# Missing Data Problem

Equinor collects a lot of data from the windmills, and naturally this unstructured data results in datasets with a lot of missing entries. The dataset at hand is a real example of one of these and a problem is analysing such a dataset. This document will try to find look after patterns in the missing data. Perhaps some tags and turbines can be separated, and perhaps some tags affects other tags in the dataset?

## Loading the dataset to the notebook

In [5]:
```
!curl -O https://eqhck20storage.blob.core.windows.net/missing-data/missing_data_task.csv
```

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 1729M  100 1729M    0     0  9005k      0  0:03:16  0:03:16 --:--:-- 9515k29M   24  416M    0     0  9257k      0  0:03:11  0:00:46  0:02:25 9436k
```

In [2]:
```
import pandas as pd
data = pd.read_csv('missing_data_task.csv', nrows=1000, index_col=[0], parse_dates=[0])
```

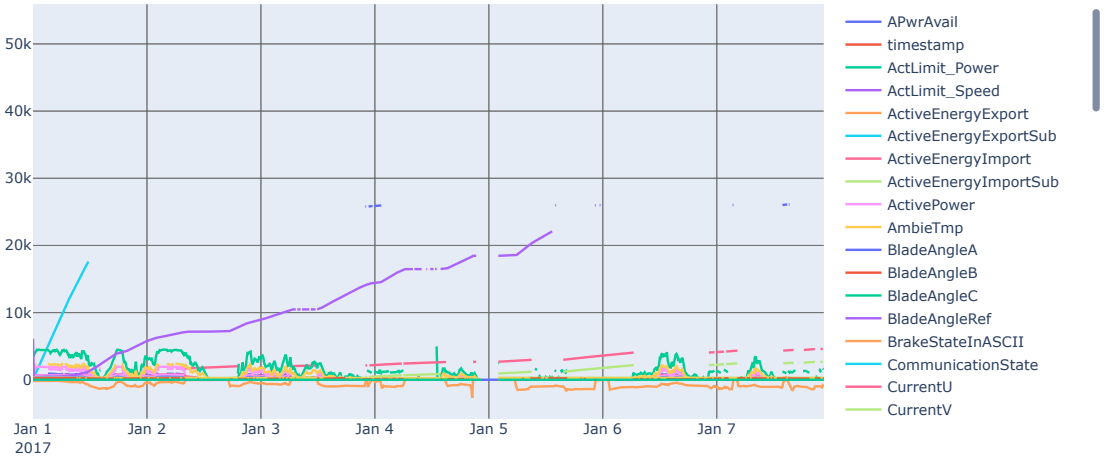
### Splitting up the data by turbine

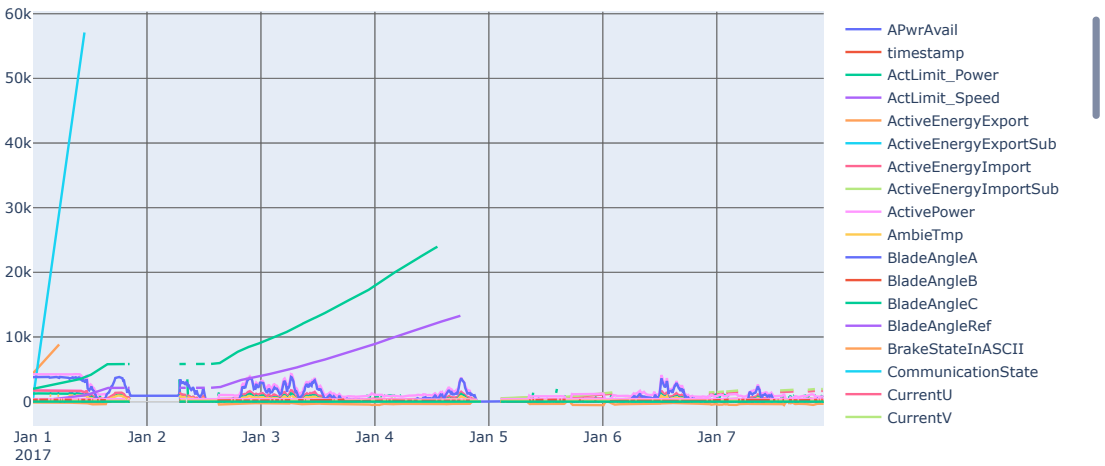
In [3]:
```
turbines = {}
for key in data.keys():
    turbine = key.split('-')[0]
    if turbine in turbines:
        continue

    turbines[turbine] = pd.DataFrame()

    columns = sorted([col.split('-')[1] for col in filter(lambda s: key[:3] in s, data.columns)])
    for col in columns:
        turbines[turbine][col] = data[f'{turbine}-{col}']
        turbines[key[:3]]["timestamp"] = turbines[key[:3]].index
T = turbines
```

## Comparison

Visualizing the column attributes for two turbines gives us a good overview of the available data.

In [4]:
```
import plotly.graph_objects as go


def visualizeTurbine(turbine):
    fig = go.Figure()

    t = turbines[turbine]
    for column in t.columns:
        fig.add_trace(go.Scatter(x=t["timestamp"], y=t[column], name=column))

    fig.show()

