analysis

June 12, 2024

1 Correlations and predictors in US Power outages

Name(s): Martin Hawks

Website Link: https://mawks12.github.io/power_outage_USA/

[ ]:
```
import pandas as pd
import numpy as np
from pathlib import Path
from scipy import stats
import plotly.express as px
pd.options.plotting.backend = 'plotly'
dataloc = Path('data')
data_raw = pd.read_excel(dataloc / 'outage.xlsx.xls')

# from dsc80_utils import * # Feel free to uncomment and use this.
```

1.1 Step 1: Introduction

I'm perticularly interested in the the number of weather related outaged over time, and if the effects of global warming can be seen in this dataset using the weather related outages as a proxy. I'm also interested in if there is a correlation between the population density of an area and things like outage duration and frequency. this is much harder to study since the data is grouped by state and not the locaiton where the outage occured.

1.2 Step 2: Data Cleaning and Exploratory Data Analysis

Format all of the data and modify time columns to encode all data in correct format for analysis.

[ ]:
```
# create a new dataframe with the selected data
formated_data = pd.DataFrame(data_raw.iloc[6:, 1:].to_numpy(), columns=data_raw.
↪iloc[4, 1:])

# drop rows with missing values in specific columns
temp_data = formated_data.dropna(subset=['OBS', 'OUTAGE.START.DATE', 'OUTAGE.
↪START.TIME', 'OUTAGE.RESTORATION.DATE', 'OUTAGE.RESTORATION.TIME'])

# convert selected columns to string type
```

```
cols_to_str = ['OUTAGE.START.DATE', 'OUTAGE.START.TIME', 'OUTAGE.RESTORATION.
↪DATE', 'OUTAGE.RESTORATION.TIME']
for col in cols_to_str:
    temp_data.loc[:, col] = temp_data.loc[:, col].astype(str)

# create new columns for outage start and end timestamps
temp_data.loc[:, 'outageStart'] = pd.to_datetime(temp_data['OUTAGE.START.DATE']␣
↪+ ' ' + temp_data['OUTAGE.START.TIME'])
temp_data.loc[:, 'outageEnd'] = pd.to_datetime(temp_data['OUTAGE.RESTORATION.
↪DATE'] + ' ' + temp_data['OUTAGE.RESTORATION.TIME'])

# select specific columns from temp_data and merge with formated_data
temp_data = temp_data[['OBS', 'outageStart', 'outageEnd']]
formated_data = formated_data.merge(temp_data, left_on='OBS', right_on='OBS',␣
↪how='left')

# infer the data types of the columns in formated_data
formated_data = formated_data.infer_objects()

# calculate the percentage of utility real GSP relative to total real GSP
formated_data['UTIL_REL_PERCEN'] = formated_data['UTIL.REALGSP'] /␣
↪formated_data['TOTAL.REALGSP'] * 100
```

```
/Users/martinhawks/miniconda3/envs/dsc80/lib/python3.8/site-
packages/pandas/core/indexing.py:1773: 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

/Users/martinhawks/miniconda3/envs/dsc80/lib/python3.8/site-
packages/pandas/core/indexing.py:1667: 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
```

Exploration of outages by state. The first is an absolute value, while the seccond is normalized against the number of people in the state.

```
[ ]: state = formated_data.groupby('POSTAL.CODE').count().reset_index()
     px.bar(state, x='POSTAL.CODE', y='OBS', title='State Outages')
```

```
[ ]: capita = formated_data.groupby('POSTAL.CODE')['OBS'].count()
     pop = formated_data.groupby('POSTAL.CODE')['POPULATION'].mean()
     capita = capita / pop  # calculate the number of outages per capita
```

```
[ ]: px.bar(capita, x=capita.index, y=[0], title='Outages per Capita')
```

```
[ ]: capita.median()
```

```
[ ]: 3.92684105058557e-06
```

```
[ ]: zcap = pd.Series(index=capita.index, data=stats.zscore(capita))
     delz = zcap.loc['DE']
     delz  # compare the number of outages per capita in Delaware to the national␣
      ↪average
```

```
[ ]: 5.723815787163536
```

Deleware is very interesting here, as it has more than 6 times the mean, and more than twice the number per capita as the seccond highest value. Why might this be?

```
[ ]: # Look at the number of outages in Delaware vs the national average
     formated_data[formated_data['POSTAL.CODE'] == 'DE'].shape[0],␣
      ↪formated_data[formated_data['POSTAL.CODE'] == 'DE']['POPULATION'].mean()
```

```
[ ]: (41, 919231.8780487805)
```

The number of outages seems fairly standard comparted to the average values of the total numbers above - perhaps deleware has a significantly lower population than other states

```
[ ]: px.bar(formated_data.groupby('POSTAL.CODE').mean().reset_index(), x='POSTAL.
      ↪CODE', y='POPULATION')
```

```
[ ]: # Overall average population
     formated_data.groupby('POSTAL.CODE')['POPULATION'].mean().mean()
```

```
[ ]: 6139512.208006512
```

Deleware does indeed seem to have a very low population, rather than a very high number of outages. Still, the very high outages per capita is odd to see - perhaps it is to do with the GSP?

Distribution of the proportion of gdp each state is responsible for in the US. This value might have some correlation with number of outages or outage response time, so we want to understand how it works before we start our analysis.

```
[ ]: # Calculate the real GSP for each state and plot the average real GSP by state
     formated_data['REALGSP'] = formated_data['PC.REALGSP.REL'] *␣
      ↪formated_data['POPULATION']
```

```
px.bar(formated_data.groupby('POSTAL.CODE')[['REALGSP']].mean().reset_index(),␣
↪x='POSTAL.CODE', y='REALGSP')
```

There seem to be a large number of peaks in states with higher population, Likely it would correlate with frequency as it is also directly correlated with population density and population.

[ ]:
```
px.scatter(formated_data, x='REALGSP', y='OUTAGE.DURATION')
```

Lots of the durations here are very low, is this because the data is not accurate or because the outages are very short? Compare with something like peak demand loss to see if there is a corelation between the two, which would be expected.

[ ]:
```
px.scatter(formated_data, x='OUTAGE.DURATION', y='DEMAND.LOSS.MW')
```

Indeed, an exponental decay relationship is present (For linear model, this may be a useful feature). Lets drop the columns with no peak demand loss and plot the above graph again.

[ ]:
```
px.scatter(formated_data[formated_data['DEMAND.LOSS.MW'] > 0], x='OUTAGE.
↪DURATION', y='DEMAND.LOSS.MW')
```

Still looks fairly similar. There are also a number of very high values, which are likely the points where total demand loss was reported instead of peak demand loss. This could be an issue for training a model, as it would unfairly weight these points. Unfortunately, there is no way to tell which points are which.

