Table of Contents

Wind Turbine Underperformance Anomaly Detection

@author: Shivank Garg

```
Created:  May 01, 2024 11:35 AM
Rev 1  :
```

Common Terms

```
PI = Plant Information
AF = Asset Framework
WTG = Wind Turbine
WPP = Wind Power Plant
TM = Ten Minutes
```

Executive Summary

--

Setup Notebook

In [1]:
```
##Import Libraries
# ! pip install mpl_toolkits
import time
import pandas as pd
import numpy as np
import requests
from mpl_toolkits import mplot3d
import matplotlib.pyplot as plt
import seaborn as sns

# plt.style.use("seaborn-colorblind")
%matplotlib inline
%config InlineBackend.print_figure_kwargs = {'bbox_inches':None}

### Supress warnings
import warnings
warnings.filterwarnings("ignore")
```

Define Helper Functions

In [2]:
```
### Timer Function
def timer(start, end):
    hours, rem = divmod(end - start, 3600)
    minutes, seconds = divmod(rem, 60)
    print("{:0>2}:{:0>2}:{:05.2f}".format(int(hours), int(minutes), seconds))

### Dummy Variable Function. Creating multiple columns for categorical variable.
def encode_and_bind(original_dataframe, feature_to_encode):
    dummies = pd.get_dummies(original_dataframe[[feature_to_encode]])
    res = pd.concat([original_dataframe, dummies], axis=1)
    res = res.drop([feature_to_encode], axis=1)
    return(res)
```

Import Required Data

Import Telemetry Data

In [3]:
```
### Merged Dataset
merged_df = pd.read_csv('./OrientMergedDataset20220701-20230630_Extra.csv')
```

In [4]:
```
merged_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 13544393 entries, 0 to 13544392
Data columns (total 12 columns):
 #   Column                          Dtype
---  ------                          -----
 0   AssetParent                     object
 1   AssetName                       object
 2   TM_ActivePower                  float64
 3   Timestamp                       object
 4   TM_PossiblePower                float64
 5   State_Fault_Interpolated        float64
 6   Windspeed_Adjusted              float64
 7   Generator_RPM                   float64
 8   TM_AmbientTemperature_Celsius   float64
 9   RotorRPM                        float64
 10  BladeAngle                      float64
 11  TM_SetPoint                     float64
dtypes: float64(9), object(3)
memory usage: 1.2+ GB
```

In [5]:
```
### Convert timestamp
merged_df['TM_ActivePower'] = pd.to_numeric(merged_df['TM_ActivePower'], errors = 'coerce')
merged_df['Timestamp'] = pd.to_datetime(merged_df['Timestamp'])
merged_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 13544393 entries, 0 to 13544392
Data columns (total 12 columns):
 #   Column                          Dtype
---  ------                          -----
 0   AssetParent                     object
 1   AssetName                       object
 2   TM_ActivePower                  float64
 3   Timestamp                       datetime64[ns]
 4   TM_PossiblePower                float64
 5   State_Fault_Interpolated        float64
 6   Windspeed_Adjusted              float64
 7   Generator_RPM                   float64
 8   TM_AmbientTemperature_Celsius   float64
 9   RotorRPM                        float64
 10  BladeAngle                      float64
 11  TM_SetPoint                     float64
dtypes: datetime64[ns](1), float64(9), object(2)
memory usage: 1.2+ GB
```

In [6]:
```
merged_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 13544393 entries, 0 to 13544392
Data columns (total 12 columns):
 #   Column                          Dtype
---  ------                          -----
 0   AssetParent                     object
 1   AssetName                       object
 2   TM_ActivePower                  float64
 3   Timestamp                       datetime64[ns]
 4   TM_PossiblePower                float64
 5   State_Fault_Interpolated        float64
 6   Windspeed_Adjusted              float64
 7   Generator_RPM                   float64
 8   TM_AmbientTemperature_Celsius   float64
 9   RotorRPM                        float64
 10  BladeAngle                      float64
 11  TM_SetPoint                     float64
dtypes: datetime64[ns](1), float64(9), object(2)
memory usage: 1.2+ GB
```

In [7]:
```
print("Number of nan values per column: ")
print()
merged_df.isna().sum()
```

```
Number of nan values per column:
```

Out[7]:
```
AssetParent                         0
AssetName                           0
TM_ActivePower                1516410
Timestamp                           0
TM_PossiblePower               245343
State_Fault_Interpolated         7367
Windspeed_Adjusted             138306
Generator_RPM                  144333
TM_AmbientTemperature_Celsius  234011
RotorRPM                       234011
BladeAngle                     146241
TM_SetPoint                         0
dtype: int64
```

In [8]:
```
df_10min = merged_df
df_10min = df_10min.set_index('Timestamp')
```

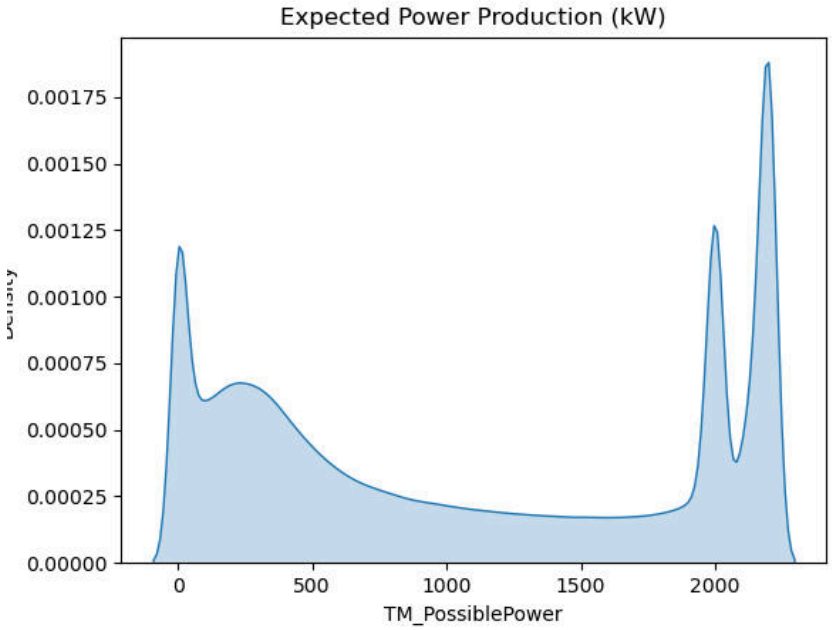
In [9]:
```
df_10min.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 13544393 entries, 2022-07-01 12:00:00 to 2023-06-30 03:40:00
Data columns (total 11 columns):
 #   Column                         Dtype
---  ------                         -----
 0   AssetParent                    object
 1   AssetName                      object
 2   TM_ActivePower                 float64
 3   TM_PossiblePower               float64
 4   State_Fault_Interpolated       float64
 5   Windspeed_Adjusted             float64
 6   Generator_RPM                  float64
 7   TM_AmbientTemperature_Celsius  float64
 8   RotorRPM                       float64
 9   BladeAngle                     float64
 10  TM_SetPoint                    float64
dtypes: float64(9), object(2)
memory usage: 1.2+ GB
```

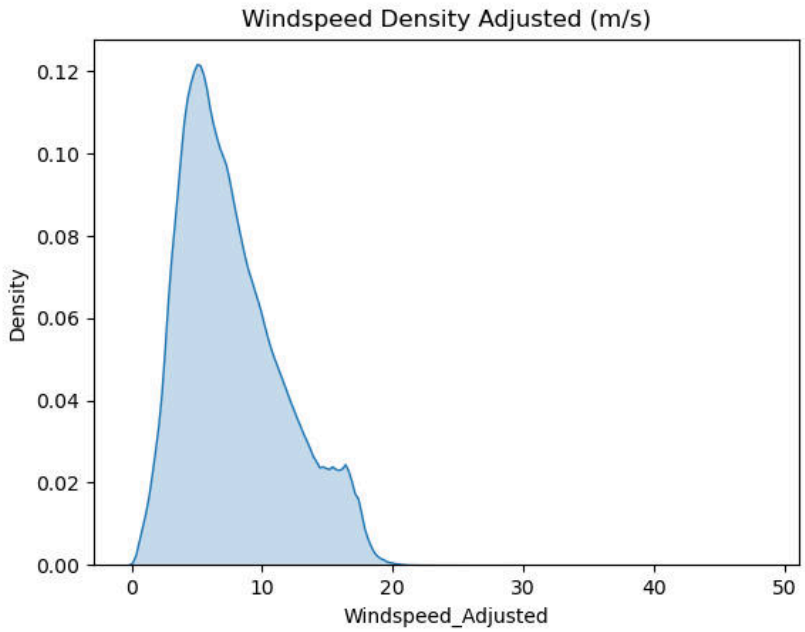
Exploratory Data Analysis

Plot Pairs

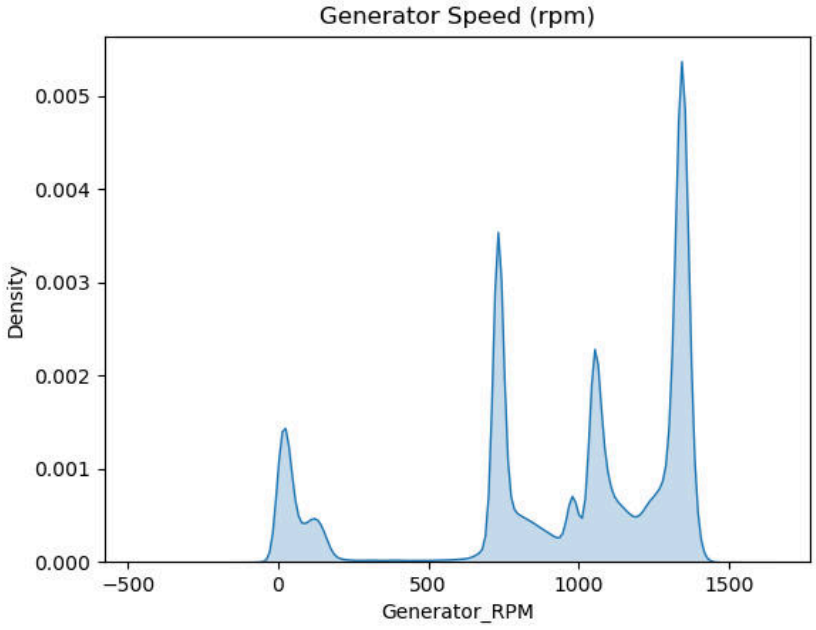
In [10]:
```
sns.kdeplot(df_10min.TM_PossiblePower, shade=True)
plt.title("Expected Power Production (kW)")
plt.show()
```



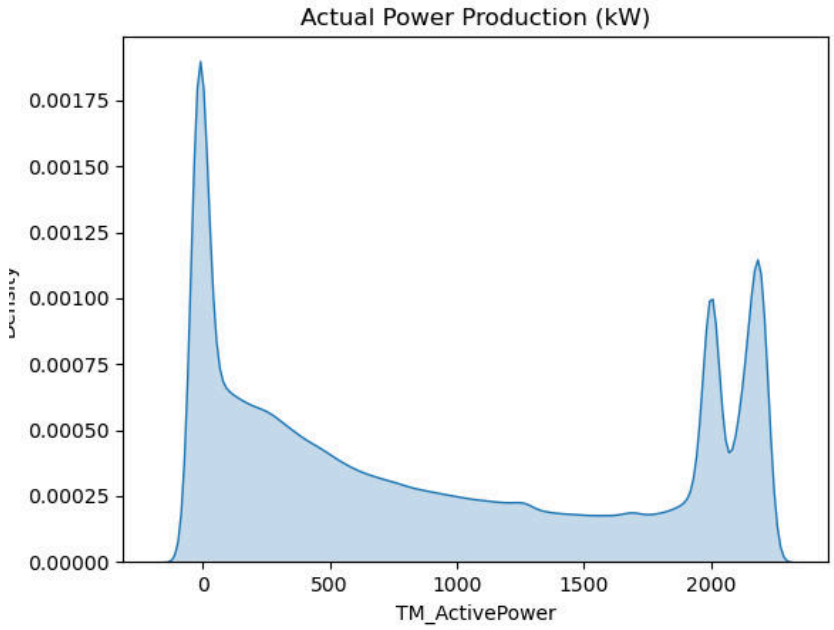
In [11]:
```
sns.kdeplot(df_10min.Windspeed_Adjusted, shade=True)
plt.title("Windspeed Density Adjusted (m/s)")
plt.show()
```



In [12]:
```
sns.kdeplot(df_10min.Generator_RPM, shade=True)
plt.title("Generator Speed (rpm)")
plt.show()
```

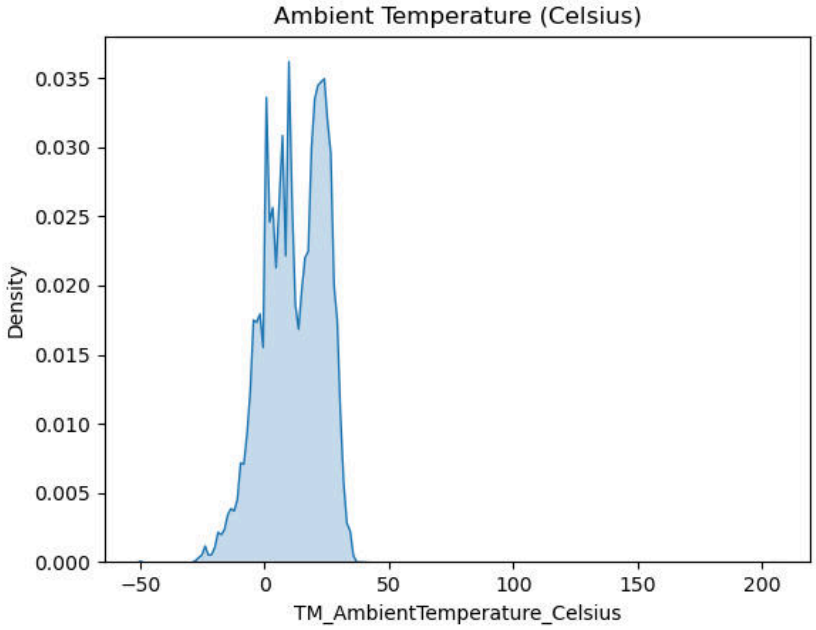


In [13]:
```
sns.kdeplot(df_10min.TM_ActivePower, shade=True)
plt.title("Actual Power Production (kW)")
plt.show()
```

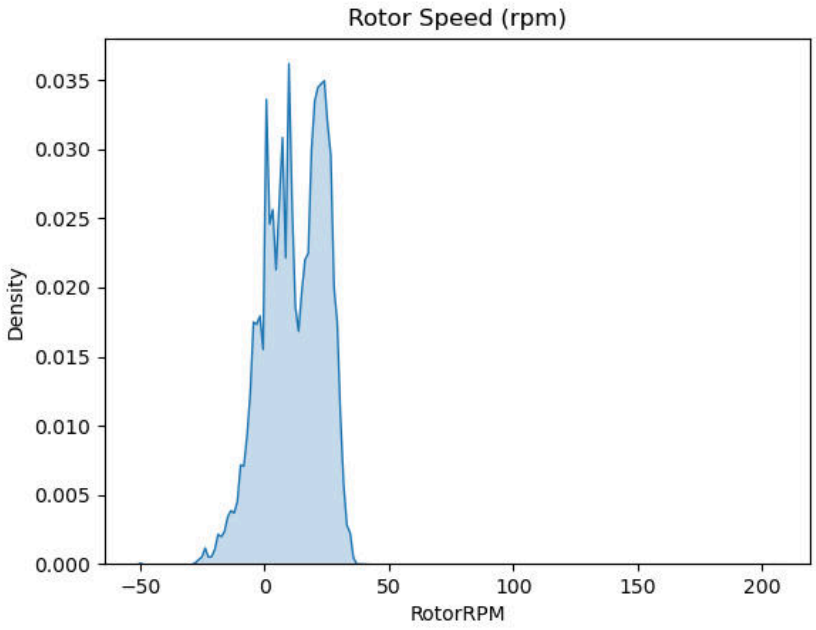


In [14]:
```
# sns.kdeplot(df_10min.State_Fault_Interpolated, shade=True)
# plt.title("Turbine Status: Alarm Codes")
# plt.show()
```

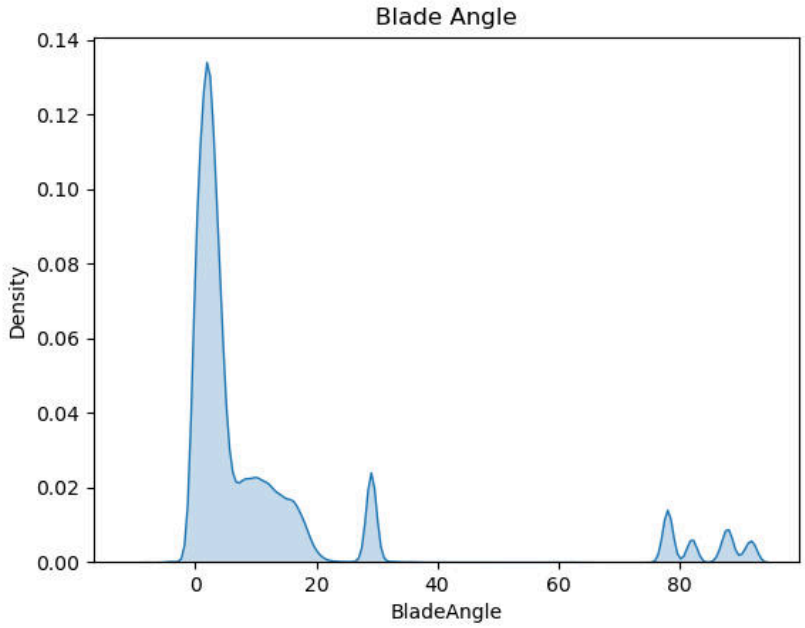
In [15]:
```
sns.kdeplot(df_10min.TM_AmbientTemperature_Celsius, shade=True)
plt.title("Ambient Temperature (Celsius)")
plt.show()
```



In [16]:
```
sns.kdeplot(df_10min.RotorRPM, shade=True)
plt.title("Rotor Speed (rpm)")
plt.show()
```

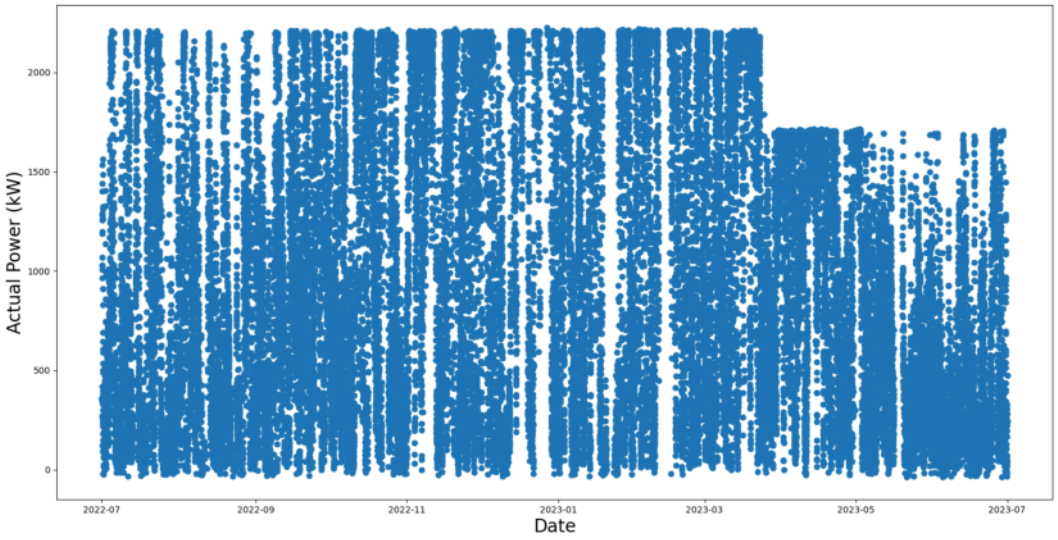


In [17]:
```
sns.kdeplot(df_10min.BladeAngle, shade=True)
plt.title("Blade Angle")
plt.show()
```



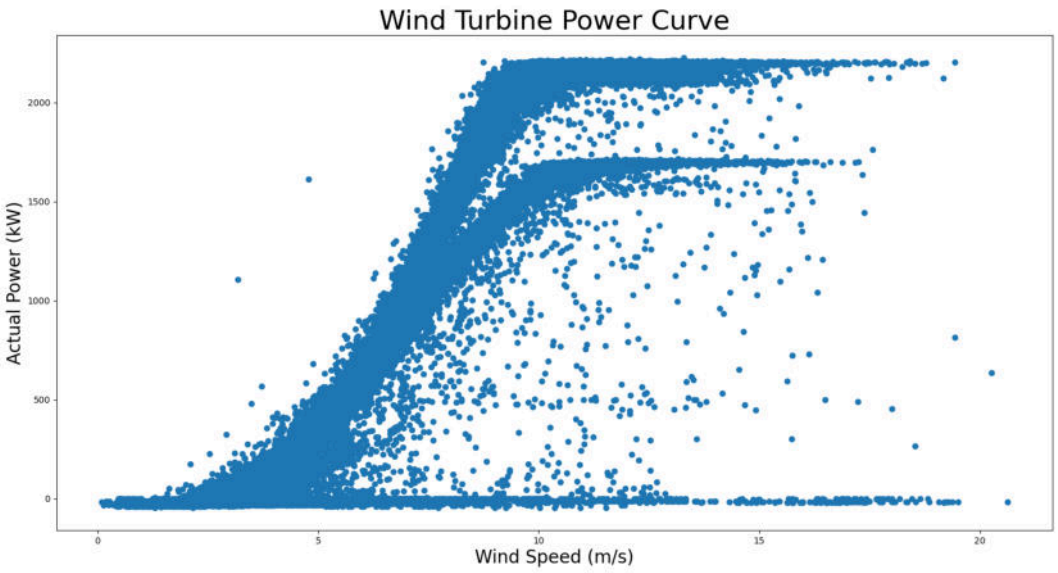
Plot Active Power

In [18]:
```
df = df_10min[df_10min['AssetName']== 'ORI-Ve-42']
df = df[df['State_Fault_Interpolated']== 2.0]
# Plot line chart including average, minimum and maximum temperature

fig = plt.figure(figsize=(20,10))
plt.scatter(x = df.index,y=df['TM_ActivePower'])

plt.xlabel("Date", fontsize=20)
plt.ylabel("Actual Power (kW)", fontsize=20)
plt.show()
```



Plot Power Curve

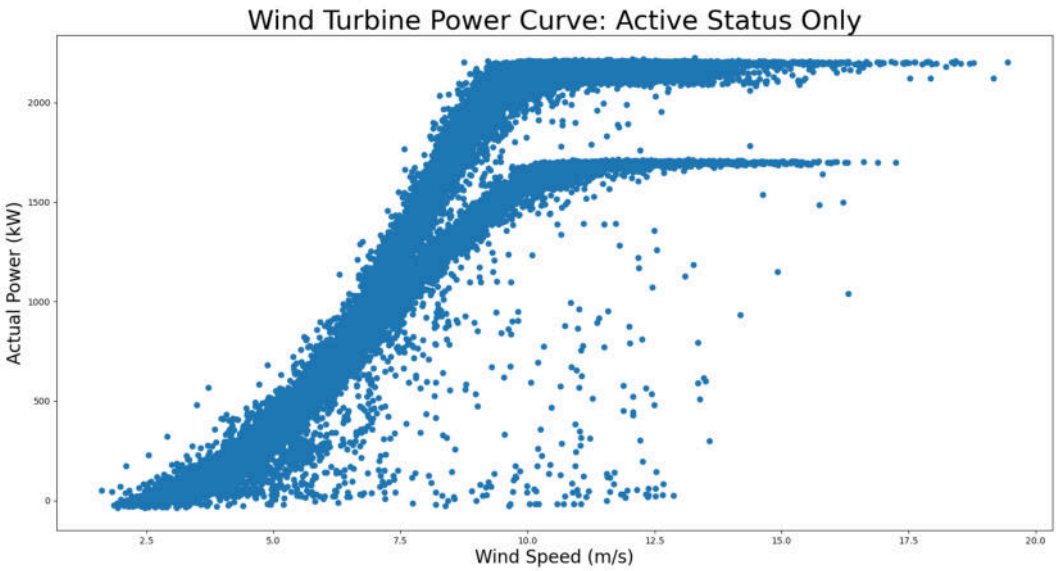
In [19]:
```
df = df_10min[df_10min['AssetName']== 'ORI-Ve-42']
# df = df[df['State_Fault_Interpolated']== 2.0]

fig = plt.figure(figsize=(20,10))

plt.scatter(x = df['Windspeed_Adjusted'],y=df['TM_ActivePower'])

plt.xlabel("Wind Speed (m/s)", fontsize=20)
plt.ylabel("Actual Power (kW)", fontsize=20)
plt.title("Wind Turbine Power Curve", fontsize=30)

plt.show()
```



Plot Power Curve (Only Active Turbine Status)

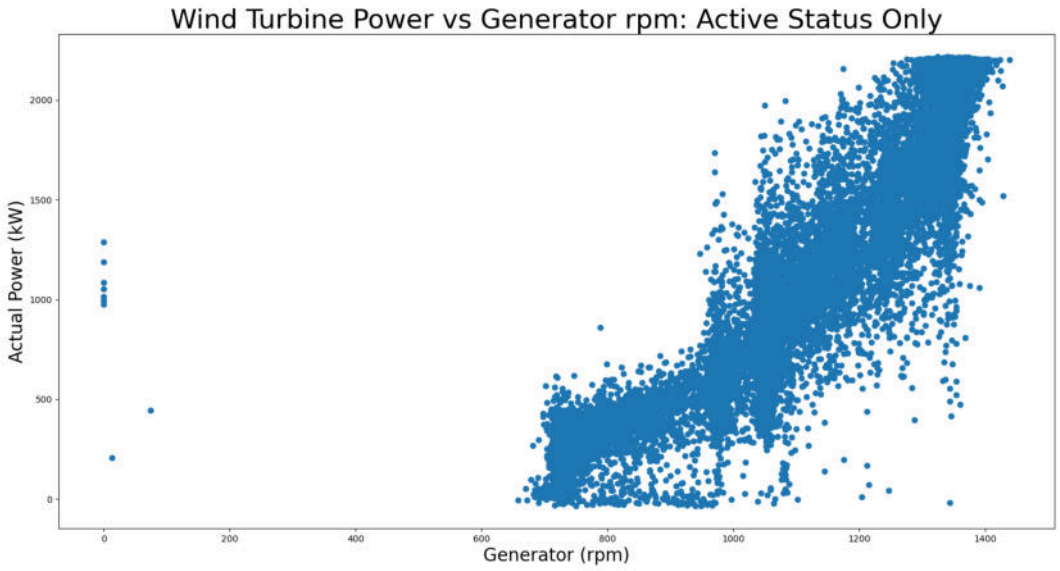
In [20]:
```
df = df_10min[df_10min['AssetName']== 'ORI-Ve-42']
df = df[df['State_Fault_Interpolated']== 2.0]
df = df[df['TM_SetPoint']== 500]

fig = plt.figure(figsize=(20,10))
```

```
plt.scatter(x = df['Windspeed_Adjusted'],y=df['TM_ActivePower'])

plt.xlabel("Wind Speed (m/s)", fontsize=20)
plt.ylabel("Actual Power (kW)", fontsize=20)
plt.title("Wind Turbine Power Curve: Active Status Only", fontsize=30)

plt.show()
```

