In [1]:
```
import numpy as np
import pandas as pd
from scipy.spatial import distance
import seaborn as sns
import matplotlib.pyplot as plt
import pathpy as pp
import sys
import plotly.express as px
sys.path.append('/Users/wastechs/Documents/git-repos/wake_effect/turbine_distances')
from interaction_matrix import interaction_matrix
```

In [2]:
```
coord = pd.read_csv('/Users/wastechs/Documents/git-repos/wake_effect/data/Pos_WTG_Brasil.c
```

Investigating Wake Effects of a Wind Farm Using Network Analysis

Authors: Gabriel Stechschulte and Régis Andréoli

Motivation and Research Question

- Wake Effects - A wind turbine can affect the power production of another wind turbine through decreased wind speeds
- Benefits of understanding wake effects:
  - Better wind farm design
  - Anomaly detection
- **R.Q.** Using a rule based calculation between any one turbine in a wind farm, wind speed, and wind angle of each turbine at time t, can an interaction network quantify a wake effect?

Brazilian Windfarm

- 32 on-shore wind turbines with a full-year of measurements from August 2013 to July 2014
- Features:
  - Wind velocity
  - Wind direction
  - Timestamp
  - Latitude and longitude of turbine(s)

In [4]:
```
px.set_mapbox_access_token(
    'pk.eyJ1IjoiZ3N0ZWNoc2NodWx0ZSIsImEiOiJja3g0d21vZDAyZnkwMndsYTM0ZjY0cHYwIn0.zEAHc1MIW
)
```

In [5]:
```
fig = px.scatter_mapbox(coord, lat="Lat.", lon="Long.",color="WTG",
                  color_continuous_scale=px.colors.cyclical.IceFire, size_max=20,zoom=12)
fig.show()
```

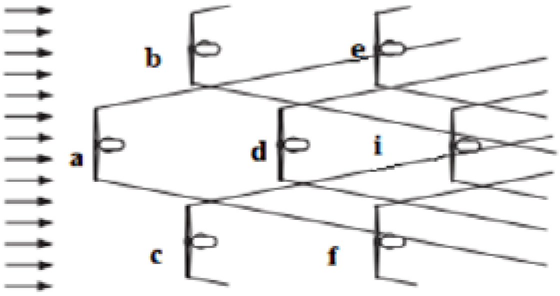
Turbine Interaction Matrix

`If turbine is within a distance of ## of another turbine, then .add_edge() else 0`

Distances:

- Euclidean: Distance between two points in space

- Based on Pythagoras and returns a distance with "no units"
- Haversine: Angular distance between two points on the surface of a sphere
  - Returns distance in 'km' or 'meters'



In [ ]:

```python
class interaction_matrix():

    def __init__(self, data):

        self.data = data

    def calculate_haver(self, tri, plot=bool):

        haversine = DistanceMetric.get_metric('haversine')

        self.data['lat_rad'] = np.radians(self.data['Lat.'])
        self.data['long_rad'] = np.radians(self.data['Long.'])

        # Calculate haversine distances (in meters)
        haversine_distances = haversine.pairwise(
            self.data[['lat_rad', 'long_rad']].to_numpy())*6373000

        # Convert to dataframe
        haversine_df = pd.DataFrame(
            haversine_distances,
            columns=self.data['WTG'].unique(),
            index=self.data['WTG'].unique()
        )

        # Subset by distances
        subset_proximity_haver = haversine_df[
            (haversine_df <= 720) & (haversine_df > 250)]
        subset_proximity_haver.fillna(value=0, inplace=True)
        subset_proximity_haver[subset_proximity_haver > 0] = 1
```

In [ ]:

```python
def calculate_euclidean(self, threshold, tri, plot=bool):

    lat = self.data['Lat.'].values
    long = self.data['Long.'].values

    self.datas = []

    for lat, long in zip(lat, long):
        self.datas.append((lat, long))

    # Calculate euclidean distance
    euclid_distances = distance.cdist(self.datas, self.datas, 'euclidean')

    # Convert to dataframe
    euclid_df = pd.DataFrame(
        euclid_distances,
```

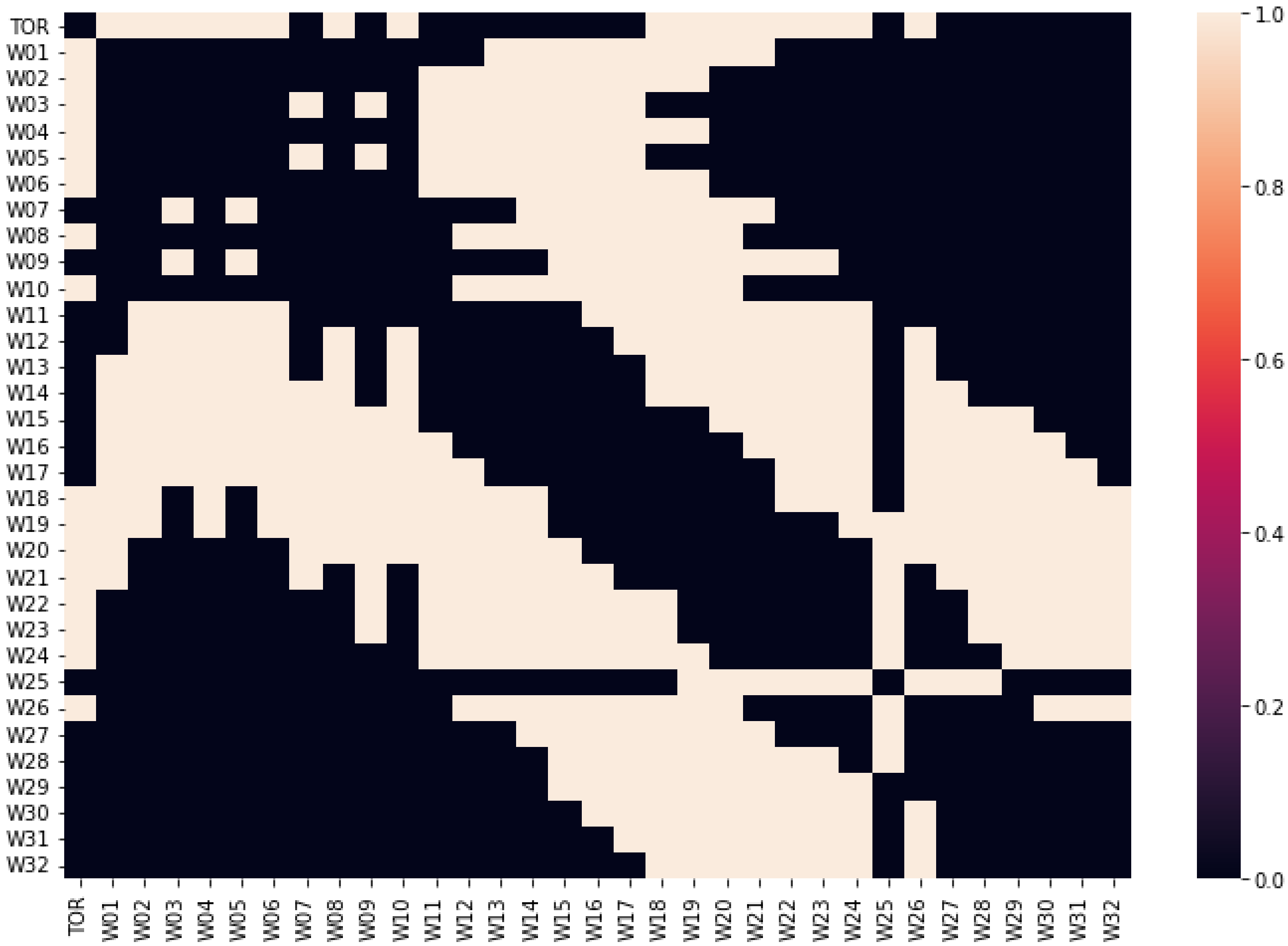
```
        columns=self.data['WTG'].unique(),
        index = self.data['WTG'].unique()
    )

    # Subset by distances
    subset_proximity_euclid = euclid_df[(euclid_df <= threshold) & (euclid_df > 0.0)]
    subset_proximity_euclid.fillna(value=0, inplace=True)
    subset_proximity_euclid[subset_proximity_euclid > 0] = 1
```

In [6]:

```
matrix = interaction_matrix(coord)
```

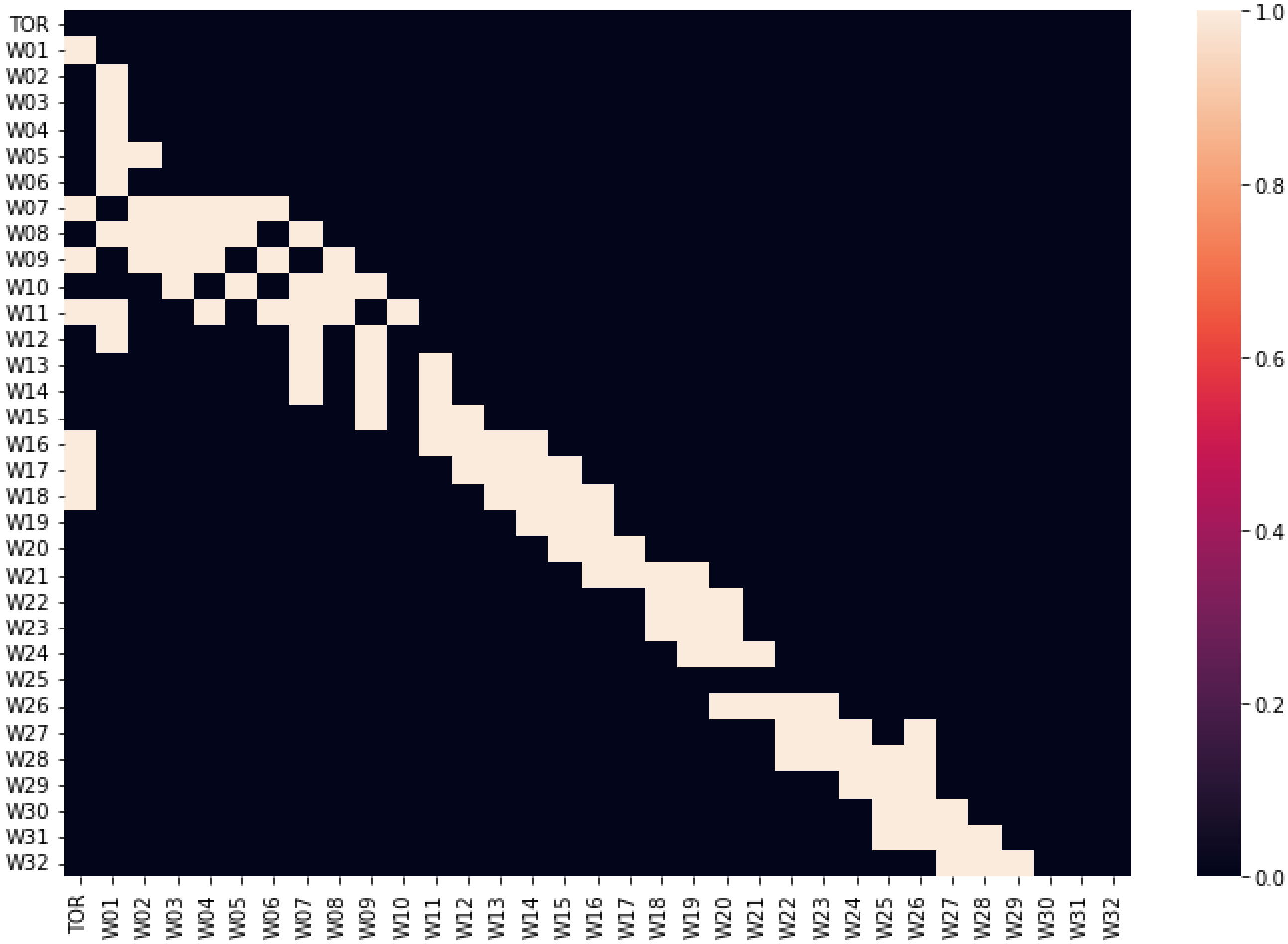
In [7]:

```
haver_interaction_full = matrix.calculate_haver(tri=False, plot=True)
```