Number of occurences of each of the given outage causes

[ ]:
```
# Groupy all of the data by cause category to see what the new distribution␣
↪looks like
px.bar(formated_data.groupby('CAUSE.CATEGORY').count().reset_index(), x='CAUSE.
↪CATEGORY', y='OBS', title='Number of Outages by Cause Category')
```

Weather seems to be the largest cause, followed interestingly by intentional attack. Why might Intentional attacks be such a common cause of outages? (Likely the answer is not in our dataset, so we will continue and instead look at weather patterns)

Exploration of proportions of outages attibuted to weather events year over year

[ ]:
```
# Get the preportion of all of the outages that are caused by severe weather
weather_year = formated_data[formated_data['CAUSE.CATEGORY'] == 'severe␣
↪weather'].groupby('YEAR').count()['OBS'] / formated_data.groupby('YEAR').
↪count()['OBS']
px.scatter(weather_year, x=weather_year.index, y='OBS', title='Severe Weather␣
↪Outages by Year').show()
```

the 2001 data point seems to be a bit of an outlier. Why might it have had such a low proportion of weather related outage?

[ ]:
```
two001 = formated_data[formated_data['YEAR'] == 2001]
px.bar(two001.groupby('CAUSE.CATEGORY').count().reset_index(), x='CAUSE.
↪CATEGORY', y='OBS', title='2001 Outages by Cause Category')
```

Lots of system Operability disruptions - Likely this is wht the weather preportion is so low. We will check if there is more detailed informaiton for this category

[ ]:
```
# Check how many values for detailed categories exist
two001_details = two001[two001['CAUSE.CATEGORY'] == 'system operability␣
↪disruption']
(~two001_details['CAUSE.CATEGORY.DETAIL'].isna()).sum()
```

[ ]: 0

Unfortunately, none of the causes in this area seem to have detailed information, so there is no ability to identify why this might have been an issue

Absolute number of outages occuring year over year

[ ]:
```
# Look at yearly outages
yearly = formated_data.groupby('YEAR').count().reset_index()
px.scatter(yearly, x='YEAR', y='OBS', title='Yearly Outages')
```

There seem to be some clustering of shorter outage durations

[ ]:
```
# Get all of the short outages and make a histogram of them to see the␣
↪distribution
filtered = formated_data[formated_data['OUTAGE.DURATION'] < 50]
filtered['OUTAGE.DURATION'] = pd.cut(filtered['OUTAGE.DURATION'].dropna().
↪astype(int), 100)
filtered = filtered.groupby('OUTAGE.DURATION').count().reset_index()
filtered['OUTAGE.DURATION'] = filtered['OUTAGE.DURATION'].astype(str)
px.bar(filtered, x='OUTAGE.DURATION', y='OBS')
```

```
/var/folders/v7/nxggzv_j5s936v9rvl85gh2w0000gn/T/ipykernel_3632/421583513.py:3:
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
```

Lots of these values seem to be 1 or 0. Look at all of these and see what they have in common

[ ]:
```
# See if any comminalities can be found in the non zero/1 outages
low_durs = formated_data[formated_data['OUTAGE.DURATION'] <= 1]
low_durs.pivot_table(index='OUTAGE.DURATION', columns='ANOMALY.LEVEL',␣
↪values='DEMAND.LOSS.MW')
```

[ ]:
```
ANOMALY.LEVEL    -1.3    -1.1  -1.0  -0.9  -0.8  -0.7  -0.6         -0.5  \
OUTAGE.DURATION
```

0.0	0.0	1040.0	0.0	1.4	0.0	0.0	0.0	0.000000
1.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	728.333333

ANOMALY.LEVEL	-0.4	-0.3	-0.1	0.1	0.3	0.6	0.8	1.6	2.3
OUTAGE.DURATION									
0.0	0.2	0.000000	0.0	NaN	0.0	NaN	NaN	0.0	NaN
1.0	157.5	18.333333	NaN	0.0	0.0	12.0	0.0	NaN	0.0

Some of these points dont seem to make any sense - how can an outage lasting 0 minutes cause a peak demand loss of 1040 MW? For most analysis, and prediction, it may make sense to drop these values

[ ]:
```python
# See if any comminalities can be found in the outages with no length
filtered = formated_data[formated_data['OUTAGE.DURATION'] > 1]
filtered['OUTAGE.DURATION'] = pd.cut(filtered['OUTAGE.DURATION'].dropna().
 ↪astype(int), 100)
filtered.pivot_table(index='OUTAGE.DURATION', columns='ANOMALY.LEVEL',␣
 ↪values='DEMAND.LOSS.MW')
```

```
/var/folders/v7/nxggzv_j5s936v9rvl85gh2w0000gn/T/ipykernel_3632/2783385590.py:3:
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
```

[ ]:

ANOMALY.LEVEL	-1.6	-1.5	-1.4	-1.3	-1.2	-1.1 \
OUTAGE.DURATION						
(-106.651, 1088.51]	NaN	258.0	211.666667	534.0	280.333333	378.333333
(1088.51, 2175.02]	NaN	NaN	464.000000	180.0	NaN	NaN
(2175.02, 3261.53]	NaN	240.0	331.000000	NaN	NaN	NaN
(3261.53, 4348.04]	NaN	NaN	130.000000	79.0	NaN	NaN
(4348.04, 5434.55]	NaN	NaN	115.500000	NaN	NaN	NaN
(5434.55, 6521.06]	NaN	NaN	NaN	NaN	340.000000	NaN
(6521.06, 7607.57]	375.0	NaN	NaN	NaN	NaN	NaN
(7607.57, 8694.08]	NaN	NaN	NaN	NaN	NaN	NaN
(8694.08, 9780.59]	NaN	NaN	NaN	NaN	NaN	NaN
(9780.59, 10867.1]	NaN	NaN	NaN	NaN	NaN	NaN
(10867.1, 11953.61]	NaN	NaN	NaN	NaN	NaN	NaN
(11953.61, 13040.12]	NaN	NaN	NaN	NaN	NaN	NaN
(13040.12, 14126.63]	NaN	NaN	NaN	500.0	NaN	NaN
(14126.63, 15213.14]	NaN	NaN	500.000000	NaN	NaN	NaN
(15213.14, 16299.65]	NaN	NaN	NaN	NaN	NaN	NaN
(16299.65, 17386.16]	NaN	NaN	NaN	NaN	NaN	NaN

```
(17386.16, 18472.67]    NaN    NaN  300.000000    NaN         NaN         NaN
(18472.67, 19559.18]    NaN    NaN         NaN    NaN         NaN         NaN
(19559.18, 20645.69]    NaN    NaN         NaN    NaN         NaN         NaN
(20645.69, 21732.2]     NaN    NaN         NaN    NaN         NaN         NaN
(22818.71, 23905.22]    NaN    NaN         NaN    NaN         NaN         NaN
(23905.22, 24991.73]    NaN    NaN         NaN    NaN         NaN         NaN
(24991.73, 26078.24]    NaN    NaN         NaN    NaN         NaN         NaN
(27164.75, 28251.26]    NaN    NaN         NaN    NaN         NaN         NaN
(45635.42, 46721.93]    NaN    NaN         NaN    NaN         NaN         NaN
(48894.95, 49981.46]    NaN    NaN         NaN    NaN         NaN         NaN
(59760.05, 60846.56]    NaN    NaN         NaN    NaN         NaN         NaN
(78230.72, 79317.23]    NaN    NaN         NaN    NaN         NaN         NaN

ANOMALY.LEVEL             -1.0        -0.9  -0.8        -0.7  …        1.0  \
OUTAGE.DURATION                                               …
(-106.651, 1088.51]     595.75  303.846154   0.0  157.000000  …  189.333333
(1088.51, 2175.02]     4000.00  300.000000   NaN  475.333333  …         NaN
(2175.02, 3261.53]         NaN  240.000000   NaN  600.000000  …         NaN
(3261.53, 4348.04]      176.50  400.000000   NaN  177.500000  …         NaN
(4348.04, 5434.55]       91.00         NaN   NaN  200.000000  …         NaN
(5434.55, 6521.06]         NaN  540.000000   NaN  637.500000  …         NaN
(6521.06, 7607.57]         NaN         NaN   NaN   91.000000  …         NaN
(7607.57, 8694.08]         NaN         NaN   NaN  800.000000  …         NaN
(8694.08, 9780.59]        0.00         NaN   NaN         NaN  …         NaN
(9780.59, 10867.1]         NaN         NaN   NaN  200.000000  …         NaN
(10867.1, 11953.61]        NaN  125.000000   NaN  240.000000  …         NaN
(11953.61, 13040.12]       NaN         NaN   NaN  506.000000  …         NaN
(13040.12, 14126.63]       NaN         NaN   NaN         NaN  …         NaN
(14126.63, 15213.14]       NaN         NaN   NaN  348.500000  …         NaN
(15213.14, 16299.65]       NaN         NaN   NaN         NaN  …         NaN
(16299.65, 17386.16]       NaN         NaN   NaN         NaN  …         NaN
(17386.16, 18472.67]       NaN         NaN   NaN         NaN  …         NaN
(18472.67, 19559.18]       NaN         NaN   NaN         NaN  …         NaN
(19559.18, 20645.69]       NaN         NaN   NaN         NaN  …         NaN
(20645.69, 21732.2]        NaN         NaN   NaN         NaN  …         NaN
(22818.71, 23905.22]       NaN         NaN   NaN         NaN  …         NaN
(23905.22, 24991.73]       NaN         NaN   NaN         NaN  …         NaN
(24991.73, 26078.24]       NaN         NaN   NaN         NaN  …         NaN
(27164.75, 28251.26]       NaN         NaN   NaN         NaN  …         NaN
(45635.42, 46721.93]       NaN         NaN   NaN         NaN  …         NaN
(48894.95, 49981.46]       NaN         NaN   NaN         NaN  …         NaN
(59760.05, 60846.56]       NaN         NaN   NaN         NaN  …         NaN
(78230.72, 79317.23]       NaN         NaN   NaN         NaN  …         NaN

ANOMALY.LEVEL                1.1    1.2    1.3    1.4    1.6   1.7    2.0  \
OUTAGE.DURATION
(-106.651, 1088.51]   177.142857  283.0  350.0    NaN  200.0  75.0  4188.5
```

(1088.51, 2175.02]	0.000000	250.0	2650.0	NaN	0.0	NaN	NaN
(2175.02, 3261.53]	59.500000	NaN	NaN	1200.0	NaN	NaN	NaN
(3261.53, 4348.04]	NaN	NaN	NaN	NaN	0.0	NaN	NaN
(4348.04, 5434.55]	NaN	270.0	NaN	NaN	NaN	NaN	NaN
(5434.55, 6521.06]	NaN	NaN	NaN	NaN	NaN	NaN	NaN
(6521.06, 7607.57]	180.000000	NaN	NaN	NaN	NaN	NaN	NaN
(7607.57, 8694.08]	NaN	NaN	NaN	NaN	NaN	NaN	NaN
(8694.08, 9780.59]	NaN	NaN	NaN	250.0	NaN	NaN	NaN
(9780.59, 10867.1]	NaN	NaN	NaN	NaN	NaN	NaN	NaN
(10867.1, 11953.61]	NaN	NaN	NaN	NaN	NaN	NaN	NaN
(11953.61, 13040.12]	NaN	NaN	NaN	NaN	NaN	NaN	NaN
(13040.12, 14126.63]	NaN	NaN	NaN	NaN	NaN	NaN	NaN
(14126.63, 15213.14]	NaN	NaN	290.0	NaN	NaN	NaN	NaN
(15213.14, 16299.65]	NaN	NaN	NaN	NaN	NaN	NaN	NaN
(16299.65, 17386.16]	NaN	NaN	NaN	NaN	NaN	NaN	NaN
(17386.16, 18472.67]	NaN	NaN	NaN	NaN	NaN	NaN	NaN
(18472.67, 19559.18]	NaN	NaN	NaN	NaN	NaN	NaN	NaN
(19559.18, 20645.69]	NaN	NaN	NaN	NaN	NaN	NaN	NaN
(20645.69, 21732.2]	NaN	NaN	NaN	NaN	NaN	NaN	NaN
(22818.71, 23905.22]	NaN	NaN	NaN	NaN	NaN	NaN	NaN
(23905.22, 24991.73]	NaN	NaN	NaN	NaN	NaN	NaN	NaN
(24991.73, 26078.24]	NaN	NaN	NaN	NaN	NaN	NaN	NaN
(27164.75, 28251.26]	NaN	NaN	NaN	NaN	NaN	NaN	NaN
(45635.42, 46721.93]	NaN	NaN	NaN	NaN	NaN	NaN	NaN
(48894.95, 49981.46]	NaN	NaN	NaN	NaN	0.0	NaN	NaN
(59760.05, 60846.56]	NaN	NaN	NaN	NaN	NaN	NaN	NaN
(78230.72, 79317.23]	NaN	NaN	NaN	NaN	NaN	NaN	NaN