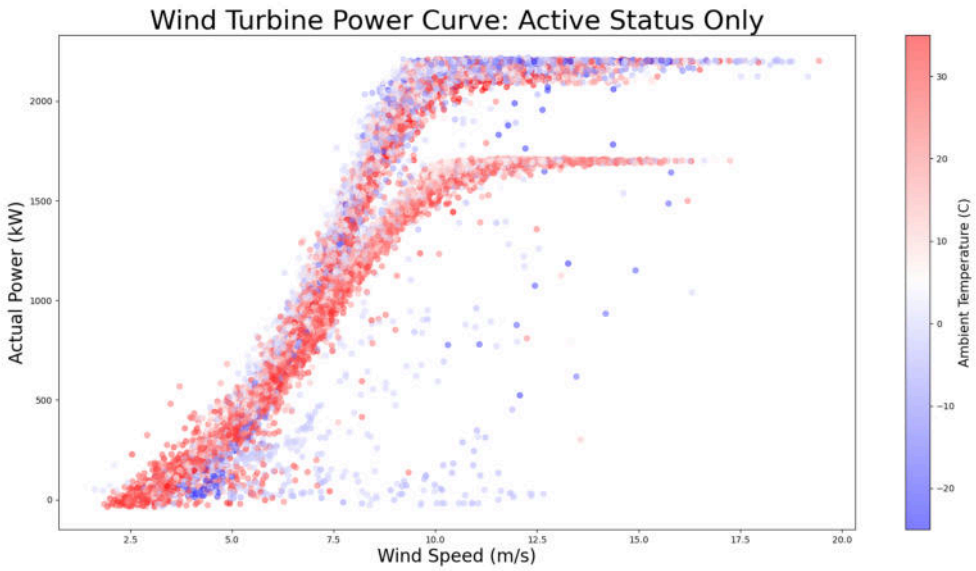


In [21]:
```
# df = df_10min[df_10min['AssetName']== 'ORI-Ve-42']
# df = df[df['State_Fault_Interpolated']== 2.0]
# df = df[df['TM_SetPoint']== 500]

# fig = plt.figure(figsize=(20,10))

# plt.scatter(x = df['TM_AmbientTemperature_Celsius'],y=df['TM_ActivePower'])

# plt.xlabel("Ambient Temperature (C)", fontsize=20)
# plt.ylabel("Actual Power (kW)", fontsize=20)
# plt.title("Wind Turbine Power Curve: Active Status Only", fontsize=30)

# plt.show()
```

In [22]:
```
df = df_10min[df_10min['AssetName']== 'ORI-Ve-42']
df = df[df['State_Fault_Interpolated']== 2.0]
df = df[df['TM_SetPoint']== 500]

fig = plt.figure(figsize=(20,10))

plt.scatter(x = df['Generator_RPM'],y=df['TM_ActivePower'])

plt.xlabel("Generator (rpm)", fontsize=20)
plt.ylabel("Actual Power (kW)", fontsize=20)
plt.title("Wind Turbine Power vs Generator rpm: Active Status Only", fontsize=30)

plt.show()
```



In [23]:
```
df = df_10min[df_10min['AssetName']== 'ORI-Ve-42']
df = df[df['State_Fault_Interpolated']== 2.0]
df = df[df['TM_SetPoint']== 500]

fig = plt.figure(figsize=(20,10))

plt.scatter(x = df['Windspeed_Adjusted'],y=df['TM_ActivePower'], c = df['TM_AmbientTemperature_Celsius'], cmap='bwr', alpha
cbar = plt.colorbar(orientation = 'vertical',label="Ambient Temperature (C)")

cbar.set_label(label="Ambient Temperature (C)", size=15)
plt.xlabel("Wind Speed (m/s)", fontsize=20)
plt.ylabel("Actual Power (kW)", fontsize=20)
plt.title("Wind Turbine Power Curve: Active Status Only", fontsize=30)

plt.show()
```



In [24]:
```
#compute mean power vs windspeed
cols = ['TM_ActivePower', 'Windspeed_Adjusted']
df = df[cols].copy()
binsize = 1
df['windspeed_bin'] = np.round(binsize*(df['Windspeed_Adjusted']/binsize))
agger = {'TM_ActivePower':['mean', 'std', 'count']}
df = df.groupby('windspeed_bin').agg(agger).reset_index().sort_values('windspeed_bin')
```

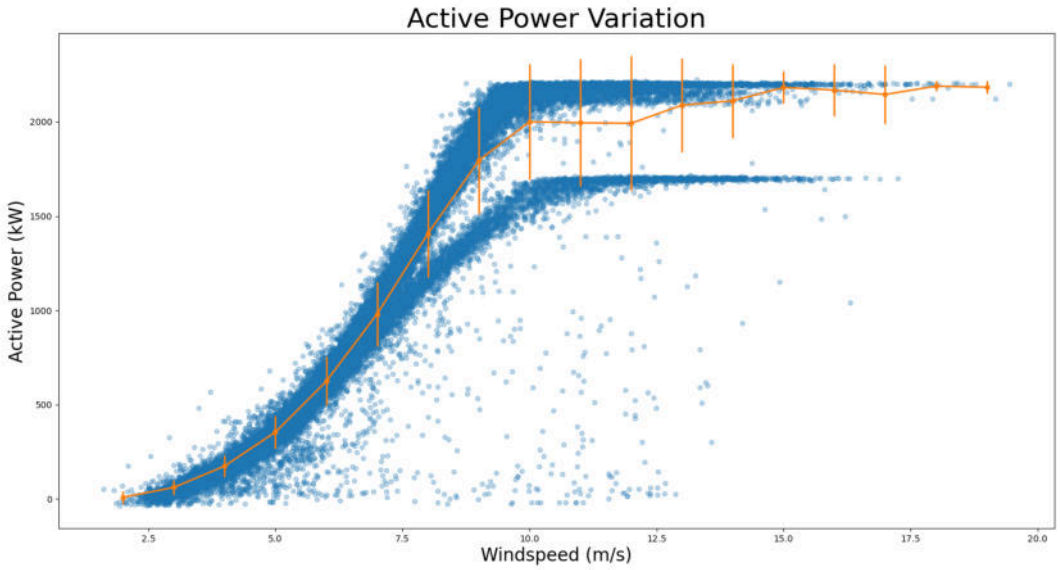
```
df['sigma'] = df['TM_ActivePower']['std']/np.sqrt(df['TM_ActivePower']['count'])
df.sample(5)
```

Out[24]:

	windspeed_bin	TM_ActivePower			sigma
		mean	std	count	
6	8.0	1406.175458	230.572994	3734	3.773300
3	5.0	353.835411	87.772496	5346	1.200450
16	18.0	2189.507667	27.505535	15	7.101899
11	13.0	2088.406522	248.397507	1330	6.811165
5	7.0	980.044995	171.544224	4399	2.586420

In [25]:

```
#matlibplot of power vs windspeed

df_temp = df_10min[df_10min['AssetName']== 'ORI-Ve-42']
df_temp = df_temp[df_temp['State_Fault_Interpolated']== 2.0]
df_temp = df_temp[df_temp['TM_SetPoint']== 500]

xp = df_temp['Windspeed_Adjusted']
yp = df_temp['TM_ActivePower']
#sns.set(font_scale=1.2, font='DejaVu Sans')
fig, ax = plt.subplots(1,1, figsize=(20, 10))
p = ax.plot(xp, yp, marker='o', linestyle='none', markersize=5, alpha=0.3, label='all')
xp = df['windspeed_bin']
yp = df['TM_ActivePower']['mean']
err = df['TM_ActivePower']['std']
p = ax.errorbar(xp, yp,err, marker='o', linestyle='-', markersize=5, linewidth=2, label=r'mean $\pm\sigma$', zorder=3)
p = ax.set_xlabel('Windspeed (m/s)', fontsize=20)
p = ax.set_ylabel('Active Power (kW)', fontsize=20)
p = ax.set_title('Active Power Variation', fontsize=30)
```



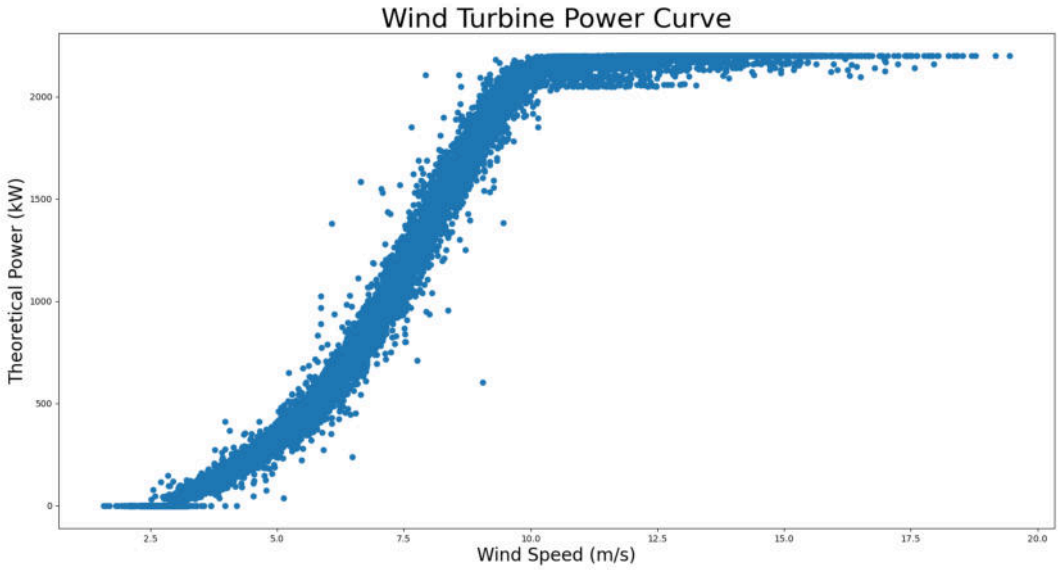
In [26]:

```
# df = df_10min[df_10min['AssetName']== 'ORI-Ve-42']
# df = df[df['State_Fault_Interpolated']== 2.0]
# df = df[df['TM_SetPoint']== 500]

# fig = plt.figure(figsize=(20,10))
# ax = plt.axes(projection='3d')

# X = df['Windspeed_Adjusted']
# Y= df['TM_ActivePower']
# Z = df['TM_AmbientTemperature_Celsius']

# # ax = plt.axes(projection='3d')
# # z = np.linspace(0, 1, 100)
# # x = z * np.sin(20 * z)
# # y = z * np.cos(20 * z)
# # c = x + y
```

```
# # ax.scatter(x, y, z, c=c)
# # ax.set_title('3d Scatter plot')
# # plt.show()

# ax.scatter(X, Y, Z, cmap='viridis', edgecolor='none')

# p = ax.set_xlabel('Windspeed (m/s)', fontsize=20)
# p = ax.set_ylabel('Active Power (kW)', fontsize=20)
# p = ax.set_zlabel('Ambient Temperature (C)', fontsize=20)
# p = ax.set_title('Active Power Variation', fontsize=30)
# ax.set_title('Active Power vs Windspeed vs Ambient Temp')
```

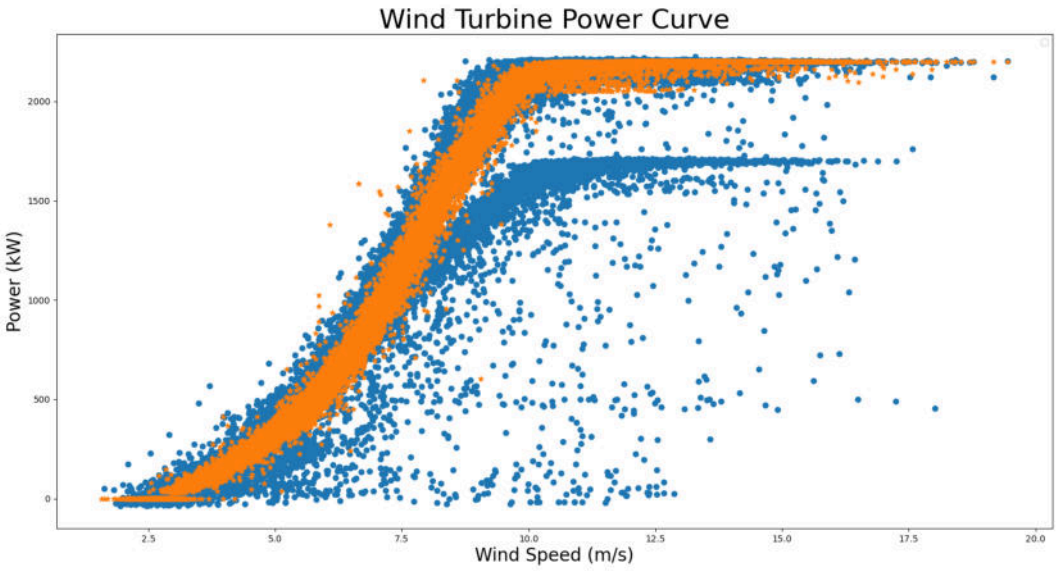
Plot Theoretical Power Curve

In [27]:
```
df = df_10min[df_10min['AssetName']== 'ORI-Ve-42']
df = df[df['State_Fault_Interpolated']== 2.0]

fig = plt.figure(figsize=(20,10))

plt.scatter(x = df['Windspeed_Adjusted'],y=df['TM_PossiblePower'])

plt.xlabel("Wind Speed (m/s)", fontsize=20)
plt.ylabel("Theoretical Power (kW)", fontsize=20)
plt.title("Wind Turbine Power Curve", fontsize=30)

plt.show()
```



In [28]:
```
df = df_10min[df_10min['AssetName']== 'ORI-Ve-42']
df = df[df['State_Fault_Interpolated']== 2.0]

fig = plt.figure(figsize=(20,10))

plt.scatter(x = df['Windspeed_Adjusted'],y=df['TM_ActivePower'])
plt.scatter(x = df['Windspeed_Adjusted'],y=df['TM_PossiblePower'], marker = '*')

plt.xlabel("Wind Speed (m/s)", fontsize=20)
plt.ylabel("Power (kW)", fontsize=20)
plt.title("Wind Turbine Power Curve", fontsize=30)

plt.legend()
plt.show()
```

```
No artists with labels found to put in legend.  Note that artists whose label start with an underscore are ignored when legend() is called with no argument.
```



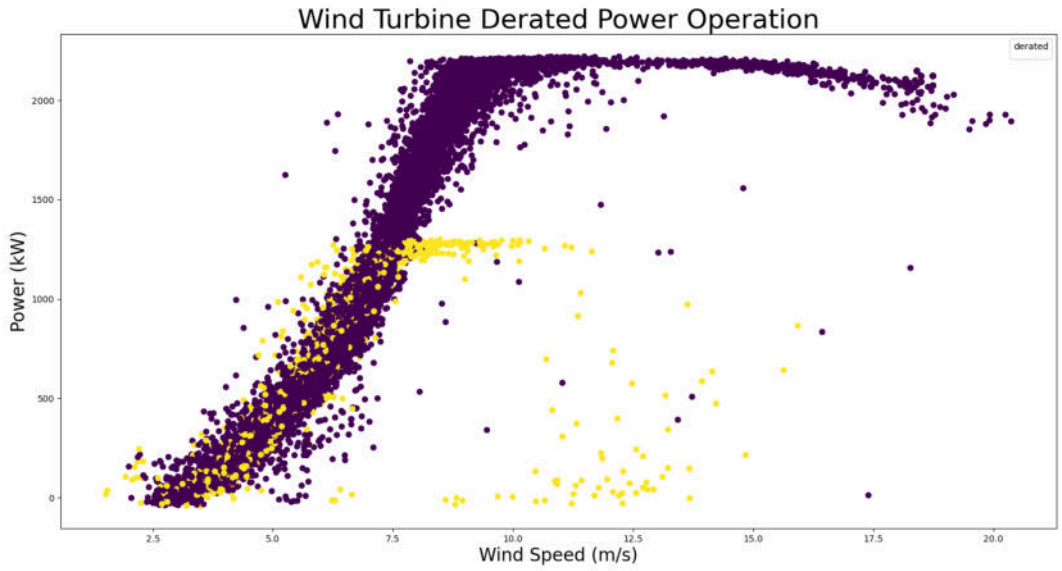
Plot Derated Operating Condition & Residual Power

The difference between the measured actual power and the power expected based on the power curve is called the power residual.
Power_residual = Power_actual – Power_expected

Wind Turbine 1

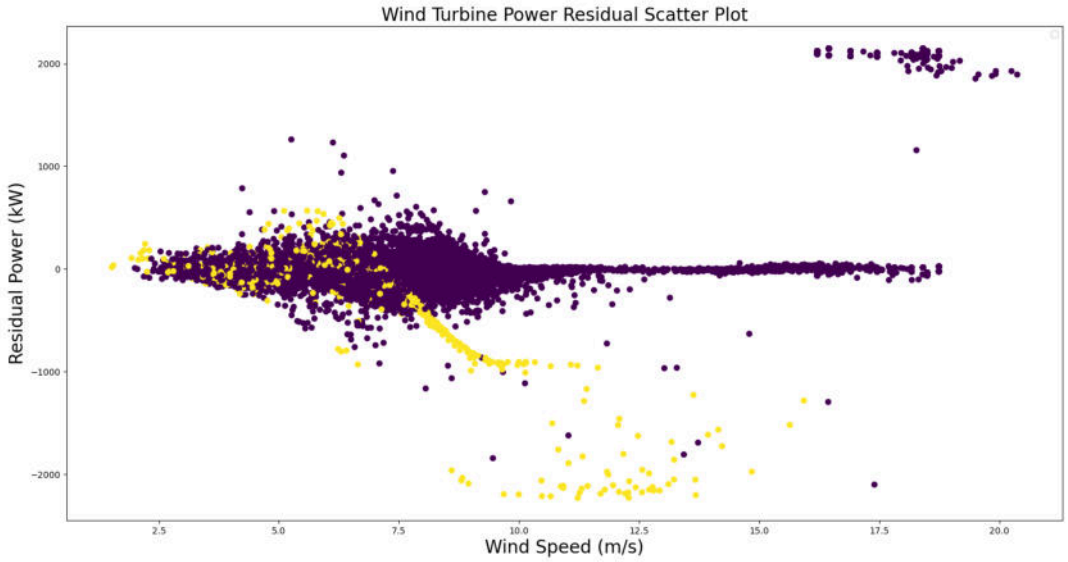
In [29]:
```
df = df_10min[df_10min['AssetName']== 'ORI-Ve-178']
df = df[df['State_Fault_Interpolated']== 2.0]
df = df[df['TM_SetPoint']== 500]

split_date = '2023-02-20 00:00:00'
df = df.loc[df.index <= split_date].copy()

split_date = '2023-01-18 00:00:00'
df['is_derated'] = np.where(df.index >= split_date, 'Yes', 'No')

fig = plt.figure(figsize=(20,10))

plt.scatter(x = df['Windspeed_Adjusted'],y=df['TM_ActivePower'], c=df.is_derated.astype('category').cat.codes)

# plt.scatter(x = df['Windspeed_Adjusted'],y=df['TM_PossiblePower'])

plt.xlabel("Wind Speed (m/s)", fontsize=20)
plt.ylabel("Power (kW)", fontsize=20)
plt.title("Wind Turbine Derated Power Operation", fontsize=30)

plt.legend(title="derated")

plt.show()
```

```
No artists with labels found to put in legend.  Note that artists whose label start with an underscore are ignored when legend() is called with no argument.
```



In [30]:
```
df["Residual_Power"] = df["TM_ActivePower"] - df["TM_PossiblePower"]

fig = plt.figure(figsize=(20,10))

plt.scatter(x = df['Windspeed_Adjusted'],y=df['Residual_Power'], c=df.is_derated.astype('category').cat.codes)

plt.xlabel("Wind Speed (m/s)", fontsize=20)
plt.ylabel("Residual Power (kW)", fontsize=20)
plt.title("Wind Turbine Power Residual Scatter Plot", fontsize=20)

plt.legend()
plt.show()
```

```
No artists with labels found to put in legend.  Note that artists whose label start with an underscore are ignored when legend() is called with no argument.
```

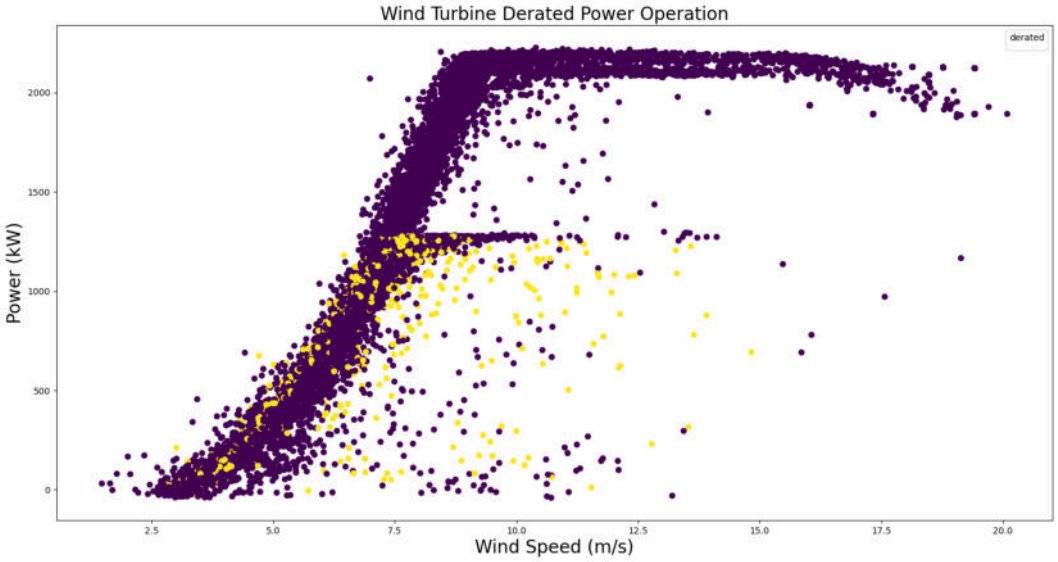


Wind Turbine 2

In [31]:
```
df = df_10min[df_10min['AssetName']== 'ORI-Ve-87']
df = df[df['State_Fault_Interpolated']== 2.0]
df = df[df['TM_SetPoint']== 500]

split_date = '2023-03-05 00:00:00'
df = df.loc[df.index <= split_date].copy()
```

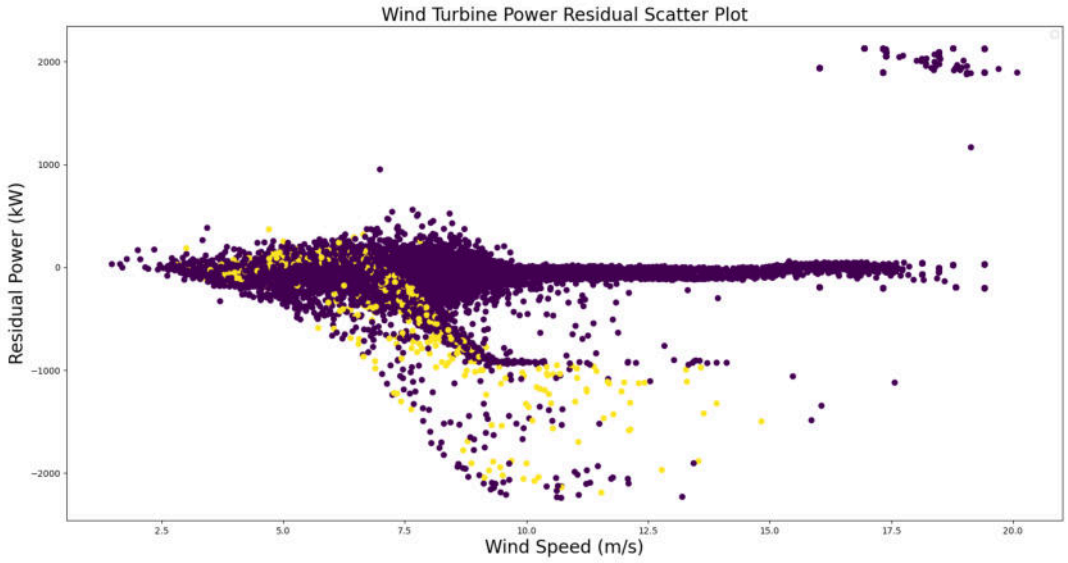
```
split_date = '2023-02-5 00:00:00'
df['is_derated'] = np.where(df.index >= split_date, 'Yes', 'No')

fig = plt.figure(figsize=(20,10))

plt.scatter(x = df['Windspeed_Adjusted'],y=df['TM_ActivePower'], c=df.is_derated.astype('category').cat.codes)

# plt.scatter(x = df['Windspeed_Adjusted'],y=df['TM_PossiblePower'])

plt.xlabel("Wind Speed (m/s)", fontsize=20)
plt.ylabel("Power (kW)", fontsize=20)
plt.title("Wind Turbine Derated Power Operation", fontsize=20)

plt.legend(title="derated")

plt.show()
```

```
No artists with labels found to put in legend.  Note that artists whose label start with an underscore are ignored when legend() is called with no argument.
```



In [32]:
```
df["Residual_Power"] = df["TM_ActivePower"] - df["TM_PossiblePower"]

fig = plt.figure(figsize=(20,10))

plt.scatter(x = df['Windspeed_Adjusted'],y=df['Residual_Power'], c=df.is_derated.astype('category').cat.codes)

plt.xlabel("Wind Speed (m/s)", fontsize=20)
plt.ylabel("Residual Power (kW)", fontsize=20)
plt.title("Wind Turbine Power Residual Scatter Plot", fontsize=20)

plt.legend()
plt.show()
```

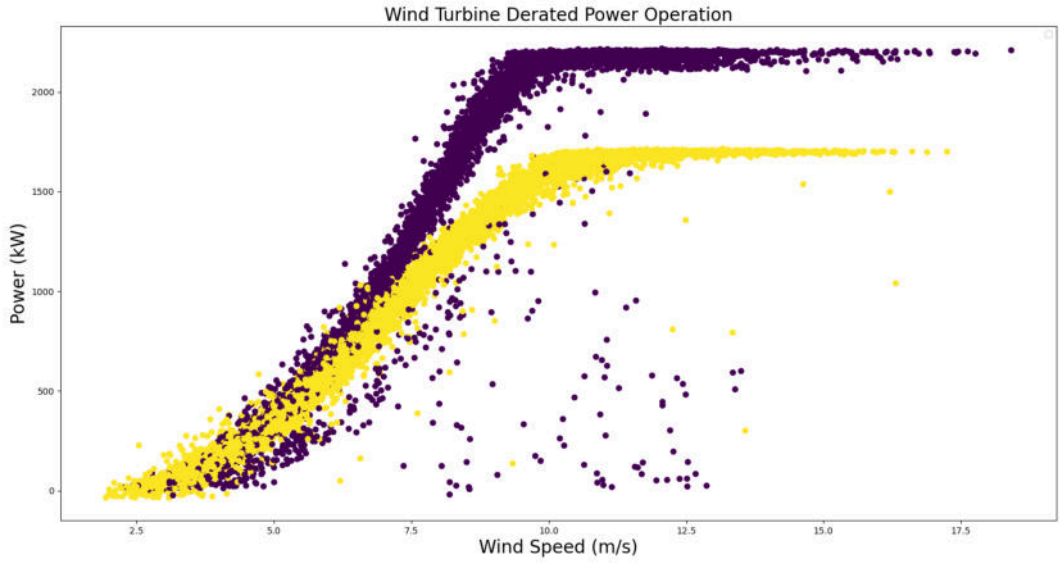
```
No artists with labels found to put in legend.  Note that artists whose label start with an underscore are ignored when legend() is called with no argument.
```



