response_flare = requests.get(solar_flare)
        response_flare.raise_for_status() #Check if succesfull request
        data_flare = response_flare.json()  #Convert response to Python dictiona

        if not data_flare:
            print("Something went wrong. Recieved data for Solar flare is empty.
            return None, None

        response_sep = requests.get(solar_energetic_particle)
        response_sep.raise_for_status()  #Check if request was successfull
        data_sep = response_sep.json()  #Convert response to Python dictionary

        if not data_sep:
            print("Something went wrong. Recieved data for Solar energetic parti
            return None, None

    except requests.exceptions.RequestException as e:
        print(f"Error occured: {e}")
        return None, None

    return data_flare, data_sep
```

Using start date - 2016-01-01 : end date - 2022-12-31

In [369…

```
data_flare, data_sep = access_api()
```

```
https://api.nasa.gov/DONKI/FLR?startDate=2016-01-01&endDate=2022-12-31&api_key=Sz
n1a8RfSl3QC6zkr1GEVpqMGqKxpbexVPSEadt3
https://api.nasa.gov/DONKI/SEP?startDate=2016-01-01&endDate=2022-12-31&api_key=Sz
n1a8RfSl3QC6zkr1GEVpqMGqKxpbexVPSEadt3
```

In [370…

```
print(type(data_flare))
print(type(data_sep))
```

```
<class 'list'>
<class 'list'>
```

In [371…

```
data_sep_df = pd.DataFrame(data_sep)
data_sep_df.head()
```

Out[371…

| | sepID | eventTime | instruments | submissionTime | versionId | |
|---|---|---|---|---|---|---|
| **0** | 2016-01-02T02:48:00-SEP-001 | 2016-01-02T02:48Z | [{'displayName': 'SOHO: COSTEP 15.8-39.8 MeV'}] | 2016-01-02T04:45Z | 1 | https://webtools. |
| **1** | 2016-01-02T04:30:00-SEP-001 | 2016-01-02T04:30Z | [{'displayName': 'GOES13: SEM/EPS >10 MeV'}] | 2016-01-02T04:41Z | 1 | https://webtools. |
| **2** | 2017-04-18T23:39:00-SEP-001 | 2017-04-18T23:39Z | [{'displayName': 'STEREO A: IMPACT 13-100 MeV'}] | 2017-04-19T12:01Z | 2 | https://webtools. |
| **3** | 2017-07-14T09:00:00-SEP-001 | 2017-07-14T09:00Z | [{'displayName': 'GOES13: SEM/EPS >10 MeV'}] | 2017-07-14T09:13Z | 1 | https://webtools. |
| **4** | 2017-07-23T10:19:00-SEP-001 | 2017-07-23T10:19Z | [{'displayName': 'STEREO A: IMPACT 13-100 MeV'}] | 2017-07-23T10:46Z | 1 | https://webtools. |

In [372…

```
data_flare_df = pd.DataFrame(data_flare)
data_flare_df.head()
```

Out[372…

| | flrID | catalog | instruments | beginTime | peakTime | endTime | classT |
|---|---|---|---|---|---|---|---|
| **0** | 2016-01-01T23:00:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2016-01-01T23:00Z | 2015-01-02T00:10Z | None | N |
| **1** | 2016-01-28T11:48:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2016-01-28T11:48Z | 2016-01-28T12:02Z | 2016-01-28T12:56Z | |
| **2** | 2016-02-04T18:15:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2016-02-04T18:15Z | 2016-02-04T18:22Z | 2016-02-04T18:28Z | |
| **3** | 2016-02-11T20:18:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2016-02-11T20:18Z | 2016-02-11T21:03Z | 2016-02-11T22:27Z | |
| **4** | 2016-02-12T10:37:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2016-02-12T10:37Z | 2016-02-12T10:47Z | 2016-02-12T10:53Z | N |

In [ ]:

With nested values in json such as display name and activityID we need to get those values out and give them their own column. But there is an issue that some records have multiple values for for linked events with activity ID. There are few options with creating multiple columns, which would be structured, but would become unreadable if there were more nested values. Leaving them as they are which is compact bbut would need additional parsing when analyzing. Then flattening the data in multiple rows which would be good for analyzing, but then again in how much could there nested. It could potentialy increase row count significantly. Then a hybrid approach which is balanced but comples to do. For this project I have decide to go ahed with creating another dataframe for these linked events. Because:

- These valuse could vary in their lengths
- There is hierchy (parent-children)
- Improves efficiency
- Easier aggregations later for analysis

In [374…

```python
def get_instrumens_and_activity(df: pd.DataFrame, data: dict)-> pd.DataFrame:

    #Get nested values from json
    nested_displayName = pd.json_normalize(data, record_path="instruments")
    #Converting to list for manipulation
    displayName_list = nested_displayName["displayName"].to_list()

    #Add displaName to dataframe
    df["instrument_displayName"] = displayName_list

    #Add bolean if there is/is not linked event
    df.loc[pd.isna(df["linkedEvents"]), "activityID"] = False
    df.loc[~pd.isna(df["linkedEvents"]), "activityID"] = True

    return df
```

In [375…

```python
data_flare_df = get_instrumens_and_activity(data_flare_df, data_flare)
data_sep_df = get_instrumens_and_activity(data_sep_df, data_sep)
```

In [376…

```python
data_flare_df.head()
```

Out[376…

| | flrID | catalog | instruments | beginTime | peakTime | endTime | classT |
|---|---|---|---|---|---|---|---|
| **0** | 2016-01-01T23:00:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2016-01-01T23:00Z | 2015-01-02T00:10Z | None | N |
| **1** | 2016-01-28T11:48:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2016-01-28T11:48Z | 2016-01-28T12:02Z | 2016-01-28T12:56Z | |
| **2** | 2016-02-04T18:15:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2016-02-04T18:15Z | 2016-02-04T18:22Z | 2016-02-04T18:28Z | |
| **3** | 2016-02-11T20:18:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2016-02-11T20:18Z | 2016-02-11T21:03Z | 2016-02-11T22:27Z | |
| **4** | 2016-02-12T10:37:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2016-02-12T10:37Z | 2016-02-12T10:47Z | 2016-02-12T10:53Z | N |

In [377…

```python
data_flare_df["endTime"].head()
```

Out[377…

```
0                   None
1      2016-01-28T12:56Z
2      2016-02-04T18:28Z
3      2016-02-11T22:27Z
4      2016-02-12T10:53Z
Name: endTime, dtype: object
```

Below is a function that creates new dataframe from the linkedevents with ID so we can join it later to the original dataframe. In this function I use inplace in functions where it can be used so that memory is not impacted by creating copy of dataframe. The only issue with inplace is that original df is altered, but it is not problem here since original is kept safe. Secon for performance is filtering before all the functions so it frees up some perfomance if the data is large.

In [378…

```
def create_activity_df(df: pd.DataFrame)-> pd.DataFrame:

    #Based on which dataframe select correct columns
    if "flrID" in df.columns:
        columns = ["flrID", "linkedEvents"]
    else:
        columns = ["sepID", "linkedEvents"]

    df = df[columns]

    df = df.dropna(subset="linkedEvents") #Drop empty rows
    df = df.explode("linkedEvents") #Turn into rows
    df.reset_index(drop=True, inplace=True) #Reset index
    actitivityID_df = pd.json_normalize(df["linkedEvents"]) #Normalize dataframe
    df["activityID"] = actitivityID_df["activityID"] #Add new column
    df.drop(columns="linkedEvents", axis=1, inplace=True) #Drop uneccesary colum

    return df
```

In [379…

```
data_flare_df.head()
```

Out[379…

| | flrID | catalog | instruments | beginTime | peakTime | endTime | classT |
|---|---|---|---|---|---|---|---|
| **0** | 2016-01-01T23:00:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2016-01-01T23:00Z | 2015-01-02T00:10Z | None | N |
| **1** | 2016-01-28T11:48:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2016-01-28T11:48Z | 2016-01-28T12:02Z | 2016-01-28T12:56Z | |
| **2** | 2016-02-04T18:15:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2016-02-04T18:15Z | 2016-02-04T18:22Z | 2016-02-04T18:28Z | |
| **3** | 2016-02-11T20:18:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2016-02-11T20:18Z | 2016-02-11T21:03Z | 2016-02-11T22:27Z | |
| **4** | 2016-02-12T10:37:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2016-02-12T10:37Z | 2016-02-12T10:47Z | 2016-02-12T10:53Z | N |

In [380…

```
flare_linked_events = create_activity_df(data_flare_df)
flare_linked_events.head()
```

Out[380…

| | flrID | activityID |
|---|---|---|
| **0** | 2016-01-01T23:00:00-FLR-001 | 2016-01-01T23:12:00-CME-001 |
| **1** | 2016-01-01T23:00:00-FLR-001 | 2016-01-02T02:48:00-SEP-001 |
| **2** | 2016-01-01T23:00:00-FLR-001 | 2016-01-02T04:30:00-SEP-001 |
| **3** | 2016-01-28T11:48:00-FLR-001 | 2016-01-28T12:24:00-CME-001 |
| **4** | 2016-02-11T20:18:00-FLR-001 | 2016-02-11T21:28:00-CME-001 |

In [381…

