

# МОДЕЛЬ СОЗДАНИЯ АРХИВА РАСЧЁТНЫХ ПОГОДНЫХ ДАННЫХ для конкретной локации по данным нескольких референсных метеостанций

**Цель:** Воссоздать архив погодных явлений для локации Агробиостанции МГУ в пос. Чашниково Солнечногорского р-она Московской области

*==== Тетрадь 5: Исследование аномалий, пропусков и ошибок, а также восстановление, исправление и моделирование значений для каждого индивидуального параметра: численные и категориальный показатели состояния воздуха ===*

## 0. Подготовка данных 5-й тетради

### 0.0. Импорт необходимых библиотек, настройки представления, константы

```
In [1]: # Python interpreter version: 3.9.12
# Системная конфигурация
import sys

# Системные утилиты
from copy import copy, deepcopy

# Работа с файловой системой
from os import listdir
from os.path import isfile, join
from os import makedirs

# Сохранение и восстановление сохранённых объектов Python
```

```
import joblib

# Вывод данных
from pprint import pprint
from io import StringIO

# Вычисления
from math import degrees, radians, cos, sin, asin, atan, sqrt, floor, log
import numpy as np # v. 1.23.4
import pandas as pd # v. 1.5.1
from scipy.stats import norm # v.1.9.3
from scipy.spatial.distance import pdist # v.1.9.3

# Работа с временем и датами
import datetime as dt
import time

# Работа со строками
import re
import pymorphy2 # v. 0.9.1

# Машинное обучение
# Последовательность исполнения
from sklearn.pipeline import Pipeline, FeatureUnion # v.1.1.3

# - подготовка данных
from sklearn.model_selection import train_test_split # v.1.1.3
from sklearn.preprocessing import (StandardScaler, MinMaxScaler, RobustScaler,
                                   QuantileTransformer, PowerTransformer, SplineTransformer,
                                   PolynomialFeatures) # v.1.1.3
from sklearn.decomposition import PCA, TruncatedSVD # v.1.1.3
from sklearn.feature_selection import (GenericUnivariateSelect,
                                       f_classif, f_regression,
                                       mutual_info_classif,
                                       mutual_info_regression) # v.1.1.3

# - подбор параметров моделей
from sklearn.model_selection import (cross_val_predict, GridSearchCV,
                                      StratifiedKFold, KFold) # v.1.1.3

# - регрессоры
from sklearn.linear_model import LinearRegression, HuberRegressor, SGDRegressor # v.1.1.3
from sklearn.tree import DecisionTreeRegressor # v.1.1.3
from sklearn.ensemble import (AdaBoostRegressor, BaggingRegressor,
                             RandomForestRegressor, StackingRegressor,
```

```
ExtraTreesRegressor,
HistGradientBoostingRegressor) # v.1.1.3
from sklearn.neural_network import MLPRegressor # v.1.1.3

# - кластеризация
from sklearn.cluster import KMeans, DBSCAN # v.1.1.3

# - классификаторы
from sklearn.ensemble import (AdaBoostClassifier,
                               GradientBoostingClassifier,
                               HistGradientBoostingClassifier,
                               RandomForestClassifier,
                               ExtraTreesClassifier,
                               StackingClassifier) # v.1.1.3
from sklearn.tree import DecisionTreeClassifier # v.1.1.3
from sklearn.svm import SVC # v.1.1.3
from sklearn.linear_model import LogisticRegression # v.1.1.3
from sklearn.neural_network import MLPClassifier # v.1.1.3

# from xgboost import XGBClassifier # v. 1.5.0
# ...
# - метрики качества
from sklearn.metrics import (max_error, mean_absolute_error, mean_squared_error,
                             r2_score, log_loss,
                             roc_auc_score, f1_score, precision_score, recall_score,
                             RocCurveDisplay) # v.1.1.3

# - Библиотека SciKit GStat:
import skgstat as skg # v. 1.0.1

# Построение визуализаций
import matplotlib as mpl # v. 3.5.3
import matplotlib.pyplot as plt # v. 3.5.3
import matplotlib.lines as mlines # v. 3.5.3
import seaborn as sns # v. 0.12.1
import seaborn.objects as so # v. 0.12.1
import missingno as msno # v. 0.4.2

# EDA tools
import sweetviz as sv
#from pandas_profiling import ProfileReport

# Настройки
import warnings
```

Настроим отображение вывода результатов кода в нескольких ячейках

```
In [2]: from IPython.core.interactiveshell import InteractiveShell  
InteractiveShell.ast_node_interactivity = "all"
```

Для удобства отображения данных изменим опции максимум отображаемых строк и столбцов.

```
In [3]: pd.set_option('display.max_rows', 400) # изменим максимум отображаемых строк  
pd.set_option('display.max_columns', 50) # изменим максимум отображаемых столбцов
```

Установим для Seaborn настройки темы по умолчанию.

In [4]: sns.set\_theme()

Определим константы и значения (сообразно предыдущим тетрадям)

```
Out[5]: ['T',
 'T_min',
 'T_max',
 'P_sea',
 'P_station',
 'P_drift',
 'Humid',
 'Dew_point',
 'Soil_T',
 'Snow_height']
```

## 0.1. Импорт данных

```
In [6]: # Определим значения переменных path (по результатам обработки данных в 4-й четверти)
# path = 'data/csv/' # общий путь к данным
# path1 = path + 'locations/' # путь к архивам метеостанций
# path2 = path + 'parameters/' # путь к архивам параметров

path = 'data/csv/' # общий путь к данным

# путь к архивам метеостанций, в последней редакции (Snow_height):
path1 = f'{path}predict/{PARAMETER62}/locations/'
# путь к архивам параметров с рассчётыми значениями:
path2 = path + 'predict/'
# путь к исходным архивам параметров:
path3 = f'{path}raw/'
```

### 0.1.1. Данные метеостанций

#### 0.1.1.1. Информация о метеостанциях

```
In [7]: # Загружаем файл с общей информацией о метеостанциях
df_stations = pd.read_csv(filepath_or_buffer=path + 'df_stations.csv', index_col=0)
df_stations.head(2)
# Загружаем файл с расстояниями между метеостанциями
df_station_dists = pd.read_csv(filepath_or_buffer=path + 'df_station_dists.csv', index_col=0)
df_station_dists.head(2)
# Загружаем файл с начальными азимутами между метеостанциями
df_station_bearings = pd.read_csv(filepath_or_buffer=path + 'df_station_bearings.csv', index_col=0)
df_station_bearings.head(2)
```

```
# Загружаем файл с линейными координатами метеостанций
df_stations_lin_coords = pd.read_csv(filepath_or_buffer=path + 'df_stations_lin_coords.csv', index_col=0)
df_stations_lin_coords.head(2)
```

Out[7]:

	station_name	station_ID	height	latitude	longitude	degree_lo	minute_lo	lo	degree_la	minute_la	la	comment
0	Вышний Волочек	26393	161	N57.583333	E34.566667	57	35	N	34	34	E	validate 26393.11.07.2005.09.06.2022.1.0
1	Старица	26499	185	N56.500000	E34.933333	56	30	N	34	56	E	validate 26499.01.02.2005.09.06.2022.1.0

Out[7]:

	station_ID	station	LaN	LoE	height	V_Volochek	Staritsa	Kashyn	Tver	Klin	Dmitrov	Volokolamsk	Mozhais
0	26393	V_Volochek	57.583333	34.566667	161	0.000000	122.520236	182.287149	114.810932	190.589931	225.030989	193.100035	246.07462
1	26499	Staritsa	56.500000	34.933333	185	122.520236	0.000000	186.562842	72.307124	111.270810	160.570705	81.891761	127.90687

Out[7]:

	station_ID	station	LaN	LoE	height	V_Volochek	Staritsa	Kashyn	Tver	Klin	Dmitrov	Volokolamsk	Mozhais
0	26393	V_Volochek	57.583333	34.566667	161	0.00000	349.720491	279.454332	315.263759	317.734920	308.198351	335.03763	339.67437
1	26499	Staritsa	56.500000	34.933333	185	169.41282	0.000000	240.670335	237.073671	280.332008	276.381371	311.44173	329.20554

Out[7]:

	station_ID	station	LaN	LoE	height	lin_ver	lin_hor
0	26393	V_Volochek	57.583333	34.566667	161	296.603273	0.00000
1	26499	Staritsa	56.500000	34.933333	185	176.108200	23.44064

## 0.1.2. Данные архивов погоды

### 0.1.2.1. Чтение архивов метеостанций

In [8]:

```
# Создадим список файлов с архивами метеостанций
list_df_locations_files = [file_name for file_name in listdir(path1) if isfile(join(path1, file_name))]
list_df_locations_files
```

```
Out[8]: ['df_Chashnikovo.csv',
 'df_Dmitrov.csv',
 'df_Kashyn.csv',
 'df_Klin.csv',
 'df_Mozhaisk.csv',
 'df_Naro_Fominsk.csv',
 'df_Nemchinovka.csv',
 'df_N_Jerusalem.csv',
 'df_Rfrnce_point.csv',
 'df_Serpukhov.csv',
 'df_Staritsa.csv',
 'df_Tver.csv',
 'df_Volokolamsk.csv',
 'df_V_Volochek.csv']
```

```
In [9]: dict_df_locations = {} # Инициализируем словарь датафреймов
for file_name in list_df_locations_files: # По списку csv файлов
    name_df = file_name[:-4] # Вычленяем название датафрейма из названия файла
    file_path = path1 + file_name # Формируем путь к файлу
    print(file_path, ' - ', end='') # КОНТРОЛЬ: Обрабатываемый файл
    # Создаём ключ (название DF) из названия файла
    # и записываем в словарь по этому ключу соответствующий DF
    dict_df_locations[f'{name_df}'] = pd.read_csv(
        file_path, index_col=0, parse_dates=True, infer_datetime_format = True, low_memory=False)
    print('O.K.')
# Выводим полученные ключи словаря
dict_df_locations.keys()
```

```
data/csv/predict/Snow_height/locations/df_Chashnikovo.csv - O.K.
data/csv/predict/Snow_height/locations/df_Dmitrov.csv - O.K.
data/csv/predict/Snow_height/locations/df_Kashyn.csv - O.K.
data/csv/predict/Snow_height/locations/df_Klin.csv - O.K.
data/csv/predict/Snow_height/locations/df_Mozhaisk.csv - O.K.
data/csv/predict/Snow_height/locations/df_Naro_Fominsk.csv - O.K.
data/csv/predict/Snow_height/locations/df_Nemchinovka.csv - O.K.
data/csv/predict/Snow_height/locations/df_N_Jerusalem.csv - O.K.
data/csv/predict/Snow_height/locations/df_Rfrnce_point.csv - O.K.
data/csv/predict/Snow_height/locations/df_Serpukhov.csv - O.K.
data/csv/predict/Snow_height/locations/df_Staritsa.csv - O.K.
data/csv/predict/Snow_height/locations/df_Tver.csv - O.K.
data/csv/predict/Snow_height/locations/df_Volokolamsk.csv - O.K.
data/csv/predict/Snow_height/locations/df_V_Volochek.csv - O.K.
```

```
Out[9]: dict_keys(['df_Chashnikovo', 'df_Dmitrov', 'df_Kashyn', 'df_Klin', 'df_Mozhaisk', 'df_Naro_Fominsk', 'df_Nemchinovka', 'df_N_Jerusalem', 'df_Rfrnce_point', 'df_Serpukhov', 'df_Staritsa', 'df_Tver', 'df_Volokolamsk', 'df_V_Volochek'])
```

```
In [10]: for name in dict_df_locations.keys():
    dict_df_locations[name].sample(3)
```

```
Out[10]:
```

	T	T_min	T_max	P_sea	P_station	P_drift	Humid	Dew_point	Soil_T	Snow_height
<b>2011-08-05 21:00:00</b>	18.271237	18.271237	23.214063	760.448873	741.482674	0.252606	58.762651	10.057337	NaN	NaN
<b>2018-06-16 03:00:00</b>	7.808328	7.808328	20.773229	766.017934	746.210641	0.458377	79.812813	4.541469	NaN	NaN
<b>2017-08-08 00:00:00</b>	15.037914	15.037914	23.131858	764.208531	744.937439	0.686255	79.379070	11.494722	NaN	NaN

```
Out[10]:
```

	T	P_station	P_sea	P_drift	Humid	Wind_dir	Wind_dir360	Wind_dir6k	Wind_speed	Gusts	Gusts_3h	Cloudnes
<b>Dmitrov_Local_time</b>												
<b>2005-10-18 18:00:00</b>	3.100000	744.200000	760.700000	-0.30000	89.000000	NNW		337.5	5625.0	4.0	NaN	NaN
<b>2019-06-27 12:00:00</b>	13.800000	739.600000	755.500000	-1.70000	85.000000	ESE		112.5	1875.0	3.0	NaN	NaN
<b>2015-12-17 12:00:00</b>	-5.567546	749.045631	766.397734	1.49307	82.370411	NaN		NaN	NaN	NaN	NaN	Nal

Out[10]:

	T	P_station	P_sea	P_drift	Humid	Wind_dir	Wind_dir360	Wind_dir6k	Wind_speed	Gusts	Gusts_3h	Cloudnes
<b>Kashyn_Local_time</b>												
<b>2009-03-07 15:00:00</b>	-1.100000	751.300000	764.300000	0.508180	53.000000	SE	135.0	2250.0	3.0	NaN	NaN	от 9 мене 100%
<b>2010-12-05 00:00:00</b>	-6.845151	749.722285	763.141416	2.822285	81.502553	NaN	NaN	NaN	NaN	NaN	NaN	NaN
<b>2021-02-08 00:00:00</b>	-21.500000	749.600000	763.900000	1.900000	84.000000	штиль	360.0	6000.0	0.0	NaN	NaN	0%

Out[10]:

	T	P_station	P_sea	P_drift	Humid	Wind_dir	Wind_dir360	Wind_dir6k	Wind_speed	Gusts	Gusts_3h	Cloudness	Wthr_curr	Prcptt
<b>Klin_Local_time</b>														
<b>2018-12-01 12:00:00</b>	-7.5	762.2	778.6	-1.0	64.0	SSE	157.5	2625.0	2.0	NaN	NaN	0%	NaN	
<b>2016-09-06 06:00:00</b>	8.6	748.0	763.2	1.5	95.0	WSW	247.5	4125.0	1.0	NaN	NaN	20–30%.	NaN	
<b>2016-04-11 15:00:00</b>	11.7	749.5	764.5	0.0	44.0	NNE	22.5	375.0	3.0	NaN	NaN	60%.	NaN	

Out[10]:

	T	P_station	P_sea	P_drift	Humid	Wind_dir	Wind_dir360	Wind_dir6k	Wind_speed	Gusts	Gusts_3h	Cloudr
<b>Mozhaisk_Local_time</b>												
<b>2014-04-02 21:00:00</b>	-2.600000	743.200000	760.600000	0.500000	85.000000	W	270.0	4500.0	1.0	NaN	NaN	70 – 8
<b>2016-04-25 15:00:00</b>	16.698875	740.476663	756.834691	-0.634487	39.974608	NaN	NaN	NaN	NaN	NaN	NaN	N
<b>2010-02-19 15:00:00</b>	-5.200000	738.200000	755.700000	-1.900000	95.000000	SE	135.0	2250.0	2.0	NaN	NaN	10

Out[10]:

	T	P_station	P_sea	P_drift	Humid	Wind_dir	Wind_dir360	Wind_dir6k	Wind_speed	Gusts	Gusts_3h	Cloudness	Wthr_cu
<b>Naro_Fominsk_Local_time</b>													
<b>2015-06-29 12:00:00</b>	18.8	744.1	760.7	0.5	58.0	NNE	22.5	375.0	5.0	NaN	NaN	от 90 менее 100%	N
<b>2014-03-29 00:00:00</b>	-2.2	746.9	764.9	0.6	46.0	N	0.0	0.0	8.0	13.0	17.0	0%	N
<b>2011-05-20 18:00:00</b>	23.4	746.6	763.2	-0.8	39.0	WSW	247.5	4125.0	3.0	NaN	NaN	70 – 80%.	N

Out[10]:

	T	P_station	P_sea	P_drift	Humid	Wind_dir	Wind_dir360	Wind_dir6k	Wind_speed	Gusts	Gusts_3h	Cloudness
Nemchinovka_Local_time												
2017-06-14 09:00:00	13.600000	729.700000	745.300000	0.371370	79.000000	SW	225.0	3750.0	3.0	NaN	NaN	NaN
2022-05-21 12:00:00	14.500000	742.200000	758.000000	-1.400000	32.000000	S	180.0	3000.0	2.0	NaN	NaN	7
2005-11-27 18:00:00	1.643764	746.476603	762.930024	0.229108	91.706017	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Out[10]:

	T	P_station	P_sea	P_drift	Humid	Wind_dir	Wind_dir360	Wind_dir6k	Wind_speed	Gusts	Gusts_3h	Cloudness	Wthr_c
N_Jerusalem_Local_time													
2020-08-23 06:00:00	9.0	748.8	763.3	-0.4	98.0	штиль	360.0	6000.0	0.0	NaN	NaN	40%.	Дым
2014-01-01 00:00:00	1.1	756.4	771.4	0.5	78.0	WSW	247.5	4125.0	1.0	NaN	NaN	100%.	Н
2007-11-25 12:00:00	0.9	743.0	758.0	0.9	92.0	WSW	247.5	4125.0	3.0	NaN	NaN	100%.	С непрерывн слабы с наблюде

Out[10]:

	T	T_min	T_max	P_sea	P_station	P_drift	Humid	Dew_point	Soil_T	Snow_height
<b>2019-05-13 15:00:00</b>	21.556131	11.221252	21.556131	766.843225	754.648105	-0.619760	39.268019	7.103674	NaN	NaN
<b>2021-07-07 21:00:00</b>	21.732270	21.732270	29.005178	767.209633	755.015916	0.253055	64.657554	14.774234	NaN	NaN
<b>2016-12-26 09:00:00</b>	-0.397621	-1.873707	-0.397621	757.913651	744.898762	-0.130682	89.130042	-1.965808	NaN	23.846624

Out[10]:

	T	P_station	P_sea	P_drift	Humid	Wind_dir	Wind_dir360	Wind_dir6k	Wind_speed	Gusts	Gusts_3h	Cloudness	W
<b>Serpukhov_Local_time</b>													
<b>2021-05-05 06:00:00</b>	5.7	740.600000	755.600000	1.000000	63.0	WSW	247.5	4125.0	4.0	NaN	NaN	70 – 80%.	
<b>2013-08-27 00:00:00</b>	10.9	750.200000	765.100000	0.600000	81.0	NE	45.0	750.0	3.0	NaN	NaN	0%	
<b>2011-03-01 21:00:00</b>	-12.6	762.835029	778.632541	1.035029	75.0	штиль	360.0	6000.0	0.0	NaN	NaN	0%	

Out[10]:

	T	P_station	P_sea	P_drift	Humid	Wind_dir	Wind_dir360	Wind_dir6k	Wind_speed	Gusts	Gusts_3h	Cloudness	Wthr_curr
<b>Staritsa_Local_time</b>													
<b>2016-12-24 21:00:00</b>	-0.1	742.6	760.0	-1.6	77.0	SW	225.0	3750.0	6.0	10.0	11.0	от 90 менее 100%	NaN
<b>2018-05-06 21:00:00</b>	12.4	750.3	767.1	0.6	58.0	N	0.0	0.0	1.0	NaN	NaN	0%	NaN
<b>2015-03-04 00:00:00</b>	0.2	742.3	759.7	0.1	96.0	SSE	157.5	2625.0	3.0	NaN	NaN	100%.	Ливневый снег слабый в срок наблюдения   или за ...

Out[10]:

	T	P_station	P_sea	P_drift	Humid	Wind_dir	Wind_dir360	Wind_dir6k	Wind_speed	Gusts	Gusts_3h	Cloudness	Wth
Tver_Local_time													
<b>2012-11-29 06:00:00</b>	-4.4	745.600000	758.622009	-0.400000	93.0	E	90.0	1500.0	5.0	NaN	NaN	100%.	непрерыв силь наблюд
<b>2018-05-20 09:00:00</b>	8.2	752.423873	764.606698	2.923873	92.0	N	0.0	0.0	1.0	NaN	NaN	100%.	Ливневый дождь слабый наб
<b>2013-01-01 21:00:00</b>	0.0	745.559296	757.996372	-0.053621	97.0	SSE	157.5	2625.0	2.0	NaN	NaN	100%.	Состо неба в о измены

Out[10]:

	T	P_station	P_sea	P_drift	Humid	Wind_dir	Wind_dir360	Wind_dir6k	Wind_speed	Gusts	Gusts_3h	Cloudness	Wthr
Volokolamsk_Local_time													
2022-05-16 06:00:00	7.2	737.3	755.1	0.3	70.0	NW	315.0	5250.0	3.0	NaN	NaN	70 – 80%.	
2019-07-17 21:00:00	15.1	737.0	754.2	-0.2	85.0	ESE	112.5	1875.0	2.0	NaN	NaN	от 90 менее 100%	Ливневый дожь, слабый(и набл)
2009-06-07 12:00:00	15.6	742.0	759.5	-0.3	55.0	SW	225.0	3750.0	2.0	NaN	NaN	от 90 менее 100%	

Out[10]:

	T	P_station	P_sea	P_drift	Humid	Wind_dir	Wind_dir360	Wind_dir6k	Wind_speed	Gusts	Gusts_3h	Cloudness	
V_Volochev_Local_time													
2008-04-02 06:00:00	-3.141194	752.971388	768.5	-0.428612	76.950035	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2008-05-22 03:00:00	3.200000	750.800000	766.6	0.108832	97.000000	штиль	360.0	6000.0	0.0	NaN	NaN	70 – 80%.	
2009-05-16 09:00:00	5.400000	746.200000	761.7	0.166644	84.000000	N	0.0	0.0	1.0	NaN	NaN	от 90 менее 100%	

### 0.1.2.2. Чтение архивов параметров

```
In [11]: # Создадим список файлов с архивами параметров
list_df_parameters_files = [file_name for file_name in listdir(path3) if isfile(join(path3, file_name))]
list_df_parameters_files
```

```
Out[11]: ['df_Cloudness.csv',
 'df_C1_bottom.csv',
 'df_C1_cirrus.csv',
 'df_C1_Cumls.csv',
 'df_C1_cumls_hi.csv',
 'df_C1_viewd.csv',
 'df_Depo_diam_mm.csv',
 'df_Dew_point.csv',
 'df_Gusts.csv',
 'df_Gusts_3h.csv',
 'df_Humid.csv',
 'df_Prcptn.csv',
 'df_Prcptn_depo.csv',
 'df_Prcptn_like.csv',
 'df_Prcptn_tdelt.csv',
 'df_P_drift.csv',
 'df_P_sea.csv',
 'df_P_station.csv',
 'df_Snow_height.csv',
 'df_Soil.csv',
 'df_Soil_cover.csv',
 'df_Soil_T.csv',
 'df_T.csv',
 'df_T_max.csv',
 'df_T_min.csv',
 'df_Visibility.csv',
 'df_Wind_dir.csv',
 'df_Wind_dir360.csv',
 'df_Wind_dir6k.csv',
 'df_Wind_speed.csv',
 'df_Wthr_3h.csv',
 'df_Wthr_3h2.csv',
 'df_Wthr_curr.csv']
```

```
In [12]: # Запишем данные архивов параметров в словарь с учётом данных предыдущих тетрадей
dict_df_parameters = {} # Инициализируем словарь датафреймов
for file_name in list_df_parameters_files: # По списку csv файлов
    name_df = file_name[:-4] # Вычленяем название датафрейма из названия файла
    # проверяем, есть ли файл с исправленными значениями параметра (по списку)
```

```
if name_df[3:] in list_const_param:
    file_path = f'{path2}{name_df[3:]}/{file_name}' # Формируем путь к файлу
else:
    file_path = f'{path3}{file_name}' # Формируем путь к файлу

print(file_path, ' - ', end='') # КОНТРОЛЬ: Обрабатываемый файл

# Создаём ключ (название DF) из названия файла
# и записываем в словарь по этому ключу соответствующий DF
dict_df_parameters[f'{name_df}'] = pd.read_csv(
    file_path, index_col=0, parse_dates=True, infer_datetime_format = True, low_memory=False)
print('O.K.')

# Выводим полученные ключи словаря
dict_df_parameters.keys()
```

```
data/csv/raw/df_Cloudness.csv - O.K.
data/csv/raw/df_Cl_bottom.csv - O.K.
data/csv/raw/df_Cl_cirrus.csv - O.K.
data/csv/raw/df_Cl_Cumls.csv - O.K.
data/csv/raw/df_Cl_cumls_hi.csv - O.K.
data/csv/raw/df_Cl_viewd.csv - O.K.
data/csv/raw/df_Depo_diam_mm.csv - O.K.
data/csv/predict/Dew_point/df_Dew_point.csv - O.K.
data/csv/raw/df_Gusts.csv - O.K.
data/csv/raw/df_Gusts_3h.csv - O.K.
data/csv/predict/Humid/df_Humid.csv - O.K.
data/csv/raw/df_Prcpttn.csv - O.K.
data/csv/raw/df_Prcpttn_depo.csv - O.K.
data/csv/raw/df_Prcpttn_like.csv - O.K.
data/csv/raw/df_Prcpttn_tdelt.csv - O.K.
data/csv/predict/P_drift/df_P_drift.csv - O.K.
data/csv/predict/P_sea/df_P_sea.csv - O.K.
data/csv/predict/P_station/df_P_station.csv - O.K.
data/csv/predict/Snow_height/df_Snow_height.csv - O.K.
data/csv/raw/df_Soil.csv - O.K.
data/csv/raw/df_Soil_cover.csv - O.K.
data/csv/predict/Soil_T/df_Soil_T.csv - O.K.
data/csv/predict/T/df_T.csv - O.K.
data/csv/predict/T_max/df_T_max.csv - O.K.
data/csv/predict/T_min/df_T_min.csv - O.K.
data/csv/raw/df_Visibility.csv - O.K.
data/csv/raw/df_Wind_dir.csv - O.K.
data/csv/raw/df_Wind_dir360.csv - O.K.
data/csv/raw/df_Wind_dir6k.csv - O.K.
data/csv/raw/df_Wind_speed.csv - O.K.
data/csv/raw/df_Wthr_3h.csv - O.K.
data/csv/raw/df_Wthr_3h2.csv - O.K.
data/csv/raw/df_Wthr_curr.csv - O.K.

Out[12]: dict_keys(['df_Cloudness', 'df_Cl_bottom', 'df_Cl_cirrus', 'df_Cl_Cumls', 'df_Cl_cumls_hi', 'df_Cl_viewd', 'df_Depo_diam_mm', 'df_Dew_point', 'df_Gusts', 'df_Gusts_3h', 'df_Humid', 'df_Prcpttn', 'df_Prcpttn_depo', 'df_Prcpttn_like', 'df_Prcpttn_tdelt', 'df_P_drift', 'df_P_sea', 'df_P_station', 'df_Snow_height', 'df_Soil', 'df_Soil_cover', 'df_Soil_T', 'df_T', 'df_T_max', 'df_T_min', 'df_Visibility', 'df_Wind_dir', 'df_Wind_dir360', 'df_Wind_dir6k', 'df_Wind_speed', 'df_Wthr_3h', 'df_Wthr_3h2', 'df_Wthr_curr'])
```

```
In [13]: for name in dict_df_parameters.keys():
    dict_df_parameters[name].sample(3)
```

	Cloudness#V_Volochev	Cloudness#Staritsa	Cloudness#Kashyn	Cloudness#Tver	Cloudness#Klin	Cloudness#Dmitrov	Cloudness#Volokolamsk	Cloudness#Leningrad
2015-10-13 18:00:00	0%	0%	20–30%.	0%	0%	NaN	NaN	NaN
2013-11-14 00:00:00	100%.	100%.	100%.	100%.	100%.	NaN	NaN	NaN
2013-07-18 12:00:00	от 90 менее 100%	70 – 80%.	60%.	70 – 80%.	40%.	70 – 80%.	от 90 менее 100%	от 90 менее 100%

	Cl_bottom#V_Volochev	Cl_bottom#Staritsa	Cl_bottom#Kashyn	Cl_bottom#Tver	Cl_bottom#Klin	Cl_bottom#Dmitrov	Cl_bottom#Volokolamsk	Cloudness#Leningrad
2014-01-03 03:00:00	600-1000	300-600	300-600	300-600	600-1000	300-600	2500 или более, или облаков нет.	2500 или более, или облаков нет.
2006-01-18 18:00:00	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2009-04-23 12:00:00	NaN	2500 или более, или облаков нет.	NaN	2500 или более, или облаков нет.	NaN	2500 или более, или облаков нет.	2500 или более, или облаков нет.	2500 или более, или облаков нет.

Out[13]:

	Cl_cirrus#V_Volochev	Cl_cirrus#Staritsa	Cl_cirrus#Kashyn	Cl_cirrus#Tver	Cl_cirrus#Klin	Cl_cirrus#Dmitrov	Cl_cirrus#Volokolamsk	Cl_cirrus#Moz
--	----------------------	--------------------	------------------	----------------	----------------	-------------------	-----------------------	---------------

<b>2018-11-07 06:00:00</b>	NaN	NaN	NaN	Перистых, перисто-кучевых или перисто-слоистых...	NaN	NaN	NaN	NaN
<b>2021-04-25 03:00:00</b>	Перистые нитевидные, иногда когтевидные, не ра...	Перистые нитевидные, иногда когтевидные, не ра...	NaN	NaN	NaN	NaN	Перистые (часто в виде полос) и перисто-слоист...	Перисто-кучевые или перисто-слоист...
<b>2015-04-21 12:00:00</b>	Перистых, перисто-кучевых или перисто-слоистых...	Перисто-кучевые одни или перисто-кучевые, сопр...	Перистые нитевидные, иногда когтевидные, не ра...	Перистых, перисто-кучевых или перисто-слоистых...	Перисто-кучевые или перисто-слоист...			

Out[13]:

	Cl_Cumls#V_Volochev	Cl_Cumls#Staritsa	Cl_Cumls#Kashyn	Cl_Cumls#Tver	Cl_Cumls#Klin	Cl_Cumls#Dmitrov	Cl_Cumls#Volokolamsk	Cl_Cumls#Moz
--	---------------------	-------------------	-----------------	---------------	---------------	------------------	----------------------	--------------

<b>2012-10-04 15:00:00</b>	Слоисто-кучевые, образовавшиеся не из кучевых.	Слоисто-кучевые, образовавшиеся не из кучевых.	Слоисто-кучевые, образовавшиеся не из кучевых.	Слоисто-кучевые, образовавшиеся не из кучевых.	Слоисто-кучевые, образовавшиеся не из кучевых.	Слоисто-кучевые, образовавшиеся не из кучевых.	Слоисто-кучевые, образовавшиеся не из кучевых.	Кучевые волокнистые
<b>2009-07-30 18:00:00</b>	NaN	Кучево-дождевые волокнистые (часто с наковальн...	NaN	NaN	Кучевые и слоисто-кучевые (но не слоисто-кучев...)	Кучевые и слоисто-кучевые (но не слоисто-кучев...)	Кучевые и слоисто-кучевые (но не слоисто-кучев...)	Слоисто-кучевые (но не слоисто-кучев...)
<b>2022-04-17 09:00:00</b>	NaN	NaN	NaN	Слоисто-кучевых, слоистых, кучевых или кучево-...	NaN	Слоисто-кучевых, слоистых, кучевых или кучево-...	NaN	Слоисто-образ...

	Cl_cumls_hi#V_Volochev	Cl_cumls_hi#Staritsa	Cl_cumls_hi#Kashyn	Cl_cumls_hi#Tver	Cl_cumls_hi#Klin	Cl_cumls_hi#Dmitrov	Cl_cumls_hi#Volokolamsk
2006-04-17 03:00:00	NaN	NaN	Высококучевые просвечающие, расположенные на...	NaN	NaN	NaN	NaN

2020-03-09 03:00:00	NaN	Высококучевых, высокослоистых или слоисто-дожд...	Высококучевых, высокослоистых или слоисто-дожд...	NaN	NaN	Высококучевые просвечающие, расположенные на...
---------------------	-----	---	---	-----	-----	---

2017-09-18 15:00:00	Высококучевых, высокослоистых или слоисто-дожд...	NaN	Высококучевые просвечающие, расположенные на...			
---------------------	---	-----	---	---	---	---

	Cl_viewd#V_Volochev	Cl_viewd#Staritsa	Cl_viewd#Kashyn	Cl_viewd#Tver	Cl_viewd#Klin	Cl_viewd#Dmitrov	Cl_viewd#Volokolamsk	Cl_viewd#M
2016-08-07 12:00:00	20–30%.	20–30%.	20–30%.	40%.	40%.	60%.	60%.	10% или ме...

2010-07-09 12:00:00	70 – 80%.	10% или менее, но не 0	40%.	50%.	40%.	20–30%.	40%.
---------------------	-----------	------------------------	------	------	------	---------	------

2018-07-06 21:00:00	90 или более, но не 100%	20–30%.	40%.	50%.	60%.	90 или более, но не 100%	NaN
---------------------	--------------------------	---------	------	------	------	--------------------------	-----

	Depo_diam_mm#V_Volochev	Depo_diam_mm#Staritsa	Depo_diam_mm#Kashyn	Depo_diam_mm#Tver	Depo_diam_mm#Klin	Depo_diam_mm#Dmitrov
2015-01-22 03:00:00	0.0	3.0	0.0	2.0	0.0	0.0

2019-12-24 03:00:00	0.0	0.0	0.0	0.0	0.0
---------------------	-----	-----	-----	-----	-----

2008-05-02 06:00:00	NaN	0.0	NaN	0.0	0.0
---------------------	-----	-----	-----	-----	-----

Out[13]:	Dew_point#V_Volochev	Dew_point#Staritsa	Dew_point#Kashyn	Dew_point#Tver	Dew_point#Klin	Dew_point#Dmitrov	Dew_point#Volokolamsk
2008-04-17 15:00:00	0.800000	0.0	-3.800000	0.9	-1.5	-4.4	-0.100000
2007-11-06 00:00:00	-7.785534	-8.1	-7.949811	-7.5	-7.5	-8.2	-7.600000



Out[13]:	Prcpttn_like#V_Volocheok	Prcpttn_like#Staritsa	Prcpttn_like#Kashyn	Prcpttn_like#Tver	Prcpttn_like#Klin	Prcpttn_like#Dmitrov	Prcpttn_like#Volokolamsk		
2005-03-17 06:00:00	NaN	NaN	NaN	NaN	NaN	NaN	NaN		
2016-02-01 09:00:00	Облака в целом образовывались или развивались.	Ливневый снег или ливневый дождь и снег.	NaN	NaN	NaN	NaN	NaN		
2007-02-22 18:00:00	NaN	NaN	NaN	NaN	NaN	NaN	NaN		
Out[13]:	Prcpttn_tdelt#V_Volocheok	Prcpttn_tdelt#Staritsa	Prcpttn_tdelt#Kashyn	Prcpttn_tdelt#Tver	Prcpttn_tdelt#Klin	Prcpttn_tdelt#Dmitrov	Prcpttn_tdelt#Volokolamsk		
2016-06-08 18:00:00	12.0	12.0	12.0	12.0	12.0	12.0	NaN		
2018-08-25 09:00:00	12.0	12.0	12.0	12.0	12.0	12.0	12.0		
2011-03-15 21:00:00	12.0	NaN	12.0	NaN	12.0	12.0	12.0		
Out[13]:	P_drift#V_Volocheok	P_drift#Staritsa	P_drift#Kashyn	P_drift#Tver	P_drift#Klin	P_drift#Dmitrov	P_drift#Volokolamsk	P_drift#Mozhaisk	P_drift#Kashyn
2018-03-28 12:00:00	0.0	0.1	0.1	0.184291	0.1	0.1	0.3	-0.3	0.1
2014-05-28 12:00:00	-0.1	0.0	-0.2	0.000000	-0.4	-0.1	-0.5	-0.2	-0.1
2021-06-04 00:00:00	0.6	0.3	0.5	1.538632	0.4	0.3	0.6	0.6	0.5



Out[13]:	Soil#V_Volochek	Soil#Staritsa	Soil#Kashyn	Soil#Tver	Soil#Klin	Soil#Dmitrov	Soil#Volokolamsk	Soil#Mozhaisk	Soil#N_Jerusalem	Soi
2015-05-25 21:00:00	Nan	Nan	Nan	Nan	Nan	Nan	Nan	Nan	Nan	Nan
2017-07-19 21:00:00	Nan	Nan	Nan	Nan	Nan	Nan	Nan	Nan	Nan	Nan
2015-10-26 09:00:00	Поверхность почвы сырая (вода застаивается на ... Поверхность почвы влажная.	Поверхность почвы влажная.	Поверхность почвы влажная.	Поверхность почвы влажная.	Поверхность почвы влажная.	Nan	Nan	Nan	Nan	Nan

Out[13]:	T#V_Volochek	T#Staritsa	T#Kashyn	T#Tver	T#Klin	T#Dmitrov	T#Volokolamsk	T#Mozhaisk	T#N_Jerusalem	T#Nemchinovka	T#Naro_Fomin
2010-05-19 00:00:00	13.6	13.9	13.0	13.1	11.1	13.1	13.5	13.5	14.0	13.797333	14
2018-09-06 15:00:00	24.6	25.0	25.4	25.2	25.5	24.8	25.3	25.1	25.6	25.100000	25
2008-04-25 00:00:00	6.0	6.0	7.6	6.2	6.0	7.6	7.3	6.4	4.9	6.274525	7

Out[13]:	T_max#V_Volochek	T_max#Staritsa	T_max#Kashyn	T_max#Tver	T_max#Klin	T_max#Dmitrov	T_max#Volokolamsk	T_max#Mozhaisk	T_max#N
2018-11-07 15:00:00	3.900000	3.9	2.200000	2.9	3.8	4.3	3.6	2.5	
2005-07-09 15:00:00	21.558103	21.4	21.018572	21.8	21.8	20.5	20.4	20.6	
2018-02-07 12:00:00	-8.900000	-7.5	-11.300000	-8.7	-10.4	-10.6	-9.9	-8.8	

Out[13]:	T_min#V_Volochek	T_min#Staritsa	T_min#Kashyn	T_min#Tver	T_min#Klin	T_min#Dmitrov	T_min#Volokolamsk	T_min#Mozhaisk	T_min#N_Jer
2010-10-04 09:00:00	-4.0	2.7	-0.2	-4.1	-0.1	-0.4	2.8	3.0	
2012-06-27 21:00:00	14.8	13.9	15.2	15.6	15.3	16.1	14.9	14.9	
2019-05-01 06:00:00	-2.5	-3.5	-3.0	-3.5	-3.5	-1.0	-1.9	-3.1	

Out[13]:	Visibility#V_Volochek	Visibility#Staritsa	Visibility#Kashyn	Visibility#Tver	Visibility#Klin	Visibility#Dmitrov	Visibility#Volokolamsk	Visibility#Moscow
2006-08-15 15:00:00	16.0	10.0	15.0	10.0	4.0	10.0		18.0
2017-04-29 18:00:00	10.0	10.0	50.0	10.0	10.0	NaN		10.0
2012-06-17 18:00:00	22.0	20.0	50.0	10.0	10.0	20.0		40.0
Out[13]:	Wind_dir#V_Volochek	Wind_dir#Staritsa	Wind_dir#Kashyn	Wind_dir#Tver	Wind_dir#Klin	Wind_dir#Dmitrov	Wind_dir#Volokolamsk	Wind_dir#Moscow
2020-03-07 15:00:00	ENE	E	E	E	SE	SE	SSE	
2007-12-25 15:00:00	SSW	SW	SW	SSW	SSW	SW	SW	
2020-11-19 21:00:00	SSW	SSW	SSW	SSW	SSW	SSW	S	
Out[13]:	Wind_dir360#V_Volochek	Wind_dir360#Staritsa	Wind_dir360#Kashyn	Wind_dir360#Tver	Wind_dir360#Klin	Wind_dir360#Dmitrov	Wind_dir360#Volokolamsk	Wind_dir360#Moscow
2022-02-09 12:00:00	202.5	225.0	225.0	202.5	180.0	225.0		
2005-11-01 21:00:00	360.0	202.5	225.0	225.0	360.0	270.0		
2008-04-28 06:00:00	Nan	360.0	Nan	360.0	360.0	157.5		

Out[13]:	Wind_dir6k#V_Volochek	Wind_dir6k#Staritsa	Wind_dir6k#Kashyn	Wind_dir6k#Tver	Wind_dir6k#Klin	Wind_dir6k#Dmitrov	Wind_dir6k#Volokola
<b>2013-02-24 12:00:00</b>	4500.0	4500.0	4500.0	4125.0	4500.0	4500.0	5000.0
<b>2018-06-27 12:00:00</b>	0.0	375.0	5625.0	5625.0	5625.0	0.0	5250.0

Out[13]:	Wind_speed#V_Volochev	Wind_speed#Staritsa	Wind_speed#Kashyn	Wind_speed#Tver	Wind_speed#Klin	Wind_speed#Dmitrov	Wind_speed#Vol
<b>2007-03-28 03:00:00</b>	0.0	0.0	2.0	2.0	0.0	0.0	
<b>2011-06-30 09:00:00</b>	1.0	2.0	2.0	2.0	2.0	2.0	
<b>2017-05-18 09:00:00</b>	1.0	3.0	3.0	2.0	3.0	3.0	

Out[13]: Wthr\_3h2#V\_Volochev Wthr\_3h2#Staritsa Wthr\_3h2#Kashyn Wthr\_3h2#Tver Wthr\_3h2#Klin Wthr\_3h2#Dmitrov Wthr\_3h2#Volokolamsk Wthr

2007-02-24 06:00:00	NaN	NaN	NaN	Облака покрывали более половины неба в течение...	NaN	NaN	NaN	NaN
2016-05-10 15:00:00	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2008-01-19 06:00:00	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Out[13]: Wthr\_curr#V\_Volochev Wthr\_curr#Staritsa Wthr\_curr#Kashyn Wthr\_curr#Tver Wthr\_curr#Klin Wthr\_curr#Dmitrov Wthr\_curr#Volokolamsk Wtl

2006-03-06 00:00:00	NaN	NaN	NaN	Снег непрерывный слабый в срок наблюдения.	Снег непрерывный слабый в срок наблюдения.	NaN	Снег непрерывный слабый в срок наблюдения.	NaN
2012-04-05 00:00:00	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2014-06-06 06:00:00	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

# 1. Теоретические основы

## 1.1. Описание задачи

Основная цель: получить статистически значимые изолированные наблюдения без ошибок и пропусков во всём поле метеостанций, по каждому параметру, на каждый заданный момент наблюдения, и на их основе смоделировать значения тех же параметров для Агробиостанции МГУ. Для реализации поставленной цели необходимо будет подобрать оптимальный способ восстановления данных для каждого параметра, который может быть положен в основу составления архивов

**метеонаблюдений для локации Агробиостанции МГУ в пос. Чашниково Солнечногорского района Московской области. ВАЖНО:**  
время наблюдений, фиксируемое в архивах погодных явлений, определяется по Гринвичу.

*На первом этапе необходимо отделить выбросы, имеющие физический смысл, от выбросов, являющихся ошибками ведения архивов. На выходе мы должны получить такие значения параметров, которые на самом деле являются реальными измерениями реальных погодных явлений (то есть имеют физический и метеорологический смысл) и потому должны быть исследованы и смоделированы, даже если они могут восприниматься как выбросы.*

Изначально нами была предпринята попытка анализа всех имеющихся данных одновременно и поиска ошибок на основе анализа выбросов за пределами трёх сигм от средней величины (то есть, вне пределов вероятности в 99,7%). Эта попытка оказалась неудачной.

- Во-первых, совокупность метеостанций составляет всего 12. При таком количестве любое аномальное значение будет приводить к сильному смещению распределения в свою сторону, и, как следствие, аномальное значение может оказаться внутри доверительного интервала, хотя и близко к его границе.
- Во-вторых, несмотря на географическую и природную близость, из-за особенностей метеорологических явлений и иногда их локального характера, запределенные значения не всегда могут свидетельствовать об ошибках.
- В-третьих, при наличии нескольких пропущенных значений совокупность еще более сужается, ошибочное значение может трактоваться, как нормальное и также находится внутри доверительного интервала.

Поэтому ниже мы будем принимать во внимание, среди прочего:

- отклонение индивидуальных значений от средней между станциями в пределах от не только 3 сигм, но и менее; при этом подсчёт средней и показателей вариации не будет включать само проверяемое значение,
- нахождение индивидуального значения в пределах заданного доверительного интервала для гипотетического нормального распределения,
- отклонение индивидуальных значений от ближайших значений наблюдения по времени (до и после),
- климатические и сезонные нормы (например, <https://ru.wikipedia.org/> - климат Московской области, климат Тверской области). Каждый параметр (группу родственных параметров) будем рассматривать отдельно. Заполнение пропусков будем выполнять отдельно и после удаления ошибок (кроме случаев, когда ошибки возможно исправить на основе вычисления взаимозависимых значений).

Для более детального анализа будем учитывать временные границы сезонов. Исходя из данных о климате Московской области, временные границы сезонов определены следующим образом.

- Зима (ниже 0°): в среднем длится с 5 ноября по 4 апреля.
- Весна (от 0° до +10°): в среднем длится с 5 апреля по 18 мая.
- Лето (выше +10°): в среднем длится с 19 мая по 14-15 сентября.
- Осень (от +10° до 0°): в среднем длится с 14-15 сентября по 4 ноября.