Wind Turbine 3

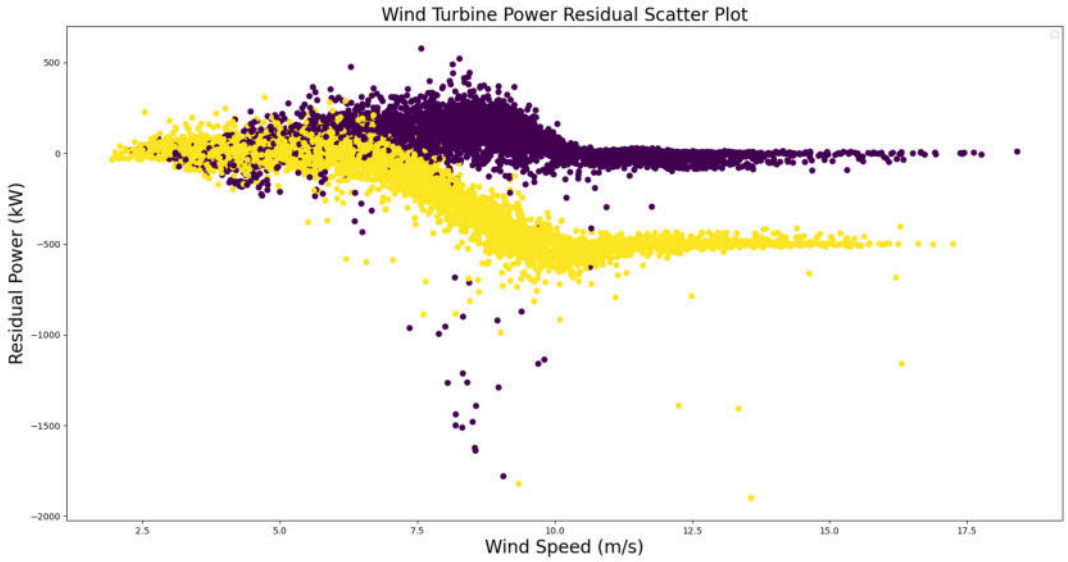
In [33]:
```
df = df_10min[df_10min['AssetName']== 'ORI-Ve-42']
df = df[df['State_Fault_Interpolated']== 2.0]
df = df[df['TM_SetPoint']== 500]

split_date = '2023-02-01 00:00:00'
df = df.loc[df.index >= split_date].copy()

split_date = '2023-03-25 00:00:00'
df['is_derated'] = np.where(df.index >= split_date, 'Yes', 'No')

fig = plt.figure(figsize=(20,10))

plt.scatter(x = df['Windspeed_Adjusted'],y=df['TM_ActivePower'],
            c=df.is_derated.astype('category').cat.codes)

plt.xlabel("Wind Speed (m/s)", fontsize=20)
plt.ylabel("Power (kW)", fontsize=20)
plt.title("Wind Turbine Derated Power Operation", fontsize=20)

plt.legend()

plt.show()
```

```
No artists with labels found to put in legend.  Note that artists whose label start with an underscore are ignored when legend() is called with no argument.
```



In [34]:
```
df["Residual_Power"] = df["TM_ActivePower"] - df["TM_PossiblePower"]

fig = plt.figure(figsize=(20,10))

plt.scatter(x = df['Windspeed_Adjusted'],y=df['Residual_Power'], c=df.is_derated.astype('category').cat.codes)

plt.xlabel("Wind Speed (m/s)", fontsize=20)
plt.ylabel("Residual Power (kW)", fontsize=20)
plt.title("Wind Turbine Power Residual Scatter Plot", fontsize=20)

plt.legend()
plt.show()
```

```
No artists with labels found to put in legend.  Note that artists whose label start with an underscore are ignored when legend() is called with no argument.
```

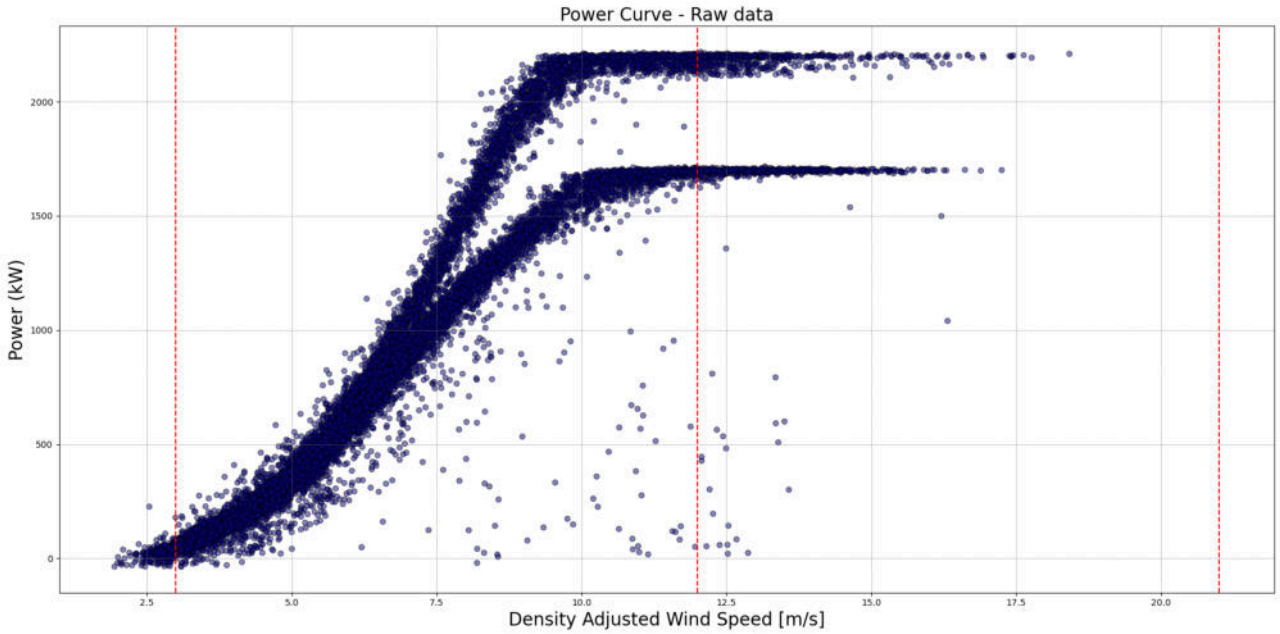


In [35]:
```
def powerCurve(data,wtn,ws_cut_in,ws_rated,ws_cut_out):
    '''
    Plot wind farm power curve with average wind speed and power values.

    Variables
    ----------
    data : DataFrame
        Turbine wind speed and power data to plot.
    wtn : int
```

```
        Number of wind turbines.
    '''



    # List of wind speed column names
#     windSpeed_cols = ['Windspeed_Adjusted' + str(wt).zfill(2) for wt in range(1,wtn+1)]  # turbine wind speed
    # List of wind speed column names
    windSpeed_cols = ['Windspeed_Adjusted'.zfill(2) for wt in range(1,wtn+1)]  # turbine wind speed
    print(windSpeed_cols)
    # List of power column names
#     power_cols = ['TM_ActivePower' + str(wt).zfill(2) for wt in range(1,wtn+1)]  # turbine active power
    power_cols = ['TM_ActivePower'.zfill(2) for wt in range(1,wtn+1)]  # turbine active power

    # - - - Calculate wind farm statistics - - -
    # Average wind speed
    data['windSpeed_avg'] = data[windSpeed_cols].mean(axis=1, skipna=False)
#     print(data['windSpeed_avg'])
    # Standard deviation of wind speed
    data['windSpeed_std'] = data[windSpeed_cols].std(ddof=0, axis=1, skipna=False)
#     print(data['windSpeed_std'])
    # Average power output
    data['power_avg'] = data[power_cols].mean(axis=1, skipna=False)

#     data.info()
    # Power curve

    fig = plt.figure(figsize=(20,10))
    plt.scatter(data['windSpeed_avg'], data['power_avg'], c=data['windSpeed_std'], cmap='plasma', edgecolor='k', alpha=0.5)
    # dots are coloured by wind speed standard deviation
    plt.xlabel('Density Adjusted Wind Speed [m/s]', fontsize=20); plt.ylabel('Power (kW)', fontsize=20)
    plt.title('Power Curve - Raw data', fontsize=20)
    plt.grid(ls='--', lw=0.5, c='grey')
    plt.tight_layout()
    plt.axvline(ws_cut_in, ls='--', lw=1.5, c='r')
    plt.axvline(ws_rated, ls='--', lw=1.5, c='r')
    plt.axvline(ws_cut_out, ls='--', lw=1.5, c='r')
    plt.show()
```

In [36]:
```
df = df[['TM_ActivePower', 'Windspeed_Adjusted']].dropna()

# Define power curve's wind speeds
ws_cut_in=3.0  # cut-in wind speed
ws_rated=12.0  # rated wind speed
ws_cut_out=21.0  # cut-out wind speed

fig = plt.figure(figsize=(20,10))
# Draw exploratory power curve with average wind farm values
powerCurve(df, 1, ws_cut_in,ws_rated,ws_cut_out)



# Add lines to the power curve plot
```

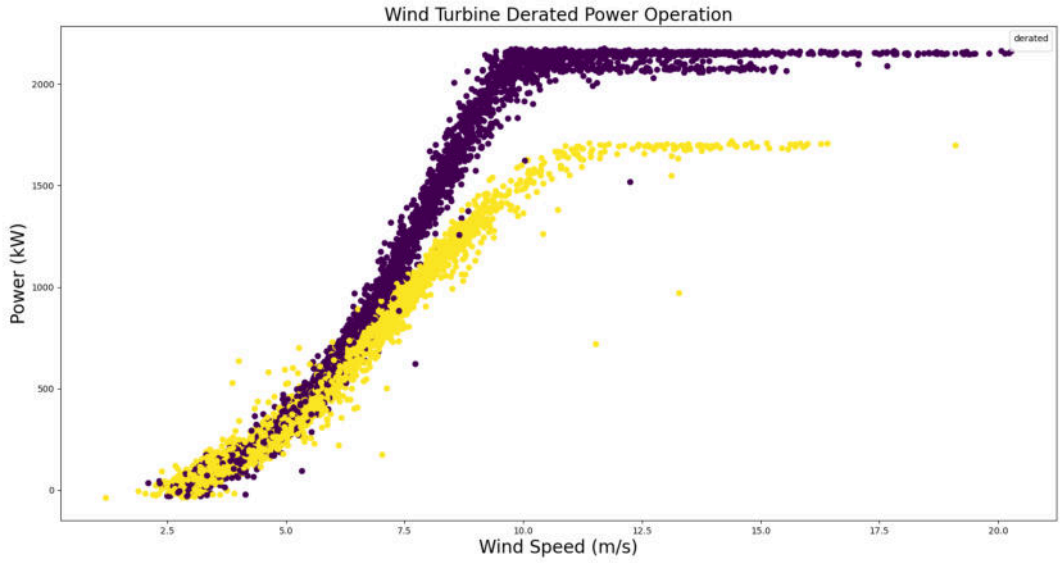
```
['Windspeed_Adjusted']
<Figure size 2000x1000 with 0 Axes>
```



Wind Turbine 4

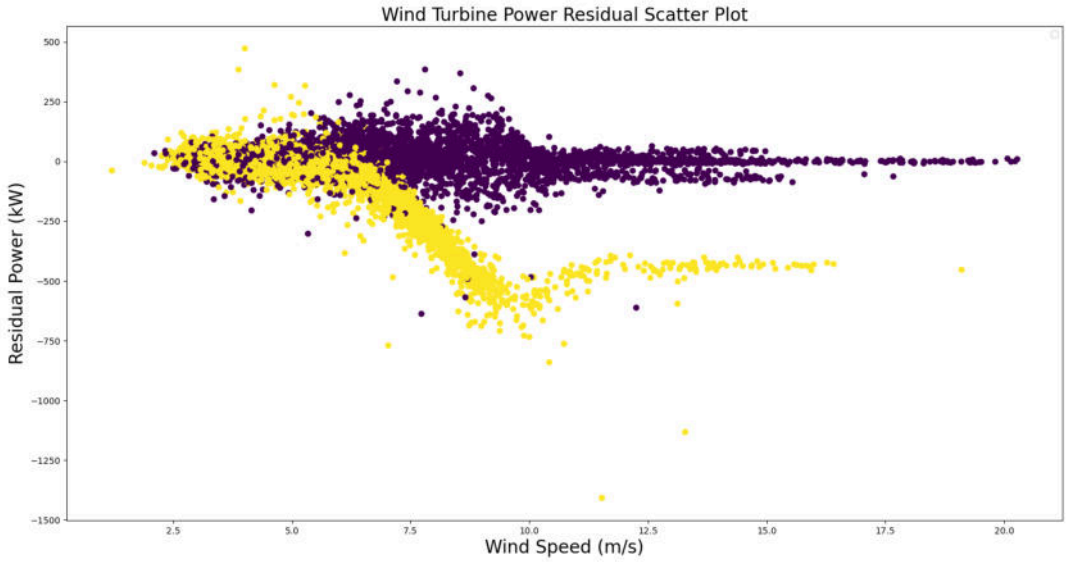
In [37]:
```
df = df_10min[df_10min['AssetName']== 'ORI-Ve-24']
df = df[df['State_Fault_Interpolated']== 2.0]
df = df[df['TM_SetPoint']== 500]

split_date = '2023-04-10 00:00:00'
df = df.loc[df.index >= split_date].copy()

split_date = '2023-05-20 00:00:00'
df['is_derated'] = np.where(df.index >= split_date, 'Yes', 'No')

fig = plt.figure(figsize=(20,10))

plt.scatter(x = df['Windspeed_Adjusted'],y=df['TM_ActivePower'], c=df.is_derated.astype('category').cat.codes)

plt.xlabel("Wind Speed (m/s)", fontsize=20)
plt.ylabel("Power (kW)", fontsize=20)
plt.title("Wind Turbine Derated Power Operation", fontsize=20)

plt.legend(title="derated")

plt.show()
```

```
No artists with labels found to put in legend.  Note that artists whose label start with an underscore are ignored when legend() is called with no argument.
```



In [38]:
```
df["Residual_Power"] = df["TM_ActivePower"] - df["TM_PossiblePower"]

fig = plt.figure(figsize=(20,10))

plt.scatter(x = df['Windspeed_Adjusted'],y=df['Residual_Power'], c=df.is_derated.astype('category').cat.codes)

plt.xlabel("Wind Speed (m/s)", fontsize=20)
plt.ylabel("Residual Power (kW)", fontsize=20)
plt.title("Wind Turbine Power Residual Scatter Plot", fontsize=20)

plt.legend()
plt.show()
```

```
No artists with labels found to put in legend.  Note that artists whose label start with an underscore are ignored when legend() is called with no argument.
```

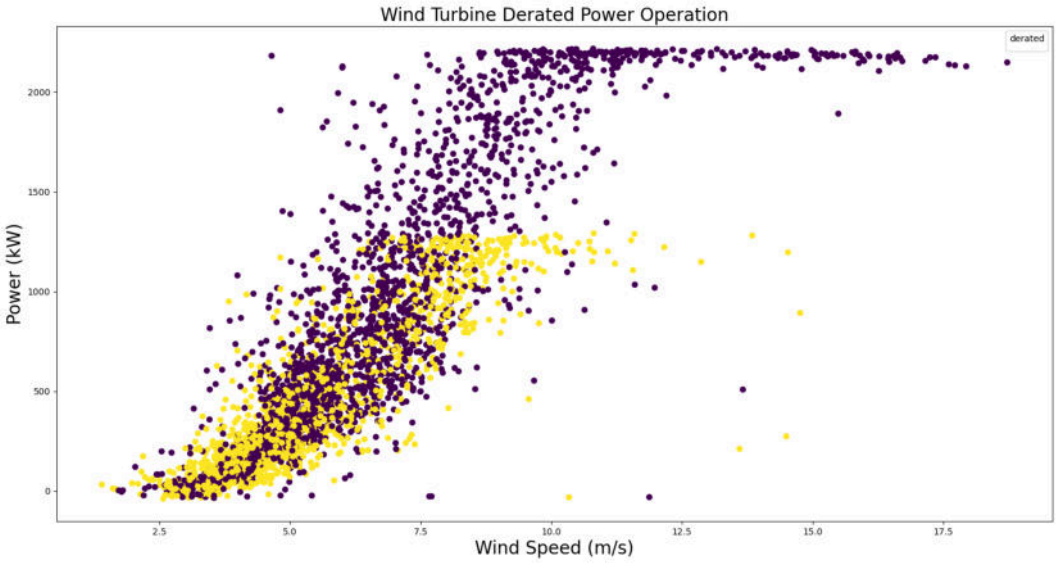


Wind Turbine 5

In [39]:
```
df = df_10min[df_10min['AssetName']== 'ORI-Ve-181']
df = df[df['State_Fault_Interpolated']== 2.0]
df = df[df['TM_SetPoint']== 500]

split_date = '2023-05-01 00:00:00'
df = df.loc[df.index >= split_date].copy()
```

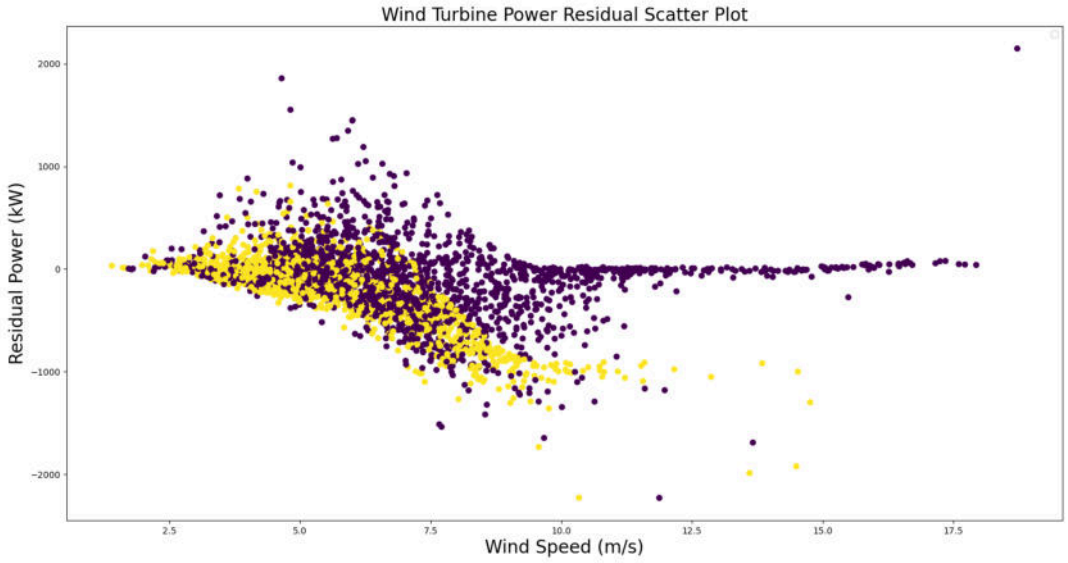
```
split_date = '2023-05-30 00:00:00'
df['is_derated'] = np.where(df.index >= split_date, 'Yes', 'No')

fig = plt.figure(figsize=(20,10))

plt.scatter(x = df['Windspeed_Adjusted'],y=df['TM_ActivePower'], c=df.is_derated.astype('category').cat.codes)

plt.xlabel("Wind Speed (m/s)", fontsize=20)
plt.ylabel("Power (kW)", fontsize=20)
plt.title("Wind Turbine Derated Power Operation", fontsize=20)

plt.legend(title="derated")

plt.show()
```

```
No artists with labels found to put in legend.  Note that artists whose label start with an underscore are ignored when legend() is called with no argument.
```



In [40]:
```
df["Residual_Power"] = df["TM_ActivePower"] - df["TM_PossiblePower"]

fig = plt.figure(figsize=(20,10))

plt.scatter(x = df['Windspeed_Adjusted'],y=df['Residual_Power'], c=df.is_derated.astype('category').cat.codes)

plt.xlabel("Wind Speed (m/s)", fontsize=20)
plt.ylabel("Residual Power (kW)", fontsize=20)
plt.title("Wind Turbine Power Residual Scatter Plot", fontsize=20)

plt.legend()
plt.show()
```

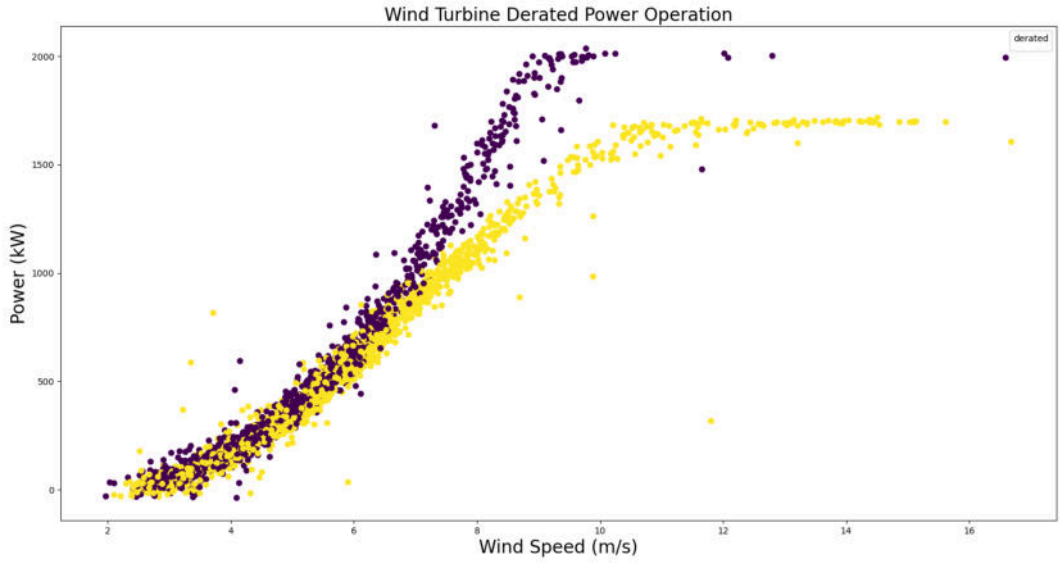
```
No artists with labels found to put in legend.  Note that artists whose label start with an underscore are ignored when legend() is called with no argument.
```



Wind Turbine 6

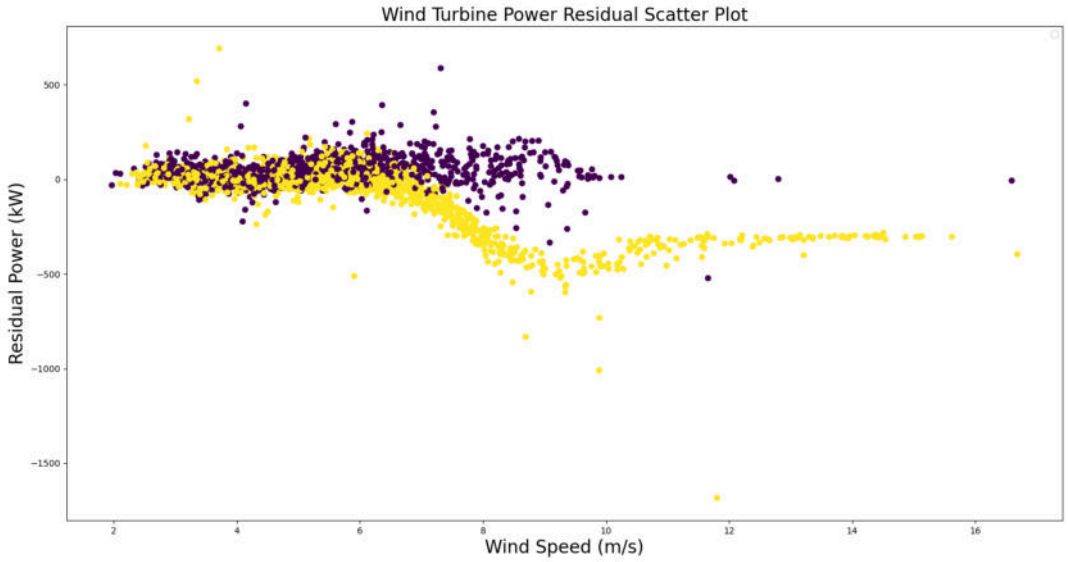
In [41]:
```
df = df_10min[df_10min['AssetName']== 'ORI-Ve-38']
df = df[df['State_Fault_Interpolated']== 2.0]
df = df[df['TM_SetPoint']== 500]

split_date = '2023-06-01 00:00:00'
df = df.loc[df.index >= split_date].copy()

split_date = '2023-06-15 00:00:00'
df['is_derated'] = np.where(df.index >= split_date, 'Yes', 'No')

fig = plt.figure(figsize=(20,10))

plt.scatter(x = df['Windspeed_Adjusted'],y=df['TM_ActivePower'], c=df.is_derated.astype('category').cat.codes)

plt.xlabel("Wind Speed (m/s)", fontsize=20)
plt.ylabel("Power (kW)", fontsize=20)
plt.title("Wind Turbine Derated Power Operation", fontsize=20)

plt.legend(title="derated")

plt.show()
```

```
No artists with labels found to put in legend.  Note that artists whose label start with an underscore are ignored when legend() is called with no argument.
```



In [42]:
```
df["Residual_Power"] = df["TM_ActivePower"] - df["TM_PossiblePower"]

fig = plt.figure(figsize=(20,10))

plt.scatter(x = df['Windspeed_Adjusted'],y=df['Residual_Power'], c=df.is_derated.astype('category').cat.codes)

plt.xlabel("Wind Speed (m/s)", fontsize=20)
plt.ylabel("Residual Power (kW)", fontsize=20)
plt.title("Wind Turbine Power Residual Scatter Plot", fontsize=20)

plt.legend()
plt.show()
```

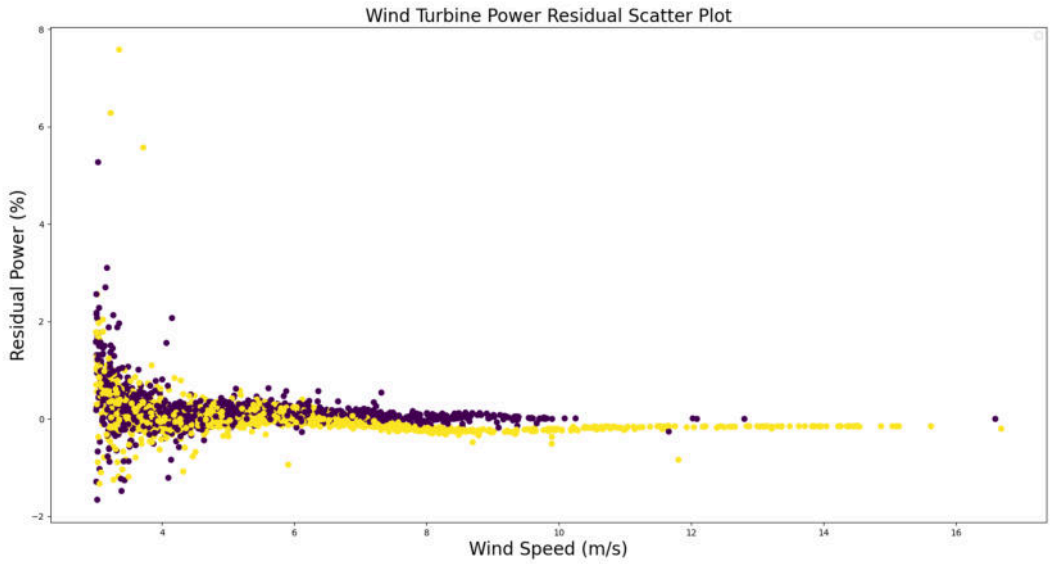
```
No artists with labels found to put in legend.  Note that artists whose label start with an underscore are ignored when legend() is called with no argument.
```



In [43]:
```
df["Residual_Power_perc"] = round((df["TM_ActivePower"] - df["TM_PossiblePower"])/df["TM_PossiblePower"],2)

fig = plt.figure(figsize=(20,10))

plt.scatter(x = df['Windspeed_Adjusted'],y=df['Residual_Power_perc'], c=df.is_derated.astype('category').cat.codes)

plt.xlabel("Wind Speed (m/s)", fontsize=20)
plt.ylabel("Residual Power (%)", fontsize=20)
plt.title("Wind Turbine Power Residual Scatter Plot", fontsize=20)
```

```
plt.legend()
plt.show()
```

```
No artists with labels found to put in legend.  Note that artists whose label start with an underscore are ignored when legend() is called with no argument.
```



Data Cleaning

Error characteristics of power curve modeling

Typically, wind data that's considered normal can be explained through a power curve. However, when dealing with outdoor settings, various complex geographical elements, ever-changing weather conditions, and the wind turbine's inherent traits contribute to a multitude of anomalies. These anomalies significantly deviate from the anticipated wind power levels as indicated by our power data. To showcase the inaccuracies inherent in power curve modeling, we delve into three distinct categories of exceptional cases.