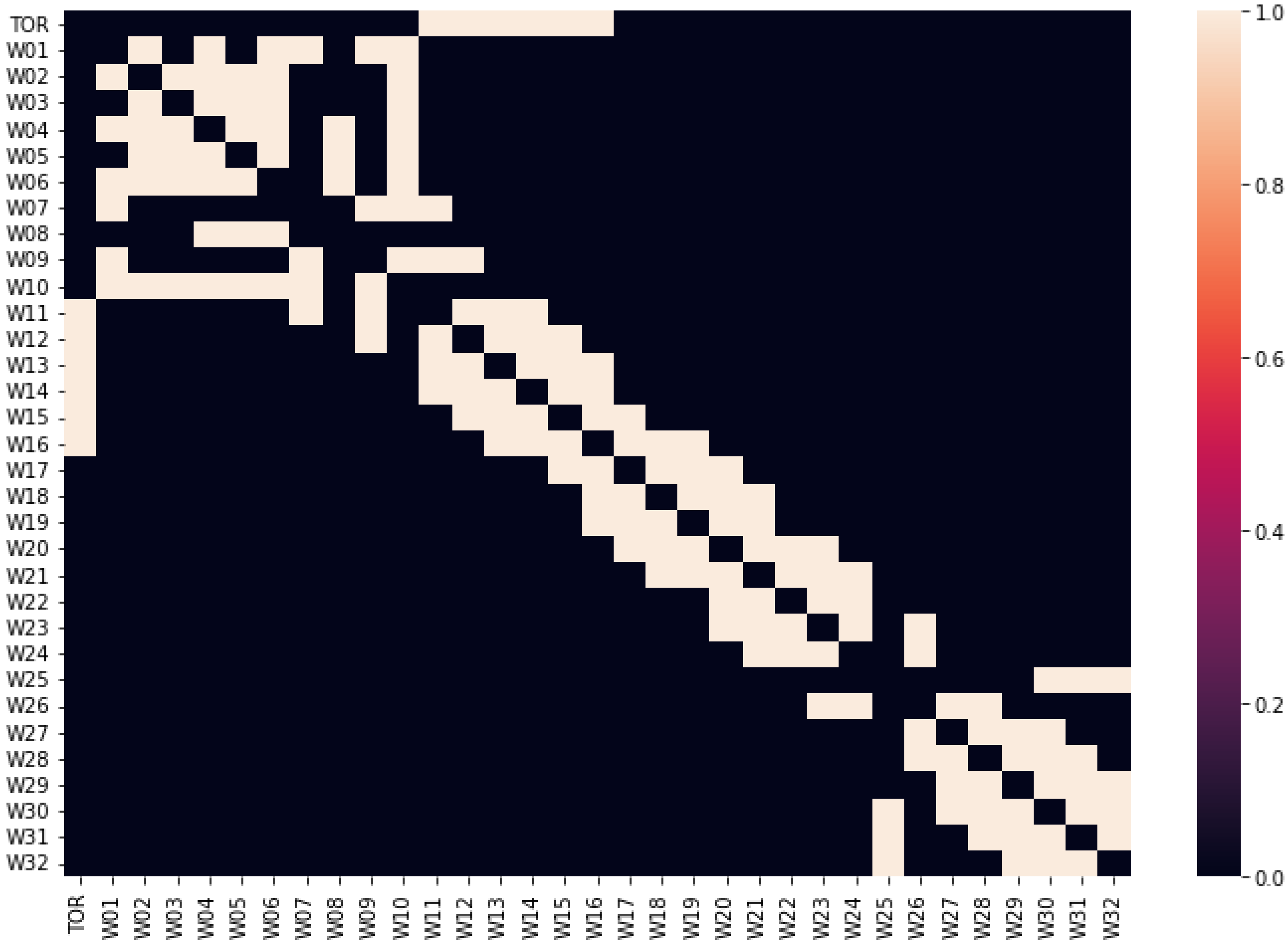
In [8]:

```
haver_interaction_tri = matrix.calculate_haver(True, True)
```



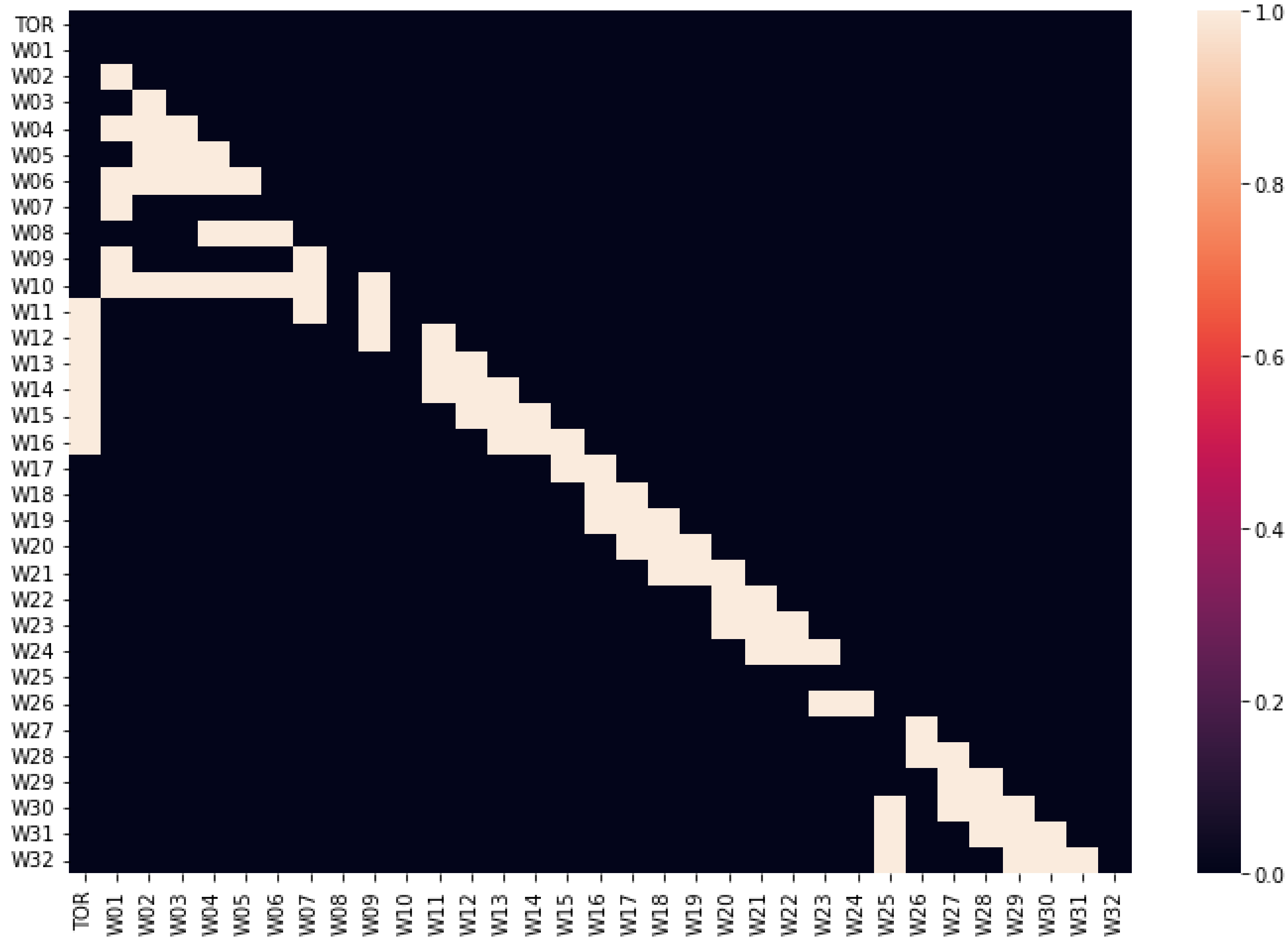
In [9]:

```
euclid_interaction_full_v1 = matrix.calculate_euclidean(threshold=0.004, tri=False, plot=T
euclid_interaction_full_v2 = matrix.calculate_euclidean(threshold=0.0035, tri=False, plot=
euclid_interaction_full_v3 = matrix.calculate_euclidean(threshold=0.003, tri=False, plot=F
euclid_interaction_full_v4 = matrix.calculate_euclidean(threshold=0.0025, tri=False, plot=
```



In [10]:

```
euclid_interaction_tri_v1 = matrix.calculate_euclidean(threshold=0.004, tri=True, plot=Tru
euclid_interaction_tri_v2 = matrix.calculate_euclidean(threshold=0.0035, tri=True, plot=Fa
euclid_interaction_tri_v3 = matrix.calculate_euclidean(threshold=0.003, tri=True, plot=Fal
euclid_interaction_tri_v4 = matrix.calculate_euclidean(threshold=0.0025, tri=True, plot=Fa
```



Building the Network

Library: `pathpy`

Rules based off of literature:

```
If turbine is within a distance of ## of another turbine, then .add_edge() else 0
```

In [11]:

```
def build_network(interaction, directed=bool):

    graph = pp.Network(directed=directed)

    cnt = 0
    for row in range(0, 33):
        for turb1 in interaction.iloc[[row]].index:
            for int, adj_turb in zip(interaction.values[row], interaction.columns):
                cnt += 1
                if int == 1:
                    graph.add_edge(turb1, adj_turb)
                else:
                    pass

    return graph
```

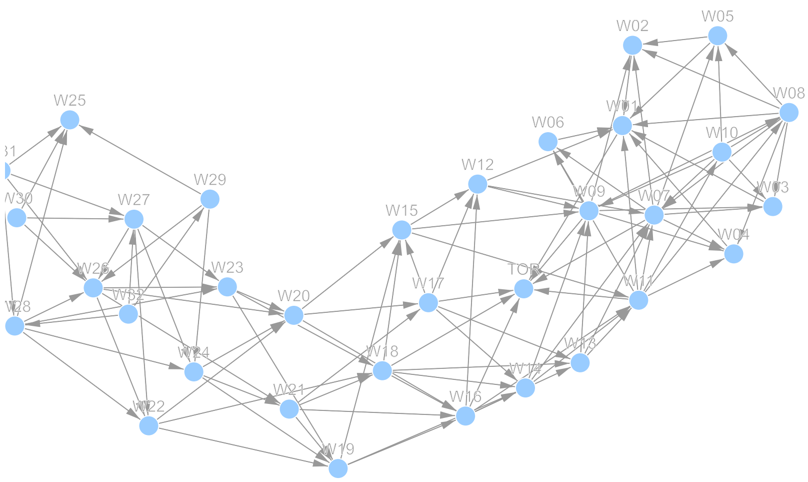
In [12]:

```
haver_full = build_network(haver_interaction_full, True)
haver_tri = build_network(haver_interaction_tri, True)
```

In [13]:

```
haver_tri
```

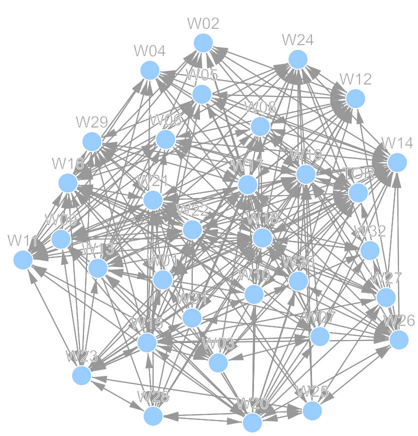
Out[13]: [save svg]



In [14]:

```
haver_full
```

Out[14]: [save svg]



In [15]:

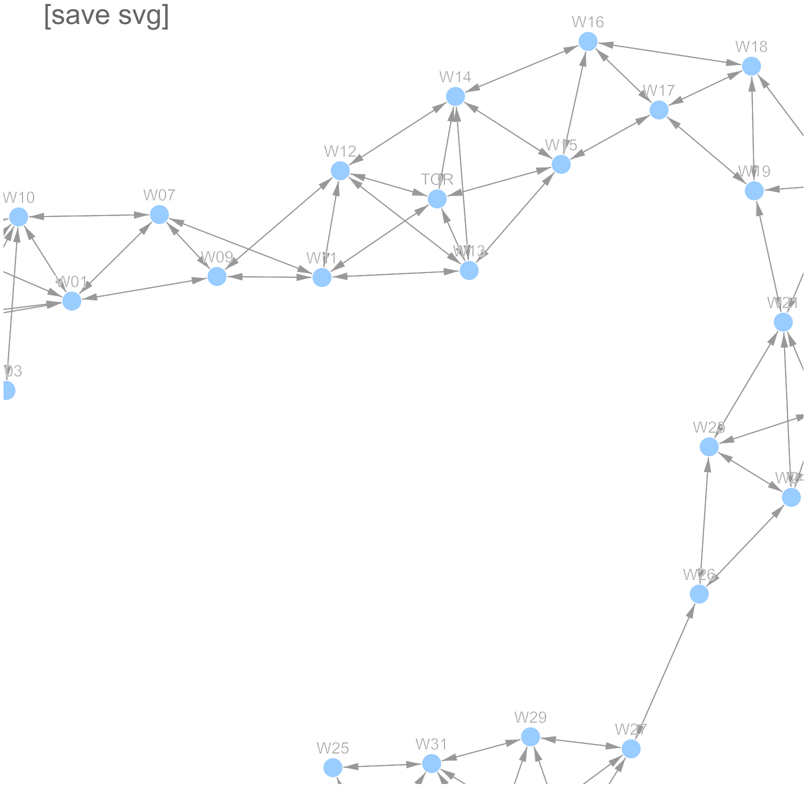
```
euclid_full_v1 = build_network(euclid_interaction_full_v1, True)
euclid_tri_v1 = build_network(euclid_interaction_tri_v1, True)

euclid_full_v2 = build_network(euclid_interaction_full_v2, True)
euclid_tri_v2 = build_network(euclid_interaction_tri_v2, True)

euclid_full_v3 = build_network(euclid_interaction_full_v3, True)
euclid_tri_v3 = build_network(euclid_interaction_tri_v3, True)
```

```
euclid_full_v4 = build_network(euclid_interaction_full_v4, True)
euclid_tri_v4 = build_network(euclid_interaction_tri_v4, True)
```