```
solar_linked_events = create_activity_df(data_sep_df)
solar_linked_events.head()
```

Out[381…

| | sepID | activityID |
|---|---|---|
| **0** | 2016-01-02T02:48:00-SEP-001 | 2016-01-01T23:00:00-FLR-001 |
| **1** | 2016-01-02T02:48:00-SEP-001 | 2016-01-01T23:12:00-CME-001 |
| **2** | 2016-01-02T04:30:00-SEP-001 | 2016-01-01T23:00:00-FLR-001 |
| **3** | 2016-01-02T04:30:00-SEP-001 | 2016-01-01T23:12:00-CME-001 |
| **4** | 2017-04-18T23:39:00-SEP-001 | 2017-04-18T19:15:00-FLR-001 |

In [382…

```
data_flare_df.head()
```

Out[382…

| | flrID | catalog | instruments | beginTime | peakTime | endTime | classT |
|---|---|---|---|---|---|---|---|
| **0** | 2016-01-01T23:00:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2016-01-01T23:00Z | 2015-01-02T00:10Z | None | N |
| **1** | 2016-01-28T11:48:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2016-01-28T11:48Z | 2016-01-28T12:02Z | 2016-01-28T12:56Z | |
| **2** | 2016-02-04T18:15:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2016-02-04T18:15Z | 2016-02-04T18:22Z | 2016-02-04T18:28Z | |
| **3** | 2016-02-11T20:18:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2016-02-11T20:18Z | 2016-02-11T21:03Z | 2016-02-11T22:27Z | |
| **4** | 2016-02-12T10:37:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2016-02-12T10:37Z | 2016-02-12T10:47Z | 2016-02-12T10:53Z | N |

In [383…

```
def check_duplicates(df: pd.DataFrame)->str:
    #Rough check if there are ny duplicates
    col = list(df.columns)
    col.remove("linkedEvents")
    col.remove("instruments")
    duplicate_rows = df[col].duplicated().sum()

    print(duplicate_rows)
```

In [384…

```
check_duplicates(data_flare_df)
check_duplicates(data_sep_df)
```

```
0
0
```

Fortunately there are no duplicates in the data. Otherwise there is a function drop_duplicates() for such an issue.

In [385…

```
data_flare_df.isna().sum()
```

Out[385…

```
flrID                      0
catalog                    0
instruments                0
beginTime                  0
peakTime                   0
endTime                   40
classType                  0
sourceLocation             0
activeRegionNum           41
note                       0
submissionTime             0
versionId                  0
link                       0
linkedEvents             225
instrument_displayName     0
activityID                 0
dtype: int64
```

In [386…

```
data_flare_df.loc[pd.isna(data_flare_df["endTime"]), "activeRegionNum"].count()
```

Out[386…

```
28
```

As we can see in the data_flare_df there is missing value for end time for solar flare. According to NASA (from which this dataset is) solar flares cant last from minutes to hours so solar flare with no endTime is impossible. https://blogs.nasa.gov/solarcycle25/2022/06/10/solar-flares-faqs/ This seems like the type of MCAR - Missing at completely random. There is no relationship between this value missing and other values. Now for the active region number I expected relationship with endTime since it could influenced by each other based on let´s say faulty equipment. But then again region probably would be registered at the beginning, but there is no missing value for beginTime. So again it seems the MCAR. LinkedEvents are alright since there could be no linked events with solar flares.

For these missing values good approach would be to remove them so that they would not impact further analysis, but approach I want to try is filling them based on regression from the data where there is available end time.

EDIT: Except the first row which has peak time 1 year before, this will be droppend as it seems to be issue with equipment that the measurements are worng in terms of year measured but also missing one value. Run into an issue with incorrect date in year where instead of 20** I got 00** this simple function below resolves this issue by catching those that start with 00 and replacing it with 20. Also timezones are lozalized

In [567…

```
#All activeRegionNum that are Nan get NONE so that it does not cause any issue l
data_flare_df["activeRegionNum"] = data_flare_df["activeRegionNum"].where(pd.not
```

In [ ]:

```

```

In [ ]:

```
def standardize_date(df: pd.DataFrame)->pd.DataFrame:
    df.drop(index=0, inplace=True)


    df.loc[df["peakTime"].astype(str).str.startswith("00"), "peakTime"] = df["pe

    return df
```

In [388…

```
def timezones_naive(df: pd.DataFrame):
    df["beginTime"] = pd.to_datetime(df["beginTime"]).dt.tz_localize(None)
    df["peakTime"] = pd.to_datetime(df["peakTime"]).dt.tz_localize(None)
    df["endTime"] = pd.to_datetime(df["endTime"]).dt.tz_localize(None)

    return df
```

In [ ]:

```
data_flare_df = standardize_date(data_flare_df)
data_flare_df = timezones_naive(data_flare_df)

#Dropping also the first linked event in flare_linked events
flare_linked_events.drop(index=0, inplace=True)
flare_linked_events.reset_index()

columns = ["beginTime", "peakTime", "endTime"]
df_time = data_flare_df[columns].copy()

df_time.head()
```

In [390…

```
data_flare_df.iloc[199]
```

Out[390…

```
flrID                                            2021-12-16T03:44:00-FLR-001
catalog                                                          M2M_CATALOG
instruments                          [{'displayName': 'GOES-P: EXIS 1.0-8.0'}]
beginTime                                                2021-12-16 03:44:00
peakTime                                                 2021-12-16 03:54:00
endTime                                                  2021-12-16 04:04:00
classType                                                               C1.3
sourceLocation                                                        S21E78
activeRegionNum                                                      12909.0
note
submissionTime                                             2021-12-17T13:18Z
versionId                                                                  2
link                           https://webtools.ccmc.gsfc.nasa.gov/DONKI/view...
linkedEvents                      [{'activityID': '2021-12-16T04:24:00-CME-001'}]
instrument_displayName                                   GOES-P: EXIS 1.0-8.0
activityID                                                              True
Name: 200, dtype: object
```

In [391…

```
df_time.dtypes
```

Out[391…

```
beginTime    datetime64[ns]
peakTime     datetime64[ns]
endTime      datetime64[ns]
dtype: object
```

In [392…

```
df_time.dropna(subset=["endTime"], inplace=True)
df_time.head()
```

Out[392…

| | beginTime | peakTime | endTime |
|---|---|---|---|
| **1** | 2016-01-28 11:48:00 | 2016-01-28 12:02:00 | 2016-01-28 12:56:00 |
| **2** | 2016-02-04 18:15:00 | 2016-02-04 18:22:00 | 2016-02-04 18:28:00 |
| **3** | 2016-02-11 20:18:00 | 2016-02-11 21:03:00 | 2016-02-11 22:27:00 |
| **4** | 2016-02-12 10:37:00 | 2016-02-12 10:47:00 | 2016-02-12 10:53:00 |
| **5** | 2016-02-13 15:18:00 | 2016-02-13 15:24:00 | 2016-02-13 15:26:00 |

In [393…

```
df_time.dropna(subset=["endTime"], inplace=True)
df_time["beginTime"] = pd.to_datetime(df_time["beginTime"])
df_time["peakTime"] = pd.to_datetime(df_time["peakTime"])
df_time["endTime"] = pd.to_datetime(df_time["endTime"])
df_time.reset_index(drop=True, inplace=True)
df_time.dtypes
```

Out[393…

```
beginTime    datetime64[ns]
peakTime     datetime64[ns]
endTime      datetime64[ns]
dtype: object
```

Now I will go ahead with preparing datetime values for conversion so that I can create a model to predict them. Based on time we already have where the values are not missing. The biggest issue with time data is that is it in cycle, but this issue is already resolved just harder to implement then when there is no cycle. It is resolved by using sinus and cosinus. Because cosinues and sinus both have cyclical graph so using with time series is very beneficial. Also it reduces dimensionality from 24 (24 hours) to 2 (2 - cos and sin). Also there is no connectivity in these data 23 hour does not know it is followed by 0 hour. Here are the steps I took to create the model:

1. Find if there is correlation between data (Yes)
2. Separate values for hours, minutes and seconds
3. Create transformations for cos and sin
4. Split data test/training datasets, usually 80/20 split
5. Create model
6. Train the model
7. Create a conversion function to turn radius back to hour, minutes and seconds
8. Create evaluation for the model
9. Use the model for prediction
10. Add the predicted data to where the data is missing in data from API

In [394…