- The initial form of anomaly occurs when the wind power value approaches zero despite the corresponding wind speed falling between the cut-in and cut-out wind speeds. Instances such as wind turbine maintenance or intentional wind reduction such as site curtailment can lead to these anomalies
- The second anomaly category resembles the first, with power values deviating significantly from the expected power curve data, though not reaching zero. Causes for these anomalies encompass wind reduction tactics, along with blade contamination from dirt, insects, or ice, as well as blade pitch malfunctions and other factors
- The last anomaly type involves wind power values surpassing the turbine's physical limitations mostly due to error in sensors such as Anemometer

It's apparent that the majority of anomalies are identified and eliminated from the original wind data. Following the removal of these anomalies, the refined wind data are employed for training power curve models.

A wind turbine's power curve essentially encapsulates its performance. This curve illustrates how wind speed correlates with the turbine's power output. Developing a model for a wind turbine's power curve assists in monitoring its performance and forecasting power generation.

In [44]:
```
whisker_length = 1
capillarity = 0.25
turbine_df_clean = pd.DataFrame(columns = ['Windspeed_Adjusted', 'TM_ActivePower'])

for i in np.linspace(start = 0, stop = 30, endpoint = False, num = int(30 / capillarity)):
    lower_bound = i
    #print("Lower Bound" ,lower_bound)
    upper_bound = i + capillarity
    #print("Upper Bound" ,upper_bound)

    if upper_bound <= 3: #Windspeed Zone Selection
        data_interval =  df[((df.Windspeed_Adjusted < upper_bound) & (df.Windspeed_Adjusted >= lower_bound))]
```

```
    elif upper_bound > 3 and upper_bound <= 15:
        data_interval =  df[((df.Windspeed_Adjusted < upper_bound) & (df.Windspeed_Adjusted >= lower_bound)) & (df.TM_Activ
    elif upper_bound > 15:
        data_interval =  df[((df.Windspeed_Adjusted < upper_bound) & (df.Windspeed_Adjusted >= lower_bound)) & (df.TM_Activ

    quantiles = data_interval.quantile([.25, .75], axis = 'rows') # calculation of the quantiles based on boxplot
    boxplot_h = quantiles.TM_ActivePower[0.75] - quantiles.TM_ActivePower[0.25]
    boxplot_ends = [quantiles.TM_ActivePower[0.25] - whisker_length*boxplot_h, quantiles.TM_ActivePower[0.75] + whisker_leng
    data_interval = data_interval[(data_interval.TM_ActivePower < boxplot_ends[1]) & (data_interval.TM_ActivePower > boxplot
    turbine_df_clean = turbine_df_clean.append(data_interval, ignore_index = False)
#     df = pd.concat([turbine_df_clean, pd.DataFrame([data_interval])], ignore_index=False)
```

In [45]:

```
def clean_turbine(data_raw, wtn, ws_cut_in, ws_rated, ws_cut_out, derate_cutoff_date,
                  k_up=1.5, k_low=1.5, anomalous=False):
    '''
    Cleans wind turbine data.

    Variables
    ----------
    data_raw : DataFrame
        Raw data to be cleaned.
    wtn : int
        Number of wind turbines.
    ws_cut_in : float
        Cut-in wind speed in m/s.
    ws_rated : float
        Rated wind speed in m/s.
    ws_cut_out : float
        Cut-out wind speed in m/s.
    k_up : float, default=1.5
        Multiplier of IQR to define upper threshold for outlier detection.
    k_low : float, default=1.5
        Multiplier of IQR to define lower threshold for outlier detection.
    anomalous : bool, default False
        If True, anomalous (flagged) periods are kept in the data set.

    Returns
    ----------
    data : pandas.DataFrame
        Cleaned data.

    Notes
    -----
    Wind speed values are in m/s.
    Power values are normalised by rated power.

    '''

    # List of wind speed column names
    windSpeed_cols = ['windSpeed_wt' + str(wt).zfill(2) for wt in range(1,wtn+1)]  # turbine wind speed

    # List of power column names
    power_cols = ['power_wt' + str(wt).zfill(2) for wt in range(1,wtn+1)]  # turbine active power

    # - - - Calculate wind farm statistics - - -
    # Average wind speed
    data_raw['windSpeed_avg'] = data_raw[windSpeed_cols].mean(axis=1, skipna=False)
    # Standard deviation of wind speed
    data_raw['windSpeed_std'] = data_raw[windSpeed_cols].std(axis=1, skipna=False)
    # Average power output
    data_raw['power_avg'] = data_raw[power_cols].mean(axis=1, skipna=False)


    # = = = = = DATA CLEANING = = = = =
    data = data_raw.copy()


    # = = = = = COMPLETENESS of data = = = = =
    print("\n########################## MISSING DATA ######################### \n")
    print("- - - - - Wind speed missing values - - - - -")
    for wt,ws in zip(range(1, wtn+1), windSpeed_cols):
        print("Turbine " + str(wt).zfill(2) + ": " +
              str(data[ws].isnull().sum()) + " (" +
              str(round(data[ws].isnull().sum()/float(len(data[ws]))*100, 1)) + "%)")

    print("\n - - - - - Power missing values - - - - -")
    for wt,pw in zip(range(1, wtn+1), power_cols):
        print("Turbine " + str(wt).zfill(2) + ": " +
              str(data[pw].isnull().sum()) + " (" +
```

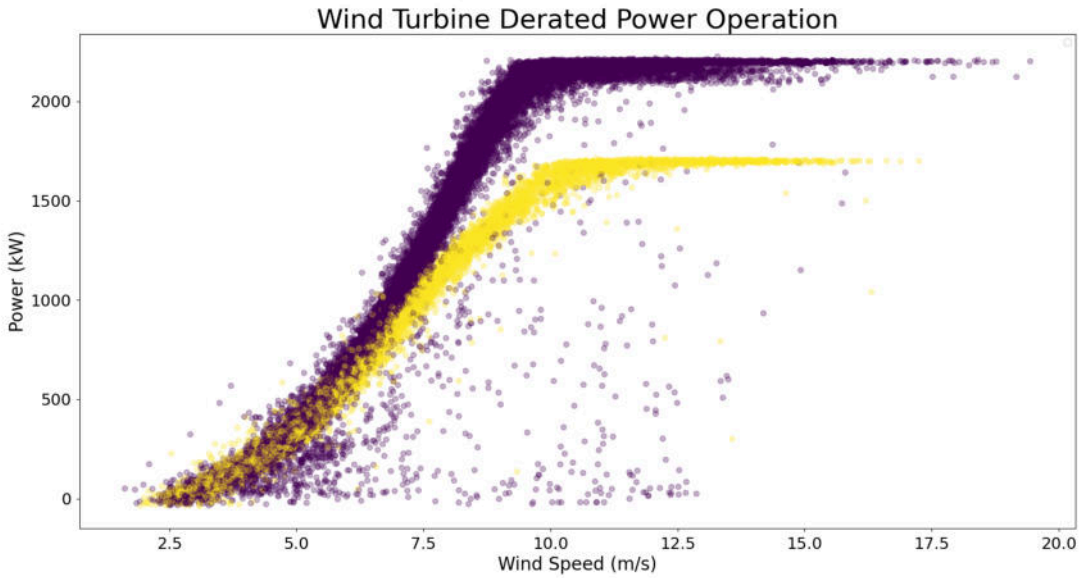
```
          str(round(data[pw].isnull().sum()/float(len(data[pw]))*100, 1)) + "%)")
print("\n#########################################################################")


# = = = = = VALIDITY of data = = = = =
# - - - Univariate extreme values - - -
# Wind speed
for ws in windSpeed_cols:
    data[ws].where(
        (data[ws]>=0) & (data[ws]<40)
        )

# Power
for pw in power_cols:
    data[pw].where(
        (data[pw]>=-0.02) & (data[pw]<1.01)
        )


# - - - Bivariate extreme values - - -
# Rules: remove erroneous data, flag anomalous data.

# Create a flag column for each turbine and set initial value to 0.
# When the data is anomalous, the flag value is set > 0.
flag_cols = ['flag_wt' + str(wt).zfill(2) for wt in range(1,wtn+1)]
for fl in flag_cols:
    data[fl] = 0

# 1. Remove instances of high power output for wind speed in [0, ws_cut_in]
for ws,pw in zip(windSpeed_cols, power_cols):
    data[pw] = data[pw].mask(
        (data[ws] >= 0) & (data[ws] < ws_cut_in) &
        (data[pw] > 0.04)
        )

# 2. Remove instances of non-zero power for wind speed > ws_cut_out + 2
for ws,pw in zip(windSpeed_cols, power_cols):
    data[pw] = data[pw].mask(
        (data[ws] >= ws_cut_out+2) &
        (data[pw] > 0)
        )

# 3. Flag instances of zero power output for wind speed in [ws_cut_in+2,
#    ws_cut_out-2]
# Create a list of "temporary" columns for partially-clean power values
power_cols_tmp = ['power_wt' + str(wt).zfill(2) + "_tmp" for wt in range(1,wtn+1)]
for ws,pw,fl,pw_tmp in zip(windSpeed_cols, power_cols, flag_cols, power_cols_tmp):
    data[fl] = data[fl].mask(
        (data[ws] > ws_cut_in+2) & (data[ws] < ws_cut_out-2) &
        (data[pw] < 0.005),
        data[fl]+1
        )
    data[pw_tmp] = data[pw].mask(data[fl] > 0)


# 4. Flag instances of low power output (<99.5% of P_nom) for wind speed in
#    [ws_rated+2, ws_cut_out-2]
for ws,pw,fl in zip(windSpeed_cols, power_cols, flag_cols):
    data[fl] = data[fl].mask(
        (data[ws] > ws_rated+2) & (data[ws] < ws_cut_out-2) &
        (data[pw] < 0.995),
        data[fl]+1
        )


# 5. Remove instances of high power output for wind speed in [ws_cut_in+0.5, ws_rated]
bin_width = 0.05  # [m/s]
for ws,pw,fl,pw_tmp in zip(windSpeed_cols, power_cols, flag_cols, power_cols_tmp):
    # 5.1 Group by wind speed values, bin width = bin_width
    grouped = data.groupby(
        pd.cut(data[ws], np.arange(ws_cut_in+0.5, ws_rated, bin_width))
        )
    for key,df in grouped:
        # 5.2 Calculate outlier threshold (Q3 + 2.5*IQR) for each group
        q25, q75 = np.percentile(df[pw_tmp].dropna(), [25,75])
        iqr = q75 - q25
        thresh_up = q75 + k_up*iqr

        # 5.3 Remove instances where power is above the threshold
```

```
            data[pw] = data[pw].mask(
                (data[ws] > key.left) & (data[ws] <= key.right) &
                (data[pw] > thresh_up)
                )


    # 6. Flag instances of low power output for wind speed in [ws_cut_in+0.5, ws_rated+2.0]
    bin_width = 0.05  # [m/s]
    for ws,pw,fl,pw_tmp in zip(windSpeed_cols, power_cols, flag_cols, power_cols_tmp):
        # 6.1 Group by wind speed values, bin width=bin_width.
        grouped = data.groupby(
            pd.cut(data[ws], np.arange(ws_cut_in+0.5, ws_rated+2.0+bin_width, bin_width))
            )

        for key,df in grouped:
            # 6.2 Calculate outlier threshold (Q1 - 2.5*IQR) for each group
            q25, q75 = np.percentile(df[pw_tmp].dropna(), [25,75])
            iqr = q75 - q25
            thresh_low = q25 - k_low*iqr

            # 6.3 Flag instances where power output is below the threshold
            data[fl] = data[fl].mask(
                (data[ws] > key.left) & (data[ws] <= key.right) &
                (data[pw] < thresh_low),
                data[fl]+1
                )


    # If anomalous=True, keep "flagged" periods in the data set, otherwise
    # remove them.
    if anomalous == True:
        return(data)
    else:
        for pw,fl in zip(power_cols, flag_cols):
            data[pw] = data[pw].mask(data[fl] > 0)
        return(data)
```

In [46]:

```
df = df_10min[df_10min['AssetName']== 'ORI-Ve-42']
df = df[df['State_Fault_Interpolated']== 2.0]
df = df[df['TM_SetPoint']== 500]

# split_date = '2023-02-01 00:00:00'
# df = df.loc[df.index >= split_date].copy()

split_date = '2023-03-25 00:00:00'
df['is_derated'] = np.where(df.index >= split_date, 'Yes', 'No')

fig = plt.figure(figsize=(20,10))

plt.scatter(x = df['Windspeed_Adjusted'],y=df['TM_ActivePower'],
            c=df.is_derated.astype('category').cat.codes,alpha=0.3)

plt.xlabel("Wind Speed (m/s)", fontsize=20)
plt.ylabel("Power (kW)", fontsize=20)
plt.title("Wind Turbine Derated Power Operation", fontsize=30)
plt.tick_params(labelsize=18, pad=6)
plt.legend()

plt.show()
```

```
No artists with labels found to put in legend.  Note that artists whose label start with an underscore are ignored when legend() is called with no argument.
```



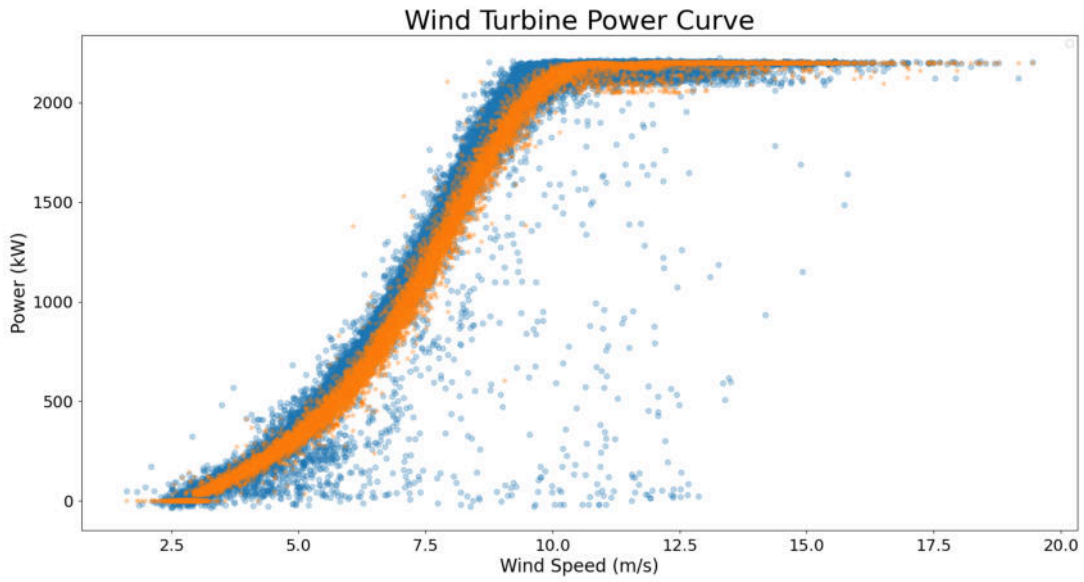
In [47]:
```
# import plotly.express as px

# # Create a scatter plot using Plotly
# fig = px.scatter(df, x='Windspeed_Adjusted', y='TM_ActivePower', color='TM_AmbientTemperature_Celsius',
#                  color_continuous_scale='temps', opacity=0.5, title='Wind Turbine ORI-Ve-42 Clean Power Curve')

# fig.update_layout(
#     xaxis_title="Wind Speed (m/s)",
#     yaxis_title="Actual Power (kW)",
#     coloraxis_colorbar_title="Ambient Temperature (C)",
#     coloraxis_colorbar_thickness=20,
#     coloraxis_colorbar_tickfont_size=15,
#     xaxis_title_font_size=20,
#     yaxis_title_font_size=20,
#     title_font_size=30
# )

# fig.update_xaxes(showgrid=True)
# fig.update_yaxes(showgrid=True)

# # fig.show()
# fig.update_layout(plot_bgcolor='white',
#     autosize=False,
#     width=1400,
#     height=700)

# fig.update_xaxes(
#     mirror=True,
#     ticks='outside',
#     showline=True,
#     linecolor='black',
#     gridcolor='lightgrey'
# )
# fig.update_yaxes(
#     mirror=True,
#     ticks='outside',
#     showline=True,
#     linecolor='black',
#     gridcolor='lightgrey'
# )
```

In [48]:
```
# import plotly.express as px

# # Create a scatter plot using Plotly
# fig = px.scatter(df, x='Windspeed_Adjusted', y='TM_ActivePower', color='TM_AmbientTemperature_Celsius',
#                  color_continuous_scale='temps', opacity=0.5, title='Wind Turbine Clean Power Curve', facet_col=df.is_dera

# fig.update_layout(
#     xaxis_title="Wind Speed (m/s)",
#     yaxis_title="Actual Power (kW)",
#     coloraxis_colorbar_title="Ambient Temperature (C)",
```

```
#     coloraxis_colorbar_thickness=20,
#     coloraxis_colorbar_tickfont_size=15,
#     xaxis_title_font_size=20,
#     yaxis_title_font_size=20,
#     title_font_size=30
# )

# # fig.show()
# fig.update_layout(plot_bgcolor='white',
#     autosize=False,
#     width=1400,
#     height=700)

# fig.update_xaxes(
#     mirror=True,
#     ticks='outside',
#     showline=True,
#     linecolor='black',
#     gridcolor='lightgrey'
# )
# fig.update_yaxes(
#     mirror=True,
#     ticks='outside',
#     showline=True,
#     linecolor='black',
#     gridcolor='lightgrey'
# )
```

In [49]:
```
fig = plt.figure(figsize=(20,10))

df = df[df['is_derated'] == 'No']

plt.scatter(x = df['Windspeed_Adjusted'],y=df['TM_ActivePower'],alpha=0.3)
plt.scatter(x = df['Windspeed_Adjusted'],y=df['TM_PossiblePower'], marker='*',alpha=0.3)

plt.xlabel("Wind Speed (m/s)", fontsize=20)
plt.ylabel("Power (kW)", fontsize=20)
plt.title("Wind Turbine Power Curve", fontsize=30)
plt.tick_params(labelsize=18, pad=6)
plt.legend()
plt.show()
```

```
No artists with labels found to put in legend.  Note that artists whose label start with an underscore are ignored when legend() is called with no argument.
```



In [50]:
```
#compute mean power vs windspeed
cols = ['TM_ActivePower', 'Windspeed_Adjusted']
df_var = df[cols].copy()
binsize = 1
df_var['windspeed_bin'] = np.round(binsize*(df['Windspeed_Adjusted']/binsize))
agger = {'TM_ActivePower':['mean', 'std', 'count']}
df_var = df_var.groupby('windspeed_bin').agg(agger).reset_index().sort_values('windspeed_bin')
```

```
df_var['sigma'] = df_var['TM_ActivePower']['std']/np.sqrt(df_var['TM_ActivePower']['count'])
df_var.sample(5)
```

Out[50]:

	windspeed_bin	TM_ActivePower			sigma
		mean	std	count	
5	7.0	1020.685286	177.024011	3135	3.161647
13	15.0	2195.988726	27.928986	1923	0.636892
7	9.0	1900.513117	214.560882	2327	4.447873
11	13.0	2169.073192	187.158578	1105	5.630262
0	2.0	14.263567	40.242073	32	7.113861

In [51]:
```
# import plotly.graph_objects as go

# xp = df['Windspeed_Adjusted']
# yp = df['TM_ActivePower']

# fig = go.Figure()

# # Scatter plot for all data points
# fig.add_trace(go.Scatter(x=xp, y=yp, mode='markers', marker=dict(size=5, opacity=0.3), name='all'))

# # Line plot for mean with error bars
# xp_mean = df_var['windspeed_bin']
# yp_mean = df_var['TM_ActivePower']['mean']
# err = df_var['TM_ActivePower']['std']
# fig.add_trace(go.Scatter(x=xp_mean, y=yp_mean, mode='lines+markers', marker=dict(size=3), line=dict(width=2, color='red'),
#                          name='mean ± σ', error_y=dict(type='data', array=err, visible=True)))

# fig.update_layout(
#     xaxis_title='Windspeed (m/s)',
#     yaxis_title='Active Power (kW)',
#     xaxis_title_font_size=20,
#     yaxis_title_font_size=20,
#     title_text='Untreated Active Power Variation',
#     title_font_size=30
# )

# # fig.show()
# fig.update_layout(plot_bgcolor='white',
#     autosize=False,
#     width=900,
#     height=500)

# fig.update_xaxes(
#     mirror=True,
#     ticks='outside',
#     showline=True,
#     linecolor='black',
#     gridcolor='lightgrey'
# )
# fig.update_yaxes(
#     mirror=True,
#     ticks='outside',
#     showline=True,
#     linecolor='black',
#     gridcolor='lightgrey'
# )
```

Outlier Removal

First, wind speed data were divided into many small independent intervals, in which outlier data was checked, detected and removed.

In [52]:
```
# df = df.reset_index()
# df = df.dropna()
```

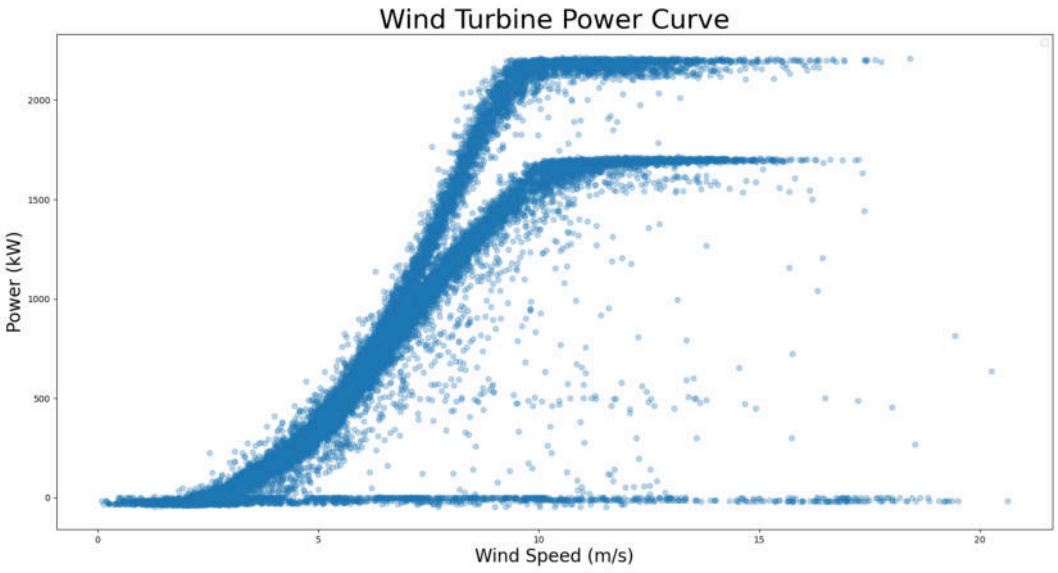
In [53]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 27662 entries, 2022-07-03 22:00:00 to 2023-03-24 02:00:00
Data columns (total 12 columns):
 #   Column                          Non-Null Count  Dtype
---  ------                          --------------  -----
 0   AssetParent                     27662 non-null  object
 1   AssetName                       27662 non-null  object
 2   TM_ActivePower                  26321 non-null  float64
 3   TM_PossiblePower                27008 non-null  float64
 4   State_Fault_Interpolated        27662 non-null  float64
 5   Windspeed_Adjusted              27244 non-null  float64
 6   Generator_RPM                   23431 non-null  float64
 7   TM_AmbientTemperature_Celsius   26988 non-null  float64
 8   RotorRPM                        26988 non-null  float64
 9   BladeAngle                      27351 non-null  float64
 10  TM_SetPoint                     27662 non-null  float64
 11  is_derated                      27662 non-null  object
dtypes: float64(9), object(3)
memory usage: 2.7+ MB
```

In [54]:
```python
def outlier_remover(d, prop, min, max):
    q_low = d[prop].quantile(min)
    q_hi  = d[prop].quantile(max)
    return df[(d[prop] < q_hi) & (d[prop] > q_low)]
```

In [55]:
```python
df = df_10min[df_10min['AssetName']== 'ORI-Ve-42']
# df = df[df['State_Fault_Interpolated']== 2.0]
# df = df[df['TM_SetPoint']== 500]

split_date = '2023-02-01 00:00:00'
df = df.loc[df.index >= split_date].copy()

split_date = '2023-03-25 00:00:00'
df['is_derated'] = np.where(df.index >= split_date, 'Yes', 'No')

fig = plt.figure(figsize=(20,10))

plt.scatter(x = df['Windspeed_Adjusted'],y=df['TM_ActivePower'],alpha=0.3)

plt.xlabel("Wind Speed (m/s)", fontsize=20)
plt.ylabel("Power (kW)", fontsize=20)
plt.title("Wind Turbine Power Curve", fontsize=30)

plt.legend()

plt.show()
```

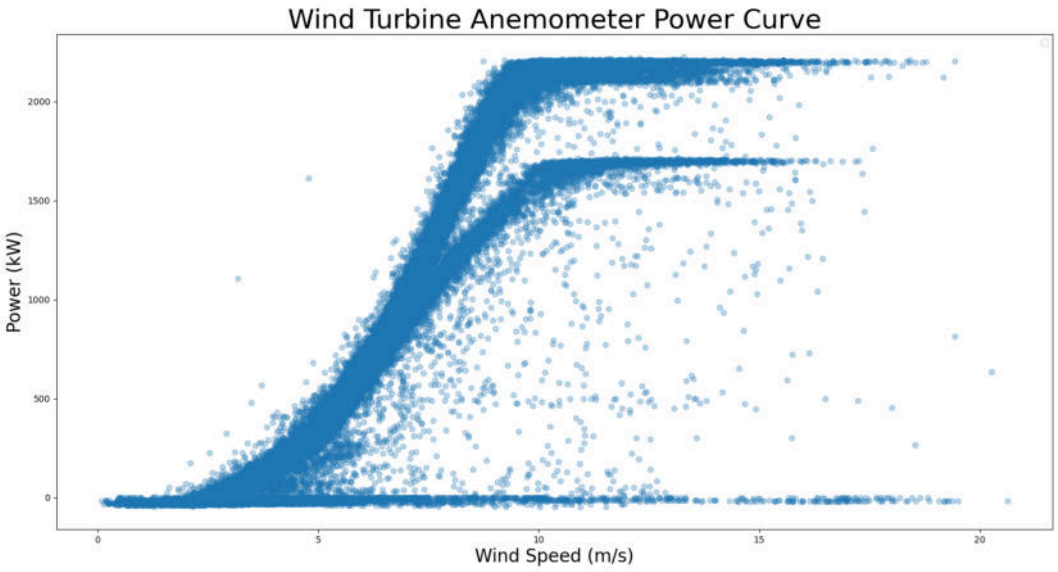
```
No artists with labels found to put in legend.  Note that artists whose label start with an underscore are ignored when legend() is called with no argument.
```



Anemometer Faults

In [56]:
```
df = df_10min[df_10min['AssetName']== 'ORI-Ve-42']
# df = df[df['State_Fault_Interpolated']== 2.0]
# df = df[df['TM_SetPoint']== 500]
# df = df[(df['TM_ActivePower']> 0) & (df['Windspeed_Adjusted']<= 3)]