In [16]:

```
euclid_full_v2
```

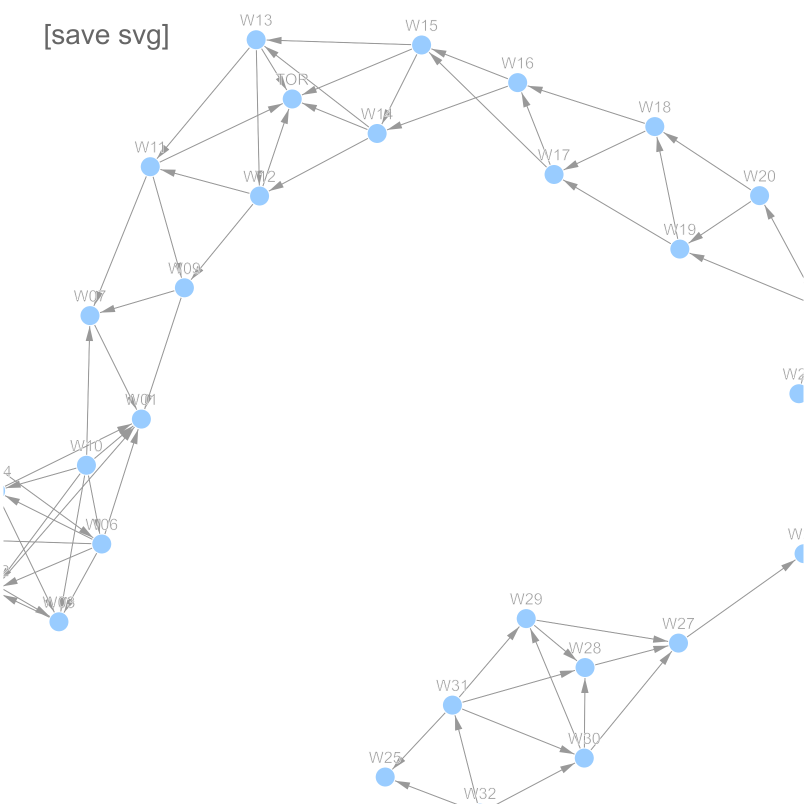
Out[16]: [save svg]



In [17]:

```
euclid_tri_v2
```

Out[17]: [save svg]



Network Centralities

- Are used as a **proxy** for quantifying a wake effect
- "What turbines are most central?, i.e., how many edges does a turbine have?"
- "How close is a turbine to all other turbines in the network?"

Degree

- Degree centrality: How many edges does a node have?
  - A node is central if it has a high degree
  - Out degree, in-degree, or the sum of both

Closeness

- Closeness centrality: Indicates how close a node is to all other nodes in the network
  - Calculated as the average of the shortest path length from the node to every other node in the network

Betweenness

- Betweenness centrality: How many shortest paths go through a certain node?

In [18]:

```
def centrality_measures(graph):
    # Calculates betweeness centrality of all nodes
    betweenness = pp.algorithms.centralities.betweenness(graph)
    # Calculates degree centrality of all nodes
    degree = pp.algorithms.centralities.degree(graph)
    # Calculates closeness centrality of all nodes
    closeness = pp.algorithms.centralities.closeness(graph)

    return betweenness, degree, closeness
```

Local Clustering Coefficient

- Quantifies how close its neighbors are to being a "clique"
- That is, it measures the fraction of its various pairs of neighbors are neighbors with each other

In [19]:

```
def clustering_coefficient(graph, name):

    cluster_coef = {}

    for turb in coord['WTG']:
        coef = pp.statistics.local_clustering_coefficient(graph, turb)
        cluster_coef[turb] = coef

    colors = [
        'grey' if (x < np.quantile(list(cluster_coef.values()), 0.75))
        else 'red' for x in list(cluster_coef.values())
    ]

    plt.figure(figsize=(14, 7))
    plt.title(name)
    sns.barplot(x=list(cluster_coef.keys()), y=list(cluster_coef.values()), palette=colors
    plt.xticks(rotation=45)
    plt.xlabel('Turbine')
    plt.ylabel('Count')
    plt.show()
```

In [20]:

```
def plotter(measure, name):

    df = pd.DataFrame.from_dict(
        measure, orient='index', columns=['measure'])

    colors = ['grey' if (x < np.quantile(df.measure.values, 0.75)) else 'red' for x in df.

    sns.set_theme(style='whitegrid')
    plt.figure(figsize=(14, 7))
    plt.title(name)
    sns.barplot(x=df.index, y=df.measure, palette=colors)
    plt.xticks(rotation=45)
    plt.xlabel('Turbine')
    plt.ylabel('Count')
    plt.show()
```

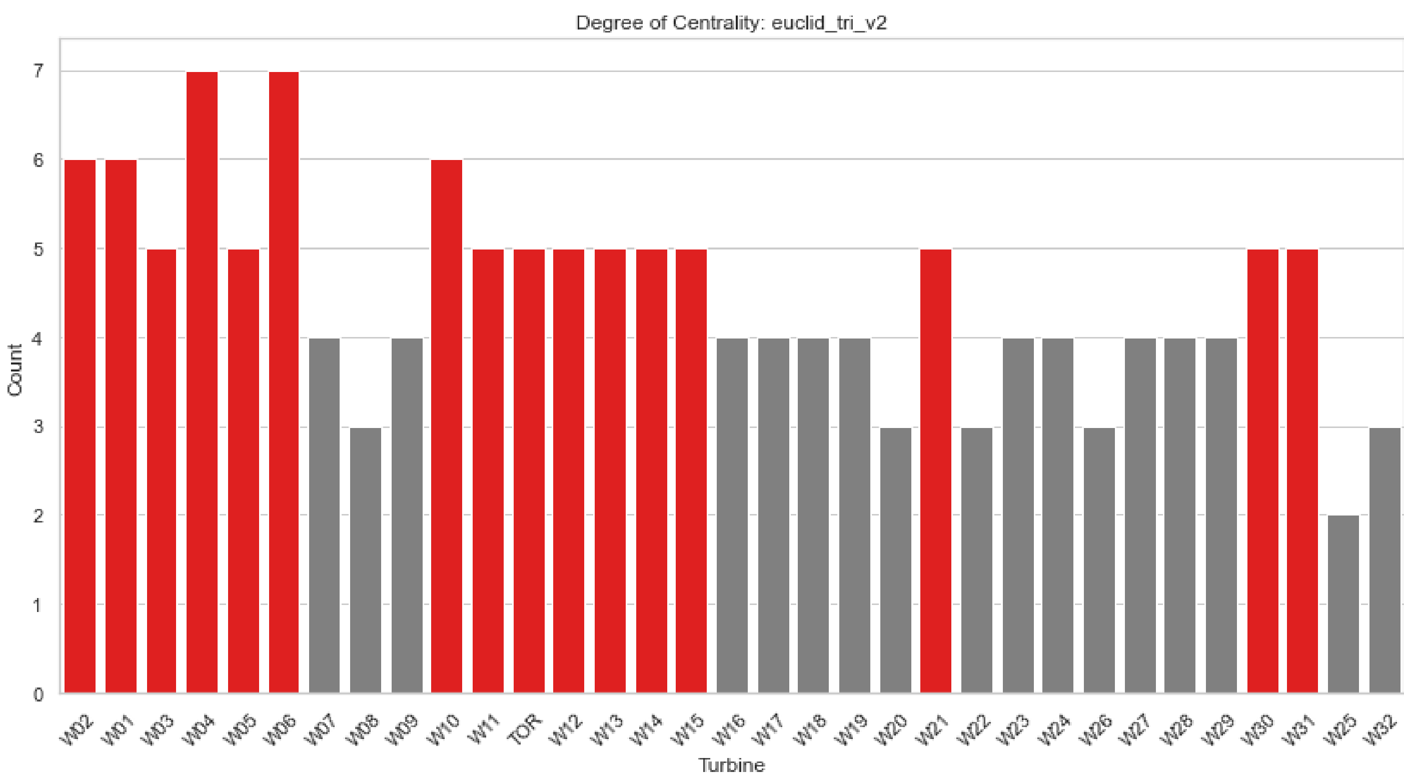
In [21]:

```
# Euclidean Distances
betweenness, degree, closeness = centrality_measures(euclid_tri_v2)
```

```
2022-01-17 21:39:37 [Severity.INFO]     Calculating betweenness centralities ...
2022-01-17 21:39:37 [Severity.INFO]     Calculating closeness in network ...
2022-01-17 21:39:37 [Severity.INFO]     finished.
```

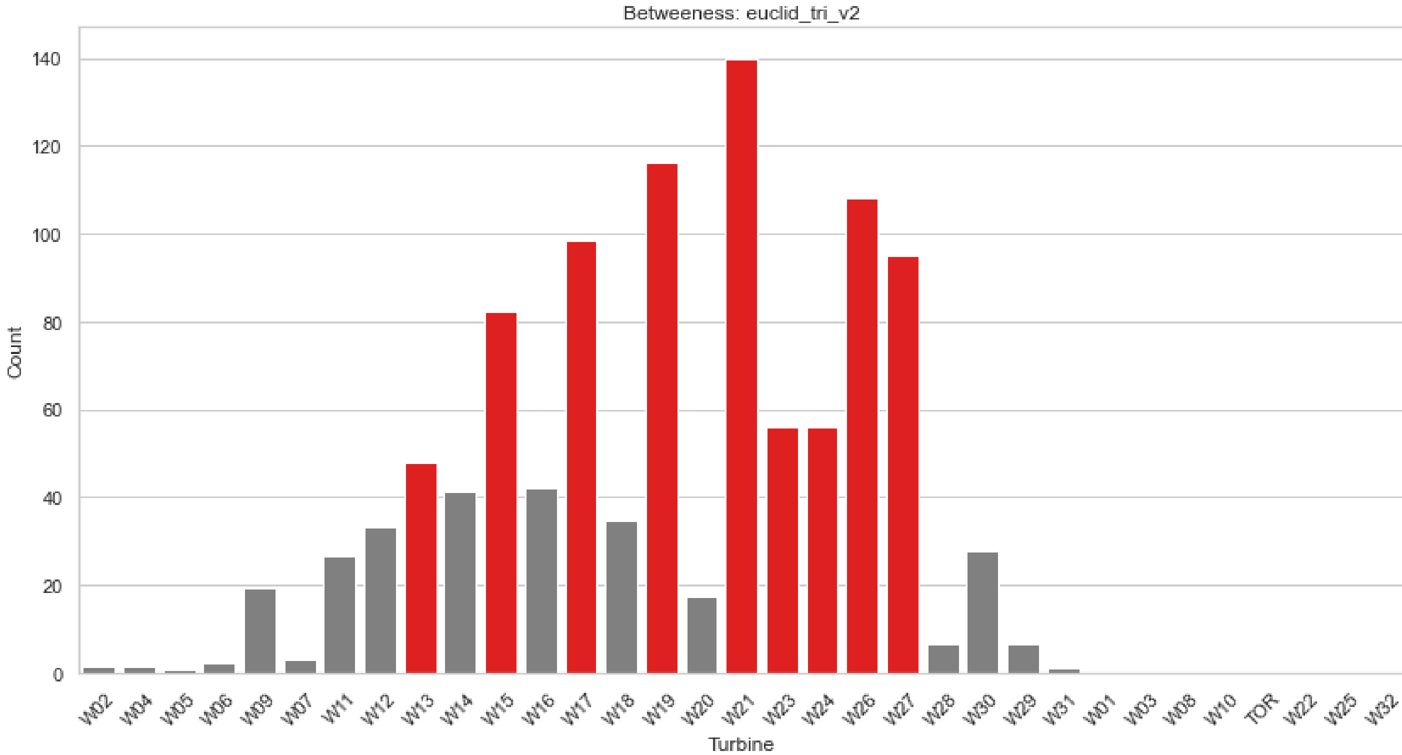
In [22]:

```
plotter(degree, 'Degree of Centrality: euclid_tri_v2')
```