```
df_time["beginTime"][0]
```

Out[394…

```
Timestamp('2016-01-28 11:48:00')
```

In [395…

```
df_time["peakTime"][0]
```

Out[395…

```
Timestamp('2016-01-28 12:02:00')
```

In [396…

```
df_time.corr()
```

Out[396…

| | beginTime | peakTime | endTime |
|---|---|---|---|
| **beginTime** | 1.000000 | 1.000000 | 0.998968 |
| **peakTime** | 1.000000 | 1.000000 | 0.998968 |
| **endTime** | 0.998968 | 0.998968 | 1.000000 |

From corr() we can see that there is a high correlation between times and that shows that they should be great predictors for predicting missing values in endTime.

In [ ]:
```
#Create new column for each part of _Time
def add_separete_time_values(df: pd.DataFrame)-> pd.DataFrame:
    df["beginTime"] = pd.to_datetime(df["beginTime"])
    df["peakTime"] = pd.to_datetime(df["peakTime"])

    df["beginTime_hour"] = df["beginTime"].dt.hour
    df["beginTime_minute"] = df["beginTime"].dt.minute
    df["beginTime_second"] = df["beginTime"].dt.second

    df["peakTime_hour"] = df["peakTime"].dt.hour
    df["peakTime_minute"] = df["peakTime"].dt.minute
    df["peakTime_second"] = df["peakTime"].dt.second

    if df["endTime"].notna().all():
        df["endTime"] = pd.to_datetime(df["endTime"])
        df["endTime_hour"] = df["endTime"].dt.hour
        df["endTime_minute"] = df["endTime"].dt.minute
        df["endTime_second"] = df["endTime"].dt.second

    return df
```

In [398…
```
df_time = add_separete_time_values(df_time)
df_time.head()
```

Out[398…

| | beginTime | peakTime | endTime | beginTime_hour | beginTime_minute | beginTime_seco |
|---|---|---|---|---|---|---|
| **0** | 2016-01-28 11:48:00 | 2016-01-28 12:02:00 | 2016-01-28 12:56:00 | 11 | 48 | |
| **1** | 2016-02-04 18:15:00 | 2016-02-04 18:22:00 | 2016-02-04 18:28:00 | 18 | 15 | |
| **2** | 2016-02-11 20:18:00 | 2016-02-11 21:03:00 | 2016-02-11 22:27:00 | 20 | 18 | |
| **3** | 2016-02-12 10:37:00 | 2016-02-12 10:47:00 | 2016-02-12 10:53:00 | 10 | 37 | |
| **4** | 2016-02-13 15:18:00 | 2016-02-13 15:24:00 | 2016-02-13 15:26:00 | 15 | 18 | |

In [399…

```
df_time.dtypes
```

Out[399…

```
beginTime            datetime64[ns]
peakTime             datetime64[ns]
endTime              datetime64[ns]
beginTime_hour                int32
beginTime_minute              int32
beginTime_second              int32
peakTime_hour                 int32
peakTime_minute               int32
peakTime_second               int32
endTime_hour                  int32
endTime_minute                int32
endTime_second                int32
dtype: object
```

In [400…

```
def sin_transformer(period:int)->FunctionTransformer:
        return FunctionTransformer(lambda x: np.sin(x / period * 2 * np.pi))

def cos_transformer(period:int)->FunctionTransformer:
        return FunctionTransformer(lambda x: np.cos(x / period * 2 * np.pi))
```

In [401…

```
def transform_time(df: pd.DataFrame, period = 60, period_h = 24)-> pd.DataFrame:

    hour_columns = df.columns[df.columns.str.contains("_hour")]
    minute_columns = df.columns[df.columns.str.contains("_minute")]
    seconds_columns = df.columns[df.columns.str.contains("_second")]

    for col in hour_columns:
        df[col + "_sin"] = sin_transformer(period_h).fit_transform(df[[col]])
        df[col + "_cos"] = cos_transformer(period_h).fit_transform(df[[col]])
    for col in minute_columns:
        df[col + "_sin"] = sin_transformer(period).fit_transform(df[[col]])
        df[col + "_cos"] = cos_transformer(period).fit_transform(df[[col]])
    for col in seconds_columns:
        df[col + "_sin"] = sin_transformer(period).fit_transform(df[[col]])
        df[col + "_cos"] = cos_transformer(period).fit_transform(df[[col]])

    return df
```

In [402…

```
df_time_transformed = transform_time(df_time)
df_time_transformed.head()
```

Out[402…

| | beginTime | peakTime | endTime | beginTime_hour | beginTime_minute | beginTime_seco |
|---|---|---|---|---|---|---|
| **0** | 2016-01-28 11:48:00 | 2016-01-28 12:02:00 | 2016-01-28 12:56:00 | 11 | 48 | |
| **1** | 2016-02-04 18:15:00 | 2016-02-04 18:22:00 | 2016-02-04 18:28:00 | 18 | 15 | |
| **2** | 2016-02-11 20:18:00 | 2016-02-11 21:03:00 | 2016-02-11 22:27:00 | 20 | 18 | |
| **3** | 2016-02-12 10:37:00 | 2016-02-12 10:47:00 | 2016-02-12 10:53:00 | 10 | 37 | |
| **4** | 2016-02-13 15:18:00 | 2016-02-13 15:24:00 | 2016-02-13 15:26:00 | 15 | 18 | |

5 rows × 30 columns

Now to split the data for training dataset and testing dataset. With this data there is an issue that they are chronological so we need different approach then splitting it randomly with train_test_split

In [403…

```
#Loop for all the columns which are transformed but not the endTime columns
time_cols = [col for col in df_time_transformed.columns
               if ("_sin" in col or "_cos" in col)
               and "endTime" not in col]

X = df_time_transformed[time_cols]

#Loop for only the endTime columns
target_cols = [col for col in df_time_transformed.columns
              if "endTime" in col and ("_sin" in col or "_cos" in col)]

y = df_time_transformed[target_cols]

split_df = int(len(df_time_transformed) * 0.8)

#Split on interval of 80/20
X_train = X.iloc[:split_df]
X_test = X.iloc[split_df:]
y_train = y.iloc[:split_df]
y_test = y.iloc[split_df:]
```

Now for the model itself. I will be using Simple linear regression but with the wrapper MultiOutputRegressor which can handle mulkti output for single output model. For the evalution I will use k-fold cross validation which is standartd method for evaluation. (https://machinelearningmastery.com/repeated-k-fold-cross-validation-with-python/)

For the parameters common numbers of repeats include 3, 5, and 10. For example, if 3 repeats of 10-fold cross-validation are used to estimate the model performance, this means that (3 * 10) or 30 different models would need to be fit and evaluated.That is good for small datasets and simple models (e.g. linear).

In [404…
```
#Creating a model (Linear Regression model)
model = LinearSVR(max_iter=10000) #Runinng into error if default value
wrapper = MultiOutputRegressor(model)
cv = RepeatedKFold(n_splits=10, n_repeats=3, random_state=1)

n_scores = cross_val_score(wrapper, X_train, y_train, scoring="neg_mean_absolute

n_scores = np.absolute(n_scores)
print("MAE: %.3f (%.3f)" % (np.mean(n_scores), np.std(n_scores)))
```

```
MAE: 0.148 (0.019)
```

In [ ]:
```
#Train final model on all training data
wrapper.fit(X_train, y_train)

#Make predictions
y_pred = wrapper.predict(X_test)

#Calculate error
test_mae = mean_absolute_error(y_test, y_pred)
print(test_mae)
```

```
0.12028861105153071
```

MAE - Mean absolute error measures the average of the absolute differences between predicted(Y´) and actual (Y)values. The 0.120 value represents the average absolute error in the sine/cosine space. On average the model prediction deviate from real values by 0.120 units.

In [406…
```
#Create dataframe with predicted endTime values
pred_columns = y_train.columns
y_pred_df = pd.DataFrame(y_pred, columns=pred_columns, index=X_test.index)
```

In [407…
```
y_pred_df.head()
```

Out[407…

| | endTime_hour_sin | endTime_hour_cos | endTime_minute_sin | endTime_minute_cos | e |
|---|---|---|---|---|---|
| **381** | -0.865995 | 0.499988 | 0.695017 | -0.322973 | |
| **382** | 0.500021 | 0.866063 | 0.037296 | -0.824428 | |
| **383** | -0.499986 | -0.866006 | 0.710713 | 0.275146 | |
| **384** | -0.707115 | -0.707119 | -0.559525 | 0.611141 | |
| **385** | -0.865994 | 0.499977 | -0.738472 | -0.624507 | |

In [408…
```
def conversion_to_time(prediction: pd.DataFrame, prefix="endTime")->pd.DataFrame
    components = {}

    hour_sin_col = f"{prefix}_hour_sin"
    hour_cos_col = f"{prefix}_hour_cos"
    minute_sin_col = f"{prefix}_minute_sin"
    minute_cos_col = f"{prefix}_minute_cos"
    second_sin_col = f"{prefix}_second_sin"
    second_cos_col = f"{prefix}_second_cos"

    if hour_sin_col in prediction.columns and hour_cos_col in prediction.columns
        hour_sin = prediction[hour_sin_col]
        hour_cos = prediction[hour_cos_col]