# split_date = '2023-02-01 00:00:00'
# df = df.loc[df.index >= split_date].copy()

# split_date = '2023-03-25 00:00:00'
df['is_anemometer_fault'] = np.where(((df['TM_ActivePower']> 0) & (df['Windspeed_Adjusted']<= 3)), 'Yes', 'No')

fig = plt.figure(figsize=(20,10))

plt.scatter(x = df['Windspeed_Adjusted'],y=df['TM_ActivePower'],alpha=0.3)

plt.xlabel("Wind Speed (m/s)", fontsize=20)
plt.ylabel("Power (kW)", fontsize=20)
plt.title("Wind Turbine Anemometer Power Curve", fontsize=30)

plt.legend()

plt.show()
```

```
No artists with labels found to put in legend.  Note that artists whose label start with an underscore are ignored when legend() is called with no argument.
```

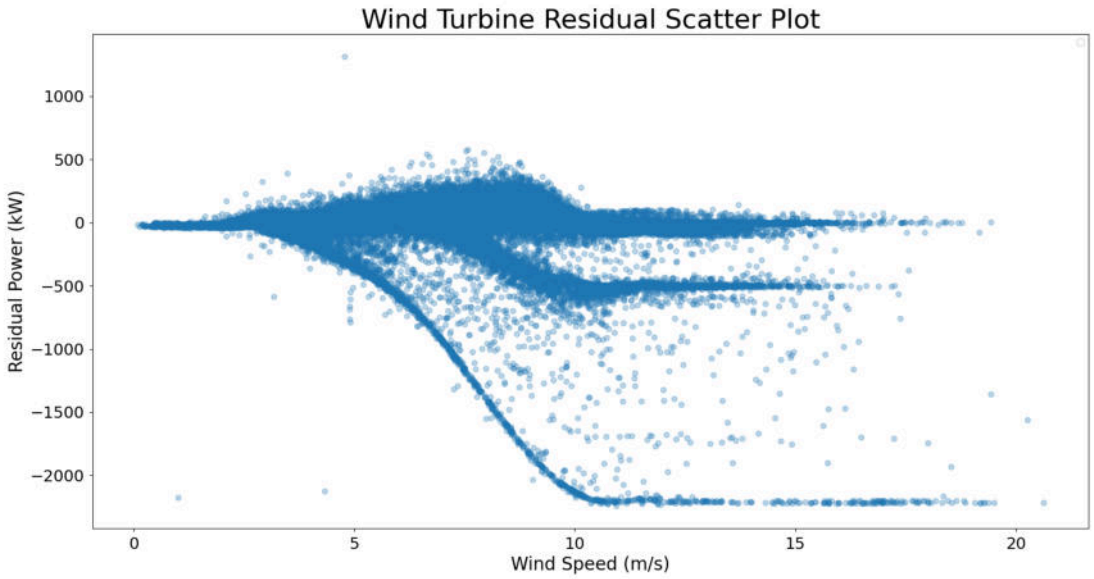


In [57]:
```
df.is_anemometer_fault.value_counts(ascending=True)
```

Out[57]:
```
Yes     1087
No     54419
Name: is_anemometer_fault, dtype: int64
```

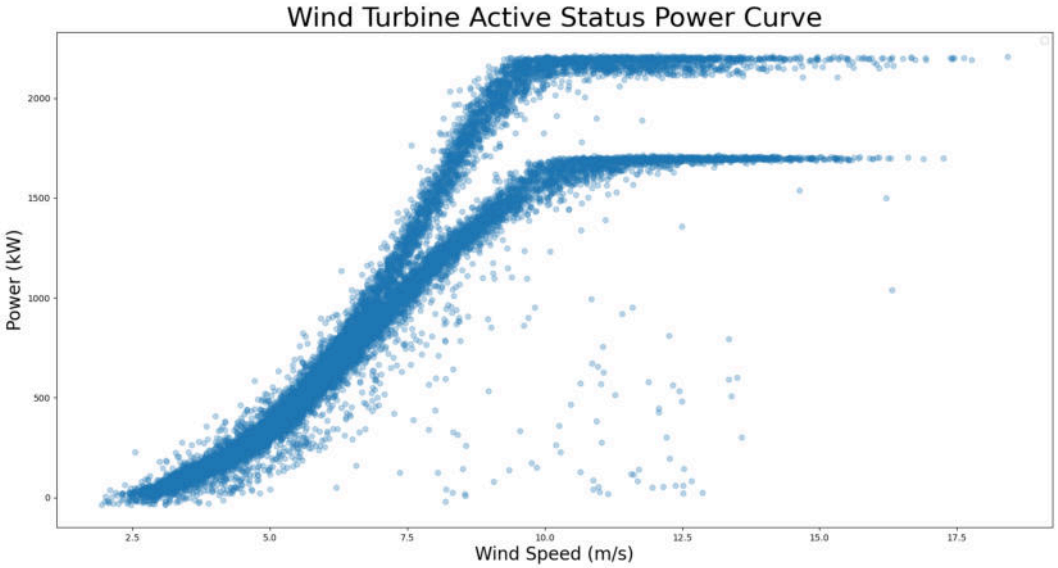
In [58]:
```
df["Residual_Power"] = df["TM_ActivePower"] - df["TM_PossiblePower"]

fig = plt.figure(figsize=(20,10))

plt.scatter(x = df['Windspeed_Adjusted'],y=df['Residual_Power'],alpha=0.3)

plt.xlabel("Wind Speed (m/s)", fontsize=20)
plt.ylabel("Residual Power (kW)", fontsize=20)
plt.title("Wind Turbine Residual Scatter Plot", fontsize=30)
plt.tick_params(labelsize=18, pad=6)
plt.legend()
plt.show()
```

```
No artists with labels found to put in legend.  Note that artists whose label start with an underscore are ignored when legend() is called with no argument.
```



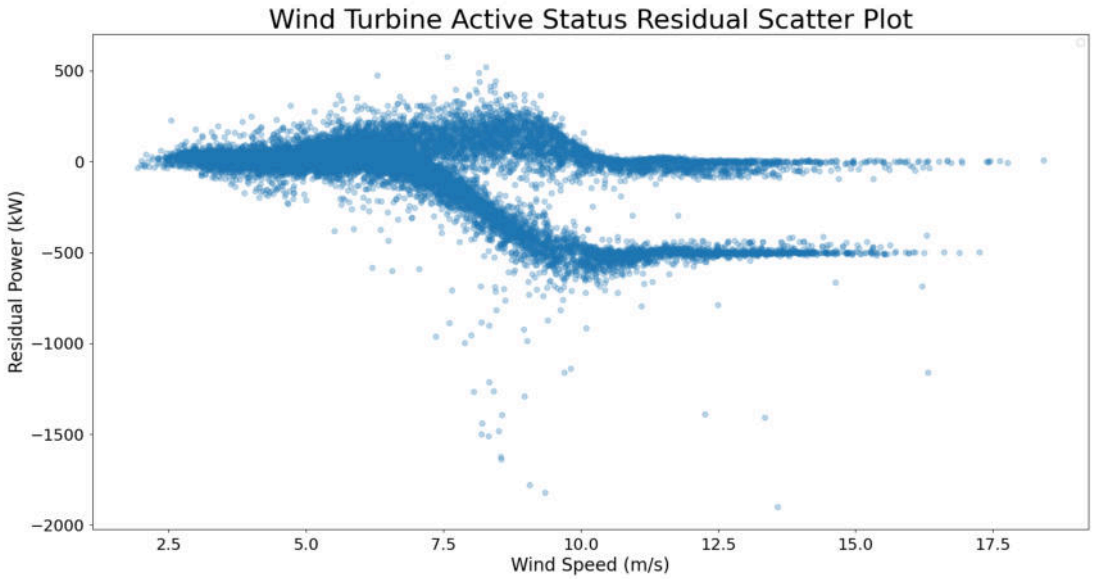
In [59]:
```
df = df_10min[df_10min['AssetName']== 'ORI-Ve-42']
df = df[df['State_Fault_Interpolated']== 2.0]
df = df[df['TM_SetPoint']== 500]

split_date = '2023-02-01 00:00:00'
df = df.loc[df.index >= split_date].copy()

split_date = '2023-03-25 00:00:00'
df['is_derated'] = np.where(df.index >= split_date, 'Yes', 'No')

fig = plt.figure(figsize=(20,10))

plt.scatter(x = df['Windspeed_Adjusted'],y=df['TM_ActivePower'],alpha=0.3)

plt.xlabel("Wind Speed (m/s)", fontsize=20)
plt.ylabel("Power (kW)", fontsize=20)
plt.title("Wind Turbine Active Status Power Curve", fontsize=30)

plt.legend()

plt.show()
```

```
No artists with labels found to put in legend.  Note that artists whose label start with an underscore are ignored when legend() is called with no argument.
```

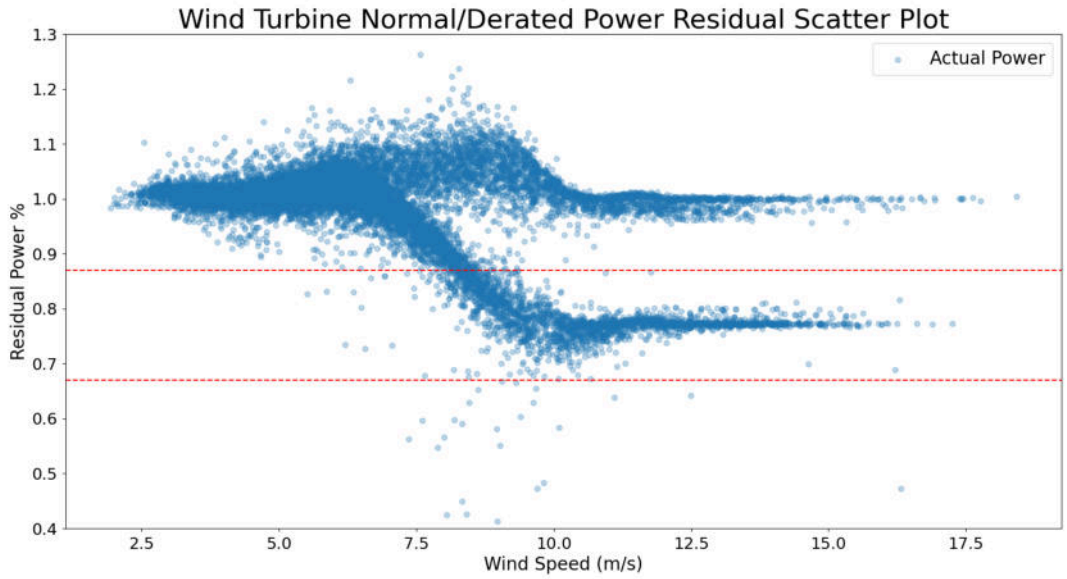


In [60]:
```
df["Residual_Power"] = df["TM_ActivePower"] - df["TM_PossiblePower"]

fig = plt.figure(figsize=(20,10))

plt.scatter(x = df['Windspeed_Adjusted'],y=df['Residual_Power'],alpha=0.3)

plt.xlabel("Wind Speed (m/s)", fontsize=20)
plt.ylabel("Residual Power (kW)", fontsize=20)
plt.title("Wind Turbine Active Status Residual Scatter Plot", fontsize=30)
plt.tick_params(labelsize=18, pad=6)
plt.legend()
plt.show()
```

```
No artists with labels found to put in legend.  Note that artists whose label start with an underscore are ignored when legend() is called with no argument.
```



In [61]:
```
df["Residual_Power"] = (1-(df["TM_PossiblePower"]-df["TM_ActivePower"])/ 2200)

fig = plt.figure(figsize=(20,10))

plt.scatter(x = df['Windspeed_Adjusted'],y=df['Residual_Power'], label = 'Actual Power',alpha=0.3)

plt.xlabel("Wind Speed (m/s)", fontsize=20)
plt.ylabel("Residual Power %", fontsize=20)
plt.ylim(0.4,1.30)
plt.axhline(0.87, ls='--', lw=1.5, c='r')
plt.axhline(0.67, ls='--', lw=1.5, c='r')
plt.title("Wind Turbine Normal/Derated Power Residual Scatter Plot", fontsize=30)
plt.tick_params(labelsize=18, pad=6)
plt.legend(fontsize=20)
plt.show()
```



Outlier Treatment

Turbine power data is affected by many outliers and this section of the notebook focuses on cleaning them. A first attempt consists in subdividing the wind speed axis into small intervals (given by the capillarity variable), calculating the quantiles of the data referred to that range in that range and eliminating the outliers of the corresponding boxplot.

The graphs generated are the plot of the clean data and a histogram that must be compared with that of the section above, since it is the histogram of the same data section.

Normal Operation

In [62]:
```
df = df_10min[df_10min['AssetName']== 'ORI-Ve-42']
df = df[df['State_Fault_Interpolated']== 2.0]
df = df[df['TM_SetPoint']== 500]
df["TM_ActivePower"] = df["TM_ActivePower"].clip(lower=0)

# split_date = '2023-02-01 00:00:00'
# df = df.loc[df.index >= split_date].copy()

split_date = '2023-03-25 00:00:00'
df['is_derated'] = np.where(df.index >= split_date, 'Yes', 'No')
df = df[df['is_derated'] == 'No']
```

In [63]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 27662 entries, 2022-07-03 22:00:00 to 2023-03-24 02:00:00
Data columns (total 12 columns):
 #   Column                          Non-Null Count  Dtype
---  ------                          --------------  -----
 0   AssetParent                     27662 non-null  object
 1   AssetName                       27662 non-null  object
 2   TM_ActivePower                  26321 non-null  float64
 3   TM_PossiblePower                27008 non-null  float64
 4   State_Fault_Interpolated        27662 non-null  float64
 5   Windspeed_Adjusted              27244 non-null  float64
 6   Generator_RPM                   23431 non-null  float64
 7   TM_AmbientTemperature_Celsius   26988 non-null  float64
 8   RotorRPM                        26988 non-null  float64
 9   BladeAngle                      27351 non-null  float64
 10  TM_SetPoint                     27662 non-null  float64
 11  is_derated                      27662 non-null  object
dtypes: float64(9), object(3)
memory usage: 2.7+ MB
```

In [64]:
```
whisker_length = 1
capillarity = 0.25
turbine_df_clean = pd.DataFrame(columns = ['Windspeed_Adjusted', 'TM_ActivePower'])
```

```
for i in np.linspace(start = 0, stop = 30, endpoint = False, num = int(30 / capillarity)):
    lower_bound = i
    #print("Lower Bound" ,lower_bound)
    upper_bound = i + capillarity
    #print("Upper Bound" ,upper_bound)

    if upper_bound <= 3: #Windspeed Zone Selection
        data_interval =  df[((df.Windspeed_Adjusted < upper_bound) & (df.Windspeed_Adjusted >= lower_bound))]
    elif upper_bound > 3 and upper_bound <= 15:
        data_interval =  df[((((df.Windspeed_Adjusted < upper_bound) & (df.Windspeed_Adjusted >= lower_bound)) & (df.TM_Activ
    elif upper_bound > 15:
        data_interval =  df[((((df.Windspeed_Adjusted < upper_bound) & (df.Windspeed_Adjusted >= lower_bound)) & (df.TM_Activ

    quantiles = data_interval.quantile([.25, .75], axis = 'rows') # calculation of the quantiles based on boxplot
    boxplot_h = quantiles.TM_ActivePower[0.75] - quantiles.TM_ActivePower[0.25]
    boxplot_ends = [quantiles.TM_ActivePower[0.25] - whisker_length*boxplot_h, quantiles.TM_ActivePower[0.75] + whisker_leng
    data_interval = data_interval[(data_interval.TM_ActivePower < boxplot_ends[1]) & (data_interval.TM_ActivePower > boxplot
    turbine_df_clean = turbine_df_clean.append(data_interval, ignore_index = False)
```

In [65]:
```
turbine_df_clean.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 22795 entries, 2023-01-24 09:50:00 to 2022-12-15 19:30:00
Data columns (total 12 columns):
 #   Column                          Non-Null Count  Dtype  
---  ------                          --------------  -----  
 0   Windspeed_Adjusted              22795 non-null  float64
 1   TM_ActivePower                  22795 non-null  float64
 2   AssetParent                     22795 non-null  object 
 3   AssetName                       22795 non-null  object 
 4   TM_PossiblePower                22697 non-null  float64
 5   State_Fault_Interpolated        22795 non-null  float64
 6   Generator_RPM                   19524 non-null  float64
 7   TM_AmbientTemperature_Celsius   22686 non-null  float64
 8   RotorRPM                        22686 non-null  float64
 9   BladeAngle                      22783 non-null  float64
 10  TM_SetPoint                     22795 non-null  float64
 11  is_derated                      22795 non-null  object 
dtypes: float64(9), object(3)
memory usage: 2.3+ MB
```

In [66]:
```
turbine_df_clean.sample(10)
```

Out[66]:

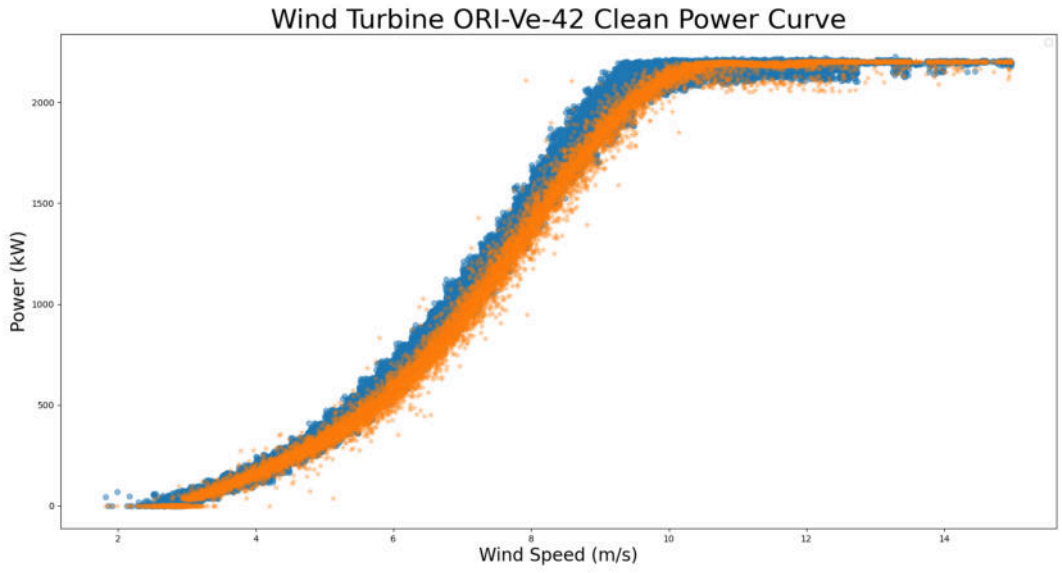
	Windspeed_Adjusted	TM_ActivePower	AssetParent	AssetName	TM_PossiblePower	State_Fault_Interpolated	Generator_RPM
2023-02-28 17:30:00	5.036642	394.34875	Orient	ORI-Ve-42	334.58646	2.0	NaN
2022-07-16 03:10:00	5.170537	379.35104	Orient	ORI-Ve-42	361.70593	2.0	806.0
2022-12-20 20:20:00	3.664124	74.16705	Orient	ORI-Ve-42	112.98022	2.0	732.0
2022-12-31 00:20:00	7.388636	1145.16300	Orient	ORI-Ve-42	1100.16770	2.0	1093.0
2022-08-15 05:40:00	5.036635	350.45102	Orient	ORI-Ve-42	328.54068	2.0	915.0
2022-11-06 01:10:00	14.894995	2202.36770	Orient	ORI-Ve-42	2200.00000	2.0	1351.0
2022-12-08 00:10:00	6.988758	1073.84420	Orient	ORI-Ve-42	985.56840	2.0	1124.0
2022-09-12 03:40:00	6.103576	659.60790	Orient	ORI-Ve-42	588.83320	2.0	973.0
2022-10-20 00:10:00	6.833736	883.10236	Orient	ORI-Ve-42	1073.21030	2.0	1050.0
2022-10-09 08:00:00	4.960701	335.08633	Orient	ORI-Ve-42	320.11435	2.0	787.0

Clean Power Curve

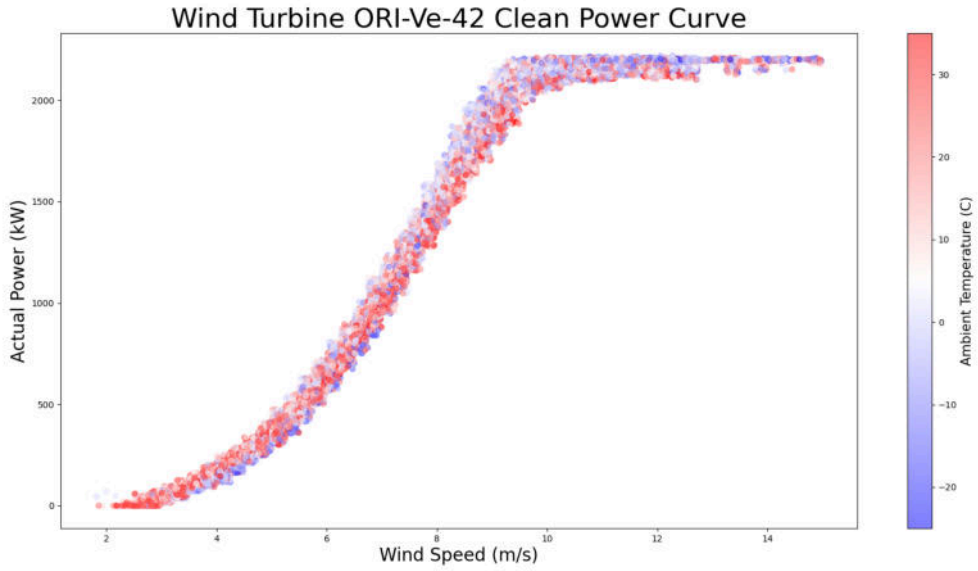
In [67]:

```
fig = plt.figure(figsize=(20,10))

df = turbine_df_clean

plt.scatter(x = df['Windspeed_Adjusted'],y=df['TM_ActivePower'],alpha=0.5)
plt.scatter(x = df['Windspeed_Adjusted'],y=df['TM_PossiblePower'], marker='*',alpha=0.3)

plt.xlabel("Wind Speed (m/s)", fontsize=20)
plt.ylabel("Power (kW)", fontsize=20)
plt.title("Wind Turbine ORI-Ve-42 Clean Power Curve", fontsize=30)

plt.legend()
plt.show()
```

```
No artists with labels found to put in legend.  Note that artists whose label start with an underscore are ignored when legend() is called with no argument.
```



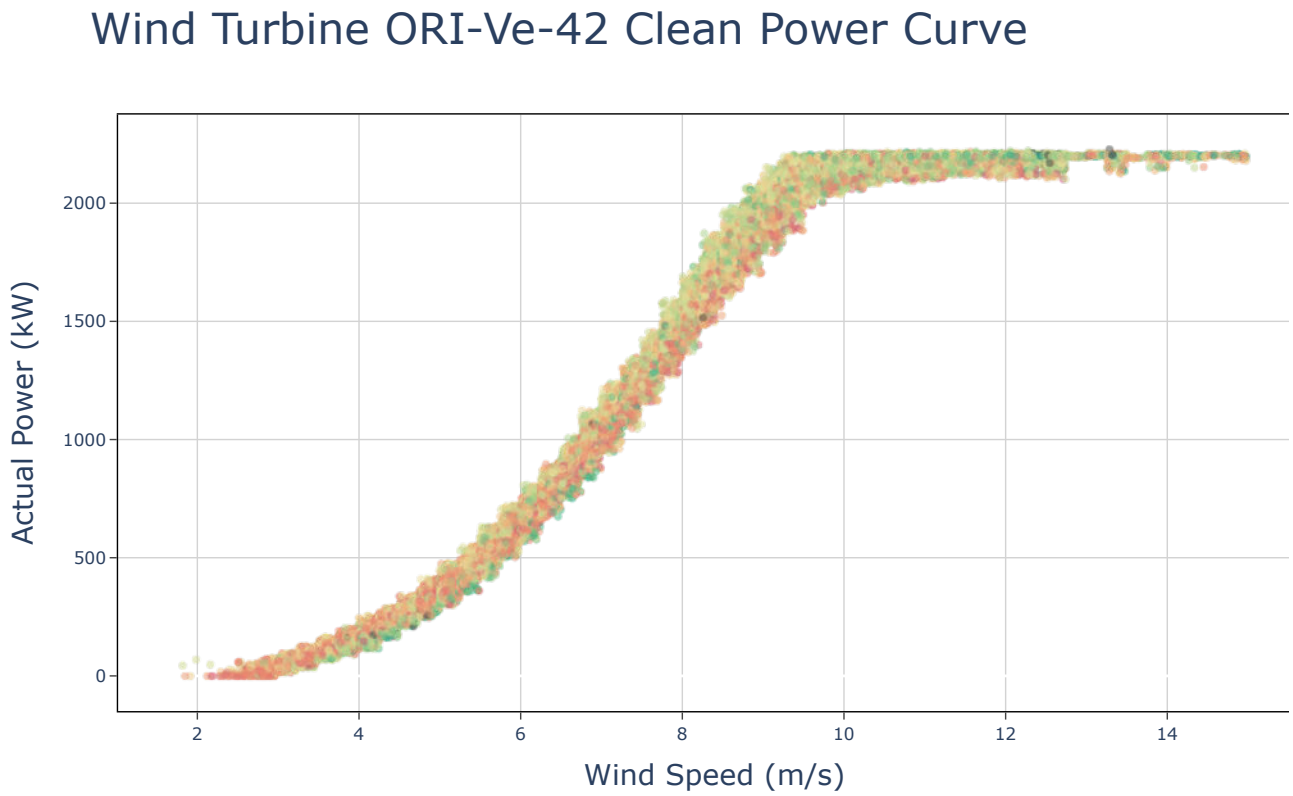
In [68]:
```
fig = plt.figure(figsize=(20,10))

plt.scatter(x = df['Windspeed_Adjusted'],y=df['TM_ActivePower'], c = df['TM_AmbientTemperature_Celsius'], cmap='bwr', alpha
cbar = plt.colorbar(orientation = 'vertical',label="Ambient Temperature (C)")

cbar.set_label(label="Ambient Temperature (C)", size=15)
plt.xlabel("Wind Speed (m/s)", fontsize=20)
plt.ylabel("Actual Power (kW)", fontsize=20)
plt.title("Wind Turbine ORI-Ve-42 Clean Power Curve", fontsize=30)

plt.show()
```



Power vs Windspeed vs Ambient Temperature

In [69]:
```
import plotly.express as px

# Create a scatter plot using Plotly
fig = px.scatter(df, x='Windspeed_Adjusted', y='TM_ActivePower', color='TM_AmbientTemperature_Celsius',
                 color_continuous_scale='temps', opacity=0.5, title='Wind Turbine ORI-Ve-42 Clean Power Curve')

fig.update_layout(
    xaxis_title="Wind Speed (m/s)",
    yaxis_title="Actual Power (kW)",
```

```
    coloraxis_colorbar_title="Ambient Temperature (C)",
    coloraxis_colorbar_thickness=20,
    coloraxis_colorbar_tickfont_size=15,
    xaxis_title_font_size=20,
    yaxis_title_font_size=20,
    title_font_size=30
)

# fig.show()
fig.update_layout(plot_bgcolor='white',
    autosize=False,
    width=1200,
    height=600)

fig.update_xaxes(
    mirror=True,
    ticks='outside',
    showline=True,
    linecolor='black',
    gridcolor='lightgrey'
)
fig.update_yaxes(
    mirror=True,
    ticks='outside',
    showline=True,
    linecolor='black',
    gridcolor='lightgrey'
)
```



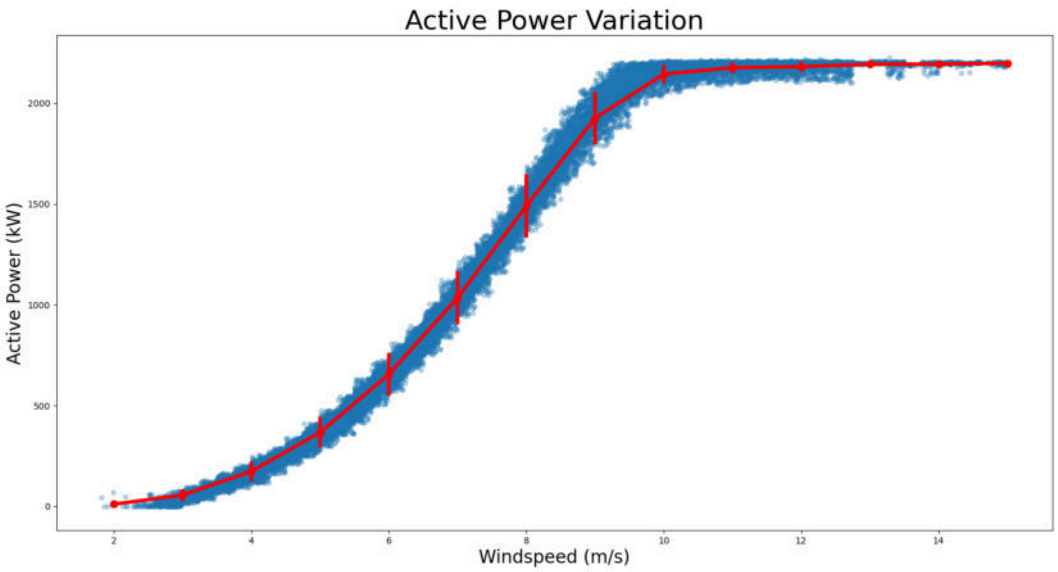
Plot Active Power Variation

In [70]:
```
#compute mean power vs windspeed
cols = ['TM_ActivePower', 'Windspeed_Adjusted']
df_var = df[cols].copy()
binsize = 1
df_var['windspeed_bin'] = np.round(binsize*(df['Windspeed_Adjusted']/binsize))
agger = {'TM_ActivePower':['mean', 'std', 'count']}
df_var = df_var.groupby('windspeed_bin').agg(agger).reset_index().sort_values('windspeed_bin')
df_var['sigma'] = df_var['TM_ActivePower']['std']/np.sqrt(df_var['TM_ActivePower']['count'])
df_var.sample(5)
```