Следует иметь в виду следующий порядок организации метеонаблюдений, принятый в Российской Федерации:

- наблюдения на всех метеорологических станциях проводят синхронно восемь раз в сутки в 00, 03, 06, 09, 12, 15, 18 и 21 ч по гринвичскому времени;
- во все сроки измеряют температуру воздуха и почвы, влажность воздуха, скорость ветра и его направление, метеорологическую дальность видимости, атмосферное давление, определяют характеристики облачности;
  - другие величины, не имеющие хорошо выраженного суточного хода, определяют не во все сроки и даже между сроками. Так, состояние поверхности почвы и осадки определяют два раза в сутки в сроки, ближайшие к 8 и 20 ч местного времени пояса, в котором расположена станция. Высоту снежного покрова, глубину промерзания почвы измеряют один раз в утренний срок, ближайший к 08 ч декретного времени данного пояса. Снегомерные съемки производят один раз в 10 дней, а весной перед началом и в период таяния снега – один раз в 5 дней. Испарение измеряют один раз в 5 дней, влажность почвы – один раз в 10 дней (на 8-й день 30 декады). Ленты термографа, гигрографа, барографа меняют в срок, ближайший к 13 ч, а плювиографа – к 20 ч местного времени.
- за начало суток на каждой станции принимают единый срок, ближайший к 20 ч, а за первый срок наблюдений – срок, ближайший к 23 ч местного времени;
- так как произвести измерения всеми приборами точно в срок наблюдений нельзя, принято при восьмисрочных наблюдениях температуру и влажность воздуха измерять за 10 мин, а давление воздуха – за 2 мин до срочного часа. Все остальные измерения начинают за 30 мин до срока и заканчивают после срока. Общая продолжительность наблюдений составляет 30 – 40 мин.

Однако используемые нами архивы метеорологических наблюдений, полученные с интернет ресурса [www.rp5.ru](http://www.rp5.ru) используют местное время метеорологических наблюдений.

## **1.2. Общий алгоритм поиска ошибок, их исправления, а также восстановления пропущенных значений**

*Основная цель при работе с выбросами: отделить выбросы имеющие физический смысл от ошибок. По своей структуре наши данные могут быть описаны в следующих измерениях:*

1. Географическое пространство метеостанций (поле):

- географическое положение (координаты и высота над уровнем моря),
- производные от них расстояния и начальные азимуты.

2. Временной ряд:

- однонаправленный, приведённой к частоте дискретизации в 1 наблюдение за 3 часа,
- определяет суточные колебания и сезонные колебания.

3. Набор параметров с различной степенью взаимной зависимости.

- дискретный ряд параметров,
- взаимозависимые параметры, которые могут служить подтверждением или опровержением для значений в увязке с временным рядом и географическими параметрами.

Для выявления ошибок необходимо:

1. Определить диапазон допустимых значений для каждого исследуемого параметра и границы вероятности отклонения от среднего наблюдаемого значения в географическом пространстве и во времени. Для каждого параметра такие границы следует устанавливать индивидуально.

2. Найти выбросы:

- в поле метеостанций на момент наблюдения
- в окне временного ряда (с учётом исследуемого параметра, его суточных и сезонных колебаний и с учётом климатических норм, если таковые существуют).

1. Если выброс является одновременно и выбросом в поле метеостанций, и в окне временного ряда, и тем более, если он не совпадает с расчётными значениями из других параметров (если зависимость между ними существует и взаимозависимые параметры не содержат ошибок), то этот выброс следует расценивать как ошибку.

Мы исходим из того, что все ошибки в архивах являются ошибками ввода данных и в большинстве случаев являются одной или одновременно несколькими следующими ошибками:

- ошибкой в разряде,
- ошибкой в знаке,
- ошибкой лишней или недостающей цифры,
- ошибкой неверной цифры (чаще всего схожей по начертанию).

Все ошибки такого рода буду приводить к очень значительными выбросам.

Для исправления ошибок возможен один или несколько способов:

1. Привести значение в соответствие с допустимым интервалом значений для поля метеостанций,
2. Привести значение в соответствие с допустимым интервалом значений для временного ряда,
3. Если возможно, произвести расчёт корректного значения, исходя из корректных значений других параметров,
4. Логически исправить ошибку (вручную, если количество ошибок позволяет это сделать),
5. Удалить ошибочное значение и заменить его прогнозом, исходя из выбранной модели экстраполяции данных.

Необходимо подобрать модель экстраполяции данных, критерием здесь могут служить:

- метрики качества,
- для хороших метрик качества - соотношение трудоемкости использования других моделей и потенциального дальнейшего улучшения метрик.

Далее нужно произвести моделирование значений для:

- пропусков в архивах,
- условной метеостанции Чашниково.

В конечном итоге полученные значения будут записаны в архивы.

## 1.3. Подход к моделированию отсутствующих значений (пропущенных и намеренно удалённых)

### 1.3.1. Метеорологические показатели, подлежащие моделированию

Конечное назначение данной модели - воссоздание тех данных, которые будут критичными для анализа снегового покрова. Поэтому была составлена иерархия данных по степени критичности для последующего исследования снегового покрова.

Сортировка параметров приводится *по убыванию степени критичности*.

1. Необходимые данные о предмете исследования:

- Snow\_height (sss) Высота снежного покрова, см;

- Soil\_cover (E') Состояние поверхности почвы со снегом или измеримым ледяным покровом.

1. Существенные данные для анализа предмета исследования:

- T(T) Температура воздуха на высоте 2 м над поверхностью земли, градусов Цельсия;
- P\_sea (P) Атмосферное давление, приведённое к среднему уровню моря, мм рт. столба;
- Humid (U) Относительная влажность воздуха на высоте 2 м над поверхностью земли, %;
- Wind\_dir (DD) Направление ветра на высоте 10-12 м над поверхностью земли, усреднённое за 10 минут непосредственно перед наблюдением, румбы;
- Wind\_speed, (Ff) - Скорость ветра на высоте 10-12 м над поверхностью земли, усреднённая за 10 минут непосредственно перед наблюдением, м/с.;
- Prcpttn (RRR) Количество выпавших осадков, мм;
- Prcpttn\_tdelt (tR) Период времени, за который выпало указанное количество осадков, ч.;

1. Желательные данные:

- Wthr\_curr (WW) Текущая погода, сообщаемая метеостанцией - *в части описания осадков только!*
- Wthr\_3h (W1) Прошедшая погода между сроками наблюдения 1 - *в части описания осадков только!*
- Wthr\_3h2 (W2) Прошедшая погода между сроками наблюдения 2 - *в части описания осадков только!*

1. Некритичные данные:

- Soil (E) Состояние поверхности почвы без снега или измеримого ледяного покрова (*желательно для смежных исследований*);
- Soil\_T (Tg) Минимальная температура поверхности почвы за ночь, градусов Цельсия (*желательно для смежных исследований*);
- Gusts (ff10) Максимальная скорость порыва ветра на высоте 10-12 м над поверхностью земли, за 10 минут непосредственно перед наблюдением, м/с.;
- Gusts\_3h (ff3) Максимальная скорость порыва ветра на высоте 10-12 м над поверхностью земли, за период между моментами наблюдения 3 часа, м/с.;
- Visibility (VV) Горизонтальная дальность видимости, км.;
- Cloudness (N) Общая облачность, %;
- Cl\_bottom (H) Высота основания самых низких облаков, м;
- Cl\_Cumls (Cl) Слоисто-кучевые, слоистые, кучевые и кучево-дождевые облака;
- Cl\_viewed (Nh) Количество всех наблюдающихся облаков Cl, или при отсутствии облаков Cl, количество всех наблюдающихся облаков Cm, %;

- Cl\_cumuls\_hi (Cm) Высоко-кучевые, высоко-слоистые и слоисто-дождевые облака;
- Cl\_cirrus (Ch) Перистые, перисто-кучевые и перисто-слоистые облака.

1. Вычисляемые и зависимые данные:

- P\_station (Po) Атмосферное давление на уровне станции, мм рт. столба;
- P\_drift (Pa) Барическая тенденция: изменение атмосферного давления за последние 3 часа, мм рт. столба;
- T\_min (Tn) Минимальная температура воздуха за прошедший период не более 12 часов, градусов Цельсия;
- T\_max (Tx) Максимальная температура воздуха за прошедший период не более 12 часов, градусов Цельсия;
- Dew\_poin (Td) Температура точки росы на высоте 2 м над поверхностью земли, градусов Цельсия.

### **1.3.2. Последовательность моделирования метеорологических показателей.**

Не смотря на заданную последовательность значимости показателей для предмета исследования, последовательность моделирования будет несколько иной.

1. Мы начнём с основополагающих групп показателей: температура, давление, влажность. Связанные с ними вычисляемые показатели будем рассчитывать сразу же. Эти данные содержат менее всего пропусков, хорошо коррелируются между собой и являются наиболее значимыми для анализа погодных явлений, по ним можно ориентироваться при оценке корректности значений других показателей.
2. Далее перейдём к моделированию хорошо коррелирующихся численных значений состояния почвы и высоты снегового покрова.
3. После чего перейдём к моделированию других показателей с учётом их зависимости между собой и значимости для предмета исследования (снегового покрова).

В первую очередь будут исследованы и смоделированы числовые непрерывные показатели, имеющие между собой лучшую корреляцию. И лишь после этого - дискретные и категориальные. Более того, для моделирования значений нам могут понадобиться показатели, отнесенные к группе "некритичных данных", в этом случае их придется исследовать и обрабатывать в первую очередь.

### **1.3.3. Выяснение параметров (фич) для модели**

Архивные данные содержат большое количество пропусков и ошибок. Как было показано в первой тетради, пропуски по некоторым показателями наблюдения зачастую занимают большие периоды по временной оси. Некоторые метеостанции не фиксировали отдельные показатели годами. В связи с этим попытка моделирования данных, опираясь на значения по временной оси не

представляется перспективной. В ряде случаев (где проущено очень много моментов наблюдения подряд) они просто не дадут результата. Надо также учитывать, что сезонные и суточные колебания, хоть и подобны, но практически никогда не дублируются. Поэтому предсказания на их основе будут иметь достаточно низкую точность. Гораздо более точные предсказания можно получить из наблюдений в различных, относительно близко расположенных, точках, объединённых одним моментом времени.

**A.** Для каждого зафиксированного момента времени  $t$  для каждого наблюдаемого параметра  $Z$  мы должны найти такую функцию  $f$  для совокупности  $station_n$ , которая с большой долей вероятности даст значение показателя  $(Z(t))$  для условной метеостанции  $hashnikovo$ . То есть:  $hashnikovo(Z(t)) = f(station_1(Z(t)), \dots, station_n(Z(t)))$ . Для каждого метеорологического показателя эта зависимость может быть разной, поэтому у нас может быть столько моделей, сколько и показателей метеонаблюдений.

Фичами для нас являются не данные архива, а данные о метеостанциях. А это - данные о географическом положении метеостанций. А географические параметры (высота над уровнем моря и положение метеостанций в пространстве относительно друг друга и метеостанции  $hashnikovo$ ) - суть величины постоянные для каждой  $station_n$ , то есть, их влияние как факторов на модель для каждого данного параметра  $Z$  будет тоже постоянным.

Временной ряд показывает только итерации наблюдений. Чем длиннее временной ряд, тем больше вариаций значений параметров мы имеем для поиска искомой функции.

В идеале, мы можем вывести для каждого наблюдаемого значения погодного явления функцию для определения значения этого явления в заданной точке по значениям на  $n$  заданных метеостанциях. Эта функция, скорее всего, будет рабочей в рамках одной климатической зоны. Коррелирующие между собой метеорологические явления проявятся в подобии функций поиска их значений для Чашниково. Но на качество предсказываемой величины корреляция параметров погоды не повлияет, потому что для каждого параметра будет искааться своя функция.

Тем не менее, используя жестко коррелирующие между собой метеорологические показатели, мы можем облегчить себе задачу и сократить количество показателей, для которых нужно составлять индивидуальную модель. Если сейчас их 29, то мы можем взять меньшее число, и уменьшить количество индивидуальных моделей. Мы знаем, что некоторые показатели погодных явлений имеют физическую зависимость между собой. Следовательно, зная (или выявив) уравнения для такой зависимости, мы можем в искомой точке воссоздать их значения, не прибегая к моделированию зависимостей между метеостанциями.

### Ограничения.

1. Для создания моделей нам нельзя брать разные метеостанции для обучения, валидации и предсказания. Нам нужно брать разные данные по моментам наблюдения.

2. Важно, чтобы между или в непосредственной близости рядом с метеостанциями и искомой точкой не было зон с другим микроклиматом (больших водоёмов, высоких неровностей рельефа, зон плотной городской застройки и климатической "грелки" вроде Москвы). Именно по этой причине мы не можем взять для расчётов показания метеостанций, расположенных в черте Москвы.

***При условии достаточно высокой корреляции между данными метеостанций, для каждой модели параметра наблюдения определяющими фичами будут географическое положение метеостанций и их результирующая дальность от искомой точки.***

**Б.** Пространственная экстраполяция может оказаться не всегда возможной. Как показано в первой тетради, целый ряд показателей имеет недостаточную корреляцию между метеостанциями. В этом случае придётся использовать другой подход. Для плохо географически коррелируемых показателей потребуется выбирать фичи исходя из физического смысла показателя и его зависимостей от других показателей. Отбор фич здесь надо будет делать индивидуально для каждого такого показателя. Принципиальным будет то, что для поиска целевой величины нам нужно будет использовать показатели, предварительно очищенные от ошибок и пропусков, чтобы получить максимально чистые предсказания целевых величин. Этот момент определяет последовательность обработки и моделирования метеорологических явлений.

***При условии недостаточно высокой корреляции между метеостанции, для каждого показателя придется создавать свою модель с учётом физических особенностей явления, которое он выражает. Набор параметров для модели в этом случае придётся определять индивидуально.***

### **1.3.4. Модели пространственной экстраполяции геостатистических данных**

Основная идея пространственной экстраполяции заключена в моделях IDW (от английского *Inverse distance weighting* - взвешивание по обратным расстояниям).

Основная идея заключается в предположении, что значения \$x\$ в точке \$i\$ (\$x\_i\$) и \$x\$ в точке \$i + h\$ (\$x\_{i+h}\$) тем ближе между собой, чем меньше расстояние \$h\$ между ними. Функция зависимости индивидуального значения показателя от значений того же показателя в других географических точках таким образом имеет вид:

$$Z_u = \frac{\sum_i^N w_i * Z(i)}{\sum_i^N w_i}$$

где \$Z\_u\$ - предсказываемое значение показателя в точке с отсутствием наблюдения, находящейся среди \$N\$ наблюдаемых точек.

Значения показателей в наблюдаемых точках  $Z(i)$  взвешиваются и усредняются. Веса рассчитываются как:

$$w_i = \frac{1}{\|\overrightarrow{u_x_i}\|}$$

где  $u$  - точка с отсутствием наблюдаемого значения, а  $x_i$  - точка с имеющимися значениями наблюдения.

Таким образом,  $\|\overrightarrow{u_x_i}\|$  - это нормированный вектор между двумя точками - Евклидово расстояние в пространстве координат (которое отнюдь не ограничиваются именно  $\mathbb{R}^2$ )

Так как более близкие точки имеют большее влияние на исковую точку, то используется обратная пропорциональность расстояниям. Отсюда и взвешивание по обратным расстояниям (IDW).

Распространённым подходом к созданию таких моделей является кригинг, основанный на вариограммах - реализован в библиотечных функциях SciKit GStat, GTools. Мы так же можем использовать взвешивание по обратным степеням расстояний (согласно закону обратных квадратов, сила воздействия многих физических явлений изменяется обратнопропорционально квадрату расстояния от источника), тогда

$$w_i = \frac{1}{\|\overrightarrow{u_x_i}\|^p}$$

где  $p$  - степень, в которую возводятся веса.

### 1.3.5. Иные модели.

В тех случаях, когда модели пространственной экстраполяции окажутся неприменимыми, конкретные модели предсказания целевых значений придётся выбирать индивидуально. В зависимости от корреляции выанных фич и целевой величиной это могут быть регрессоры, деревья, нейросети, классификаторы и кластеризаторы, ансамбли моделей. Если для пространственной экстраполяции значения параметров имеют общую с моделируемым значением структуру распределения и диапазон значений, то здесь надо будет иметь в виду необходимость предварительной нормализации входных данных. Дополнительно необходимо оговорить, что не все показатели имеют числовое выражение. Часть содержащихся в архивах данных являются категориальными.

## 2. Общие функции для поиска ошибок, моделирования и исправления значений параметров

## 2.1. Функция рассчёта средней, взвешенной по обратной степени расстояний

```
In [14]: def inverse_distance_avg(row_, param_, station_='Rfrnce_point', df_dists_=df_station_dists, power_=2):
    """
    Расчитывает среднюю, взвешенную по обратным степеням расстояния, исключая данную метеостанцию.
    РАБОТАЕТ ТОЛЬКО ДЛЯ АРХИВА МЕТЕОНАБЛЮДЕНИЙ ПО ПАРАМЕТРАМ!
    РАБОТАЕТ ТОЛЬКО ДЛЯ ПРОСТРАНСТВА МЕТЕОСТАНЦИЙ!
    Принимает:
    - серия значений из которых необходимо рассчитать среднюю, взвешенную по обратным степеням расстояния;
    - название параметра, для которого рассчитывается средняя;
    - название метеостанции для которой рассчитывается средняя (её значение исключается из подсчёта средней!)
        default='Rfrnce_point' - геометрический центр поля метеостанций;
    - df_dists_ DF с расстояниями между метеостанциями default=df_station_dists;
    - power_ степень для вычисления обратных весов default=2
    Возвращает:
    - значение средней, взвешенной по обратным степеням расстояния от точки, заданной параметром stations_
        (вес w = 1000000(d**power_), во избежание слишком малых значений). Во избежание представления 1 в виде
        десятичной дроби с 99..998 после запятой, результат округляется до 12 знака после запятой.
    ПРИМЕЧАНИЕ: чтобы вычислить общую среднюю для всего поля метеостанций без исключений необходимо использовать значение
        staton_ по умолчанию - 'Rfrnce_pint'
    """
    # удаляем из серии значений row_ все значения не являющиеся значениями параметров (символ # в названии индекса -
    # признак того, что этому индексу соответствует значение параметра)
    counter_ = 0 # счётчик для индекса
    ##
    # print(row_.index.tolist())
    for item_ in (row_.index.tolist()): # название индекса по индексу серии row_
        if '#' not in item_: # если в названии индекса нет символа '#'
            row_ = row_.drop(row_.index[counter_]) # удаляем из серии элемент (не являющийся значением параметра)
        else:
            counter_ += 1 # в противном случае увеличиваем счётчик на 1

    list_param_per_inv_dists_ = [] # Список значений параметров, умноженных на обратную степень расстояний
    weight_sum_ = 0 # сумма весов для вычисления средней

    # объединяем в zip названия df станций (индекс серии) и значений параметра им соответствующих (собственно серия)
    # имя df станции и значение параметра в zip
    for name_st_, param_val_ in zip(row_.index, row_):
        # к каждому названию df станции применяем уменьшение слева на длинну строки с названием параметра + 1 знак ('#')
        # так как название столбцов состоит из названия параметра + '#' + название станции
        name_st_ = name_st_[len(param_) + 1 :]
```

```

# расстояние равно значению строке в df_station_dists где station равно станции, для которой вычисляется средняя
# а name_st - столбцу в df_station_dists
## 
#     print(station_, name_st_)
distance_ = df_station_dists.loc[(df_station_dists.station == station_), name_st_].values[0]

# если расстояние не равно 0 (чтобы исключить искомую станцию) и текущее значение не является NaN
if (distance_ != 0) and (not pd.isna(param_val_)):
    # добавляем в список значение параметра для этой станции умноженное на обратную степень расстояния
    # умножим его на 1000 в степени power_, чтобы исключить чрезвычайно малые значения в делителе при расчёте средней
    # множитель 1000 в степени power_ нужен чтобы компенсировать увеличение делителя при увеличении степени
    list_param_per_inv_dists_.append(((1000/distance_)**power_)*param_val_)
    # увеличиваем сумму весов на текущую обратную степень расстояния, умноженную также 1000 в степени power_
    weight_sum_ += (1000/distance_)**power_

else:
    pass

# результат: сумма значений параметров (кроме искомой станции), умноженных на обратные степени расстояний
# до соответствующей станции от искомой станции и, делённая на сумму обратных степеней расстояний.
result_ = np.around(np.nanmean(list_param_per_inv_dists_)/weight_sum_, 12) if weight_sum_ != 0 else np.nan
# Сумма без NaN

return result_

```

## 2.2. Функция пространственной экстраполяции данных методом кригинга

Сначала определим необходимые для функции значения

Для нашего случая, погрешность в исчислении расстояний между угловыми координатами и прямоугольными (в соответствующей проекции) крайне незначительна. Поэтому, для вариаграммы будем использовать углы широт и долгот.

```
In [15]: # Определим df с полным списком координат всех точек: Нужно для данной функции
df_coords_full = df_station_dists[["LoE", "LaN"]]
df_coords_full.index = df_station_dists.station
# df_coords_full = df_stations_lin_coords[["lin_hor", "lin_ver"]]
# df_coords_full.index = df_stations_lin_coords.station
```

```
In [16]: # Вывод на экран через print() сильно замедляет работу. Чтобы временно заблокировать вывод на экран предупреждений типа:  
# 'Warning: for %d locations, not enough neighbors were found within the range.' % self.no_points_error  
# определим отдельный класс с пустым выводом и направим на него вывод функции  
class NullIO(StringIO): # Класс нулевого вывода  
    def write(self, txt):  
        pass
```

```
In [17]: # сохраним стандартные настройки вывода  
real_stdout = sys.stdout # сделаем backup стандартного вывода
```

```
In [18]: def kriging_extrapolation(row_, df_coords_=df_coords_full):  
    """  
    Расчитывает значения для пространственной экстраполяции данных с помощью модели обычного кригинга.  
    РАБОТАЕТ ТОЛЬКО ДЛЯ АРХИВА МЕТЕОНАБЛЮДЕНИЙ ПО ПАРАМЕТРАМ!  
    РАБОТАЕТ ТОЛЬКО ДЛЯ ПРОСТРАНСТВА МЕТЕОСТАНЦИЙ!  
  
    ВНИМАНИЕ! Данная функция привязана к структуре данных в df_stations (и производных от него) и dict_df_parameters!  
    Предполагается, что перечень метеостанций в df_stations идёт в той же последовательности, что и в row_!  
  
    ПРИМЕЧАНИЕ: Параметры вариограммы зафиксированы (estimator='matheron', model='spherical', dist_func='euclidean',  
    bin_func='ward', maxlag=0.99999, n_lags=4, normalize=False, use_nugget=False, samples=len(vals_v), fit_method='trf',)  
  
    Принимает:  
    - row_: серию значений из которых необходимо произвести пространственную экстраполяцию;  
    - df_coords_ (по умолчанию df_coords_full) датафрейм с координатами метеостанций, сообразно индексу row_.  
    Возвращает:  
    - массив предиктов для всех метеостанций в row_  
    ЕСЛИ В ПРОЦЕССЕ ОПРЕДЕЛЕНИЯ ВАРИОГРАММЫ ВОЗНИКАЮТ ОШИБКИ  
    (КАК ПРАВИЛО ИЗ-ЗА МАЛОГО КОЛИЧЕСТВА ЗНАЧЕНИЙ В ПОЛЕ МЕТЕОСТАНЦИЙ), ТО ФУНКЦИЯ ПРЕРЫВАЕТСЯ БЕЗ ВОЗВРАЩЕНИЯ ЗНАЧЕНИЙ  
    """  
  
    # Для построения вариограммы нужны только точки, для которых есть значения:  
    # - получаем массив координат  
    # - получаем массив значений  
    # Берём координаты только соответствующие ряду значений и без NaN:  
    coords_v_ = np.array(df_coords_full)[:len(row_)][~np.isnan(row_)]  
    vals_v_ = np.array(row_.dropna())  
  
    try:  
        # Определяем вариограмму  
        V_ = skg.Variogram(coordinates=coords_v_,
```

```
        values=vals_v_,
        estimator='matheron',
        model='spherical',
        dist_func='euclidean',
        bin_func='ward',
        maxlag=0.99999, # Используем всю матрицу расстояний
        n_lags=4, # количество бин не должно быть очень большим, чтобы не уменьшать effective distance
        normalize=False,
        use_nugget=False,
        samples=len(vals_v),
        fit_method='trf'
    )

except ValueError: # Уберём количество сэмплов из определения вариограммы (когда количество сэмплов слишком велико)
###
##    print('\nВозникла ошибка ValueError, строим вариограмму без учета количества сэмплов.')
###
try:
    V_ = skg.Variogram(coordinates=coords_v_,
                         values=vals_v_,
                         estimator='matheron',
                         model='spherical',
                         dist_func='euclidean',
                         bin_func='ward',
                         maxlag=0.99999, # Используем всю матрицу расстояний
                         n_lags=4,
                         normalize=False,
                         use_nugget=False,
                         fit_method='trf'
    );
except:
###
##    print('\nНовая ошибка вариограммы. Выход из функции.')
###
    return
except:
###
##    print('\nИная ошибка вариограммы. Выход из функции.')
###
    return

# Определяем модель кrigинга
model_ok_ = skg.OldinaryKriging(V_, min_points=1, max_points=14, mode='exact');
```

```

# координаты точки предикта:
predict_coords_ = np.array(df_coords_full)[:len(row_)] # здесь берём все координаты, кроме Reference_point

sys.stdout = NullIO() # определим вывод print() в класс нулевого вывода

# Получаем массив предиктов для всех точек
arr_predict_ = model_ok_.transform(predict_coords_[:,0], predict_coords_[:,1]);

sys.stdout = real_stdout # восстановим стандартный вывод

return arr_predict_ # Возвращаем массив предиктов для всех точек

```

## 2.3. Функция рассчёта пределов n сигм относительно средней (как опция: взвешенной по степени обратных расстояний)

In [19]:

```

# функция вычисления пределов n сигм для ряда значений row_ в DF
def sigma_n_limits(row_, n_sigma_ = 3, IDW_ = False, param_ = None, station_='Rfrnce_point', inclusive_= True):
    """ ТОЛЬКО ДЛЯ АРХИВА ПАРАМЕТРОВ
    Вычисляет пределы в n сигм от средней по обратным квадратам или средней арифметической для серии значений.
    Принимает:
    - row_ значения из ряда DF (default = None),
    - n_sigma_ количество n на которое умножается среднеквадратическое отклонение для вычисления доверительного интервала
    (default=3),
    - IDW_ (boolean) использовать ли усреднение по обратным квадратам расстояний (default=False),
    - param_ название параметра для вычисления IDW (default=None). Обязателен для IDW_=True, а так же
        при установки sigma_inclusive_ = False, в противном случае вернёт ошибку,
    - station_ - название метеостанции, для которой вычисляются пределы (значение для неё исключается из подсчёта
        средней и сигм (default='Rfrnce_point' - геометрический центр поля метеостанций),
    - inclusive_ (при IDW_=True имеет смысл только при указании station_, отличной от 'Rfrnce_point') -
        включать ли значение для данной метеостанции в расчёт средней,
        при station_='Rfrnce_point' значение для данной метеостанции будет включено
        в подсчёт средней независимо от значения inclusive_ (boolean, deafault=True).
    Возвращает:
    - кортеж - нижний и верхний порог доверительного интервала.
    """
    # Во избежание разночтений установим в True включение всех значений в подсчёт средней и сигмы
    if station_ == 'Rfrnce_point':
        inclusive_ = True
        # Rfrnce_point не входит в число метеостанций, а это значит, что при его указании в расчёте обеих величин
        # должны участвовать все реальные метеостанции. Так как для Rfrnce_point значения не определены,

```

```

# при расчётах средней и сигмы NaN будет выброшен, и сам Rfrnce_point в подсчёт не войдёт,
# но войдут все без исключения метеостанции, для которых значение не равно NaN.

# Только если индекс row_ не является datetime64
if row_.index.inferred_type != "datetime64":
    # удаляем из серии значений row_ все значения не являющиеся значениями параметров (символ # в названии индекса -
    # признак того, что этому индексу соответствует значение параметра)
    counter_ = 0 # счётчик для индекса
    for item_ in (row_.index.tolist()): # название индекса по индексу серии row_
        if '#' not in item_: # если в названии индекса нет символа '#'
            row_ = row_.drop(row_.index[counter_]) # удаляем из серии элемент (не являющийся значением параметра)
        else:
            counter_ += 1 # в противном случае увеличиваем счётчик на 1

    # Ниже перебираем все возможные сочетания булевых значений:
    # использовать IDV, включить в подсчёт средней данную метеостанцию
    # и включить в подсчёт сигмы данную метеостанцию
    if IDW_ and inclusive_ : # Если IDW_=True, использовать данную станцию для подсчёта
        # средней и сигмы
        # Находим среднюю по обратным квадратам расстояния относительно центра поля метеостанций
        # при station_='Rfrnce_point'
        mean_value_ = inverse_distance_avg(row_, param_, station_ ='Rfrnce_point', df_dists_ = df_station_dists, power_ =2)
        # сигма по серии, игнорируя nan, степень свободы=1 (где количество нeNaN больше 1)
        std_value_ = np.nanstd(row_, ddof=1) if len(row_.dropna()) > 1 else np.nan
    elif IDW_ and not inclusive_:
        # Находим среднюю по обратным квадратам расстояния относительно данной метеостанций
        mean_value_ = inverse_distance_avg(row_, param_, station_ , df_dists_ = df_station_dists, power_ =2)
        row_ = row_.drop(labels=param_ + '#' + station_) # удаляем значение данной метеостанции из ряда
        std_value_ = np.nanstd(row_, ddof=1) if len(row_.dropna()) > 1 else np.nan
    elif not IDW_ and inclusive_:
        mean_value_ = np.nanmean(row_) if len(row_.dropna()) > 0 else np.nan # средняя по серии, игнорируя nan
        # сигма по серии, игнорируя nan, степень свободы=1 (где количество нeNaN больше 1)
        std_value_ = np.nanstd(row_, ddof=1) if len(row_.dropna()) > 1 else np.nan
    elif not IDW_ and not inclusive_:
        row_ = row_.drop(labels=param_ + '#' + station_) # удаляем значение данной метеостанции из ряда
        mean_value_ = np.nanmean(row_) if len(row_.dropna()) > 0 else np.nan # средняя по серии, игнорируя nan
        std_value_ = np.nanstd(row_, ddof=1) if len(row_.dropna()) > 1 else np.nan

    upper_level_ = mean_value_ + n_sigma_ * std_value_ # верхний порог п сигм
    lower_level_ = mean_value_ - n_sigma_ * std_value_ # нижний порог п сигм

return (lower_level_ , upper_level_ )

```

## 2.4. Функция вычисления вероятностного интервала нормального распределения относительно средней (как опция: взвешенной по степени обратных расстояний)

```
In [20]: # функция оценки границ вероятности для нормального распределения относительно средней, или средней,
# взвешенной по обратным квадратам расстояния с учетом среднеквадратического отклонения для данного ряда значений.
def norm_probability_limits(
    row_, confidence_=0.95, IDW_=False, param_=None, station_='Rfrnce_point', inclusive_=True):
    """ ТОЛЬКО ДЛЯ АРХИВА ПАРАМЕТРОВ
    Вычисляет пределы в n сигм от средней по обратным квадратам или средней арифметической для серии значений.
    Принимает:
        - row_ - значения из ряда DF (default = None),
        - confidence_ - значение вероятности для оценки границ (float в диапазоне от 0 до 1) (default=0.95),
        - IDW_ (boolean) использовать ли усреднение по обратным квадратам расстояний (default=False),
        - param_ - название параметра для вычисления IDW (default=None). Обязателен для IDW_=True, а так же
            при установки sigma_inclusive_ = False, в противном случае вернёт ошибку,
        - station_ - название метеостанции, для которой вычисляются пределы (значение для неё исключается из подсчёта
            средней и сигм (default='Rfrnce_point' - геометрический центр поля метеостанций),
        - inclusive_ (при IDW_=True имеет смысл только при указании station_, отличной от 'Rfrnce_point') -
            включать ли значение для данной метеостанции в расчёт средней,
            при station_='Rfrnce_point' значение для данной метеостанции будет включено
            в подсчёт средней независимо от значения inclusive_ (boolean, deafault=True).
    Возвращает:
        - кортеж - нижний и верхний порог доверительного интервала.
    """
    # Во избежание разночтений установим в True включение всех значений в подсчёт средней и сигмы
    if station_ == 'Rfrnce_point':
        # Rfrnce_point не входит в число метеостанций, а это значит, что при его указании в расчёте обеих величин
        # должны участвовать все реальные метеостанции. Так как для Rfrnce_point значения не определены,
        # при расчётах средней и сигма NaN будет выброшен, и сам Rfrnce_point в подсчёте не войдёт,
        # но войдут все без исключения метеостанции, для которых значение не равно NaN.
        inclusive_ = True

    # Только если индекс row_ не является datetime64
    if row_.index.inferred_type != "datetime64":
        # удаляем из серии значений row_ все значения не являющиеся значениями параметров
        # (символ # в названии индекса - признак того, что этому индексу соответствует значение параметра)
        counter_ = 0 # счётчик для индекса
        for item_ in (row_.index.tolist()): # название индекса по индексу серии row_
            if '#' not in item_: # если в названии индекса нет символа '#'
```

```

        row_ = row_.drop(row_.index[counter_]) # удаляем из серии элемент (не являющийся значением параметра)
    else:
        counter_ += 1 # в противном случае увеличиваем счётчик на 1

    # Ниже перебираем все возможные сочетания булевых значений:
    # использовать IDV, включить в подсчёт средней данную метеостанцию
    # и включить в подсчёт сигмы данную метеостанцию
    if IDW_ and inclusive_ : # Если IDW_=True, использовать данную станцию для подсчёта
        # средней и сигмы
        # Находим среднюю по обратным квадратам расстояния относительно центра поля метеостанций
        # при station_='Rfrnce_point'
        mean_value_ = inverse_distance_avg(row_, param_, station_='Rfrnce_point',
                                            df_dists_=df_station_dists, power_=2)
        # сигма по серии, игнорируя nan, степень свободы=1 (где количество нeNaN больше 1)
        std_value_ = np.nanstd(row_, ddof=1) if len(row_.dropna()) > 1 else np.nan
    elif IDW_ and not inclusive_:
        row_ = row_.drop(labels=param_ + '#' + station_) # удаляем значение данной метеостанции из ряда
        # Находим среднюю по обратным квадратам расстояния относительно данной метеостанций
        mean_value_ = inverse_distance_avg(row_, param_, station_, df_dists_=df_station_dists, power_=2)
        std_value_ = np.nanstd(row_, ddof=1) if len(row_.dropna()) > 1 else np.nan
    elif not IDW_ and inclusive_:
        mean_value_ = np.nanmean(row_) if len(row_.dropna()) > 0 else np.nan # средняя по серии, игнорируя nan
        # сигма по серии, игнорируя nan, степень свободы=1 (где количество нeNaN больше 1)
        std_value_ = np.nanstd(row_, ddof=1) if len(row_.dropna()) > 1 else np.nan
    elif not IDW_ and not inclusive_:
        row_ = row_.drop(labels=param_ + '#' + station_) # удаляем значение данной метеостанции из ряда
        mean_value_ = np.nanmean(row_) if len(row_.dropna()) > 0 else np.nan # средняя по серии, игнорируя nan
        std_value_ = np.nanstd(row_, ddof=1) if len(row_.dropna()) > 1 else np.nan

    # вычисляем границы вероятности распределения confidence для нормального распределения
    # относительно mean_value_ с scale равном std_value_ только,
    # если std_value_ не является NaN и std_value_ != 0
    if (pd.notna(std_value_) and (std_value_ != 0)):
        result_ = norm.interval(
            confidence=confidence_,
            loc=mean_value_,
            scale=std_value_
        )
    elif (pd.notna(std_value_) and (std_value_ == 0)): # std_value_ существует и равно 0 (все элементы равны)
        result_ = (mean_value_, mean_value_) # Чтобы далее это не было расценено как выброс
    else:
        result_ = np.nan # иначе - вернём NaN

return result_

```

## 2.5. Функция оценки индивидуального отклонения признака в отношении к средней

In [21]:

```
def individ_variance(row_, value_, mean_):
    """Вычисляет отношение абсолютного отклонения частной величины от средней к этой средней
ПРИНИМАЕТ:
- индивидуальное значение,
- среднюю.
ВОЗВРАЩАЕТ:
- частное от индивидуального абсолютного отклонения и средней"""

    return (abs(value_ - mean_)) / mean_ if mean_ != 0 else (abs(value_ + 1 - mean_)) / (mean_ + 1 * len(row_))
# Если знаменатель - mean_ окажется равным нулю, сдвинем значение на 1 и изменим среднюю
# (если к каждому значению прибавить k, то средняя увеличится на k*n)
```

## 2.6. Функция вычисления выбросов в поле метеостанций вне пределов доверительного интервала, определённого либо в количестве сигм, либо в диапазоне границ вероятности нормального распределения

In [22]:

```
# функция для вычисления выбросов вне пределов доверительного интервала, определённого либо в количестве сигм,
# либо в диапазоне границ вероятности нормального распределения
def field_outliers(row_, method_='sigma', criterium_=3, IDW_=False, param_=None, station_='Rfrnce_point', inclusive_=True):
    """ ТОЛЬКО ДЛЯ АРХИВА ПАРАМЕТРОВ
    Вычисляет выбросы в поле метеостанций и выводит их характеристики
    Принимает:
    - row_ - значения из ряда DF (default = None),
    - method_{'sigma' | 'norm'} - какую функцию использовать для оценки выбросов:
        среднеквадратическое отклонение или границы вероятности нормального распределения (default='sigma'),
    - criterium_ - числовое значение передаваемое функции оценки границ распределения, зависит от параметра 'method':
        для 'sigma' - множитель для среднеквартатического отклонения, для 'norm' - вероятностные границы (от 0 до 1),
    - IDW_ (boolean) использовать ли усреднение по обратным квадратам расстояний (default=False),
    - param_ - название параметра для вычисления IDW (default=None). Обязателен для IDW_=True, а так же
        при установки sigma_inclusive_ = False, в противном случае вернёт ошибку,
    - station_ - название метеостанции, для которой вычисляются пределы (значение для неё исключается из подсчёта
        средней и сигм
    (default='Rfrnce_point' - геометрический центр поля метеостанций),
```

- `inclusive_` (при `IDW_=True` имеет смысл только при указании `station_`, отличной от '`Rfrnce_point`') - включать ли значение для данной метеостанции в расчёт средней, при `station_='Rfrnce_point'` значение для данной метеостанции будет включено в подсчёт средней независимо от значения `inclusive_` (boolean, `default=True`).