visualizeTurbine("T01")
visualizeTurbine("T02")
```




## Nullity Matrix

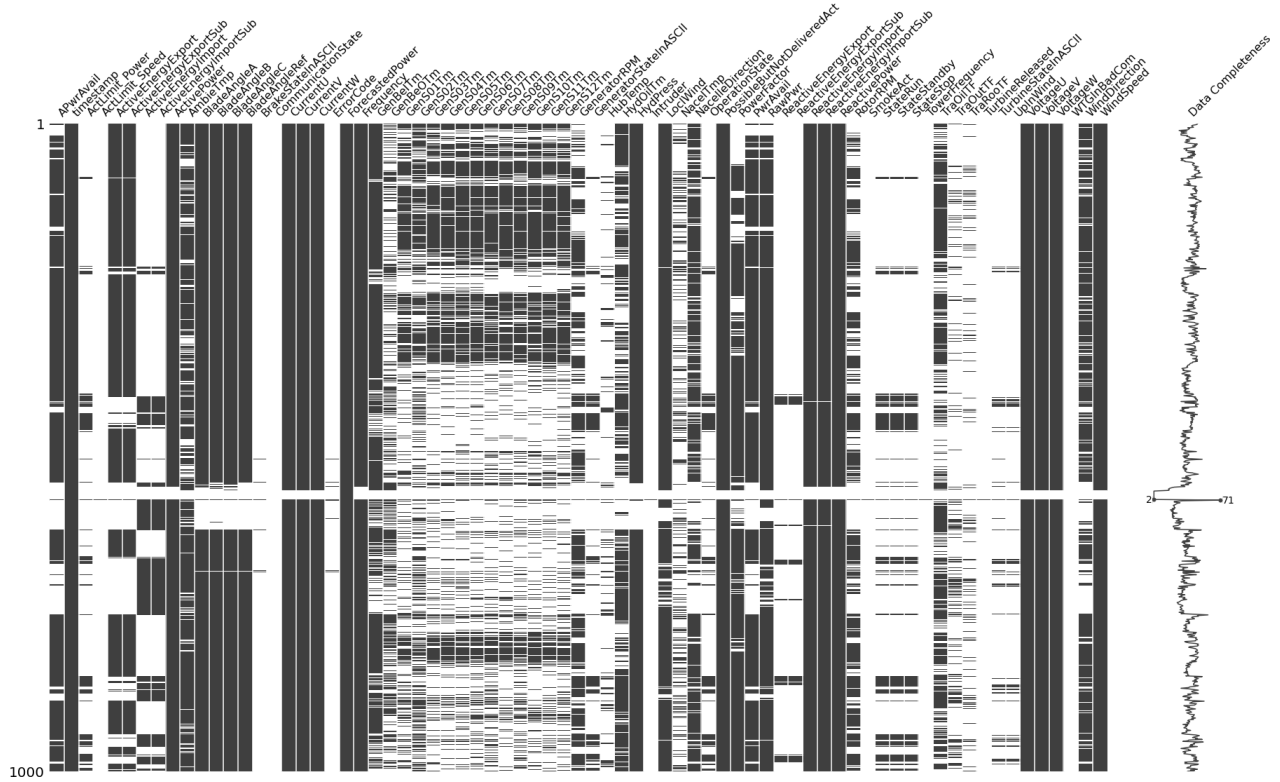
The function `msno.matrix` results in a nullity matrix, a data-dense display that lets you quickly visually pick out patterns in data completion. The sparkline at right summarizes the general shape of the turbine-data completeness and points out the rows with the maximum and minimum nullity in the dataset for turbine 04. The purpose of drawing the matrix for all the columns is to see what values are missing, and to get a general overview of the missing data.

At a glance one may notice that every column except `ForecastedPower` has some missing data. A deeper look reveals that some of the columns seem to act similarly (like `StateRun`, `StateStandby` and `StateStop`) and others have nearly the opposite nullity (like `ActiveEnergyExport` and `ActiveEnergyImport`).

In [ ]:
```
!pip install missingno
```

In [7]:
```
import missingno as msno
msno.matrix(turbines["T04"], labels=True, figsize=(30, 17))
```

Out[7]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f848329b2e0>
```

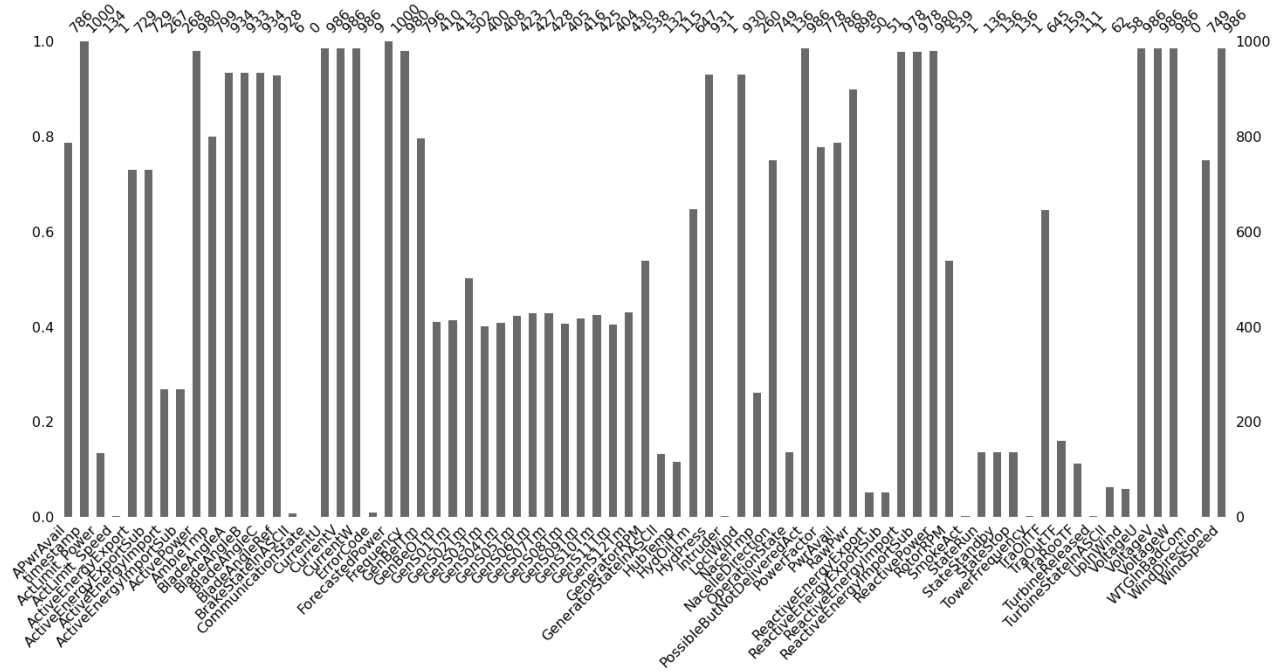



## Bar chart | Number of non-null values per attribute

The bar chart gives a much more simpler overview of the data. Let us look at turbine 04 again.