Out[70]:

	windspeed_bin	TM_ActivePower			sigma
		mean	std	count	
5	7.0	1035.104471	132.896453	2911	2.463158
2	4.0	174.413337	49.340335	1848	1.147760
8	10.0	2144.282734	46.331606	1705	1.122058
13	15.0	2198.064376	5.286928	1069	0.161702
7	9.0	1925.890555	130.424721	2185	2.790193

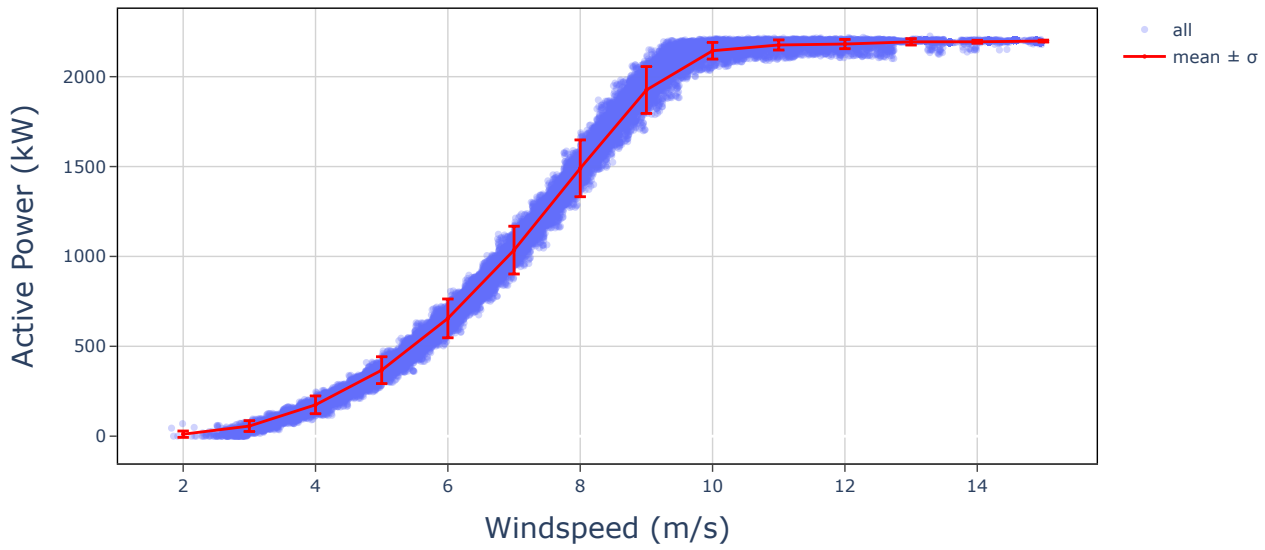
In [71]:
```
#matlibplot of power vs windspeed

xp = df['Windspeed_Adjusted']
yp = df['TM_ActivePower']
#sns.set(font_scale=1.2, font='DejaVu Sans')
fig, ax = plt.subplots(1,1, figsize=(20, 10))
p = ax.plot(xp, yp, marker='o', linestyle='none', markersize=5, alpha=0.3, label='all')
xp = df_var['windspeed_bin']
yp = df_var['TM_ActivePower']['mean']
err = df_var['TM_ActivePower']['std']
p = ax.errorbar(xp, yp,err, marker='o', linestyle='-', markersize=8, linewidth=4, label=r'mean $\pm\sigma$', zorder=3, c='re
p = ax.set_xlabel('Windspeed (m/s)', fontsize=20)
p = ax.set_ylabel('Active Power (kW)', fontsize=20)
p = ax.set_title('Active Power Variation', fontsize=30)
```



In [72]:
```
import plotly.graph_objects as go

xp = df['Windspeed_Adjusted']
yp = df['TM_ActivePower']

fig = go.Figure()

# Scatter plot for all data points
fig.add_trace(go.Scatter(x=xp, y=yp, mode='markers', marker=dict(size=5, opacity=0.3), name='all'))

# Line plot for mean with error bars
xp_mean = df_var['windspeed_bin']
yp_mean = df_var['TM_ActivePower']['mean']
err = df_var['TM_ActivePower']['std']
fig.add_trace(go.Scatter(x=xp_mean, y=yp_mean, mode='lines+markers', marker=dict(size=3), line=dict(width=2, color='red'),
                         name='mean ± σ', error_y=dict(type='data', array=err, visible=True)))

fig.update_layout(
    xaxis_title='Windspeed (m/s)',
    yaxis_title='Active Power (kW)',
    xaxis_title_font_size=20,
    yaxis_title_font_size=20,
    title_text='Active Power Variation',
```

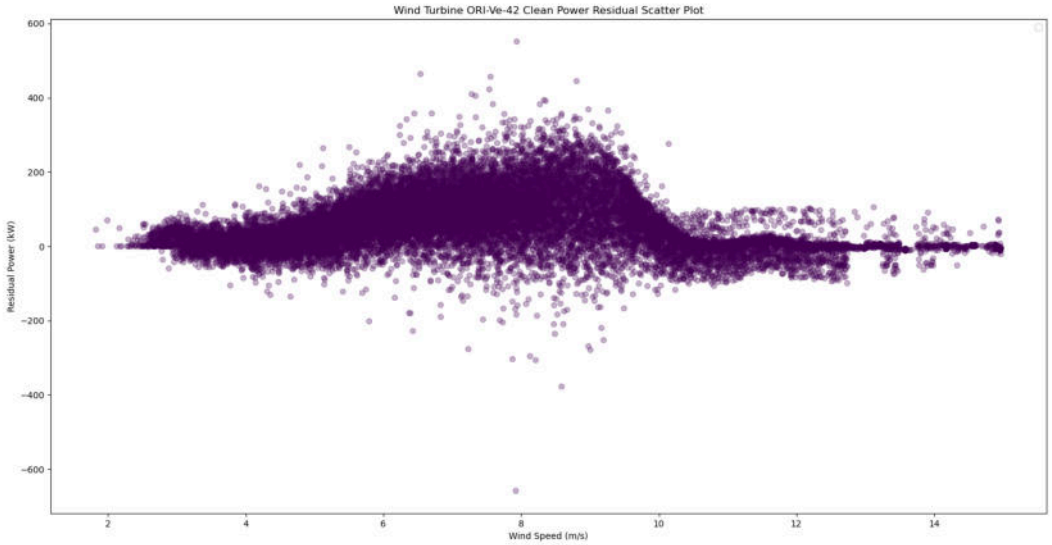
```
    title_font_size=30
)

# fig.show()
fig.update_layout(plot_bgcolor='white',
    autosize=False,
    width=900,
    height=500)

fig.update_xaxes(
    mirror=True,
    ticks='outside',
    showline=True,
    linecolor='black',
    gridcolor='lightgrey'
)
fig.update_yaxes(
    mirror=True,
    ticks='outside',
    showline=True,
    linecolor='black',
    gridcolor='lightgrey'
)
```

Active Power Variation



In [73]:
```
df["Residual_Power"] = df["TM_ActivePower"] - df["TM_PossiblePower"]

fig = plt.figure(figsize=(20,10))

plt.scatter(x = df['Windspeed_Adjusted'],y=df['Residual_Power'], c=df.is_derated.astype('category').cat.codes,alpha=0.3)

plt.xlabel("Wind Speed (m/s)")
plt.ylabel("Residual Power (kW)")
plt.title("Wind Turbine ORI-Ve-42 Clean Power Residual Scatter Plot")

plt.legend()
plt.show()
```

```
No artists with labels found to put in legend.  Note that artists whose label start with an underscore are ignored when legend() is called with no argument.
```



Derated Operation

In [74]:
```
df = df_10min[df_10min['AssetName']== 'ORI-Ve-42']
df = df[df['State_Fault_Interpolated']== 2.0]
df = df[df['TM_SetPoint']== 500]
df["TM_ActivePower"] = df["TM_ActivePower"].clip(lower=0)

split_date = '2023-02-01 00:00:00'
df = df.loc[df.index >= split_date].copy()

split_date = '2023-03-25 00:00:00'
df['is_derated'] = np.where(df.index >= split_date, 'Yes', 'No')
df = df[df['is_derated'] == 'Yes']
```

In [75]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 11330 entries, 2023-03-29 14:10:00 to 2023-06-30 09:40:00
Data columns (total 12 columns):
 #   Column                         Non-Null Count  Dtype
---  ------                         --------------  -----
 0   AssetParent                    11330 non-null  object
 1   AssetName                      11330 non-null  object
 2   TM_ActivePower                 10823 non-null  float64
 3   TM_PossiblePower               11328 non-null  float64
 4   State_Fault_Interpolated       11330 non-null  float64
 5   Windspeed_Adjusted             11327 non-null  float64
 6   Generator_RPM                  0 non-null      float64
 7   TM_AmbientTemperature_Celsius  11325 non-null  float64
 8   RotorRPM                       11325 non-null  float64
 9   BladeAngle                     11325 non-null  float64
 10  TM_SetPoint                    11330 non-null  float64
 11  is_derated                     11330 non-null  object
dtypes: float64(9), object(3)
memory usage: 1.1+ MB
```

In [76]:
```
whisker_length = 1
capillarity = 0.25
turbine_df_clean_derate = pd.DataFrame(columns = ['Windspeed_Adjusted', 'TM_ActivePower'])

for i in np.linspace(start = 0, stop = 30, endpoint = False, num = int(30 / capillarity)):
    lower_bound = i
    #print("Lower Bound" ,lower_bound)
    upper_bound = i + capillarity
    #print("Upper Bound" ,upper_bound)

    if upper_bound <= 3: #Windspeed Zone Selection
        data_interval =  df[((df.Windspeed_Adjusted < upper_bound) & (df.Windspeed_Adjusted >= lower_bound))]
    elif upper_bound > 3 and upper_bound <= 15:
        data_interval =  df[(((df.Windspeed_Adjusted < upper_bound) & (df.Windspeed_Adjusted >= lower_bound)) & (df.TM_Activ
```

```
    elif upper_bound > 15:
        data_interval =  df[((df.Windspeed_Adjusted < upper_bound) & (df.Windspeed_Adjusted >= lower_bound)) & (df.TM_Activ

    quantiles = data_interval.quantile([.25, .75], axis = 'rows') # calculation of the quantiles based on boxplot
    boxplot_h = quantiles.TM_ActivePower[0.75] - quantiles.TM_ActivePower[0.25]
    boxplot_ends = [quantiles.TM_ActivePower[0.25] - whisker_length*boxplot_h, quantiles.TM_ActivePower[0.75] + whisker_leng
    data_interval = data_interval[(data_interval.TM_ActivePower < boxplot_ends[1]) & (data_interval.TM_ActivePower > boxplot
    turbine_df_clean_derate = turbine_df_clean_derate.append(data_interval, ignore_index = False)
```

In [77]:
```
turbine_df_clean_derate.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 9989 entries, 2023-06-06 19:20:00 to 2023-04-01 02:20:00
Data columns (total 12 columns):
 #   Column                         Non-Null Count  Dtype
---  ------                         --------------  -----
 0   Windspeed_Adjusted             9989 non-null   float64
 1   TM_ActivePower                 9989 non-null   float64
 2   AssetParent                    9989 non-null   object
 3   AssetName                      9989 non-null   object
 4   TM_PossiblePower               9988 non-null   float64
 5   State_Fault_Interpolated       9989 non-null   float64
 6   Generator_RPM                  0 non-null      float64
 7   TM_AmbientTemperature_Celsius  9988 non-null   float64
 8   RotorRPM                       9988 non-null   float64
 9   BladeAngle                     9988 non-null   float64
 10  TM_SetPoint                    9989 non-null   float64
 11  is_derated                     9989 non-null   object
dtypes: float64(9), object(3)
memory usage: 1014.5+ KB
```

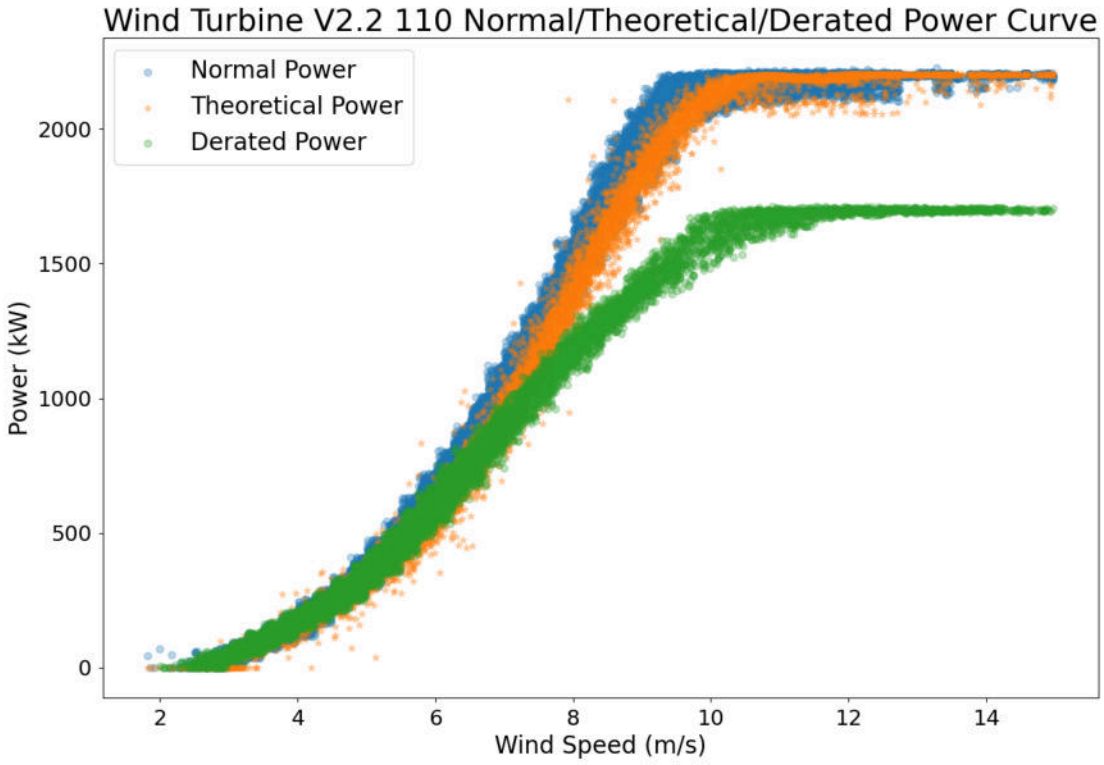
In [78]:
```
turbine_df_clean_derate.sample(10)
```

Out[78]:

	Windspeed_Adjusted	TM_ActivePower	AssetParent	AssetName	TM_PossiblePower	State_Fault_Interpolated	Generator_RPM
2023-06-05 04:00:00	5.345714	346.192500	Orient	ORI-Ve-42	399.31726	2.0	NaN
2023-06-21 11:50:00	4.883380	374.319100	Orient	ORI-Ve-42	305.36832	2.0	NaN
2023-05-05 00:20:00	7.799323	1127.588700	Orient	ORI-Ve-42	1291.62520	2.0	NaN
2023-04-14 06:30:00	11.774662	1697.583400	Orient	ORI-Ve-42	2188.21500	2.0	NaN
2023-04-07 06:00:00	9.967999	1651.397200	Orient	ORI-Ve-42	2116.85640	2.0	NaN
2023-05-11 14:50:00	5.532009	469.220800	Orient	ORI-Ve-42	442.72440	2.0	NaN
2023-04-26 23:40:00	6.305880	702.919800	Orient	ORI-Ve-42	665.89984	2.0	NaN
2023-03-28 03:20:00	7.250971	992.809450	Orient	ORI-Ve-42	1038.63230	2.0	NaN
2023-06-02 10:30:00	3.203862	63.354145	Orient	ORI-Ve-42	53.52344	2.0	NaN
2023-04-25 08:00:00	5.012161	360.348900	Orient	ORI-Ve-42	327.45930	2.0	NaN

In [79]:
```
fig = plt.figure(figsize=(15,10))

df = turbine_df_clean
df2 = turbine_df_clean_derate
```

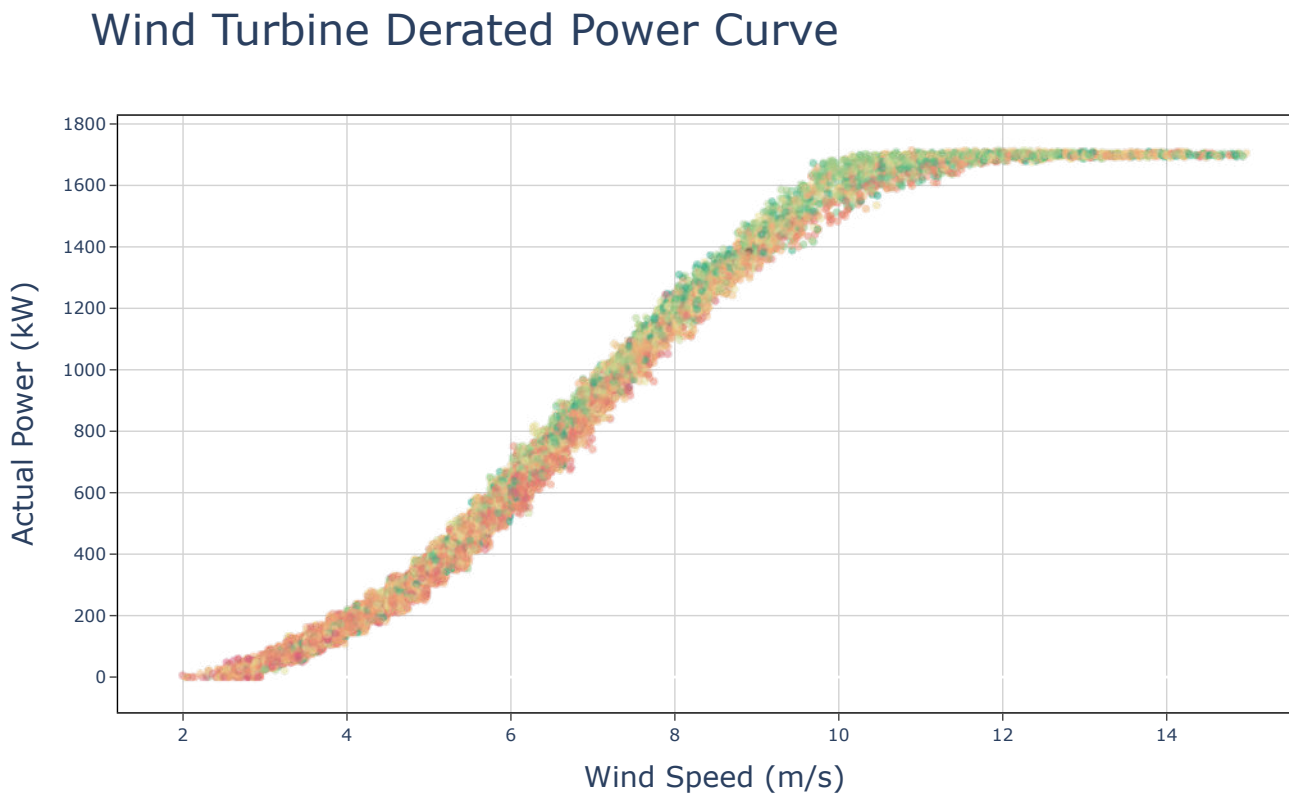
```
plt.scatter(x = df['Windspeed_Adjusted'],y=df['TM_ActivePower'], label = 'Normal Power',alpha=0.3)
plt.scatter(x = df['Windspeed_Adjusted'],y=df['TM_PossiblePower'], marker='*', label = 'Theoretical Power',alpha=0.3)
plt.scatter(x = df2['Windspeed_Adjusted'],y=df2['TM_ActivePower'], label = 'Derated Power',alpha=0.3)

plt.xlabel("Wind Speed (m/s)", fontsize=20)
plt.ylabel("Power (kW)", fontsize=20)
plt.title("Wind Turbine V2.2 110 Normal/Theoretical/Derated Power Curve", fontsize=26)
plt.tick_params(labelsize=18, pad=6)
plt.legend(fontsize=20)
plt.show()
```



Power vs Windspeed vs Ambient Temperature

In [80]:
```
import plotly.express as px

# Create a scatter plot using Plotly
fig = px.scatter(df2, x='Windspeed_Adjusted', y='TM_ActivePower', color='TM_AmbientTemperature_Celsius',
                 color_continuous_scale='temps', opacity=0.5, title='Wind Turbine Derated Power Curve')

fig.update_layout(
    xaxis_title="Wind Speed (m/s)",
    yaxis_title="Actual Power (kW)",
    coloraxis_colorbar_title="Ambient Temperature (C)",
    coloraxis_colorbar_thickness=20,
    coloraxis_colorbar_tickfont_size=15,
    xaxis_title_font_size=20,
    yaxis_title_font_size=20,
    title_font_size=30
)

# fig.show()
fig.update_layout(plot_bgcolor='white',
    autosize=False,
    width=1200,
    height=600)

fig.update_xaxes(
    mirror=True,
    ticks='outside',
    showline=True,
    linecolor='black',
    gridcolor='lightgrey'
```

```
)
fig.update_yaxes(
    mirror=True,
    ticks='outside',
    showline=True,
    linecolor='black',
    gridcolor='lightgrey'
)
```



Active Power Variation

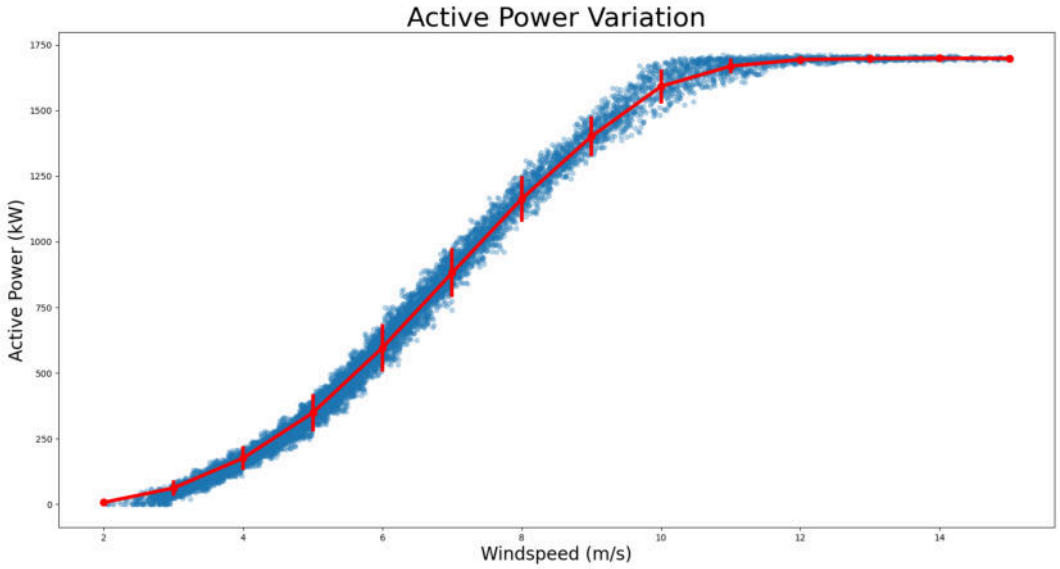
In [81]:
```
#compute mean power vs windspeed
cols = ['TM_ActivePower', 'Windspeed_Adjusted']
df_var = df2[cols].copy()
binsize = 1
df_var['windspeed_bin'] = np.round(binsize*(df2['Windspeed_Adjusted']/binsize))
agger = {'TM_ActivePower':['mean', 'std', 'count']}
df_var = df_var.groupby('windspeed_bin').agg(agger).reset_index().sort_values('windspeed_bin')
df_var['sigma'] = df_var['TM_ActivePower']['std']/np.sqrt(df_var['TM_ActivePower']['count'])
df_var.sample(5)
```

Out[81]:

	windspeed_bin	TM_ActivePower			sigma
		mean	std	count	
9	11.0	1668.753555	28.204051	402	1.406690
11	13.0	1697.559516	5.058841	200	0.357714
0	2.0	7.533340	9.002553	30	1.643634
2	4.0	176.218678	46.164913	1557	1.169951
3	5.0	348.965437	70.395967	1950	1.594155

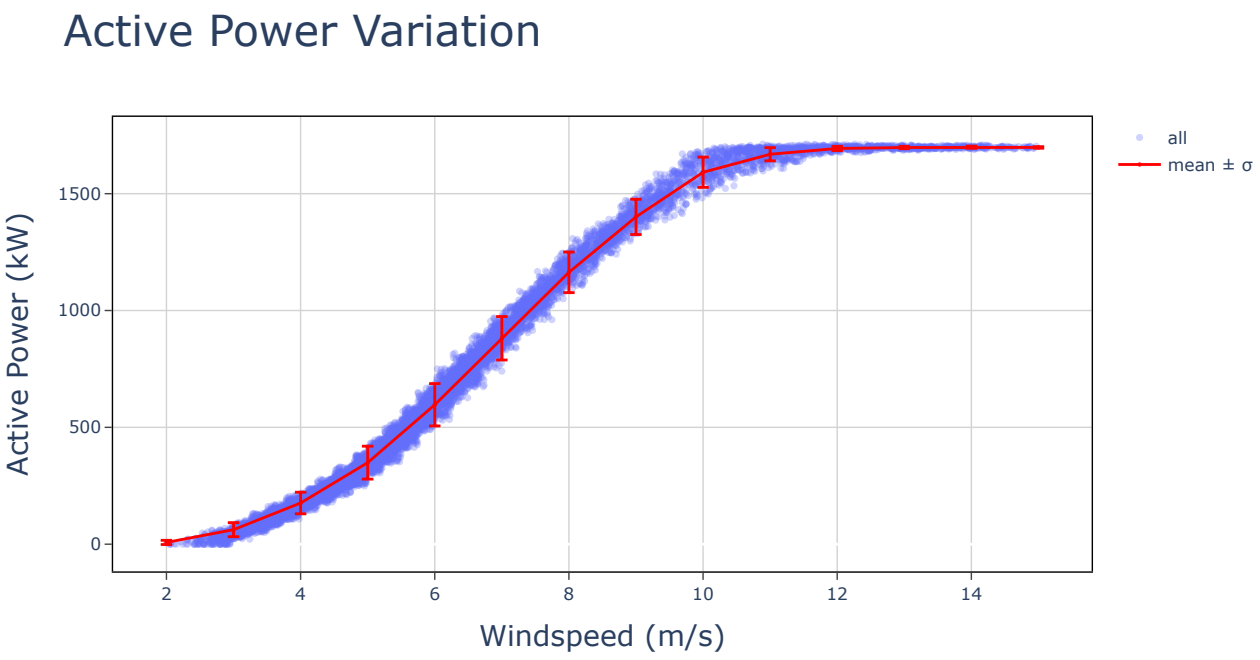
In [82]:
```
#matlibplot of power vs windspeed

xp = df2['Windspeed_Adjusted']
yp = df2['TM_ActivePower']
#sns.set(font_scale=1.2, font='DejaVu Sans')
fig, ax = plt.subplots(1,1, figsize=(20, 10))
p = ax.plot(xp, yp, marker='o', linestyle='none', markersize=5, alpha=0.3, label='all')
```