Возвращает:

- Если выбросы обнаружены - список из кортежей (на каждый выброс свой кортеж): значение выброса, его индекс в серии, границы  $\sigma$  сигм или вероятности, ему соответствующие, и индивидуальная вариация (абсолютное отклонение в отношении к средней)
- Если выбросов нет - возвращает `NaN`

**ВНИМАНИЕ:** Единственное значение в строке будет всегда расцениваться как выброс

```

"""
list_outliers_ = [] # Список для значений и параметров выбросов
## 
if len(row_.dropna()) == 1: # Если имеем единственное значение в строке, сразу вывести его атрибуты, как выброса
    val_ = row_.dropna().values[0] # Получаем единственное значение, отбросив все NaN
    # Вычисляем индекс этого значения
    idx_ = row_.tolist().index(val_)
    # добавить выброс и его параметры в список:
    list_outliers_.append((val_, idx_, val_, val_))
return list_outliers_ # Вывести список
## 

# В зависимости от 'method_' определим границы доверительного интервала вызовом соответствующей функции
if method_ == 'sigma':
    limits_ = sigma_n_limits(row_=row_, n_sigma_ = criterium_, IDW_ = IDW_, param_ = param_, station_=station_,
                             inclusive_= inclusive_)
elif method_ == 'norm':
    limits_ = norm_probability_limits(row_=row_, confidence_=criterium_,
                                        IDW_ = IDW_, param_ = param_, station_=station_,
                                        inclusive_=inclusive_)

list_outliers_ =[ ] # Список для значений и параметров выбросов
for i_, val_ in enumerate(row_): # по порядковому номеру и значению в полученной строке DF

    # Если Limits_!=NaN и значение вне границ доверительного интервала
    if pd.notna(limits_) and (val_ < limits_[0] or val_ > limits_[1]):
        # порядковый номер значения в row_, приведённом к списку
        idx_ = i_
        # добавить выброс и его параметры в список:
        list_outliers_.append((val_, idx_, limits_[0], limits_[1]))

if len(list_outliers_) > 0: # Если выбросы есть (длина списка больше 0)

```

```
        return list_outliers_ # Вывести список
    else:
        return np.nan # Иначе вывести NaN
```

## 2.7. Функция вычисления выбросов во временном ряду по каждой метеостанции вне пределов доверительного интервала, определённого либо в количестве сигм, либо в диапазоне границ вероятности нормального распределения

```
In [23]: def time_outliers(df_, row_, td_symbol_='D', td_quant_='5', equal_hours_=False,
                     method_='sigma', criterium_=3, inclusive_=True):
    """
    Вычисляет выбросы во временных рядах по каждой метеостанции в поле метеостанций и выводит их характеристики
    ПРИНИМАЕТ:
        - df_ - датафрейм, в котором ищутся выбросы,
        - row_ строка со значением параметра по метеостанциям на определённый момент наблюдения,
        - td_symbol_ - строковое значение периода для передачи Timedelta (default='D' - day),
        - td_quant_ - строковое значение количества периодов для передачи Timedelta (default='5')
        - equal_hours_ - boolean использовать ли для вычислений наблюдения на один и тот же час (default=False)
        - method_ {'sigma'|'norm'} - использовать для оценки выбросов количество сигм или вероятностные границы нормального
            распределения (default='sigma')
        - criterium_ (float) значения для критерия определения выброса: для метода 'sigma' - количество сигм,
            для метода 'norm' - значение вероятности (от 0 до 1) (default - для 'sigma' - =3)?
        - inclusive_ - boolean использовать ли текущее значение параметра для подсчёта средней и сигмы (default=True).
    ВОЗВРАЩАЕТ:
        - Если выбросы обнаружены - список из кортежей (на каждый выброс свой кортеж):
            значение выброса, его индекс в серии, границы n сигм или вероятности, ему соответствующие.
        - Если выбросов нет - возвращает NaN
    ВАЖНО: при методе 'norm' значения для Timedelta должны включать в себя более 3 моментов наблюдения
    ВНИМАНИЕ: Единственное значение в строке будет всегда расцениваться как выброс
    """

    td_string_ = f'{td_quant_}{td_symbol_}' # Стока для определения периода Timedelta
    start_time_ = row_.name - pd.Timedelta(td_string_) # Время начала периода выборки для вычисления средней и сигмы
    end_time_ = row_.name + pd.Timedelta(td_string_) # Время окончания периода выборки для вычисления средней и сигмы
    obsrvtn_hour_ = row_.name.hour # Час наблюдения

    # Создаём маску для выбора значений из df_,
    mask_ = (df_.index >= start_time_) & (df_.index <= end_time_) # временной интервал для выборки
```

```

if not inclusive_: # если данное значение параметра не включать в подсчёт средний и сигмы:
    mask_ = mask_ & (df_.index != row_.name)
if equal_hours_: # если используем фиксированный час наблюдений
    mask_ = mask_ & (df_.index.hour == obsrvtn_hour_)

list_outliers_ =[] # Список для значений и параметров выбросов

# Проходим по ряду значений параметра между метеостанциями на данный момент времени:
for label_ in row_.index: # Проходим по ряду значений для метеостанций
    ser_ = df_.loc[mask_][label_] # для каждой станции берём серию значений в соответствии с временным интервалом
    checked_value_ = row_[label_] # значение, проверяющееся на выброс - текущее значение, определённое row_

    # В зависимости от 'method_' определим границы доверительного интервала вызовом соответствующей функции
    if method_ == 'sigma':
        limits_ = sigma_n_limits(row_=ser_, n_sigma=criterium_, IDW=False, param=None, station=None,
                                  inclusive_=True) # мы уже сделали все операции, связанные с inclusive_ - нечего исключать
    elif method_ == 'norm':
        limits_ = norm_probability_limits(row_=ser_, confidence=criterium_,
                                            IDW=False, param=None, station=None,
                                            inclusive_=True) # мы уже сделали все операции, связанные с inclusive_ - нечего исключать

    # Если limits_ !=NaN и значение вне границ доверительного интервала, или границы неопределены
    # (т.е. единственное значение в серии)
    if (
        pd.notna(limits_) and (
            (checked_value_ < limits_[0] or checked_value_ > limits_[1]) or
            (pd.isna(limits_[0]) or pd.isna(limits_[1])))
        )
    ):
        # порядковый номер столбца для значения с label
        idx_ = df_.columns.get_loc(label_)
        # добавить выброс и его параметры в список:
        list_outliers_.append((checked_value_, idx_, limits_[0], limits_[1]))
    elif pd.isna(limits_): # norm_probability_limits вернула NaN - может единственное значение в серии
        idx_ = df_.columns.get_loc(label_)
        # добавить выброс и его параметры в список:
        list_outliers_.append((checked_value_, idx_, np.nan, np.nan))

    if len(list_outliers_) > 0: # Если выбросы есть (длина списка больше 0)
        return list_outliers_ # Вывести список значений и параметров выбросов
    else:
        return np.nan # Иначе вывести NaN

```

## 2.8. Функция поиска референсного значения для параметра для сравнения с расчётным значением

```
In [24]: def reference_param_extractor(station_, param_, dict_archive_, idx_station_):
    """
    Функция поиска референсного значения параметра в словаре DFs архивов метеостанций на основе подобных индексов DFs
    Принимает:
    - название станции;
    - параметр, для которого надо найти значение;
    - название словаря DF, где надо искать значение (словарь архивов метеостанций!);
    - индекс DateTime переданного функции текущего значения station_ в текущем DF.
    Выводит:
    - значение искомого параметра.
    ПРЕДПОЛАГАЕТСЯ, что поиск значений производится по индексу в обоих сопоставляемых DFs
    """
    name_df_ = f'df_{station_}' # преобразуем название станции в название ключа её архива в словаре архивов
    df_arch_ = dict_archive_[name_df_] # DF архива по ключу в словаре архивов
    # значение параметра в DF архива, где индекс совпадает с индексом station_:
    value_ = df_arch_.at[idx_station_, param_]
    # print(idx_station_, value_)
    return value_
```

## 2.9. Функция вывода списка названия станций из списка кортежей выбросов

```
In [25]: def stations_from_outliers(row_, column_name_: str, param_: str):
    """
    Функция вывода списка названия станций из списка кортежей выбросов
    Принимает:
    - серию (строку) из датафрейма
    - название столбца, содержащего списки кортежей с параметрами выбросов
    - название параметра по которому произошёл выброс - без 'df_'
    Возвращает:
    - список названий метеостанций к которым относятся выбросы значений
    ПРЕДПОЛАГАЕТСЯ, что столбец column_name_ в качестве значений содержит списки кортежей
    """
    # По списку кортежей в row row_[column_name_] получаем значение индекса столбца метеостанции и
    # вычленяем из названия этого столбца называние станции - название столбца после символа #.
```

```
list_stations_ = [row_.index[i[1]][len(param_)+1 :] for i in row_[column_name_]]  
return list_stations_
```

## 2.10. Функция создания логических масок для разбиения архивного датафрейма на климатические сезоны

In [26]:

```
def season_masks(df_:pd.DataFrame):  
    """  
        Функция создания логических временных масок для разбиения датафрейма на климатические сезоны.  
        Принимает:  
        - Датафрейм с индексом TimeStamp  
        Возвращает  
        - Словарь с масками, где ключами являются названия сезонов (spring, summer, autumn, winter)  
    """  
  
    # Определим логические маски для климатических сезонов  
    mask_winter_ = (  
        (df_.index.month == 11) & (df_.index.day >= 5)  
    ) | (  
        (df_.index.month > 11) | (df_.index.month < 4)  
    ) | (  
        (df_.index.month == 4) & (df_.index.day <= 4)  
    )  
  
    mask_spring_ = (  
        (df_.index.month == 4) & (df_.index.day >= 5)  
    ) | (  
        (df_.index.month == 5) & (df_.index.day <= 18)  
    )  
  
    mask_summer_ = (  
        (df_.index.month == 5) & (df_.index.day >= 19)  
    ) | (  
        (df_.index.month > 5) & (df_.index.month < 9)  
    ) | (  
        (df_.index.month == 9) & (df_.index.day <= 14)  
    )  
  
    mask_autumn_ = (  
        (df_.index.month == 9) & (df_.index.day >= 15)  
    ) | (  
        (df_.index.month == 10)
```

```
) | (
    (df_.index.month == 11) & (df_.index.day <= 4)
)
dict_season_masks_ = {'spring': mask_spring_, 'summer': mask_summer_, 'autumn': mask_autumn_, 'winter': mask_winter_}
return dict_season_masks_
```

## 2.11. Функция замены некорректных значений в архивах метеостанций на корректные

```
In [27]: # Функция замены значений в архивах МЕТЕОСТАНЦИЙ
def correct_errors_stations(x_corr_, station_, param_, dict_archive_, idx_x_):
    """
    Заменяет значения в архивах на корректные: только архивы по метеостанциям!
    Принимает:
        корректное значение параметра;
        название станции;
        параметр, для которого надо заменить значение в архиве;
        название словаря DF архивов, где надо заменить значение;
        индекс переданного функции текущего x_ в df_x_.

    Выводит:
        - (Строку с подтверждением замены значений.)
    Возвращает:
        - ПРЕДПОЛАГАЕТСЯ, что поиск значений производится по индексу в обоих сопоставляемых DFs

    ВНИМАНИЕ! ПОЛЬЗОВАТЬСЯ С ОСТОРОЖНОСТЬЮ, ЧТОБЫ НЕ ПОВРЕДИТЬ АРХИВЫ!
    """
    name_df_ = f'df_{station_}' # преобразуем название станции в название ключа её архива в словаре архивов
    df_arch_ = dict_archive_[name_df_] # DF архива по ключу в словаре архивов
    # записываем значение параметра в DF архив, где индекс совпадает с индексом x_:
    x_faulty_ = df_arch_.at[idx_x_, param_] # значение, подлежащее замене
    df_arch_.at[idx_x_, param_] = x_corr_

##    # print(f'Параметр {param_} в архиве {name_df_} на момент наблюдения {idx_x_} установлен в значение {x_corr_}, '
##          # f'(прежнее значение {x_faulty_})')
##    return None
```

## 2.12. Функция замены некорректных значений в архивах параметров на корректные

```
In [28]: # Функция замены значений в архивах ПАРАМЕТРОВ
def correct_errors_parameters(x_corr_, station_, param_, dict_archive_= dict_df_parameters, idx_x_= ""):
    """
    Заменяет значения в архивах на корректные: только архивы по параметрам!
    Принимает:
        корректное значение параметра;
        название станции;
        параметр, для которого надо заменить значение в архиве;
        название словаря DF архивов, где надо заменить значение;
        индекс переданного функции текущего x_ в df_x_.

    Выводит:
        - (Строку с подтверждением замены значения.
    Возвращает:
        ПРЕДПОЛАГАЕТСЯ, что поиск значений производится по индексу в обоих сопоставляемых DFs

    ВНИМАНИЕ! ПОЛЬЗОВАТЬСЯ С ОСТОРОЖНОСТЬЮ, ЧТОБЫ НЕ ПОВРЕДИТЬ АРХИВЫ!
    """
    name_df_ = f'df_{param_}' # преобразуем название параметра в название ключа её архива в словаре архивов
    df_arch_ = dict_archive_[name_df_] # DF архива по ключу в словаре архивов
    col_name = f'{param_}#{station_}' # Формируем название столбца
    # записываем значение параметра в DF архив, где индекс совпадает с индексом x_:
    x_faulty_ = df_arch_.at[idx_x_, col_name] # значение, подлежащее замене
    df_arch_.at[idx_x_, col_name] = x_corr_

    ##
    # print(f'Параметр {col_name} в архиве {name_df_} на момент наблюдения {idx_x_} установлен в значение {x_corr_}, '
    #       f'(прежнее значение {x_faulty_})')
    # return None
```

## 2.13. Функция замены NaN на среднюю величину, взвешенную по обратным степеням расстояний между метеостанциями

```
In [29]: def row_nan_idw_correct (row_, name_param_, power_):
    """
    Функция замены NaN на средневзвешенные по обратным квадратам расстояний idw = inverted distance weight
    Для каждого NaN в строке row_ :
        - вычисляет необходимые для inverse_distance_avg параметры;
        - вызывает inverse_distance_avg;
        - полученный результат использует сразу для:
            - замены NaN в текущем ряду (для последующих вычислений),
            - замены NaN в архивах метеостанций,
```

- замены NaN в архивах параметров.

Принимает:

- ряд значений,
- название параметра,
- степень для весов для функции `inverse_distance_avg`

Возвращает: -

三

## 2.14. Функция замены NaN на значение, полученное пространственной экстраполяцией методом кригинга

In [30]:

```
def row_nan_kriging_correct (row_, name_param_):
    """
    ВНИМАНИЕ! Данная функция привязана к структуре данных в df_stations (и производных от него) и dict_df_parameters!
    Предполагается, что перечень метеостанций в df_stations идёт в той же последовательности, что и в row_!

    Функция замены NaN на значение, полученное пространственной экстраполяцией методом кригинга
    Для каждого NaN в строке row_ :
        - вычисляет индексы пропущенных значений;
        - определяет вариограмму
        - вызывает модель ordinary.kriging;
        - полученный результат использует сразу для:
            - замены NaN в текущем ряду (для последующих вычислений),
            - замены NaN в архивах метеостанций,
            - замены NaN в архивах параметров.
    ЕСЛИ В РЯДУ ВСЕ ЗНАЧЕНИЯ ОПРЕДЕЛЕНЫ, ИЛИ В ПРОЦЕССЕ В ПРОЦЕССЕ ОПРЕДЕЛЕНИЯ ВАРИОГРАММЫ ВОЗНИКАЮТ ОШИБКИ
    (КАК ПРАВИЛО ИЗ-ЗА МАЛОГО КОЛИЧЕСТВА ЗНАЧЕНИЙ В ПОЛЕ МЕТЕОСТАНЦИЙ), ТО ФУНКЦИЯ ПРЕРЫВАЕТСЯ

    Принимает:
        - ряд значений,
        - название параметра для поиска значений
    Возвращает:
    """

    # Определяем индексы значений NaN в row_
    arr_idx_nan_ = np.where(np.isnan(row_))[0] # np.where только с условием - выводит кортеж из массива пнтур!
    # Если в массив индексов NaN пустой (есть все нужные значения) или
    # если разница между длинной ряда и длинной массива индексов NaN меньше 3 (ограничение сэмплов для вариограммы)
    # то выйти из функции
    if (len(arr_idx_nan_) == 0) or ((len(row_) - len(arr_idx_nan_)) < 3):
        #
        #     print(f"\nВыход из функции из-за количества значений."
        #           f"Количество переданных определённых значений={len(row_) - len(arr_idx_nan_)}.")
        #
    #
    return # выход из функции

    # Вызовем функцию kriging_extrapolation и получаем массив предиктов для row_
    arr_predict_ = kriging_extrapolation(row_=row_ ,
```

```

        df_coords_=df_coords_full)

# если массив предиктов неопределён из-за невозможности построить вариограмму (не возвращён np.ndarray)
if not isinstance(arr_predict_, np.ndarray):
    return # выход из функции

# Заменяем NaN в текущем ряду значений
row_[arr_idx_nan_] = arr_predict_[arr_idx_nan_] # присваиваем элементам row_ предикты по индексу бывших NaN-ов

for idx_ in arr_idx_nan_:
    station_ = row_.index[idx_][len(name_param_) + 1 :]
    # Заменяем значения в архивах метеостанций на корректные.
    correct_errors_stations(x_corr_ = arr_predict_[idx_],
                             station_ = station_,
                             param_ = name_param_,
                             dict_archive_ = dict_df_locations,
                             idx_x_ = row_.name)

    # Заменяем значения в архивах параметров на корректные.
    correct_errors_parameters(x_corr_ = arr_predict_[idx_],
                             station_ = station_,
                             param_ = name_param_,
                             dict_archive_ = dict_df_parameters,
                             idx_x_ = row_.name)

#     return None

```

## 2.15 Функция нахождения значения параметра для ближайшей от искомой метеостанции.

In [31]:

```

def closest_value(station_, observation_, param_):
    """ Функция поиска ближайшей метеостанции со значением параметра отличным от NaN
    ПРИНИМАЕТ:
    - station_ - название искомой метеостанции,
    - observation_ - момент наблюдения,
    - param_ - параметр, для которого ищется значение
    ВОЗВРАЩАЕТ:
    кортеж значений (название ближайшей метеостанции, значение параметра на ней,
    расстояние до неё от искомой станции, начальный азимут от искомой станции)
    ВНИМАНИЕ: 1. НЕ ПРИМЕНЯТЬ К РЯДАМ СПЛОШНЫХ NaN
                2. Если все ближайшие значения - NaN, то в качестве ближайшей станции возвращается Rfrnce_point,
                   а в качестве ближайшего значения - значение самой исходной станции

```

```
"""
smallest_ = 1 # Начальное значения порядка в ряду минимальных по порядку
# Начальные значение
closest_dist_ = 0
closest_bearing_ = 0
closest_station_ = None
closest_val_ = np.nan # Начальное значение ближайшего значения показателя

while np.isnan(closest_val_):
    smallest_ += 1 # переходим к предыдущему минимуму по порядку
    # принудительно назначим Reference_point ближайшей станцией для случая, когда единственное значение
    # существует только для исходной станции. Определим для Rfrnce_pnt то же значение, что для исходной станции
    # и выйдем из цикла
    if smallest_ > len(df_station_dists.station): # вышли итерацией за количество станций
        closest_dist_ = df_station_dists[station_].loc[df_station_dists.station == 'Rfrnce_point'].values[0]
        closest_station_ = 'Rfrnce_point'
        closest_val_ = dict_df_locations[f'df_{station_}'].at[observation_, param_]
        continue

#     print(smallest_, station_, observation_, closest_val_, closest_dist_, closest_bearing_, closest_station_)

# минимальное расстояние до станции, где есть наблюдение параметра
closest_dist_ = df_station_dists[station_].nsmallest(smallest_).iloc[-1]
# ближайшая станция:
closest_station_ = (df_station_dists.loc[df_station_dists[station_] == closest_dist_]
                     .station
                     .values[0])

# За исключением Чашниково и условного центра
if (closest_station_ == 'Chashnikovo') or (closest_station_ == 'Rfrnce_point'):
    continue
# Значение на ближайшей станции, где оно есть
closest_val_ = dict_df_locations[f'df_{closest_station_}'].at[observation_, param_]
# начальный азимут на ближайшую станцию, где есть наблюдение параметра
closest_bearing_ = (df_station_bearings[df_station_bearings.station == closest_station_]
                     .loc[:,station_]
                     .values[0])
return (closest_val_, closest_dist_, closest_bearing_, closest_station_)
```

**3. Исследование и обработка индивидуальных показателей метеорологических наблюдений: Температура воздуха ( $T$ ,  $T_{min}$ ,  $T_{max}$ )**

См. тетрадь 2

**4. Исследование и обработка индивидуальных показателей метеорологических наблюдений: Атмосферное давление ( $P_{sea}$ ,  $P_{station}$ ,  $P_{drift}$ )**

См. тетрадь 3

**5. Исследование и обработка индивидуальных показателей метеорологических наблюдений: Концентрация водяных паров в воздухе - относительная влажность воздуха и температура точки росы ( $Humid$ ,  $Dew\_point$ )**

См. тетрадь 3

**6. Исследование и обработка индивидуальных показателей метеорологических наблюдений: Численные показатели состояния почвы - температура почвы, высота снегового покрова ( $Soil\_T$ ,  $Snow\_height$ )**

См. тетрадь 4

# **7. Исследование и обработка индивидуальных показателей метеорологических наблюдений: Дальность видимости (Visibility)**

## **7.1. Дальность видимости: Visibility**

Метеорологическая дальность видимости (МДВ) – S представляет собой условное выражение прозрачности атмосферы и равна предельному расстоянию, на котором в дневное время видны черные предметы достаточно больших угловых размеров (больше  $15' \cdot 15'$ ), проектирующиеся на северной стороне неба у горизонта. Угловой размер предмета должен быть больше  $15' \cdot 15'$  для сохранения постоянства порога контрастной чувствительности зрения. Если угловой размер предмета будет меньше  $15' \cdot 15'$ , то он не будет наблюдаваться не потому, что он закрывается туманом или дымкой, а вследствие недостаточной остроты зрения наблюдателя.

Дальность видимости на метеостанциях измеряется как в дневное, так и вночное время суток. Метеорологическая дальность видимости SM (МДВ) равна расстоянию, на котором под воздействием атмосферной дымки в светлое время суток контраст абсолютно черной поверхности, проектирующейся на фоне дымки и имеющей угловые размеры не менее  $0,5^\circ$ , принимает пороговое значение = 0,02.

В настоящее время, метеорологическая дальность видимости определяется с помощью приборов. Действие прибора основано на оптическом раздвоении изображения наблюдаемых объектов с последующим приведением к равенству яркости этих изображений (метод фотометрирования) или с последующим гашением одного из них поворотом поляроида (метод гашения). Зная угол поворота поляроида и расстояние до наблюдаемого объекта, по таблицам или по формуле, которая приведена в описании прибора, определяют значение МДВ.

Согласно Международной шкале видимости метеорологическую дальность видимости на метеорологических станциях характеризуют в баллах. Такая шкала видимости принята с целью внесения единобразия при определении прозрачности атмосферы и удобства при шифровке метеорологических сводок.

Туманом называется такое состояние атмосферы, когда МДВ днем меньше 1 км.

Дымкой называется такое состояние атмосферы, когда МДВ лежит в пределах 1 – 20 км. Международная шкала метеорологической дальности видимости содержит 10 баллов, обозначаемых каждой одной цифрой.

### *Отличная видимость*

9 баллов: МДВ 50 км и более, удельная прозрачность атмосферы  $0,925\$$  и более

### *Хорошая видимость*

8 баллов, МДВ 20—50 км, удельная прозрачность атмосферы  $0,823-0,925\$$

### *Дымка*

- слабая  
7 баллов, МДВ 10—20 км, удельная прозрачность атмосферы  $0,676-0,823\$$
- заметная  
6 баллов, МДВ 4—10 км, удельная прозрачность атмосферы  $0,376-0,676\$$
- сильная  
5 баллов, МДВ 2—4 км, удельная прозрачность атмосферы  $0,141-0,376\$$
- очень сильная  
4 балла, МДВ 1—2 км, удельная прозрачность атмосферы  $0,02-0,141\$$

### *Туман*

- слабый 3 балла, МДВ 0,5—1 км, удельная прозрачность атмосферы  $10^{-3.4}-0.02\$$
- заметный  
2 балла, МДВ 200—500 м, удельная прозрачность атмосферы  $10^{-8.5}-10^{-3.4}\$$
- сильный 1 балл, МДВ 50—200 м, удельная прозрачность атмосферы  $10^{-34}-10^{-8.5}\$$
- очень сильный  
0 баллов, МДВ менее 50 м, менее  $10^{-34}\$$

(источник: <https://studfile.net/preview/6179887/page:2/>)

## **7.1.1. Поиск и удаление ошибок показателя дальности видимости: Visibility**

Для данного раздела обозначим константу названия параметра.

In [32]:

```
PARAMETER71 = 'Visibility'
```

## Создаём временный DF для работы с параметром Snow\_height

```
In [33]: param_df_name = f'df_{PARAMETER71}' # преобразуем полученное значение в df_PARAMETER71 - ключ словаря dict_df_parameters  
# Создадим временный df  
df_tmp71 = dict_df_parameters[param_df_name].copy(deep=True)  
df_tmp71.sample(5, random_state=56)
```

Out[33]:

	Visibility#V_Volochek	Visibility#Staritsa	Visibility#Kashyn	Visibility#Tver	Visibility#Klin	Visibility#Dmitrov	Visibility#Volokolamsk	Visibility#Moscow
2014-02-22 03:00:00	10.0	20.0	50.0	10.0	10.0	10.0	10.0	10.0
2015-05-16 03:00:00	10.0	10.0	10.0	10.0	10.0	10.0	10.0	10.0
2020-06-18 18:00:00	20.0	NaN	50.0	10.0	10.0	10.0	10.0	10.0
2019-12-02 21:00:00	10.0	10.0	10.0	10.0	4.0	4.0	4.0	4.0
2008-06-05 18:00:00	NaN	20.0	NaN	10.0	20.0	15.0	40.0	

Определим последовательность действий, для поиска ошибок показателя "Высота снегового покрова".

1. Проверим синхронность наблюдения всеми метеостанциями.
2. Определяем выбросы в поле метеостанций - формируем кандидатов в ошибки.
3. Отдельно проверяем, есть ли единичные значения в рядах - это тоже кандидаты в ошибки.
4. Проверяем каждый из этих кандидатов во временном ряду (только эти выбросы, а не весь DF!) - не подтвердилось - отбрасываем
5. То, что осталось и есть ошибка.

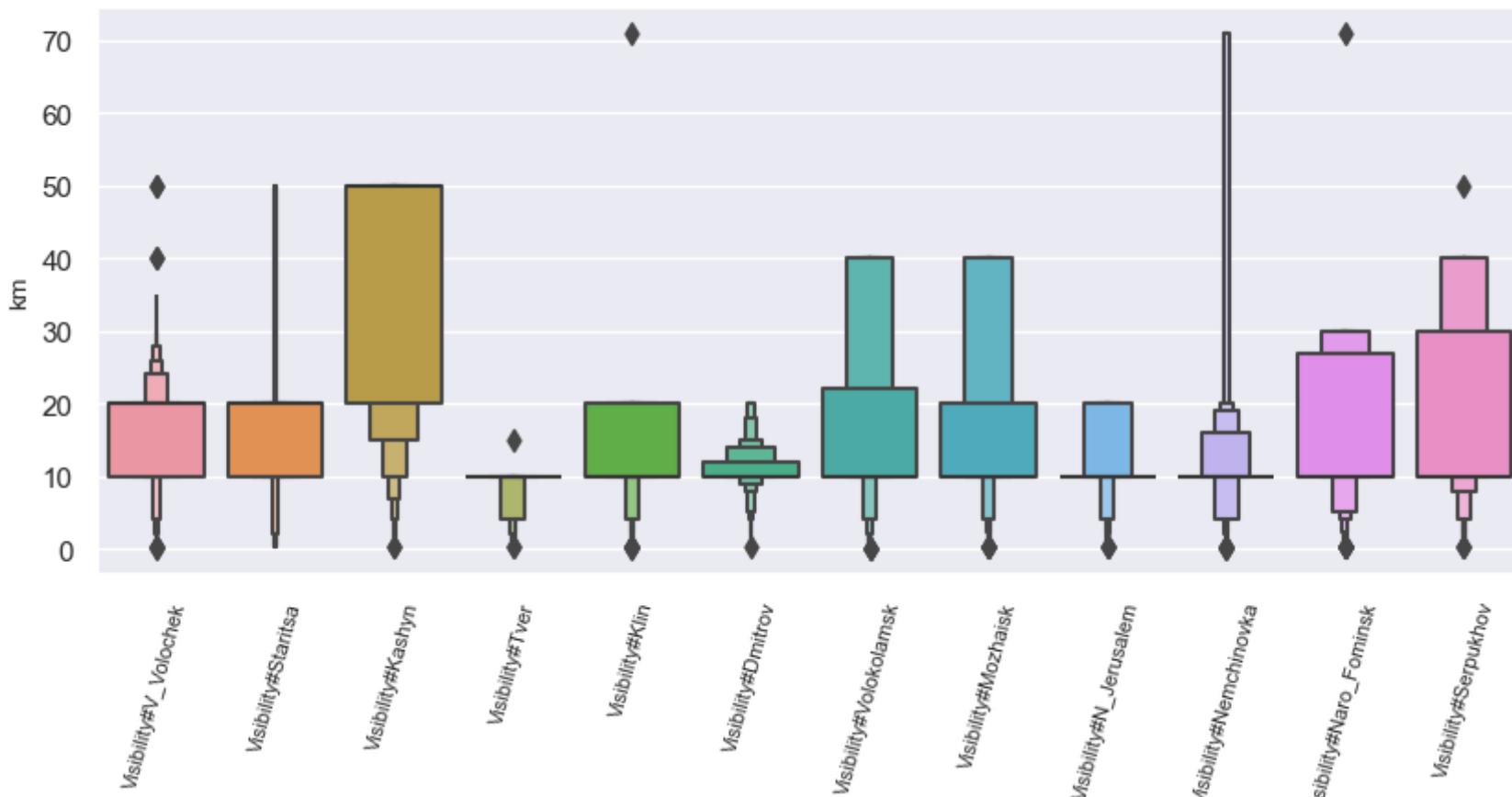
**Визуализируем архив дальности видимости (Visibility) по сезонам**

Исходя из данных о климате Московской области, временные границы сезонов определены следующим образом.

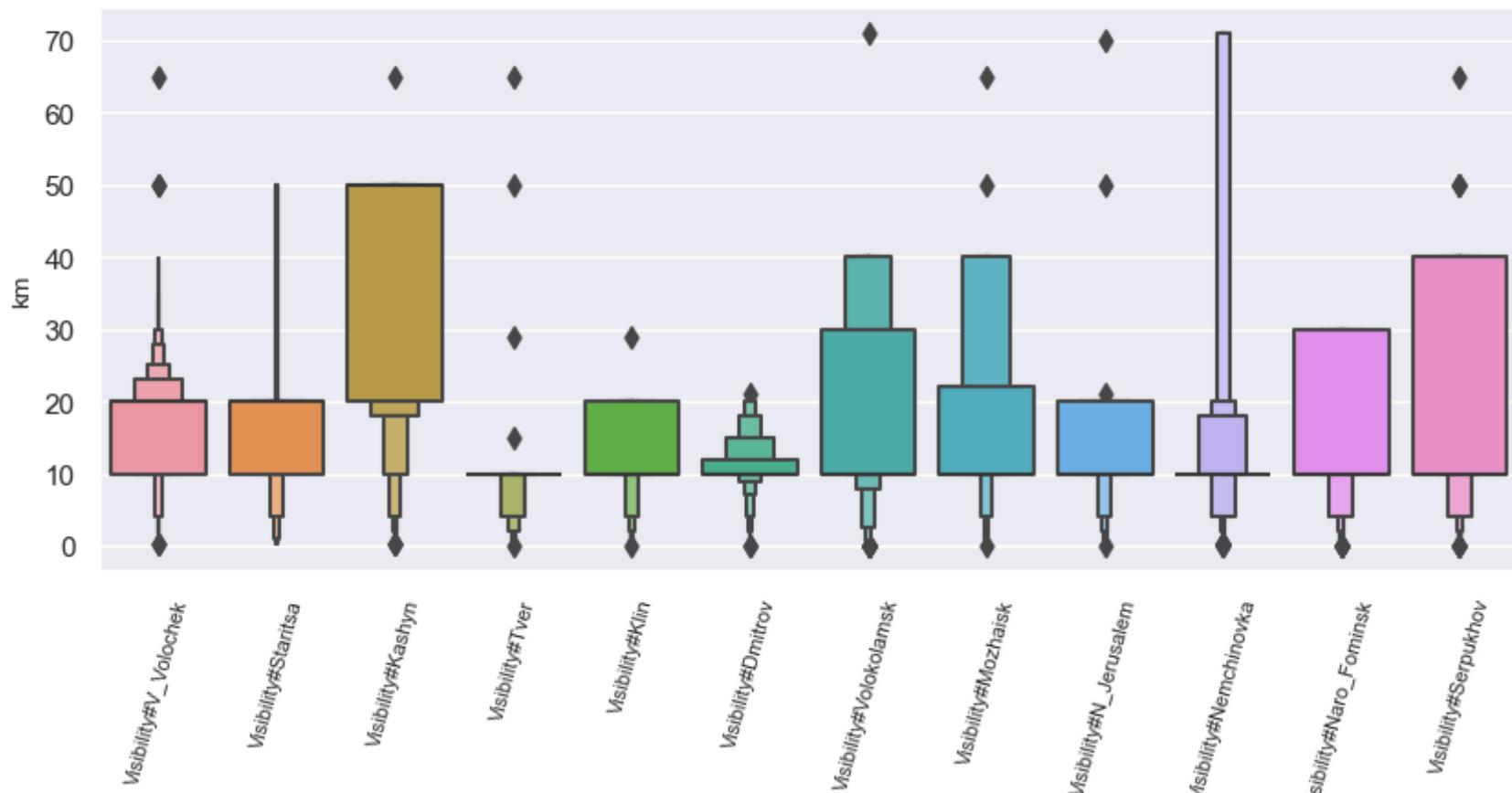
- Зима (ниже 0°): В среднем длится с 5 ноября по 4 апреля
- Весна (от 0° до +10°): В среднем длится с 5 апреля по 18 мая
- Лето (выше +10°): В среднем длится с 19 мая по 14-15 сентября
- Осень (от +10° до 0°): В среднем длится с 14-15 сентября по 4 ноября

```
In [34]: # В цикле выведем графики давления по метеостанциями в зависимости от сезона
# используем функцию создания масок климатических сезонов
for season_name, season_mask in season_masks(df_tmp71).items():
    fig, ax = plt.subplots(figsize=(10, 4))
    g = sns.boxenplot(data=df_tmp71[season_mask],
                       ax=ax)
    dummy = plt.xticks(rotation=75, size=8)
    dummy = g.set_ylabel('km', size=10)
    dummy = g.set_title(f'Распределение значений {PARAMETER71} в разрезе метеостанций:\n'
                        f'{season_name}')
plt.show()
```

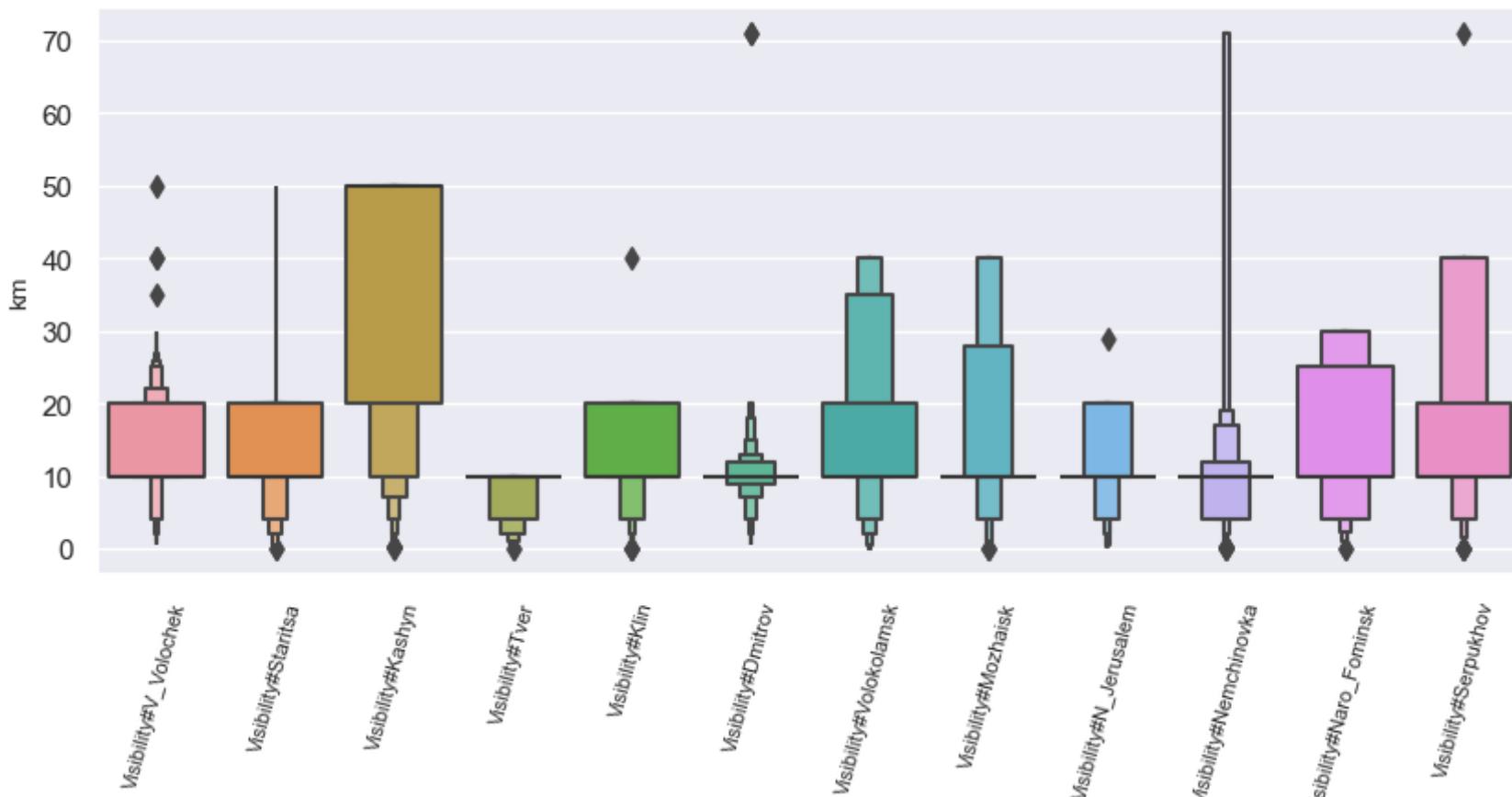
## Распределение значений Visibility в разрезе метеостанций: spring



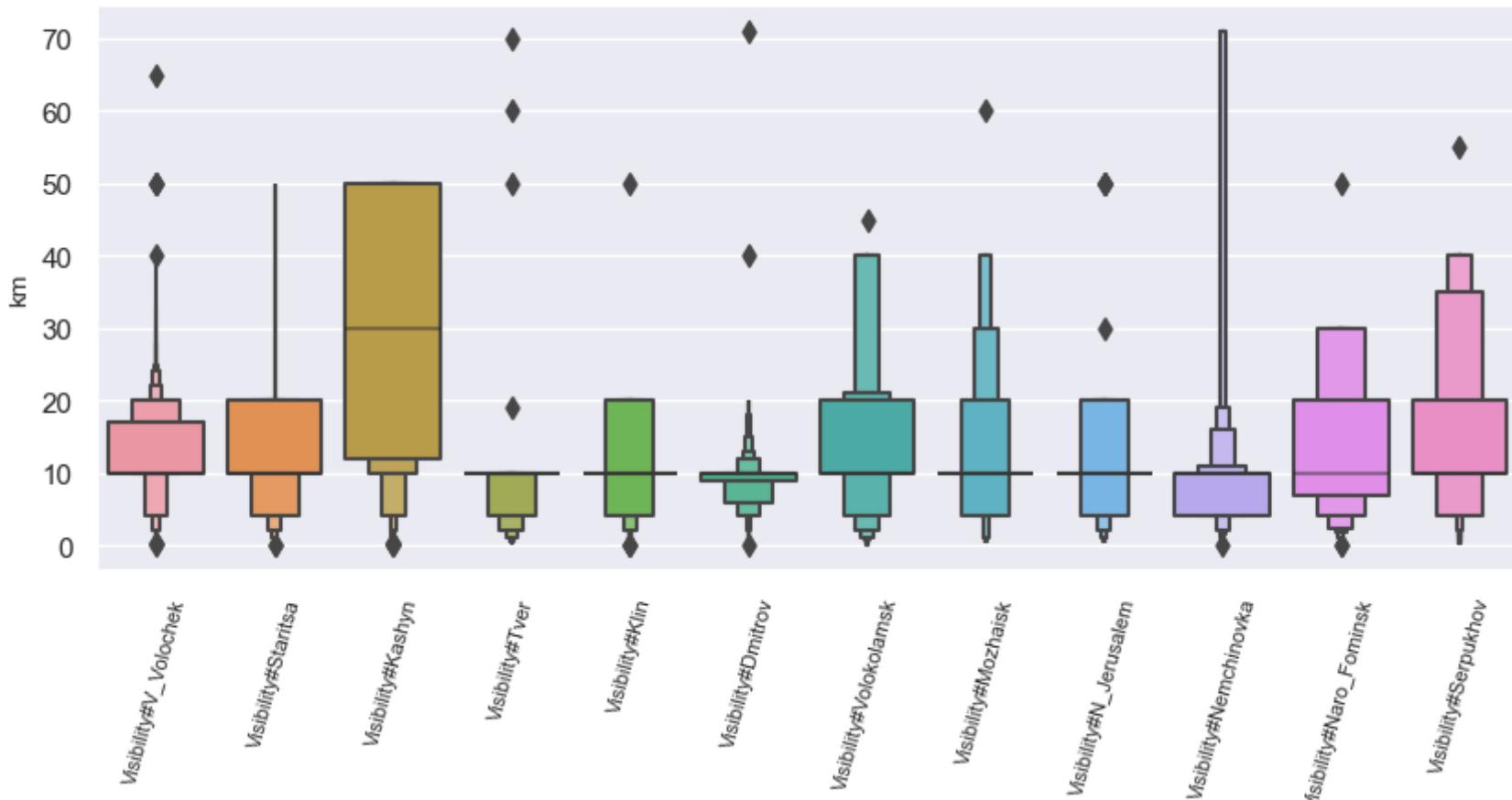
### Распределение значений Visibility в разрезе метеостанций: summer



Распределение значений Visibility в разрезе метеостанций:  
autumn



## Распределение значений Visibility в разрезе метеостанций: winter



Как видно из графиков, данные о дальности видимости сильно разнятся между метеостанциями. Из имеющихся данных невозможно определить, вызвано ли это ошибками или особенностями местных метеорологических условий. Тем не менее, так как разброс данных подобен от сезона к сезону, можно предположить, что метеорологическая дальность видимости имеет свои особенности, определяемые географией каждой локации.

Выведем минимальное и максимальное значения, а также значение медианы и средней для всего DF.

```
In [35]: print(f'Минимальное значение: {np.nanmin(df_tmp71)},\n      f'Максимальное значение: {np.nanmax(df_tmp71)},\n      f'Среднее значение: {df_tmp71.mean()}\n      f'Медиана: {df_tmp71.median()}'
```

```
f'Средняя: {np.nanmean(df_tmp71)},\n'
f'Медиана: {np.nanmedian(df_tmp71)}')
```

Минимальное значение: 0.025,  
Максимальное значение: 71.0,  
Средняя: 15.11261356700268,  
Медиана: 10.0

Определим, с какого момента началась фиксация дальности видимости метеостанциями.

```
In [36]: df_tmp71.dropna(how='all').index.min() # удаляем сплошные NaNы, выводим минимум индекса
```

```
Out[36]: Timestamp('2005-02-01 03:00:00')
```

Дальности видимости фиксировалась с начала архивов.

Выясним в какие часы происходила фактическая фиксация дальности видимости По стандарту она должна фиксироваться каждые 3 часа.

```
In [37]: arr_moments = np.array(df_tmp71.dropna(how="all").index.hour.unique().sort_values())
arr_moments
```

```
Out[37]: array([ 0,  3,  6,  9, 12, 15, 18, 21], dtype=int64)
```

```
In [38]: for moment in arr_moments: # по массиву моментов фиксации метеонаблюдений
    print(f'В {moment} час(а/ов) встречается '
          f'{df_tmp71[df_tmp71.index.hour == moment].dropna(how="all").count().sum().astype(int)} раз(а)')
```

В 0 час(а/ов) встречается 64299 раз(а)  
В 3 час(а/ов) встречается 68316 раз(а)  
В 6 час(а/ов) встречается 64246 раз(а)  
В 9 час(а/ов) встречается 71269 раз(а)  
В 12 час(а/ов) встречается 65119 раз(а)  
В 15 час(а/ов) встречается 71097 раз(а)  
В 18 час(а/ов) встречается 65023 раз(а)  
В 21 час(а/ов) встречается 71281 раз(а)

Дальность видимости фиксировалась относительно равномерно.

**Найдем выбросы в поле метеостанций.**

Параметры:

- используем среднюю по обратным квадратам расстояния от центра поля,
- включаем проверяющее значение в подсчёт средней (автоматически, так как средняя рассчитывается от центра поля для всех станций),
- определим границы доверительного интервала в 3 сигмы.

```
In [39]: start_time = time.time() # для замера времени выполнения кода

df_tmp71 = df_tmp71.assign(field_out=df_tmp71.apply(lambda x: field_outliers(row=x,
                                                               method='sigma',
                                                               criterium=3,
                                                               IDW=True,
                                                               param=PARAMETER71,
                                                               station='Rfrnce_point',
                                                               inclusive=True),
                           axis=1)
)

end_time = time.time()
elapsed_time = end_time - start_time
time_format = time.strftime("%H:%M:%S", time.gmtime(elapsed_time))
print(f"На выполнение кода ушло: {time_format}")

На выполнение кода ушло: 00:03:31
```

```
In [40]: df_tmp71.dropna(subset=["field_out"]).sample(5, random_state=56)
```

Out[40]:

	Visibility#V_Volochek	Visibility#Staritsa	Visibility#Kashyn	Visibility#Tver	Visibility#Klin	Visibility#Dmitrov	Visibility#Volokolamsk	Visibility#M
2020-01-08 03:00:00	10.0	10.0	20.0	10.0	10.0	10.0	10.0	10.0
2018-05-30 12:00:00	20.0	20.0	50.0	10.0	10.0	10.0	10.0	10.0
2021-04-07 21:00:00	10.0	20.0	10.0	10.0	10.0	10.0	10.0	10.0
2017-11-28 18:00:00	10.0	10.0	20.0	10.0	4.0	4.0	2.0	
2020-05-08 18:00:00	20.0	20.0	50.0	10.0	10.0	10.0	10.0	10.0



Используемые формулы определения выбросов специально обозначают как выброс в поле метеостанций случай, когда в ряду есть единственное значение. Проверим, есть ли ситуации, когда существует только одно значение параметра в поле метеостанций.

In [41]:

```
# Если значение является единственным в строке, то True, иначе False,
# убираем NaN (берём ряд без столбца field_out) и суммируем результат
single_values_count = df_tmp71.apply(lambda x : True if (len(x[:-1].dropna()) == 1) else False, axis = 1).dropna().sum()
print(f'Количество случаев единичных значений в рядах моментов наблюдений: {single_values_count}'')
```

Количество случаев единичных значений в рядах моментов наблюдений: 9

Проверим максимальное количество выбросов в поле метеостанций на момент наблюдения.

In [42]:

```
# Находим максимальное количество выбросов в строке поля метеостанций
max_field_outliers = df_tmp71.field_out.dropna().apply(lambda x: len(x)).max()
print(f'Максимальное количество выбросов в поле метеостанций за весь период наблюдения не превышает '
      f'{max_field_outliers}'')
```

Максимальное количество выбросов в поле метеостанций за весь период наблюдения не превышает 1

Для дальнейшей работы с аномалиями создадим столбец `error_at`, куда запишем кортеж из `TimeStamp` и названий станций с выбросами.

```
In [43]: # Определяем столбец error_at (помним, что в field_out у нас всегда БОЛЕЕ 1 выброса),  
# значит вторым элементом кортежа в error_at будет СПИСОК станций, по которым найдены выбросы  
  
df_tmp71 = df_tmp71.assign(error_at =  
                           df_tmp71.  
                           apply(lambda x: # Вычленим DateTime index и название станции с выбросом  
                                 # пропустим NaN, проверив, является ли x.field_out списком  
                                 (x.name,  
                                  stations_from_outliers(  
                                      row_=x,  
                                      column_name_= 'field_out', # используем выбросы в поле метеостанций  
                                      param_=PARAMETER71)  
                                 ) if isinstance(x.field_out, list) else np.nan,  
                                 axis=1  
                           )  
                           )
```

```
In [44]: df_tmp71.dropna(subset=["error_at"]).sample(5, random_state=56)
```

Out[44]:

	Visibility#V_Volochek	Visibility#Staritsa	Visibility#Kashyn	Visibility#Tver	Visibility#Klin	Visibility#Dmitrov	Visibility#Volokolamsk	Visibility#Mc
--	-----------------------	---------------------	-------------------	-----------------	-----------------	--------------------	------------------------	---------------

<b>2020-01-08 03:00:00</b>	10.0	10.0	20.0	10.0	10.0	10.0	10.0	10.0
<b>2018-05-30 12:00:00</b>	20.0	20.0	50.0	10.0	10.0	10.0	10.0	10.0
<b>2021-04-07 21:00:00</b>	10.0	20.0	10.0	10.0	10.0	10.0	10.0	10.0
<b>2017-11-28 18:00:00</b>	10.0	10.0	20.0	10.0	4.0	4.0	4.0	2.0
<b>2020-05-08 18:00:00</b>	20.0	20.0	50.0	10.0	10.0	10.0	10.0	10.0



**Для подтверждения, являются ли отобранные значения ошибками, найдём выбросы во временном ряду**

Среди выбросов в поле метеостанций найдем выбросы во временном окне.

1й Вариант (для одного и того же часа наблюдения за несколько дней) Параметры:

- используем границы нормального распределения,
- не включаем проверяемое значение в подсчёт средней и сигмы,
- определим границы доверительного интервала в 97%,
- определим временное окно в +/- 3 дня ('7D'),
- включим в подсчёт только моменты наблюдения на данный час (equalhours=True).

2й Вариант (для всех моментов наблюдения за несколько дней) Параметры:

- используем границы нормального распределения,
- не включаем проверяемое значение в подсчёт средней и сигмы,
- определим границы доверительного интервала в 97%,
- определим временное окно в +/- 24 часов ('48H'),
- включим в подсчёт все моменты наблюдения (equalhours=False).