        #Handle NaN and zeros
        if np.all(np.isclose(hour_sin, 0) & np.isclose(hour_cos, 0)):
            hour = 0
        else:

            hour_sin = np.clip(hour_sin, -1, 1)
            hour_cos = np.clip(hour_cos, -1, 1)

            hour_radians = np.arctan2(hour_sin, hour_cos)
            hour = (hour_radians % (2 * np.pi)) * 24 / (2 * np.pi)

        components[f"{prefix}_hour"] = hour.fillna(0).round(0).astype(int) % 24

    if minute_sin_col in prediction.columns and minute_cos_col in prediction.col
        minute_sin = prediction[minute_sin_col]
        minute_cos = prediction[minute_cos_col]

        if np.all(np.isclose(minute_sin, 0) & np.isclose(minute_cos, 0)):
            minute = 0
        else:
            minute_sin = np.clip(minute_sin, -1, 1)
            minute_cos = np.clip(minute_cos, -1, 1)

            minute_radians = np.arctan2(minute_sin, minute_cos)
            minute = (minute_radians % (2 * np.pi)) * 60 / (2 * np.pi)

        components[f"{prefix}_minute"] = minute.fillna(0).round(0).astype(int) %

    if second_sin_col in prediction.columns and second_cos_col in prediction.col
        second_sin = prediction[second_sin_col]
        second_cos = prediction[second_cos_col]

        if np.all(np.isclose(second_sin, 0) & np.isclose(second_cos, 0)):
            second = 0
        else:
            second_sin = np.clip(second_sin, -1, 1)
            second_cos = np.clip(second_cos, -1, 1)

            second_radians = np.arctan2(second_sin, second_cos)
            second = (second_radians % (2 * np.pi)) * 60 / (2 * np.pi)

        components[f"{prefix}_second"] = second.fillna(0).round(0).astype(int) %

    results_df = pd.DataFrame(components)

    hour_col = f"{prefix}_hour"
    minute_col = f"{prefix}_minute"
    second_col = f"{prefix}_second"

    if all(k in results_df for k in [hour_col, minute_col, second_col]):
        results_df[f"{prefix}_formated"] = (
            results_df[hour_col].astype(str).str.zfill(2) + ":" +
            results_df[minute_col].astype(str).str.zfill(2) + ":" +
            results_df[second_col].astype(str).str.zfill(2)
        )

    return results_df
```

In [409…

```
y_pred_df.head()
```

Out[409…

| | endTime_hour_sin | endTime_hour_cos | endTime_minute_sin | endTime_minute_cos | e |
|---|---|---|---|---|---|
| **381** | -0.865995 | 0.499988 | 0.695017 | -0.322973 | |
| **382** | 0.500021 | 0.866063 | 0.037296 | -0.824428 | |
| **383** | -0.499986 | -0.866006 | 0.710713 | 0.275146 | |
| **384** | -0.707115 | -0.707119 | -0.559525 | 0.611141 | |
| **385** | -0.865994 | 0.499977 | -0.738472 | -0.624507 | |

In [410…

```
results_df = conversion_to_time(y_pred_df)
results_df.head()
```

Out[410…

| | endTime_hour | endTime_minute | endTime_second | endTime_formated |
|---|---|---|---|---|
| **381** | 20 | 19 | 0 | 20:19:00 |
| **382** | 2 | 30 | 0 | 02:30:00 |
| **383** | 14 | 11 | 0 | 14:11:00 |
| **384** | 15 | 53 | 0 | 15:53:00 |
| **385** | 20 | 38 | 0 | 20:38:00 |

Now for prediction of where the end time is missing.

In [411…

```
columns = ["beginTime", "peakTime", "endTime"]
missing_endTime_df = data_flare_df[columns].copy()
missing_endTime_df = missing_endTime_df[missing_endTime_df["endTime"].isna()]
missing_endTime_df.head()
```

Out[411…

| | beginTime | peakTime | endTime |
|---|---|---|---|
| **13** | 2016-07-07 07:49:00 | 2016-07-07 07:56:00 | NaT |
| **14** | 2016-07-10 00:53:00 | 2016-07-10 00:59:00 | NaT |
| **39** | 2017-04-18 09:29:00 | 2017-04-18 09:41:00 | NaT |
| **40** | 2017-04-18 19:15:00 | 2017-04-18 20:10:00 | NaT |
| **41** | 2017-06-02 17:51:00 | 2017-06-02 17:57:00 | NaT |

In [412…

```
missing_endTime_df = add_separete_time_values(missing_endTime_df)
missing_endTime_transformed_df = transform_time(missing_endTime_df)
missing_endTime_transformed_df.head()
```

Out[412…

| | beginTime | peakTime | endTime | beginTime_hour | beginTime_minute | beginTime_se |
|---|---|---|---|---|---|---|
| **13** | 2016-07-07 07:49:00 | 2016-07-07 07:56:00 | NaT | 7 | 49 | |
| **14** | 2016-07-10 00:53:00 | 2016-07-10 00:59:00 | NaT | 0 | 53 | |
| **39** | 2017-04-18 09:29:00 | 2017-04-18 09:41:00 | NaT | 9 | 29 | |
| **40** | 2017-04-18 19:15:00 | 2017-04-18 20:10:00 | NaT | 19 | 15 | |
| **41** | 2017-06-02 17:51:00 | 2017-06-02 17:57:00 | NaT | 17 | 51 | |

5 rows × 21 columns

In [413…

```
#Loop for all the columns which are transformed but not the endTime columns
time_cols_2 = [col for col in missing_endTime_transformed_df.columns
               if ("_sin" in col or "_cos" in col)
               and "endTime" not in col]

x_missing = missing_endTime_transformed_df[time_cols_2]

endtime_predict = wrapper.predict(x_missing)
endtime_predict_df = pd.DataFrame(endtime_predict, columns=pred_columns, index=m
```

In [414…

```
endtime_predict_df.head()
```

Out[414…

| | endTime_hour_sin | endTime_hour_cos | endTime_minute_sin | endTime_minute_cos | en |
|---|---|---|---|---|---|
| **13** | 0.965941 | -0.258803 | 0.385402 | 0.906454 | |
| **14** | 0.000009 | 1.000035 | 0.709196 | 0.663143 | |
| **39** | 0.707116 | -0.707104 | -0.774298 | 0.372383 | |
| **40** | -0.865996 | 0.499995 | 0.934158 | -0.632657 | |
| **41** | -0.965931 | -0.258821 | 0.528049 | 0.798089 | |

In [415…

```
endtime_predict_converted_df = conversion_to_time(endtime_predict_df)
endtime_predict_converted_df.head()
```

Out[415…

|  | endTime_hour | endTime_minute | endTime_second | endTime_formated |
|---|---|---|---|---|
| **13** | 7 | 4 | 0 | 07:04:00 |
| **14** | 0 | 8 | 0 | 00:08:00 |
| **39** | 9 | 49 | 0 | 09:49:00 |
| **40** | 20 | 21 | 0 | 20:21:00 |
| **41** | 17 | 6 | 0 | 17:06:00 |

In [416…
```
data_flare_df.update(endtime_predict_converted_df["endTime_formated"])
```

In [ ]:
```
#Appending predicted values to the dataframe based on their indexes
def append_predicted_time(df: pd.DataFrame, predicted: pd.DataFrame)->pd.DataFra
    df["peakTime"] = pd.to_datetime(df["peakTime"])

    predicted["endTime_formated_date"] = pd.to_datetime(df["peakTime"].dt.strfti

    predicted["endTime_formated_date"] = pd.to_datetime(predicted["endTime_forma

    df.loc[df["endTime"].isna(), "endTime"] = predicted.loc[df["endTime"].isna()

    return df
```

Checking if the original df and dataframe with predicted values have the same length

In [418…
```
print(len(data_flare_df[data_flare_df["endTime"].isna()]))
print(len(endtime_predict_converted_df["endTime_formated"]))
```