```
xp = df_var['windspeed_bin']
yp = df_var['TM_ActivePower']['mean']
err = df_var['TM_ActivePower']['std']
p = ax.errorbar(xp, yp,err, marker='o', linestyle='-', markersize=8, linewidth=4, label=r'mean $\pm\sigma$', zorder=3, c='re
p = ax.set_xlabel('Windspeed (m/s)', fontsize=20)
p = ax.set_ylabel('Active Power (kW)', fontsize=20)
p = ax.set_title('Active Power Variation', fontsize=30)
```



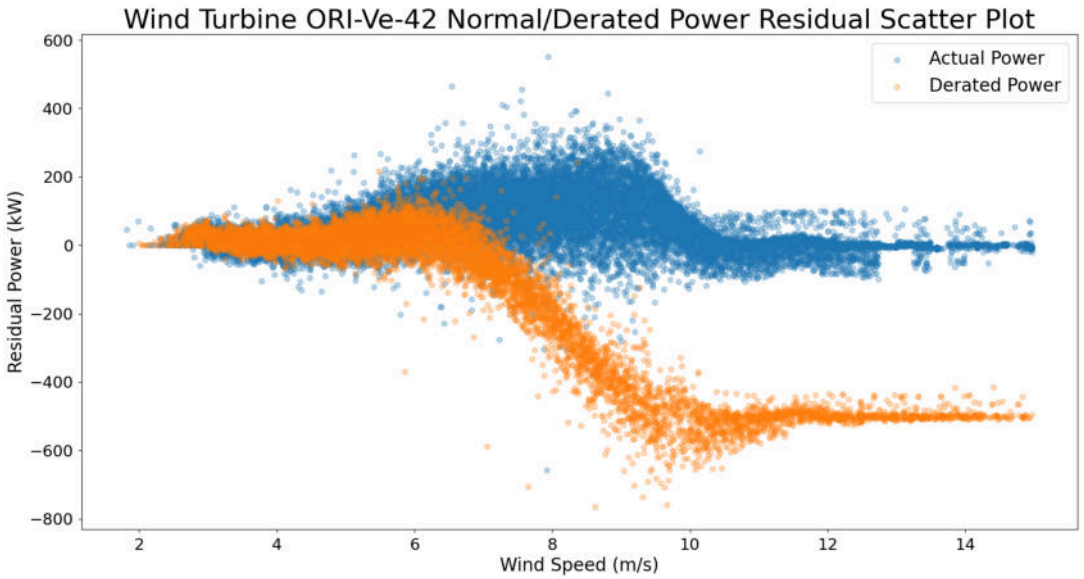
In [83]:
```
import plotly.graph_objects as go

xp = df2['Windspeed_Adjusted']
yp = df2['TM_ActivePower']

fig = go.Figure()

# Scatter plot for all data points
fig.add_trace(go.Scatter(x=xp, y=yp, mode='markers', marker=dict(size=5, opacity=0.3), name='all'))

# Line plot for mean with error bars
xp_mean = df_var['windspeed_bin']
yp_mean = df_var['TM_ActivePower']['mean']
err = df_var['TM_ActivePower']['std']
fig.add_trace(go.Scatter(x=xp_mean, y=yp_mean, mode='lines+markers', marker=dict(size=3), line=dict(width=2, color='red'),
                         name='mean ± σ', error_y=dict(type='data', array=err, visible=True)))

fig.update_layout(
    xaxis_title='Windspeed (m/s)',
    yaxis_title='Active Power (kW)',
    xaxis_title_font_size=20,
    yaxis_title_font_size=20,
    title_text='Active Power Variation',
    title_font_size=30
)

# fig.show()
fig.update_layout(plot_bgcolor='white',
    autosize=False,
    width=900,
    height=500)

fig.update_xaxes(
    mirror=True,
    ticks='outside',
    showline=True,
    linecolor='black',
    gridcolor='lightgrey'
)
fig.update_yaxes(
    mirror=True,
    ticks='outside',
    showline=True,
    linecolor='black',
```

```
    gridcolor='lightgrey'
)
```

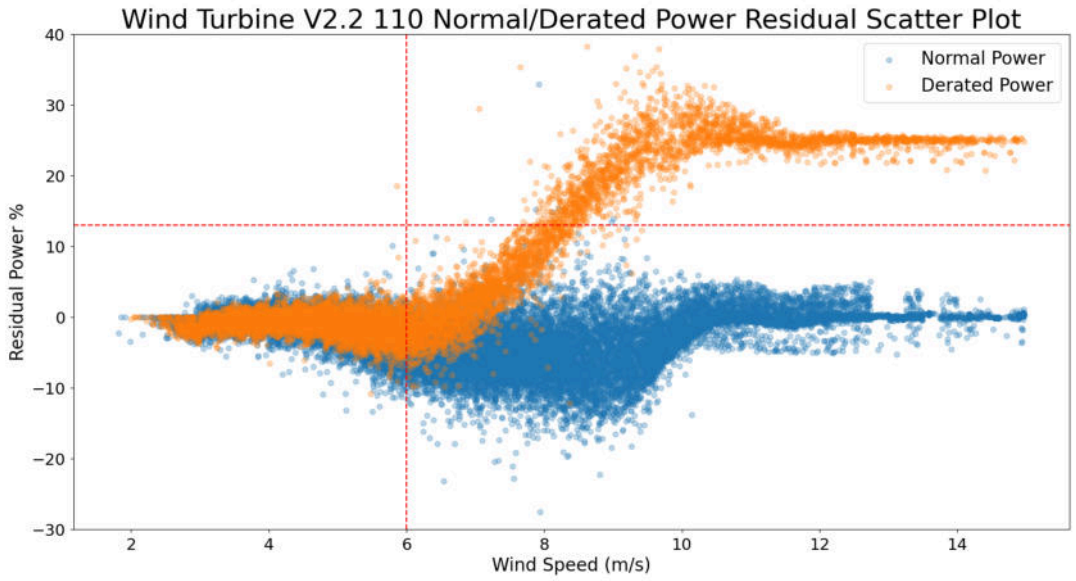


Compare Theoretical vs Active vs Derated power curve

In [84]:
```
df["Residual_Power"] = df["TM_ActivePower"] - df["TM_PossiblePower"]
df2["Residual_Power"] = df2["TM_ActivePower"] - df2["TM_PossiblePower"]

fig = plt.figure(figsize=(20,10))

plt.scatter(x = df['Windspeed_Adjusted'],y=df['Residual_Power'], label = 'Actual Power',alpha=0.3)
plt.scatter(x = df2['Windspeed_Adjusted'],y=df2['Residual_Power'], label = 'Derated Power',alpha=0.3)

plt.xlabel("Wind Speed (m/s)", fontsize=20)
plt.ylabel("Residual Power (kW)", fontsize=20)
plt.title("Wind Turbine ORI-Ve-42 Normal/Derated Power Residual Scatter Plot", fontsize=30)
plt.tick_params(labelsize=18, pad=6)
plt.legend(fontsize=20)
plt.show()
```



In [85]:
```
df["Residual_Power"] = 100*((df["TM_PossiblePower"]-df["TM_ActivePower"])/ 2000)
df2["Residual_Power"] = 100*((df2["TM_PossiblePower"]- df2["TM_ActivePower"])/ 2000)

fig = plt.figure(figsize=(20,10))

plt.scatter(x = df['Windspeed_Adjusted'],y=df['Residual_Power'], label = 'Normal Power',alpha=0.3, c="#1f77b4")
plt.scatter(x = df2['Windspeed_Adjusted'],y=df2['Residual_Power'], label = 'Derated Power',alpha=0.3, c="#ff7f0e")

plt.xlabel("Wind Speed (m/s)", fontsize=20)
plt.ylabel("Residual Power %", fontsize=20)

plt.ylim(-30,40)
plt.axhline(13, ls='--', lw=1.5, c='r')
plt.axvline(6, ls='--', lw=1.5, c='r')
plt.title("Wind Turbine V2.2 110 Normal/Derated Power Residual Scatter Plot", fontsize=30)
plt.tick_params(labelsize=18, pad=6)
plt.legend(fontsize=20)
plt.show()
```

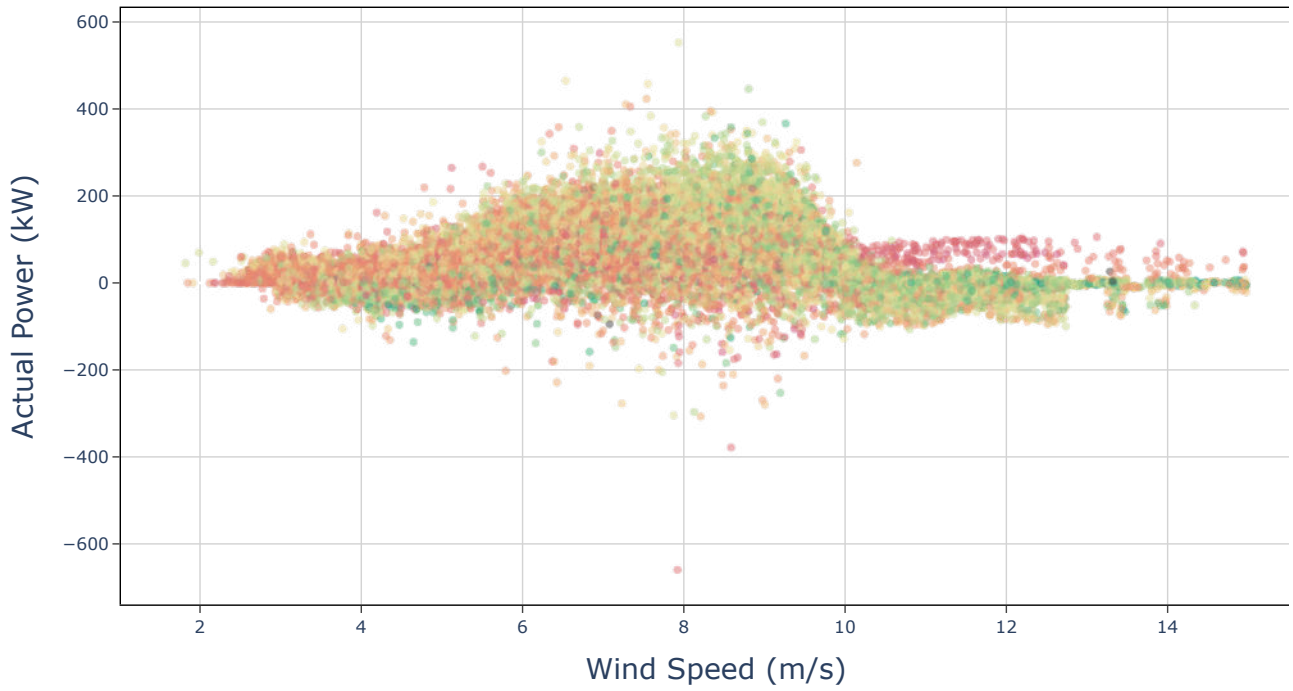


In [86]:
```
# import plotly.graph_objects as go

# df["Residual_Power"] = df["TM_ActivePower"] - df["TM_PossiblePower"]
# df2["Residual_Power"] = df2["TM_ActivePower"] - df2["TM_PossiblePower"]

# fig = go.Figure()

# # Scatter plot for residual power of normal operation
# fig.add_trace(go.Scatter(x=df['Windspeed_Adjusted'], y=df['Residual_Power'], mode='markers', marker=dict(opacity=0.3),
#                          name='Normal Operation'))

# # Scatter plot for residual power of derated operation
# fig.add_trace(go.Scatter(x=df2['Windspeed_Adjusted'], y=df2['Residual_Power'], mode='markers', marker=dict(opacity=0.3),
#                          name='Derated Operation'))

# fig.update_layout(
#     xaxis_title='Wind Speed (m/s)',
#     yaxis_title='Residual Power (kW)',
#     xaxis_title_font_size=20,
#     yaxis_title_font_size=20,
#     title_text='Wind Turbine ORI-Ve-42 Normal/Derated Power Residual Scatter Plot',
#     title_font_size=30,
#     legend_title_text='Operation Type',
#     legend_font_size=20
# )

# # fig.show()
# fig.update_layout(plot_bgcolor='white',
#     autosize=False,
#     width=900,
#     height=500)
```

```
# fig.update_xaxes(
#     mirror=True,
#     ticks='outside',
#     showline=True,
#     linecolor='black',
#     gridcolor='lightgrey'
# )
# fig.update_yaxes(
#     mirror=True,
#     ticks='outside',
#     showline=True,
#     linecolor='black',
#     gridcolor='lightgrey'
# )
```

In [87]:

```
import plotly.express as px

df["Residual_Power"] = df["TM_ActivePower"] - df["TM_PossiblePower"]

# Create a scatter plot using Plotly
fig = px.scatter(df, x='Windspeed_Adjusted', y='Residual_Power', color='TM_AmbientTemperature_Celsius',
                 color_continuous_scale='temps', opacity=0.5, title='Wind Turbine Derated Power Curve')

fig.update_layout(
    xaxis_title="Wind Speed (m/s)",
    yaxis_title="Actual Power (kW)",
    coloraxis_colorbar_title="Ambient Temperature (C)",
    coloraxis_colorbar_thickness=20,
    coloraxis_colorbar_tickfont_size=15,
    xaxis_title_font_size=20,
    yaxis_title_font_size=20,
    title_font_size=30
)

# fig.show()
fig.update_layout(plot_bgcolor='white',
    autosize=False,
    width=1200,
    height=600)

fig.update_xaxes(
    mirror=True,
    ticks='outside',
    showline=True,
    linecolor='black',
    gridcolor='lightgrey'
)
fig.update_yaxes(
    mirror=True,
    ticks='outside',
    showline=True,
    linecolor='black',
    gridcolor='lightgrey'
)
```



Modelling

In [88]:
```
from sklearn.model_selection import train_test_split
from sklearn.model_selection import KFold,RepeatedKFold
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import cross_val_predict

from sklearn import svm, tree, linear_model, neighbors, naive_bayes, ensemble, discriminant_analysis, gaussian_process
#from xgboost import XGBClassifier
#from lightgbm import LGBMRegressor

#Common Model Helpers
from sklearn.preprocessing import OneHotEncoder, LabelEncoder
from sklearn import feature_selection
from sklearn import model_selection
from sklearn import metrics

import warnings
warnings.filterwarnings('ignore')

from sklearn.preprocessing import StandardScaler
from sklearn.feature_selection import RFECV
from sklearn.model_selection import GridSearchCV
```

Untreated Dataset

In [89]:
```
df = df_10min[df_10min['AssetName']== 'ORI-Ve-42']
```

In [90]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 55506 entries, 2022-07-03 22:00:00 to 2023-06-30 09:40:00
Data columns (total 11 columns):
 #   Column                          Non-Null Count  Dtype  
---  ------                          --------------  -----  
 0   AssetParent                     55506 non-null  object 
 1   AssetName                       55506 non-null  object 
 2   TM_ActivePower                  51041 non-null  float64
 3   TM_PossiblePower                54666 non-null  float64
 4   State_Fault_Interpolated        55474 non-null  float64
 5   Windspeed_Adjusted              54936 non-null  float64
 6   Generator_RPM                   36349 non-null  float64
 7   TM_AmbientTemperature_Celsius   54600 non-null  float64
 8   RotorRPM                        54600 non-null  float64
 9   BladeAngle                      55066 non-null  float64
 10  TM_SetPoint                     55506 non-null  float64
dtypes: float64(9), object(2)
memory usage: 5.1+ MB
```

In [91]:
```
df = df_10min[df_10min['AssetName']== 'ORI-Ve-42']
df = df[df['State_Fault_Interpolated']== 2.0]
df = df[df['TM_SetPoint']>= 500]
df["TM_ActivePower"] = df["TM_ActivePower"].clip(lower=0)

split_date = '2023-03-25 00:00:00'
df['is_derated'] = np.where(df.index >= split_date, 'Yes', 'No')

df["Residual_Power"] = df["TM_ActivePower"] - df["TM_PossiblePower"]
```

In [92]:
```
df.is_derated = df.is_derated.map(dict(Yes=1, No=0))
df = df.sort_index(ascending=True)
```

In [93]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 38992 entries, 2022-07-01 00:00:00 to 2023-07-01 00:00:00
Data columns (total 13 columns):
 #   Column                          Non-Null Count  Dtype  
---  ------                          --------------  -----  
 0   AssetParent                     38992 non-null  object 
 1   AssetName                       38992 non-null  object 
 2   TM_ActivePower                  37144 non-null  float64
 3   TM_PossiblePower                38336 non-null  float64
 4   State_Fault_Interpolated        38992 non-null  float64
 5   Windspeed_Adjusted              38571 non-null  float64
 6   Generator_RPM                   23431 non-null  float64
 7   TM_AmbientTemperature_Celsius   38313 non-null  float64
 8   RotorRPM                        38313 non-null  float64
 9   BladeAngle                      38676 non-null  float64
 10  TM_SetPoint                     38992 non-null  float64
 11  is_derated                      38992 non-null  int64  
 12  Residual_Power                  36785 non-null  float64
dtypes: float64(10), int64(1), object(2)
memory usage: 4.2+ MB
```

In [ ]:
```
fig = plt.figure(figsize=(20,10))

plt.scatter(x = df['Windspeed_Adjusted'],y=df['TM_ActivePower'],
            c=df.is_derated.astype('category').cat.codes,alpha=0.3)

plt.xlabel("Wind Speed (m/s)", fontsize=20)
plt.ylabel("Power (kW)", fontsize=20)
plt.title("Training Dataset", fontsize=30)

plt.legend()

plt.show()
```

```
No artists with labels found to put in legend.  Note that artists whose label start with an underscore are ignored when legend() is called with no argument.
```

In [ ]:
```
# Define power curve's wind speeds
ws_cut_in=3.0  # cut-in wind speed
ws_rated=12.0  # rated wind speed
ws_cut_out=21.0  # cut-out wind speed
rated_power = 2200
```

In [ ]:
```
# import plotly.express as px
```

```
# # df = df_merged

# # Create a scatter plot using Plotly
# fig = px.scatter(df, x='Windspeed_Adjusted', y='TM_ActivePower', color='TM_AmbientTemperature_Celsius',
#                  color_continuous_scale='temps', opacity=0.3, title='Wind Turbine Untreated Power Curves')

# # Add vertical lines

# fig.add_shape(dict(type="line", x0=ws_cut_in, x1=ws_cut_in, y0=-100, y1=rated_power+100, line=dict(color="red", width=1.5,
# fig.add_shape(dict(type="line", x0=ws_rated, x1=ws_rated, y0=-100, y1=rated_power+100, line=dict(color="red", width=1.5, d
# fig.add_shape(dict(type="line", x0=ws_cut_out, x1=ws_cut_out, y0=-100, y1=rated_power+100, line=dict(color="red", width=1.

# fig.update_layout(
#     xaxis_title="Wind Speed (m/s)",
#     yaxis_title="Actual Power (kW)",
#     coloraxis_colorbar_title="Ambient Temperature (C)",
#     coloraxis_colorbar_thickness=20,
#     coloraxis_colorbar_tickfont_size=15,
#     xaxis_title_font_size=20,
#     yaxis_title_font_size=20,
#     title_font_size=30
# )

# # fig.show()
# fig.update_layout(plot_bgcolor='white',
#     autosize=False,
#     width=1500,
#     height=700)

# fig.update_xaxes(
#     mirror=True,
#     ticks='outside',
#     showline=True,
#     linecolor='black',
#     gridcolor='lightgrey'
# )
# fig.update_yaxes(
#     mirror=True,
#     ticks='outside',
#     showline=True,
#     linecolor='black',
#     gridcolor='lightgrey'
# )
```

In [ ]:

```
import plotly.express as px

# df = df_merged

# Create a scatter plot using Plotly
fig = px.scatter(df, x='Windspeed_Adjusted', y='TM_ActivePower', color='TM_AmbientTemperature_Celsius',
                 color_continuous_scale='temps', opacity=0.5, title='Wind Turbine Untreated Power Curves', facet_col=df.is_d

# Add vertical lines

fig.add_shape(dict(type="line", x0=ws_cut_in, x1=ws_cut_in, y0=-100, y1=rated_power+100, line=dict(color="red", width=1.5, d
fig.add_shape(dict(type="line", x0=ws_rated, x1=ws_rated, y0=-100, y1=rated_power+100, line=dict(color="red", width=1.5, das
fig.add_shape(dict(type="line", x0=ws_cut_out, x1=ws_cut_out, y0=-100, y1=rated_power+100, line=dict(color="red", width=1.5,

fig.update_layout(
    xaxis_title="Wind Speed (m/s)",
    yaxis_title="Actual Power (kW)",
    coloraxis_colorbar_title="Ambient Temperature (C)",
    coloraxis_colorbar_thickness=20,
    coloraxis_colorbar_tickfont_size=15,
    xaxis_title_font_size=20,
    yaxis_title_font_size=20,
    title_font_size=30
)

# fig.show()
fig.update_layout(plot_bgcolor='white',
    autosize=False,
    width=1100,
    height=500)

fig.update_xaxes(
    mirror=True,
    ticks='outside',
    showline=True,
```

```
    linecolor='black',
    gridcolor='lightgrey'
)
fig.update_yaxes(
    mirror=True,
    ticks='outside',
    showline=True,
    linecolor='black',
    gridcolor='lightgrey'
)
```

Train/Test Split

In [ ]:
```
df = df[["Windspeed_Adjusted", "TM_AmbientTemperature_Celsius","Residual_Power","TM_ActivePower","is_derated"]]
df.info()
```

In [ ]:
```
# ### Tranform training data
# df = df_sub.apply(pd.to_numeric) # convert all columns of DataFrame
df = df.dropna()
x = df[["Windspeed_Adjusted", "TM_ActivePower","TM_AmbientTemperature_Celsius","Residual_Power"]]
y = df['is_derated']
```

In [ ]:
```
x_train,x_test,y_train,y_test = train_test_split(x, y, test_size=0.25, random_state=13)
```

In [ ]:
```
x_train.shape, x_test.shape
```

In [ ]:
```
y_test.value_counts(ascending=True)
```

Feature Scaling

In [ ]:
```
cols = x_train.columns
```

In [ ]:
```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

x_train = scaler.fit_transform(x_train)

x_test = scaler.transform(x_test)
```

In [ ]:
```
x_train = pd.DataFrame(x_train, columns=[cols])
```

In [ ]:
```
x_test = pd.DataFrame(x_test, columns=[cols])
```

In [ ]:
```
x_train.head()
```

Fit K Neighbours Classifier to the training set

In [ ]:
```
# import KNeighbors ClaSSifier from sklearn
from sklearn.neighbors import KNeighborsClassifier


# instantiate the model
knn = KNeighborsClassifier(n_neighbors=2)


# fit the model to the training set
knn.fit(x_train, y_train)
```

Predict test-set results

In [ ]:
```
y_pred = knn.predict(x_test)

y_pred
```

In [ ]:
```
# probability of getting output as Yes - Derated operation
knn.predict_proba(x_test)[:,0]
```

In [ ]:
```
# probability of getting output as No - Normal operation

knn.predict_proba(x_test)[:,1]
```

In [ ]:
```
from sklearn.metrics import accuracy_score

print('Model accuracy score: {0:0.4f}'. format(accuracy_score(y_test, y_pred)))
```

Compare the train-set and test-set accuracy

In [ ]:
```
y_pred_train = knn.predict(x_train)
```

In [ ]:
```
print('Training-set accuracy score: {0:0.4f}'. format(accuracy_score(y_train, y_pred_train)))
```

Check for overfitting and underfitting

In [ ]:
```
# print the scores on training and test set

print('Training set score: {:.4f}'.format(knn.score(x_train, y_train)))

print('Test set score: {:.4f}'.format(knn.score(x_test, y_test)))
```

Confusion matrix

A confusion matrix is a tool for summarizing the performance of a classification algorithm. A confusion matrix will give us a clear picture of classification model performance and the types of errors produced by the model. It gives us a summary of correct and incorrect predictions broken down by each category. The summary is represented in a tabular form.

Four types of outcomes are possible while evaluating a classification model performance. These four outcomes are described below:-

True Positives (TP) – True Positives occur when we predict an observation belongs to a certain class and the observation actually belongs to that class.

True Negatives (TN) – True Negatives occur when we predict an observation does not belong to a certain class and the observation actually does not belong to that class.

False Positives (FP) – False Positives occur when we predict an observation belongs to a certain class but the observation actually does not belong to that class. This type of error is called Type I error.

False Negatives (FN) – False Negatives occur when we predict an observation does not belong to a certain class but the observation actually belongs to that class. This is a very serious error and it is called Type II error.

These four outcomes are summarized in a confusion matrix given below.

In [ ]:
```
# Print the Confusion Matrix with k =2 and slice it into four pieces

from sklearn.metrics import confusion_matrix

cm = confusion_matrix(y_test, y_pred)

print('Confusion matrix\n\n', cm)

print('\nTrue Positives(TP) = ', cm[0,0])

print('\nTrue Negatives(TN) = ', cm[1,1])

print('\nFalse Positives(FP) = ', cm[0,1])

print('\nFalse Negatives(FN) = ', cm[1,0])
```

In [ ]:
```
y_test.value_counts(ascending=True)
```

In [ ]:
```
# visualize confusion matrix with seaborn heatmap

plt.figure(figsize=(10,6))

cm_matrix = pd.DataFrame(data=cm, columns=['Actual Normal Operation:1', 'Actual Derated Operation'],
                                 index=['Predict Normal Operation:1', 'Predict Derated Operation'])

sns.heatmap(cm_matrix, annot=True, fmt='d', cmap='YlGnBu')
```

In [ ]:
```
error_rate = []
# Will take some time
for i in range(1,40):

 knn = KNeighborsClassifier(n_neighbors=i)
```

```
 knn.fit(x_train,y_train)
 pred_i = knn.predict(x_test)
 error_rate.append(np.mean(pred_i != y_test))
```

In [ ]:
```
plt.figure(figsize=(10,6))
plt.plot(range(1,40),error_rate,color="blue", linestyle="dashed", marker="o",
 markerfacecolor="red", markersize=10)
plt.title('Error Rate vs. K Value')
plt.xlabel('K')
plt.ylabel('Error Rate')
```

In [ ]:
```
# import KNeighbors ClaSSifier from sklearn
from sklearn.neighbors import KNeighborsClassifier


# instantiate the model
knn = KNeighborsClassifier(n_neighbors=19)


# fit the model to the training set
knn.fit(x_train, y_train)
```

In [ ]:
```
# print the scores on training and test set

print('Training set score: {:.4f}'.format(knn.score(x_train, y_train)))

print('Test set score: {:.4f}'.format(knn.score(x_test, y_test)))
```

In [ ]:
```
y_pred = knn.predict(x_test)

y_pred
```

In [ ]:
```
from sklearn.metrics import classification_report

print(classification_report(y_test, y_pred))
```

Treated Dataset

In [ ]:
```
df_merged = pd.concat([turbine_df_clean,turbine_df_clean_derate], axis=0)
df_merged = df_merged.sort_index(ascending=True)
```

In [ ]:
```
df_merged.info()
```

In [ ]:
```
df_merged.is_derated = df_merged.is_derated.map(dict(Yes=1, No=0))
```

In [ ]:
```
df_merged.head(10)
```

In [ ]:
```
df_merged.sample(5)
```

In [ ]:
```
#  df_merged.to_csv("df_merged.csv")
```

In [ ]:
```
df_merged.is_derated.value_counts()
```

In [ ]:
```
# Define power curve's wind speeds
ws_cut_in=3.0  # cut-in wind speed
ws_rated=12.0  # rated wind speed
ws_cut_out=21.0  # cut-out wind speed
rated_power = 2200
```

In [ ]:
```
# import plotly.express as px

# df = df_merged

# # Create a scatter plot using Plotly
# fig = px.scatter(df, x='Windspeed_Adjusted', y='TM_ActivePower', color='TM_AmbientTemperature_Celsius',
#                  color_continuous_scale='temps', opacity=0.5, title='Wind Turbine Treated Power Curves')

# # Add vertical lines

# fig.add_shape(dict(type="line", x0=ws_cut_in, x1=ws_cut_in, y0=-100, y1=rated_power+100, line=dict(color="red", width=1.5,
# fig.add_shape(dict(type="line", x0=ws_rated, x1=ws_rated, y0=-100, y1=rated_power+100, line=dict(color="red", width=1.5, d
# fig.add_shape(dict(type="line", x0=ws_cut_out, x1=ws_cut_out, y0=-100, y1=rated_power+100, line=dict(color="red", width=1.