In [8]:
```
msno.bar(turbines["T04"], labels=True)
```

Out[8]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f8483115430>
```




## Heatmap

The heatmap is probably the most important visualization for this dataset, because it measures nullity correlation: how strongly the presence or absence of one variable affects the presence of another. Nullity correlation ranges from -1 (if one variable appears the other definitely does not) to 0 (variables appearing or not appearing have no effect on one another) to 1 (if one variable appears the other definitely also does).

Remember: correlation does not equal causation! For example, the nullity of `GeneratorStateInASCII` correlates almost perfectly with the nullity of `ActLimit_Power`, but this is only because they both are very sparse columns.

In [11]:
```
msno.heatmap(turbines["T01"], figsize=((50,50)))
```

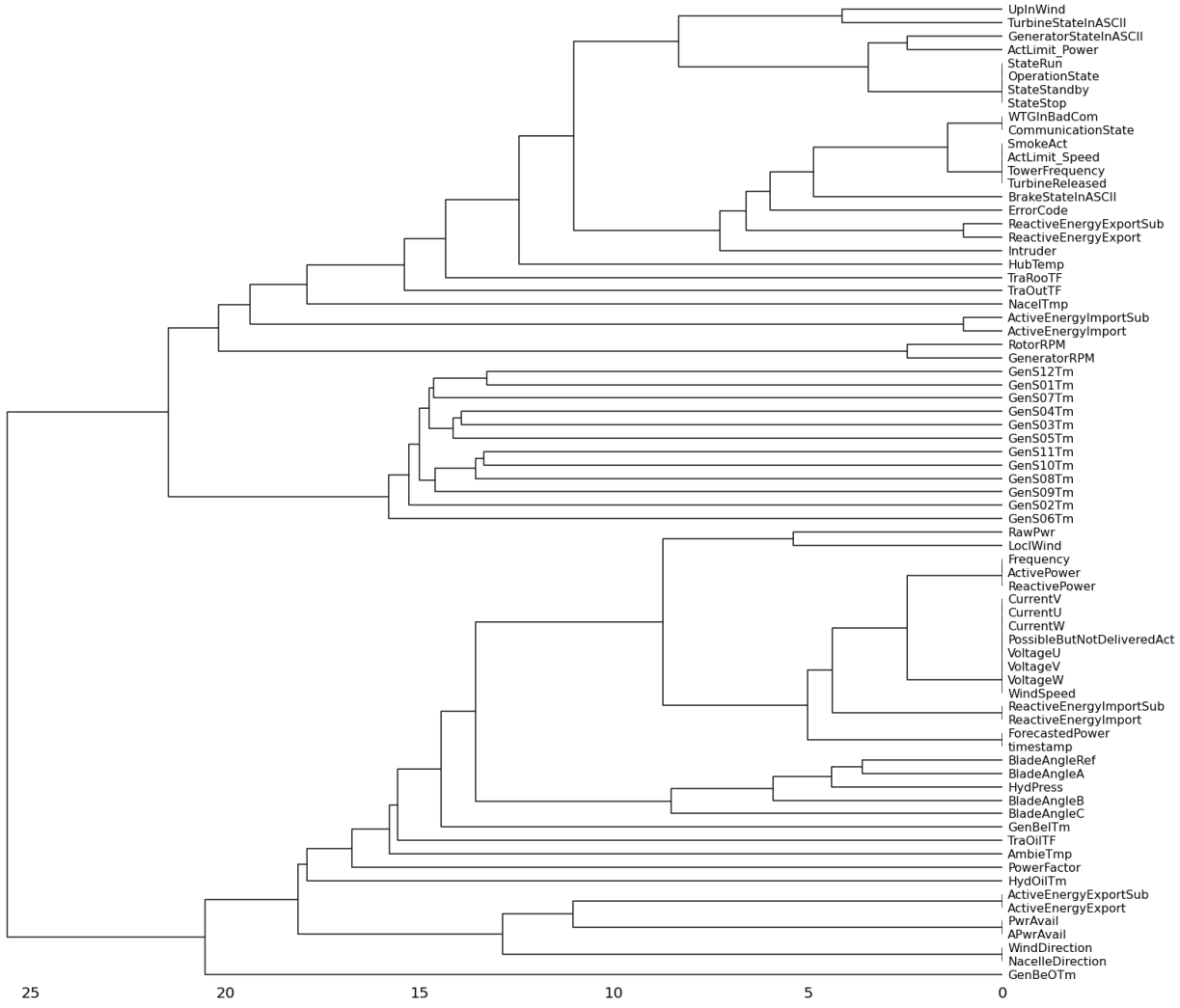
Out[11]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f8473b67100>
```


## Dendrogram

The dendrogram allows us to more fully correlate variable completion, revealing trends deeper than the pairwise ones visible in the correlation heatmap. The dendrogram uses a hierarchical clustering algorithm to bin variables against one another by their nullity correlation (measured in terms of binary distance). Below we have drawn out a dendrogram for the first turbine in the dataset.

In [12]:
```
msno.dendrogram(turbines["T01"])
```

Out[12]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f8474bd9b50>
```



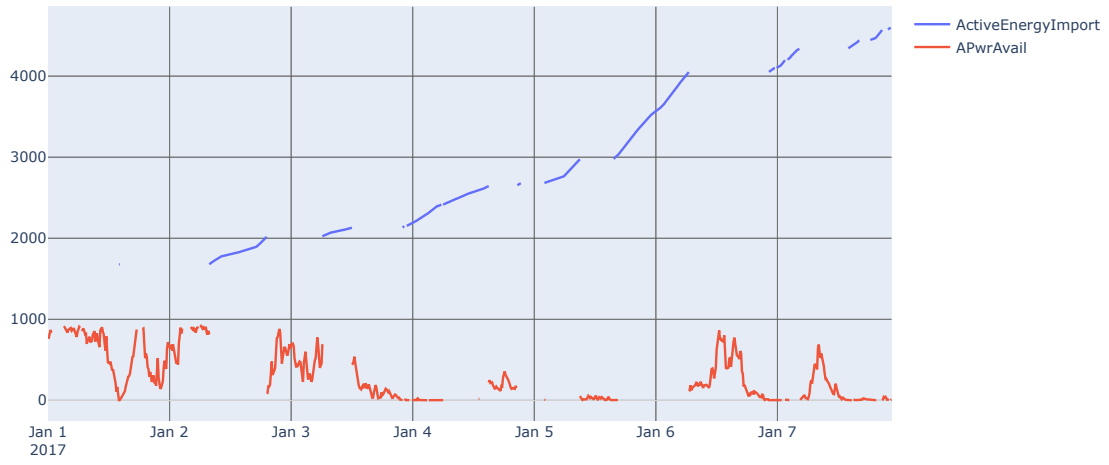

In [14]:
```
from utils import nullity_filter, nullity_sort
import numpy as np
def create_corr_matrix(turbine):

    turbines[turbine] = nullity_filter(turbines[turbine], filter=None, n=0, p=0)
    turbines[turbine] = nullity_sort(turbines[turbine], sort=None, axis='rows')

    # Remove completely filled or completely empty variables.
    turbines[turbine] = turbines[turbine].iloc[:,[i for i, n in enumerate(np.var(turbines[turbine].isnull(), axis='rows')) if n > 0]]