ANOMALY.LEVEL	2.2	2.3
OUTAGE.DURATION		
(-106.651, 1088.51]	0.0	96.0
(1088.51, 2175.02]	0.0	0.0
(2175.02, 3261.53]	4.0	NaN
(3261.53, 4348.04]	NaN	NaN
(4348.04, 5434.55]	NaN	NaN
(5434.55, 6521.06]	NaN	NaN
(6521.06, 7607.57]	NaN	NaN
(7607.57, 8694.08]	NaN	NaN
(8694.08, 9780.59]	NaN	NaN
(9780.59, 10867.1]	NaN	NaN
(10867.1, 11953.61]	NaN	NaN
(11953.61, 13040.12]	NaN	NaN
(13040.12, 14126.63]	NaN	NaN
(14126.63, 15213.14]	NaN	NaN
(15213.14, 16299.65]	NaN	NaN
(16299.65, 17386.16]	NaN	NaN
(17386.16, 18472.67]	NaN	NaN

```
(18472.67, 19559.18]    NaN    NaN
(19559.18, 20645.69]    NaN    NaN
(20645.69, 21732.2]     NaN    NaN
(22818.71, 23905.22]    NaN    NaN
(23905.22, 24991.73]    NaN    NaN
(24991.73, 26078.24]    NaN    NaN
(27164.75, 28251.26]    NaN    NaN
(45635.42, 46721.93]    NaN    NaN
(48894.95, 49981.46]    NaN    NaN
(59760.05, 60846.56]    NaN    NaN
(78230.72, 79317.23]    NaN    NaN

[28 rows x 36 columns]
```

[ ]:
```python
pivoter = formated_data.copy()
pivoter['OUTAGE.DURATION'] = pd.qcut(formated_data['OUTAGE.DURATION'], 10,␣
 ↪labels=False)
pivot2 = formated_data.pivot_table(index='OUTAGE.DURATION', columns='NERC.
 ↪REGION', values='TOTAL.PRICE', aggfunc='mean')
pivot2
```

[ ]:
```
NERC.REGION      ECAR  FRCC  FRCC, SERC  HECO  HI   MRO       NPCC  PR  \
OUTAGE.DURATION
0.0               NaN   NaN         NaN   NaN NaN  9.28  14.726154 NaN
1.0               NaN   NaN         NaN   NaN NaN  9.28  15.001538 NaN
2.0               NaN   NaN         NaN   NaN NaN   NaN        NaN NaN
3.0               NaN   NaN         NaN   NaN NaN   NaN        NaN NaN
4.0               NaN   NaN         NaN   NaN NaN  6.20        NaN NaN
…                …   …          …   …  ..   …          …  ..
49320.0           NaN   NaN         NaN   NaN NaN   NaN        NaN NaN
49427.0           NaN   NaN         NaN   NaN NaN   NaN        NaN NaN
60480.0           NaN   NaN         NaN   NaN NaN   NaN  17.810000 NaN
78377.0           NaN   NaN         NaN   NaN NaN   NaN        NaN NaN
108653.0          NaN   NaN         NaN   NaN NaN   NaN        NaN NaN

NERC.REGION            RFC      SERC   SPP   TRE       WECC
OUTAGE.DURATION
0.0              10.987187  9.150000  9.56  8.56   7.220769
1.0              10.768421  9.431667   NaN  9.04   9.304444
2.0              11.310000  8.800000   NaN   NaN   8.837500
3.0               8.880000       NaN  6.77   NaN  13.320000
4.0                    NaN       NaN   NaN   NaN   8.350000
…                     …       …    …    …        …
49320.0                NaN       NaN   NaN   NaN   7.910000
49427.0                NaN       NaN   NaN   NaN  14.340000
60480.0                NaN       NaN   NaN   NaN        NaN
78377.0           8.840000       NaN   NaN   NaN        NaN
```

```
108653.0          10.280000        NaN    NaN    NaN         NaN

[847 rows x 13 columns]
```

1.3 Step 3: Assessment of Missingness

To get a sense of missingess, make a dataframe with all the datapoints containing a missing value

[ ]:
```python
def missing_points(df: pd.DataFrame):
    hasna = np.repeat(False, df.shape[0])
    for col in df.columns:
        hasna = (hasna | df[col].isna())
    return df.loc[hasna]

# Define a function that returns the columns with any missing values
missing_all = missing_points(formated_data)
```

Since the hurricane name column only applies to very few data points, and is therefore nan for most (MD), drop that one to see which datapoints might contain some form of unintentional missingess

[ ]:
```python
no_hur_missing = missing_points(formated_data.drop(columns=['HURRICANE.NAMES']))
no_hur_missing.shape[0]
```

[ ]:
```
1039
```

Find all of the columns that contain any missing data

[ ]:
```python
hasna = np.repeat(False, formated_data.shape[1])
index = 0
for col in formated_data.columns: # Check for missing values in each column
    if np.any(formated_data[col].isna()):
        hasna[index] = True
    index += 1
has_missing = formated_data.loc[:, hasna]
has_missing.columns
```

[ ]:
```
Index(['MONTH', 'CLIMATE.REGION', 'ANOMALY.LEVEL', 'CLIMATE.CATEGORY',
       'OUTAGE.START.DATE', 'OUTAGE.START.TIME', 'OUTAGE.RESTORATION.DATE',
       'OUTAGE.RESTORATION.TIME', 'CAUSE.CATEGORY.DETAIL', 'HURRICANE.NAMES',
       'OUTAGE.DURATION', 'DEMAND.LOSS.MW', 'CUSTOMERS.AFFECTED', 'RES.PRICE',
       'COM.PRICE', 'IND.PRICE', 'TOTAL.PRICE', 'RES.SALES', 'COM.SALES',
       'IND.SALES', 'TOTAL.SALES', 'RES.PERCEN', 'COM.PERCEN', 'IND.PERCEN',
       'POPDEN_UC', 'POPDEN_RURAL', 'outageStart', 'outageEnd'],
      dtype='object', name=4)
```

[ ]:
```python
nomissing = formated_data.loc[:, ~hasna] # also check the inverse
nomissing.columns
```

```
[ ]: Index(['OBS', 'YEAR', 'U.S._STATE', 'POSTAL.CODE', 'NERC.REGION',
        'CAUSE.CATEGORY', 'RES.CUSTOMERS', 'COM.CUSTOMERS', 'IND.CUSTOMERS',
        'TOTAL.CUSTOMERS', 'RES.CUST.PCT', 'COM.CUST.PCT', 'IND.CUST.PCT',
        'PC.REALGSP.STATE', 'PC.REALGSP.USA', 'PC.REALGSP.REL',
        'PC.REALGSP.CHANGE', 'UTIL.REALGSP', 'TOTAL.REALGSP', 'UTIL.CONTRI',
        'PI.UTIL.OFUSA', 'POPULATION', 'POPPCT_URBAN', 'POPPCT_UC',
        'POPDEN_URBAN', 'AREAPCT_URBAN', 'AREAPCT_UC', 'PCT_LAND',
        'PCT_WATER_TOT', 'PCT_WATER_INLAND', 'UTIL_REL_PERCEN', 'REALGSP'],
       dtype='object', name=4)
```