```
39
39
```

In [419…
```
data_flare_df.head(40)
```

Out[419…

| | flrID | catalog | instruments | beginTime | peakTime | endTime | class |
|---|---|---|---|---|---|---|---|
| **1** | 2016-01-28T11:48:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2016-01-28 11:48:00 | 2016-01-28 12:02:00 | 2016-01-28 12:56:00 | |
| **2** | 2016-02-04T18:15:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2016-02-04 18:15:00 | 2016-02-04 18:22:00 | 2016-02-04 18:28:00 | |
| **3** | 2016-02-11T20:18:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2016-02-11 20:18:00 | 2016-02-11 21:03:00 | 2016-02-11 22:27:00 | |
| **4** | 2016-02-12T10:37:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2016-02-12 10:37:00 | 2016-02-12 10:47:00 | 2016-02-12 10:53:00 | |
| **5** | 2016-02-13T15:18:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2016-02-13 15:18:00 | 2016-02-13 15:24:00 | 2016-02-13 15:26:00 | |
| **6** | 2016-02-14T19:20:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2016-02-14 19:20:00 | 2016-02-14 19:26:00 | 2016-02-14 19:29:00 | |
| **7** | 2016-02-15T10:41:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2016-02-15 10:41:00 | 2016-02-15 11:00:00 | 2016-02-15 11:06:00 | |
| **8** | 2016-02-17T04:54:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2016-02-17 04:54:00 | 2016-02-17 05:01:00 | 2016-02-17 05:07:00 | |
| **9** | 2016-03-16T06:34:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2016-03-16 06:34:00 | 2016-03-16 06:45:00 | 2016-03-16 06:57:00 | |
| **10** | 2016-04-09T12:08:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2016-04-09 12:08:00 | 2016-04-09 13:42:00 | 2016-04-09 16:00:00 | |
| **11** | 2016-04-18T00:14:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: | 2016-04-18 00:14:00 | 2016-04-18 00:29:00 | 2016-04-18 00:39:00 | |

| | flrID | catalog | instruments | beginTime | peakTime | endTime | class |
|---|---|---|---|---|---|---|---|
| | | | SEM/XRS 1.0-8.0'}] | | | | |
| **12** | 2016-06-27T09:42:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2016-06-27 09:42:00 | 2016-06-27 09:58:00 | 2016-06-27 10:24:00 | |
| **13** | 2016-07-07T07:49:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2016-07-07 07:49:00 | 2016-07-07 07:56:00 | NaT | |
| **14** | 2016-07-10T00:53:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2016-07-10 00:53:00 | 2016-07-10 00:59:00 | NaT | |
| **15** | 2016-07-21T00:41:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2016-07-21 00:41:00 | 2016-07-21 00:46:00 | 2016-07-21 01:15:00 | |
| **16** | 2016-07-21T01:34:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2016-07-21 01:34:00 | 2016-07-21 01:48:00 | 2016-07-21 03:15:00 | |
| **17** | 2016-07-23T01:46:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2016-07-23 01:46:00 | 2016-07-23 02:11:00 | 2016-07-23 02:23:00 | |
| **18** | 2016-07-23T05:00:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2016-07-23 05:00:00 | 2016-07-23 05:16:00 | 2016-07-23 05:24:00 | |
| **19** | 2016-07-23T05:27:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2016-07-23 05:27:00 | 2016-07-23 05:31:00 | 2016-07-23 05:33:00 | |
| **20** | 2016-07-24T06:09:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2016-07-24 06:09:00 | 2016-07-24 06:20:00 | 2016-07-24 06:32:00 | |
| **21** | 2016-07-24T17:30:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2016-07-24 17:30:00 | 2016-07-24 17:43:00 | 2016-07-24 18:12:00 | |

| | flrID | catalog | instruments | beginTime | peakTime | endTime | class |
|---|---|---|---|---|---|---|---|
| **22** | 2016-08-07T14:37:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2016-08-07 14:37:00 | 2016-08-07 14:44:00 | 2016-08-07 14:48:00 | |
| **23** | 2016-08-09T00:34:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2016-08-09 00:34:00 | 2016-08-09 00:42:00 | 2016-08-09 00:52:00 | |
| **24** | 2016-11-29T17:19:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2016-11-29 17:19:00 | 2016-11-29 17:23:00 | 2016-11-29 17:26:00 | |
| **25** | 2016-11-29T23:29:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2016-11-29 23:29:00 | 2016-11-29 23:38:00 | 2016-11-30 23:40:00 | |
| **26** | 2016-12-10T16:48:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2016-12-10 16:48:00 | 2016-12-10 17:15:00 | 2016-12-10 17:35:00 | |
| **27** | 2017-01-21T07:23:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2017-01-21 07:23:00 | 2017-01-21 07:26:00 | 2017-01-21 07:37:00 | |
| **28** | 2017-03-27T11:07:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2017-03-27 11:07:00 | 2017-03-27 11:12:00 | 2017-03-27 12:43:00 | |
| **29** | 2017-03-27T17:55:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2017-03-27 17:55:00 | 2017-03-27 18:20:00 | 2017-03-27 18:47:00 | |
| **30** | 2017-04-01T19:30:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2017-04-01 19:30:00 | 2017-04-01 19:56:00 | 2017-04-01 20:13:00 | |
| **31** | 2017-04-01T21:35:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2017-04-01 21:35:00 | 2017-04-01 21:48:00 | 2017-04-01 22:05:00 | |
| **32** | 2017-04-02T02:43:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2017-04-02 02:43:00 | 2017-04-02 02:46:00 | 2017-04-02 02:51:00 | |

| | flrID | catalog | instruments | beginTime | peakTime | endTime | class |
|---|---|---|---|---|---|---|---|
| **33** | 2017-04-02T07:48:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2017-04-02 07:48:00 | 2017-04-02 08:02:00 | 2017-04-02 08:13:00 | I |
| **34** | 2017-04-02T12:54:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2017-04-02 12:54:00 | 2017-04-02 13:00:00 | 2017-04-02 13:11:00 | I |
| **35** | 2017-04-02T18:18:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2017-04-02 18:18:00 | 2017-04-02 18:38:00 | 2017-04-02 19:28:00 | I |
| **36** | 2017-04-02T20:28:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2017-04-02 20:28:00 | 2017-04-02 20:33:00 | 2017-04-02 20:38:00 | I |
| **37** | 2017-04-03T00:54:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2017-04-03 00:54:00 | 2017-04-03 01:05:00 | 2017-04-03 01:12:00 | I |
| **38** | 2017-04-03T14:21:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2017-04-03 14:21:00 | 2017-04-03 14:29:00 | 2017-04-03 14:34:00 | I |
| **39** | 2017-04-18T09:29:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2017-04-18 09:29:00 | 2017-04-18 09:41:00 | NaT | |
| **40** | 2017-04-18T19:15:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2017-04-18 19:15:00 | 2017-04-18 20:10:00 | NaT | |

In [420…

```
data_flare_df = append_predicted_time(data_flare_df, endtime_predict_converted_d
data_flare_df.head(40)
```

Out[420…

| | flrID | catalog | instruments | beginTime | peakTime | endTime | class |
|---|---|---|---|---|---|---|---|
| **1** | 2016-01-28T11:48:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2016-01-28 11:48:00 | 2016-01-28 12:02:00 | 2016-01-28 12:56:00 | |
| **2** | 2016-02-04T18:15:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2016-02-04 18:15:00 | 2016-02-04 18:22:00 | 2016-02-04 18:28:00 | |
| **3** | 2016-02-11T20:18:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2016-02-11 20:18:00 | 2016-02-11 21:03:00 | 2016-02-11 22:27:00 | |
| **4** | 2016-02-12T10:37:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2016-02-12 10:37:00 | 2016-02-12 10:47:00 | 2016-02-12 10:53:00 | |
| **5** | 2016-02-13T15:18:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2016-02-13 15:18:00 | 2016-02-13 15:24:00 | 2016-02-13 15:26:00 | |
| **6** | 2016-02-14T19:20:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2016-02-14 19:20:00 | 2016-02-14 19:26:00 | 2016-02-14 19:29:00 | |
| **7** | 2016-02-15T10:41:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2016-02-15 10:41:00 | 2016-02-15 11:00:00 | 2016-02-15 11:06:00 | |
| **8** | 2016-02-17T04:54:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2016-02-17 04:54:00 | 2016-02-17 05:01:00 | 2016-02-17 05:07:00 | |
| **9** | 2016-03-16T06:34:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2016-03-16 06:34:00 | 2016-03-16 06:45:00 | 2016-03-16 06:57:00 | |
| **10** | 2016-04-09T12:08:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2016-04-09 12:08:00 | 2016-04-09 13:42:00 | 2016-04-09 16:00:00 | |
| **11** | 2016-04-18T00:14:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: | 2016-04-18 00:14:00 | 2016-04-18 00:29:00 | 2016-04-18 00:39:00 | |

| | flrID | catalog | instruments | beginTime | peakTime | endTime | class |
|---|---|---|---|---|---|---|---|
| | | | SEM/XRS 1.0-8.0'}] | | | | |
| **12** | 2016-06-27T09:42:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2016-06-27 09:42:00 | 2016-06-27 09:58:00 | 2016-06-27 10:24:00 | |
| **13** | 2016-07-07T07:49:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2016-07-07 07:49:00 | 2016-07-07 07:56:00 | 2016-07-07 07:04:00 | |
| **14** | 2016-07-10T00:53:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2016-07-10 00:53:00 | 2016-07-10 00:59:00 | 2016-07-10 00:08:00 | |
| **15** | 2016-07-21T00:41:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2016-07-21 00:41:00 | 2016-07-21 00:46:00 | 2016-07-21 01:15:00 | |
| **16** | 2016-07-21T01:34:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2016-07-21 01:34:00 | 2016-07-21 01:48:00 | 2016-07-21 03:15:00 | |
| **17** | 2016-07-23T01:46:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2016-07-23 01:46:00 | 2016-07-23 02:11:00 | 2016-07-23 02:23:00 | |
| **18** | 2016-07-23T05:00:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2016-07-23 05:00:00 | 2016-07-23 05:16:00 | 2016-07-23 05:24:00 | |
| **19** | 2016-07-23T05:27:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2016-07-23 05:27:00 | 2016-07-23 05:31:00 | 2016-07-23 05:33:00 | |
| **20** | 2016-07-24T06:09:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2016-07-24 06:09:00 | 2016-07-24 06:20:00 | 2016-07-24 06:32:00 | |
| **21** | 2016-07-24T17:30:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2016-07-24 17:30:00 | 2016-07-24 17:43:00 | 2016-07-24 18:12:00 | |

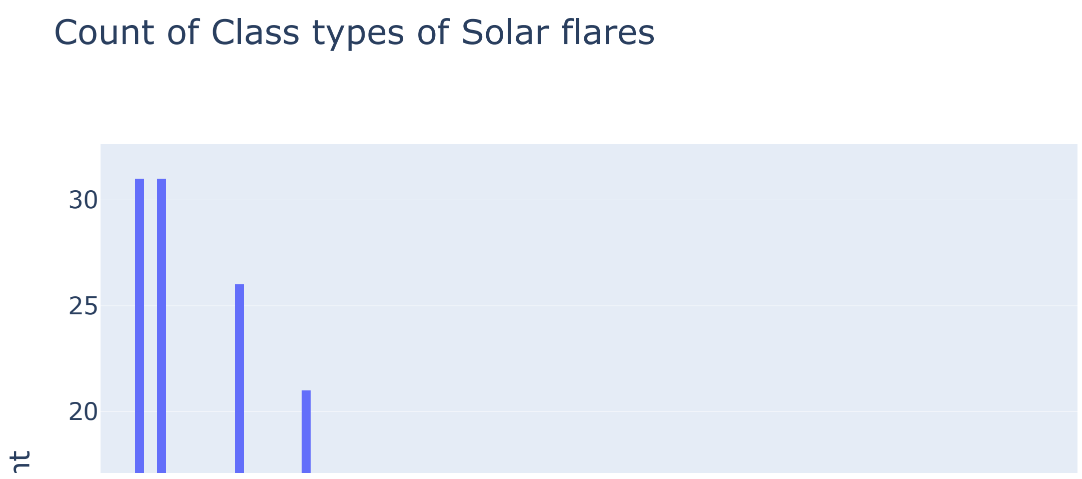
| | flrID | catalog | instruments | beginTime | peakTime | endTime | class |
|---|---|---|---|---|---|---|---|
| **22** | 2016-08-07T14:37:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2016-08-07 14:37:00 | 2016-08-07 14:44:00 | 2016-08-07 14:48:00 | |
| **23** | 2016-08-09T00:34:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2016-08-09 00:34:00 | 2016-08-09 00:42:00 | 2016-08-09 00:52:00 | |
| **24** | 2016-11-29T17:19:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2016-11-29 17:19:00 | 2016-11-29 17:23:00 | 2016-11-29 17:26:00 | |
| **25** | 2016-11-29T23:29:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2016-11-29 23:29:00 | 2016-11-29 23:38:00 | 2016-11-30 23:40:00 | |
| **26** | 2016-12-10T16:48:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2016-12-10 16:48:00 | 2016-12-10 17:15:00 | 2016-12-10 17:35:00 | |
| **27** | 2017-01-21T07:23:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2017-01-21 07:23:00 | 2017-01-21 07:26:00 | 2017-01-21 07:37:00 | |
| **28** | 2017-03-27T11:07:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2017-03-27 11:07:00 | 2017-03-27 11:12:00 | 2017-03-27 12:43:00 | |
| **29** | 2017-03-27T17:55:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2017-03-27 17:55:00 | 2017-03-27 18:20:00 | 2017-03-27 18:47:00 | |
| **30** | 2017-04-01T19:30:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2017-04-01 19:30:00 | 2017-04-01 19:56:00 | 2017-04-01 20:13:00 | |
| **31** | 2017-04-01T21:35:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2017-04-01 21:35:00 | 2017-04-01 21:48:00 | 2017-04-01 22:05:00 | |
| **32** | 2017-04-02T02:43:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2017-04-02 02:43:00 | 2017-04-02 02:46:00 | 2017-04-02 02:51:00 | |

| | flrID | catalog | instruments | beginTime | peakTime | endTime | class |
|---|---|---|---|---|---|---|---|
| **33** | 2017-04-02T07:48:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2017-04-02 07:48:00 | 2017-04-02 08:02:00 | 2017-04-02 08:13:00 | I |
| **34** | 2017-04-02T12:54:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2017-04-02 12:54:00 | 2017-04-02 13:00:00 | 2017-04-02 13:11:00 | I |
| **35** | 2017-04-02T18:18:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2017-04-02 18:18:00 | 2017-04-02 18:38:00 | 2017-04-02 19:28:00 | I |
| **36** | 2017-04-02T20:28:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2017-04-02 20:28:00 | 2017-04-02 20:33:00 | 2017-04-02 20:38:00 | I |
| **37** | 2017-04-03T00:54:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2017-04-03 00:54:00 | 2017-04-03 01:05:00 | 2017-04-03 01:12:00 | I |
| **38** | 2017-04-03T14:21:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2017-04-03 14:21:00 | 2017-04-03 14:29:00 | 2017-04-03 14:34:00 | I |
| **39** | 2017-04-18T09:29:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2017-04-18 09:29:00 | 2017-04-18 09:41:00 | 2017-04-18 09:49:00 | |
| **40** | 2017-04-18T19:15:00-FLR-001 | M2M_CATALOG | [{'displayName': 'GOES15: SEM/XRS 1.0-8.0'}] | 2017-04-18 19:15:00 | 2017-04-18 20:10:00 | 2017-04-18 20:21:00 | |

Now that the End Time with missing values have been resolved. Let do some exploration of the dataset with classical graph and plotly library.

In [573…
```
pio.renderers.default = "plotly_mimetype+notebook"
```

In [574…
```
fig_his = px.histogram(data_flare_df,
                       title="Count of Class types of Solar flares",
                       labels={"classType": "Class type"},
                       x="classType")
fig_his.show()
```



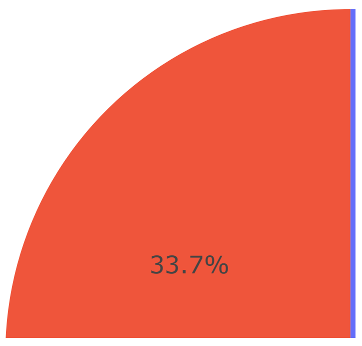
In the graph above we can see counts of each Solar flare type in histogram. Lot of types only appear once so lets visualize it in pie graph with only the parent class A,B,c etc.

In [575…