    # Create and mask the correlation matrix. Construct the base heatmap.
    corr_mat = turbines[turbine].isnull().corr()
    return corr_mat
```

After creating the nullity correlation matrix for a given turbine, we look at the nullity correlation between the different attributes for the turbine. We have decided to set the limit to -0.5, because that is pretty significant, and will give a good indicator on the missing data in one attribute given another attribute.

In [15]:
```
patternCounter = {} # total occurences of that tuple

def findCorr(limit, corr):
    corr_indexes = [x for x in corr.values.flatten()]
    corr_indexes = [i if x <= limit else -1 for i,x in enumerate(corr_indexes)]
    return corr_indexes

def getColumnNames(index, columns):
    x_index = int(index%len(columns))
    y_index = int(index/len(columns))
    return columns[x_index], columns[y_index]

for turbine in turbines:
    corr_mat = create_corr_matrix(turbine)
    corr_indexes = findCorr(-0.5, corr_mat)
    for i in corr_indexes:
        if i != -1:
            s = getColumnNames(i, corr_mat.columns)
            if s not in patternCounter:
                patternCounter[s] = 1
            else:
                patternCounter[s] += 1
patternCounter = {k: v for k, v in sorted(patternCounter.items(), key=lambda item: item[1], reverse=True)}
patterns = list(patternCounter.keys())
```

In [16]:
```
turbines = {}
for key in data.keys():
    turbine = key.split('-')[0]
    if turbine in turbines:
        continue

    turbines[turbine] = pd.DataFrame()

    columns = sorted([col.split('-')[1] for col in filter(lambda s: key[:3] in s, data.columns)])
    for col in columns:
        turbines[turbine][col] = data[f'{turbine}-{col}']
    turbines[key[:3]]["timestamp"] = turbines[key[:3]].index
```

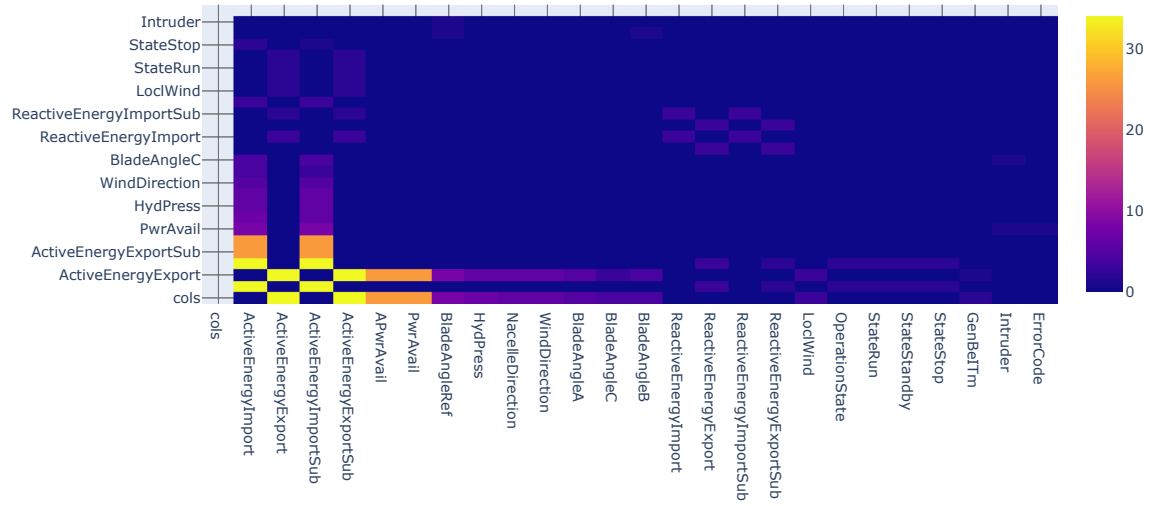
In [17]:
```
import plotly.graph_objects as go

fig = go.Figure()
t = turbines["T01"]
s = patterns[8]
for atr in s:
    fig.add_trace(go.Scatter(x=t["timestamp"], y=t[atr], name=atr))
fig.show()
```




Above, we see a pattern for the subcomponent for the "imported active energy" and "active power availability". We can see when there are missing data points for "active power availability", there is data for "imported active energy", and vice versa. We have discovered one of many patterns regarding missing data in the dataset.

In [18]:
```
corr_table = pd.DataFrame()

columns = {}
for key in patternCounter.keys():
    columns[key[0]] = True
    columns[key[1]] = True

corr_table["cols"] = columns.keys()
corr_table.set_index("cols")

for k in columns.keys():
    counts = []
    for j in columns.keys():
        counts.append(patternCounter[(k, j)] if (k, j) in patternCounter else 0)
    corr_table[k] = counts
```

In [19]:
```
import plotly.graph_objects as go