Perhaps there is some kind of correlation with the states and the missing values, if some states store data differently

```
[ ]: # make a table of the missing values by state
formated_data['DEMAND.LOSS.MW'].isna()
formated_data['POSTAL.CODE']
postal_loss = formated_data[['POSTAL.CODE', 'DEMAND.LOSS.MW']]
postal_loss.loc[:, 'Missing'] = postal_loss['DEMAND.LOSS.MW'].isna()
postal_loss = postal_loss.groupby('POSTAL.CODE').sum()
postal_loss
```

```
/Users/martinhawks/miniconda3/envs/dsc80/lib/python3.8/site-
packages/pandas/core/indexing.py:1667: 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
```

```
[ ]: 4            DEMAND.LOSS.MW  Missing
POSTAL.CODE
AK                     35.0        0
AL                    583.0        4
AR                   1499.0       15
AZ                  12457.0       18
CA                 105480.0       52
CO                   1701.0        4
CT                    255.0       11
DC                   3840.0        7
DE                     95.0       18
FL                  32183.0        5
GA                   7916.0        1
HI                   2680.0        0
IA                   1350.0        4
ID                    932.0        1
```

```
IL                           1928.0       37
IN                           6332.0       22
KS                           1250.0        4
KY                           1035.0        8
LA                           3816.0       23
MA                          23922.0        8
MD                          11440.0       26
ME                            305.0       11
MI                          35524.0       44
MN                            345.0       10
MO                           1721.0        6
MS                             30.0        2
MT                              0.0        3
NC                          24688.0       13
ND                           1805.0        0
NE                           1543.0        0
NH                              0.0        5
NJ                           2521.0       17
NM                           1040.0        5
NV                             56.0        3
NY                          43627.0       37
OH                          23261.0       21
OK                           1785.0       14
OR                            604.0       17
PA                           4280.0       38
SC                          11898.0        1
SD                            457.0        0
TN                           4630.0       21
TX                          33125.0       67
UT                           3907.0       20
VA                          15639.0        7
VT                              0.0        4
WA                           8782.0       56
WI                           1449.0       11
WV                            724.0        2
WY                            107.0        2
```

Is there perhaps an association between the wealth of the state and the quality of the data? This could be a potential source of bias in the data - if the data is missing in states with lower GDP, the data might have a bias when predicting on certain states.

[ ]:
```
# Merge the 'postal_loss' DataFrame with the mean of 'TOTAL.REALGSP' grouped by␣
↪'POSTAL.CODE'
gsp_postal_missing = postal_loss.merge(formated_data.groupby('POSTAL.
↪CODE')['TOTAL.REALGSP'].mean(), left_index=True, right_index=True)
```

```python
# Add a new column 'Totals' to 'gsp_postal_missing' DataFrame, which represents␣
↪the count of observations per 'POSTAL.CODE'
gsp_postal_missing['Totals'] = formated_data.groupby('POSTAL.CODE').
↪count()['OBS']

# Calculate the proportion of missing values per 'POSTAL.CODE' and store it in␣
↪the 'Missing_prop' column of 'gsp_postal_missing' DataFrame
gsp_postal_missing['Missing_prop'] = gsp_postal_missing['Missing'] /␣
↪gsp_postal_missing['Totals']

# Create a scatter plot using the 'px.scatter' function from the Plotly Express␣
↪library
# The x-axis represents the mean of 'TOTAL.REALGSP' per 'POSTAL.CODE'
# The y-axis represents the proportion of missing values per 'POSTAL.CODE'
# The title of the plot is set to 'Missing Demand Loss by Real GSP'
px.scatter(gsp_postal_missing, x='TOTAL.REALGSP', y='Missing_prop',␣
↪title='Missing Demand Loss by Real GSP')
```

Perhaps a weak correlation? Run a test to see if the correlation is significant.

[ ]:
```python
stats.pearsonr(gsp_postal_missing['TOTAL.REALGSP'],␣
↪gsp_postal_missing['Missing_prop'])
```

[ ]:
```
PearsonRResult(statistic=0.01487506360937882, pvalue=0.9183377304861762)
```

Clearly, there is no correlation between missingness of the peak demand loss and the total GSP of each state.

test against NERC region as well

[ ]:
```python
# Create a copy of the 'formated_data' DataFrame
missing_nerc = formated_data.copy()

# Calculate the missing values per NERC region
missing_nerc.loc[:, 'NERC'] = missing_nerc['DEMAND.LOSS.MW'].isna()
missing_nerc = missing_nerc.groupby('NERC.REGION').sum()

# Calculate the proportion of missing values per NERC region
missing_nerc['Missing_prop'] = missing_nerc['NERC'] / formated_data.
↪groupby('NERC.REGION').count()['OBS']

# Merge the missing values DataFrame with the count of observations per NERC␣
↪region
missing_nerc = missing_nerc.merge(formated_data.groupby('NERC.REGION')['OBS'].
↪count(), left_index=True, right_index=True)

# Select the columns 'Missing_prop' and 'OBS_x' for display
missing_nerc[['Missing_prop', 'OBS_x']]
```

[ ]:

```
            Missing_prop    OBS_x
NERC.REGION
ASCC            0.000000     1534
ECAR            0.088235    14024
FRCC            0.090909    45540
FRCC, SERC      1.000000     1047
HECO            0.000000     4557
HI              0.000000     1516
MRO             0.565217    19137
NPCC            0.513333   180347
PR              0.000000     1517
RFC             0.548926   222443
SERC            0.400000   163380
SPP             0.641791    78424
TRE             0.549550    25821
WECC            0.394678   418058
```

This is much more interesting - some regions have no missing data, while others have a lot. This seems a lot more like a correlation. Also, there seems to be a data point which exists in two regions - why might this be?