Поскольку выбросы за один момент наблюдения при различных параметрах поиска могут существовать одновременно в нескольких метеостанциях, мы сформируем список координат выбросов и выведем их соответствующие значения в отдельную серию

```
In [45]: start_time = time.time() # для замера времени выполнения кода

# Нельзя удалять NaN в поле field_out непосредственно в df_tmp71 (Это приведёт к некорректным временным рядам!)
# Удалим NaN в столбце "field_out" в результирующем df

# Определим серию для записи списков с данными выбросов, индекс равен общему индексу архивов, тип данных - объект,
# название серии - будущее имя соответствующего столбца в DF
ser_time_out7D = pd.Series(index = df_tmp71.index, dtype='object', name="time_out_7D")

for elem in df_tmp71.dropna(subset=["error_at"]).error_at.tolist(): # поэлементно в списке значений столбца error_at
    # присваиваем элементу серии по соответствующему datetime индексу значение - пустой список
    ser_time_out7D.at[elem[0]] = []

for station in elem[1]: # по метеостанциям, указанным в списке (2й элемент кортежа error_at)
    # Определяем вербальные координаты ячеек с выбросами: TimeStamp и название столбца, соответствующего станции:
    idxs = (elem[0], PARAMETER71+'#'+station)

    # Вызываем функцию time_outliers с обозначенными выше параметрами
    val_func = time_outliers(df=df_tmp71,
                             # В качестве серии row_ передаём серию из одного значения: на момент наблюдения,
                             # где индекс серии соответствует соответствует названию столбца (idxs[1])
                             row_=df_tmp71.loc[idxs[0]][df_tmp71.loc[idxs[0]].index == idxs[1]],
                             td_symbol_='D',
                             td_quant_='3',
                             equal_hours_=True,
                             method_='norm',
                             criterium_=0.97,
                             inclusive_=False)

    # Присваиваем элементу серии по соответствующему datetime индексу результат вызова функции (список)
    if isinstance(val_func, float): # Если time_outliers вернула NaN (то есть значение float, то ничего не делаем
```

```

    pass
else: # Иначе, если time_outliers вернула список
    list_out = ser_time_out7D.at[idxs[0]] # Получаем список, находящийся в серии по индексу
    list_out = list_out + val_func # Расширяем его за счёт вывода функции (конкатенация)
    ser_time_out7D.at[idxs[0]] = list_out # Записываем в серию обновлённый список

#         ser_time_out7D.at[idxs[0]]

end_time = time.time()
elapsed_time = end_time - start_time
time_format = time.strftime("%H:%M:%S", time.gmtime(elapsed_time))
print(f"На выполнение кода ушло: {time_format}")

```

На выполнение кода ушло: 00:01:20

```

In [46]: start_time = time.time() # для замера времени выполнения кода

# Нельзя удалять NaN в поле field_out непосредственно в df_tmp71 (Это приведёт к некорректным временным рядам!)
# Удалим NaN в столбце "field_out" в результирующем df

# Определим серию для записи списков с данными выбросов, индекс равен общему индексу архивов, тип данных - объект,
# название серии - будущее имя соответствующего столбца в DF
ser_time_out48H = pd.Series(index = df_tmp71.index, dtype='object', name="time_out_48H")

for elem in df_tmp71.dropna(subset=["error_at"]).error_at.tolist(): # поэлементно в списке значений столбца error_at
    # присваиваем элементу серии по соответствующему datetime индексу значение - пустой список
    ser_time_out48H.at[elem[0]] = []

    for station in elem[1]: # по метеостанциям, указанным в списке (2й элемент кортежа error_at)
        # Определяем вербальные координаты ячеек с выбросами: TimeStamp и название столбца, соответствующего станции:
        idxs = (elem[0], PARAMETER71+'#'+station)

        # Вызываем функцию time_outliers с обозначенными выше параметрами
        val_func = time_outliers(df=df_tmp71,
                                # В качестве серии row_ передаём серию из одного значения: на момент наблюдения,
                                # где индекс серии соответствует соответствует названию столбца (idxs[1])
                                row_=df_tmp71.loc[idxs[0]][df_tmp71.loc[idxs[0]].index == idxs[1]],
                                td_symbol_='H',
                                td_quant_='24',
                                equal_hours_=False,
                                method_='norm',
                                criterium_=0.97,
                                inclusive_=False)

```

```
# Присваиваем элементу серии по соответствующему datetime индексу результат вызова функции (список)
if isinstance (val_func, float): # Если time_outliers вернула NaN (то есть значение float, то ничего не делаем
    pass
else: # Иначе, если time_outliers вернула список
    list_out = ser_time_out48H.at[idxs[0]] # Получаем список, находящийся в серии по индексу
    list_out = list_out + val_func # Расширяем его за счёт вывода функции (контакенация)
    ser_time_out48H.at[idxs[0]] = list_out # Записываем в серию обновлённый список

#         ser_time_out7D.at[idxs[0]]

end_time = time.time()
elapsed_time = end_time - start_time
time_format = time.strftime("%H:%M:%S", time.gmtime(elapsed_time))
print(f"На выполнение кода ушло: {time_format}")
```

На выполнение кода ушло: 00:00:39

```
In [47]: # Индекс серии совпадает с индексом всех архивов, а значения в серии упорядочены по этому индексу.
# Поэтому произведём объединение по индексу
df_tmp71 = (df_tmp71
            .merge(ser_time_out7D, left_index=True, right_index=True) # производим объединение
            )
df_tmp71 = (df_tmp71
            .merge(ser_time_out48H, left_index=True, right_index=True)
            )
```

```
In [48]: df_tmp71.head()
```

Out[48]:

	Visibility#V_Volochek	Visibility#Staritsa	Visibility#Kashyn	Visibility#Tver	Visibility#Klin	Visibility#Dmitrov	Visibility#Volokolamsk	Visibility#M
2022-06-09 21:00:00	20.0	10.0	20.0	10.0	NaN	NaN	10.0	
2022-06-09 18:00:00	20.0	10.0	20.0	10.0	NaN	NaN	10.0	
2022-06-09 15:00:00	20.0	10.0	20.0	10.0	NaN	NaN	10.0	
2022-06-09 12:00:00	20.0	20.0	20.0	10.0	NaN	NaN	10.0	
2022-06-09 09:00:00	20.0	20.0	20.0	10.0	10.0	NaN	10.0	



Заменим пустые списки в столбце time\_out\_48H и time\_out7D на NaN

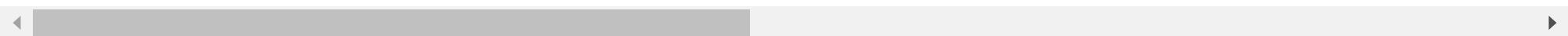
```
In [49]: df_tmp71.time_out_48H = df_tmp71.time_out_48H.apply(lambda x: np.nan if x == [] else x)
df_tmp71.time_out_7D = df_tmp71.time_out_7D.apply(lambda x: np.nan if x == [] else x)
```

```
In [50]: # Выводим рандомные строки из df_tmp71, удалив NaN в столбцах time_out48H и time_out7D
df_tmp71.dropna(subset=['time_out_7D','time_out_48H'], how='all').sample(5, random_state=56)
```

Out[50]:

	Visibility#V_Volochek	Visibility#Staritsa	Visibility#Kashyn	Visibility#Tver	Visibility#Klin	Visibility#Dmitrov	Visibility#Volokolamsk	Visibility#M
--	-----------------------	---------------------	-------------------	-----------------	-----------------	--------------------	------------------------	--------------

<b>2017-12-01 09:00:00</b>	10.0	10.0	50.0	10.0	10.0	10.0	10.0	10.0
<b>2021-07-05 03:00:00</b>	10.0	10.0	10.0	10.0	NaN	10.0	10.0	10.0
<b>2016-11-09 12:00:00</b>	20.0	10.0	50.0	10.0	10.0	10.0	10.0	10.0
<b>2020-11-30 09:00:00</b>	10.0	10.0	50.0	10.0	10.0	10.0	10.0	4.0
<b>2015-03-22 12:00:00</b>	10.0	10.0	18.0	10.0	10.0	10.0	10.0	10.0



### Применимость критериев ошибочных значений

Рассмотрим полученные данные об аномальных значениях.

In [51]:

```
# Определяем столбец error_48H_at и error_7D_at
# Вторым элементом кортежа в error_at является СПИСОК станций, по которым найдены выбросы,

df_tmp71 = (df_tmp71
    .assign(error_48H_at =
        df_tmp71.dropna(subset=["time_out_48H"])
        .apply(lambda x: # Вычленим DateTime index и название станции с выбросом
               # пропустим NaN, проверив, является ли x.field_out списком
               (x.name,
                stations_from_outliers(
                    row=x,
                    column_name_= 'time_out_48H', # используем выбросы во временном окне
                    error_48H_at=error_48H_at,
                    error_7D_at=error_7D_at)))
    .reset_index()
    .drop(['time_out_48H'], axis=1))
```

```
        param_=PARAMETER71)
    ) if isinstance(x.field_out, list) else np.nan,
    axis=1
)
)
.assign(error_7D_at =
    df_tmp71.dropna(subset=["time_out_7D"])
    .apply(lambda x: # Вычленим DateTime index и название станции с выбросом
           # пропустим NaN, проверив, является ли x.field_out списком
           (x.name,
            stations_from_outliers(
                row_=x,
                column_name_= 'time_out_7D', # используем выбросы во временном окне
                param_=PARAMETER71)
           ) if isinstance(x.field_out, list) else np.nan,
           axis=1
)
)
)
```

```
In [52]: df_tmp71.dropna(subset=["error_48H_at", "error_7D_at"], how='all')
```

Out[52]:

	Visibility#V_Volochev	Visibility#Staritsa	Visibility#Kashyn	Visibility#Tver	Visibility#Klin	Visibility#Dmitrov	Visibility#Volokolamsk	Visibility#M
<b>2022-05-10 21:00:00</b>	10.0	10.0	10.0	10.0	10.0	10.0	10.0	10.0
<b>2022-04-30 21:00:00</b>	10.0	10.0	10.0	10.0	10.0	10.0	10.0	10.0
<b>2022-04-24 18:00:00</b>	10.0	10.0	10.0	10.0	10.0	10.0	10.0	10.0
<b>2022-03-28 03:00:00</b>	10.0	10.0	10.0	10.0	10.0	Nan	10.0	10.0
<b>2022-03-20 18:00:00</b>	20.0	50.0	20.0	10.0	10.0	10.0	10.0	10.0
...	...	...	...	...	...	...	...	...
<b>2006-11-24 15:00:00</b>	6.0	4.0	3.5	2.0	4.0	5.0	1.9	
<b>2006-03-30 09:00:00</b>	26.0	2.0	15.0	0.5	0.5	1.2	0.2	
<b>2006-02-19 21:00:00</b>	12.0	10.0	50.0	10.0	10.0	8.0	10.0	
<b>2005-11-24 15:00:00</b>	Nan	10.0	40.0	10.0	10.0	10.0	2.8	
<b>2005-11-08 21:00:00</b>	16.0	10.0	50.0	10.0	10.0	10.0	1.0	

719 rows × 18 columns

Найдём общие выбросы как в поле метеостанций, так и во временном окне. (У нас могут быть случаи, когда при проверке выброса в поле метеостанций, выбросов для того же значения во временном окне не обнаруживается).

Для контроля, найдём пересечение списков в столбцах error\_at, error\_48H\_at, error\_7D\_at и выведем его в отдельный столбец cross\_check

```
In [53]: df_tmp71 = df_tmp71.assign(  
    cross_check = df_tmp71  
    .dropna(subset=["error_at", "error_48H_at", "error_7D_at"])  
    .apply(  
        lambda x: list(set(x.error_at[1]) & set(x.error_48H_at[1]) & set(x.error_7D_at[1])),  
        axis=1  
    )  
    )
```

```
In [54]: df_tmp71.dropna(subset=["cross_check"]).sample(7, random_state=56)
```

Out[54]:

	Visibility#V_Volochek	Visibility#Staritsa	Visibility#Kashyn	Visibility#Tver	Visibility#Klin	Visibility#Dmitrov	Visibility#Volokolamsk	Visibility#M
<b>2020-08-24 09:00:00</b>	10.0	10.0	10.0	4.0	10.0	10.0	10.0	10.0
<b>2022-03-11 12:00:00</b>	20.0	50.0	20.0	10.0	10.0	10.0	10.0	10.0
<b>2022-04-30 21:00:00</b>	10.0	10.0	10.0	10.0	10.0	10.0	10.0	10.0
<b>2019-07-31 03:00:00</b>	10.0	10.0	10.0	4.0	10.0	10.0	10.0	10.0
<b>2018-01-21 21:00:00</b>	50.0	10.0	10.0	10.0	4.0	4.0	4.0	4.0
<b>2009-10-24 09:00:00</b>	12.0	2.0	5.0	10.0	10.0	6.0	6.0	6.0
<b>2022-03-20 12:00:00</b>	20.0	50.0	20.0	10.0	10.0	10.0	10.0	10.0



In [55]:

```
# Подсчитаем количество выбросов в поле метеостанций
count_field_out = (df_tmp71
    .error_at # по столбцу error_at
    .dropna()
    .apply(lambda x: len(x[1]) # вычислим длины списков с перечнем метеостанций с выбросами
          )
    ).sum() # просуммируем их в общий итог

# Подсчитаем из них количество выбросов во временном окне
count_time_out24H = (df_tmp71
    .error_48H_at # по столбцу error_48H_at
    .dropna()
    .apply(lambda x: len(x[1]) # вычислим длины списков с перечнем метеостанций с выбросами
          )
```

```

        )
    ).sum() # просуммируем их в общий итог
# Подсчитаем из них количество выбросов во временном окне
count_time_out7D = (df_tmp71
    .error_7D_at # по столбцу error_7D_at
    .dropna()
    .apply(lambda x: len(x[1]) # вычислим длины списков с перечнем метеостанций с выбросами
          )
    ).sum() # просуммируем их в общий итог

# Подсчитаем из них количество выбросов в контрольном столбце
count_cross_out = (df_tmp71
    .cross_check # по столбцу double_error_at
    .dropna()
    .apply(lambda x: len(x) # вычислим длины списков с перечнем метеостанций с выбросами
          )
    ).sum() # просуммируем их в общий итог

print(f'В поле метеостанций выявлено аномальных значений: {count_field_out}, из них:\n'
      f'Подтверждается аномалиями в промежутке +/- 24 часа по всем моментам наблюдения: '
      f'{count_time_out24H}\n'
      f'Подтверждается аномалиями в промежутке +/- 3 дня на один и тот же час наблюдения: '
      f'{count_time_out7D}\n'
      f'Проверка: пересечение множеств выбросов в поле метеостанций и во временных окнах насчитывает '
      f'{count_cross_out} элемент(ов)')

```

В поле метеостанций выявлено аномальных значений: 13160,  
из них:  
Подтверждается аномалиями в промежутке +/- 24 часа по всем моментам наблюдения: 313  
Подтверждается аномалиями в промежутке +/- 3 дня на один и тот же час наблюдения: 509  
Проверка: пересечение множеств выбросов в поле метеостанций и во временных окнах насчитывает 103 элемент(ов)

### **Определим конечный критерий ошибочных значений:**

1. Аномальное значение выявлено в поле метеостанций И ПРИ ЭТОМ
  - A. Аномальное значение выявлено во временном ряду в промежутке +/- 24 часа по всем моментам наблюдения И
  - B. Аномальное значение выявлено во временном ряду в промежутке +/- 3 дня по одному и тому же часу наблюдения

### **Удалим (заменим на NaN) все ошибочные значения**

Выведем только строки с аномальными значениями, которые в соответствии с указанными выше критериями являются ошибками

```
In [56]: df_tmp71.dropna(subset=['cross_check'])
```

Out[56]:

	Visibility#V_Volochek	Visibility#Staritsa	Visibility#Kashyn	Visibility#Tver	Visibility#Klin	Visibility#Dmitrov	Visibility#Volokolamsk	Visibility#Moscow
2021-08-19 15:00:00	20.0	20.0	20.0	10.0	10.0	10.0	10.0	10.0
2021-07-22 18:00:00	20.0	50.0	20.0	10.0	10.0	10.0	10.0	10.0
2021-07-13 03:00:00	10.0	0.2	10.0	10.0	NaN	10.0	10.0	10.0
2021-07-05 03:00:00	10.0	10.0	10.0	10.0	NaN	10.0	10.0	10.0
2021-06-22 00:00:00	10.0	10.0	10.0	10.0	10.0	10.0	10.0	NaN
2021-06-12 12:00:00	10.0	10.0	4.0	10.0	10.0	10.0	10.0	10.0
2021-05-27 18:00:00	20.0	50.0	20.0	10.0	10.0	10.0	10.0	10.0
2021-05-27 15:00:00	20.0	50.0	20.0	10.0	10.0	10.0	10.0	10.0
2021-05-23 12:00:00	20.0	50.0	20.0	10.0	10.0	10.0	10.0	10.0
2021-05-23 09:00:00	20.0	50.0	20.0	10.0	10.0	10.0	10.0	10.0
2021-05-20 03:00:00	10.0	10.0	10.0	10.0	NaN	10.0	10.0	10.0



	Visibility#V_Volochek	Visibility#Staritsa	Visibility#Kashyn	Visibility#Tver	Visibility#Klin	Visibility#Dmitrov	Visibility#Volokolamsk	Visibility#Moscow
2020-12-22 12:00:00	20.0	10.0	50.0	10.0	10.0	10.0	10.0	4.0
2020-09-04 06:00:00	10.0	10.0	20.0	50.0	10.0	10.0	10.0	10.0
2020-08-24 09:00:00	10.0	10.0	10.0	4.0	10.0	10.0	10.0	10.0
2020-07-03 00:00:00	10.0	10.0	10.0	4.0	10.0	10.0	10.0	10.0
2020-05-13 03:00:00	10.0	10.0	10.0	4.0	NaN	10.0	10.0	10.0
2020-04-29 06:00:00	10.0	10.0	10.0	4.0	NaN	10.0	10.0	10.0
2020-04-24 06:00:00	10.0	10.0	20.0	10.0	NaN	10.0	10.0	10.0
2020-04-10 00:00:00	10.0	10.0	20.0	10.0	10.0	10.0	10.0	10.0
2020-04-02 03:00:00	10.0	10.0	10.0	10.0	10.0	10.0	10.0	10.0
2020-01-04 21:00:00	0.5	10.0	10.0	10.0	10.0	10.0	10.0	10.0
2019-10-14 18:00:00	10.0	10.0	10.0	1.0	10.0	10.0	10.0	10.0

	Visibility#V_Volochek	Visibility#Staritsa	Visibility#Kashyn	Visibility#Tver	Visibility#Klin	Visibility#Dmitrov	Visibility#Volokolamsk	Visibility#Mtsensk
<b>2019-10-14 15:00:00</b>	10.0	10.0	10.0	2.0	10.0	10.0	10.0	10.0
<b>2019-07-31 03:00:00</b>	10.0	10.0	10.0	4.0	10.0	10.0	10.0	10.0
<b>2019-06-23 00:00:00</b>	10.0	4.0	20.0	10.0	10.0	10.0	10.0	10.0
<b>2019-05-25 21:00:00</b>	10.0	10.0	20.0	10.0	10.0	10.0	10.0	10.0
<b>2018-12-29 00:00:00</b>	10.0	10.0	50.0	10.0	10.0	10.0	10.0	10.0
<b>2018-12-28 21:00:00</b>	10.0	10.0	50.0	10.0	4.0	4.0	4.0	4.0
<b>2018-12-06 09:00:00</b>	20.0	10.0	4.0	10.0	4.0	4.0	4.0	4.0
<b>2018-11-18 00:00:00</b>	10.0	10.0	10.0	10.0	10.0	10.0	10.0	10.0
<b>2018-10-08 06:00:00</b>	10.0	10.0	10.0	10.0	10.0	4.0	10.0	10.0
<b>2018-09-28 09:00:00</b>	10.0	10.0	10.0	2.0	10.0	10.0	10.0	10.0
<b>2018-09-25 00:00:00</b>	10.0	10.0	10.0	2.0	10.0	10.0	10.0	10.0

	Visibility#V_Volochek	Visibility#Staritsa	Visibility#Kashyn	Visibility#Tver	Visibility#Klin	Visibility#Dmitrov	Visibility#Volokolamsk	Visibility#Moscow
2018-08-10 00:00:00	10.0	10.0	20.0	10.0	10.0	10.0	10.0	10.0
2018-06-10 03:00:00	10.0	10.0	20.0	10.0	10.0	10.0	10.0	10.0
2018-06-10 00:00:00	10.0	10.0	20.0	10.0	10.0	10.0	10.0	10.0
2018-06-08 09:00:00	10.0	10.0	20.0	10.0	10.0	10.0	10.0	10.0
2018-05-02 00:00:00	10.0	10.0	10.0	4.0	10.0	10.0	10.0	10.0
2018-04-28 00:00:00	10.0	10.0	20.0	10.0	10.0	10.0	10.0	10.0
2018-04-10 21:00:00	10.0	10.0	10.0	4.0	10.0	10.0	10.0	10.0
2018-02-26 03:00:00	10.0	10.0	20.0	10.0	10.0	10.0	10.0	10.0
2018-01-21 21:00:00	50.0	10.0	10.0	10.0	4.0	4.0	4.0	4.0
2018-01-16 06:00:00	10.0	10.0	0.5	10.0	10.0	10.0	10.0	10.0
2017-12-30 15:00:00	10.0	4.0	50.0	4.0	10.0	10.0	10.0	10.0

	Visibility#V_Volochek	Visibility#Staritsa	Visibility#Kashyn	Visibility#Tver	Visibility#Klin	Visibility#Dmitrov	Visibility#Volokolamsk	Visibility#M
<b>2017-12-30 12:00:00</b>	10.0	10.0	50.0	10.0	10.0	10.0	10.0	10.0
<b>2017-11-26 06:00:00</b>	10.0	10.0	10.0	10.0	10.0	10.0	10.0	10.0
<b>2017-02-04 00:00:00</b>	10.0	10.0	20.0	10.0	10.0	10.0	10.0	4.0
<b>2017-02-03 18:00:00</b>	10.0	10.0	20.0	4.0	10.0	10.0	10.0	4.0
<b>2017-01-29 15:00:00</b>	10.0	10.0	20.0	10.0	10.0	10.0	10.0	10.0
<b>2016-12-26 09:00:00</b>	10.0	20.0	50.0	10.0	10.0	10.0	10.0	10.0
<b>2016-12-12 09:00:00</b>	20.0	10.0	10.0	10.0	4.0	10.0	10.0	4.0
<b>2016-12-05 00:00:00</b>	65.0	10.0	20.0	4.0	4.0	10.0	10.0	4.0
<b>2016-12-02 21:00:00</b>	10.0	10.0	50.0	10.0	10.0	4.0	4.0	4.0
<b>2016-12-02 18:00:00</b>	10.0	10.0	50.0	10.0	10.0	4.0	4.0	4.0
<b>2016-12-02 15:00:00</b>	20.0	10.0	50.0	10.0	10.0	10.0	10.0	10.0

	Visibility#V_Volochek	Visibility#Staritsa	Visibility#Kashyn	Visibility#Tver	Visibility#Klin	Visibility#Dmitrov	Visibility#Volokolamsk	Visibility#Moscow
<b>2015-06-11 03:00:00</b>	10.0	10.0	20.0	10.0	10.0	10.0	10.0	10.0
<b>2015-06-04 03:00:00</b>	10.0	10.0	20.0	10.0	10.0	10.0	10.0	10.0
<b>2015-03-22 15:00:00</b>	10.0	10.0	15.0	10.0	10.0	10.0	10.0	10.0
<b>2015-03-22 12:00:00</b>	10.0	10.0	18.0	10.0	10.0	10.0	10.0	10.0
<b>2015-02-27 15:00:00</b>	20.0	10.0	50.0	10.0	10.0	10.0	10.0	4.0
<b>2014-12-31 15:00:00</b>	20.0	4.0	8.0	4.0	4.0	4.0	4.0	4.0
<b>2014-11-27 15:00:00</b>	10.0	4.0	50.0	2.0	4.0	4.0	10.0	
<b>2014-11-27 12:00:00</b>	10.0	1.0	50.0	1.0	2.0	4.0	4.0	
<b>2014-08-20 21:00:00</b>	10.0	10.0	12.0	10.0	10.0	10.0	10.0	
<b>2014-08-15 00:00:00</b>	10.0	10.0	12.0	10.0	10.0	10.0	10.0	
<b>2014-07-06 03:00:00</b>	10.0	10.0	7.0	10.0	10.0	10.0	10.0	

	Visibility#V_Volochek	Visibility#Staritsa	Visibility#Kashyn	Visibility#Tver	Visibility#Klin	Visibility#Dmitrov	Visibility#Volokolamsk	Visibility#M
<b>2014-07-02 18:00:00</b>	10.0	NaN	15.0	10.0	10.0	10.0	10.0	10.0
<b>2014-06-22 00:00:00</b>	10.0	10.0	15.0	10.0	10.0	10.0	10.0	10.0
<b>2014-06-09 00:00:00</b>	10.0	10.0	20.0	10.0	10.0	10.0	10.0	10.0
<b>2014-04-21 03:00:00</b>	10.0	10.0	20.0	10.0	10.0	10.0	10.0	10.0
<b>2014-04-21 00:00:00</b>	10.0	10.0	20.0	10.0	10.0	10.0	10.0	10.0
<b>2014-02-11 09:00:00</b>	4.0	1.0	50.0	1.0	2.0	4.0	2.0	2.0
<b>2014-02-04 06:00:00</b>	10.0	10.0	20.0	10.0	10.0	10.0	10.0	10.0
<b>2013-09-28 00:00:00</b>	50.0	10.0	12.0	10.0	4.0	10.0	10.0	10.0
<b>2013-06-04 06:00:00</b>	20.0	10.0	40.0	10.0	10.0	15.0	10.0	10.0
<b>2013-06-04 03:00:00</b>	10.0	10.0	40.0	10.0	10.0	12.0	10.0	10.0
<b>2013-05-30 00:00:00</b>	10.0	10.0	4.0	10.0	10.0	10.0	10.0	10.0

	Visibility#V_Volochek	Visibility#Staritsa	Visibility#Kashyn	Visibility#Tver	Visibility#Klin	Visibility#Dmitrov	Visibility#Volokolamsk	Visibility#M
2009-10-24 09:00:00	12.0	2.0	5.0	10.0	10.0	6.0		6.0
2009-01-27 03:00:00	14.0	10.0	50.0	4.0	10.0	9.0		2.0
2009-01-26 09:00:00	7.0	10.0	30.0	2.0	10.0	10.0		4.2
2006-11-24 15:00:00	6.0	4.0	3.5	2.0	4.0	5.0		1.9

```
In [57]: # Создадим список координат ячеек, содержащих значения, определённые как ошибки
list_error_coords = df_tmp71.dropna(subset=['cross_check']).error_at.tolist()
# list_error_coords
```

```
In [58]: # Восстановим исходное состояние df_tmp1
df_tmp71 = dict_df_parameters[param_df_name].copy(deep=True)
```

```
In [59]: for error_coords in list_error_coords: # по списку координат ошибочных значений
    for station in error_coords[1]: # перечень метеостанций в каждом списке координат ошибочных значений для станций
        # Преобразуем координату столбца и присваиваем ячейке NaN
        df_tmp71.at[error_coords[0], PARAMETER71+'#'+station] = np.nan
# df_tmp71
```

```
In [60]: # Проверим, все ли ошибки заменены на NaN
# Количество нeNaN значений в df_tmp71 по координатам, указанным в списках list_error_coords и list_error_at

counter_notna1 = 0 # счётчик нeNaN значений
for error_coords in list_error_coords: # по списку координат ошибочных значений
    for station in error_coords[1]: # перечень метеостанций в каждом списке координат ошибочных значений для станций
        # подсчитаем количество нeNaN значений и прибавим их к счётчику нeNaN значений.
        counter_notna1 += np.sum(pd.notna(df_tmp71.at[error_coords[0], PARAMETER71+'#'+station]))
print(f'По результатам удаления выявленных аномальных выбросов осталось значений: {counter_notna1}')

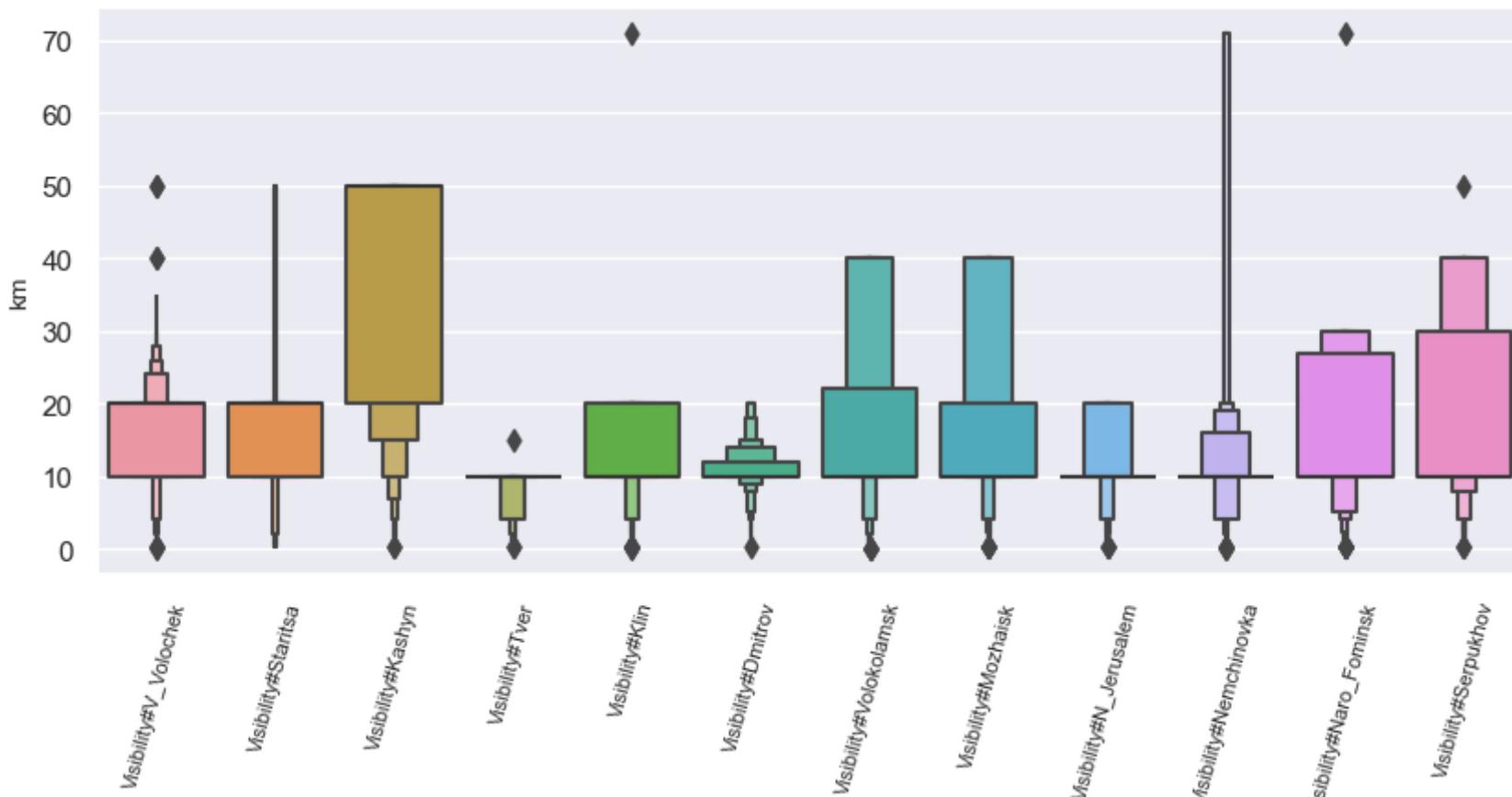
# df_tmp71
```

По результатам удаления выявленных аномальных выбросов осталось значений: 0

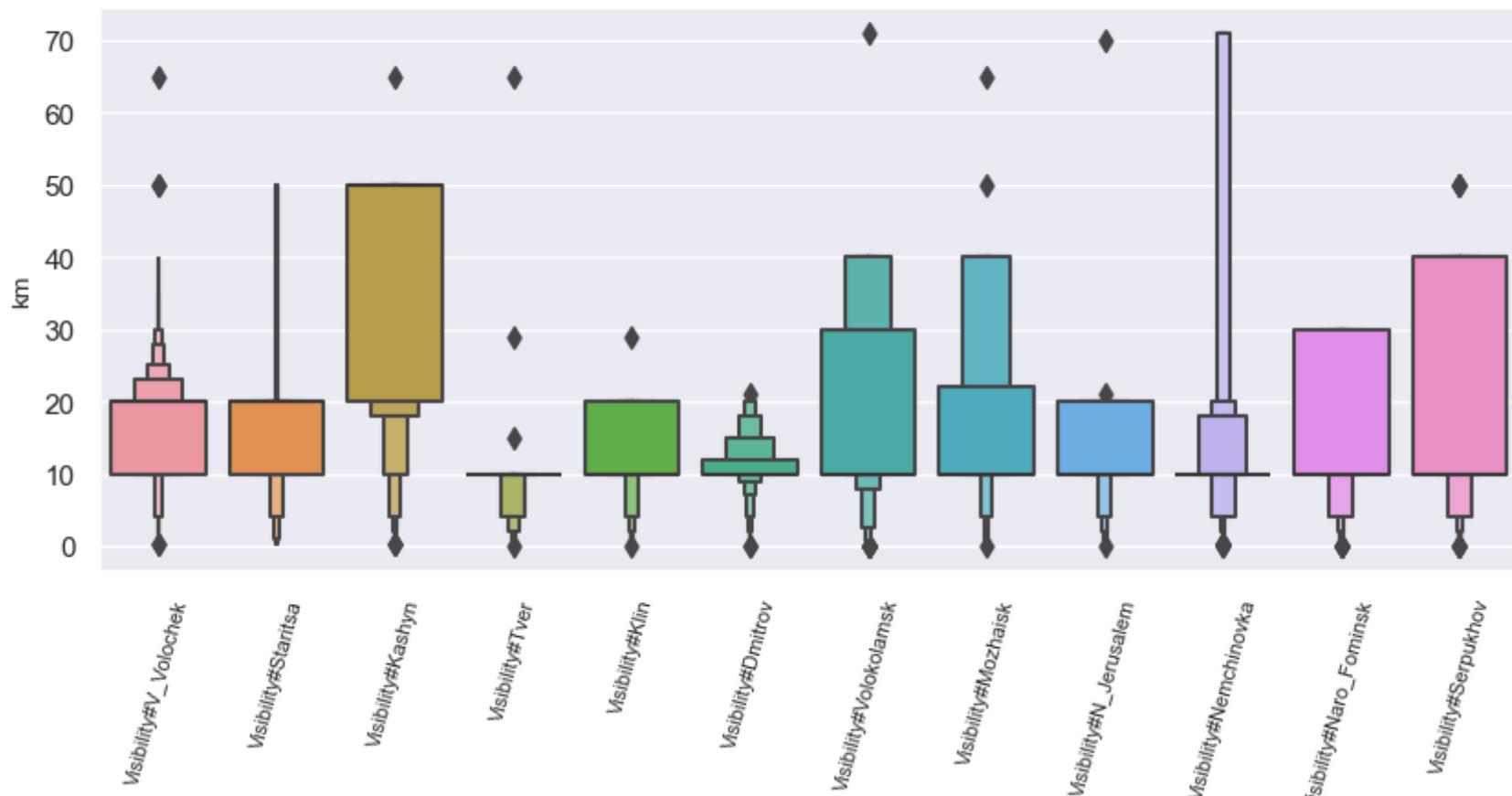
### Визуализируем архив дальности видимости (Visibility) по сезонам после удаления аномальных значений

```
In [61]: # В цикле выведем графики давления по метеостанциями в зависимости от сезона
# используем функцию создания масок климатических сезонов
for season_name, season_mask in season_masks(df_tmp71).items():
    fig, ax = plt.subplots(figsize=(10, 4))
    g = sns.boxenplot(data=df_tmp71[season_mask],
                       ax=ax)
    dummy = plt.xticks(rotation=75, size=8)
    dummy = g.set_ylabel('km', size=10)
    dummy = g.set_title(f'Распределение значений {PARAMETER71} в разрезе метеостанций:\n'
                        f'{season_name}')
plt.show()
```

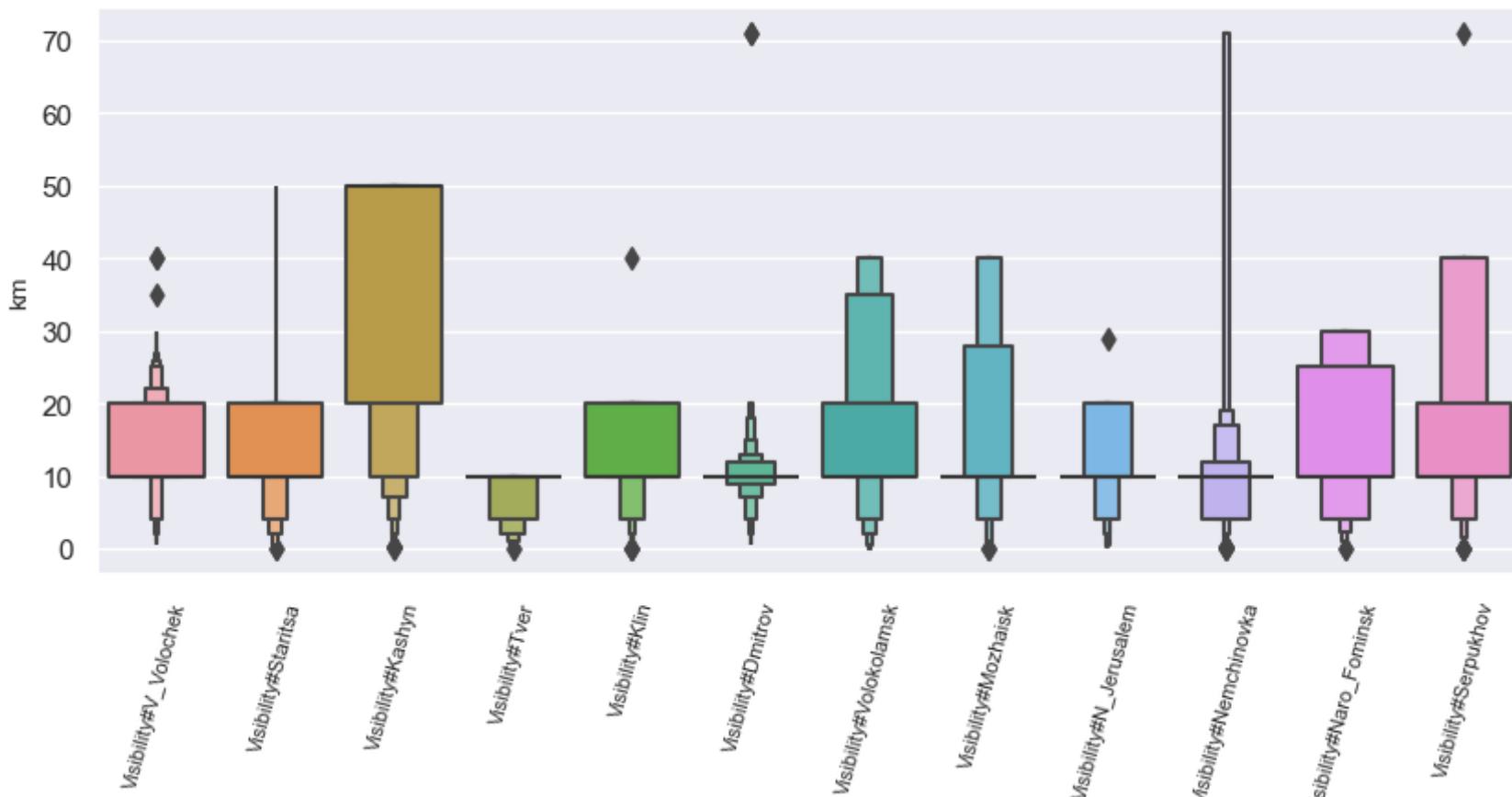
## Распределение значений Visibility в разрезе метеостанций: spring



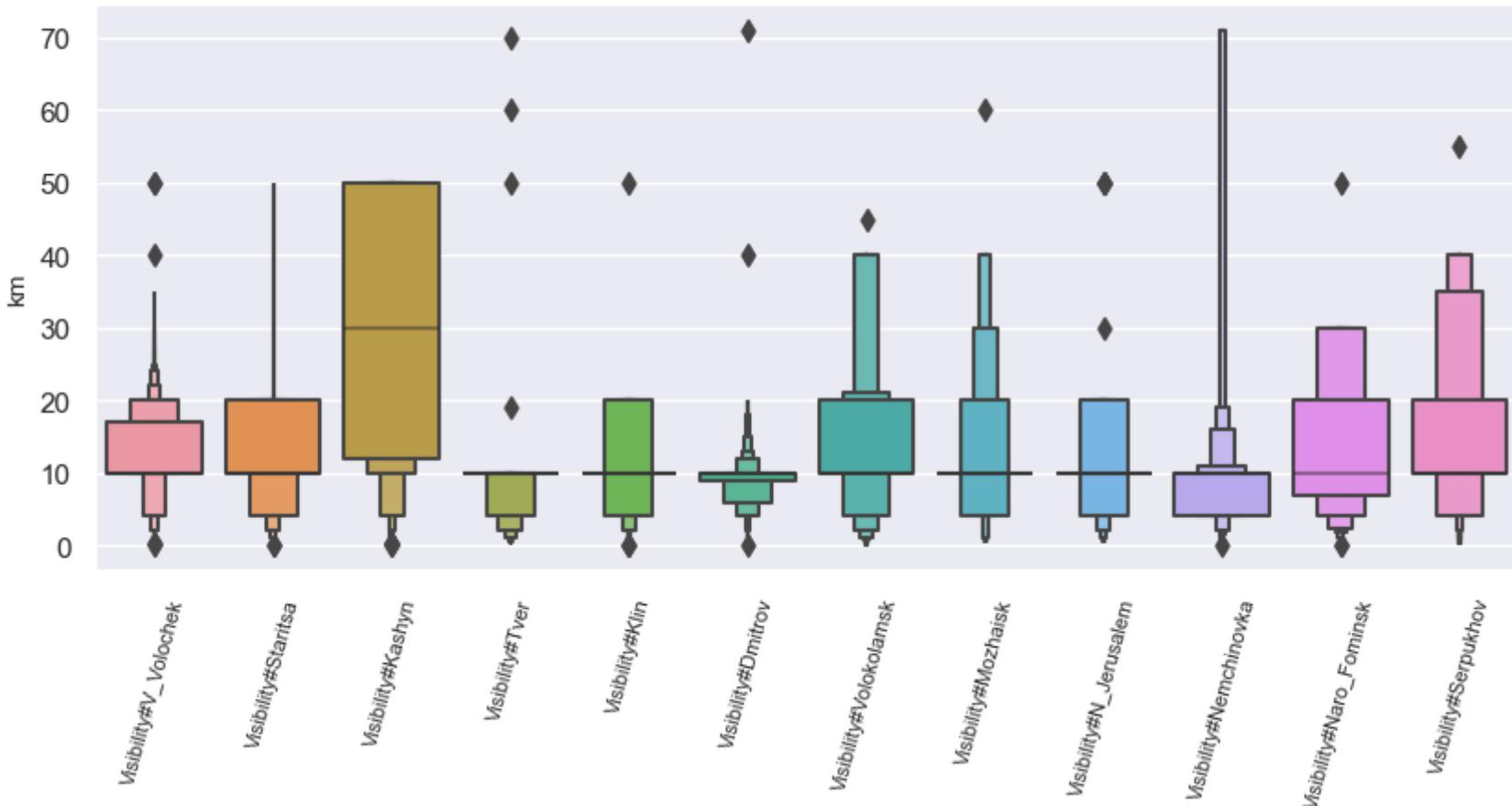
Распределение значений Visibility в разрезе метеостанций:  
summer



Распределение значений Visibility в разрезе метеостанций:  
autumn



## Распределение значений Visibility в разрезе метеостанций: winter



Как видно из графиков, сильный разброс видимости между метеостанциями остался. Попробуем всё-таки восстановить пропущенные значения и воссоздать архив для локации пос. Чашниково, исходя из подобия близлежащим метеостанциям.

Выведем минимальное и максимальное значения, а также значение медианы и средней для всего DF.

```
In [62]: print(f'Минимальное значение: {np.nanmin(df_tmp71)},\n'
      f'Максимальное значение: {np.nanmax(df_tmp71)},\n'
      f'Средняя: {np.nanmean(df_tmp71)},\n'
      f'Медиана: {np.nanmedian(df_tmp71)})')
```

Минимальное значение: 0.025,  
Максимальное значение: 71.0,  
Средняя: 15.110390632081947,  
Медиана: 10.0

### Сохраним очищенные данные в файл параметров

```
In [63]: # # Определённые выше пути к файлам данных:  
# path  
# raw_path1  
# raw_path2  
  
# Создадим новые значения директорий  
clean_path = f'{path}clean/'  
  
makedirs(clean_path, exist_ok=True)  
  
# Запишем текущие данные в файл  
print('df_'+PARAMETER71 + '.csv ->', end=' ')  
dict_df_parameters['df_'+PARAMETER71].to_csv(  
    path_or_buf=f'{clean_path}df_{PARAMETER71}.csv'  
)  
print('DONE!')  
  
df_Visibility.csv -> DONE!
```

### 7.1.2. Восстановление "сплошных" NaN методом средней между соседними моментами наблюдения для показателя метеорологической дальности видимости

Модели пространственной экстраполяции не могут экстраполировать значения в рамках одного момента наблюдения, если значения во всех точках наблюдения неизвестны. Есть ли такие случаи в архиве температуры почвы (в периоды, когда такие наблюдения проводились)?