```
data_flare_df["parentClass"] = data_flare_df["classType"].str[0]
parent_counts = data_flare_df["parentClass"].value_counts()

fig_pie = px.pie(data_flare_df, values=parent_counts.values, names=parent_counts
fig_pie.show()
```

Counts of solar flare types



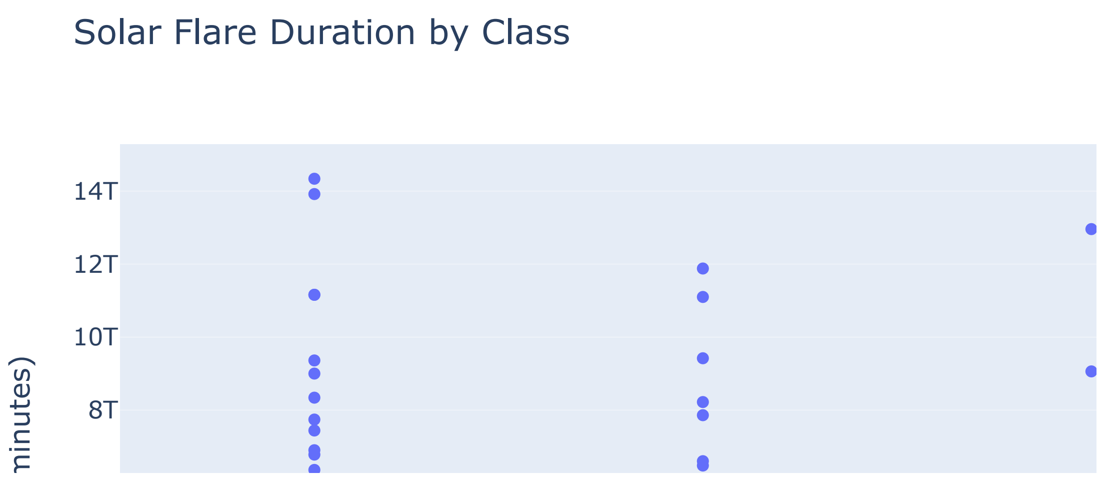
Here we see that there are dominant M class solar flares M which can cause brief radio blackouts that affect Earth's polar regions and minor radiation storms. Followed by C lass which are weak and have very limited consequences. (https://solar-center.stanford.edu/sid/activities/flare.html)

In [576…

```
data_flare_df["duration"] = list(data_flare_df["endTime"] - data_flare_df["begin
plot_df = data_flare_df.copy()
plot_df = plot_df[abs(plot_df["duration"]) <= pd.Timedelta(days=1)]

fig_box = px.box(
    plot_df,
    x="parentClass",
    y="duration",
    title="Solar Flare Duration by Class",
    labels={"parentClass": "Flare class", "duration": "Duration (minutes)"}
)

fig_box.show()
```



Here we can see that even when the max of duration on 1 there is many outliers in all clases, there is not enough data for class A. That is why the bos plot is non existent for this class.

In [425...
```
long_flares = data_flare_df[data_flare_df["duration"] > pd.Timedelta(days=1)]
print(long_flares["duration"].count())
```

```
4
```

So there are are 4 indetified solar flares that were over 1 day long which is statistically impossible so these are either wrong measurements or were incorrectly append by the trained model.

In [426...
```
minus_time_flares = data_flare_df[data_flare_df["duration"] <= pd.Timedelta(days
print(minus_time_flares["duration"].count())
```

```
9
```

Now we have 9 values which have minus values of timedelta endTime-Starttime, which could mean that the model incorrectly predicted the values of endTime for some of the records.

In [427...
```
normal_flares = data_flare_df[data_flare_df["duration"] >= pd.Timedelta(days=0)]
normal_flares = normal_flares[normal_flares["duration"] <= pd.Timedelta(days=1)]
```