[ ]:

```python
def tvd(s1, s2):  # Total Variation Distance function
    return np.abs(s1 - s2).sum() / 2

test = formated_data[['DEMAND.LOSS.MW', 'NERC.REGION', 'OBS']]
N = 10_000
tvds = np.repeat(0.0, N)
for i in range(N):  # Shuffle the missing values and calculate the TVD for each␣
↪iteration
    test['Shuffled'] = np.random.permutation(test['DEMAND.LOSS.MW'].isna())
    grouped = test.groupby('NERC.REGION').sum()
    grouped.loc[:, 'Missing_prop'] = grouped['Shuffled'] / test.groupby('NERC.
↪REGION').count()['OBS']
    grouped.loc[:, 'non_missing_prop'] = 1 - grouped['Missing_prop']
    tvds[i] = tvd(grouped['Missing_prop'], grouped['non_missing_prop'])

obs = tvd(missing_nerc['Missing_prop'], 1 - missing_nerc['Missing_prop'])
p = np.mean(tvds >= obs)  # Calculate the p-value
p
```

```
/var/folders/v7/nxggzv_j5s936v9rvl85gh2w0000gn/T/ipykernel_3632/1158676869.py:8:
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

[ ]: 0.0

Given this P value, it would seem that the missingess of the Peak loss column is very much MAR dependent on the region that the outage occured in. Keep note, as this could be a source of bias in the data.

1.4 Step 4: Hypothesis Testing

Analysis of outages caused by weather events vs. outages caused by non-weather events. We wonder if the development of better technologies and more stable systems have made weather a smaller problem for power outages H_0: There is no correlation betweeen time and the preportion of weather related power outages
H_1: Weather related outages have decreased over time

[ ]:
```
# make series with normalized values for weather outages
weather = (formated_data[formated_data['CAUSE.CATEGORY'] == 'severe weather'].
 ↪groupby('YEAR').count().sort_values('OBS', ascending=False) / formated_data.
 ↪groupby('YEAR').count())
px.scatter(weather, x=weather.index, y='OBS', title='Severe Weather Outages by␣
 ↪Year')
```

[ ]:
```
no_drop = stats.pearsonr(weather.index, weather['OBS'])
weather.drop(2001, inplace=True)
drop = stats.pearsonr(weather.index, weather['OBS'])
no_drop, drop
```

[ ]:
```
(PearsonRResult(statistic=-0.40759750516416526, pvalue=0.10437683084481637),
 PearsonRResult(statistic=-0.7561000778006577, pvalue=0.0007019393734256857))
```

Since we cannot just drop a data point without any justification, we will fail to reject our null hypothesis, however, the 2001 outlier is still interesting.

Dropping the 2001 data point has a massive effect on the r value and the p value of the r correlation. the r value becomes very strongly negetive from weakly negative. The p value becomes well below the 0.05 threshold from 0.5. We already found that there are a lot of system Operability disrutions in this year, why might that value be so high?

Correlation between peak demand loss and total cost of electricity in the area - It would be interesting to see if grids that had more income where able to prevent or midigate outages so avoid inconvinienceing people using the grid

H_1: Higher costs of electricity will result in lower peak demand loss during an outage
H_0: Peak demand loss is completely independent of the cost of electricity in a region

[ ]:
```
# Start by getting the relevant columns, and dropping nan vals
temp_data = formated_data[['TOTAL.PRICE', 'DEMAND.LOSS.MW', 'OUTAGE.DURATION']].
 ↪dropna()
```

```
temp_data['OUTAGE.DURATION'] = temp_data['OUTAGE.DURATION'].astype(float)
temp_data.loc[:, 'DEMAND.LOSS.MW'] = temp_data['DEMAND.LOSS.MW'].astype(float)
px.scatter(temp_data, x='TOTAL.PRICE', y='DEMAND.LOSS.MW', title='Cost vs␣
↪Demand Loss')
```

[ ]:
```
stats.pearsonr(temp_data['TOTAL.PRICE'], temp_data['DEMAND.LOSS.MW'])
```

[ ]:
```
PearsonRResult(statistic=0.04455305349818451, pvalue=0.20924488838558938)
```

Again, not enough correlation to reject our null Hypothesis - however, this test may not be entirely accurate, as one issue with the data is how the Peak demand loss values are generated. According to the data documentation, some of these values are not peak demand loss but rather total demand loss, and with no way to tell them apart, we will be unable to preform a more rigorous test.

1.5 Step 5: Framing a Prediction Problem

We will be training a model to predict the outage duration of a power outage, idealy to make a prediction after the power is lost. Thus, features like customers affected will be accesible to the model, but features like peak demand loss will not be, since that feature is dependent on the length of the outage

[ ]:
```
px.bar(formated_data.groupby('CAUSE.CATEGORY').mean().reset_index(), x='CAUSE.
↪CATEGORY', y='OUTAGE.DURATION', title='Outages by Cause')
```

Lots of outliers seem to exist in this dataset, and the way that the price vs affected clusters seem to group looks as though a DesisionTree/RandomForest regressor would be good for making predicitons here

It also seems like there are certain causes that cause longer outage times, so oneHotEncoding this information will likely also be a good feature to include in the model

[ ]:
```
peek = formated_data.groupby('CAUSE.CATEGORY').count()
look = peek[['TOTAL.PRICE', 'CUSTOMERS.AFFECTED', 'OUTAGE.DURATION']].loc['fuel␣
↪supply emergency']
look
```

[ ]:
```
4
TOTAL.PRICE           50
CUSTOMERS.AFFECTED     7
OUTAGE.DURATION       38
Name: fuel supply emergency, dtype: int64
```

Note that the values we want to train on contain mostly nan values for the fuel supply emergency column - find a way to impute this so that there is enough data of this type to train the model on

1.6 Step 6: Baseline Model

For out Baseline Model, we will start by using a RandomForestRegressor, including the Customers Affected and a OneHotEncoding of the Cause catagory, trained via a gridsearch. For this baseline, we will not try to find perfect parameters, simply look over a couple of spaced ones to find an ideal outcome.

[ ]:
```
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder, StandardScaler
from sklearn.model_selection import GridSearchCV
from sklearn.experimental import enable_iterative_imputer
from sklearn.impute import KNNImputer, SimpleImputer, IterativeImputer
# define the training data
training_attrs = ['CAUSE.CATEGORY', 'CUSTOMERS.AFFECTED', 'TOTAL.SALES',␣
 ↪'POPDEN_UC', 'PC.REALGSP.CHANGE', 'ANOMALY.LEVEL', 'PCT_WATER_INLAND']
```

2 Breakdown of feature selection

- Cause: some causes seem to be highly correlated with longer outages, likely because they take longer to fix
- Customers Affected: The more customers that are affected by an outage, (in theory) the higher the priority of fixing that outage woul be
- Total.Sales: the total amount of power that is put out to the customers would likely imply a larger grid, and probably better infrastructer, making outage response time much better
- POPDEN_UC: Population density in urban clusters would correlate to the number of people that are affected, and how large the grid is - like above, larger grid likely means better ability to fix it
- PC.REALGSP.CHANGE: the change in states gross product year on year is probably correlated with how well it is able to run it's power grid, and how much money is in the state at a time. If this goes down suddenly, a state is likely less able to handle difficult outages, because there may be some budget cuts
- Anomaly level: A measruement of how much of an el nino year it is. If this value is more extreme, the weather may be more severe, and therefore make fixing outaged more difficult
- PCT_WATER_INLAND: in theory, this could be usefull in combination with the previous data point, since more water inland probably means more storms and therefore more difficulty fixing power

[ ]:
```
no_nans = formated_data.dropna(subset=['OUTAGE.DURATION'])
X_train, X_test, y_train, y_test = train_test_split(no_nans, no_nans['OUTAGE.
 ↪DURATION'], test_size=0.2, random_state=42)
```

One issue with the features here is that many of the missing values seems to be highly correlated with being a fuel supply emergency, which also seems to indicate much higher outage times. Becasue of the missingness, the model likely ownt be able to make use of this as well, so we will impute some missing values so we can still use these data points. To do this, we are using the Itterative Imputer from sklearn - many of these values seem to be somewhat dependent on a category, so we dont want to impute solely on one value accross all nans