fig = go.Figure(data=go.Heatmap(
                   z=corr_table,
                   x=corr_table.columns,
                   y=corr_table.columns,
                   hoverongaps = False))
fig.show()
```




Above we can see the intensity of nullity correlation between the attributes for all the turbines in the dataset. To the right, you can see the intensity of the correlation (which is negative). The heatmap shows us that ActiveEnergyImport and ActiveEnergyExport have high nullity correlation, due to the fact that the turbine does not import and export data at the same time. We also see that ActiveEnergyExport/ActiveEnergyExportSub and APwrAvail have high nullity correlation, because the turbine does not export power when it does not have any available power to export.

# Dealing With Null Values

When actually using the data in such a sparse dataframe, it's important that the NaN (not present) values in the dataframe are handled properly. This is rarely a simple task. There are many special cases, and for these domain knowledge is often needed. We cite our sources for wind turbine specific information at the bottom of this chapter. As a guiding principle, we prefer gauge values to be interpolated linearly, state values to be treated as step functions, and we drop regions in time where most columns are empty. Sources

## Interpolation Map

We suggest a suite of interpolation methods for the different columns in the dataframe. Combining these, the data should be able to be used as a dense matrix. We use a the following color coding scheme

| Color | Meaning |
|---|---|
| Blue | Linear interpolation between points |
| Orange | Pad with zeros |
| Red | Step function interpolation |
| Green | Drop |
| Gray | Unknown |

## Columns

The columns are listed in their entirety here, along with information about what they represent, how their NaNs behave, and what method to use to fill NaN values.

### Power and Energy

**APwrAvail**: The available active power that can be used (NaN along with with RawPwr and LoclWind) [Linear]

**ActLimit_Power**: Maximum allowed active power usage (unpredictable NaNs, slightly correlated with EnergyImportPower NaNs) [Linear]

**ActLimit_Speed [Unknown]**: Unknown limiter (sparse data, likely because this is only reported when changed) [Step]

**ActiveEnergyExport**: Active power actually exported (NaN when importing or not running) [Pad importing/Drop not running]

**ActiveEnergyExportSub [1]**: Subcomponent of energy actually exported through active power (NaN when importing or not running) [Pad importing/Drop not running]

**ActiveEnergyImport**: Active power actually imported (NaN when exporting or not running) [Pad exporting/Drop not running]

**ActiveEnergyImportSub [1]**: Subcomponent of energy actually imported through active power (NaN when exporting or not running) [Pad exporting/Drop not running]

**ActivePower**: Signed active power being either supplied or consumed (NaN when not running) [Drop/Linear]

**ReactiveEnergyExport**: Reactive power exported, power resulting from capacitance and inductance (similar in properties to active counterpart) [Pad importing/Drop not running]

**ReactiveEnergyExportSub**: Subcomponent of energy actually exported as reactive power (NaN when importing or not running) [Pad importing/Drop not running]

**ReactiveEnergyImport**: Reactivy power actually imported (NaN when exporting or not running) [Pad exporting/Drop not running]

**ReactiveEnergyImportSub**: Subcomponent of energy actually imported through reactive power (NaN when exporting or not running) [Pad exporting/Drop not running]

**ReactivePower**: Signed reactive power input/output (NaN when not running) [Drop/Linear]

**ForecastedPower**: Predicted apparent power for the windmill. (Never NaN)

For the remaining sensor values we assume that the times that the windmill is not reporting have already been dropped from the dataframe, or filled in with alternative interpolated values.

**PowerFactor**: The ratio between working power and apparent power (NaN when the columns it depends on are NaN) [Linear]

**PwrAvail**: The power available for export (NaN when there's no power available) [Pad]

**RawPwr**: Signed apparent power output (mostly NaN when not running) [Linear]

## Blades

**BladeAngleA**: Angle of windmill blade A (NaN when not running) [Linear]

**BladeAngleB**: Angle of windmill blade B (NaN when not running) [Linear]

**BladeAngleC**: Angle of windmill blade C (NaN when not running) [Linear]

**BladeAngleRef [2]**: Reference angle for windmill blades. The blade angle is the offset from this reference (NaN when not running) [Linear]

## State Toggles

**BrakeStateInASCII**: Broadcast for change in brake state. It seems this data used to be ASCII, but has later been converted to an integral enum (NaN when there is no change in brake state) [Step]

**CommunicationState**: Broadcasted change in communication state. This is an integral enum (NaN when there is no change in communication state) [Step]

**ErrorCode**: Error code for when something happens. This always has a value when there is a break state update. (NaN when there is no change in error code) [Step]

**GeneratorStateInASCII**: Broadcasted updates in generator state. This used to be an ASCII string but has since been converted to an integral enum. (NaNs when there are no updates to generator state) [Step]

**StateRun**: Running state (NaN along with other state variables) [Step]

**StateStandby**: Standby state (NaN along with other state variables) [Step]

**StateStop**: Stop state (NaN along with other state variables) [Step]

**TurbineStateInASCII**: Enumerated state for turbine (NaNs correlate almost perfectly with UpInWind) [Step]

## Transformer Metrics

**CurrentU**: Current through electrical transformer phase U, part of phase sequence U-V-W (NaN only when not running) [Linear]

**CurrentV**: Current through electrical transformer phase V, part of phase sequence U-V-W (NaN only when not running) [Linear]

**CurrentW**: Current through electrical transformer phase W, part of phase sequence U-V-W (NaN only when not running) [Linear]

**VoltageU**: The voltage over transformer phase U (NaN only when not running) [Linear]

**VoltageV**: The voltage over transformer phase V (NaN only when not running) [Linear]

**VoltageW**: The voltage over transformer phase W (NaN only when not running) [Linear]

## Generator Sensors

**GenBelTm**: Unknown continuous sensor value for generator. (Unpredictable NaNs) [Linear]

**GenBeOTm**: Unknown continuous sensor value for generator. (Unpredictable NaNs) [Linear]

**GenS01Tm**: Temperature sensor on generator(Unpredictable NaNs) [Linear]

**GenS02Tm**: Temperature sensor on generator(Unpredictable NaNs) [Linear]

**GenS03Tm**: Temperature sensor on generator(Unpredictable NaNs) [Linear]

**GenS04Tm**: Temperature sensor on generator(Unpredictable NaNs) [Linear]

**GenS05Tm**: Temperature sensor on generator(Unpredictable NaNs) [Linear]

**GenS06Tm**: Temperature sensor on generator(Unpredictable NaNs) [Linear]

**GenS07Tm**: Temperature sensor on generator(Unpredictable NaNs) [Linear]

**GenS08Tm**: Temperature sensor on generator(Unpredictable NaNs) [Linear]

**GenS09Tm**: Temperature sensor on generator(Unpredictable NaNs) [Linear]

**GenS10Tm**: Temperature sensor on generator(Unpredictable NaNs) [Linear]

**GenS11Tm**: Temperature sensor on generator(Unpredictable NaNs) [Linear]

**GenS12Tm**: Temperature sensor on generator(Unpredictable NaNs) [Linear]

**GeneratorRPM**: Field magnet rotation rate for generator (large proportion of NaNs) [Linear]

### Miscellaneous

**AmbieTmp**: Ambient temperature external to windmill (NaN when not measuring) [Linear]

**Frequency**: Unknown frequency sensor somewhere on the windmill (NaN when not running) [Linear]

**HubTemp [4]**: Temperature gauge for hub, the part that connects the shaft to the blades (significant number of NaNs) [Linear]

**HydOilTm**: Hydrolic oil temperature sensor (large proportion of NaNs) [Linear]

**HydPress**: Hydraulic pressure sensor (mostly NaNs when not running) [Linear]

**Intruder** [Unknown]: Possibly a sensor for the maintenance hatch (not NaN only when maintenance is going on) [Unknown]

**LoclWind**[Unknown]: Wind related sensor value (mostly NaN when not running) [Linear]

**NacelTmp**: Temperature sensor on the housing for the generating components of the turbine (frequent NaNs) [Linear]

**NacelleDirection**: Direction that the housing of the turbine is facing (frequent NaNs) [Linear]

**OperationState**[Unknown]: Seems to be supposed to enumerate operation states, but instead has gibberish float values (mostly NaN) [Unknown]

**PossibleButNotDeliveredAct**: The amount of active power available for export that isn't being exported (NaN when not running) [Linear]

**RotorRPM**: Revolutions per minute for the rotor. (NaNs when not being polled) [Linear]

**SmokeAct [Unknown]**: Unknown column with almost no data [Unknown]

**UpInWind [Unknown]**: Unknown variable that correlates well with TurbineStateInASCII [Unknown]

**WTGInBadCom [Unknown]**: Wind Turbine Generator ? (very sparse data) [Unknown]

**WindDirection**: Wind direction (NaN along with other directional sensors, in particular Nacelle) [Linear]

**WindSpeed**: Wind speed as measured by the anemometer on the turbine. [Linear]

**TowerFrequency [Unknown]**: Unknown (very sparse) [Unknown]

**TraOilTF [Unknown]**: Unknown (random NaNs) [Unknown]

**TraOutTF [Unknown]**: Unknown (random NaNs) [Unknown]

**TraRooTF [Unknown]**: Unknown (random NaNs) [Unknown]

**TurbineReleased [Unknown]**: Unknown (very sparse) [Unknown]

### References


[1] http://www.ieso.ca/en/sector-participants/market-operations/-/media/490945b625004c60b45982771a644356.ashx (http://www.ieso.ca/en/sector-participants/market-operations/-/media/490945b625004c60b45982771a644356.ashx)

[2] https://books.google.no/books?id=2nYE5Eyi2twC&pg=PA535&lpg=PA535&dq=windmill+blade+%22reference+angle%22&source=bl&ots=ud2DuO73hn&sig=ACfU3U0j_ISg_PgvqukzuQVOjAoyByvD_Q&hl=e (https://books.google.no/books?id=2nYE5Eyi2twC&pg=PA535&lpg=PA535&dq=windmill+blade+%22reference+angle%22&source=bl&ots=ud2DuO73hn&sig=ACfU3U0j_ISg_PgvqukzuQVOjAoyByvD_Q&hl=e

[3] https://electronics.stackexchange.com/questions/195185/why-are-the-letters-u-v-and-w-used-in-ac-motors-to-represent-the-windings (https://electronics.stackexchange.com/questions/195185/why-are-the-letters-u-v-and-w-used-in-ac-motors-to-represent-the-windings)

[4] http://mstudioblackboard.tudelft.nl/duwind/Wind%20energy%20online%20reader/Static_pages/hub_type.htm (http://mstudioblackboard.tudelft.nl/duwind/Wind%20energy%20online%20reader/Static_pages/hub_type.htm)


# Applying our Methods to the Dataset

Now that we understand the features of our dataset and how to deal with data sparsity, we apply our fill methods to the columns of the dataset.

In [20]:
```
## First get prepare a dataframe for the first turbine