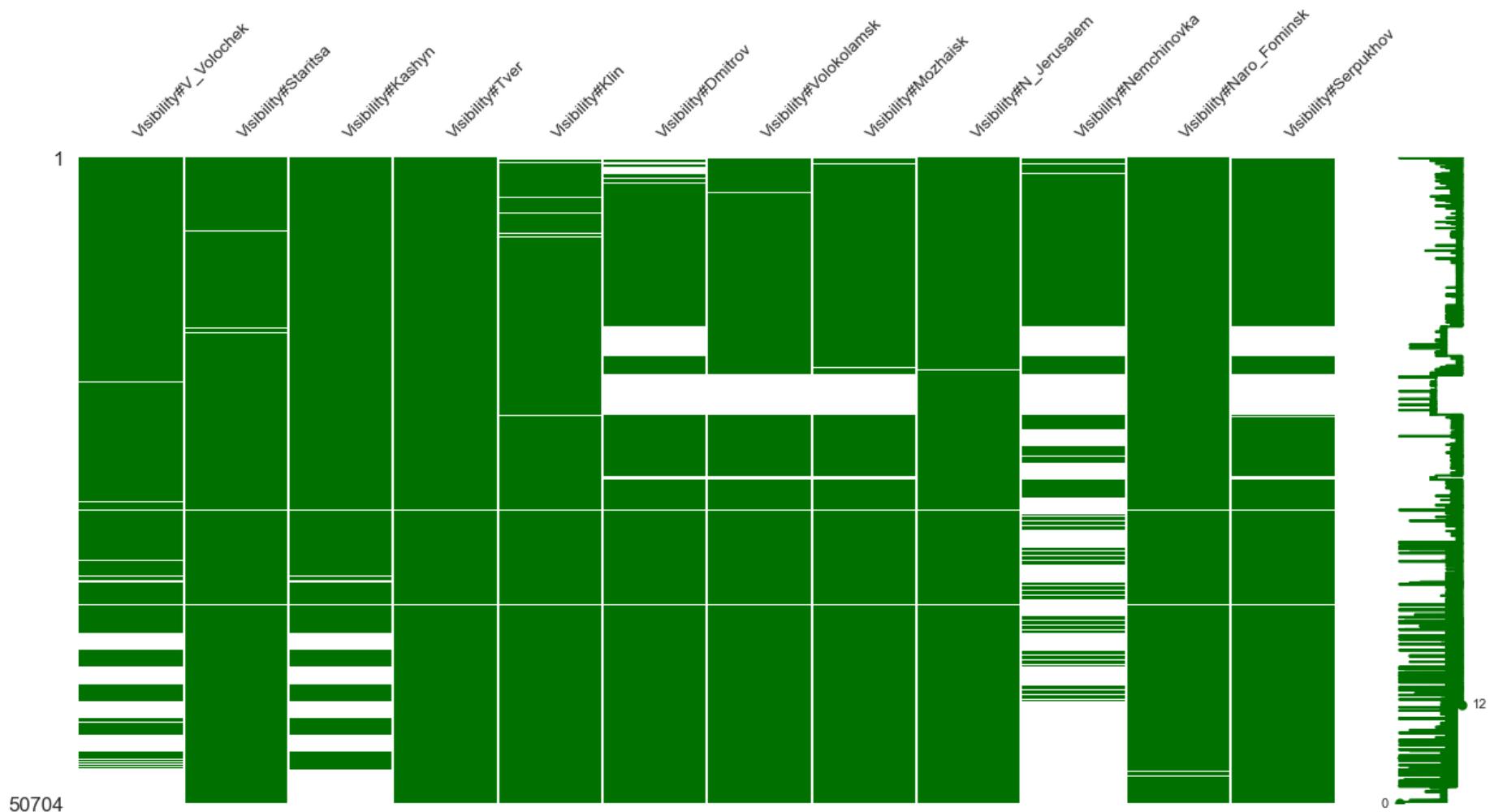
```
In [64]: # Подсчитаем количество строк со "сплошными" NaN в df_tmp71  
# построчно подсчитаем сумму количества NaN,  
# и если оно количеству столбцов в DF (boolean), подсчитаем через сумму количество таких строк  
all_nans_count = sum(df_tmp71.apply(lambda x: sum(x.isna()), axis=1)==len(df_tmp71.keys()))  
print(f'Количество строк со сплошными NaN равно {all_nans_count}')
```

Количество строк со сплошными NaN равно 158

Визуализируем структуру пропущенных данных о дальности видимости.

```
In [65]: msno.matrix(df_tmp71,  
                   color=(0, 100/225, 0),  
                   figsize=(15, 7),  
                   fontsize=10); # выводим график из библиотеки "missingno"  
plt.show()
```

```
Out[65]: <AxesSubplot: >
```



Очевидно такие строки нужно заполнить до пространственной экстраполяции. Представляется, что лучшим способом заполнения пропущенных строк будет средняя по тем же часам наблюдения между двумя соседними днями

```
In [66]: # Создадим список DateTime индексов строк со сплошными NaN  
# Выбираем из датафрейма строки, где количество NaN равно количеству столбцов - то есть сплошные NaN  
list_time_total_nans = df_tmp71[df_tmp71.apply(lambda x: sum(x.isna()), axis=1)==len(df_tmp71.keys())].index.tolist()
```

```
In [67]: # По списку  
for idx in list_time_total_nans[:-1]: # кроме самого раннегоTimeStamp, для которого нельзя вычислить start_date  
    # определяем начало и конец временного интервала  
    start_time = idx - pd.Timedelta('3H')  
    end_time = idx + pd.Timedelta('3H')  
    while sum(df_tmp71.loc[start_time].notna()) == 0: # Пока ряд от start_time тоже состоит из сплошных NaN  
        start_time = start_time - pd.Timedelta('3H') # Уменьшаем start_time на 3 часа  
    while sum(df_tmp71.loc[end_time].notna()) == 0: # Пока ряд от end_time тоже состоит из сплошных NaN  
        end_time = end_time + pd.Timedelta('3H') # Увеличиваем end_time на 3 часа  
  
    # по ряду сплошных NaN заменяем NaN на средние значения их соседних двух рядов  
    for label in df_tmp71.loc[idx].index:  
        df_tmp71.at[idx, label] = np.mean([df_tmp71.at[start_time, label], df_tmp71.at[end_time, label]])
```

```
In [68]: # Подсчитаем количество строк со "сплошными" NaN в df_tmp71  
# построчно подсчитаем сумму количества NaN,  
# и если оно количеству столбцов в DF (boolean), подсчитаем через сумму количество таких строк  
all_nans_count = sum(df_tmp71.apply(lambda x: sum(x.isna()), axis=1)==len(df_tmp71.keys()))  
print(f'Количество строк со сплошными NaN равно {all_nans_count}')
```

Количество строк со сплошными NaN равно 1

Всё верно, это самая начальная строка. Она должна остаться

Поскольку в процессе удаления ошибочных значений удалялись и единичные значения в строках, и часть строк могла превратиться в сплошные NaN. Проверим, сколько единичных значений исправлено методом восстановления сплошных NaN

```
In [69]: # Проверим, все ли ошибки заменены на NaN  
# Количество нeNaN значений в df_tmp71 по координатам, указанным в списках list_error_coords и list_error_at  
  
counter_notna1 = 0 # счётчик нeNaN значений  
for error_coords in list_error_coords: # по списку координат ошибочных значений  
    for station in error_coords[1]: # перечень метеостанций в каждом списке координат ошибочных значений для станций
```

```
# подсчитаем количество неNaN значений и прибавим их к счётчику неNaN значений.
counter_notna1 += np.sum(pd.notna(df_tmp71.at[error_coords[0], PARAMETER71+'#'+ station]))
print(f'По результатам удаления выявленных аномальных выбросов осталось значений: {counter_notna1}')

# df_tmp71
```

По результатам удаления выявленных аномальных выбросов осталось значений: 0

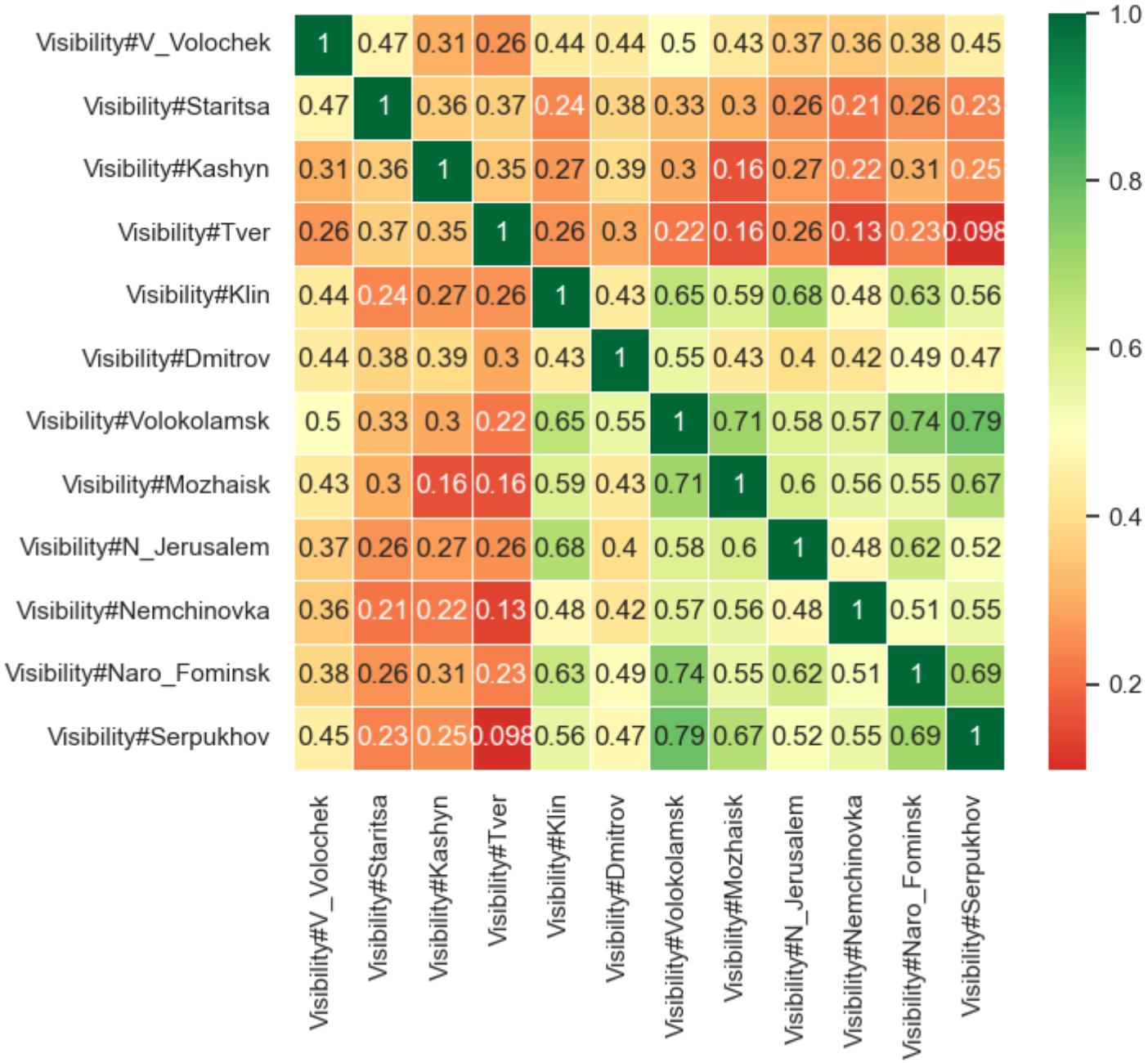
Получившееся значение показывает количество исправленных ошибочных значений, там где был применён метод восстановления сплошных NaN. В данном случае, исправлений индивидуальных ошибочных значений при заполнении сплошных NaN не произошло.

### 7.1.3. Подбор модели для восстановления пропущенных и удалённых значений показателя температуры почвы, а также для моделирования значений для искомой точки

Выведем корреляционную матрицу для показателя метеорологической дальности видимости

```
In [70]: df_corr_matrix = df_tmp71.corr()
fig, ax = plt.subplots(figsize=(7, 6))
ax = sns.heatmap(data=df_corr_matrix, # Выводим график heatmap
                  cmap='RdYlGn', # matplotlib цветовая палитра (желательно расходящаяся)
                  center=0.5, # центральное значение (для расходящейся цветовой карты):
                  # заметная связь по шкале Чеддока
                  annot=True, # определим подписи значений
                  linewidths=0.5 # толщина разделяющих линий между ячейками
                 )
dummy = plt.suptitle(f'Корреляция параметра {PARAMETER71} между метеостанциями после удаления аномалий ')
plt.show()
```

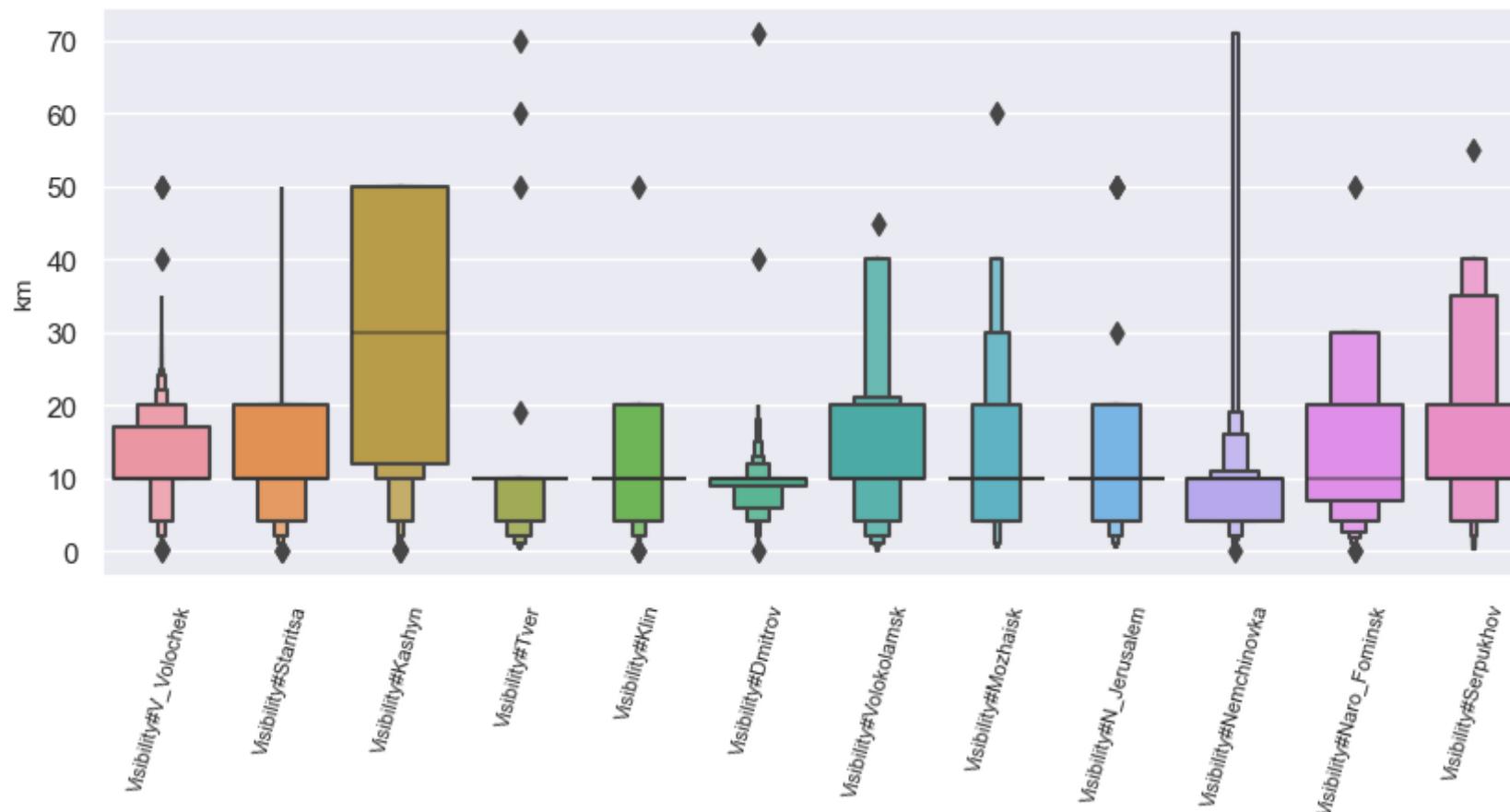
## Корреляция параметра Visibility между метеостанциями после удаления аномалий



Еще раз посмотрим на распределение значений по каждой станции

```
In [71]: # выведем графики давления по метеостанциями
fig, ax = plt.subplots(figsize=(10, 4))
g = sns.boxenplot(data=df_tmp71[season_mask],
                   ax=ax)
dummy = plt.xticks(rotation=75, size=8)
dummy = g.set_ylabel('km', size=10)
dummy = g.set_title(f'Распределение значений {PARAMETER71} в разрезе метеостанций:\n')
plt.show()
```

Распределение значений Visibility в разрезе метеостанций:



Корреляция по показателю метеорологической дальности видимости между метеостанциями достаточно мала, а распределение данных сильно разнится между метеостанциями, имеются случаи сильного смещения в разные стороны. Поэтому не следует ожидать слишком высоких метрик качества для модели предсказания данного показателя.

Как видно из графиков, распределение метеорологической видимости для метеостанции *Кашин* принципиально отличается от других. Поэтому для определения модели пространственной экстраполяции, данные по ней имеет смысл удалить из датасета.

Так построение датасета для обучения моделей пространственной экстраполяции и моделей регрессии будет различным по форме, рассмотрим эти модели отдельно.

### Подготовим данные для моделей пространственной экстраполяции

Создадим набор данных для обучения и валидации модели IDW.

1. Используем данные в df\_tmp71.
2. Уберём все строки с NaN.
3. Выделим рандомно массив известных значений для разных метеостанций и разных моментов наблюдения - он потребуется для разделения на обучающую и валидационную часть и для расчёта метрик качества.

Создадим датафрейм за исключением метеостанции *Кашин*, удалим все NaN и применим трансформацию, так как во входных данных всё еще есть знаительные выбросы

Учитывая особенность датасета: X представлен координатами значений, а y - самими значениями, вычисляемыми из других значений построчно, - нам необходимо трансформировать весь датасет (то есть, все значения), иначе функция IDW не будет работать.

```
In [72]: # Создаём датафрейм исходных данных для проверки работы моделей
df_true = df_tmp71.drop(columns=[PARAMETER71+"#"+"Kashyn"]).dropna(how='any')
```

```
In [73]: # Трансформируем данные
idx = df_true.index # запоминаем значения индекса исходного DF
cols = df_true.columns # запоминаем значения столбцов исходного DF

# Определим объект класса для трансформации вводных данных. Используем установки по умолчанию.
# scaler = MinMaxScaler() # Объект MinMax шкалирования
# scaler = StandardScaler() # определим объект Z-score
scaler = PowerTransformer() # Объект Power трансформера (даёт лучшие метрики качества для IDW)
```

```

# scaler = RobustScaler() # Объект Robust шкалирования
# scaler = QuantileTransformer(random_state=56) # Объект Quantile трансформера

# Объект Spline трансформера - для данной модели неприменим: увеличивает количество столбцов,
# а для них не существует соответствующих метеостанций.
# scaler = SplineTransformer()

# переопределим df_test, трансформируя входные данные
df_test = pd.DataFrame(scaler.fit_transform(df_true), index=idx, columns=cols)
df_test.sample(5, random_state=56)

```

Out[73]:

	Visibility#V_Volochek	Visibility#Staritsa	Visibility#Tver	Visibility#Klin	Visibility#Dmitrov	Visibility#Volokolamsk	Visibility#Mozhaisk	Visibility#Leningrad
2019-07-10 12:00:00	-1.966225	1.003720	0.407935	-0.098267	0.006484	-0.152729	-0.114161	0.000000
2013-03-01 15:00:00	-2.452472	-2.321752	-2.329132	-0.098267	-0.681677	-2.522108	-0.114161	0.000000
2010-11-03 09:00:00	-2.081584	-1.661019	-2.320378	-0.098267	0.006484	0.215180	-0.285150	0.000000
2010-01-03 15:00:00	0.420031	1.003720	0.407935	-0.098267	0.006484	1.963050	2.255856	0.000000
2014-07-18 15:00:00	-0.761378	1.003720	0.407935	-0.098267	0.006484	-0.152729	-0.114161	0.000000

◀ ▶

In [74]:

```
# df_test.info()
df_test.shape
```

Out[74]:

```
(21783, 11)
```

Создадим массив индексов для выборки моделируемых и проверочных значений. Массив будет включать последовательно все индексы строк и случайный индекс столбца.

```
In [75]: # Зафиксируем RandomState
rs56 = np.random.RandomState(56)
# Создаём 1мерный массив с последовательностью, равной количеству рядов в df_test
arr_row_index = np.arange(0, df_test.shape[0])
# Создаём 1мерный массив со случайными целыми числами в диапазоне количества столбцов в df_test
arr_column_index = rs56.randint(0, df_test.shape[1], df_test.shape[0])
# Соединяем 2 массива и транспонируем полученный массив
arr_idx_data = np.vstack((arr_row_index, arr_column_index)).T

arr_row_index
arr_column_index
arr_idx_data

Out[75]: array([    0,     1,     2, ..., 21780, 21781, 21782])

Out[75]: array([5, 4, 0, ..., 0, 1, 3])

Out[75]: array([[  0,    5],
   [  1,    4],
   [  2,    0],
   ...,
   [21780,    0],
   [21781,    1],
   [21782,    3]])
```

Создадим тренировочный и обучающий массивы. Для этого:

- определим \$X\$ как массив координат ячеек в архиве - (np.array),
- определим \$y\$ как массив значений ячеек в множестве \$X\$ - (np.array).

```
In [76]: # Разделим наши данные на обучающую и валидационную части.
# используем индексы значений как параметры модели
X = arr_idx_data
# результирующие значения (фактические значения измерений, соответствующие индексам), выведем в список и преобразуем в np.array
y = np.array([df_test.iloc[x, y] for x, y in arr_idx_data])

x_train, x_test, y_train, y_test = train_test_split(X, y, train_size=0.6, test_size=0.4, random_state=56)
x_train.shape
y_train.shape
x_test.shape
y_test.shape
```

```
Out[76]: (13069, 2)
Out[76]: (13069,)
Out[76]: (8714, 2)
Out[76]: (8714,)
```

### 7.1.3.1. Модель средней, взвешенной по степени обратных расстояний (IDW)

Эта модель уже задана в виде функции *inverse\_distance\_avg*.

Модель применяется для каждого отдельно взятого момента наблюдения. Входные данные зафиксированы:

- Матрица расстояний между точками в поле метеостанций (df\_stations)
- Известные значения показателей для точек в поле метеостанций (архивы параметров).

Ожидаемые выходные данные:

- значение показателя для моделируемой точки (по сути, все значения NaN, включая значения для Агробиостанции МГУ в Чашниково).

Модель создана специально для имеющегося набора данных, поэтому для её работы не потребуется значительных преобразований архивов. Сама модель написана "вручную", без использования библиотечных функций машинного обучения.

Выше мы использовали формулу средней, взвешенной по обратным квадратам расстояния (по умолчанию в *inverse\_distance\_avg* задан параметр степени равный 2). Однако степень, в которую возводятся обратные величины расстояний - это единственный параметр, который можно менять в нашей реализации модели IDW.

Попробуем, как работает модель с различными значениями степени для обратных расстояний, и выделим ту степень, которая обеспечивает лучшие метрики качества.

**Обучающий датасет** Подберём лучшую степень для весов - обратных расстояний, ориентируясь на лучшие значения метрик качества

```
In [77]: start_time = time.time() # для замера времени выполнения кода
r2_idw_tr, mae_idw_tr, rmse_idw_tr = 0, 0, 0 # обнулим значения метрик качества
```

```

iterations = 0 # количество итераций
power = 1.75 # Начальное значение степени (опровергнуто начальные степени от 1)
power_increment = 0.25 # шаг увеличения степени
list_metrics=[] # список для фиксации результатов итераций

r2_idw_tr_old = 0 # Устанавливаем в 0 предыдущее значение R2
print('ОБУЧАЮЩИЙ ДАТАСЕТ, IDW:\n')

# Зададим предел R2 для поиска оптимального веса
while r2_idw_tr_old <= r2_idw_tr:
    power += power_increment # Увеличиваем степень на 1 шаг
    iterations +=1 # Увеличиваем счётчик проходов цикла
    r2_idw_tr_old = r2_idw_tr # Запоминаем предыдущее значение R2

    # Создаём y_predict_idw_tr по индексам в массиве x_train
    # используем функцию inverse_distance_avg
    # Проходим по массиву и считываем из df_test данные по координатам, выводим результат списком
    y_predict_idw_tr = [
        inverse_distance_avg(row=df_test.iloc[x],
                             param_=PARAMETER71,
                             station_=df_test.keys()[y][len(PARAMETER71)+1:],
                             df_dists_=df_station_dists,
                             power_=power)
        for x, y in x_train
    ]

    # Расчитываем метрики качества
    max_e_idw_tr = max_error(y_train, y_predict_idw_tr)
    mae_idw_tr = mean_absolute_error(y_train, y_predict_idw_tr)
    mse_idw_tr = mean_squared_error(y_train, y_predict_idw_tr)
    rmse_idw_tr = mean_squared_error(y_train, y_predict_idw_tr, squared=False)
    r2_idw_tr = r2_score(y_train, y_predict_idw_tr)

    if r2_idw_tr > r2_idw_tr_old:
        best_power_idw_tr = power
        best_max_e_idw_tr = max_e_idw_tr
        best_mae_idw_tr = mae_idw_tr
        best_mse_idw_tr = mse_idw_tr
        best_rmse_idw_tr = rmse_idw_tr
        best_r2_idw_tr = r2_idw_tr

    # замер времени:
    chk_time = time.time()
    elapsed_time = chk_time - start_time

```

```

time_formatted = time.strftime("%H:%M:%S", time.gmtime(elapsed_time))

print(f'Elapsed time={time_formatted}\n'
      f'Текущие значения: iterations={iterations}, power={power},\n'
      f'Максимальная ошибка (MAX_E) IDW = {max_e_idw_tr:.7f}\n'
      f'Средняя абсолютная ошибка (MAE) IDW = {mae_idw_tr:.7f}\n'
      f'Средний квадрат ошибки (MSE) IDW = {mse_idw_tr:.7f}\n'
      f'Средняя квадратическая ошибка (RMSE) IDW = {rmse_idw_tr:.7f}\n'
      f'Коэффициент детерминации (R2) IDW = {r2_idw_tr:.7f}\n'
      )
print(f'ЛУЧШИЕ значения: power={best_power_idw_tr},\n'
      f'Максимальная ошибка (MAX_E) IDW = {best_max_e_idw_tr:.7f}\n'
      f'Средняя абсолютная ошибка (MAE) IDW = {best_mae_idw_tr:.7f}\n'
      f'Средний квадрат ошибки (MSE) IDW = {best_mse_idw_tr:.7f}\n'
      f'Средняя квадратическая ошибка (RMSE) IDW = {best_rmse_idw_tr:.7f}\n'
      f'Коэффициент детерминации (R2) IDW = {best_r2_idw_tr:.7f}')
)

```

ОБУЧАЮЩИЙ ДАТАСЕТ, IDW:

```

Elapsed time=00:00:45
Текущие значения: iterations=1, power=2.0,
Максимальная ошибка (MAX_E) IDW = 4.8790276
Средняя абсолютная ошибка (MAE) IDW = 0.5350468
Средний квадрат ошибки (MSE) IDW = 0.5818737
Средняя квадратическая ошибка (RMSE) IDW = 0.7628065
Коэффициент детерминации (R2) IDW = 0.4159327

```

```

Elapsed time=00:01:30
Текущие значения: iterations=2, power=2.25,
Максимальная ошибка (MAX_E) IDW = 4.8703971
Средняя абсолютная ошибка (MAE) IDW = 0.5349551
Средний квадрат ошибки (MSE) IDW = 0.5859667
Средняя квадратическая ошибка (RMSE) IDW = 0.7654846
Коэффициент детерминации (R2) IDW = 0.4118242

```

```

ЛУЧШИЕ значения: power=2.0,
Максимальная ошибка (MAX_E) IDW = 4.8790276
Средняя абсолютная ошибка (MAE) IDW = 0.5350468
Средний квадрат ошибки (MSE) IDW = 0.5818737
Средняя квадратическая ошибка (RMSE) IDW = 0.7628065
Коэффициент детерминации (R2) IDW = 0.4159327

```

Несколько запусков кода выше, с различными параметрами степени (от 1 до 10), показали, что, судя по R2, лучшим значением степени на обучающем массиве является 2. Оно даёт лучшие метрики качества. Однако, метрики качества невысоки. Это объяснимо, так как изначальные данные плохо коррелируют между собой.

Применим эту степень для валидационного массива.

## Валидационный датасет

```
In [78]: # Создаём y_predict_idw_vld по индексам в x_test
# используем функцию inverse_distance_avg
# Проходим по массиву и считываем из df_test данные по координатам, выводим результатом списком

y_predict_idw_vld = [
    inverse_distance_avg(row=df_test.iloc[x],
                          param_=PARAMETER71,
                          station_=df_test.keys()[y][len(PARAMETER71)+1:],
                          df_dists_=df_station_dists,
                          power_=best_power_idw_tr) # берём лучшее значение степени, полученное из кода выше на тренинговом датасете
    for x, y in x_test
]
# y_predict_idw_vld

max_e_idw_vld = max_error(y_test, y_predict_idw_vld)
mae_idw_vld = mean_absolute_error(y_test, y_predict_idw_vld)
mse_idw_vld = mean_squared_error(y_test, y_predict_idw_vld)
rmse_idw_vld = mean_squared_error(y_test, y_predict_idw_vld, squared=False)
r2_idw_vld = r2_score(y_test, y_predict_idw_vld)
print(f'ВАЛИДАЦИОННЫЙ ДАТАСЕТ, IDW:\n'
      f'Максимальная ошибка (MAX_E) IDW = {max_e_idw_vld:.7f}\n'
      f'Средняя абсолютная ошибка (MAE) IDW = {mae_idw_vld:.7f}\n'
      f'Средний квадрат ошибки (MSE) IDW = {mse_idw_vld:.7f}\n'
      f'Средняя квадратическая ошибка (RMSE) IDW = {rmse_idw_vld:.7f}\n'
      f'Коэффициент детерминации (R2) IDW = {r2_idw_vld:.7f}')
)
```

ВАЛИДАЦИОННЫЙ ДАТАСЕТ, IDW:

Максимальная ошибка (MAX\_E) IDW = 6.1235087  
Средняя абсолютная ошибка (MAE) IDW = 0.5399484  
Средний квадрат ошибки (MSE) IDW = 0.5834279  
Средняя квадратическая ошибка (RMSE) IDW = 0.7638245  
Коэффициент детерминации (R2) IDW = 0.4140897

Проверим метрики качества на обратно-трансформированном датасете в соотношении с исходными данными

```
In [79]: # Исходные значения y, соответствующие y_test
y_true = [df_true.iat[idx,col] for idx, col in x_test]

# Копируем трансформированный датасет и заменяем в нём целевые значения на предикты
df_predict = df_test.copy(deep=True)
for i, coords in enumerate(x_test):
    df_predict.iat[coords[0], coords[1]] = y_predict_idw_vld[i]

# Обратная трансформация датасета с предиктами
arr_inv_trans = scaler.inverse_transform(df_predict)

# Создаём список обратно-трансформированных предиктов по координатам в x_test.
# Заменяем NaN на 0.05 (минимальное значение для Visibility).
y_inv_trans = np.nan_to_num([arr_inv_trans[idx,col] for idx, col in x_test], 1)

# np.isnan(y_true).sum()
# np.isnan(df_predict).sum()
# np.isnan(arr_inv_trans).sum()
# np.isnan(y_inv_trans).sum()

max_e_idw_inv_trans = max_error(y_true, y_inv_trans)
mae_idw_inv_trans = mean_absolute_error(y_true, y_inv_trans)
mse_idw_inv_trans = mean_squared_error(y_true, y_inv_trans)
rmse_idw_inv_trans = mean_squared_error(y_true, y_inv_trans, squared=False)
r2_idw_inv_trans = r2_score(y_true, y_inv_trans)
print(f'ОБРАТНО ТРАНСФОРМИРОВАННЫЙ ВАЛИДАЦИОННЫЙ ДАТАСЕТ, IDW:\n'
      f'Максимальная ошибка (MAX_E) IDW = {max_e_idw_inv_trans:.7f}\n'
      f'Средняя абсолютная ошибка (MAE) IDW = {mae_idw_inv_trans:.7f}\n'
      f'Средний квадрат ошибки (MSE) IDW = {mse_idw_inv_trans:.7f}\n'
      f'Средняя квадратическая ошибка (RMSE) IDW = {rmse_idw_inv_trans:.7f}\n'
      f'Коэффициент детерминации (R2) IDW = {r2_idw_inv_trans:.7f}'
      )
```

ОБРАТНО ТРАНСФОРМИРОВАННЫЙ ВАЛИДАЦИОННЫЙ ДАТАСЕТ, IDW:

Максимальная ошибка (MAX\_E) IDW = 61.4924681  
Средняя абсолютная ошибка (MAE) IDW = 3.2226042  
Средний квадрат ошибки (MSE) IDW = 32.6671185  
Средняя квадратическая ошибка (RMSE) IDW = 5.7155156  
Коэффициент детерминации (R2) IDW = 0.4831635

## ВЫВОД

Модель средней, взвешенной по обратным расстояниям, даёт не самые высокие метрики качества (по сравнению с показателями, обработанными ранее в других тетрадях).

### 7.1.3.2. Кrigинг и вариограммы в реализации библиотеки SciKit GStat

Опробуем работу модели кригинга на уже сформированных тренинговой и валидационной выборках

Координатная сетка для точек наблюдения в данной модели строится на основе "плоской" земной поверхности. Разность между расстояниями по прямой и по поверхности сферы игнорируется (впрочем, для нашего региона исследования это не приведёт к существенным ошибкам).

```
In [80]: # Загружаем координаты из df_coords_full (определён в разделе определения функции кригинга 2.2):
# Создаём DF с координатами нужных нам точек
df_coords = df_coords_full[["LoE", "LaN"]][:-2]

# df_coords
```

#### Обучающий датасет

Проверим работу модели кригинга сначала на данных обучающей выборки. На ней будем подбирать наиболее подходящие параметры вариограмм и модели кригинга (которые дают лучшие метрики и выдают меньшее количество ошибок из-за недостаточности наблюдений в поле метеостанций).

Представляется полезным сравнить работу модели обратных степеней расстояний и кригинга на одних и тех же данных.

В процессе исследования работоспособности модели код ниже многократно запускался с различными параметрами.

```
In [81]: # Намеренно оставим закомментированные части кода, они могут использоваться для отладки
start_time = time.time() # для замера времени выполнения кода
# counter = 0
y_predict_kriging_tr = [] # список предиктов для x_test

for x in x_train: # x[0] ряд в df_test, x[1] столбец, соответствующий названию станции
#     counter += 1
##     if counter >15:
```



```

#
#                               model='spherical',
#                               dist_func='euclidean',
#                               bin_func='even',
#                               azimuth=0,
#                               tolerance=90,
#                               maxlag='full',
#                               n_lags=4)
#
V_South = skg.DirectionalVariogram(coordinates=coords_v,
#
#                               values=vals_v,
#                               estimator='matheron',
#                               model='spherical',
#                               dist_func='euclidean',
#                               bin_func='even',
#                               azimuth=-90,
#                               tolerance=90,
#                               maxlag='full',
#                               n_lags=4)
#
V_West = skg.DirectionalVariogram(coordinates=coords_v,
#
#                               values=vals_v,
#                               estimator='matheron',
#                               model='spherical',
#                               dist_func='euclidean',
#                               bin_func='even',
#                               azimuth=180,
#                               tolerance=90,
#                               maxlag='full',
#                               n_lags=4)

##
#      V=V_West
##
```

**except ValueError:** # Уберём количество сэмплов из определения вариограммы (когда количество сэмплов слишком велико)

```

try:
    V_ = skg.Variogram(coordinates=coords_v,
                        values=vals_v,
                        estimator='matheron',
                        model='spherical',
                        dist_func='euclidean',
                        bin_func='ward',
                        maxlag=0.99999, # Используем всю матрицу расстояний
                        n_lags=4, # количество бин не должно быть очень большим, чтобы не уменьшать effective distance
                        normalize=False,
                        use_nugget=False,
                        #fit_method='ml',
```

```

        #entropy_bins = 1
    );
except: # Возникает ошибка - не можем построить вариаграмму
    val_predict = np.nan
    y_predict_kriging_tr.append(val_predict)
    continue
except: # возникает другая ошибка (помимо ValueError) - не можем построить вариаграмму
    val_predict = np.nan
    y_predict_kriging_tr.append(val_predict)
    continue

# Определяем модель кригинга
model_ok = skg.OrdinaryKriging(V, min_points=1, max_points=14, mode='exact')

# координаты точки предикта:
predict_coords = np.array(df_coords)
# Получаем предикт для тестируемой точки (получаем массив предиктов, выбираем из него индекс точки предикта)
# В процессе работы model_ok.transform выдается много сообщений типа:
# 'Warning: for %d Locations, not enough neighbors were found within the range.' % self.no_points_error
# "Заглушим" их, направив их в класс вывода, который ничего не делает (определен выше)

sys.stdout = NullIO() # определим вывод print() в класс нулевого вывода

val_predict = model_ok.transform(predict_coords[:,0], predict_coords[:,1])[x[1]]
sys.stdout = real_stdout # восстановим стандартный вывод

# добавляем предикт в список предиктов
y_predict_kriging_tr.append(val_predict)

## 
# if V.describe()['effective_range'] < 1:
#     dm = pdist(df_coords[['LoE', 'LaN']]) # массив пар расстояний
#     print(f'Итерация: {counter}, позиция в x_test: {x}\nКоличество пар расстояний: {len(dm)}\n'
#           f'Effective Range: {V.describe()["effective_range"]}\n'
#           f'Количество пар расстояний внутри Effective range: {sum(dm <= V.describe()["effective_range"])}\n'
#           f'Количество пар расстояний вне Effective range: {sum(dm > V.describe()["effective_range"])}\n'
#           f'Предикт: {val_predict}, координаты предикта: {predict_coords[x[1]]}, станция: {df_coords.iloc[x[1]].name}'
#           )
#     fig = V.plot(show=False)
#     plt.show()
##
y_predict_kriging_tr = np.array(y_predict_kriging_tr) # превратим список предиктов в массив

```

```
# замер времени:  
chk_time = time.time()  
elapsed_time = chk_time - start_time  
time_formatted = time.strftime("%H:%M:%S", time.gmtime(elapsed_time))
```

```
In [82]: if np.isnan(np.array(y_predict_kriging_tr)).sum() == 0:  
    max_e_kriging_tr = max_error(y_train, y_predict_kriging_tr)  
    mae_kriging_tr = mean_absolute_error(y_train, y_predict_kriging_tr)  
    mse_kriging_tr = mean_squared_error(y_train, y_predict_kriging_tr)  
    rmse_kriging_tr = mean_squared_error(y_train, y_predict_kriging_tr, squared=False)  
    r2_kriging_tr = r2_score(y_train, y_predict_kriging_tr)  
else:  
    max_e_kriging_tr = np.nan  
    mae_kriging_tr = np.nan  
    mse_kriging_tr = np.nan  
    rmse_kriging_tr = np.nan  
    r2_kriging_tr = np.nan  
  
print('ОБУЧАЮЩИЙ ДАТАСЕТ, КРИГИНГ')  
print(f'Elapsed time={time_formatted}\n'  
      f'Значения метрик:\n'  
      f'R2={r2_kriging_tr:.7f}, MAX_E={max_e_kriging_tr:.7f}, MAE={mae_kriging_tr:.7f}, MSE={mse_kriging_tr}, '  
      f'RMSE={rmse_kriging_tr:.7f}\n'  
)  
  
print(f'Количество значений, которые не удалось предсказать: {pd.isna([y_predict_kriging_tr]).sum()}\n'  
      f'Это составляет {pd.isna([y_predict_kriging_tr]).sum()/len(y_predict_kriging_tr):.4%} от обучающего массива данных')
```

ОБУЧАЮЩИЙ ДАТАСЕТ, КРИГИНГ

Elapsed time=00:00:13

Значения метрик:

R2=nan, MAX\_E=nan, MAE=nan, MSE=nan, RMSE=nan

Количество значений, которые не удалось предсказать: 13069

Это составляет 100.0000% от обучающего массива данных

Таким образом, модель кригинга не может найти предикты.

```
In [83]: # Попробуем оценить качество работы модели в случаях, когда ей удалось найти предикты.
```

```
In [84]: # Оставим в y_train и y_predict_kriging_tr только значения неравные NaN  
# y_train_kriging_shrunk = y_train[~np.isnan(y_predict_kriging_tr)]
```

```
# y_predict_kriging_tr_shrunk = y_predict_kriging_tr[~np.isnan(y_predict_kriging_tr)]  
  
# max_e_kriging_tr = max_error(y_train_kriging_shrunk, y_predict_kriging_tr_shrunk)  
# mae_kriging_tr = mean_absolute_error(y_train_kriging_shrunk, y_predict_kriging_tr_shrunk)  
# mse_kriging_tr = mean_squared_error(y_train_kriging_shrunk, y_predict_kriging_tr_shrunk)  
# rmse_kriging_tr = mean_squared_error(y_train_kriging_shrunk, y_predict_kriging_tr_shrunk, squared=False)  
# r2_kriging_tr = r2_score(y_train_kriging_shrunk, y_predict_kriging_tr_shrunk)
```

```
In [85]: print(f'КРИГИНГ на ОБУЧАЮЩЕМ датасете (для случаев, где удалось найти предикты):\n'  
      f'Максимальная ошибка (MAX_E) Kriging = {max_e_kriging_tr:.7f}, Kriging - IDW = '  
      f'{(max_e_kriging_tr - max_e_idw_tr):.7f}\n'  
      f'Средняя абсолютная ошибка (MAE) Kriging = {mae_kriging_tr:.7f}, Kriging - IDW = '  
      f'{(mae_kriging_tr - mae_idw_tr):.7f}\n'  
      f'Средний квадрат ошибки (MSE) Kriging = {mse_kriging_tr:.7f}, Kriging - IDW = '  
      f'{(mse_kriging_tr - mse_idw_tr):.7f}\n'  
      f'Средняя квадратическая ошибка (RMSE) Kriging = {rmse_kriging_tr:.7f}, '  
      f'Kriging - IDW = {(rmse_kriging_tr - rmse_idw_tr):.7f}\n'  
      f'Коэффициент детерминации (R2) Kriging = {r2_kriging_tr:.7f}, Kriging - IDW = '  
      f'{(r2_kriging_tr - r2_idw_tr):.7f}'  
)
```

КРИГИНГ на ОБУЧАЮЩЕМ датасете (для случаев, где удалось найти предикты):  
Максимальная ошибка (MAX\_E) Kriging = nan, Kriging - IDW = nan  
Средняя абсолютная ошибка (MAE) Kriging = nan, Kriging - IDW = nan  
Средний квадрат ошибки (MSE) Kriging = nan, Kriging - IDW = nan  
Средняя квадратическая ошибка (RMSE) Kriging = nan, Kriging - IDW = nan  
Коэффициент детерминации (R2) Kriging = nan, Kriging - IDW = nan

## Валидационный датасет

Посмотрим, как модель будет отрабатывать на валидационной выборке.

```
In [86]: start_time = time.time() # для замера времени выполнения кода  
# counter = 0  
y_predict_kriging_vld = [] # список предиктов для x_test  
  
for x in x_test: # x[0] ряд в df_test, x[1] столбец, соответствующий названию станции  
#     counter += 1  
##  
#     if counter >15:  
#         break  
#     else:  
#         counter +=1
```

```

##  

# Найдем по координатам x_test ряд в df_test  

row = df_test.iloc[x[0]]  

# Присваиваем проверяемому значению NaN  

row.iloc[x[1]] = np.nan  

# Для построения вариограммы:  

# - получаем массив координат без указанной точки  

# (удаляем соответствующий ряд из df_coords, преобразуем оставшийся df в массив)  

# - получаем массив значений  

idx = x[1]  

coords_v = np.array(df_coords.drop(index = df_coords.iloc[x[1]].name))[:, :2]  

vals_v = np.array(row.dropna())  

try:  

    # Определяем вариограмму  

    V = skg.Variogram(coordinates=coords_v,  

                        values=vals_v,  

                        estimator='matheron',  

                        model='spherical',  

                        dist_func='euclidean',  

                        bin_func='ward',  

                        maxlag=0.99999, # Используем всю матрицу расстояний  

                        n_lags=4, # количество бин не должно быть очень большим, чтобы не уменьшать effective distance  

                        normalize=False,  

                        use_nugget=False,  

                        samples=len(vals_v),  

                        fit_method='trf',  

                        )  

except ValueError: # Уберём количество сэмплов из определения вариограммы (когда количество сэмплов слишком велико)
    try:  

        V_ = skg.Variogram(coordinates=coords_v,  

                            values=vals_v,  

                            estimator='matheron',  

                            model='spherical',  

                            dist_func='euclidean',  

                            bin_func='ward',  

                            maxlag=0.99999, # Используем всю матрицу расстояний  

                            n_lags=4, # количество бин не должно быть очень большим, чтобы не уменьшать effective distance  

                            normalize=False,  

                            use_nugget=False,  

                            #fit_method='ml',  

                            #entropy_bins = 1  

                            );  

    except: # Возникает ошибка - не можем построить вариаграмму

```

```

    val_predict = np.nan
    y_predict_kriging_vld.append(val_predict)
    continue
except: # возникает другая ошибка (помимо ValueError) - не можем построить вариаграмму
    val_predict = np.nan
    y_predict_kriging_vld.append(val_predict)
    continue

# Определяем модель кригинга
model_ok = skg.OrdinaryKriging(V, min_points=1, max_points=14, mode='exact')

# координаты точки предикта:
predict_coords = np.array(df_coords)
# Получаем предикт для тестируемой точки (получаем массив предиктов, выбираем из него индекс точки предикта)
# В процессе работы model_ok.transform выдается много сообщений типа:
# 'Warning: for %d Locations, not enough neighbors were found within the range.' % self.no_points_error
# "Заглушим" их, направив их в класс вывода, который ничего не делает (определен выше)

sys.stdout = NullIO() # определим вывод print() в класс нулевого вывода

val_predict = model_ok.transform(predict_coords[:,0], predict_coords[:,1])[x[1]]
sys.stdout = real_stdout # восстановим стандартный вывод

# добавляем предикт в список предиктов
y_predict_kriging_vld.append(val_predict)

y_predict_kriging_vld = np.array(y_predict_kriging_vld)

# замер времени:
chk_time = time.time()
elapsed_time = chk_time - start_time
time_formatted = time.strftime("%H:%M:%S", time.gmtime(elapsed_time))

```

In [87]:

```

if np.isnan(np.array(y_predict_kriging_vld)).sum() == 0:
    max_e_kriging_vld = max_error(y_test, y_predict_kriging_vld)
    mae_kriging_vld = mean_absolute_error(y_test, y_predict_kriging_vld)
    mse_kriging_vld = mean_squared_error(y_test, y_predict_kriging_vld)
    rmse_kriging_vld = mean_squared_error(y_test, y_predict_kriging_vld, squared=False)
    r2_kriging_vld = r2_score(y_test, y_predict_kriging_vld)
else:
    max_e_kriging_vld = np.nan
    mae_kriging_vld = np.nan

```

```

mse_kriging_vld = np.nan
rmse_kriging_vld = np.nan
r2_kriging_vld = np.nan

print(f'Elapsed time={time_formatted}\n'
      f'Значения метрик:\n'
      f'R2={r2_kriging_vld:.7f}, MAX_E={max_e_kriging_vld}, MAE={mae_kriging_vld:.7f}, MSE={mse_kriging_vld}, '
      f'RMSE={rmse_kriging_vld:.7f}\n')
print('ВАЛИДАЦИОННЫЙ ДАТАСЕТ, КРИГИНГ')
print(f'Количество значений, которые не удалось предсказать: {np.isnan([y_predict_kriging_vld]).sum()}\n'
      f'Это составляет {pd.isna([y_predict_kriging_vld]).sum()/len(y_predict_kriging_vld):.4%} '
      f'от валидационного массива данных')

```

Elapsed time=00:00:08

Значения метрик:

R2=nan, MAX\_E=nan, MAE=nan, MSE=nan, RMSE=nan

ВАЛИДАЦИОННЫЙ ДАТАСЕТ, КРИГИНГ

Количество значений, которые не удалось предсказать: 8714

Это составляет 100.0000% от валидационного массива данных

```

In [88]: # y_test_kriging_shrunk = y_test[~np.isnan(y_predict_kriging_vld)]
# y_predict_kriging_vld_shrunk = y_predict_kriging_vld[~np.isnan(y_predict_kriging_vld)]

# max_e_kriging_vld = max_error(y_test_kriging_shrunk, y_predict_kriging_vld_shrunk)
# mae_kriging_vld = mean_absolute_error(y_test_kriging_shrunk, y_predict_kriging_vld_shrunk)
# mse_kriging_vld = mean_squared_error(y_test_kriging_shrunk, y_predict_kriging_vld_shrunk)
# rmse_kriging_vld = mean_squared_error(y_test_kriging_shrunk, y_predict_kriging_vld_shrunk, squared=False)
# r2_kriging_vld = r2_score(y_test_kriging_shrunk, y_predict_kriging_vld_shrunk)

```

```

In [89]: print(f'Кригинг на ВАЛИДАЦИОННОМ датасете (для случаев, где удалось найти предикты):\n'
      f'Максимальная ошибка (MAX_E) Kriging = {max_e_kriging_vld:.7f}, Kriging - IDW = '
      f'{(max_e_kriging_vld - max_e_idw_vld):.7f}\n'
      f'Средняя абсолютная ошибка (MAE) Kriging = {mae_kriging_vld:.7f}, '
      f'Kriging - IDW = {(mae_kriging_vld - mae_idw_vld):.7f}\n'
      f'Средний квадрат ошибки (MSE) Kriging = {mse_kriging_vld:.7f}, Kriging - IDW = '
      f'{(mse_kriging_vld - mse_idw_vld):.7f}\n'
      f'Средняя квадратическая ошибка (RMSE) Kriging = {rmse_kriging_vld:.7f}, '
      f'Kriging - IDW = {(rmse_kriging_vld - rmse_idw_vld):.7f}\n'
      f'Коэффициент детерминации (R2) Kriging = {r2_kriging_vld:.7f}, Kriging - IDW = '
      f'{(r2_kriging_vld - r2_idw_vld):.7f}')

```

Кrigинг на ВАЛИДАЦИОННОМ датасете (для случаев, где удалось найти предикты):

Максимальная ошибка (MAX\_E) Kriging = nan, Kriging - IDW = nan

Средняя абсолютная ошибка (MAE) Kriging = nan, Kriging - IDW = nan

Средний квадрат ошибки (MSE) Kriging = nan, Kriging - IDW = nan

Средняя квадратическая ошибка (RMSE) Kriging = nan, Kriging - IDW = nan

Коэффициент детерминации (R2) Kriging = nan, Kriging - IDW = nan

## ВЫВОД

Из-за низкой пространственной корреляции данных модель кригинга оказывается неприменимой

### 7.1.3.3. Модель кригинга, совмещённая с IDW (для значений, которые кригинг не может предсказать)

## ВЫВОД

Так как модель кригинга оказалась неприменимой к нашим данным, использовать её совместно с IDW не получится.