```
[ ]: trans = ColumnTransformer([  # define the column transformer - one hot encode␣
     ↪the cause category and scale the rest
         ('cat', OneHotEncoder(handle_unknown='ignore'), ['CAUSE.CATEGORY']),
         ('scale', StandardScaler(), [
             'CUSTOMERS.AFFECTED', 'TOTAL.SALES', 'POPDEN_UC',
             'POPPCT_UC', 'PC.REALGSP.CHANGE', 'PCT_WATER_INLAND',
             'ANOMALY.LEVEL'
             ])],
             remainder='drop'  # drop any columns not specified, since this will be␣
     ↪passed all of the columns
         )

     pipe = Pipeline([  # define the pipeline
         ('trans', trans),
         ('fill_nans', IterativeImputer()),  # fill the nans that might exist
         ('model', RandomForestRegressor())
     ])

     param_grid = {  # define the parameter grid for the grid search - we are not␣
     ↪trying a large number, since this is just a baseline
         'model__n_estimators': [10, 50, 100],
         'model__max_depth': [10, 50, 100]
     }

     grid = GridSearchCV(pipe, param_grid, cv=5, scoring='r2')
     grid.fit(X_train, y_train)
     grid.score(X_test, y_test)
```

```
[ ]: 0.2178480728718809
```

2.1 Step 7: Final Model

The Likely the biggest issue with our old model is the imputation scheme, as we can preform a probabalistic imputation accross certain columns if we know they are correlated. To do this we will define a custom imputation class. We will hard-code a couple of imputation groups, since we will only be needing this for this model

```
[ ]: from sklearn.base import BaseEstimator, TransformerMixin
     from sklearn.impute import SimpleImputer

     class CustomImputer(BaseEstimator, TransformerMixin):
         """
         CustomImputer is a class that imputes missing values in a pandas DataFrame␣
     ↪using a specified strategy.

         Parameters:
         -----------
```

```
    strategy : str, optional
        The imputation strategy to use. Default is 'mean'.
    """

    def __init__(self, strategy='mean'):
        self.strategy = strategy
        self.imputer = KNNImputer()
        self.groups = {
            'CUSTOMERS.AFFECTED': 'CAUSE.CATEGORY',
            'TOTAL.SALES': 'NERC.REGION',
        }

    def _impute(self, ser: pd.Series):
        """
        Imputes missing values in a pandas Series using random sampling from␣
↪non-missing values.

        Parameters:
        -----------
        ser : pd.Series
            The Series to impute missing values for.

        Returns:
        --------
        pd.Series
            The Series with imputed missing values.
        """
        nans = ser[ser.isna()]
        if nans.shape[0] == 0 or nans.shape[0] == ser.shape[0]:
            return ser
        newvals = np.random.choice(ser.dropna(), nans.shape[0])
        ser.loc[nans.index] = newvals
        return ser

    def fit(self, X, y=None):
        """
        Fits the imputer to the data.

        Parameters:
        -----------
        X : pd.DataFrame
            The input data.
        y : None
            Ignored.

        Returns:
        --------
```

```
        self
        """
        return self

    def transform(self, X: pd.DataFrame, y=None):
        """
        Transforms the input data by imputing missing values.

        Parameters:
        -----------
        X : pd.DataFrame
            The input data to transform.
        y : None
            Ignored.

        Returns:
        --------
        pd.DataFrame
            The transformed data with imputed missing values.
        """
        for val in self.groups:
            X.loc[:, val] = X.groupby(self.groups[val])[val].transform(self.
↪_impute)
        return X
```

Seccondly, we will modify some of the variables we are passing the model. First, since the preportion of water inland isn't really a usefull feature on it's own, we will multiply it with the anomaly level. Finaly, we will add a new feature, called likely_loss, which is the percentage of a states population that was affected multiplied by the total sales, as this proably correlates with the amount of demand lost and therefore the priority of fixing the outage.

[ ]:
```
class FeatureMultiplier(BaseEstimator, TransformerMixin):
    """
    A transformer class that multiplies two input features and creates a new␣
↪feature.

    Parameters:
    -----------
    feature1 : str
        The name of the first feature to be multiplied.
    feature2 : str
        The name of the second feature to be multiplied.
    new_feature : str
        The name of the new feature to be created.

    Methods:
    --------
```

```
    fit(X, y=None)
        Fit the transformer to the data.

    transform(X)
        Transform the data by multiplying the specified features and creating a␣
↪new feature.

    """

    def __init__(self, feature1, feature2, new_feature):
        self.feature1 = feature1
        self.feature2 = feature2
        self.new_feature = new_feature

    def fit(self, X, y=None):
        """
        Fit the transformer to the data.

        Parameters:
        -----------
        X : array-like, shape (n_samples, n_features)
            The input data.

        y : array-like, shape (n_samples,), optional (default=None)
            The target values.

        Returns:
        --------
        self : object
            Returns the instance itself.

        """
        return self

    def transform(self, X):
        """
        Transform the data by multiplying the specified features and creating a␣
↪new feature.

        Parameters:
        -----------
        X : array-like, shape (n_samples, n_features)
            The input data.

        Returns:
        --------
        X_transformed : array-like, shape (n_samples, n_features + 1)
```

```
            The transformed data with the new feature added.

        """
        X[self.new_feature] = X[self.feature1] * X[self.feature2]
        return X
```

[ ]:
```
class FeatureDivider(BaseEstimator, TransformerMixin):
    """
    A transformer class that divides two features and creates a new feature.

    Parameters:
    -----------
    feature1 : str
        The name of the first feature to be divided.
    feature2 : str
        The name of the second feature to be divided.
    new_feature : str
        The name of the new feature to be created.

    Methods:
    --------
    fit(X, y=None)
        Fit the transformer on the input data.

    transform(X)
        Transform the input data by dividing the specified features and␣
 ↪creating a new feature.