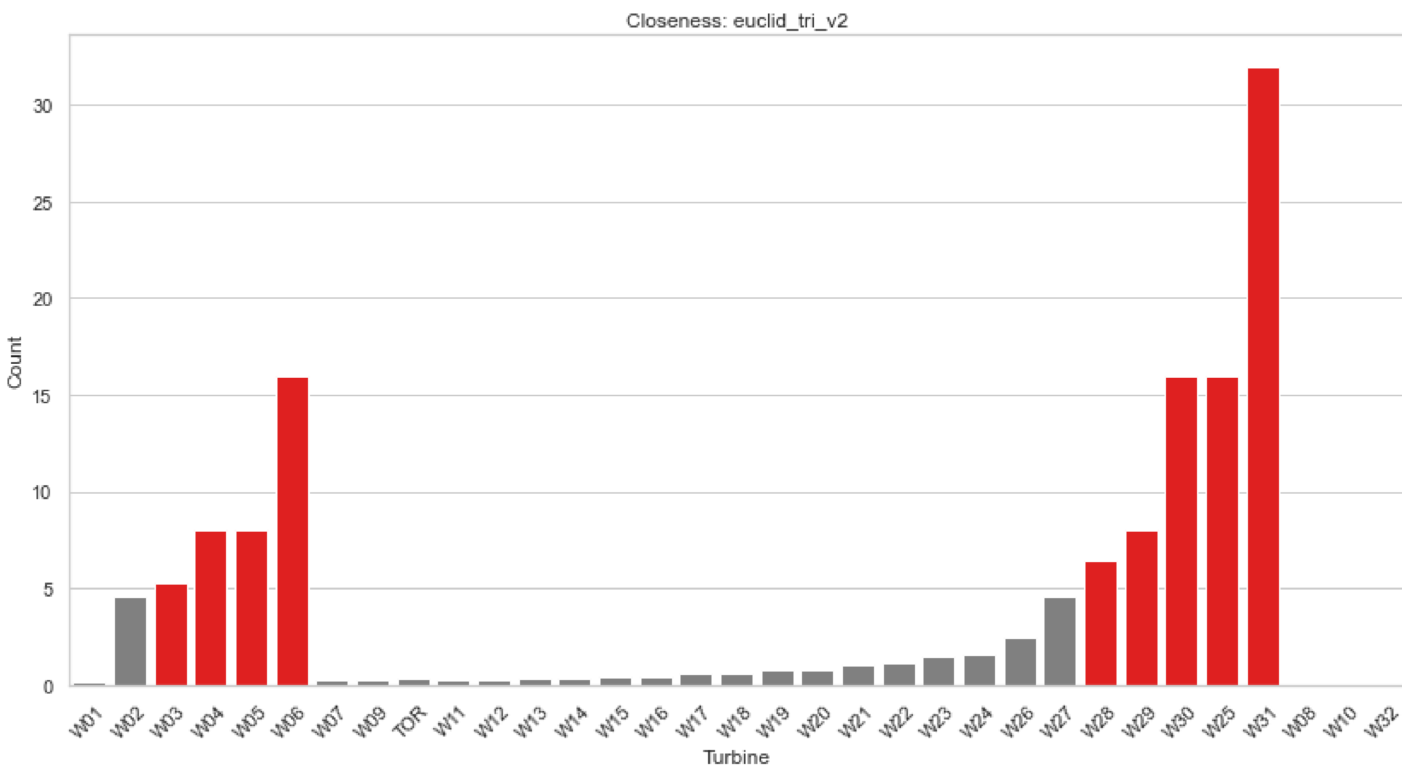


In [23]:

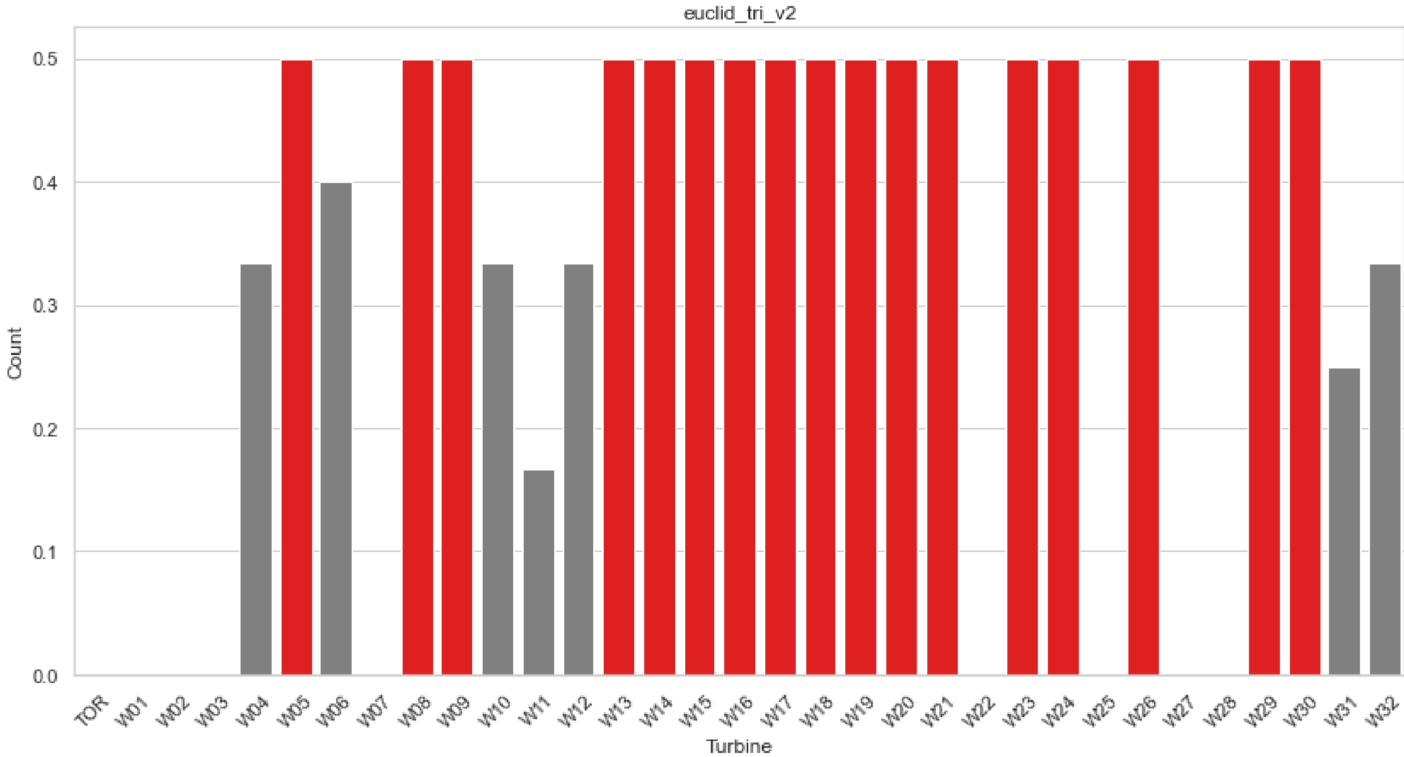
```
plotter(betweenness, 'Betweeness: euclid_tri_v2')
```



In [24]:
```
plotter(closeness, 'Closeness: euclid_tri_v2')
```



In [25]:
```
clustering_coefficient(euclid_tri_v2, 'euclid_tri_v2')
```



In [26]:

```
# Calculates the mean (in/out)-degree of a directed or undirected network
print(f"Avg degree: {pp.statistics.mean_degree(euclid_tri_v2, degree='indegree')}")
print(f"Avg degree: {pp.statistics.mean_degree(euclid_tri_v2, degree='outdegree')}")
```

```
Avg degree: 2.242424242424242
Avg degree: 2.242424242424242
```

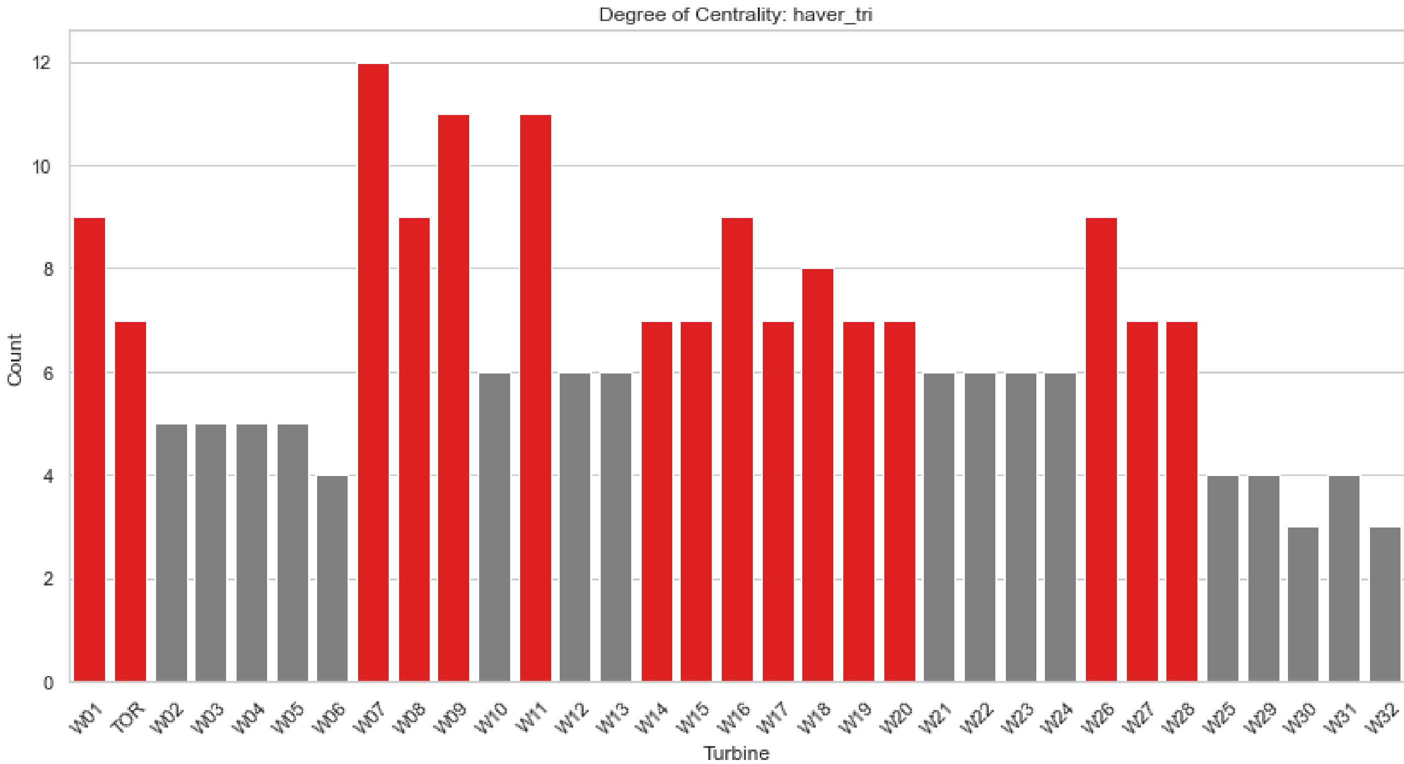
In [27]:

```
# Haversine Distances
betweenness_h, degree_h, closeness_h = centrality_measures(haver_tri)
```

```
2022-01-17 21:39:45 [Severity.INFO]     Calculating betweenness centralities ...
2022-01-17 21:39:45 [Severity.INFO]     Calculating closeness in network ...
2022-01-17 21:39:45 [Severity.INFO]     finished.
```