# fig.update_layout(
```

```
#     xaxis_title="Wind Speed (m/s)",
#     yaxis_title="Actual Power (kW)",
#     coloraxis_colorbar_title="Ambient Temperature (C)",
#     coloraxis_colorbar_thickness=20,
#     coloraxis_colorbar_tickfont_size=15,
#     xaxis_title_font_size=20,
#     yaxis_title_font_size=20,
#     title_font_size=30
# )

# # fig.show()
# fig.update_layout(plot_bgcolor='white',
#     autosize=False,
#     width=1500,
#     height=700)

# fig.update_xaxes(
#     mirror=True,
#     ticks='outside',
#     showline=True,
#     linecolor='black',
#     gridcolor='lightgrey'
# )
# fig.update_yaxes(
#     mirror=True,
#     ticks='outside',
#     showline=True,
#     linecolor='black',
#     gridcolor='lightgrey'
# )
```

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

df = df_merged

# Create a scatter plot using Plotly
fig = px.scatter(df, x='Windspeed_Adjusted', y='TM_ActivePower', color='TM_AmbientTemperature_Celsius',
                 color_continuous_scale='temps', opacity=0.5, title='Wind Turbine Treated Power Curves', facet_col=df.is_der

# Add vertical lines

fig.add_shape(dict(type="line", x0=ws_cut_in, x1=ws_cut_in, y0=-100, y1=rated_power+100, line=dict(color="red", width=1.5, d
fig.add_shape(dict(type="line", x0=ws_rated, x1=ws_rated, y0=-100, y1=rated_power+100, line=dict(color="red", width=1.5, das
fig.add_shape(dict(type="line", x0=ws_cut_out, x1=ws_cut_out, y0=-100, y1=rated_power+100, line=dict(color="red", width=1.5,


fig.update_layout(
    xaxis_title="Wind Speed (m/s)",
    yaxis_title="Actual Power (kW)",
    coloraxis_colorbar_title="Ambient Temperature (C)",
    coloraxis_colorbar_thickness=20,
    coloraxis_colorbar_tickfont_size=15,
    xaxis_title_font_size=20,
    yaxis_title_font_size=20,
    title_font_size=30
)

# fig.show()
fig.update_layout(plot_bgcolor='white',
    autosize=False,
    width=1100,
    height=500)

fig.update_xaxes(
    mirror=True,
    ticks='outside',
    showline=True,
    linecolor='black',
    gridcolor='lightgrey'
)
fig.update_yaxes(
    mirror=True,
    ticks='outside',
    showline=True,
    linecolor='black',
    gridcolor='lightgrey'
)
```

Train/Test Split

In [ ]:
```
df = df_merged
```

In [ ]:
```
df = df[["TM_ActivePower","Windspeed_Adjusted","TM_AmbientTemperature_Celsius", "Residual_Power","is_derated"]]
# df = df[["TM_ActivePower","Windspeed_Adjusted","Residual_Power","is_derated"]]
df.info()
```

In [ ]:
```
# ### Tranform training data
# df = df_sub.apply(pd.to_numeric) # convert all columns of DataFrame
df = df.dropna()
x = df[["Windspeed_Adjusted", "TM_AmbientTemperature_Celsius","TM_ActivePower","Residual_Power"]]
# x = df[["Windspeed_Adjusted", "TM_ActivePower","Residual_Power"]]
y = df['is_derated']
```

In [ ]:
```
x_train,x_test,y_train,y_test = train_test_split(x, y, test_size=0.25, random_state=13)
```

In [ ]:
```
x_train.shape, x_test.shape
```

In [ ]:
```
y_test.value_counts(ascending=True)
```

In [ ]:
```
# x_train.to_csv("x_train.csv")
# x_test.to_csv("x_test.csv")
# y_train.to_csv("y_train.csv")
# y_test.to_csv("y_test.csv")
```

Feature Scaling

In [ ]:
```
cols = x_train.columns
```

In [ ]:
```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

x_train = scaler.fit_transform(x_train)

x_test = scaler.transform(x_test)
```

In [ ]:
```
x_train = pd.DataFrame(x_train, columns=[cols])
```

In [ ]:
```
x_test = pd.DataFrame(x_test, columns=[cols])
```

In [ ]:
```
x_train.head()
```

kNN

In machine learning, k Nearest Neighbours or kNN is the simplest of all machine learning algorithms. It is a non-parametric algorithm used for classification and regression tasks. Non-parametric means there is no assumption required for data distribution. So, kNN does not require any underlying assumption to be made. In both classification and regression tasks, the input consists of the k closest training examples in the feature space. The output depends upon whether kNN is used for classification or regression purposes.

- In kNN classification, the output is a class membership. The given data point is classified based on the majority of type of its neighbours. The data point is assigned to the most frequent class among its k nearest neighbours. Usually k is a small positive integer. If k=1, then the data point is simply assigned to the class of that single nearest neighbour.
- In kNN regression, the output is simply some property value for the object. This value is the average of the values of k nearest neighbours.

kNN is a type of instance-based learning or lazy learning. Lazy learning means it does not require any training data points for model generation. All training data will be used in the testing phase. This makes training faster and testing slower and costlier. So, the testing phase requires more time and memory resources.

In kNN, the neighbours are taken from a set of objects for which the class or the object property value is known. This can be thought of as the training set for the kNN algorithm, though no explicit training step is required. In both classification and regression kNN algorithm, we can assign weight to the contributions of the neighbours. So, nearest neighbours contribute more to the average than the more distant ones.

The kNN algorithm intuition is very simple to understand. It simply calculates the distance between a sample data point and all the other training data points. The distance can be Euclidean distance or Manhattan distance. Then, it selects the k nearest data points where k can be any integer. Finally, it assigns the sample data point to the class to which the majority of the k data points belong.

Now, we will see kNN algorithm in action. Suppose, we have a dataset with two variables which are classified as Red and Blue.

In kNN algorithm, k is the number of nearest neighbours. Generally, k is an odd number because it helps to decide the majority of the class. When k=1, then the algorithm is known as the nearest neighbour algorithm.

Now, we want to classify a new data point X into Blue class or Red class. Suppose the value of k is 3. The kNN algorithm starts by calculating the distance between X and all the other data points. It then finds the 3 nearest points with least distance to point X.

In the final step of the kNN algorithm, we assign the new data point X to the majority of the class of the 3 nearest points. If 2 of the 3 nearest points belong to the class Red while 1 belong to the class Blue, then we classify the new data point as Red.

Fit K Neighbours Classifier to the training set

In [ ]:
```
# import KNeighbors ClaSSifier from sklearn
from sklearn.neighbors import KNeighborsClassifier


# instantiate the model
knn = KNeighborsClassifier(n_neighbors=2)


# fit the model to the training set
knn.fit(x_train, y_train)
```

Predict test-set results

In [ ]:
```
y_pred = knn.predict(x_test)

y_pred
```

In [ ]:
```
# probability of getting output as Yes - Derated operation
knn.predict_proba(x_test)[:,0]
```

In [ ]:
```
# probability of getting output as No - Normal operation

# knn.predict_proba(x_test)[:,1]
```

In [ ]:
```
from sklearn.metrics import accuracy_score

print('Model accuracy score: {0:0.4f}'. format(accuracy_score(y_test, y_pred)))
```

Compare the train-set and test-set accuracy

In [ ]:
```
y_pred_train = knn.predict(x_train)
```

In [ ]:
```
print('Training-set accuracy score: {0:0.4f}'. format(accuracy_score(y_train, y_pred_train)))
```

Check for overfitting and underfitting

In [ ]:
```
# print the scores on training and test set

print('Training set score: {:.4f}'.format(knn.score(x_train, y_train)))

print('Test set score: {:.4f}'.format(knn.score(x_test, y_test)))
```

Confusion matrix

A confusion matrix is a tool for summarizing the performance of a classification algorithm. A confusion matrix will give us a clear picture of classification model performance and the types of errors produced by the model. It gives us a summary of correct and incorrect predictions broken down by each category. The summary is represented in a tabular form.

Four types of outcomes are possible while evaluating a classification model performance. These four outcomes are described below:-

True Positives (TP) – True Positives occur when we predict an observation belongs to a certain class and the observation actually belongs to that class.

True Negatives (TN) – True Negatives occur when we predict an observation does not belong to a certain class and the observation actually does not belong to that class.

False Positives (FP) – False Positives occur when we predict an observation belongs to a certain class but the observation actually does not belong to that class. This type of error is called Type I error.

False Negatives (FN) – False Negatives occur when we predict an observation does not belong to a certain class but the observation actually belongs to that class. This is a very serious error and it is called Type II error.

These four outcomes are summarized in a confusion matrix given below.

In [ ]:
```
# Print the Confusion Matrix with k =2 and slice it into four pieces

from sklearn.metrics import confusion_matrix

cm = confusion_matrix(y_test, y_pred)

print('Confusion matrix\n\n', cm)

print('\nTrue Positives(TP) = ', cm[0,0])

print('\nTrue Negatives(TN) = ', cm[1,1])

print('\nFalse Positives(FP) = ', cm[0,1])

print('\nFalse Negatives(FN) = ', cm[1,0])
```

In [ ]:
```
from sklearn.metrics import classification_report

print(classification_report(y_test, y_pred))
```

In [ ]:
```
y_test.value_counts(ascending=True)
```

In [ ]:
```
# visualize confusion matrix with seaborn heatmap

plt.figure(figsize=(10,6))

cm_matrix = pd.DataFrame(data=cm, columns=['Actual Normal Operation:1', 'Actual Derated Operation'],
                                 index=['Predict Normal Operation:1', 'Predict Derated Operation'])

sns.heatmap(cm_matrix, annot=True, fmt='d', cmap='YlGnBu')
```

While building the kNN classifier model, one question that come to my mind is what should be the value of nearest neighbours (k) that yields highest accuracy. This is a very important question because the classification accuracy depends upon our choice of k.

The number of neighbours (k) in kNN is a parameter that we need to select at the time of model building. Selecting the optimal value of k in kNN is the most critical problem. A small value of k means that noise will have higher influence on the result. So, probability of overfitting is very high. A large value of k makes it computationally expensive in terms of time to build the kNN model. Also, a large value of k will have a smoother decision boundary which means lower variance but higher bias.

The data scientists choose an odd value of k if the number of classes is even. We can apply the elbow method to select the value of k. To optimize the results, we can use Cross Validation technique. Using the cross-validation technique, we can test the kNN algorithm with different values of k. The model which gives good accuracy can be considered to be an optimal choice. It depends on individual cases and at times best process is to run through each possible value of k and test our result.

In [ ]:
```
error_rate = []
# Will take some time
for i in range(1,40):

 knn = KNeighborsClassifier(n_neighbors=i)
 knn.fit(x_train,y_train)
 pred_i = knn.predict(x_test)
 error_rate.append(np.mean(pred_i != y_test))
```

In [ ]:
```
plt.figure(figsize=(10,6))
plt.plot(range(1,40),error_rate,color="blue", linestyle="dashed", marker="o",
 markerfacecolor="red", markersize=10)
plt.title('Error Rate vs. K Value')
plt.xlabel('K')
plt.ylabel('Error Rate')
```

In [ ]:
```
# instantiate the model with k=5
knn_16 = KNeighborsClassifier(n_neighbors=16)

# fit the model to the training set
knn_16.fit(x_train, y_train)

# predict on the test-set
y_pred_16 = knn_16.predict(x_test)


print('Model accuracy score with k=16 : {0:0.4f}'. format(accuracy_score(y_test, y_pred_16)))
```

Compare the train-set and test-set accuracy

In [ ]:
```
y_pred = knn_16.predict(x_test)

y_pred
```

In [ ]:
```
from sklearn.metrics import accuracy_score

print('Model accuracy score: {0:0.4f}'. format(accuracy_score(y_test, y_pred)))
```

In [ ]:
```
y_pred_train = knn_16.predict(x_train)
```

In [ ]:
```
print('Training-set accuracy score: {0:0.4f}'. format(accuracy_score(y_train, y_pred_train)))
```

In [ ]:
```
# print the scores on training and test set

print('Training set score: {:.4f}'.format(knn_16.score(x_train, y_train)))

print('Test set score: {:.4f}'.format(knn_16.score(x_test, y_test)))
```

In [ ]:
```
from sklearn.metrics import classification_report

print(classification_report(y_test, y_pred))
```

SVM

In [ ]:
```
from sklearn import svm

clf = svm.SVC()
clf.fit(x_train, y_train)

# predict on the test-set
y_pred = clf.predict(x_test)

print('Model accuracy score with svm : {0:0.4f}'. format(accuracy_score(y_test, y_pred)))
```

In [ ]:
```
# x_test.sample(1)
```

In [ ]:
```
# print the scores on training and test set

print('Training set score: {:.4f}'.format(clf.score(x_train, y_train)))

print('Test set score: {:.4f}'.format(clf.score(x_test, y_test)))
```

In [ ]:
```
from sklearn.metrics import classification_report

print(classification_report(y_test, y_pred))
```

In [ ]:
```
# pd.DataFrame(y_pred).to_csv("y_pred.csv")
```

Decision Trees

In [ ]:
```
from sklearn.tree import DecisionTreeClassifier

clf = DecisionTreeClassifier(criterion='entropy', max_depth=3, random_state=0)
clf.fit(x_train, y_train)

# predict on the test-set
y_pred = clf.predict(x_test)
```

```
print('Model accuracy score with decision tree : {0:0.4f}'. format(accuracy_score(y_test, y_pred)))
```

In [ ]:
```
# print the scores on training and test set

print('Training set score: {:.4f}'.format(clf.score(x_train, y_train)))

print('Test set score: {:.4f}'.format(clf.score(x_test, y_test)))
```

In [ ]:
```
from sklearn.metrics import classification_report

print(classification_report(y_test, y_pred))
```

In [ ]:
```
text_representation = tree.export_text(clf)
print(text_representation)
```

General Detection Algorithm

In [ ]:
```
### Merged Dataset
wtg_metadata = pd.read_csv('./wtg_static_data.csv')

del wtg_metadata['AssetParent']
```

In [ ]:
```
df_batch = pd.merge(merged_df,
              wtg_metadata,
              left_on='AssetName',
              right_on='AssetName',
              how='left')
```

In [ ]:
```
df_batch.head()
```

In [ ]:
```
df_batch.info()
```

Set Parameters

In [ ]:
```
############---Rev---##############

ptc_value= 38.690
onm_value= 0.340
print(ptc_value - onm_value)

residual_normal_threshold = 2.5
residual_derate_threshold = 13.0

rated_windspeed_reduction = 3.0
```

Create Helper Functions

In [ ]:
```
def derate_preprocessing(df, active_status, unactive_status, datecol):
    ### Calculate Timestamp transformations
    df[datecol]=pd.to_datetime(df[datecol])
    df['date']=df[datecol].dt.date

    df['TM_ActivePower'] = pd.to_numeric(df['TM_ActivePower'], errors = 'coerce')
    df["TM_ActivePower"] = df["TM_ActivePower"].clip(lower=0) ### Remove negative active power values

    df['TM_PossiblePower'] = pd.to_numeric(df['TM_PossiblePower'], errors = 'coerce')
    df['State_Fault_Interpolated'] = pd.to_numeric(df['State_Fault_Interpolated'], errors = 'coerce')
    df['TM_SetPoint'] = pd.to_numeric(df['TM_SetPoint'], errors = 'coerce')
    df["TM_SetPoint"] = df["TM_SetPoint"].astype(int, errors = 'ignore') ####---Rev 7---####
    df['Windspeed_Adjusted'] = pd.to_numeric(df['Windspeed_Adjusted'], errors = 'coerce')
    df['operating_capacity'] = pd.to_numeric(df['operating_capacity'], errors = 'coerce')
    df['Rated_Windspeed'] = pd.to_numeric(df['Rated_Windspeed'], errors = 'coerce')
    df['InterconnectCapacity'] = df['InterconnectCapacity'].astype(int)

    df['TM_ExpectedPower'] = df[['TM_PossiblePower','operating_capacity']].min(axis=1) ####---Rev 8---####

    conditions  = [
                   (df['TM_SetPoint'].lt(df['InterconnectCapacity'])),  ### Site Curtailment
                   df['State_Fault_Interpolated'] == unactive_state,  ###Inactive 7.0 System OK/Weather
                   df['State_Fault_Interpolated'] != active_state,  ###Offline
                   df['TM_ActivePower'] <= 1.0,  ###Not Producing
```

```
                    df['State_Fault_Interpolated'] == active_state ###Active 2.0
                    ]

    choices     = ['Site Curtailment', 'Inactive', 'Offline', 'Not Producing', 'Active']

    df["status"] = np.select(conditions, choices, default=np.nan)

    return df

def derate_algorithm(df, residual_derate_threshold, residual_normal_threshold, rated_windspeed_reduction):
    df["Residual_Power_perc"] = 100*((df["TM_ExpectedPower"]-df["TM_ActivePower"]) / df['operating_capacity']) ####---Rev 9-

    df["underperformance"] = np.where((df['status'] == 'Active') &
                            (df['Residual_Power_perc'] >= residual_derate_threshold),
                            'Yes', 'No')

    df["derated"] = np.where((df['status'] == 'Active') &
                            (df['Windspeed_Adjusted'] >= (df['Rated_Windspeed'] - rated_windspeed_reduction)) &
                            (df['Residual_Power_perc'] >= residual_derate_threshold),
                            'Yes', 'No')

    df["normal"] = np.where((df['status'] == 'Active') &
                            (df['TM_ActivePower'] >= (df['operating_capacity'] - 50.0)),
                            'Yes', 'No')

    return df

def create_metric(df, datecol):
    ##########################################
    #########--- Calculate Asset Age ---##############
    ##########################################
#     df['age'] = (df[datecol] - df['install_date']).dt.days // 365
#     ### Evaluate PTC Eligibility
#     df["ptc_eligible"] = np.where((df['age'] <= 10),
#                               'Yes', 'No')

    ##########################################
    #########--- PTC Inclusion Script ---##############
    ##########################################
#     df["RT_LMP"] = np.where((df['ptc_eligible'] == 'Yes') & (df["RT_LMP"] > -(ptc_value - onm_value)),
#                             df["RT_LMP"] + ptc_value, df["RT_LMP"])

#     df["RT_LMP"] = df["RT_LMP"].clip(lower=0)

    ##########################################
    #########--- Lost Capacity ---###################
    ##########################################
    ### Calculate Lost Capacity
    df["LostCapacity"] = np.where((df['derated'] == 'Yes'),
                              df["operating_capacity"] - df["TM_ActivePower"], 0)

#     ##########################################
#     #########--- Lost Energy ---#####################
#     ##########################################
#     ### Calculate Lost Energy for Underperformance
#     df["LostEnergy_up"] = np.where((df['underperformance'] == 'Yes'),
#                                    df["TM_ExpectedPower"] - df["TM_ActivePower"], 0) ####---Rev 9---####

#     ### Convert kW to kWh by dividing by 6
#     df["LostEnergy_up"] = df["LostEnergy_up"]/6


    ### Calculate Lost Energy for Derates
    df["LostEnergy"] = np.where((df['derated'] == 'Yes'),
                              df["TM_ExpectedPower"] - df["TM_ActivePower"], 0) ####---Rev 9---####

    ### Convert kW to kWh by dividing by 6
    df["LostEnergy"] = df["LostEnergy"]/6

#     ##########################################
#     #########--- Lost Market Potential ---##############
#     ##########################################
#     df["LostMarketPotential_up"] = np.where((df['underperformance'] == 'Yes'),
#                                             df["LostEnergy_up"] * (df["RT_LMP"] / 1000), 0)

#     df["LostMarketPotential"] = np.where((df['derated'] == 'Yes'),
#                                          df["LostEnergy"] * (df["RT_LMP"] / 1000), 0)

    return df
```

In [ ]:
```
active_state = 2.0
unactive_state = 7.0

processed_data = derate_preprocessing(df = df_batch,
                                      active_status=active_state,
                                      unactive_status=unactive_state,
                                      datecol='Timestamp')
```

In [ ]:
```
derate_df = derate_algorithm(df = processed_data,
                             residual_derate_threshold=residual_derate_threshold,
                             residual_normal_threshold=residual_normal_threshold,
                             rated_windspeed_reduction=rated_windspeed_reduction)
derate_df['Derate_Timestamp'] = np.where(derate_df["derated"]=='Yes',derate_df['Timestamp'], pd.NaT)
derate_df["Derate_Timestamp"]= pd.to_datetime(derate_df["Derate_Timestamp"])
```

In [ ]:
```
derate_df = create_metric(df = derate_df, datecol='Timestamp')
```

In [ ]:
```
# derate_df[derate_df['derated']=='Yes'].head()
```

In [ ]:
```
derate_counts = (
    derate_df
    .assign(st=derate_df["status"]=='Active')
    .assign(rt = derate_df['Windspeed_Adjusted'] >= (derate_df['Rated_Windspeed'] - rated_windspeed_reduction))
    .assign(dr=derate_df["derated"]=='Yes')
    .assign(nr=derate_df["normal"]=='Yes')
    .assign(up=derate_df["underperformance"]=='Yes')
#     .assign(uplenergy=np.where(derate_df["underperformance"]=='Yes',derate_df.LostEnergy_up,0))
#     .assign(uplmarket=np.where(derate_df["underperformance"]=='Yes',derate_df.LostMarketPotential_up,0))
    .assign(pp=np.where(derate_df["derated"]=='Yes',derate_df.TM_ActivePower,np.nan)) ####---Rev 9---####
    .assign(lc=np.where(derate_df["derated"]=='Yes',derate_df.LostCapacity,np.nan))
    .assign(lenergy=np.where(derate_df["derated"]=='Yes',derate_df.LostEnergy,0))
#     .assign(lmarket=np.where(derate_df["derated"]=='Yes',derate_df.LostMarketPotential,0))
    .groupby(['AssetParent','AssetName','Rated_Windspeed','date'])
    .agg(
        Count=("dr", "size"),
        Active=("st", "sum"),
        Normal=("nr", "sum"),
        Underperformance=("up", "sum"),
#         Underperformance_LostEnergy_kwh=("uplenergy", "sum"),
#         Underperformance_LostMarketPotential_usd=("uplmarket", "sum"),
        RatedWindspeedCount=("rt", "sum"),
        Derated=("dr", "sum"),
        PeakPower_kw=("pp", "max"), ####---Rev 9---####
        LostCapacity_kw=("lc", "mean"),
        LostEnergy_kwh=("lenergy", "sum")
#         LostMarketPotential_usd=("lmarket", "sum")
         )
    .reset_index()
    )

derate_counts["ActivePerc"] = round(100*(derate_counts["Active"]/ 144),2)
derate_counts["NormalPerc"] = round(100*(derate_counts["Normal"]/ derate_counts['RatedWindspeedCount']),2)
derate_counts["UpPerc"] = round(100*(derate_counts["Underperformance"]/ derate_counts['Active']),2)
derate_counts["DeratedPerc"] = round(100*(derate_counts["Derated"]/ derate_counts['RatedWindspeedCount']),2)
derate_counts["TotalPerc"] = round(derate_counts["ActivePerc"]*(derate_counts["DeratedPerc"]/100),2)
```

In [ ]:
```
# display(derate_counts.sort_values(by=['TotalPerc','NormalPerc'], ascending=False))
```

In [ ]:
```
derate_counts['is_derated'] = np.where((derate_counts['TotalPerc'] >= 24) & (derate_counts['Derated'] >= 40), 'Yes', 'No') #
derate_counts['is_normal'] = np.where((derate_counts['Normal'] >=12), 'Yes', 'No') ### Approx 2 hours

derate_counts['Normal_Date'] = np.where(derate_counts["is_normal"]=='Yes',derate_counts.date, np.nan)
derate_counts['Derate_Date'] = np.where(derate_counts["is_derated"]=='Yes',derate_counts.date, np.nan)

### Calculate core metrics when derated
derate_counts["PeakPower_kw"] = np.where(derate_counts["is_derated"]=='Yes', derate_counts.PeakPower_kw,np.nan)
derate_counts["LostCapacity_kw"] = np.where(derate_counts["is_derated"]=='Yes', derate_counts.LostCapacity_kw,np.nan)
derate_counts["LostTime_hrs"] = np.where(derate_counts["is_derated"]=='Yes', round(((derate_counts["Derated"])/6),2),0)
# derate_counts["LostEnergy_kwh"]= np.where(derate_counts["is_derated"]=='Yes',derate_counts.LostEnergy_kwh,0)
# derate_counts["LostMarketPotential_usd"] = np.where(derate_counts["is_derated"]=='Yes',derate_counts.LostMarketPotential_u
```

Visually highlight detected derates

In [ ]:
```
derate_df.info()
```

In [ ]:

```
import matplotlib.pyplot as plt
import matplotlib.dates as md
from matplotlib.pyplot import figure
figure(figsize=(20, 6), dpi=80)

### The following code to create a dataframe and remove duplicated rows is always executed and acts as a preamble for your s
dataset = derate_df[derate_df['AssetName']== 'ORI-Ve-178']
dataset = dataset.sort_values('Timestamp')

dataset = dataset[['Timestamp', 'TM_ActivePower', 'TM_ExpectedPower', 'derated', 'normal','status']]
dataset = dataset.drop_duplicates()

dataset = dataset.rename(columns={'TM_ActivePower': 'Produced Power',
                                  'TM_ExpectedPower': 'Expected Power',
                                 'status': 'Turbine Status'})

dataset = dataset[dataset['Turbine Status'] == 'Active']
dataset['Timestamp'] = pd.to_datetime(dataset['Timestamp'])
dataset = dataset.set_index('Timestamp')

split_date1 = pd.to_datetime('2023-02-20 00:00:00')
split_date2 = pd.to_datetime('2023-03-20 00:00:00')
dataset = dataset.loc[(dataset.index >= split_date1) & (dataset.index <= split_date2)].copy()

a = min(dataset['Produced Power'].min(), dataset['Expected Power'].min())
b = max(dataset['Produced Power'].max(), dataset['Expected Power'].max())

plt.plot(dataset.index,
        dataset['Produced Power'],
        color="#1F77B4", label = 'Produced Power',
        alpha=0.8)
plt.plot(dataset.index,
        dataset['Expected Power'],
        color="#2CA02C", label = 'Expected Power',
        alpha=0.6)


### Highlight
plt.fill_between(dataset.index,
                 a - 50,
                 b + 50,
                where=(dataset['derated'] == 'Yes'),
                color='#FFAE49', alpha=0.3)

plt.fill_between(dataset.index,
                 a - 50,
                 b + 50,
                where=(dataset['normal'] == 'Yes'),
                color='#00BFBF', alpha=0.3)

plt.fill_between(dataset.index,
                 a - 50,
                 b + 50,
                where=(dataset['Turbine Status'] == 'Offline'),
                color='#44A5C2', alpha=0.3)

plt.fill_between(dataset.index,
                 a - 50,
                 b + 50,
                where=(dataset['Turbine Status'] == 'Site Curtailment'),
                color='#F4B811', alpha=0.3)
plt.xticks( rotation=25)

ax=plt.gca()

# Define the date format
date_form = md.DateFormatter("%m-%d")
ax.xaxis.set_major_formatter(date_form)
# Ensure a major tick for each day using (interval=1)
ax.xaxis.set_major_locator(md.DayLocator(interval=1))

# Set y-axis to always start at 0
ax.set_ylim(ymin=0)

plt.xlabel("Timestamp", fontsize=24)
plt.ylabel("Power (kW)", fontsize=24)
plt.title("Telemetry Data (Orange: Detected Anomaly Event)", fontsize=40)
plt.tick_params(labelsize=14, pad=6)
```

```
plt.legend(fontsize=20)
plt.tight_layout()
plt.show()
```

Export Notebook

In [ ]:
```
# !jupyter nbconvert --to qtpdf Wind_Derate_Detection_Vestas.ipynb
```

In [ ]:
```
!jupyter nbconvert --to html Wind_Turbine_Underperformance_Anomaly_Detection.ipynb
```

In [ ]:
```
!jupyter nbconvert --to webpdf --allow-chromium-download Wind_Turbine_Underperformance_Anomaly_Detection.ipynb
```