    """

    def __init__(self, feature1, feature2, new_feature):
        self.feature1 = feature1
        self.feature2 = feature2
        self.new_feature = new_feature

    def fit(self, X, y=None):
        """
        Fit the transformer on the input data.

        Parameters:
        -----------
        X : array-like or dataframe
            The input data to be transformed.
        y : array-like, optional
            The target variable. Default is None.

        Returns:
```

```
        --------
        self : FeatureDivider
            The fitted transformer object.

        """
        return self

    def transform(self, X):
        """
        Transform the input data by dividing the specified features and␣
 ↪creating a new feature.

        Parameters:
        -----------
        X : array-like or dataframe
            The input data to be transformed.

        Returns:
        --------
        X : array-like or dataframe
            The transformed data with the new feature added.

        """
        X[self.new_feature] = X[self.feature1] / X[self.feature2]
        return X
```

[ ]:
```
trans = ColumnTransformer([  # define the column transformer - more or less the␣
 ↪same as before, but dropping some columns
    ('cat', OneHotEncoder(handle_unknown='ignore'), ['CAUSE.CATEGORY']),
    ('scale', StandardScaler(), [
        'CUSTOMERS.AFFECTED', 'TOTAL.SALES', 'POPDEN_UC',
        'POPPCT_UC', 'PC.REALGSP.CHANGE', 'likely_loss',
        'weather'
        ])],
        remainder='drop'
    )

new_pipe = Pipeline([
    ('impute', CustomImputer()), # use the custom imputer first
    ('make_weather', FeatureMultiplier('ANOMALY.LEVEL', 'PCT_WATER_INLAND',␣
 ↪'weather')),
    ('make_pop', FeatureDivider('CUSTOMERS.AFFECTED', 'POPULATION',␣
 ↪'prop_customers')),
    ('make_loss', FeatureMultiplier('prop_customers', 'TOTAL.SALES',␣
 ↪'likely_loss')),  # create new features
    ('trans', trans), # transform the data
    ('fill_final', KNNImputer()),  # fill the nans
```

```
    ('model', RandomForestRegressor())
])
```

Now we will tune the same hyperparameters as before, since those are the most effective for the RandomForestRegressor. Again, we will tune with only a few patameters, and optimize one getting an idea for how good these are

[ ]:
```
params = {
    'model__n_estimators': [30, 40, 50, 60, 70, 80, 90, 100],
    'model__max_depth': [10, 20, 30, 40, 50, 60, 70, 80]
}

grid = GridSearchCV(new_pipe, params, cv=5, scoring='r2')
grid.fit(X_train, y_train)
grid.score(X_test, y_test)
```

[ ]:
```
0.29187607299037066
```

[ ]:
```
grid.best_params_
```

[ ]:
```
{'model__max_depth': 50, 'model__n_estimators': 60}
```

Now that we have some idea of where we want to aim, lets narrow the area we are training on

[ ]:
```
new_params = {
    'model__n_estimators': [55, 56, 57, 58, 59, 60, 61, 62, 63, 64],
    'model__max_depth': [45, 46, 47, 48, 49, 50, 51, 52, 53, 54]
}

grid = GridSearchCV(new_pipe, new_params, cv=5, scoring='r2')
grid.fit(X_train, y_train)
grid.score(X_test, y_test)
```

[ ]:
```
0.23566699371261224
```

[ ]:
```
best = grid.best_estimator_
best.fit(no_nans, no_nans['OUTAGE.DURATION'])
```

```
/Users/martinhawks/miniconda3/envs/dsc80/lib/python3.8/site-
packages/pandas/core/indexing.py:1773: 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

/var/folders/v7/nxggzv_j5s936v9rvl85gh2w0000gn/T/ipykernel_3632/1746629757.py:64
```

```
: 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

/var/folders/v7/nxggzv_j5s936v9rvl85gh2w0000gn/T/ipykernel_3632/3391565793.py:63
: 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
```

```
[ ]: Pipeline(steps=[('impute', CustomImputer()),
                     ('make_weather',
                      FeatureMultiplier(feature1='ANOMALY.LEVEL',
                                        feature2='PCT_WATER_INLAND',
                                        new_feature='weather')),
                     ('make_pop',
                      FeatureDivider(feature1='CUSTOMERS.AFFECTED',
                                     feature2='POPULATION',
                                     new_feature='prop_customers')),
                     ('make_loss',
                      FeatureMultiplier(feature1='prop_customers',
                                        feature2='TOTAL.SALES',
                                        new_fe…='likely_loss')),
                     ('trans',
                      ColumnTransformer(transformers=[('cat',
     OneHotEncoder(handle_unknown='ignore'),
                                                       ['CAUSE.CATEGORY']),
                                                      ('scale', StandardScaler(),
                                                       ['CUSTOMERS.AFFECTED',
                                                        'TOTAL.SALES', 'POPDEN_UC',
                                                        'POPPCT_UC',
                                                        'PC.REALGSP.CHANGE',
                                                        'likely_loss',
                                                        'weather'])])),
                     ('fill_final', KNNImputer()),
                     ('model',
                      RandomForestRegressor(max_depth=52, n_estimators=60))])
```

Overall, an improvement, although the model is still not very effective

2.2 Step 8: Fairness Analysis

For our fairness analysis, we will group by the average wealth of each region, to see if our model is less accurate for poorer regions than it is for wealthier ones. We will again group by the NERC region, since there are many states with too few data points to get an accurate result

H_0: our model is fair, and does not have higher accuracy for any region
H_1 our model is more accurate on wealthier regions than poor ones

[ ]:
```
def scores(df: pd.DataFrame):
    return best.score(df, df['OUTAGE.DURATION'])

grouped_scores = no_nans.groupby('NERC.REGION').apply(scores).dropna()
stats.chisquare(grouped_scores)
```

```
/Users/martinhawks/miniconda3/envs/dsc80/lib/python3.8/site-
packages/sklearn/metrics/_regression.py:918: UndefinedMetricWarning:

R^2 score is not well-defined with less than two samples.

/Users/martinhawks/miniconda3/envs/dsc80/lib/python3.8/site-
packages/sklearn/metrics/_regression.py:918: UndefinedMetricWarning:

R^2 score is not well-defined with less than two samples.

/Users/martinhawks/miniconda3/envs/dsc80/lib/python3.8/site-
packages/sklearn/metrics/_regression.py:918: UndefinedMetricWarning:

R^2 score is not well-defined with less than two samples.
```

[ ]:
```
Power_divergenceResult(statistic=3.7226256300270517, pvalue=0.92870230221801)
```

Given the outcome of our test, we will fail to reject the null hypothesis, and determine that our model is reletively fair accross all of the regions