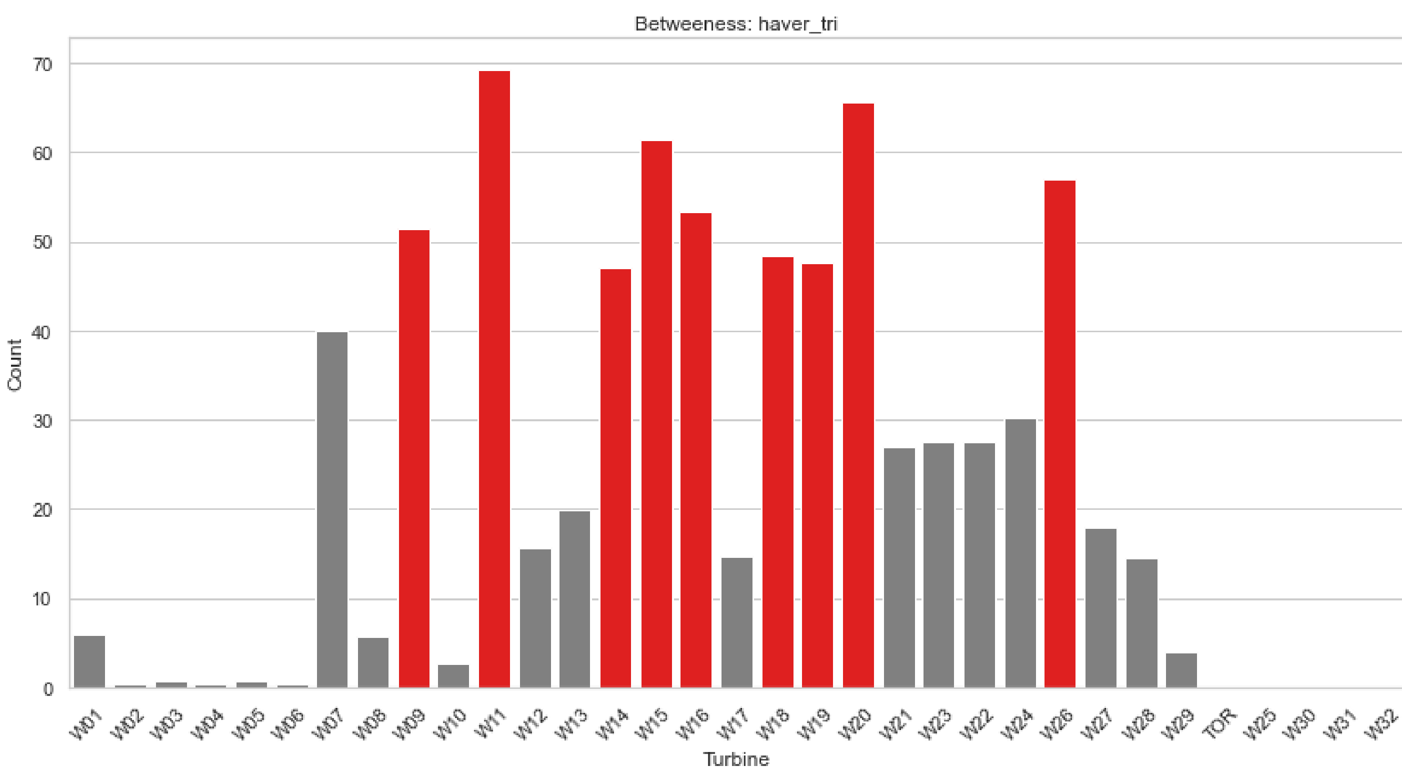
In [28]:

```
plotter(degree_h, 'Degree of Centrality: haver_tri')
```



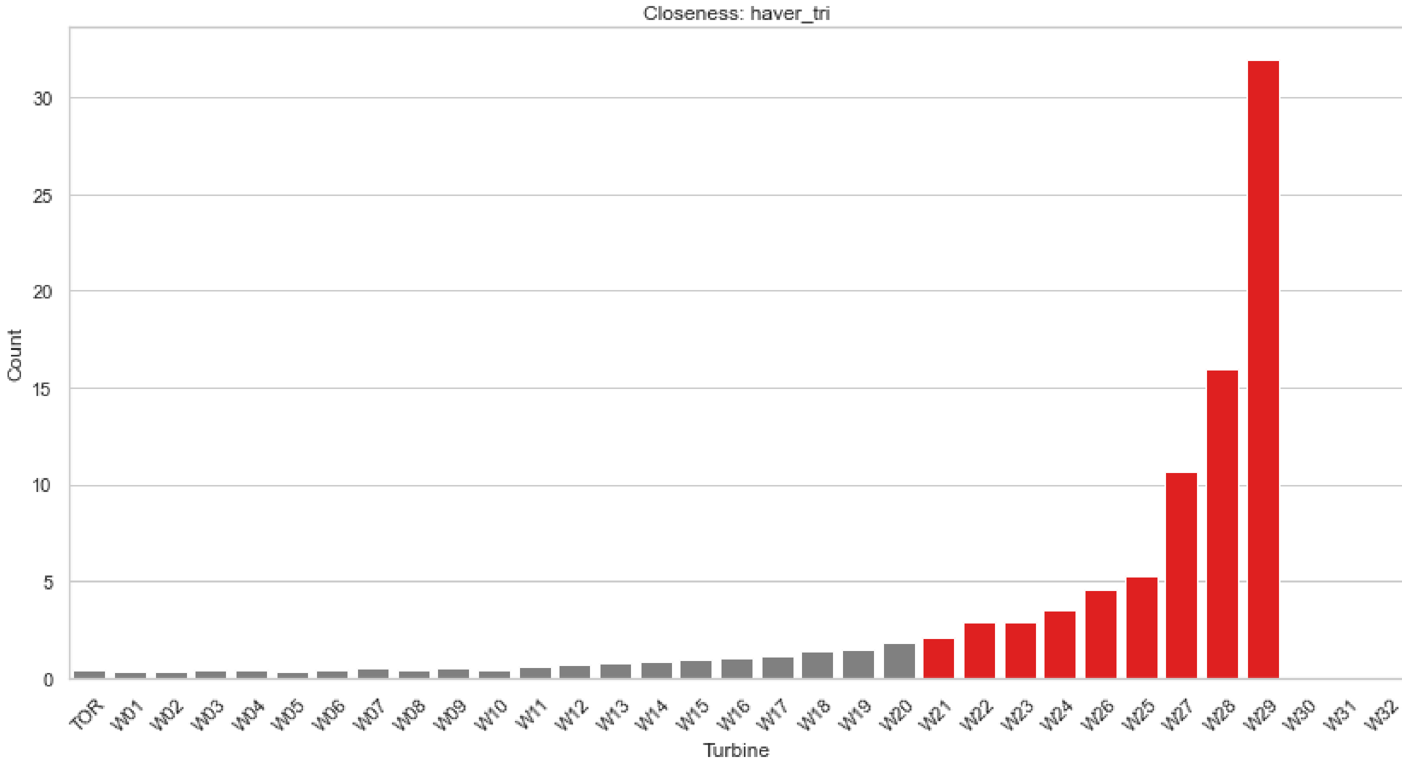
In [29]:

```
plotter(betweenness_h, 'Betweeness: haver_tri')
```



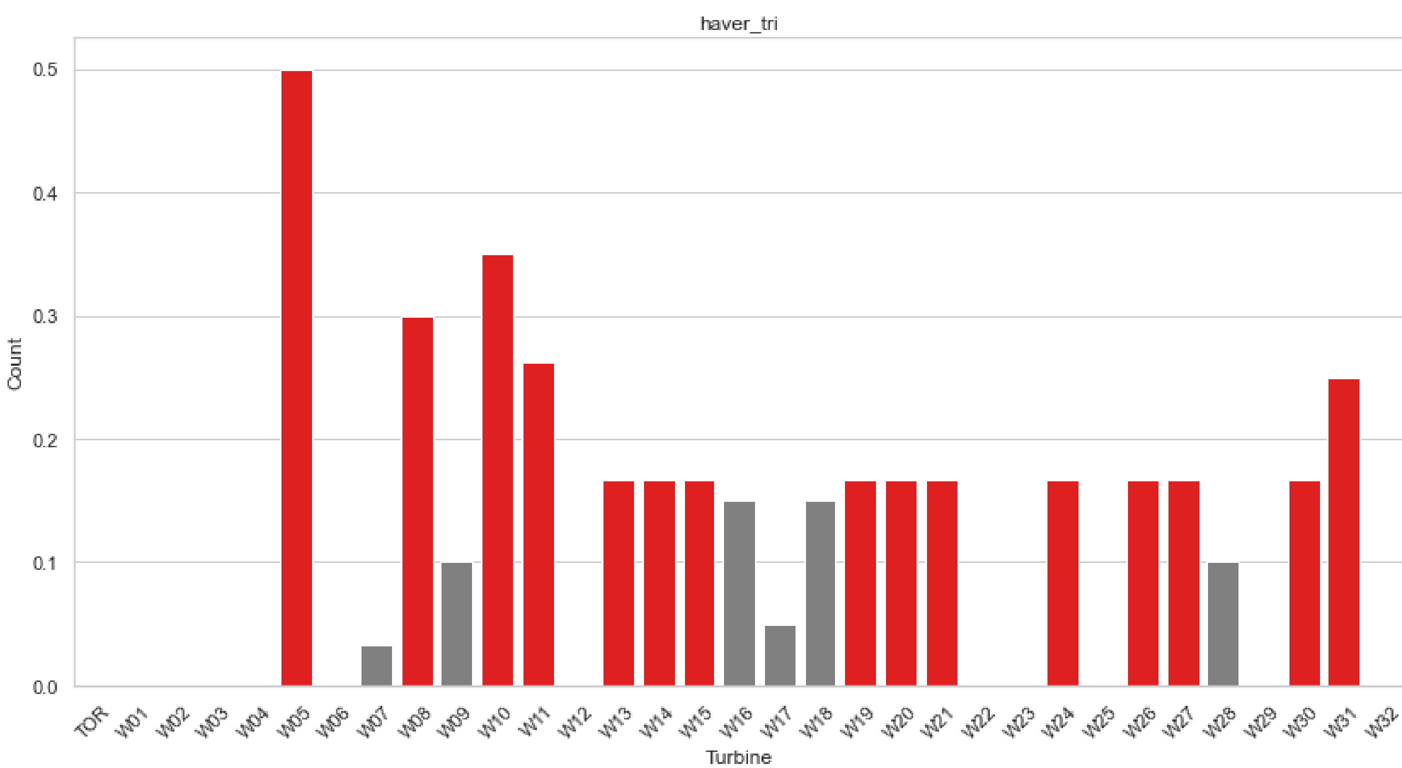
In [30]:

```
plotter(closeness_h, 'Closeness: haver_tri')
```



In [31]:

```
clustering_coefficient(haver_tri, 'haver_tri')
```



In [32]:

```
# Calculates the mean (in/out)-degree of a directed or undirected network
print(f"Avg degree: {pp.statistics.mean_degree(haver_tri, degree='indegree')}")
print(f"Avg degree: {pp.statistics.mean_degree(haver_tri, degree='outdegree')}")
```

```
Avg degree: 3.303030303030303
Avg degree: 3.303030303030303
```

Data Preparation - Wind turbine features

This first part covers a qualitative analysis of the dataset covering:

- The distribution of measurements per wind angle
- The wind speed measurements for selected turbines

In [33]:
```
# load data and merge data frames

wind_dir = pd.read_csv('/Users/wastechs/Documents/git-repos/wake_effect/data/Wind_directic
wind_v = pd.read_csv('/Users/wastechs/Documents/git-repos/wake_effect/data/Windfarm.csv')
df = wind_dir.merge(wind_v, left_on='Unnamed: 0', right_on='Unnamed: 0')
```

In [11]:
```
# load data and merge data frames
wind_dir = pd.read_csv('C:/Users/regis/OneDrive/Dokumente/GitHub/wake_effect/data/Wind_dir
wind_v = pd.read_csv('C:/Users/regis/OneDrive/Dokumente/GitHub/wake_effect/data/Windfarm.c
df = wind_dir.merge(wind_v, left_on='Unnamed: 0', right_on='Unnamed: 0')
```

In [34]:
```
# Wind Direction Analysis
# Prepare the data

df1 = df.drop(columns = ['10.0'])
df1 = df1.drop(columns = ['40.0'])
df1 = df1.drop(columns = ['60.0'])
df1['wind_dir'] = ((df1['80.0'] + df1['100.0'])/2).round(0)
df1['80.0'] = df1['80.0'].round(0)
df1['100.0'] = df1['100.0'].round(0)
df1 = df1.rename(columns={'80.0': 'eighty', '100.0': 'hundred'})
df1.head(4)
```

Out[34]:

	Unnamed: 0	eighty	hundred	WTG1	WTG2	WTG3	WTG4	WTG5	WTG6	WTG7	...	WTG24	WTG25	W
0	2013-08-01 00:00:00	111.0	113.0	11.3	12.4	10.4	9.8	9.9	10.5	11.4	...	12.5	12.2	
1	2013-08-01 00:10:00	113.0	116.0	11.6	11.9	10.4	10.7	10.6	10.7	12.0	...	13.0	12.5	
2	2013-08-01 00:20:00	110.0	112.0	11.8	12.5	10.8	10.6	10.4	10.5	11.9	...	12.1	11.7	
3	2013-08-01 00:30:00	113.0	116.0	11.7	11.6	10.6	10.4	10.1	10.3	11.7	...	13.2	10.9	

4 rows × 36 columns

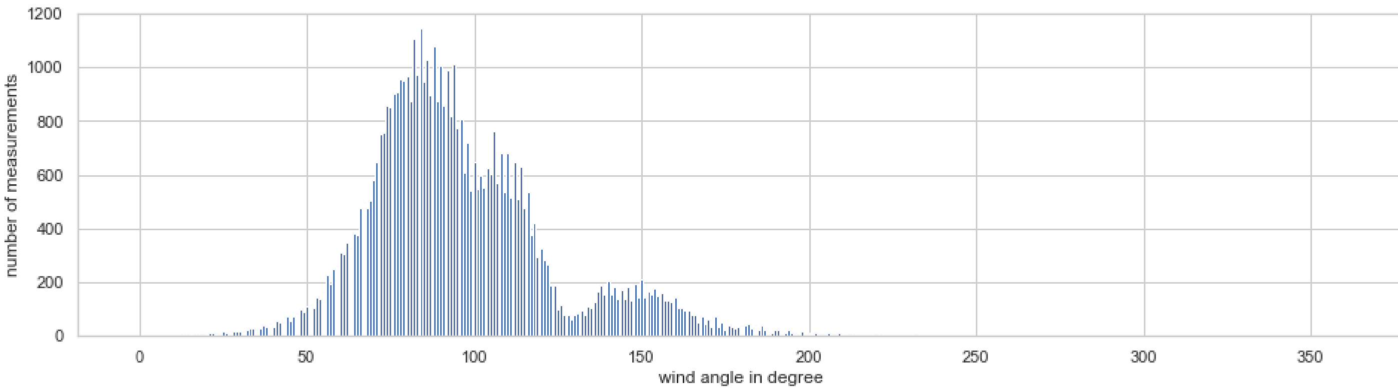
Bar Plot of the Distribution of Measurement Entries Per Angle for Height of 80m, 100m and Mean

In [35]:
```
# count amount of entries per angle for 80m, 100m and mean

df_eighty = pd.DataFrame()
df_eighty['eighty'] = df1['eighty']
df_eighty['count_rows'] = df1.groupby(['eighty'])['eighty'].transform('count')
df_eighty = df_eighty.drop_duplicates(subset=None, keep='first', inplace=False, ignore_inc

df_hundred = pd.DataFrame()
df_hundred['hundred'] = df1['hundred']
df_hundred['count_rows'] = df1.groupby(['hundred'])['hundred'].transform('count')
```

```python
df_hundred = df_hundred.drop_duplicates(subset=None, keep='first', inplace=False, ignore_i

df_w = pd.DataFrame()
df_w['wind_dir'] = df1['wind_dir']
df_w['count_rows'] = df1.groupby(['wind_dir'])['wind_dir'].transform('count')
df_w = df_w.drop_duplicates(subset=None, keep='first', inplace=False, ignore_index=False)

wind_dir_80 = df_eighty['eighty']
amount_80 = df_eighty['count_rows']

wind_dir_100 = df_hundred['hundred']
amount_100 = df_hundred['count_rows']

wind_dir_mean = df_w['wind_dir']
amount_mean = df_w['count_rows']
```

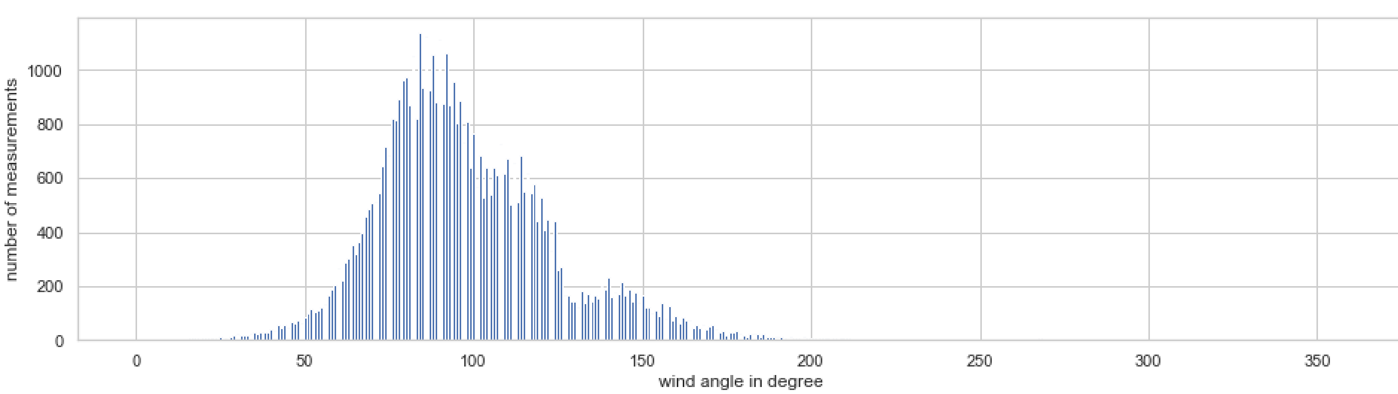
In [58]:

```python
# Creates bar plot of the distribution of entries per angle at 80m, 100m and mean

fig = plt.figure()
plt.bar(wind_dir_80, amount_80)
fig.set_size_inches(16, 4)
plt.ylabel("number of measurements")
plt.xlabel("wind angle in degree")
plt.show()
```



In [59]:

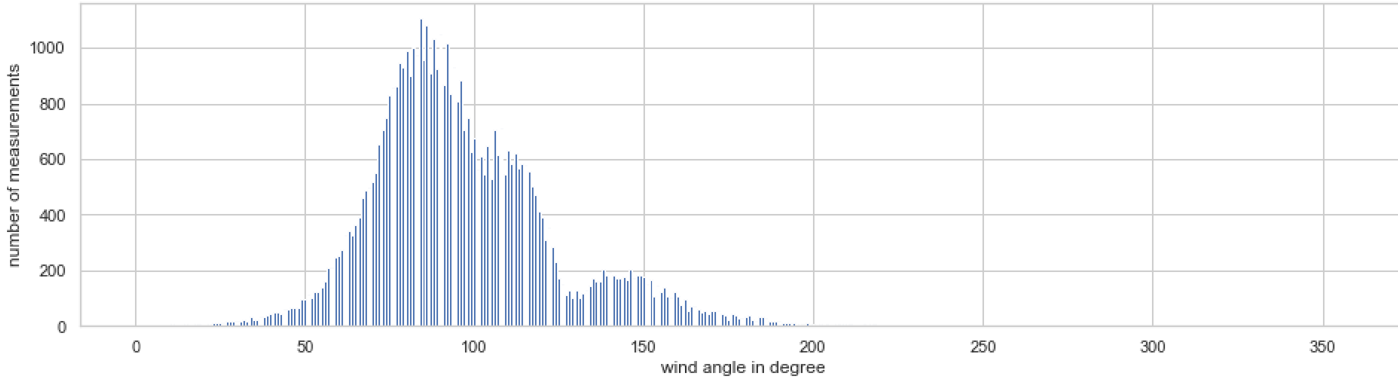
```python
fig = plt.figure()
plt.bar(wind_dir_100, amount_100)
fig.set_size_inches(16, 4)
plt.ylabel("number of measurements")
plt.xlabel("wind angle in degree")
plt.show()
```



In [60]:

```python
fig = plt.figure()
plt.bar(wind_dir_mean, amount_mean)
fig.set_size_inches(16, 4)
```

```
plt.ylabel("number of measurements")
plt.xlabel("wind angle in degree")
plt.show()
```



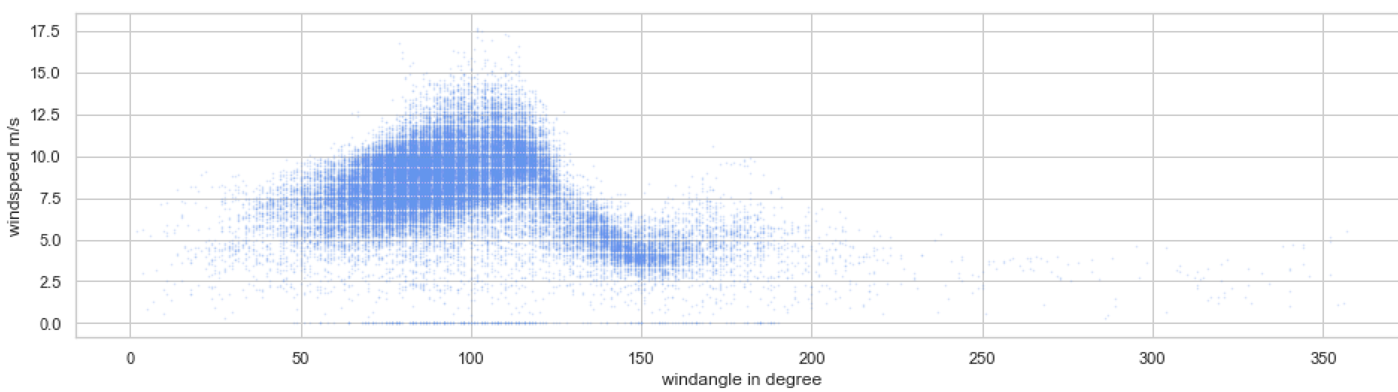
Short Visual Analysis of the Windspeed Records Per Angle for Selected Turbines

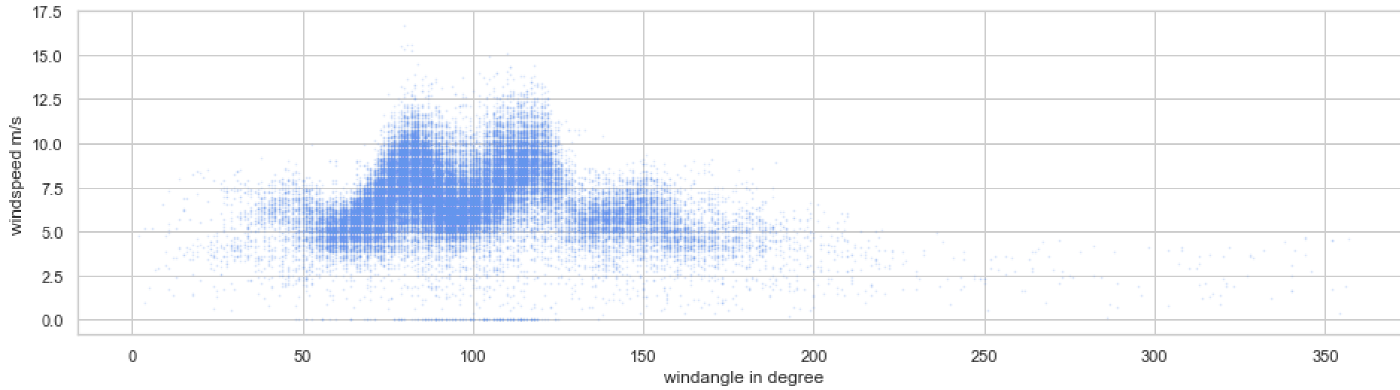
In [39]:

```
# Wind Speed Analysis
# Input: turbine nr
# Output: plot of speed per angle

def wind_speed(turbine):
    t = "WTG" + str(turbine)
    wind_dir = df1['wind_dir']
    amount = df1[t]
    fig = plt.figure()
    plt.scatter(wind_dir, amount, color='cornflowerblue', s=0.5, alpha=0.175)
    plt.ylabel("windspeed m/s")
    plt.xlabel("windangle in degree")
    fig.set_size_inches(16, 4)
    plt.show()
```

In [40]:

```
wind_speed(30)
wind_speed(5)
```





Data Manipulation - Wind Angles and Weighted Matrix

This first part covers calculation and dataframe preparation for the network analysis.

```
- function to get absolute and relative wind speed-difference of two turbines at given angle
- function to create and plot windspeed-difference of two turbiens for all angle
- Function to create a windspeed-difference confusion matrix of all turbines at given angle
```