## ПРИМЕЧАНИЕ

Для подбора лучших параметров входных данных для моделей пространственной экстраполяции были использованы следующие методы трансформации данных с соответственными значениями метрики R2 для IDW:

- StandardScaler - 0,4,
- MinMaxScaler - 0,2,
- RobustScaler - 0,35,
- QuantileTransformer - 0,4,
- PowerTransformer 0,414,
- SplineTransformer - неприменим так как увеличивает число столбцов в датасете, для которых нет метеостанций. Модель Kriging оказалась неспособной дать предсказания ни с одним способом трансформации (ни без неё).

Выше оставлены результаты для Power трансформации.

### 7.1.3.4. Создание общего датасета.

Для построения датасета параметров, независимо от пространственного расположения метеостанций, потребуется создать датафрейм с иной структурой. Так как для предсказания значений в пос. Чашниково **мы можем использовать только уже смоделированные параметры, нам необходимо будет взять только их**. Возьмём из названных только те параметры, которые гипотетически могут быть связаны с метеорологической дальностью видимости.

Обновим архивы, чтобы удалить в них аномальные значения и вставить значения, заполняющие сплошные NaN

## Перенесём уже исправленные сплошные NaN и удалённые аномалии в архивы

```
In [90]: # Перенесём уже исправленные значения в dict_df_parameters, оставшиеся NaN исправим по результатам моделирования
dict_df_parameters['df_'+PARAMETER71] = df_tmp71.copy(deep=True)

# Перенесём уже исправленные значения в dict_df_locations, оставшиеся NaN исправим по результатам моделирования
for name_df in dict_df_locations.keys():
    # За исключением Чашниково и условного центра - здесь для них значения ещё не определены
    if (name_df == 'df_Chashnikovo') or (name_df == 'df_Rfrnce_point'):
        continue
    else:
        dict_df_locations[name_df].loc[:, PARAMETER71] = df_tmp71.loc[:, PARAMETER71 + '#' + name_df[3:]]
```

## Создадим единый датафрейм для анализа взаимосвязи различных параметров

Предварительный отбор данных

Окончательно определиться с необходимыми данными можно будет по итогам анализа их корреляции. Дополнительно используем (рассчитаем) данные об изменениях температуры и давления за последние 3 часа, а также возьмём значения показателя на ближайшей метеостанции, где он наблюдался, с параметрами этой станции.

Для начала выберем из уже обработанных данных, те параметры, которые физически могут быть связаны с результатом наблюдения дальностью видимости:

- географическое положение
  - высота над уровнем моря,
  - расстояние от условной центральной точки,
  - начальный азимут на условную центральную точку;
- момент наблюдения
  - время,
  - месяц,
  - час;
- температура
  - наблюдаемая температура,

- температурная тенденция 3 часа
- максимальная температура за предшествующие 12 часов,
- минимальная температура за предшествующие 12 часов;
- давление,
- барическая тенденция за 3 часа
- влажность,
- температура точки росы,
- значение показателя дальности видимости на ближайшей станции, где оно зафиксировано,
- расстояние до такой ближайшей станции,
- начальный азимут на такую ближайшую станцию,
- наблюдаемое значение показателя дальности видимости.

```
In [91]: # Создаём пустой датафрейм с параметрами, которые могут иметь взаимную зависимость
# с метеорологической дальностью видимости
list_column_selection = [
    "Station", "Height", "Distance", "Bearing",
    "Observation", "Month", "Hour",
    "T", "T_min", "T_max", "T_drift",
    "P_station", "P_drift", "Humid", "Dew_point",
    # "Wind_dir360", "Wind_speed",
    "Closest_val", # Значение параметра на ближайшей метеостанции
    "Closest_dist", # Расстояние до ближайшей станции
    "Closest_bearing", # Начальный азимут на ближайшую станцию
    "Closest_station",
    "Visibility"]
df_total71=pd.DataFrame(
    columns = list_column_selection
)
```

```
In [92]: df_total71=pd.DataFrame(
    columns = list_column_selection
)
# В цикле по датафреймам метеостанций выберем нужные столбцы, определим дополнительные, и присоединим их к df_total71
for name in dict_df_locations.keys():
    station = name[3:]
    # print(station)
    closest_station = 'Rfrnce_point' # Начальное присвоение для входа в следующий цикл
    smallest = 2 # Для nsmallest() - меньший по порядку 1 - эта сама станция
    # расстояние до ближайшей станции:
```

```

closest_dist = df_station_dists[name[3:]].nsmallest(smallest).iloc[[-1]].values[0]
# ближайшая станция:
closest_station = (df_station_dists.loc[df_station_dists[name[3:]] == closest_dist]
                    .station
                    .values[0])
# начальный азимут на ближайшую станцию (от - name[3:], на - closest_station):
closest_bearing = (df_station_bearings[df_station_bearings.station == closest_station]
                    .loc[:,name[3:]]
                    .values[0])
# СЕРИЯ со значениями показателя на ближайшей станции
ser_closest_val = (dict_df_locations[f'df_{closest_station}'][PARAMETER71]
                    if ((closest_station != 'Rfrnce_point') and
                        (closest_station != 'Chashnikovo'))
                    else np.nan)
# closest_dist, closest_station, closest_val

df_total71 = (
    pd.concat(
        [df_total71,
         (dict_df_locations[name][[
             "T", "T_min", "T_max",
             "P_station", "P_drift", "Humid", "Dew_point",
             ]] # показатели есть в архивах, ниже идут расчётные показатели и данные других DF
         .assign(Visibility =
                 dict_df_locations[name].Visibility if (
                     (station != 'Rfrnce_point') and
                     (station != 'Chashnikovo'))
                 else np.nan)
         .assign(Month = dict_df_locations[name].index.month)
         .assign(Hour = dict_df_locations[name].index.hour)
         .assign(T_drift = (dict_df_locations[name]["T"] - dict_df_locations[name]["T"].shift(periods=3, freq='H'))))
         .assign(Station = station)
         .assign(Height = df_station_bearings[df_station_bearings.station == station]
                 .height
                 .values[0])
         .assign(Distance = df_station_dists[df_station_dists.station == station]
                 .Rfrnce_point
                 .values[0])
         .assign(Bearing = df_station_bearings[df_station_bearings.station == station]
                 .Rfrnce_point
                 .values[0])
         .assign(Closest_val = ser_closest_val) # Значение столбца Visibility для ближайшей к данной станции
         .assign(Closest_station = closest_station)
         .assign(Closest_dist = closest_dist) # Расстояние до ближайшей станции
        ]
    )
)

```

```

        .assign(Closest_bearing = closest_bearing) # Начальный азимут на ближайшую станцию
        .reset_index()
        .rename(columns={f'{station}_Local_time' : 'Observation', "index": 'Observation'})
        # удаляем последнюю строку в каждом DF - она состоит из NaN
        .dropna(subset=["T", "T_min", "T_max", "P_station", "P_drift", "Humid", "Dew_point"], how='all')
    )
],
axis=0,
ignore_index=True
)
)

df_total71.sample(5)

```

Out[92]:

	Station	Height	Distance	Bearing	Observation	Month	Hour	T	T_min	T_max	T_drift	P_station	P_drift	H
<b>676125</b>	V_Volochev	161	174.287843	328.934475	2016-08-16 15:00:00	8	15	18.400000	9.700000	18.400000	3.700000	742.900000	0.700000	48.0
<b>132633</b>	Kashyn	138	152.988800	36.271056	2011-10-02 12:00:00	10	12	5.900000	5.000000	5.900000	0.400000	748.200000	0.800000	79.0
<b>564038</b>	Tver	132	68.195735	352.158514	2020-04-12 18:00:00	4	18	8.400000	1.700000	8.400000	0.800000	749.700000	-1.400000	28.0
<b>44229</b>	Chashnikovo	215	71.720525	108.834256	2007-04-21 06:00:00	4	6	2.875272	2.875272	7.515012	-1.211453	728.666919	0.278495	83.2
<b>106107</b>	Kashyn	138	152.988800	36.271056	2020-10-30 06:00:00	10	6	3.800000	3.800000	7.800000	-3.900000	750.100000	-0.400000	89.0

Выше мы определили столбцы значений для ближайшей станции, исходя из всей серии наблюдений по ней. Нам необходимо заменить значения NaN в наблюдениях ближайшем метеостанции. Поэтому, в случаях, если Closest\_val содержит NaN следует заменить ближайшую метеостанцию на следующую ближайшую, там где есть наблюдения для данного параметра

Используем функцию нахождения значения параметра для ближайшей от искомой метеостанции.

In [93]:

```

%%time
# В цикле вычислим величины для замены NaN через функцию closest_value(station_, observation_, param_),
# произведём df_total71.update()

```

```

idx = df_total71[pd.isna(df_total71.Closest_val)].index # Сохраним индекс для выборки NaN в df_total71.Closest_val
# Для каждого заменяемого столбца определим список
list_closest_val = []
list_closest_dist = []
list_closest_bearing = []
list_closest_station= []

# В цикле переберём названия станций и моменты наблюдения (zip)
for station, observation in zip(df_total71[pd.isna(df_total71.Closest_val)].Station,
                                   df_total71[pd.isna(df_total71.Closest_val)].Observation):
    # определим кортеж значений с помощью функции
    tup_closest = closest_value(station_=station,
                                 observation_=observation,
                                 param_=PARAMETER71)
    # добавим значения в список
    list_closest_val.append(tup_closest[0])
    list_closest_dist.append(tup_closest[1])
    list_closest_bearing.append(tup_closest[2])
    list_closest_station.append(tup_closest[3])

# Создадим датафрейм из названий заменяемых столбцов и соответствующих им списков, а также сохрённого индекса
# Обновим общий датафрейм (обновлению подлежат только указанные столбцы и только по указанным индексам)
df_total71.update(
    pd.DataFrame(
        {"Closest_val": list_closest_val,
         "Closest_dist": list_closest_dist,
         "Closest_bearing": list_closest_bearing,
         "Closest_station": list_closest_station},
        index=idx))

```

CPU times: total: 21min 5s

Wall time: 21min 16s

Заменим NaN в столбцах изменения температуры и давления (drift) на 0, а в столбцах минимума и максимума температуры - на текущую температуру. Эти NaN обусловлены начальным момент наблюдения в архивах.

```

In [94]: dict_values = {"T_drift": 0, "P_drift": 0}
df_total71.fillna(value=dict_values, inplace=True)
df_total71.T_min.fillna(df_total71["T"], inplace=True)
df_total71.T_max.fillna(df_total71["T"], inplace=True)

# df_total71[df_total71.Observation == pd.Timestamp("2005-02-01 03:00:00")]

```

**Выберем данные для моделирования: оставим только потенциальные фичи и целевую переменную**

```
In [95]: data71 = (
    df_total71
    .drop(columns=["Station", "Observation", "Closest_station"])
    #     .dropna(how='any')
    .astype(float)
)
data71.head(3)
```

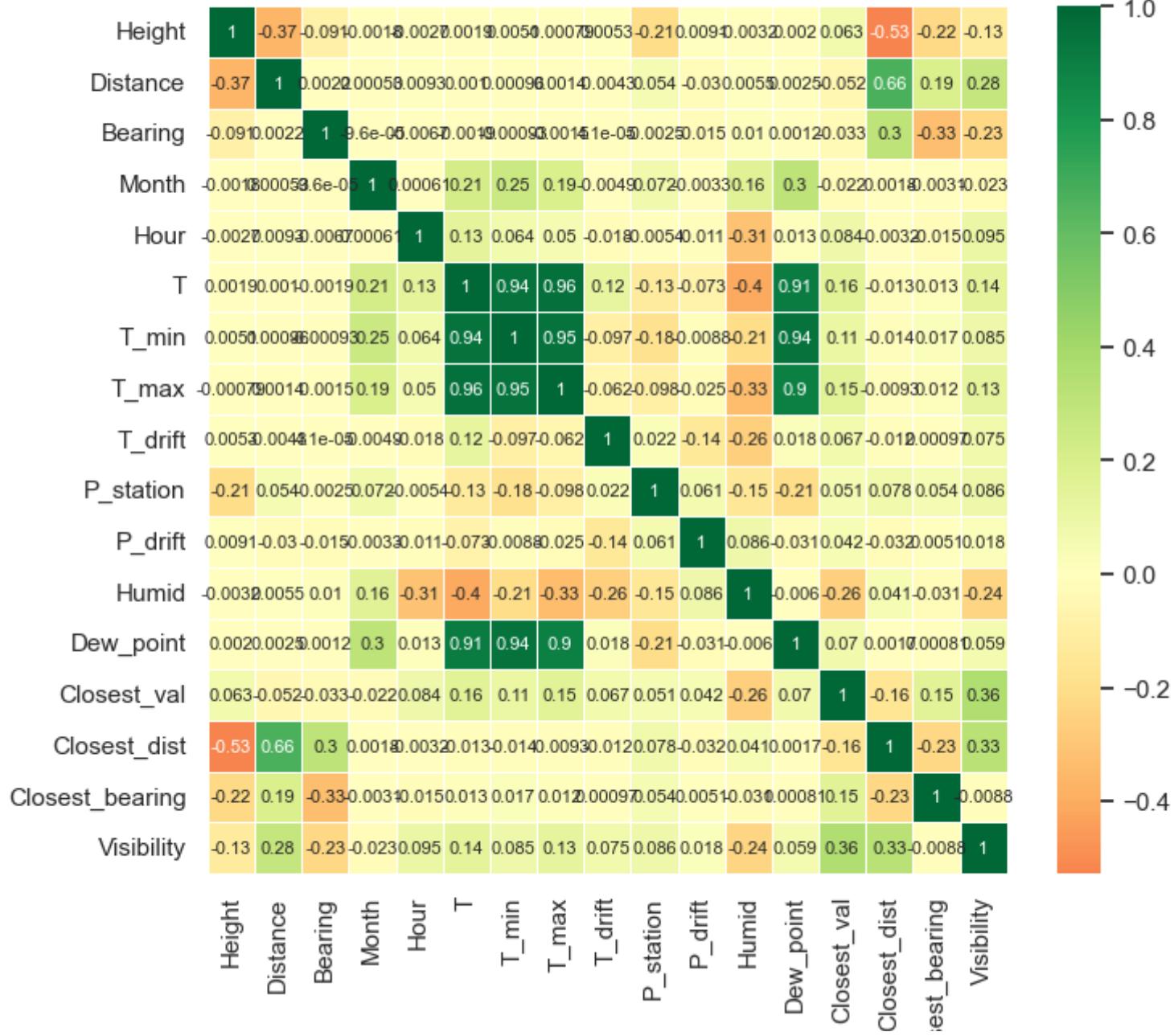
```
Out[95]: Height Distance Bearing Month Hour T T_min T_max T_drift P_station P_drift Humid Dew_point Closest_val C
0 215.0 71.720525 108.834256 6.0 21.0 18.330035 18.330035 27.774218 -5.249743 739.949808 0.001137 81.786605 15.159135 10.0
1 215.0 71.720525 108.834256 6.0 18.0 23.579777 22.158215 27.361772 -3.781994 739.948671 0.035801 53.876986 13.683913 10.0
2 215.0 71.720525 108.834256 6.0 15.0 27.361772 11.047725 27.361772 1.470969 739.912870 -0.731062 35.710055 10.836164 10.0
```

## Выяснение зависимостей

Построим корреляционную матрицу.

```
In [96]: df_corr_matrix = data71.dropna(subset=[ 'Visibility' ]).corr(method='pearson')
fig, ax = plt.subplots(figsize=(8, 7))
ax = sns.heatmap(data=df_corr_matrix, # Выводим график heatmap
                  cmap='RdYlGn', # matplotlib цветовая палитра (желательно расходящаяся)
                  center=0, # центральное значение (для расходящейся цветовой карты):
                           # заметная связь по шкале Чеддока
                  annot=True, # определим подписи значений
                  annot_kws=dict(fontsize=8),
                  linewidths=0.5 # толщина разделяющих линий между ячейками
)
dummy = plt.suptitle(f'Корреляция параметров для модели предсказания {PARAMETER71}', size=12)
plt.show()
```

## Корреляция параметров для модели предсказания Visibility



Как видно из графика ни один из признаков не имеет корреляцию с целевым выше чем слабая. Есть часть признаков (температуры) хорошо коррелирующих между собой. Корреляция с соседней метеостанции выше, чем с другими признаками, но её недостаточно для кригинга, как было показано выше.

Уберём хорошо коррелирующие между собой признаки минимальной и максимальной температуры за 12 часов. Оставим только саму температуру в момент наблюдения

```
In [97]: data71.drop(columns=["T_min", "T_max"], inplace=True)
```

То, что ни один из признаков не имеет сколько-нибудь заметной корреляции представляется несколько странным, впрочем объяснимым многофакторностью и зависимостью дальности видимости от неких локальных параметров, не учитываемых в погодных архивах. Попробуем разделить датасет дальности видимости на классы, согласно приведенной выше шкале или машинным способом. Посмотрим, что будет с корреляцией признаков.

### 7.1.3.5. Модели классификации дальности видимости

Определим классы вручную, по таблице баллов дальности видимости.

```
In [98]: # Определим столбец для хранения признака класса (категории видимости)
# и столбец для хранения признака ранга (балла видимости)
data71 = data71.assign(Cat = np.nan,
                       Rank = np.nan)
# data71
```

```
In [99]: data71.loc[data71.Visibility>=50, "Cat"] = 3 # Excellent
data71.loc[data71.Visibility>=50, "Rank"] = 9

data71.loc[(data71.Visibility>=20) & (data71.Visibility<50), "Cat"] = 2 # Good
data71.loc[(data71.Visibility>=20) & (data71.Visibility<50), "Rank"] = 8

data71.loc[(data71.Visibility>=10) & (data71.Visibility<20), "Cat"] = 1 # Mist
data71.loc[(data71.Visibility>=10) & (data71.Visibility<20), "Rank"] = 7

data71.loc[(data71.Visibility>=4) & (data71.Visibility<10), "Cat"] = 1 # Mist
data71.loc[(data71.Visibility>=4) & (data71.Visibility<10), "Rank"] = 6
```

```

data71.loc[(data71.Visibility>=2) & (data71.Visibility<4), "Cat"] = 1 # Mist
data71.loc[(data71.Visibility>=2) & (data71.Visibility<4), "Rank"] = 5

data71.loc[(data71.Visibility>=1) & (data71.Visibility<2), "Cat"] = 1 # Mist
data71.loc[(data71.Visibility>=1) & (data71.Visibility<2), "Rank"] = 4

data71.loc[(data71.Visibility>=0.5) & (data71.Visibility<1), "Cat"] = 0 # Fog
data71.loc[(data71.Visibility>=0.5) & (data71.Visibility<1), "Rank"] = 3

data71.loc[(data71.Visibility>=0.2) & (data71.Visibility<0.5), "Cat"] = 0 # Fog
data71.loc[(data71.Visibility>=0.2) & (data71.Visibility<0.5), "Rank"] = 2

data71.loc[(data71.Visibility>=0.05) & (data71.Visibility<0.2), "Cat"] = 0 # Fog
data71.loc[(data71.Visibility>=0.05) & (data71.Visibility<0.2), "Rank"] = 1

data71.loc[(data71.Visibility<0.05), "Cat"] = 0 # Fog
data71.loc[(data71.Visibility<0.05), "Rank"] = 0

data71.sample(10)

```

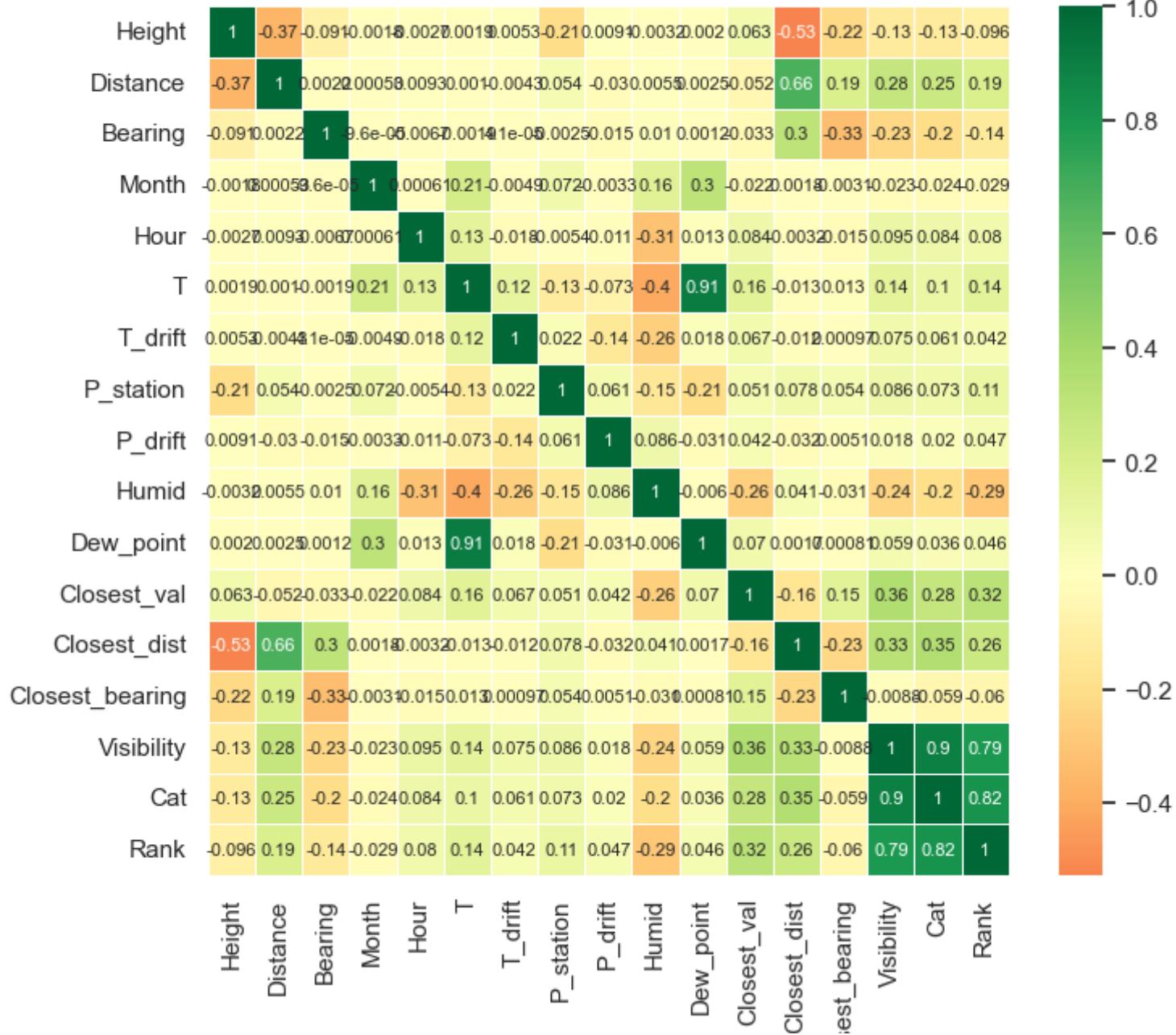
	Height	Distance	Bearing	Month	Hour	T	T_drift	P_station	P_drift	Humid	Dew_point	Closest_val	Closest_dist	Cls
<b>265231</b>	193.0	104.007002	157.684533	6.0	9.0	7.400000	1.900000	737.900000	1.800000	88.000000	5.600000	10.0	46.580297	
<b>538133</b>	185.0	75.614832	292.050594	10.0	0.0	4.600000	-0.100000	752.400000	-0.100000	84.000000	2.100000	10.0	72.307124	
<b>420034</b>	138.0	0.000000	0.000000	7.0	15.0	18.187217	2.487217	743.761277	0.112929	66.852588	11.921294	10.0	27.394472	
<b>102560</b>	138.0	152.988800	36.271056	1.0	15.0	-3.600000	0.500000	741.500000	-0.800000	77.000000	-7.100000	10.0	114.342862	
<b>158225</b>	167.0	40.677898	76.562251	5.0	9.0	13.800000	1.000000	739.000000	0.300000	91.000000	12.300000	4.0	48.595243	
<b>403033</b>	162.0	60.291616	129.908183	12.0	21.0	-5.200000	0.500000	735.700000	-0.100000	88.000000	-6.900000	4.0	48.595243	
<b>179798</b>	167.0	40.677898	76.562251	12.0	18.0	-19.300000	-1.000000	762.800000	-0.800000	86.000000	-21.000000	10.0	48.595243	
<b>219259</b>	185.0	81.696291	183.314261	10.0	0.0	1.800000	-0.300000	763.000000	0.700000	78.000000	-1.600000	10.0	46.580297	
<b>97228</b>	179.0	90.907994	81.187324	7.0	6.0	16.400000	0.700000	748.500000	-0.200000	70.000000	11.000000	20.0	50.469249	
<b>419925</b>	138.0	0.000000	0.000000	7.0	6.0	13.475993	0.396850	748.799148	-0.516099	87.390680	11.421453	10.0	27.394472	

Снова построим корреляционную матрицу, с учётом зависимостей для кассов и рангов видимости.

In [100...]

```
df_corr_matrix = data71.dropna(subset=["Visibility", "Cat", "Rank"]).corr(method='pearson')
fig, ax = plt.subplots(figsize=(8, 7))
ax = sns.heatmap(data=df_corr_matrix, # Выводим график heatmap
                  cmap='RdYlGn', # matplotlib цветовая палитра (желательно расходящаяся)
                  center=0, # центральное значение (для расходящейся цветовой карты):
                           # заметная связь по шкале Чеддока
                  annot=True, # определим подписи значений
                  annot_kws=dict(fontsize=8),
                  linewidths=0.5 # толщина разделяющих линий между ячейками
)
dummy = plt.suptitle(f'Корреляция параметров для модели предсказания {PARAMETER71}', size=12)
plt.show()
```

## Корреляция параметров для модели предсказания Visibility



Посмотрим, какое количество наблюдений приходится на каждую категорию и на каждый балл видимости

```
In [101...]: data71.groupby("Cat").agg('count').iloc[:,0]
data71.groupby("Rank").agg('count').iloc[:,0]
```

```
Out[101]: Cat
0.0      4558
1.0    373902
2.0    139595
3.0     23994
Name: Height, dtype: int64
Out[101]: Rank
0.0        5
1.0      159
2.0      892
3.0     3502
4.0     4038
5.0     9206
6.0    46896
7.0   313762
8.0   139595
9.0     23994
Name: Height, dtype: int64
```

Учитывая, что 0 баллу видимости соответствует всего 5 наблюдений, мы можем столкнуться со сложностями при обучении классификации. Поэтому, остановим выбор на классификации по категориями видимости.

Произведём отбор параметров и их трансформацию

```
In [102...]: # Определим фичи и целевую переменную
# Поскольку данные имеют NaN в целевой переменной, удалим NaN
X = data71.dropna(how='any').iloc[:, :-3]
y = data71.dropna(how='any').Cat.astype('int')
```

```
In [103...]: # Преобразование исходных данных, выбор лучшего набора параметров, уменьшение размерности
q_transformer = QuantileTransformer( # Объект трансформации данных
    n_quantiles=1000,
```

```
        output_distribution='uniform',
        ignore_implicit_zeros=False,
        subsample=100000,
        random_state=56,
        copy=True)

gu_selector = GenericUnivariateSelect( # Объект выбора фичей
    score_func=mutual_info_classif,
    mode='k_best',
    param=8)

tsvd = TruncatedSVD( # Объект уменьшения размерности tSVD
    n_components=4,
    algorithm='randomized',
    n_iter=5,
    n_oversamples=10,
    power_iteration_normalizer='auto',
    random_state=56,
    tol=0.0)

X_trans = q_transformer.fit_transform(X)
X_sel = gu_selector.fit_transform(X_trans, y)
X_svd = tsvd.fit_transform(X_sel)

X_svd.shape
```

Out[103]: (542049, 4)

In [104...]

```
q_transformer.get_feature_names_out()
gu_selector.get_feature_names_out()
tsvd.get_feature_names_out()
```

Out[104]: array(['Height', 'Distance', 'Bearing', 'Month', 'Hour', 'T', 'T\_drift',
 'P\_station', 'P\_drift', 'Humid', 'Dew\_point', 'Closest\_val',
 'Closest\_dist', 'Closest\_bearing'], dtype=object)

Out[104]: array(['x0', 'x1', 'x2', 'x7', 'x9', 'x11', 'x12', 'x13'], dtype=object)

Out[104]: array(['truncatedsvd0', 'truncatedsvd1', 'truncatedsvd2', 'truncatedsvd3'],
 dtype=object)

In [105...]

```
# Разделение датасета на тренинговую и валидационную часть: используем датасет после уменьшения размерности
x_train, x_test, y_train, y_test = train_test_split(
    X_svd, y,
    test_size=0.4,
```

```
train_size=0.6,
random_state=56,
shuffle=True,
stratify=y)
# stratify, для сбалансированного распределения классов между разделами
x_train.shape, x_test.shape, y_train.shape, y_test.shape
```

Out[105]: ((325229, 4), (216820, 4), (325229,), (216820,))

## Создаём модели обучения для решения задачи классификации ансамблем алгоритмов

In [106...]

```
# Создаём объекты для каждого из алгоритмов обучения
rfcl = RandomForestClassifier( # Random Forest Classifier
    n_estimators=300, criterion='gini', max_depth=None,
    min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0,
    max_features='sqrt', max_leaf_nodes=None,
    min_impurity_decrease=0.0, bootstrap=True, oob_score=False,
    n_jobs=6, random_state=56, verbose=1, warm_start=False,
    class_weight='balanced_subsample', ccp_alpha=0.0, max_samples=None)

mlpcl = MLPClassifier( # Machine Learning Classifier
    hidden_layer_sizes=(100,), activation='relu', solver='adam', alpha=0.0001, batch_size='auto',
    learning_rate='constant', learning_rate_init=0.001, power_t=0.5, max_iter=200, shuffle=True,
    random_state=56, tol=0.0001, verbose=1,
    warm_start=False, momentum=0.9, nesterovs_momentum=True, early_stopping=True, validation_fraction=0.1,
    beta_1=0.9, beta_2=0.999, epsilon=1e-08, n_iter_no_change=10, max_fun=15000)

hgbcl = HistGradientBoostingClassifier( # Histogram-based Gradient Boosting Classification Tree
    loss='log_loss', learning_rate=0.1, max_iter=100, max_leaf_nodes=31, max_depth=None,
    min_samples_leaf=20, l2_regularization=0.0, max_bins=255,
    categorical_features=None, monotonic_cst=None, warm_start=False,
    early_stopping='auto', scoring='loss', validation_fraction=0.1,
    n_iter_no_change=10, tol=1e-07, verbose=1, random_state=56
)

# Объект разбики массива данных на слои для кросс-валидации
skfcl = StratifiedKFold(n_splits=3, shuffle=True, random_state=56)

# Зададим мета-алгоритм
metacl = AdaBoostClassifier(
    base_estimator=DecisionTreeClassifier(), n_estimators=50,
    learning_rate=1.0, algorithm='SAMME.R', random_state=56)
```

```
list_cl_models = [rfcl, mlpcl, hgbcl] # Список базовых моделей-классификаторов
```

```
In [107...]  
# # Параметры базовых алгоритмов для поиска оптимальных  
# dict_rfcl_params = {  
#     'max_features': ['sqrt', 'Log2', None]}  
  
# dict_mlpcl_params = {  
#     'hidden_layer_sizes':[(100,), (200, 100), (300, 200, 100), (300, 250, 200, 150, 100)]}  
  
# dict_hgbcl_params = {  
#     'Loss': ['log_loss', 'auto'],  
#     'Learning_rate': np.arange(0.1, 0.4, 0.1),  
#     'max_iter': np.arange(100, 400, 100)}  
  
# list_cl_params = [dict_rfcl_params, dict_mlpcl_params, dict_hgbcl_params] # список словарей параметров  
  
# # Определим параметры для мета-алгоритма AdaBoostClassifier  
# dict_metacl_params = {  
#     'n_estimators': [50, 100, 200],  
#     'Learning_rate': np.arange(0.1, 0.6, 0.1)}
```

```
In [108...]  
# %time  
# # Произведём поиск оптимальных параметров для базовых алгоритмов обучения  
  
# list_cl_best_params = [] # список лучших параметров  
# meta_mtrx_cl = np.empty((x_test.shape[0], len(list_cl_models))) # матрица для мета-алгоритма (для след. пункта)  
  
# n=0 # счётчик  
# for model, params in zip(list_cl_models, list_cl_params): # цикл по моделям и их словарям параметров  
#     print(f"Grid-searching for {model}")  
#     grid = GridSearchCV(  
#         estimator=model, param_grid=params,  
#         scoring='precision_macro',  
#         n_jobs=6, # 6 из 8 логических процессоров на моем компьютере  
#         refit=True,  
#         cv=skfcl, verbose=4, # для вывода информации о работе кода  
#         pre_dispatch='2*n_jobs', error_score=np.nan, return_train_score=False  
#         ) # определяем grid поиск, n_jobs=-1 для мультипроцес.  
  
#     grid.fit(x_train, y_train) # обучаем grid  
#     list_cl_best_params.append(grid.best_params_) # дополняем лучшие параметры в список  
#     meta_mtrx_cl[:, n] = grid.predict(x_test) # Заполнение мета-матрицы предиктами базовых моделей на валид. данных
```

```
#     list_cl_models[n] = model # записываем обученную модель в список моделей
#
#     print(f"\n{model}: {grid.best_params_}") # выводим название алгоритма и его лучшие параметры
#     print(f'Scoring: {grid.score(x_test, y_test)}\n') # выводим лучшее значение метрики
#     n += 1
```

Результат кода выше:

```
RandomForestClassifier(class_weight='balanced_subsample', n_estimators=300, n_jobs=6, random_state=56, verbose=1): {'max_features': None}
Scoring: 0.6254276035165822
```

```
MLPClassifier(early_stopping=True, random_state=56, verbose=1): {'hidden_layer_sizes': (300, 200, 100)} Scoring: 0.7991212225314982
```

```
HHistGradientBoostingClassifier(random_state=56, verbose=1): {'learning_rate': 0.1, 'loss': 'log_loss', 'max_iter': 100} Scoring: 0.6737260209970422
```

CPU times: total: 3h 51min 19s Wall time: 3h 32min 54s

```
In [109...]: # # Выведем итог
# for model, params in zip(list_cl_models, list_cl_best_params):
#     print(f"{model}: \n{params}\n")

# meta_mtrx_cl
```

```
In [110...]: # list_cl_best_params
```

```
In [111...]: list_cl_best_params = [
    {'max_features': None},
    {'hidden_layer_sizes': (300, 200, 100)},
    {'learning_rate': 0.1, 'loss': 'log_loss', 'max_iter': 100}
]
```

```
In [112...]: # # Произведём поиск оптимальных параметров для мета-алгоритма
# meta_grid_cl = GridSearchCV(
#     metacl, dict_metacl_params,
#     scoring='roc_auc_ovr', n_jobs=6, # 6 из 8 логических процессоров на моем компьютере
#     refit=True,
#     cv=skfcl, verbose=3, # для вывода информации о работе кода
#     pre_dispatch='2*n_jobs', error_score=np.nan, return_train_score=False
```

```
#      ) # определяем grid поиск  
# meta_grid_cl.fit(meta_mtrx_cl, y_test) # обучаем grid
```

```
In [113...]  
# meta_cl_best_params = meta_grid_cl.best_params_  
# meta_cl_best_params  
# meta_grid_cl.best_score_
```

```
In [114...]  
# # установим лучшие параметры для алгоритмов  
# for model, best_params in zip(list_cl_models, list_cl_best_params):  
#     model.set_params(**best_params)  
  
# metacl.set_params(**meta_cl_best_params)
```

```
In [119...]  
# Создаём стэкинг из базовых алгоритмов и метаалгоритма  
stkcl = StackingClassifier(  
    estimators=[('rfcl', rfcl), ('mlpcl', mlpcl), ('hgbcl', hgbcl)], final_estimator=metacl,  
    cv=skfcl, stack_method='auto',  
    n_jobs=6, passthrough=False, verbose=1)
```

```
In [116...]  
# %%time  
# # Обучим стэкинг  
# stkcl.fit(x_train, y_train)
```

## Сохраним обученные модели классификации

```
In [117...]  
# # Создадим название директории  
# model_path = f'data/models/{PARAMETER71}'  
# # Создадим директорию  
# makedirs(model_path, exist_ok=True)  
  
# # Создадим регулярное выражение для вычисления названия алгоритма из модели  
# reg_expr1 = r'(\.+)' # Все символы кроме новой строки только 1 раз, от открывающей скобки до конца  
# # компилируем регулярное выражение, определив точку как универсальный любой символ  
# regex1 = re.compile(reg_expr1, flags=re.DOTALL)  
  
# # Запишем базовые модели классификации в файлы joblib  
# for model in list_cl_models:  
#     model_name = re.sub(regex1, '', str(model)) # удаляем в строке описания модели скобку и все знаки после неё  
#     file_name = f'{model_path}{model_name}.joblib'  
#     print(f'Saving {file_name} ->', end=' ')  
#     dummy = joblib.dump(model, file_name, compress=9) # добавим "пустую" переменную, для избежания лишнего вывода
```

```

#     print('DONE! ')

# # Запишем мета-алгоритм и стэкинг классификации в файлы joblib
# for model in [metacl, stkcl]:
#     model_name = re.sub(regex1, '', str(model)) # удаляем в строке описания модели скобку и все знаки после неё
#     file_name = f'{model_path}{model_name}.joblib'
#     print(f'Saving {file_name} ->', end=' ')
#     dummy = joblib.dump(model, file_name, compress=5) # добавим "пустую" переменную, для избежания лишнего вывода
#     print('DONE!')

```

## Восстановим модели классификации

In [120...]

```

# Название директории
model_path = f'data/models/{PARAMETER71}/'

# Создадим регулярное выражение для вычисления названия алгоритма из модели
reg_expr1 = r'(\.+)' # Все символы кроме новой строки только 1 раз, от открывающей скобки до конца
# компилируем регулярное выражение, определив точку как универсальный любой символ
regex1 = re.compile(reg_expr1, flags=re.DOTALL)

# Восстановим базовые модели классификации из файлов joblib
n=0
for model in list_cl_models:
    model_name = re.sub(regex1, '', str(model)) # заменяем в строке модели скобку и все знаки после неё
    file_name = f'{model_path}{model_name}.joblib'
    print(f'Loading {file_name} ->', end=' ')
    list_cl_models[n] = (joblib.load(file_name)) # запишем восстановленную модель в список восстановленных моделей
    print('DONE!')

# Восстановим мета-алгоритм классификации из файлов joblib
model_name = re.sub(regex1, '', str(metacl)) # удаляем в строке описания модели скобку и все знаки после неё
file_name = f'{model_path}{model_name}.joblib'
print(f'Loading {file_name} ->', end=' ')
metacl = joblib.load(file_name) # загрузим восстановленную модель
print('DONE!')

# Восстановим стэкинг классификации из файлов joblib
model_name = re.sub(regex1, '', str(stkcl)) # удаляем в строке описания модели скобку и все знаки после неё
file_name = f'{model_path}{model_name}.joblib'
print(f'Loading {file_name} ->', end=' ')
stkcl = joblib.load(file_name) # загрузим восстановленную модель
print('DONE!')