```
avg_duration = normal_flares["duration"].mean()
print(avg_duration)
```

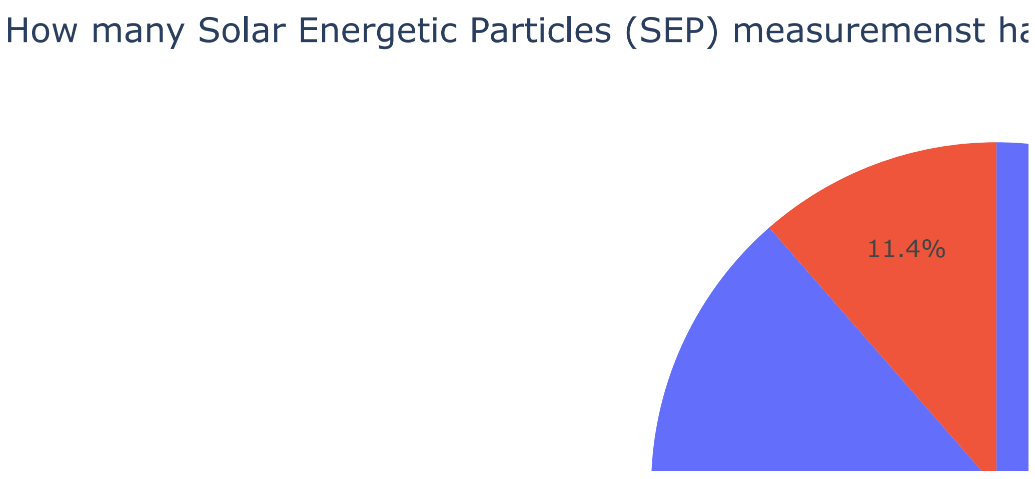
```
0 days 00:33:15.029821073
```

So the avg duration of solar flare is about 33min. A solar flare itself doesn't last more than a day, but its effects (CMEs, SEPs, geomagnetic storms) can persist for several days. That is where the the data for data_sep df comes into play.

In [577…

```
activity_counts = data_sep_df["activityID"].value_counts()

fig = px.pie(data_sep, values=activity_counts.values, names=activity_counts.inde
fig.show()
```



We can see that stunning 88,6% have linked event or are linked event. In the next step I will try to link these two dataframes based on these linked activities. Meaning that 88% of Solar energatic particles could be result of Solar flare or Coronal Mass Ejection(which I do not track here.) I have laready creted a dataframe for this at the begininng called flare_linked_events_df which tracks what solar flare was id with flrID and what activity linked to it.

In [430…

```
flare_linked_events.head()
```

Out[430…

| | flrID | activityID |
|---|---|---|
| **0** | 2016-01-01T23:00:00-FLR-001 | 2016-01-01T23:12:00-CME-001 |
| **1** | 2016-01-01T23:00:00-FLR-001 | 2016-01-02T02:48:00-SEP-001 |
| **2** | 2016-01-01T23:00:00-FLR-001 | 2016-01-02T04:30:00-SEP-001 |
| **3** | 2016-01-28T11:48:00-FLR-001 | 2016-01-28T12:24:00-CME-001 |
| **4** | 2016-02-11T20:18:00-FLR-001 | 2016-02-11T21:28:00-CME-001 |

In [431…

```
flare_linked_events.shape
```

Out[431…

```
(349, 2)
```

In [432…

```
solar_linked_events.head()
```

Out[432…

| | sepID | activityID |
|---|---|---|
| **0** | 2016-01-02T02:48:00-SEP-001 | 2016-01-01T23:00:00-FLR-001 |
| **1** | 2016-01-02T02:48:00-SEP-001 | 2016-01-01T23:12:00-CME-001 |
| **2** | 2016-01-02T04:30:00-SEP-001 | 2016-01-01T23:00:00-FLR-001 |
| **3** | 2016-01-02T04:30:00-SEP-001 | 2016-01-01T23:12:00-CME-001 |
| **4** | 2017-04-18T23:39:00-SEP-001 | 2017-04-18T19:15:00-FLR-001 |

In [433…

```
only_sep_linked_events = flare_linked_events.loc[flare_linked_events["activityID
only_sep_linked_events.head()
```

Out[433…

| | flrID | activityID |
|---|---|---|
| **1** | 2016-01-01T23:00:00-FLR-001 | 2016-01-02T02:48:00-SEP-001 |
| **2** | 2016-01-01T23:00:00-FLR-001 | 2016-01-02T04:30:00-SEP-001 |
| **22** | 2017-04-18T19:15:00-FLR-001 | 2017-04-18T23:39:00-SEP-001 |
| **25** | 2017-07-14T01:07:00-FLR-001 | 2017-07-14T09:00:00-SEP-001 |
| **33** | 2017-09-04T20:15:00-FLR-001 | 2017-09-04T22:56:00-SEP-001 |

In [434…

```
print(only_sep_linked_events.shape)
```

```
(58, 2)
```

There is about 58 linked events from Solar flares which resulted in Solar energetic particles. Out of 349 events 58 of them is linked to SEP, but CME and SEP can happen together as one is the precursor to other. Because SEP can be produced without the CME, but these are rather shortlived and do not have any shockwave. Ön the other hand the 291 which are results of CME are shockdriven as the travel through space and are much longer compared to when this happen with only Solar Flare. Unfortunately there is not endTime for SEP so we can only compare them based on their eventTime and submissionTime. Let´s think of these as startTime and endTime. Following the logic from the Solar Flare dataset the endTime here could be the time the SEP event is submiited to be save the record.

In [435…
```
data_sep_df["submissionTime"] = pd.to_datetime(data_sep_df["submissionTime"]).dt
data_sep_df["eventTime"] = pd.to_datetime(data_sep_df["eventTime"]).dt.tz_locali

duration_sep = data_sep_df["submissionTime"] - data_sep_df["eventTime"]
duration_sep
```

Out[435…
```
0      0 days 01:57:00
1      0 days 00:11:00
2      0 days 12:22:00
3      0 days 00:13:00
4      0 days 00:27:00
             ...
65     0 days 13:15:00
66     0 days 13:16:00
67     0 days 13:14:00
68    -1 days +23:33:00
69    -1 days +23:31:00
Length: 70, dtype: timedelta64[ns]
```

In [436…
```
duration_sep = duration_sep.loc[duration_sep >= pd.Timedelta(days=0)]
duration_sep.head()
```

Out[436…
```
0   0 days 01:57:00
1   0 days 00:11:00
2   0 days 12:22:00
3   0 days 00:13:00
4   0 days 00:27:00
dtype: timedelta64[ns]
```

In [437…
```
avg_duration_sep = duration_sep.mean()
print(avg_duration_sep)
```

```
0 days 15:15:09.090909090
```

For 2 result there was a negative time. Which would be impossible to submit an event before it happens. So without those two the average duration of SEP is around 15h15m. That is before submission. Given that the submission can be saved after the equipment catches the radiation from these SEP. We can say that it takes at around average of 15h to have these SEP logged into system.

In [438…
```
only_sep_linked_events_id = only_sep_linked_events["flrID"]
only_sep_linked_events_id.head()
```

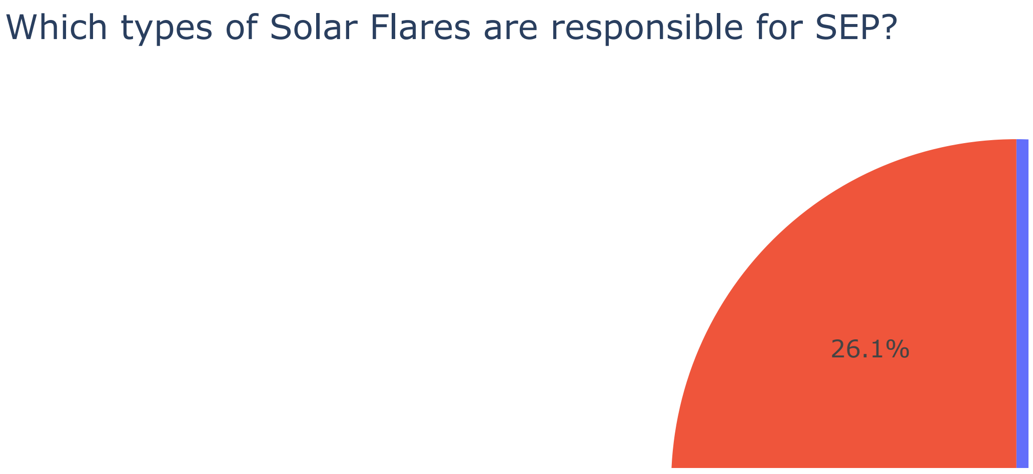
Out[438…
```
1     2016-01-01T23:00:00-FLR-001
2     2016-01-01T23:00:00-FLR-001
22    2017-04-18T19:15:00-FLR-001
25    2017-07-14T01:07:00-FLR-001
33    2017-09-04T20:15:00-FLR-001
Name: flrID, dtype: object
```

In [439…
```
parent_class_type_linked = data_flare_df.loc[data_flare_df["flrID"].isin(only_se
parent_class_type_linked
```

Out[439…
```
40     C
46     M
58     M
66     X
84     X
86     C
87     C
108    M
133    C
147    C
173    M
185    X
188    M
222    M
250    M
258    X
261    M
342    C
383    M
420    M
421    M
424    X
431    M
Name: parentClass, dtype: object
```

In [578…
```
#Create pie graph for SEP
parent_class_type_linked = data_flare_df.loc[data_flare_df["flrID"].isin(only_se
activity_counts_linked = parent_class_type_linked.value_counts()

fig = px.pie(parent_class_type_linked, values=activity_counts_linked.values, nam
fig.show()
```



Above is a pie graph that show % of what type of flare is responsible for SEP. Where the first largest piece of pie of 52.2% is type M which is also second strongest solar flare. Next is the strongest type of flare X type and at 26,1% is the C type which is the third strongest type of flare. M and X can cause issues with radio an Earth and other longer lasting effects. The C type is not that strong to cause any major issues such as the two types after.

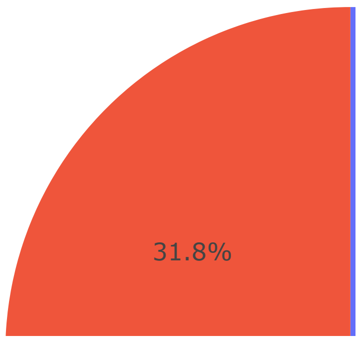
Let´s also check if these stronger classes of solar flare are also responsible fort CME.

In [579…