# Use only first turbine
df = turbines['T01']

# Drop where ActivePower is NaN per the described method above
df = df[df['ActivePower'].notna()].drop(columns=['timestamp'])

print('This DataFrame has a lot of NaN and inf values:')
df
```

```
This DataFrame has a lot of NaN and inf values:
```

Out[20]:

| | APwrAvail | ActLimit_Power | ActLimit_Speed | ActiveEnergyExport | ActiveEnergyExportSub | ActiveEnergyImport | ActiveEnergyImportSub | ActivePower | AmbieTmp | E |
|---|---|---|---|---|---|---|---|---|---|---|
| **2017-01-01 00:00:00** | 773.366200 | NaN | NaN | inf | inf | NaN | 0.0000 | 1632.496200 | NaN | |
| **2017-01-01 00:10:00** | 773.366200 | NaN | NaN | inf | 318.90344 | NaN | NaN | 1631.199600 | NaN | |
| **2017-01-01 00:20:00** | 826.193400 | NaN | NaN | inf | 553.95610 | NaN | NaN | 1768.645600 | 8.181367 | |
| **2017-01-01 00:30:00** | 868.578060 | NaN | NaN | inf | 808.75430 | NaN | NaN | 1878.861800 | 8.167484 | |
| **2017-01-01 00:40:00** | 836.021700 | NaN | NaN | inf | 1069.50320 | NaN | NaN | 1889.235100 | NaN | |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... | ... | |
| **2017-01-07 21:50:00** | 8.782145 | 1237.5913 | NaN | inf | inf | 4574.967 | 2666.8276 | 6.741624 | 11.457616 | |
| **2017-01-07 22:00:00** | 3.762405 | 1129.9746 | NaN | NaN | NaN | 4581.217 | 2672.5813 | -24.717600 | 12.049932 | |
| **2017-01-07 22:10:00** | NaN | NaN | NaN | NaN | NaN | 4590.592 | 2682.6504 | -30.633617 | 12.318325 | |
| **2017-01-07 22:20:00** | 6.958530 | 1663.9187 | NaN | NaN | NaN | 4599.967 | 2692.7192 | -20.371760 | 12.003657 | |
| **2017-01-07 22:30:00** | 15.068816 | 1543.0569 | NaN | inf | inf | NaN | NaN | 29.073572 | 11.365067 | |

973 rows × 72 columns

In [21]:
```
# Dictionary of preferences for how to deal with missing data
missing_preference = {
    'unknown': ['Intruder', 'OperationState', 'SmokeAct', 'UpInWind',
                'WTGInBadCom', 'TowerFrequency', 'TraOilTF', 'TraOutTF',
                'TraRooTF', 'TurbineReleased'],
    'step': ['ActLimit_Speed', 'BrakeStateInASCII', 'CommunicationState',
             'ErrorCode', 'GeneratorStateInASCII', 'StateRun', 'StateStandby',
             'StateStop', 'TurbineStateInASCII'],
    'linear': ['APwrAvail', 'ActLimit_Power', 'ActivePower', 'ReactivePower',
               'PowerFactor', 'RawPwr',
               *[f'BladeAngle{l}' for l in ('A', 'B', 'C')],
               'BladeAngleRef', 'CurrentU', 'CurrentV', 'CurrentW', 'VoltageU',
               'VoltageV', 'VoltageW', 'Frequency', 'GenBeITm', 'GenBeOTm',
               *[f'GenS{i:02}Tm' for i in range(1, 13)],
               'GeneratorRPM', 'AmbieTmp', 'HubTemp', 'HydOilTm', 'HydPress',
               'LoclWind', 'NacelTmp', 'NacelleDirection',
               'PossibleButNotDeliveredAct', 'RotorRPM', 'WindDirection',
               'WindSpeed'
               ],
    'pad': ['ActiveEnergyExport', 'ActiveEnergyExportSub', 'ActiveEnergyImport',
            'ActiveEnergyImportSub', 'ReactiveEnergyExportSub',
            'ReactiveEnergyImport', 'ReactiveEnergyImportSub', 'PwrAvail'],
}
```

In [22]:
```
# Discard unknown columns
patched = df.drop(columns=missing_preference['unknown'])

# Treat infinite values as nans
patched = patched.replace([-np.inf, np.inf], np.nan)

# Perform step interpolation
st = missing_preference['step']
patched.loc[:, st] = patched.loc[:, st].ffill().bfill()

# Perform linear interpolation
lin = missing_preference['linear']
patched.loc[:, lin] = patched.loc[:, lin].interpolate().bfill()

# Perform zero padding
pad = missing_preference['pad']
patched.loc[:, pad] = patched.loc[:, pad].fillna(method='pad')

# Discard columns with no data
patched = patched.dropna(thresh=len(patched), axis=1)

# Check that there are no nans in the remaining dataframe
print('There are', patched.isna().sum().sum(), 'nan entries in \'patched\'.')