```

```
Loading data/models/Visibility/HistGradientBoostingClassifier.joblib -> DONE!
Loading data/models/Visibility/MLPClassifier.joblib -> DONE!
Loading data/models/Visibility/HistGradientBoostingClassifier.joblib -> DONE!
Loading data/models/Visibility/AdaBoostClassifier.joblib -> DONE!
Loading data/models/Visibility/StackingClassifier.joblib -> DONE!
```

## Предикты и метрики качества классификации

In [121...]

```
%%time
# Получим предикты для обучающей выборки
y_pred_train = stkcl.predict(x_train)

[Parallel(n_jobs=6)]: Using backend ThreadingBackend with 6 concurrent workers.
[Parallel(n_jobs=6)]: Done 38 tasks      | elapsed:    2.2s
[Parallel(n_jobs=6)]: Done 188 tasks      | elapsed:   10.2s
[Parallel(n_jobs=6)]: Done 300 out of 300 | elapsed:   16.1s finished
CPU times: total: 2min 23s
Wall time: 38 s
```

In [122...]

```
# Выведем метрики качества для обучающей выборки
print("ЭТАП ОБУЧЕНИЯ:")
print(f"Stacking score: {stkcl.score(x_train, y_train)}")
print(f"Precision score: {precision_score(y_train, y_pred_train, average='macro')}")
print(f"Recall score: {recall_score(y_train, y_pred_train, average='macro')}")
print(f"F1 score: {f1_score(y_train, y_pred_train, average='macro')}")
rocauc_tr = roc_auc_score(
    y_test, stkcl.predict_proba(x_test),
    average='macro', sample_weight=None,
    max_fpr=None, multi_class='ovr', labels=None)
print(f"ROC-AUC score: {rocauc_tr}")

# Построим графики кривой ROC для каждого класса
n_classes = len(pd.unique(y)) # Количество классов в датасете
fig, ax = plt.subplots(figsize=(5,5))

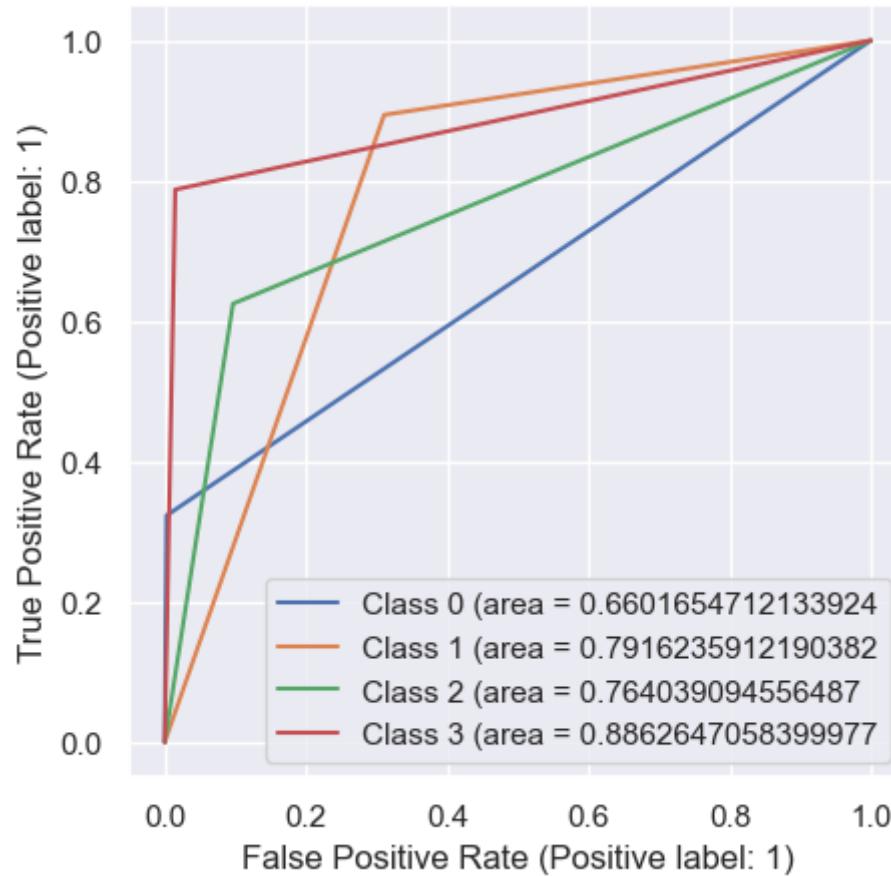
for i in range(n_classes):
    # Выделим значения y для каждого класса
    y_train_bin, y_pred_train_bin = np.where(y_train.eq(i), 1, 0), np.where(y_pred_train==i, 1, 0)
    g = RocCurveDisplay.from_predictions(
        y_train_bin, y_pred_train_bin,
        drop_intermediate=False,
        label=f'Class {i} (area = {roc_auc_score(y_train_bin, y_pred_train_bin)})',
```

```
    ax=ax)
plt.show()
```

ЭТАП ОБУЧЕНИЯ:

```
[Parallel(n_jobs=6)]: Using backend ThreadingBackend with 6 concurrent workers.
[Parallel(n_jobs=6)]: Done  38 tasks      | elapsed:   2.0s
[Parallel(n_jobs=6)]: Done 188 tasks      | elapsed:   9.1s
[Parallel(n_jobs=6)]: Done 300 out of 300 | elapsed:  14.2s finished
Stacking score: 0.8150964397393836
Precision score: 0.6966920974535211
Recall score: 0.6572447258631705
F1 score: 0.6701842055849249
```

```
[Parallel(n_jobs=6)]: Using backend ThreadingBackend with 6 concurrent workers.
[Parallel(n_jobs=6)]: Done  38 tasks      | elapsed:   1.3s
[Parallel(n_jobs=6)]: Done 188 tasks      | elapsed:   7.2s
[Parallel(n_jobs=6)]: Done 300 out of 300 | elapsed:  11.7s finished
ROC-AUC score: 0.8649501721666479
```



In [123...]

```
# Получим предикты для x_test
y_predict = stkcl.predict(x_test)
```

```
[Parallel(n_jobs=6)]: Using backend ThreadingBackend with 6 concurrent workers.
[Parallel(n_jobs=6)]: Done 38 tasks      | elapsed:    1.3s
[Parallel(n_jobs=6)]: Done 188 tasks      | elapsed:    7.5s
[Parallel(n_jobs=6)]: Done 300 out of 300 | elapsed:   12.0s finished
```

In [124...]

```
# Выведем метрики качества
print("ЭТАП ВАЛИДАЦИИ:")
print(f"Stacking score: {stkcl.score(x_test, y_test)}")
print(f"Precision score: {precision_score(y_test, y_predict, average='macro')}")
print(f"Recall score: {recall_score(y_test, y_predict, average='macro')}")
print(f"F1 score: {f1_score(y_test, y_predict, average='macro')}")
r_a = roc_auc_score(
```

```

y_test, stkcl.predict_proba(x_test),
average='macro', sample_weight=None,
max_fpr=None, multi_class='ovr', labels=None)
print(f"ROC-AUC score: {r_a}")

# Постротим графики кривой ROC для каждого класса
n_classes = len(pd.unique(y)) # Количество классов в датасете
fig, ax = plt.subplots(figsize=(5,5))

for i in range(n_classes):
    # Выделим значения у для каждого класса
    y_test_binar, y_predict_binar = np.where(y_test.eq(i), 1, 0), np.where(y_predict==i, 1, 0)
    g = RocCurveDisplay.from_predictions(
        y_test_binar, y_predict_binar,
        drop_intermediate=False,
        label=f'Class {i} (area = {roc_auc_score(y_test_binar, y_predict_binar)})',
        ax=ax)
plt.show()

```

#### ЭТАП ВАЛИДАЦИИ:

```

[Parallel(n_jobs=6)]: Using backend ThreadingBackend with 6 concurrent workers.
[Parallel(n_jobs=6)]: Done  38 tasks      | elapsed:   1.3s
[Parallel(n_jobs=6)]: Done 188 tasks      | elapsed:   7.7s
[Parallel(n_jobs=6)]: Done 300 out of 300 | elapsed:  12.2s finished

```

Stacking score: 0.8015404482981274

Precision score: 0.6482985828850027

Recall score: 0.6142542782237876

F1 score: 0.624842770991383

```

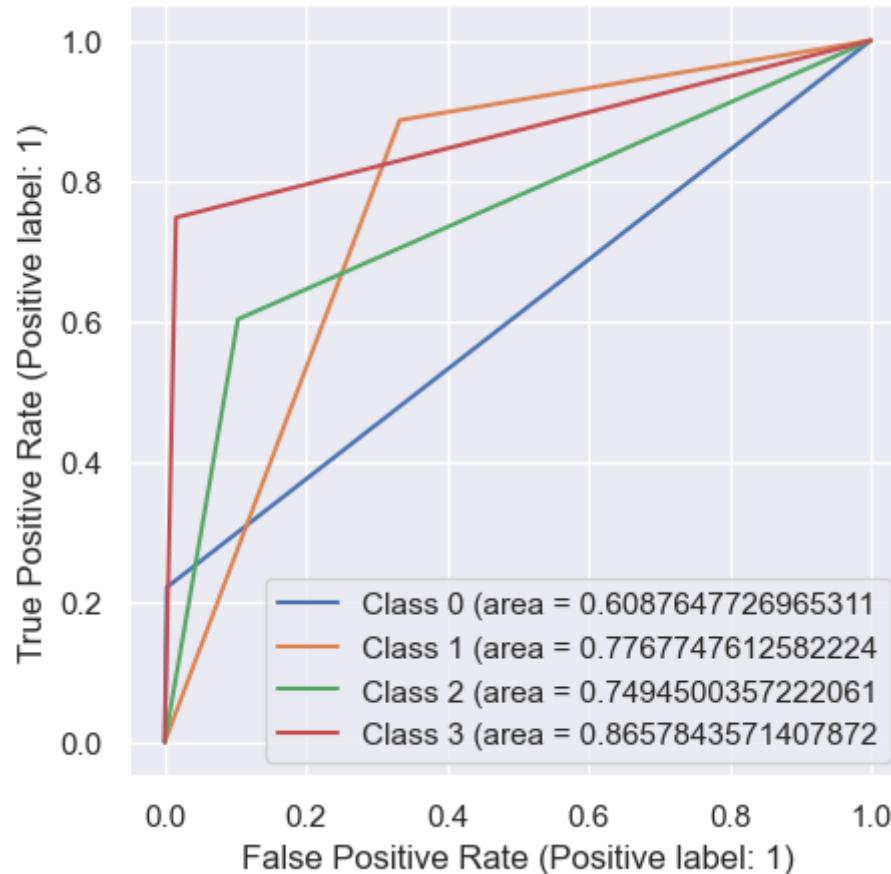
[Parallel(n_jobs=6)]: Using backend ThreadingBackend with 6 concurrent workers.
[Parallel(n_jobs=6)]: Done  38 tasks      | elapsed:   1.3s
[Parallel(n_jobs=6)]: Done 188 tasks      | elapsed:   7.6s
[Parallel(n_jobs=6)]: Done 300 out of 300 | elapsed:  12.2s finished

```

```

ROC-AUC score: 0.8649501721666479

```



Произведём классификацию для рабочих данных (там, где дальность видимости неопределена).

```
In [125]: data71[(pd.isna(data71.Visibility))].sample(5, random_state=56)
```

Out[125]:	Height	Distance	Bearing	Month	Hour	T	T_drift	P_station	P_drift	Humid	Dew_point	Closest_val	Closest_dist	Clos
23975	215.0	71.720525	108.834256	3.0	0.0	0.710429	-3.170200	754.188398	0.370388	48.897517	-8.800919	10.0	26.627460	
151606	138.0	152.988800	36.271056	4.0	21.0	2.397962	-4.540011	756.035658	-0.504310	54.865836	-5.746090	15.0	109.414691	
322278	175.0	99.058072	126.256571	4.0	9.0	3.011428	2.175797	740.287471	-0.192936	67.190430	-2.473477	10.0	39.077968	
42177	215.0	71.720525	108.834256	1.0	18.0	-6.703833	0.566242	765.722742	1.481808	81.116060	-9.392701	10.0	26.627460	
26125	215.0	71.720525	108.834256	7.0	6.0	16.621061	-1.151526	738.108530	0.306384	78.268072	12.817978	10.0	26.627460	

In [126...]

```
# Определим фичи
# Поскольку данные _drift в 1 и 2 рядах от начала архива уже не содержат NaN, не будем удалять NaN
X_wrk = data71[(pd.isna(data71.Visibility))].iloc[:, :-3]
X_wrk.shape
```

Out[126]:

```
(167793, 14)
```

In [127...]

```
# Сохраним индекс для прогнозируемых значений.
idx_wrk = X_wrk.index
```

In [128...]

```
# Применим обученную трансформацию данных
X_trans_w = q_transformer.fit_transform(X_wrk)
# Выберем те же фичи, что и на этапах обучения/валидации
cols = list(gu_selector.get_support(indices=True)) # Список горизонтальных индексов для выборки фичей
X_sel_w = X_trans_w[:, cols]
# Выполним SVD преобразование
X_svd_w = tsvd.fit_transform(X_sel_w)

X_svd_w.shape
```

Out[128]:

```
(167793, 4)
```

In [129...]

```
# Выполняем предсказание значений классов видимости для рабочей части датасета
y_predict_w = stkcl.predict(X_svd_w)
```

```
[Parallel(n_jobs=6)]: Using backend ThreadingBackend with 6 concurrent workers.
[Parallel(n_jobs=6)]: Done 38 tasks      | elapsed:    0.2s
[Parallel(n_jobs=6)]: Done 188 tasks      | elapsed:    1.4s
[Parallel(n_jobs=6)]: Done 300 out of 300 | elapsed:    2.5s finished
```

```
In [130]: y_predict_w
```

```
Out[130]: array([1, 1, 0, ..., 2, 1, 1])
```

```
In [131... # Создадим серию данных из предсказаний и обновим датасет
```

```
ser_y_predict = pd.Series(y_predict_w, index=idx_wrk, name="Cat")  
data71.update(ser_y_predict, overwrite=False)
```

```
In [132... data71[pd.isna(data71.Cat)]
```

```
Out[132]: Height Distance Bearing Month Hour T T_drift P_station P_drift Humid Dew_point Closest_val Closest_dist Closest_bearing Visibility Cat I
```

Строк без указания категории дальности видимости больше не осталось.

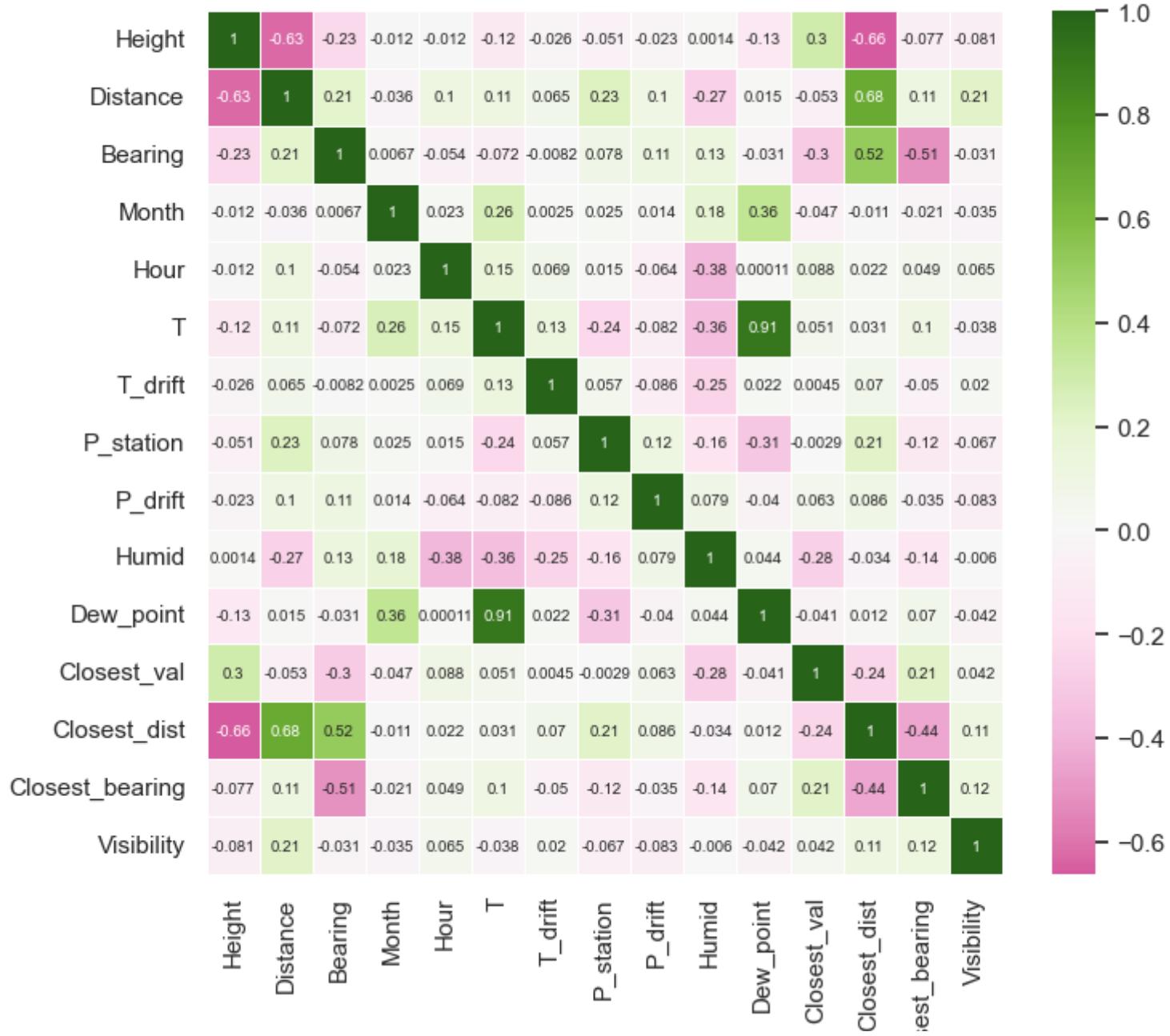
### 7.1.3.6. Модели регрессии для предсказания дальности видимости по каждому классу отдельно

**Проанализируем корреляцию параметров внутри категорий видимости**

```
In [133... # По отсортированным уникальным значениям категорий, кроме NaN (индекс 0)  
for cat in set(data71.dropna(how='any').Cat):  
    if data71[data71.Cat == cat].shape[0] == 0:  
        continue # если данному классу соответствует 0 наблюдений - следующий класс  
    # Составляем матрицу корреляции, выбираем все столбцы до Visibility включительно  
    df_corr_matrix = data71[data71.Cat == cat].loc[:, : "Visibility"].corr(method='pearson')  
    fig, ax = plt.subplots(figsize=(8, 7))  
    ax = sns.heatmap(data=df_corr_matrix, # Выводим график heatmap  
                      cmap='PiYG', # matplotlib цветовая палитра (желательно расходящаяся)  
                      center=0, # центральное значение (для расходящейся цветовой карты):  
                      # заметная связь по шкале Чеддока  
                      annot=True, # определим подписи значений  
                      annot_kws=dict(fontsize=7),  
                      linewidths=0.5 # толщина разделяющих линий между ячейками  
    )  
    dummy = plt.suptitle(f'Корреляция параметров для модели предсказания {PARAMETER71} категории {cat:.0f}\n' +  
                         f'Количество наблюдений = {data71[data71.Cat == cat].shape[0]}', size=10  
    )
```

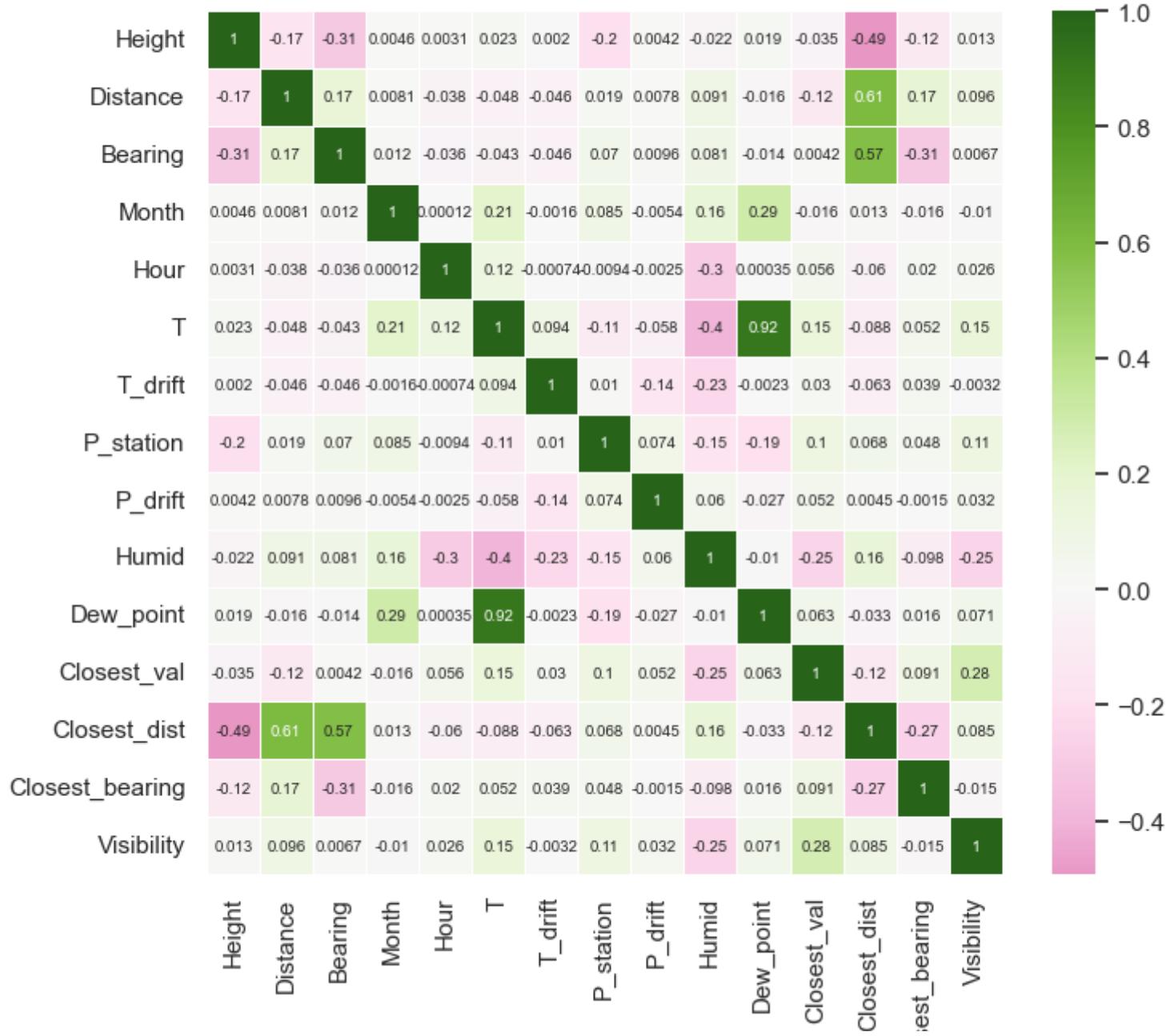
```
plt.show()
```

Корреляция параметров для модели предсказания Visibility категории 0  
Количество наблюдений = 18405



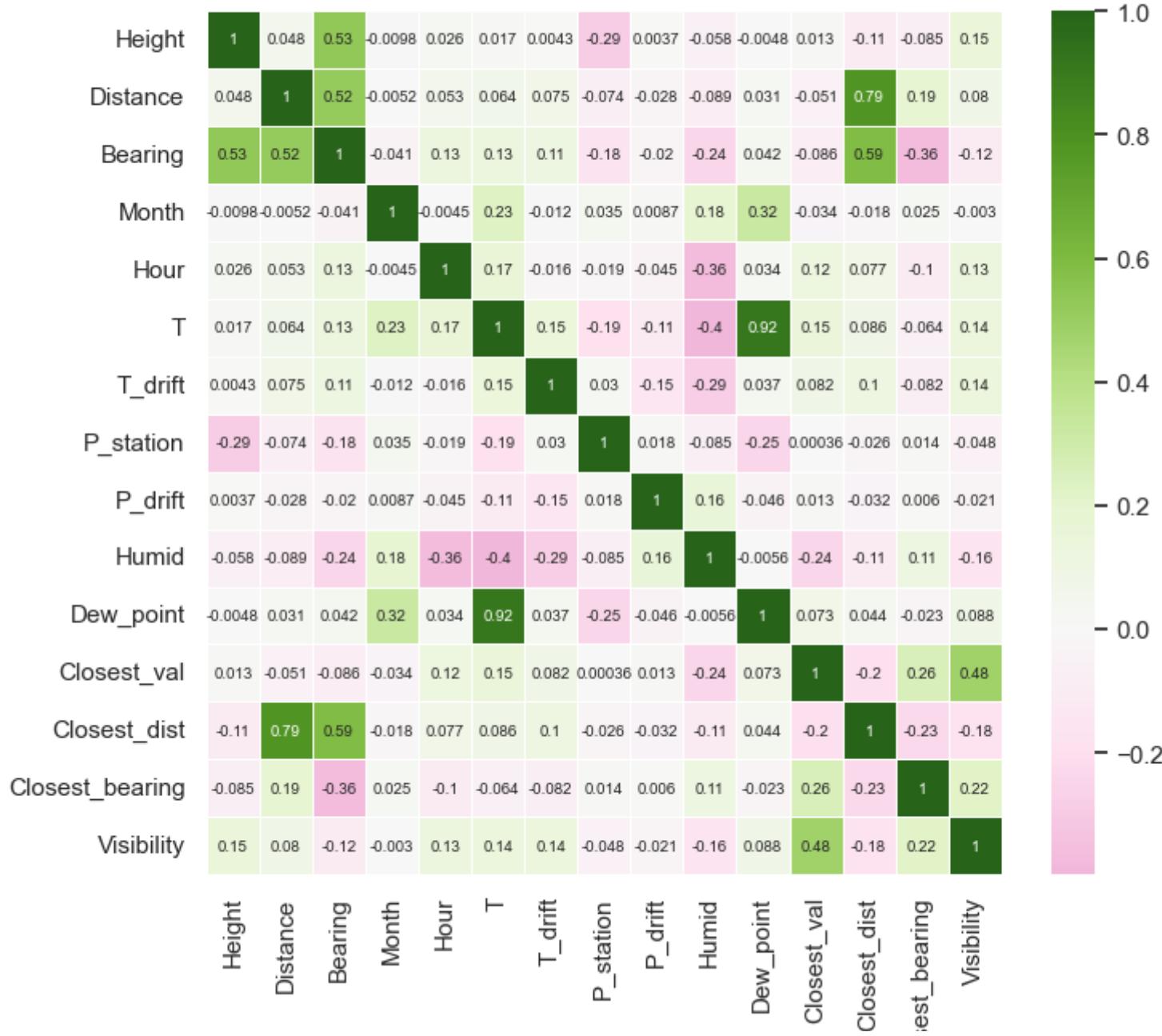


Корреляция параметров для модели предсказания Visibility категории 1  
Количество наблюдений = 462359



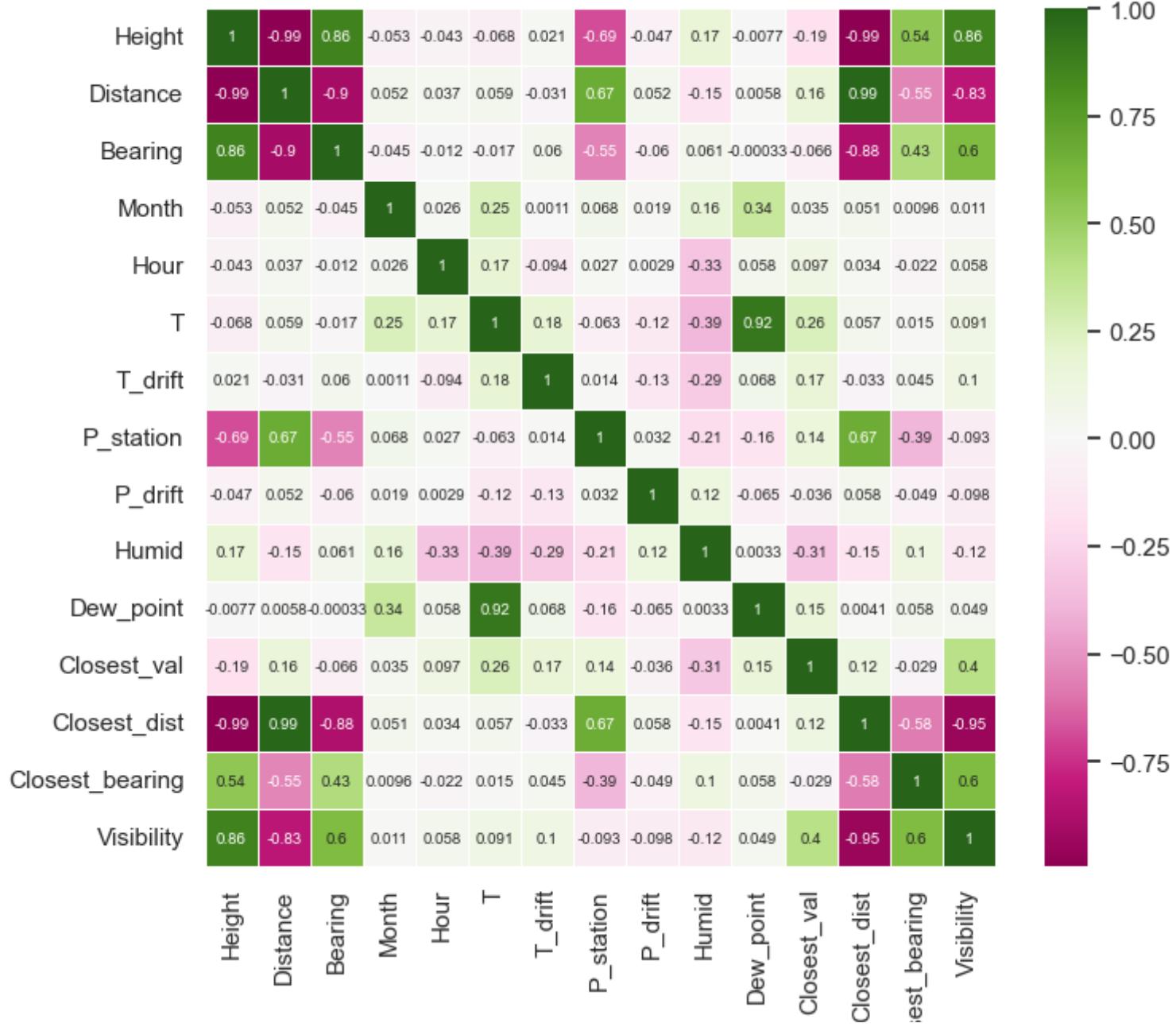


Корреляция параметров для модели предсказания Visibility категории 2  
Количество наблюдений = 199709





Корреляция параметров для модели предсказания Visibility категории 3  
Количество наблюдений = 29369



Как видно из графиков, корреляция параметров различается между классами категорий видимости.

### Разделяем датасет на классы для моделирования значения дальности видимости по каждому классу отдельно

In [134...]

```
# Определяем словари для хранения фичей и целевой переменной по категориям видимости
dict_X = dict()
dict_y = dict()
# Список значений категорий видимости
list_cat = list(set(data71.dropna(how='any').Cat.astype('int')))

# В цикле произведём разбиения датасета на части, сообразно классам с выделением фичей и целевой переменной
for cat in list_cat:
    dict_X[cat] = data71[data71.Cat == cat].dropna(how='any').iloc[:, :-3]
    dict_y[cat] = data71[data71.Cat == cat].dropna(how='any').Visibility
    dict_X[cat].shape, dict_y[cat].shape
```

Out[134]: ((4558, 14), (4558,))

Out[134]: ((373902, 14), (373902,))

Out[134]: ((139595, 14), (139595,))

Out[134]: ((23994, 14), (23994,))

### Создаём модели обучения для решения задачи нелинейной регрессии ансамблем алгоритмов

In [135...]

```
# Создаём объекты для каждого из алгоритмов обучения,
rfrg = RandomForestRegressor( # Random Forest Regressor
    n_estimators=300, criterion='squared_error', max_depth=None,
    min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0,
    max_features=1, max_leaf_nodes=None,
    min_impurity_decrease=0.0, bootstrap=True, oob_score=False,
    n_jobs=6, random_state=56, verbose=1, warm_start=False,
    ccp_alpha=0.0, max_samples=None
)
mlprg = MLPRegressor( # Machine Learning Regressor
    hidden_layer_sizes=(100,), activation='relu', solver='adam', alpha=0.0001, batch_size='auto',
```

```

        learning_rate='constant', learning_rate_init=0.001, power_t=0.5, max_iter=400, shuffle=True,
        random_state=56, tol=0.0005, verbose=1,
        warm_start=False, momentum=0.9, nesterovs_momentum=True, early_stopping=True, validation_fraction=0.1,
        beta_1=0.9, beta_2=0.999, epsilon=1e-08, n_iter_no_change=10, max_fun=15000
    )

hgbrg = HistGradientBoostingRegressor( # Histogram-based Gradient Boosting Regressor Tree
    loss='squared_error', quantile=None, learning_rate=0.1, max_iter=400, max_leaf_nodes=31, max_depth=None,
    min_samples_leaf=20, l2_regularization=0.0, max_bins=255,
    categorical_features=None, monotonic_cst=None, warm_start=False,
    early_stopping='auto', scoring='loss', validation_fraction=0.1,
    n_iter_no_change=10, tol=1e-07, verbose=1, random_state=56
)

# Объект разбики массива данных на слои для кросс-валидации
skfrg = StratifiedKFold(n_splits=3, shuffle=True, random_state=56)

list_rg_models = [rfrg, mlprg, hgbrg] # Список базовых моделей

# Заполним словарь для хранения объектов базовых алгоритмов для каждой категории видимости.
dict_rg_base_models = dict()
for cat in list_cat:
    dict_rg_base_models[cat] = deepcopy(list_rg_models)

# Зададим мета-алгоритм
metarg = AdaBoostRegressor(
    base_estimator=DecisionTreeRegressor(), n_estimators=50,
    learning_rate=1.0, loss='linear', random_state=56)

# Заполним словарь для хранения объектов мета-алгоритма для каждой категории видимости.
dict_metarg = dict()
for cat in list_cat:
    dict_metarg[cat] = deepcopy(metarg)

# Создадим объект K-fold, так как Stratified K-fold предназначен для разбиения классов.
# А в нашем случае, целевая переменная - непрерывная величина
kfrg = KFold(n_splits=3, shuffle=True, random_state=56)

```

In [136...]

```

# Параметры базовых алгоритмов обучения для grid_search
dict_rfrg_params = {
    'max_features': ['sqrt', 'log2']}

```

```

dict_mlprg_params = {
    'hidden_layer_sizes':[(200, 100), (300, 200, 100), (500, 400, 300, 200, 100)]}

dict_hgbrg_params = {
    'learning_rate': np.arange(0.1, 0.4, 0.1)}

list_rg_params = [dict_rfgrg_params, dict_mlprg_params, dict_hgbrg_params] # список словарей параметров

# Определим параметры для мета-алгоритма AdaBoostClassifier
dict_metarg_params = {
    'n_estimators': [50, 100, 200],
    'learning_rate': np.arange(0.1, 0.6, 0.1),
    'loss':['linear', 'square', 'exponential']
}

```

В циклах произведём трансформацию данных, подбор параметров, обучение и валидацию для каждого алгоритма

In [137...]

```
# Создаём словари для хранения получаемых объектов
dict_X_trans = dict()
dict_X_sel = dict()
dict_X_svd = dict()
dict_rg_best_params = dict()
dict_metarg = dict()
dict_rg_meta_mtrix = dict()
dict_train_test = dict()
```

In [138...]

```
# Заменим параметр критерия отбора фичей для GenericUnivariateSelect с классифицирующего на регрессионный
gu_selector.set_params(**{'score_func': mutual_info_regression})
```

Out[138]:

```
▼
          GenericUnivariateSelect
GenericUnivariateSelect(mode='k_best', param=8,
                        score_func=<function mutual_info_regression at 0x000002568FD3FEE0>)
```

In [139...]

```
%time
for cat in list_cat:
    print(f'Transforming features for class: {cat}')
    # Произведём трансформацию данных и отбор параметров, используя определенные выше трансформеры
    dict_X_trans[cat] = q_transformer.fit_transform(dict_X[cat])
    dict_X_sel[cat] = gu_selector.fit_transform(dict_X_trans[cat], dict_y[cat])
    dict_X_svd[cat] = tsvd.fit_transform(dict_X_sel[cat])
```

```

dict_X_svd[cat].shape
print(f'Train-test splitting for class: {cat}')
# Разделение датасета на тренинговую и валидационную часть: используем датасет после уменьшения размерности
x_train, x_test, y_train, y_test = train_test_split(
    dict_X_svd[cat], dict_y[cat],
    test_size=0.4,
    train_size=0.6,
    random_state=56,
    shuffle=True
)
dict_train_test[cat] = [x_train, x_test, y_train, y_test]
print(f'class {cat}: {x_train.shape}, {x_test.shape}, {y_train.shape}, {y_test.shape}\n')

```

Transforming features for class: 0  
Train-test splitting for class: 0  
class 0: (2734, 4), (1824, 4), (2734,), (1824,)

Transforming features for class: 1  
Train-test splitting for class: 1  
class 1: (224341, 4), (149561, 4), (224341,), (149561,)

Transforming features for class: 2  
Train-test splitting for class: 2  
class 2: (83757, 4), (55838, 4), (83757,), (55838,)

Transforming features for class: 3  
Train-test splitting for class: 3  
class 3: (14396, 4), (9598, 4), (14396,), (9598,)

CPU times: total: 2min 25s  
Wall time: 2min 23s

In [140...]

```

# %%time
# # В цикле произведём поиск оптимальных параметров для базовых алгоритмов обучения
# for cat in list_cat:
#     list_rg_models = deepcopy(dict_rg_base_models[cat])
#     list_rg_best_params = [] # список лучших параметров
#     # Выбодим значения train-test из словаря
#     x_train = dict_train_test[cat][0]
#     x_test = dict_train_test[cat][1]
#     y_train = dict_train_test[cat][2]
#     y_test = dict_train_test[cat][3]
#
#     # матрица для мета-алгоритма (для след. пункта)

```

```

#     meta_mtrx_rg = np.empty((x_test.shape[0], len(list_rg_models)))

#     print(f'Ensenble grid-search for class: {cat}')
#     n=0 # счётчик
#     for model, params in zip(list_rg_models, list_rg_params): # цикл по моделям и их словарям параметров
#         print(f"Grid-searching for {model}. Class: {cat}, Model No.{n}")
#         grid = GridSearchCV(
#             estimator=model, param_grid=params,
#             scoring='neg_root_mean_squared_error',
#             n_jobs=6, # 6 из 8 логических процессоров на моем компьютере
#             refit=True,
#             cv=kfrg, verbose=4, # для вывода информации о работе кода
#             pre_dispatch='2*n_jobs', error_score=np.nan, return_train_score=False
#         ) # определяем grid поиск, n_jobs=-1 для мультипроцес.

#         grid.fit(x_train, y_train) # обучаем grid
#         list_rg_best_params.append(grid.best_params_) # дополняем лучшие параметры в список
#         dict_rg_best_params[cat] = list_rg_best_params # записываем список в словарь по ключу соответствующего класса
#         meta_mtrx_rg[:, n] = grid.predict(x_test) # Заполнение мета-матрицы предиктами базовых моделей на валид. данных
#         list_rg_models[n] = deepcopy(model) # Обновляем список базовых алгоритмов обученной моделью

#         print(f'\n{model}: {grid.best_params_}') # выводим название алгоритма и его лучшие параметры
#         print(f'Scoring: {grid.score(x_test, y_test)}\n') # выводим лучшее значение метрики
#         n += 1

# Обновляем словарь базовых алгоритмов вновь полученным списком моделей для данного класса
# dict_rg_base_models[cat] = deepcopy(list_rg_models)
# # записываем список лучших параметров в словарь по ключу соответствующего класса
# dict_rg_best_params[cat] = list_rg_best_params
# # записываем мета-матрицу в словарь по ключу соответствующего класса
# dict_rg_meta_mtrx[cat] = meta_mtrx_rg

```

In [141...]

```

# # Выведем основные результаты grid search
# dict_rg_best_params
# dict_rg_meta_mtrx
# dict_rg_base_models

```

In [142...]

```

dict_best_params = {
    0: [{ 'max_features': 'sqrt'},
        {'hidden_layer_sizes': (500, 400, 300, 200, 100)},
        {'learning_rate': 0.1}],
    1: [{ 'max_features': 'sqrt'},
        {'hidden_layer_sizes': (500, 400, 300, 200, 100)},
        {'learning_rate': 0.1}],
}

```

```
2: [{'max_features': 'sqrt'},
     {'hidden_layer_sizes': (500, 400, 300, 200, 100)},
     {'learning_rate': 0.1}],
3: [{{'max_features': 'sqrt'},
     {'hidden_layer_sizes': (500, 400, 300, 200, 100)},
     {'learning_rate': 0.1}}]
```

```
In [143... dict_metarg
dict_metarg_params
```

```
Out[143]: {}
```

```
Out[143]: {'n_estimators': [50, 100, 200],
            'learning_rate': array([0.1, 0.2, 0.3, 0.4, 0.5]),
            'loss': ['linear', 'square', 'exponential']}
```

```
In [144... # %time
# # В цикле произведём поиск оптимальных параметров для мета-алгоритма
# dict_metarg_best_params = dict() # словарь для хранения лучших параметров мета-алгоритма по классам

# # В цикле произведём поиск оптимальных параметров для базовых алгоритмов обучения
# for cat in list_cat:
#     print(f"Grid-searching for meta-algorythm for category: {cat}")
#     # Выбодим из словарей занчения
#     y_test = dict_train_test[cat][3]
#     meta_mtrx_rg = dict_rg_meta_mtrx[cat]
#     # Обучаем grid
#     meta_grid = GridSearchCV(
#         dict_metarg[cat], dict_metarg_params,
#         scoring='neg_root_mean_squared_error', n_jobs=6, # 6 из 8 логических процессоров на моем компьютере
#         refit=True,
#         cv=kfrg, verbose=3, # для вывода информации о работе кода
#         pre_dispatch='2*n_jobs', error_score=np.nan, return_train_score=False
#     ) # определяем grid поиск, n_jobs=-1 для мултипроцес.) # определяем grid поиск
#     meta_grid.fit(meta_mtrx_rg, y_test) # обучаем grid
#     dict_metarg_best_params[cat] = meta_grid.best_params_ # записываем найденные параметры в словарь по классам
#     print(f'Meta-grid best parameters: {meta_grid.best_params_}')
#     print(f'Meta-grid best score: {meta_grid.best_score_}\n')
```

```
In [145... # # установим лучшие параметры для алгоритмов
# for cat in list_cat:
#     list_rg_models = dict_rg_base_models[cat]
#     list_rg_best_params = dict_rg_best_params[cat]
```

```

#     metarg_best_params = dict_metarg_best_params[cat]

#     n=0 # Счётчик
#     for model, best_params in zip(list_rg_models, list_rg_best_params):
#         model.set_params(**best_params) # Устанавливаем параметры базового алгоритма
#         list_rg_models[n] = deepcopy(model) # Обновляем алгоритм в списке базовых алгоритмов
#         n += 1

#     # Обновляем словарь базовых алгоритмов для данной категории
#     dict_rg_base_models[cat] = deepcopy(list_rg_models)
#     # Устанавливаем параметры мета-алгоритма в словаре по ключам категорий видимости
#     dict_metarg[cat].set_params(**metarg_best_params)

```

In [146...]

```

# %time
# # В цикле создаём и обучаем стэкинг из базовых алгоритмов и метаалгоритма для каждого класса
# dict_rg_stack = dict() # словарь для хранения объектов стэкинга
# for cat in list_cat:
#     # Создаём в словаре объект стэкинга на основе словарей уже обученных алгоритмов с лучшими параметрами
#     dict_rg_stack[cat] = StackingRegressor(
#         estimators=[('rfr', dict_rg_base_models[cat][0]),
#                     ('mlpr', dict_rg_base_models[cat][1]),
#                     ('hgbr', dict_rg_base_models[cat][2])],
#         final_estimator=dict_metarg[cat],
#         cv=kfrg,
#         n_jobs=6, passthrough=False, verbose=1)

#     # Обучаем стэкинг для каждого класса
#     # dict_train_test[cat] = [x_train, x_test, y_train, y_test]
#     dict_rg_stack[cat].fit(dict_train_test[cat][0], dict_train_test[cat][2])

```

## Сохраним словарь обученных моделей стэкинга в файл

In [147...]

```

# # Создадим название директории
# model_path = f'data/models/{PARAMETER71}'
# # Создадим директорию
# makedirs(model_path, exist_ok=True)

# # Запишем словарь стэкинга regressоров в файл joblib
# file_name = f'{model_path}dict_rg_stack.joblib'
# print(f'Saving {file_name} ->', end=' ')
# dummy = joblib.dump(dict_rg_stack, file_name, compress=5) # добавим "пустую" переменную, для избежания лишнего вывода

```

```
# print('DONE!')
```

## Восстановим словарь стекинга по классам категорий видимости

In [151...]

```
# Название директории
model_path = f'data/models/{PARAMETER71}/'

# Восстановим словарь стекинга регрессоров из файлов joblib
file_name = f'{model_path}dict_rg_stack.joblib'
print(f'Loading {file_name} ->', end=' ')
dict_rg_stack = joblib.load(file_name) # загрузим восстановленную модель
print('DONE!')
```

Loading data/models/Visibility/dict\_rg\_stack.joblib -> DONE!