```
#Create pie graph for CME
only_cme_linked_events = flare_linked_events.loc[flare_linked_events["activityID
only_cme_linked_events_id = only_cme_linked_events["flrID"]
parent_class_type_linked = data_flare_df.loc[data_flare_df["flrID"].isin(only_cm
activity_counts_linked = parent_class_type_linked.value_counts()

fig = px.pie(parent_class_type_linked, values=activity_counts_linked.values, nam
fig.show()
```

Which classes of Solar Flares are responsible for CME?



In [442…

```
x_class_occurence = data_flare_df.loc[data_flare_df["parentClass"] == "X", "clas
x_class_occurence.head(15)
```

Out[442…

```
65     X2.2
66     X9.3
73     X1.3
84     X8.2
139    X1.5
185    X1.0
258    X1.3
270    X1.1
282    X2.2
298    X1.1
301    X1.1
312    X1.5
424    X1.0
Name: classType, dtype: object
```

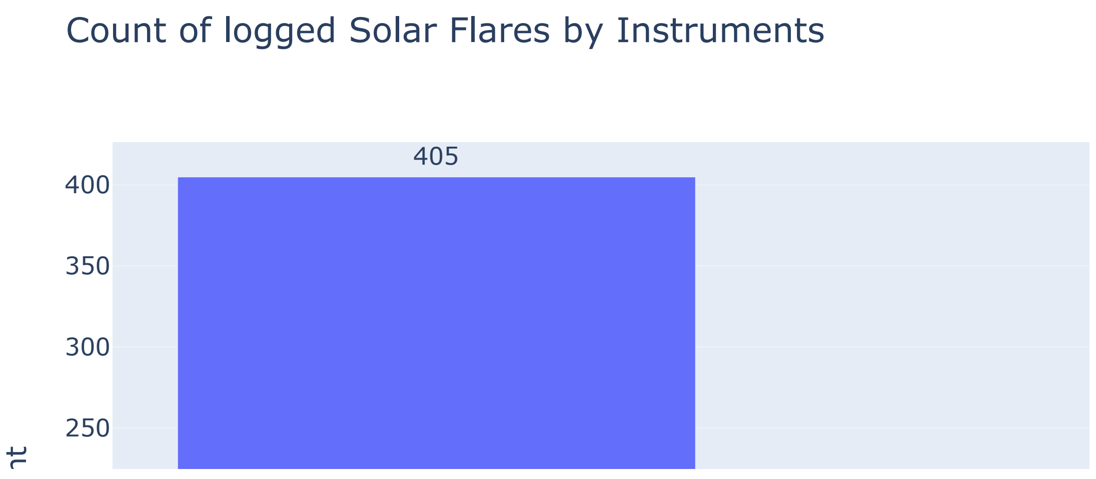
Here we can see what Solar Flares and which are responsible for CME and their %. The 3 top classes are C,M,B which are the stronger types with X being the strongest. But why is the strongest not in the top spot? That is because there are only 13 of them recorded in given data from 2013-2022 and all but 2 of these flares are on the lower end of 1-10 scale. Many conditions exists for there to be CME during Solar flare. That is why X class is not represented that much despite being the strongest class. (https://science.nasa.gov/sun/solar-storms-and-flares/)

For the last visulaization I just want to show which intrument was used the most for catching or recording given dataflares in the dataset.

In [580…

```python
instruments = data_flare_df["instrument_displayName"]
instruments = instruments.value_counts()

fig_bar = px.bar(instruments, x=instruments.index, y=instruments.values,
                 text=instruments, title="Count of logged Solar Flares by Instru
fig_bar.update_traces(texttemplate="%{text:.0f}", textposition="outside")
fig_bar.update_layout(uniformtext_minsize=8, uniformtext_mode="hide", xaxis_titl
fig_bar.show()
```



Here it is shown that most Solar flares were logged or captured by GOES-P which are types of weather monitoring instruments that detect solar flares. (https://iopscience.iop.org/article/10.1088/1742-6596/2543/1/012011/pdf)

Now the last step is simple loading the selected dataframes to locally hosted databse (PsotgresSQL). With psycopg2 connection with created DB called "sunlog" and 4 other tables wit Solar Flares, SEP and Linked events for both of these datasets. Here I will show how to create table through sql.get_schema() where we can get schema for given dataframe to create table if we do not want to create it by writing each query.

In [528…

```python
#Connection to DBS through psycog2
def connect_to_db(db="postgres"):
    try:
        conn = psycopg2.connect(database=db,
                                host="localhost",
                                user="postgres",
                                password="Tessina",
                                port="5432")
    except Exception as error:
        print("There has been an error: " + error)
    print("Connection successful!")
    return conn
```

In [ ]:
```
#In postgres SQL DB cannot be created in transaction -> Disable transaction with
conn = connect_to_db()
conn.set_isolation_level(psycopg2.extensions.ISOLATION_LEVEL_AUTOCOMMIT)
cur = conn.cursor()

new_db = "sun_log"
cur.execute(f"CREATE DATABASE {new_db};")
conn.close()
```

In [529…
```
conn = connect_to_db(new_db)
cur = conn.cursor()
```

```
Connection successful!
```

Unfortunately sql.get_schema supports on SQLAlchemy engine/connection or sqlite3. I want to show classical approach with creating a table and DB from scratch.

In [ ]:
```
ddl = pd.io.sql.get_schema(data_flare_df, "solar_flare", con=conn)
print(ddl)
```

In [531…
```
cur.execute("""CREATE TABLE IF NOT EXISTS solar_flare (
            flrID VARCHAR(255) PRIMARY KEY NOT NULL,
            catalog VARCHAR(255),
            beginTime TIMESTAMP,
            peakTime TIMESTAMP,
            endTime TIMESTAMP,
            classType VARCHAR(4),
            sourceLocation VARCHAR(10),
            activeRegionNum VARCHAR(15),
            note TEXT,
            submissionTime TIMESTAMP,
            versionId INTEGER,
            link VARCHAR(255),
            instrument_displayName VARCHAR(255),
            activityID BOOLEAN,
            parentClass VARCHAR(1),
            duration INT);
""")
```

In [ ]:
```
ddl2 = pd.io.sql.get_schema(data_sep_df, "solar_energy_particles", con=conn)
print(ddl2)
```

In [533…
```
cur.execute("""CREATE TABLE IF NOT EXISTS solar_energy_particles (
            sepID VARCHAR(255) PRIMARY KEY NOT NULL,
            eventTime TIMESTAMP,
            submissionTime TIMESTAMP,
            versionId INTEGER,
            link VARCHAR(255),
            instrument_displayName VARCHAR (255),
            activityID BOOLEAN);
""")
conn.commit()
```

In [534…

```
cur.execute("""CREATE TABLE IF NOT EXISTS solar_linked_events (
            sepID VARCHAR(255),
            activityID VARCHAR(255),
            FOREIGN KEY (sepID) REFERENCES solar_energy_particles(sepID),
            id INT PRIMARY KEY NOT NULL);
""")
conn.commit()
```

In [535…

```
cur.execute("""CREATE TABLE IF NOT EXISTS flare_linked_events (
            flrID VARCHAR(255),
            activityID VARCHAR(255),
            FOREIGN KEY (flrID) REFERENCES solar_flare(flrID),
            id INT PRIMARY KEY NOT NULL);
""")
conn.commit()
```

In [536…

```
def copy_from_stringio(conn, df, table):
    #Save dataframe to an in memory buffer

    if "instruments" in df.columns:
        df = df.drop(columns=["instruments"])
    if "linkedEvents" in df.columns:
        df = df.drop(columns=["linkedEvents"])

    buffer = StringIO()
    df.to_csv(buffer, index=False, header=True, sep=",", quoting=1)
    buffer.seek(0)

    cursor = conn.cursor()
    try:
        cursor.copy_expert(f"COPY {table} FROM stdin WITH CSV HEADER", buffer)
        conn.commit()
    except (Exception, psycopg2.DatabaseError) as error:
        print("Error: %s" % error)
        conn.rollback()
        cursor.close()
        return print("There has been an error when loading into database.")
    print("copy_from_stringio() done")
    cursor.close()
```

In [537…

```
data_flare_df["duration"] = pd.to_timedelta(data_flare_df["duration"], errors="c
data_flare_df["duration"] = data_flare_df["duration"].fillna(0).astype(int)
```

In [ ]:

```
copy_from_stringio(conn, data_flare_df, "solar_flare")
```

In [543…

```
copy_from_stringio(conn, data_sep_df, "solar_energy_particles")
```

```
copy_from_stringio() done
```

In [ ]:

```
#Creating id columns to be primary keys in DB
solar_linked_events["id"] = solar_linked_events.index
flare_linked_events["id"] = flare_linked_events.index
```

In [565…
```
copy_from_stringio(conn, flare_linked_events, "flare_linked_events")
```
```
copy_from_stringio() done
```

In [ ]:
```
copy_from_stringio(conn, solar_linked_events, "solar_linked_events")
```
```
copy_from_stringio() done
```

In [566…
```
cur.close()
```