In [41]:

```
# Prepare the data
df2 = df1.groupby(['wind_dir']).mean().round(2)
df2 = df2.drop(columns = ['eighty'])
df2 = df2.drop(columns = ['hundred'])
df2.head(5)
```

Out[41]:

	WTG1	WTG2	WTG3	WTG4	WTG5	WTG6	WTG7	WTG8	WTG9	WTG10	...	WTG23	WTG24
wind_dir													
2.0	4.90	5.0	5.20	4.6	4.8	4.3	4.90	4.20	5.10	4.9	...	5.0	3.7
4.0	3.25	3.1	3.15	2.7	3.1	2.8	3.45	2.45	3.45	3.2	...	3.7	3.3
5.0	1.40	1.7	1.00	1.1	1.8	1.1	1.00	1.40	1.20	1.3	...	0.5	0.8
6.0	5.20	5.5	5.70	5.3	5.2	4.9	5.30	4.40	5.60	5.4	...	4.8	3.9
7.0	3.00	2.9	2.90	2.4	2.8	2.1	2.60	2.10	3.00	2.8	...	2.1	2.3

5 rows × 32 columns

In [42]:

```
# Function: get absolute or relative wind speed difference
# Input: turbine nr, turbine nr, angle, relative change: 1 -> yes
# Output: [wind diff]

def wind_dif(turbine1, turbine2, angle, relative=0):
    if relative == 0:
        t1 = "WTG" + str(turbine1)
        t2 = "WTG" + str(turbine2)
        try:
            wind_dif = (df2[t1].loc[[angle]]) - (df2[t2].loc[[angle]])
            wind_dif = wind_dif.iloc[0].round(1)
            return wind_dif
        except:
            print("NaN")
```

```
    else:
        t1 = "WTG" + str(turbine1)
        t2 = "WTG" + str(turbine2)
        try:
            wind_dif = (df2[t1].loc[[angle]]) - (df2[t2].loc[[angle]])
            t1 = df2[t1].loc[[angle]]
            t1 = t1.iloc[0].round(1)
            wind_dif = wind_dif.iloc[0].round(1)
            wind_dif = wind_dif / (wind_dif + t1)
            return wind_dif
        except:
            print("NaN")
```

In [44]:

```
wind_dif(5, 4, 95)
```

Out[44]:

```
-1.4
```

In [45]:

```
wind_dif(5, 4, 95, 1)
```

Out[45]:

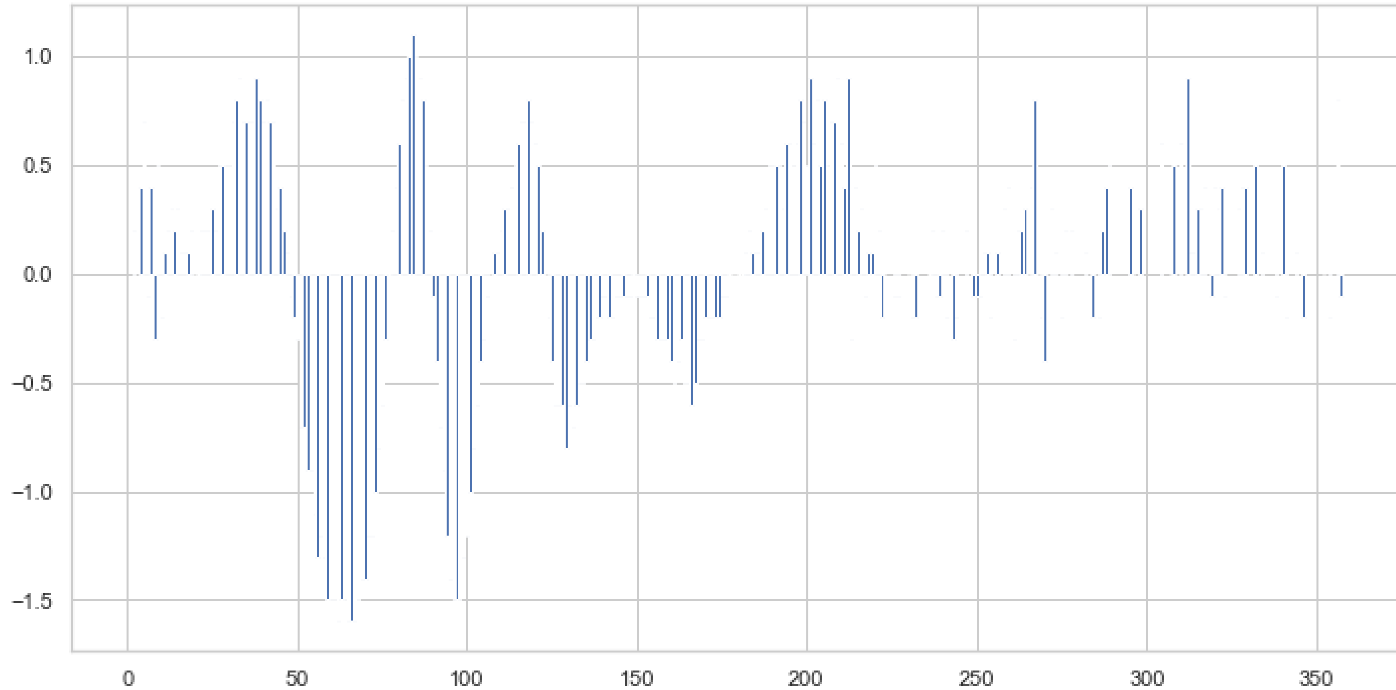
```
-0.27450980392156865
```

In [46]:

```
# Function create plot of absolute wind speed diff over all angle
# Input: Nr. of turbine, Nr. of turbine, relative change: 1 -> yes
# Output: plot

def wd_plot(turbine1, turbine2, relative=0):
    t1 = "WTG" + str(turbine1)
    t2 = "WTG" + str(turbine2)
    c = []
    count = 0
    for i in df2.iterrows():
        index = int(df2.iloc[[count]].index.values)
        number = wind_dif(turbine1, turbine2, index, relative)
        c = c + [[index] + [number]]
        count = count + 1
    c = pd.DataFrame(c, columns=['wind_dir', 'wind_diff'])
    wind_dir = c['wind_dir']
    amount = c['wind_diff']
    fig = plt.figure()
    plt.bar(wind_dir, amount)
    fig.set_size_inches(12, 6)
```

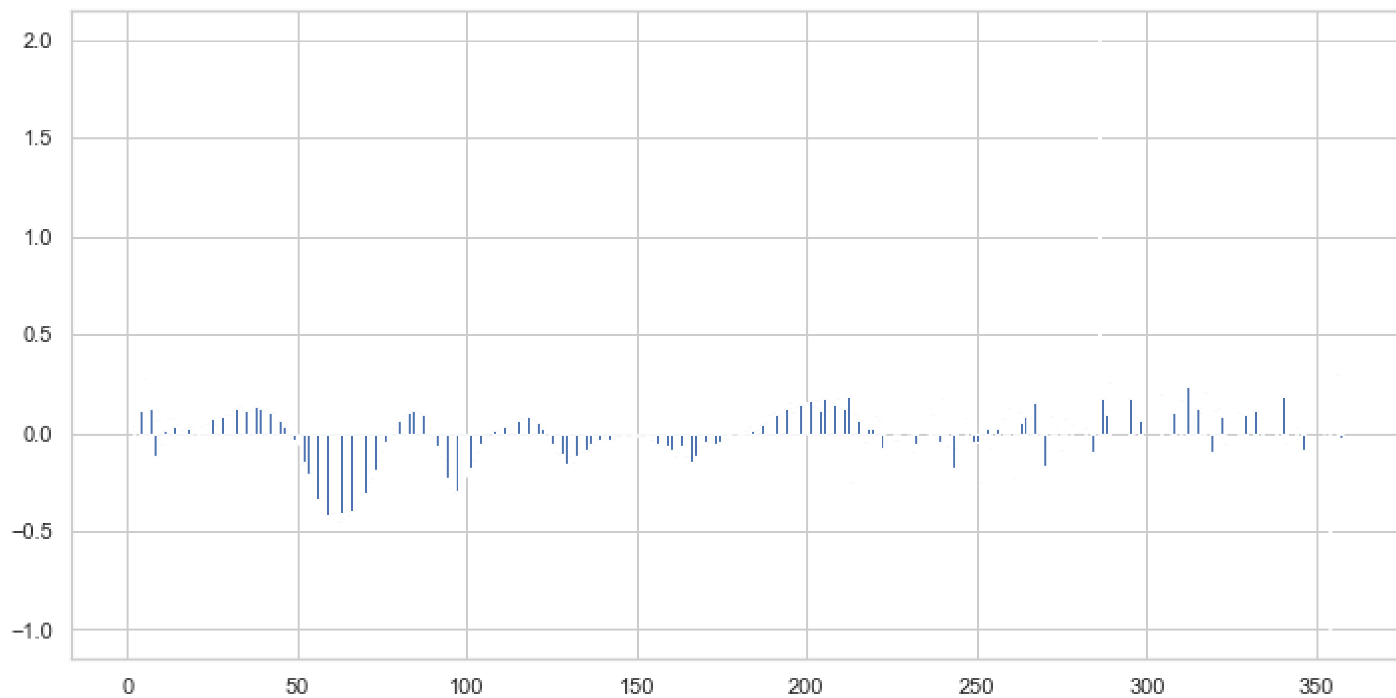
In [56]:

```
wd_plot(5, 4)
```



In [57]:

```
wd_plot(5, 4, 1)
```



In [48]:

```
# Function create confusion matrix of wind speed diff by angle
# Input: angle
# Output: confusion matrix

def conf_ma(angle):
    col2 = ['turbines','WTG1','WTG2','WTG3','WTG4','WTG5','WTG6','WTG7','WTG8','WTG9','WTG
    row2 = ['WTG1','WTG2','WTG3','WTG4','WTG5','WTG6','WTG7','WTG8','WTG9','WTG10','WTG11'
    ma_weights = pd.DataFrame(columns=col2)
    ma_weights['turbines'] = row2
    ma_weights = ma_weights.set_index('turbines')

    for i in ma_weights:
        turbine1 = i[3:]
        for n in ma_weights:
            turbine2 = n[3:]
```

```python
            number = wind_dif(turbine1, turbine2, angle)
            if number == None:
                print("No entries for this angle")
                return

            ma_weights[n].loc[[i]] = number

    return ma_weights
```

In [49]:

```python
conf_ma(95)
```

Out[49]:

turbines	WTG1	WTG2	WTG3	WTG4	WTG5	WTG6	WTG7	WTG8	WTG9	WTG10	...	WTG23	WTG24
WTG1	0.0	-0.2	0.1	0.7	2.1	0.7	0.0	0.6	-0.2	-0.0	...	-0.2	-0.3
WTG2	0.2	0.0	0.3	0.9	2.3	0.9	0.2	0.8	-0.1	0.2	...	-0.0	-0.2
WTG3	-0.1	-0.3	0.0	0.6	2.0	0.6	-0.1	0.5	-0.3	-0.1	...	-0.3	-0.4
WTG4	-0.7	-0.9	-0.6	0.0	1.4	0.0	-0.7	-0.1	-0.9	-0.7	...	-0.9	-1.0
WTG5	-2.1	-2.3	-2.0	-1.4	0.0	-1.4	-2.1	-1.5	-2.3	-2.1	...	-2.3	-2.4
WTG6	-0.7	-0.9	-0.6	-0.0	1.4	0.0	-0.7	-0.1	-1.0	-0.7	...	-0.9	-1.0
WTG7	-0.0	-0.2	0.1	0.7	2.1	0.7	0.0	0.6	-0.2	-0.0	...	-0.2	-0.3
WTG8	-0.6	-0.8	-0.5	0.1	1.5	0.1	-0.6	0.0	-0.8	-0.6	...	-0.8	-0.9
WTG9	0.2	0.1	0.3	0.9	2.3	1.0	0.2	0.8	0.0	0.2	...	0.0	-0.1
WTG10	0.0	-0.2	0.1	0.7	2.1	0.7	0.0	0.6	-0.2	0.0	...	-0.2	-0.3
WTG11	0.1	-0.1	0.1	0.8	2.2	0.8	0.1	0.7	-0.2	0.1	...	-0.1	-0.3
WTG12	0.0	-0.2	0.1	0.7	2.1	0.7	0.0	0.6	-0.2	-0.0	...	-0.2	-0.3
WTG13	-1.8	-2.0	-1.8	-1.2	0.2	-1.1	-1.8	-1.2	-2.1	-1.9	...	-2.1	-2.2
WTG14	0.0	-0.2	0.1	0.7	2.1	0.7	0.0	0.6	-0.2	0.0	...	-0.2	-0.3
WTG15	0.2	-0.0	0.2	0.8	2.2	0.9	0.2	0.8	-0.1	0.1	...	-0.1	-0.2
WTG16	0.1	-0.1	0.1	0.8	2.2	0.8	0.1	0.7	-0.2	0.1	...	-0.1	-0.3
WTG17	0.1	-0.1	0.2	0.8	2.2	0.8	0.1	0.7	-0.1	0.1	...	-0.1	-0.2
WTG18	0.2	-0.0	0.2	0.9	2.3	0.9	0.2	0.8	-0.1	0.2	...	-0.0	-0.2
WTG19	0.0	-0.2	0.1	0.7	2.1	0.7	0.0	0.6	-0.2	0.0	...	-0.2	-0.3
WTG20	-0.1	-0.3	-0.0	0.6	2.0	0.6	-0.1	0.5	-0.3	-0.1	...	-0.3	-0.4
WTG21	0.0	-0.2	0.1	0.7	2.1	0.7	0.0	0.6	-0.2	-0.0	...	-0.2	-0.3
WTG22	0.4	0.2	0.5	1.1	2.5	1.1	0.4	1.0	0.2	0.4	...	0.2	0.1
WTG23	0.2	0.0	0.3	0.9	2.3	0.9	0.2	0.8	-0.0	0.2	...	0.0	-0.1
WTG24	0.3	0.2	0.4	1.0	2.4	1.0	0.3	0.9	0.1	0.3	...	0.1	0.0
WTG25	-0.2	-0.4	-0.1	0.5	1.9	0.5	-0.2	0.4	-0.4	-0.2	...	-0.4	-0.5
WTG26	0.2	-0.0	0.2	0.9	2.3	0.9	0.2	0.8	-0.1	0.1	...	-0.0	-0.2
WTG27	0.1	-0.1	0.2	0.8	2.2	0.8	0.1	0.7	-0.1	0.1	...	-0.1	-0.2
WTG28	0.3	0.1	0.3	1.0	2.4	1.0	0.3	0.9	0.0	0.2	...	0.1	-0.1
WTG29	0.2	-0.0	0.2	0.9	2.3	0.9	0.2	0.8	-0.1	0.2	...	-0.0	-0.2

turbines	WTG1	WTG2	WTG3	WTG4	WTG5	WTG6	WTG7	WTG8	WTG9	WTG10	...	WTG23	WTG24
WTG30	0.5	0.3	0.6	1.2	2.6	1.2	0.5	1.1	0.2	0.5	...	0.3	0.2
WTG31	0.2	0.1	0.3	0.9	2.3	1.0	0.2	0.8	0.0	0.2	...	0.0	-0.1
WTG32	-1.5	-1.6	-1.4	-0.8	0.6	-0.8	-1.5	-0.9	-1.7	-1.5	...	-1.7	-1.8

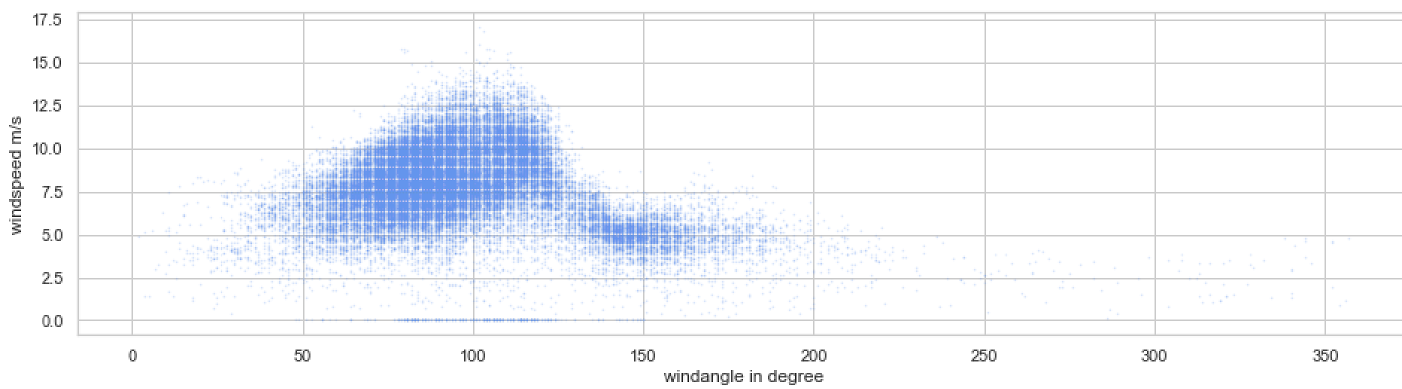
32 rows × 32 columns

Short Analysis

- W04, W06, W02, W01, and W10

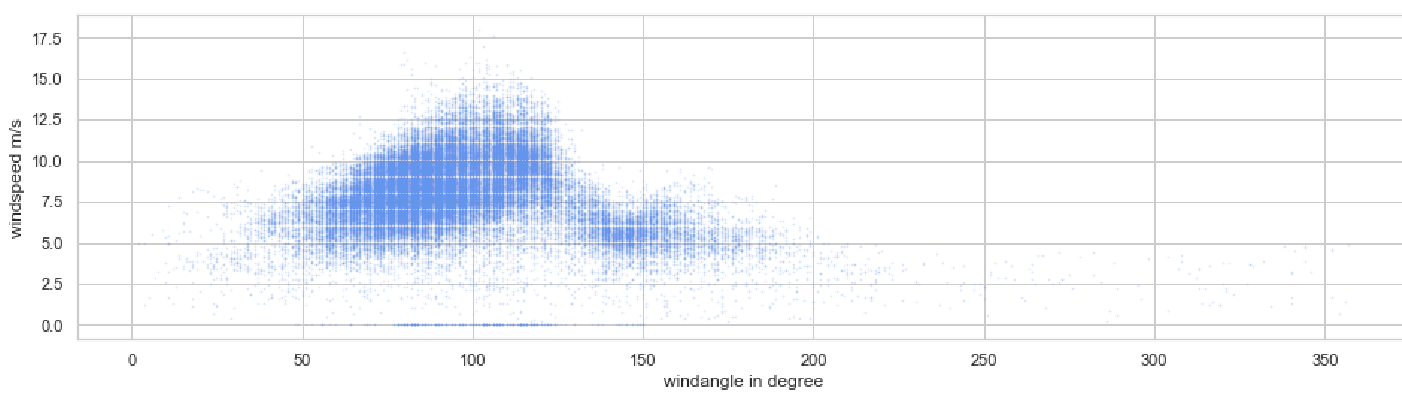
In [51]:

```
wind_speed(1)
```



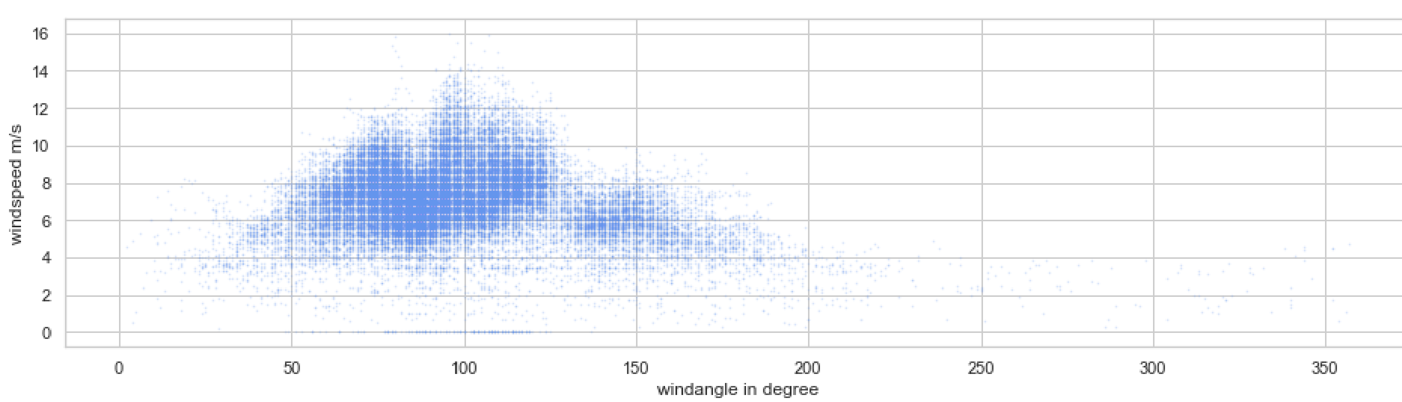
In [52]:

```
wind_speed(2)
```

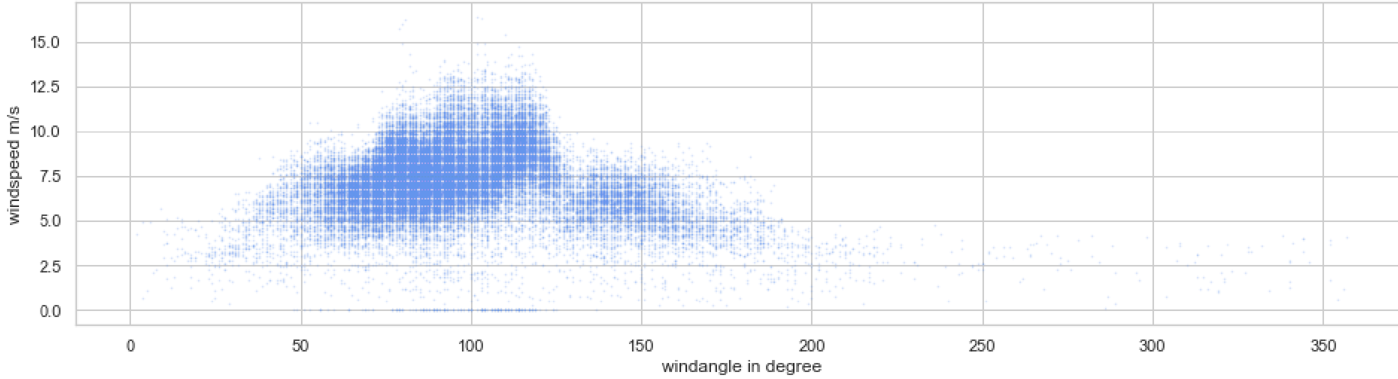


In [53]:

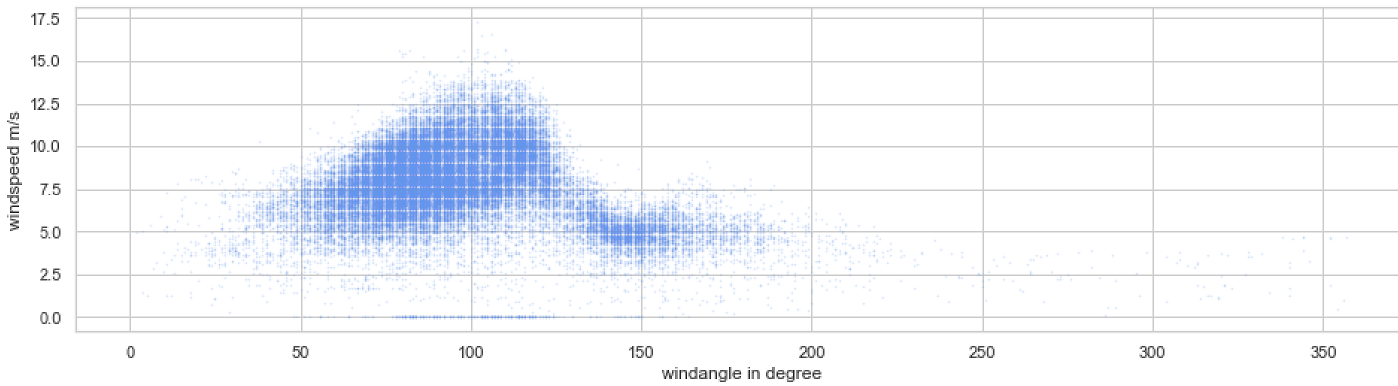
```
wind_speed(4)
```



In [54]:
```
wind_speed(6)
```



In [55]:
```
wind_speed(10)
```



Next Steps: Further Research / Possibilities

- Use confusion matrix to get weighted graph network
- Develop more sophisticated turbine interaction rules, e.g. weighted proximity based on wind angle (simulate wind tunnel)
- Rethink the model -> improve model, consider more effects to decrease the generalisation of the model