patched
```

```
There are 0 nan entries in 'patched'.
```

Out[22]:

| | APwrAvail | ActLimit_Power | ActLimit_Speed | ActiveEnergyImportSub | ActivePower | AmbieTmp | BladeAngleA | BladeAngleB | BladeAngleC | BladeAngleRef | ... | Roto |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **2017-01-01 00:00:00** | 773.366200 | 1316.23410 | 8.726741 | 0.0000 | 1632.496200 | 8.181367 | -0.756830 | -2.882732 | -1.288745 | -3.115802 | ... | 3.2 |
| **2017-01-01 00:10:00** | 773.366200 | 1316.23410 | 8.726741 | 0.0000 | 1631.199600 | 8.181367 | -0.741568 | -2.775841 | -1.253416 | -3.011352 | ... | 3.2 |
| **2017-01-01 00:20:00** | 826.193400 | 1316.23410 | 8.726741 | 0.0000 | 1768.645600 | 8.181367 | -0.541362 | -2.054323 | -0.948710 | -2.271349 | ... | 3.2 |
| **2017-01-01 00:30:00** | 868.578060 | 1316.23410 | 8.726741 | 0.0000 | 1878.861800 | 8.167484 | -0.247788 | -0.900228 | -0.428723 | -1.046272 | ... | 3.2 |
| **2017-01-01 00:40:00** | 836.021700 | 1316.23410 | 8.726741 | 0.0000 | 1889.235100 | 8.176739 | 0.114804 | 0.403557 | 0.148213 | 0.379960 | ... | 3.2 |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | |
| **2017-01-07 21:50:00** | 8.782145 | 1237.59130 | 8.726741 | 2666.8276 | 6.741624 | 11.457616 | -0.342055 | -1.408798 | -0.609780 | -0.274846 | ... | 1.1 |
| **2017-01-07 22:00:00** | 3.762405 | 1129.97460 | 8.726741 | 2672.5813 | -24.717600 | 12.049932 | 2.023600 | 7.843170 | 3.323798 | 9.304917 | ... | 0.9 |
| **2017-01-07 22:10:00** | 5.360468 | 1396.94665 | 8.726741 | 2682.6504 | -30.633617 | 12.318325 | 4.840840 | 18.024593 | 7.907637 | 19.119694 | ... | 0.1 |
| **2017-01-07 22:20:00** | 6.958530 | 1663.91870 | 8.726741 | 2692.7192 | -20.371760 | 12.003657 | 2.499425 | 9.039020 | 3.933211 | 11.223968 | ... | 1.2 |
| **2017-01-07 22:30:00** | 15.068816 | 1543.05690 | 8.726741 | 2692.7192 | 29.073572 | 11.365067 | -0.753688 | -2.820936 | -1.251944 | -2.526278 | ... | 1.2 |

973 rows × 56 columns

## Testing the Filled Dataset

Finally we put our processed dataset to use. The data is used to train and test a gradient boosted regressor. For now let's predict the wind speed given the other information.

In [26]:
```
!pip install lightgbm
```

```
Defaulting to user installation because normal site-packages is not writeable
Collecting lightgbm
  Using cached lightgbm-2.3.1-py2.py3-none-manylinux1_x86_64.whl (1.2 MB)
Requirement already satisfied: numpy in /usr/lib/python3.8/site-packages (from lightgbm) (1.18.1)
Requirement already satisfied: scipy in /usr/lib/python3.8/site-packages (from lightgbm) (1.4.1)
Requirement already satisfied: scikit-learn in /usr/lib/python3.8/site-packages (from lightgbm) (0.22.2.post1)
Installing collected packages: lightgbm
Successfully installed lightgbm-2.3.1
```

In [27]:
```
import lightgbm as lgb
from sklearn.model_selection import train_test_split
```

In [28]:
```
# Prepare data for training
X = patched.drop(columns=['WindSpeed'])
y = patched['WindSpeed']

X_train, X_test, y_train, y_test =\
  train_test_split(X, y.values.flatten(), test_size=0.10, random_state=42, shuffle=False)

train_data = lgb.Dataset(X_train, label=y_train)
validation_data = lgb.Dataset(X_test, label=y_test)
```

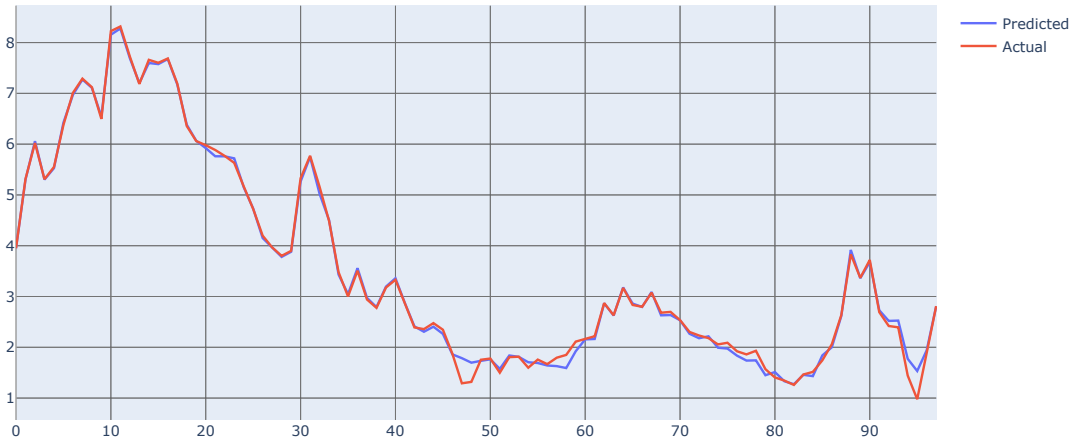
In [29]:
```
# Create gradient boosted regressor model
param = {'objective': 'regression_l2', "num_leaves": 200, "num_estimators": 100, "max_depth": 10, "max_bin": 100, "early_stopping_round": 30}
param['metric'] = 'l2'

# Train model
bst = lgb.train(param, train_data, 1000, valid_sets=[validation_data])
```

```
[1]     valid_0's l2: 4.55567
Training until validation scores don't improve for 30 rounds
[2]     valid_0's l2: 3.70057
[3]     valid_0's l2: 3.00632
[4]     valid_0's l2: 2.43918
[5]     valid_0's l2: 1.98452
[6]     valid_0's l2: 1.616
[7]     valid_0's l2: 1.31266
[8]     valid_0's l2: 1.07063
[9]     valid_0's l2: 0.866731
[10]    valid_0's l2: 0.697819
[11]    valid_0's l2: 0.569519
[12]    valid_0's l2: 0.462847
[13]    valid_0's l2: 0.378305
[14]    valid_0's l2: 0.307146
[15]    valid_0's l2: 0.247069
[16]    valid_0's l2: 0.202643
[17]    valid_0's l2: 0.166109
[18]    valid_0's l2: 0.136576
[19]    valid_0's l2: 0.111851
[20]    valid_0's l2: 0.0925649
[21]    valid_0's l2: 0.077521
[22]    valid_0's l2: 0.0647165
[23]    valid_0's l2: 0.0546736
[24]    valid_0's l2: 0.0463366
[25]    valid_0's l2: 0.0394305
[26]    valid_0's l2: 0.03337
[27]    valid_0's l2: 0.029377
[28]    valid_0's l2: 0.0262106
[29]    valid_0's l2: 0.0237427
[30]    valid_0's l2: 0.0217058
[31]    valid_0's l2: 0.0198476
[32]    valid_0's l2: 0.0184248
[33]    valid_0's l2: 0.0172483
[34]    valid_0's l2: 0.0163962
[35]    valid_0's l2: 0.0154936
[36]    valid_0's l2: 0.0148453
[37]    valid_0's l2: 0.0143423
[38]    valid_0's l2: 0.0140282
[39]    valid_0's l2: 0.0137305
[40]    valid_0's l2: 0.0136801
[41]    valid_0's l2: 0.0134238
[42]    valid_0's l2: 0.0133478
[43]    valid_0's l2: 0.0130758
[44]    valid_0's l2: 0.0129509
[45]    valid_0's l2: 0.0129465
[46]    valid_0's l2: 0.0128744
[47]    valid_0's l2: 0.0128106
[48]    valid_0's l2: 0.0127935
[49]    valid_0's l2: 0.012691
[50]    valid_0's l2: 0.0126304
[51]    valid_0's l2: 0.0127389
[52]    valid_0's l2: 0.0126961
[53]    valid_0's l2: 0.0127311
[54]    valid_0's l2: 0.0127097
[55]    valid_0's l2: 0.0126506
[56]    valid_0's l2: 0.0127279
[57]    valid_0's l2: 0.0126647
[58]    valid_0's l2: 0.012761
[59]    valid_0's l2: 0.0128157
[60]    valid_0's l2: 0.0128675
[61]    valid_0's l2: 0.0129914
[62]    valid_0's l2: 0.0130374
[63]    valid_0's l2: 0.0131271
[64]    valid_0's l2: 0.0130764
[65]    valid_0's l2: 0.0132114
[66]    valid_0's l2: 0.0132463
[67]    valid_0's l2: 0.01329
[68]    valid_0's l2: 0.0132584
[69]    valid_0's l2: 0.013305
[70]    valid_0's l2: 0.0133474
[71]    valid_0's l2: 0.0133958
[72]    valid_0's l2: 0.0133752
[73]    valid_0's l2: 0.0133754
[74]    valid_0's l2: 0.0135465
[75]    valid_0's l2: 0.0135097
[76]    valid_0's l2: 0.0136542
[77]    valid_0's l2: 0.0136474
[78]    valid_0's l2: 0.0136557
[79]    valid_0's l2: 0.0136009
[80]    valid_0's l2: 0.0137919
Early stopping, best iteration is:
[50]    valid_0's l2: 0.0126304

/home/marcelroed/.local/lib/python3.8/site-packages/lightgbm/engine.py:153: UserWarning:

Found `early_stopping_round` in params. Will use it instead of argument
```

### Results on Validation Set

Let's try out the regression model on our moderately sized validation set.

In [30]:
```
import plotly.graph_objects as go

fig = go.Figure()

fig.add_trace(go.Scatter(
    x=list(range(len(X_test))),
    y=bst.predict(X_test),
    # color=[i % 48 for i in range(len(X_test))],
   name='Predicted',
),
)
fig.add_trace(go.Scatter(
    x=list(range(len(y_test))),
    y=y_test,
   name='Actual',
))

fig.show()
```



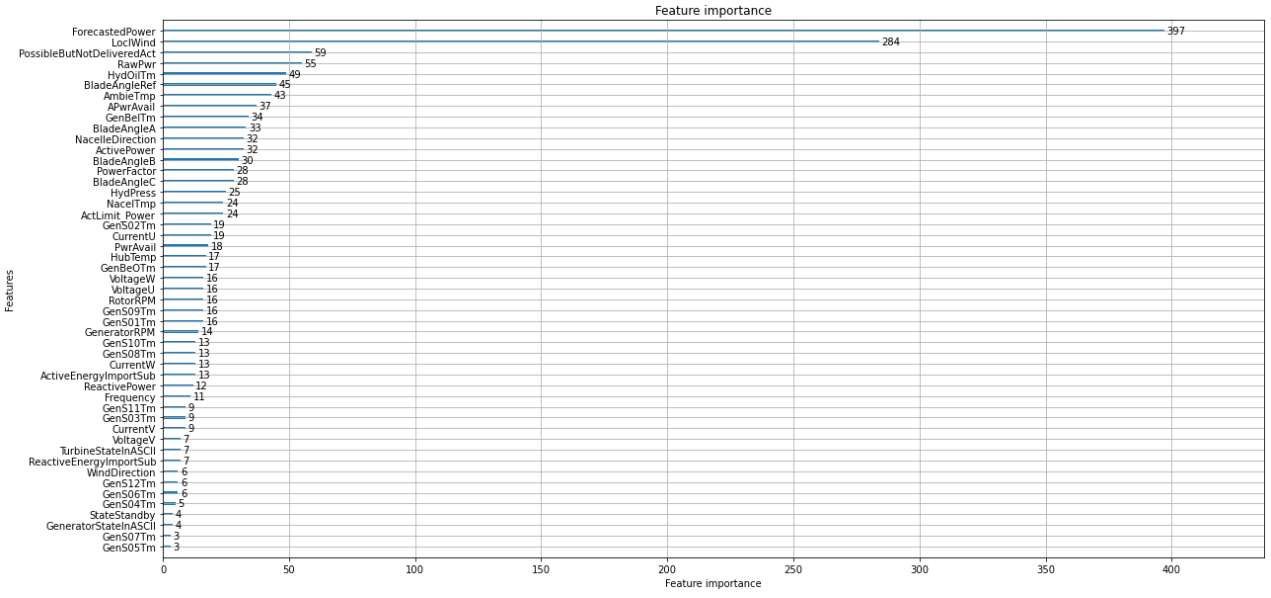

The model predicts the curve quite well.

### Importance

Now have a look at the importance plots for the gradient boosted model. These signify what features of the dataset are the most useful for determining the proper value of our response variable. Notably, the most significant features are also features that have a low NaN ratio in the original DataFrame. However, several of these features did contain NaNs, and would not be usable if it weren't for the way they were filled in earlier.

In [31]:
```
lgb.plot_importance(bst, figsize=(20, 10))
```

Out[31]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f846f11d610>
```




# Conclusion

Missing data is an inevitable problem when handling large quantities of data, and this problem should be handled with care. We can look for patterns in the data, and understand how the nullity (or "missingness") of features are related, and many tools exist for this job.

By understanding the data at hand and using the right tool for the right job, we can fill in the gaps in a fairly sparse dataframe, and practically use it as if it were a complete dataset.