## Предикты и метрики качества регрессии

Этапа обучения

In [152...]

```
%%time

dict_y_train_predict=dict() # словарь предиктов для каждого класса на обучающей выборке

# В цикле получим предикты стекинга на обучающей выборке для каждого класса
for cat in list_cat:
    # dict_train_test[cat] = [x_train, x_test, y_train, y_test]
    dict_y_train_predict[cat] = dict_rg_stack[cat].predict(dict_train_test[cat][0])
```

```
[Parallel(n_jobs=6)]: Using backend ThreadingBackend with 6 concurrent workers.  
[Parallel(n_jobs=6)]: Done 38 tasks | elapsed: 0.0s  
[Parallel(n_jobs=6)]: Done 188 tasks | elapsed: 0.0s  
[Parallel(n_jobs=6)]: Done 300 out of 300 | elapsed: 0.0s finished  
[Parallel(n_jobs=6)]: Using backend ThreadingBackend with 6 concurrent workers.  
[Parallel(n_jobs=6)]: Done 38 tasks | elapsed: 1.4s  
[Parallel(n_jobs=6)]: Done 188 tasks | elapsed: 6.2s  
[Parallel(n_jobs=6)]: Done 300 out of 300 | elapsed: 9.5s finished  
[Parallel(n_jobs=6)]: Using backend ThreadingBackend with 6 concurrent workers.  
[Parallel(n_jobs=6)]: Done 38 tasks | elapsed: 0.2s  
[Parallel(n_jobs=6)]: Done 188 tasks | elapsed: 1.0s  
[Parallel(n_jobs=6)]: Done 300 out of 300 | elapsed: 1.6s finished  
[Parallel(n_jobs=6)]: Using backend ThreadingBackend with 6 concurrent workers.  
[Parallel(n_jobs=6)]: Done 38 tasks | elapsed: 0.0s  
[Parallel(n_jobs=6)]: Done 188 tasks | elapsed: 0.0s  
[Parallel(n_jobs=6)]: Done 300 out of 300 | elapsed: 0.1s finished  
CPU times: total: 3min 25s  
Wall time: 1min 1s
```

In [153...]

```
# Выведем метрики качества для обучающей выборки  
# dict_train_test[cat] = [x_train, x_test, y_train, y_test]  
print("ЭТАП ОБУЧЕНИЯ:")  
for cat in list_cat:  
    print(f"PARAMETER{cat}, class {cat}:")  
    r2 = r2_score(dict_train_test[cat][2], dict_y_train_predict[cat],  
                  sample_weight=None, multioutput='uniform_average', force_finite=True)  
    rmse = mean_squared_error(dict_train_test[cat][2], dict_y_train_predict[cat],  
                             sample_weight=None, multioutput='uniform_average', squared=False)  
    mae = mean_absolute_error(dict_train_test[cat][2], dict_y_train_predict[cat],  
                            sample_weight=None, multioutput='uniform_average')  
    max_e = max_error(dict_train_test[cat][2], dict_y_train_predict[cat])  
  
    print(f"R2: {r2}")  
    print(f"RMSE: {rmse}")  
    print(f"MAE: {mae}")  
    print(f"MAXIMUM ERROR: {max_e}\n")
```

ЭТАП ОБУЧЕНИЯ:  
Visibility, class 0:  
R2: -0.1333827046634839  
RMSE: 0.17814397752563293  
MAE: 0.10822940309809649  
MAXIMUM ERROR: 0.7

Visibility, class 1:  
R2: 0.22904366168392853  
RMSE: 2.4581781956202233  
MAE: 1.236934859616637  
MAXIMUM ERROR: 15.0

Visibility, class 2:  
R2: 0.6550343817662962  
RMSE: 4.3303560579457026  
MAE: 2.3583362956293765  
MAXIMUM ERROR: 25.0

Visibility, class 3:  
R2: 0.9790686817916153  
RMSE: 0.5229519519951925  
MAE: 0.015073631564323424  
MAXIMUM ERROR: 21.0

## Этап валидации

In [154...]

```
%time

dict_y_test_predict=dict() # словарь предиктов для каждого класса на обучающей выборке

# В цикле получим предикты стэкинга на тестовой выборке для каждого класса
for cat in list_cat:
    # dict_train_test[cat] = [x_train, x_test, y_train, y_test]
    dict_y_test_predict[cat] = dict_rg_stack[cat].predict(dict_train_test[cat][1])
```

```
[Parallel(n_jobs=6)]: Using backend ThreadingBackend with 6 concurrent workers.
[Parallel(n_jobs=6)]: Done  38 tasks    | elapsed:   0.0s
[Parallel(n_jobs=6)]: Done 188 tasks    | elapsed:   0.0s
[Parallel(n_jobs=6)]: Done 300 out of 300 | elapsed:   0.0s finished
[Parallel(n_jobs=6)]: Using backend ThreadingBackend with 6 concurrent workers.
[Parallel(n_jobs=6)]: Done  38 tasks    | elapsed:   0.7s
[Parallel(n_jobs=6)]: Done 188 tasks    | elapsed:   4.0s
[Parallel(n_jobs=6)]: Done 300 out of 300 | elapsed:   7.0s finished
[Parallel(n_jobs=6)]: Using backend ThreadingBackend with 6 concurrent workers.
[Parallel(n_jobs=6)]: Done  38 tasks    | elapsed:   0.0s
[Parallel(n_jobs=6)]: Done 188 tasks    | elapsed:   0.5s
[Parallel(n_jobs=6)]: Done 300 out of 300 | elapsed:   0.9s finished
[Parallel(n_jobs=6)]: Using backend ThreadingBackend with 6 concurrent workers.
[Parallel(n_jobs=6)]: Done  38 tasks    | elapsed:   0.0s
[Parallel(n_jobs=6)]: Done 188 tasks    | elapsed:   0.0s
[Parallel(n_jobs=6)]: Done 300 out of 300 | elapsed:   0.0s finished
CPU times: total: 2min 18s
Wall time: 42.2 s
```

In [155...]

```
# Выведем метрики качества для тестовой выборки
# dict_train_test[cat] = [x_train, x_test, y_train, y_test]
print("ЭТАП ВАЛИДАЦИИ:")
for cat in list_cat:
    print(f"PARAMETER{cat}, class {cat}:")
    r2 = r2_score(dict_train_test[cat][3], dict_y_test_predict[cat],
                  sample_weight=None, multioutput='uniform_average', force_finite=True)
    rmse = mean_squared_error(dict_train_test[cat][3], dict_y_test_predict[cat],
                               sample_weight=None, multioutput='uniform_average', squared=False)
    mae = mean_absolute_error(dict_train_test[cat][3], dict_y_test_predict[cat],
                               sample_weight=None, multioutput='uniform_average')
    max_e = max_error(dict_train_test[cat][3], dict_y_test_predict[cat])

    print(f"R2: {r2}")
    print(f"RMSE: {rmse}")
    print(f"MAE: {mae}")
    print(f"MAXIMUM ERROR: {max_e}\n")
```

ЭТАП ВАЛИДАЦИИ:

Visibility, class 0:  
R2: -0.15230765186057016  
RMSE: 0.17805374379693484  
MAE: 0.10910889629916429  
MAXIMUM ERROR: 0.7

Visibility, class 1:  
R2: 0.16504953864126892  
RMSE: 2.562891500723672  
MAE: 1.3039467579966186  
MAXIMUM ERROR: 15.0

Visibility, class 2:  
R2: 0.6149169410411757  
RMSE: 4.587615939863994  
MAE: 2.5255228918131043  
MAXIMUM ERROR: 25.0

Visibility, class 3:  
R2: 0.9811590875358314  
RMSE: 0.4853554474794731  
MAE: 0.013648676807668264  
MAXIMUM ERROR: 21.0

7.1.4. Применение выбранных моделей для исправления, восстановления данных и получения предиктов показателя метеорологической дальности видимости *Visibility* для Агробиостанции МГУ в пос. Чашниково и для центральной точки поля метеостанций; фиксация исправлений в архивах

7.1.4.1. Предсказания стекинга регрессоров для рабочих данных, разбитых по классам категорий дальности видимости (там, где дальность видимости неопределена).\*\*

```
In [156]: data71[(pd.isna(data71.Visibility))].sample(5, random_state=56)
```

Out[156]:

	Height	Distance	Bearing	Month	Hour	T	T_drift	P_station	P_drift	Humid	Dew_point	Closest_val	Closest_dist	Clos
<b>23975</b>	215.0	71.720525	108.834256	3.0	0.0	0.710429	-3.170200	754.188398	0.370388	48.897517	-8.800919	10.0	26.627460	
<b>151606</b>	138.0	152.988800	36.271056	4.0	21.0	2.397962	-4.540011	756.035658	-0.504310	54.865836	-5.746090	15.0	109.414691	
<b>322278</b>	175.0	99.058072	126.256571	4.0	9.0	3.011428	2.175797	740.287471	-0.192936	67.190430	-2.473477	10.0	39.077968	
<b>42177</b>	215.0	71.720525	108.834256	1.0	18.0	-6.703833	0.566242	765.722742	1.481808	81.116060	-9.392701	10.0	26.627460	
<b>26125</b>	215.0	71.720525	108.834256	7.0	6.0	16.621061	-1.151526	738.108530	0.306384	78.268072	12.817978	10.0	26.627460	

In [157...]

```
# Определим фичи для каждого класса категории видимости
# Поскольку данные _drift в 1 и 2 рядах от начала архива уже не содержат NaN, не будем удалять NaN
X_wrk = data71[(pd.isna(data71.Visibility))].iloc[:, :-3]
X_wrk.shape
```

Out[157]:

```
(167793, 14)
```

In [158...]

```
# Определим фичи для каждого класса категории видимости
# Поскольку данные _drift в 1 и 2 рядах от начала архива уже не содержат NaN, не будем удалять NaN
dict_X_wrk = dict()

# В цикле произведём разбиения рабочего датасета на части, сообразно предсказанным классам
for cat in list_cat:
    dict_X_wrk[cat] = data71[(pd.isna(data71.Visibility)) & (data71.Cat == cat)].iloc[:, :-3]
    dict_X_wrk[cat].shape
```

Out[158]:

```
(13847, 14)
(88457, 14)
(60114, 14)
(5375, 14)
```

In [159...]

```
# В цикле сохраним индекс для прогнозируемых значений.
dict_idx_wrk = dict()
for cat in list_cat:
    dict_idx_wrk[cat] = dict_X_wrk[cat].index
```

In [160...]

```
# В цикле применим обученную трансформацию данных
dict_X_svd_wrk = dict()
for cat in list_cat:
    X_trans_w = q_transformer.fit_transform(dict_X_wrk[cat])
    # Выберем те же фичи, что и на этапах обучения/валидации
    cols = list(gu_selector.get_support(indices=True)) # Список горизонтальных индексов для выборки фичей
    X_sel_w = X_trans_w[:, cols]
    # Выполним SVD преобразование
    X_svd_w = tsvd.fit_transform(X_sel_w)
    dict_X_svd_wrk[cat] = deepcopy(X_svd_w)
    dict_X_svd_wrk[cat].shape
```

Out[160]: (13847, 4)

Out[160]: (88457, 4)

Out[160]: (60114, 4)

Out[160]: (5375, 4)

In [161...]

```
# В цикле выполним предсказание значений дальности видимости для рабочей части датасета
dict_y_predict_wrk = dict()
for cat in list_cat:
    dict_y_predict_wrk[cat] = dict_rg_stack[cat].predict(dict_X_svd_wrk[cat])
    dict_y_predict_wrk[cat]
```

```
[Parallel(n_jobs=6)]: Using backend ThreadingBackend with 6 concurrent workers.
[Parallel(n_jobs=6)]: Done  38 tasks      | elapsed:   0.0s
[Parallel(n_jobs=6)]: Done 188 tasks      | elapsed:   0.0s
[Parallel(n_jobs=6)]: Done 300 out of 300 | elapsed:   0.0s finished
```

Out[161]: array([0.5, 0.5, 0.4, ..., 0.5, 0.5, 0.5])

```
[Parallel(n_jobs=6)]: Using backend ThreadingBackend with 6 concurrent workers.
[Parallel(n_jobs=6)]: Done  38 tasks      | elapsed:   0.0s
[Parallel(n_jobs=6)]: Done 188 tasks      | elapsed:   0.4s
[Parallel(n_jobs=6)]: Done 300 out of 300 | elapsed:   0.7s finished
```

Out[161]: array([10., 10., 10., ..., 10., 5., 10.])

```
[Parallel(n_jobs=6)]: Using backend ThreadingBackend with 6 concurrent workers.
[Parallel(n_jobs=6)]: Done  38 tasks      | elapsed:   0.0s
[Parallel(n_jobs=6)]: Done 188 tasks      | elapsed:   0.2s
[Parallel(n_jobs=6)]: Done 300 out of 300 | elapsed:   0.4s finished
```

```
Out[161]: array([32.66666667, 30.          , 40.          , ... , 25.          ,
   29.          , 20.          ])
[Parallel(n_jobs=6)]: Using backend ThreadingBackend with 6 concurrent workers.
[Parallel(n_jobs=6)]: Done  38 tasks      | elapsed:    0.0s
[Parallel(n_jobs=6)]: Done 188 tasks      | elapsed:    0.0s
[Parallel(n_jobs=6)]: Done 300 out of 300 | elapsed:    0.0s finished
Out[161]: array([50., 50., 50., ..., 50., 60., 50.])
```

```
In [162...]: # В цикле создадим серию данных из предсказаний и обновим датасет
for cat in list_cat:
    ser_y_predict = pd.Series(dict_y_predict_wrk[cat], index=dict_idx_wrk[cat], name="Visibility")
    data71.update(ser_y_predict, overwrite=False)
```

```
In [163...]: data71[pd.isna(data71.Visibility)]
```

```
Out[163]: Height Distance Bearing Month Hour T T_drift P_station P_drift Humid Dew_point Closest_val Closest_dist Closest_bearing Visibility Cat I
```

Строк без указания дальности видимости больше не осталось.

Установим категории и баллы видимости для всех значений Visibility

```
In [164...]: data71.loc[data71.Visibility>=50, "Cat"] = 3 # Excellent
data71.loc[data71.Visibility>=50, "Rank"] = 9

data71.loc[(data71.Visibility>=20) & (data71.Visibility<50), "Cat"] = 2 # Good
data71.loc[(data71.Visibility>=20) & (data71.Visibility<50), "Rank"] = 8

data71.loc[(data71.Visibility>=10) & (data71.Visibility<20), "Cat"] = 1 # Mist
data71.loc[(data71.Visibility>=10) & (data71.Visibility<20), "Rank"] = 7

data71.loc[(data71.Visibility>=4) & (data71.Visibility<10), "Cat"] = 1 # Mist
data71.loc[(data71.Visibility>=4) & (data71.Visibility<10), "Rank"] = 6

data71.loc[(data71.Visibility>=2) & (data71.Visibility<4), "Cat"] = 1 # Mist
data71.loc[(data71.Visibility>=2) & (data71.Visibility<4), "Rank"] = 5

data71.loc[(data71.Visibility>=1) & (data71.Visibility<2), "Cat"] = 1 # Mist
data71.loc[(data71.Visibility>=1) & (data71.Visibility<2), "Rank"] = 4
```

```

data71.loc[(data71.Visibility>=0.5) & (data71.Visibility<1), "Cat"] = 0 # Fog
data71.loc[(data71.Visibility>=0.5) & (data71.Visibility<1), "Rank"] = 3

data71.loc[(data71.Visibility>=0.2) & (data71.Visibility<0.5), "Cat"] = 0 # Fog
data71.loc[(data71.Visibility>=0.2) & (data71.Visibility<0.5), "Rank"] = 2

data71.loc[(data71.Visibility>=0.05) & (data71.Visibility<0.2), "Cat"] = 0 # Fog
data71.loc[(data71.Visibility>=0.05) & (data71.Visibility<0.2), "Rank"] = 1

data71.loc[(data71.Visibility<0.05), "Cat"] = 0 # Fog
data71.loc[(data71.Visibility<0.05), "Rank"] = 0

data71.sample(10)

```

Out[164]:	Height	Distance	Bearing	Month	Hour	T	T_drift	P_station	P_drift	Humid	Dew_point	Closest_val	Closest_dist	Clc
<b>352433</b>	175.0	99.058072	126.256571	12.0	0.0	-0.431766	-0.764611	742.428320	-0.137165	98.131873	-0.690110	0.5	39.077968	
<b>254336</b>	193.0	104.007002	157.684533	2.0	6.0	-0.300000	-0.900000	754.200000	0.900000	70.000000	-5.200000	20.0	46.580297	
<b>42103</b>	215.0	71.720525	108.834256	1.0	0.0	-3.787058	0.242632	744.487426	-0.945495	94.861372	-4.488116	10.0	26.627460	
<b>313161</b>	175.0	99.058072	126.256571	5.0	0.0	10.600000	-4.100000	747.100000	0.300000	76.000000	6.500000	10.0	39.077968	
<b>2807</b>	215.0	71.720525	108.834256	6.0	0.0	21.185624	-4.804851	746.472397	0.210181	86.175835	18.779049	10.0	26.627460	
<b>506725</b>	156.0	170.601489	149.816257	3.0	3.0	-11.600000	-1.200000	735.600000	-0.300000	74.000000	-15.300000	25.0	69.057060	
<b>325042</b>	175.0	99.058072	126.256571	4.0	21.0	9.100000	-1.400000	736.900000	0.300000	34.000000	-5.900000	10.0	39.077968	
<b>184585</b>	167.0	40.677898	76.562251	4.0	9.0	9.000000	4.200000	749.000000	0.400000	56.000000	0.600000	10.0	48.595243	
<b>517111</b>	185.0	75.614832	292.050594	12.0	18.0	-8.500000	0.500000	741.200000	0.400000	93.000000	-9.500000	10.0	72.307124	
<b>251752</b>	185.0	81.696291	183.314261	9.0	9.0	17.700000	3.900000	746.000000	0.000000	80.000000	14.200000	23.0	46.580297	

◀ ▶

#### 7.1.4.2. Перенесение полученных предиктов в архивы

```

In [165...]: # Добавим в df_tmp71 столбцы для будущих значений для Чашниково и центральной точки, заполним их NaN
# - это необходимо, чтобы вычислить значения для архивов
# Создадим столбцы col_name со значениями pr.nan
# Переименуем столбец PARAMETER71#Chashnikovo и PARAMETER71#Rfrnce_point
df_tmp71 = (df_tmp71.

```

```

        assign(col_name1 = np.nan,
               col_name2 = np.nan).
        rename(columns={"col_name1": PARAMETER71+'#'+ 'Chashnikovo',
                       "col_name2": PARAMETER71+'#'+ 'Rfrnce_point'})
    )
df_tmp71.sample(3, random_state=56)

```

Out[165]:

	Visibility#V_Volochek	Visibility#Staritsa	Visibility#Kashyn	Visibility#Tver	Visibility#Klin	Visibility#Dmitrov	Visibility#Volokolamsk	Visibility#M
<b>2014-02-22 03:00:00</b>	10.0	20.0	50.0	10.0	10.0	10.0	10.0	10.0
<b>2015-05-16 03:00:00</b>	10.0	10.0	10.0	10.0	10.0	10.0	10.0	10.0
<b>2020-06-18 18:00:00</b>	20.0	NaN	50.0	10.0	10.0	10.0	10.0	10.0

In [166...]

```

# Добавим в архив параметров Visibility столбец для Чашниково и центральной точки, заполним их NaN
# Создадим столбцы col_name со значениями np.nan
# Переименуем столбец PARAMETER71#Chashnikovo и PARAMETER71#Rfrnce_point
dict_df_parameters['df_'+PARAMETER71] = (dict_df_parameters['df_'+PARAMETER71].
                                         assign(col_name1 = np.nan,
                                                col_name2 = np.nan).
                                         rename(columns={"col_name1": PARAMETER71+'#'+ 'Chashnikovo',
                                                        "col_name2": PARAMETER71+'#'+ 'Rfrnce_point"})
                                         )
dict_df_parameters['df_'+PARAMETER71].sample(3, random_state=56)

```

Out[166]:

	Visibility#V_Volochek	Visibility#Staritsa	Visibility#Kashyn	Visibility#Tver	Visibility#Klin	Visibility#Dmitrov	Visibility#Volokolamsk	Visibility#Moscow
<b>2014-02-22 03:00:00</b>	10.0	20.0	50.0	10.0	10.0	10.0	10.0	10.0
<b>2015-05-16 03:00:00</b>	10.0	10.0	10.0	10.0	10.0	10.0	10.0	10.0
<b>2020-06-18 18:00:00</b>	20.0	NaN	50.0	10.0	10.0	10.0	10.0	10.0

In [167...]

```
# В архивах метеостанций df Чашниково и df для центральной точки
# создадим столбцы с назначением PARAMETER71

dict_df_locations['df_Chashnikovo'] = (dict_df_locations['df_Chashnikovo'].
                                         assign(col_name = np.nan).
                                         rename(columns={"col_name": PARAMETER71})
                                         )
dict_df_locations['df_Rfrnce_point'] = (dict_df_locations['df_Rfrnce_point'].
                                         assign(col_name = np.nan).
                                         rename(columns={"col_name": PARAMETER71}
                                         )
                                         )

dict_df_locations['df_Chashnikovo'].sample(3, random_state=56)
dict_df_locations['df_Rfrnce_point'].sample(3, random_state=56)
```

Out[167]:

	T	T_min	T_max	P_sea	P_station	P_drift	Humid	Dew_point	Soil_T	Snow_height	Visibility
<b>2014-02-22 03:00:00</b>	-4.156558	-4.156558	-2.109643	768.336709	747.597791	0.812775	76.635021	-7.639754	NaN	NaN	NaN
<b>2015-05-16 03:00:00</b>	9.181730	9.181730	10.434571	745.095535	725.921747	-1.043037	95.837940	8.552121	NaN	NaN	NaN
<b>2020-06-18 18:00:00</b>	27.439052	25.366130	30.241632	761.480701	743.060948	-0.492627	58.118042	18.459938	NaN	NaN	NaN

Out[167]:

	T	T_min	T_max	P_sea	P_station	P_drift	Humid	Dew_point	Soil_T	Snow_height	Visibility
2014-02-22 03:00:00	-3.589183	-3.794602	-2.700734	767.373725	754.041734	0.616282	75.006151	-7.367199	NaN	NaN	NaN
2015-05-16 03:00:00	7.789650	7.789650	8.797603	746.708428	734.256500	-0.826706	94.058645	6.893682	NaN	NaN	NaN
2020-06-18 18:00:00	29.404954	25.572651	29.941094	760.796222	749.008692	-0.723724	41.950321	15.114705	NaN	NaN	NaN

### Перенесение уже исправленных сплошных NaN в архивы

Исправленные сплошные NaN уже перенесены в архивы выше, перед началом классификации дальности видимости по категориям

In [168...]

```
# Подсчитаем количество оставшихся строк со сплошными NaN
all_nans_count = sum(
    dict_df_parameters['df_'+PARAMETER71]
    .apply(lambda x: sum(x.isna()), axis=1)==len(dict_df_parameters['df_'+PARAMETER71].keys()))
)
print(f'Количество строк со сплошными NaN равно {all_nans_count}'')
```

Количество строк со сплошными NaN равно 1

In [169...]

```
dict_df_parameters['df_'+PARAMETER71].tail(3)
```

Out[169]:

	Visibility#V_Volochek	Visibility#Staritsa	Visibility#Kashyn	Visibility#Tver	Visibility#Klin	Visibility#Dmitrov	Visibility#Volokolamsk	Visibility#Mo
2005-02-01 06:00:00	NaN	4.0	NaN	10.0	10.0	9.0		20.0
2005-02-01 03:00:00	NaN	4.0	NaN	10.0	10.0	9.0		20.0
2005-02-01 00:00:00	NaN	NaN	NaN	NaN	NaN	NaN		NaN

Всё верно - это начальная строка архивов

In [170...]

```
# # Перенесём уже исправленные значения в dict_df_parameters, оставшиеся NaN исправим ниже
# dict_df_parameters['df_'+PARAMETER71] = df_tmp71.copy(deep=True)

# # Перенесём уже исправленные значения в dict_df_locations, оставшиеся NaN исправим ниже
# for name_df in dict_df_locations.keys():
#     dict_df_locations[name_df].loc[:, PARAMETER71] = df_tmp71.loc[:, PARAMETER71 + '#' + name_df[3:]]
```

### 7.1.4.3. Заполнение NaN в архивах результатами предсказания регрессоров сообразно разбиению на классы

**Обновим столбец Visibility в df\_total71 за счёт предиктов, полученных в data71**

In [171...]

```
df_total71.update(data71.Visibility, overwrite=False)
```

In [172...]

```
df_total71.head()
```

Out[172]:

	Station	Height	Distance	Bearing	Observation	Month	Hour	T	T_min	T_max	T_drift	P_station	P_drift	Humid
0	Chashnikovo	215	71.720525	108.834256	2022-06-09 21:00:00	6	21	18.330035	18.330035	27.774218	-5.249743	739.949808	0.001137	81.786605
1	Chashnikovo	215	71.720525	108.834256	2022-06-09 18:00:00	6	18	23.579777	22.158215	27.361772	-3.781994	739.948671	0.035801	53.876986
2	Chashnikovo	215	71.720525	108.834256	2022-06-09 15:00:00	6	15	27.361772	11.047725	27.361772	1.470969	739.912870	-0.731062	35.710055
3	Chashnikovo	215	71.720525	108.834256	2022-06-09 12:00:00	6	12	25.890803	9.040396	25.890803	3.732588	740.643932	-0.445373	36.287145
4	Chashnikovo	215	71.720525	108.834256	2022-06-09 09:00:00	6	9	22.158215	6.288497	22.158215	11.110490	741.089305	0.024346	47.179575

In [173...]

```
# В цикле по названиям столбцов
for col in df_tmp71.keys().tolist():
    station = col[len(PARAMETER71)+1:] # Вычленяем название станции
    # формируем серию для обновления значений данного столбца из данных df_total71
    ser_updater = pd.Series(np.array(df_total71[df_total71.Station == station].Visibility),
                           index = df_total71[df_total71.Station == station].Observation,
```

### Перенесём значения предиктов в df\_tmp71

```
        name = col)
df_tmp71.update(ser_updater, overwrite=False) # обновляем исходный DF
```

In [174...]  
# Проверим количество оставшихся NaN  
pd.isna(df\_tmp71).sum().sum()

Out[174]: 14

Всё верно - это начальная строка архива

## Обновим архив параметров

In [175...]  
# Перенесём полученные значения df\_tmp71 в архив параметров  
dict\_df\_parameters['df\_'+PARAMETER71].update(df\_tmp71, overwrite=False)

## Обновим архив метеостанций

In [176...]  
**for** col **in** df\_tmp71.keys().tolist():
 station = col[len(PARAMETER71)+1:] # Вычленяем название станции
 # формируем серию для обновления значений данного столбца из данных df\_total71
 ser\_updater = pd.Series(df\_tmp71[col],
 name = PARAMETER71)
 dict\_df\_locations["df\_"+station].update(ser\_updater, overwrite=False) # обновляем DF архива

## 7.1.5. Визуализация графиков дальности видимости Visibility для условной метеостанции Чашниково

In [177...]  
data1 = dict\_df\_parameters['df\_'+PARAMETER71][PARAMETER71+"#"+"Chashnikovo"]
# Строим график в цикле для каждого года
**for** year **in** data1.index.year.unique(): # по перечню уникальных лет в data1, по годам
 fig, ax = plt.subplots(figsize=(10, 4))
 g1 = sns.lineplot(data=data1[data1.index.year==year],
 color='steelblue',
 linewidth=0.75,
 ax=ax)

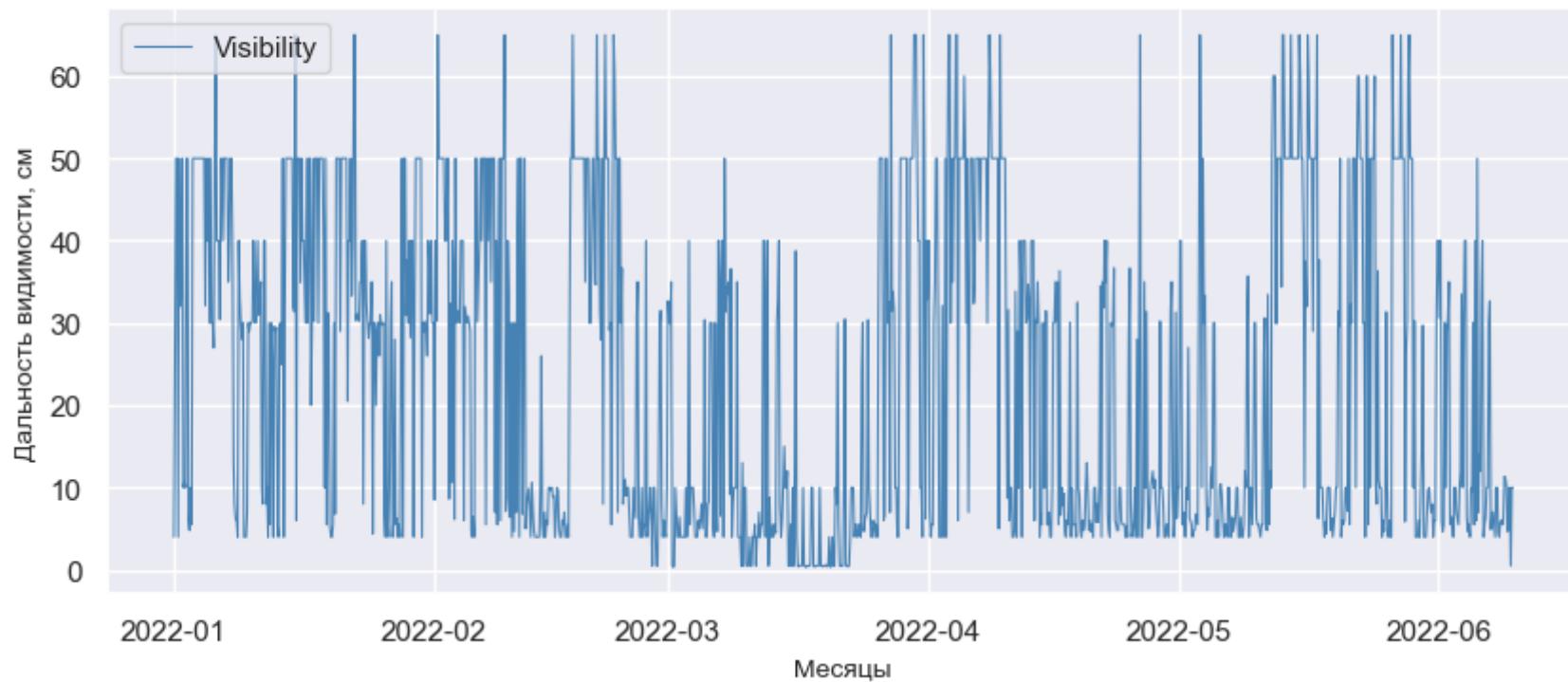
 # Добавляем легенду

```
blue_line = mlines.Line2D([], [], color='steelblue', label=f'{PARAMETER71}', linewidth=0.75)

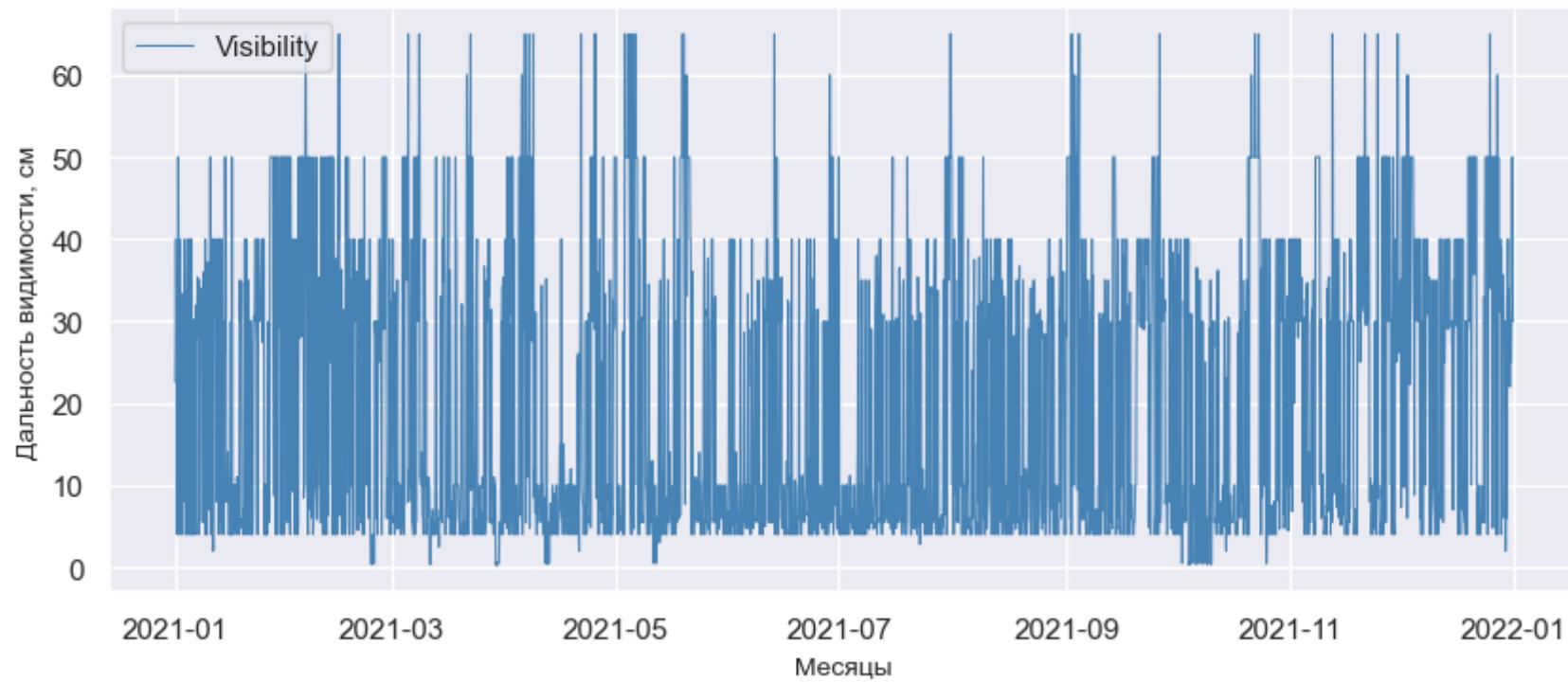
dummy = ax.legend(handles=[blue_line])

dummy = ax.set_ylabel('Дальность видимости, см', size=10)
dummy = ax.set_xlabel('Месяцы', size=10)
dummy = plt.title(f'Чашниково: Ежедневная динамика параметра \n'
                  f'дальности видимости {PARAMETER71}, {year} год\n')
plt.show()
```

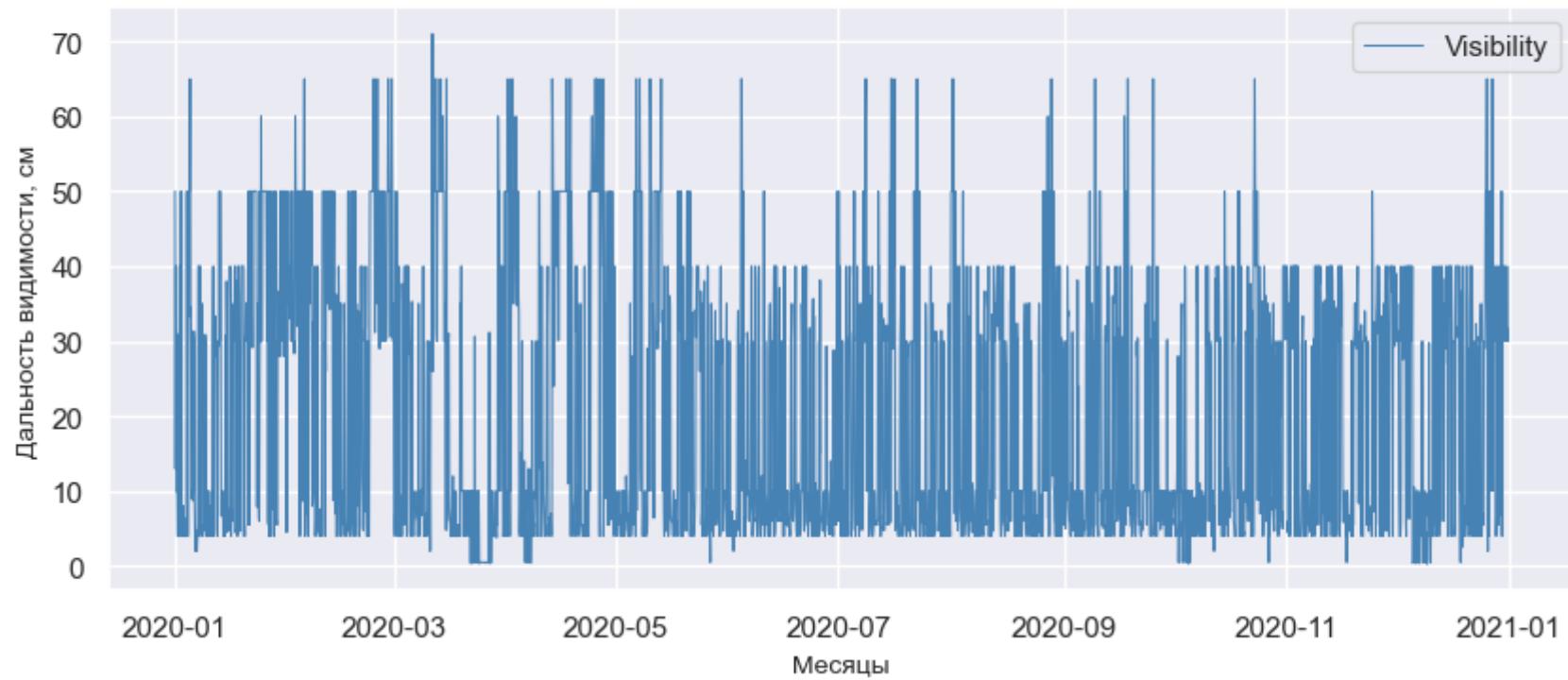
Чашниково: Ежедневная динамика параметра  
 дальности видимости Visibility, 2022 год



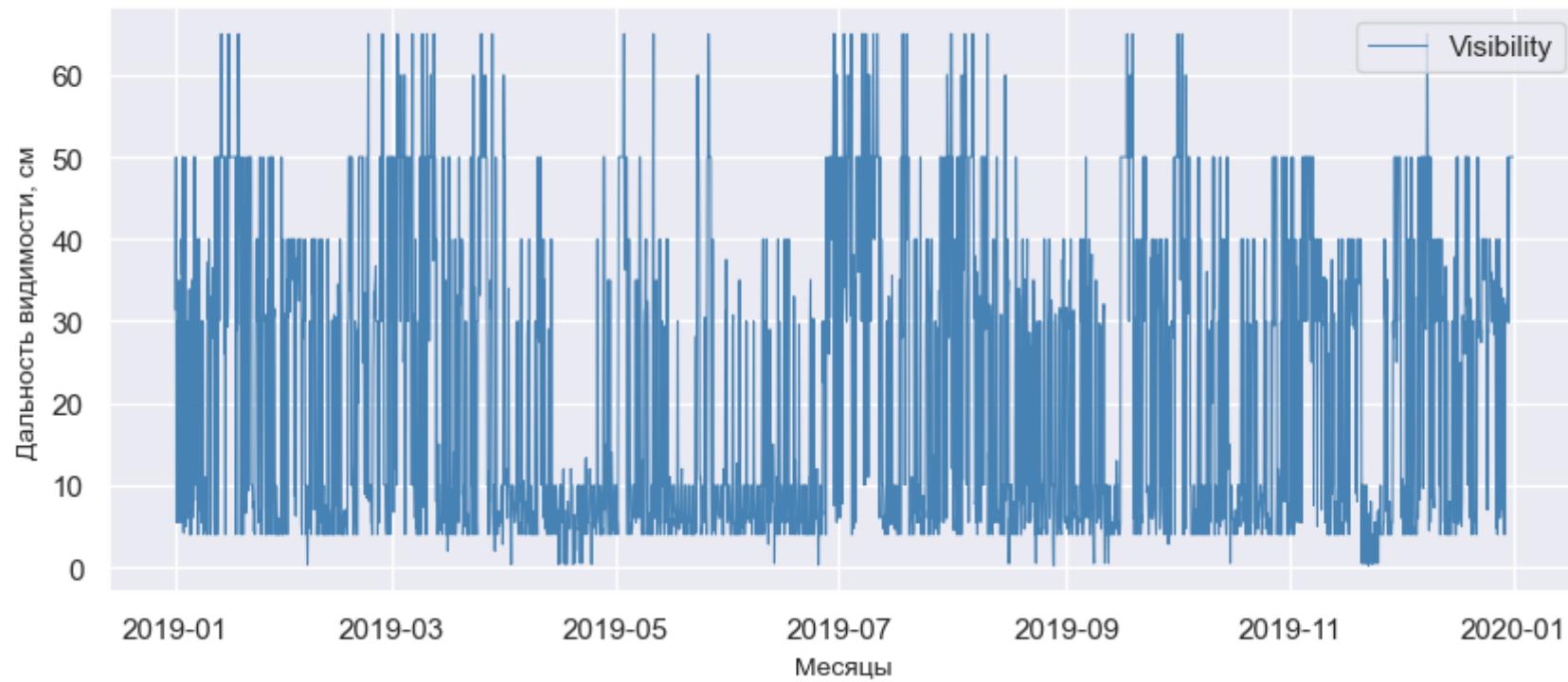
Чашниково: Ежедневная динамика параметра  
 дальности видимости Visibility, 2021 год



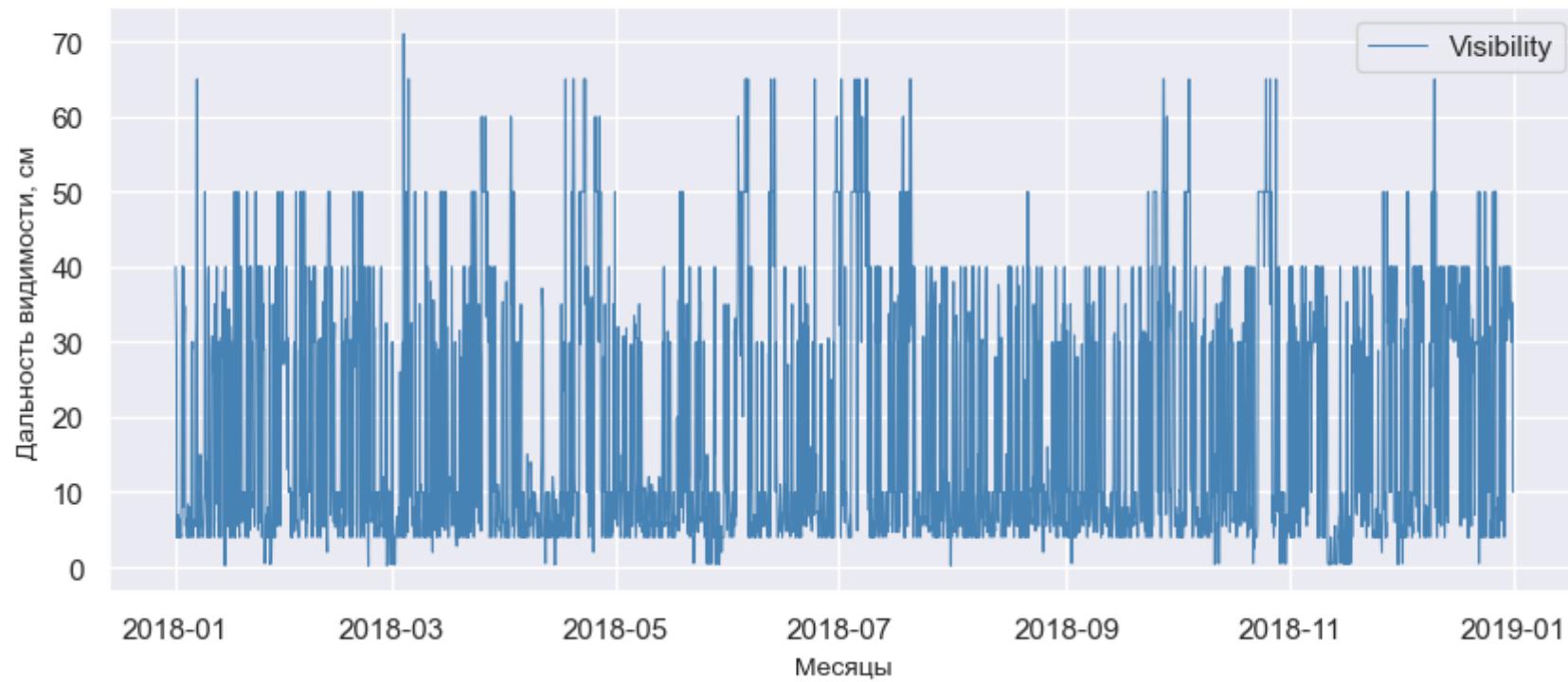
Чашниково: Ежедневная динамика параметра  
 дальности видимости Visibility, 2020 год



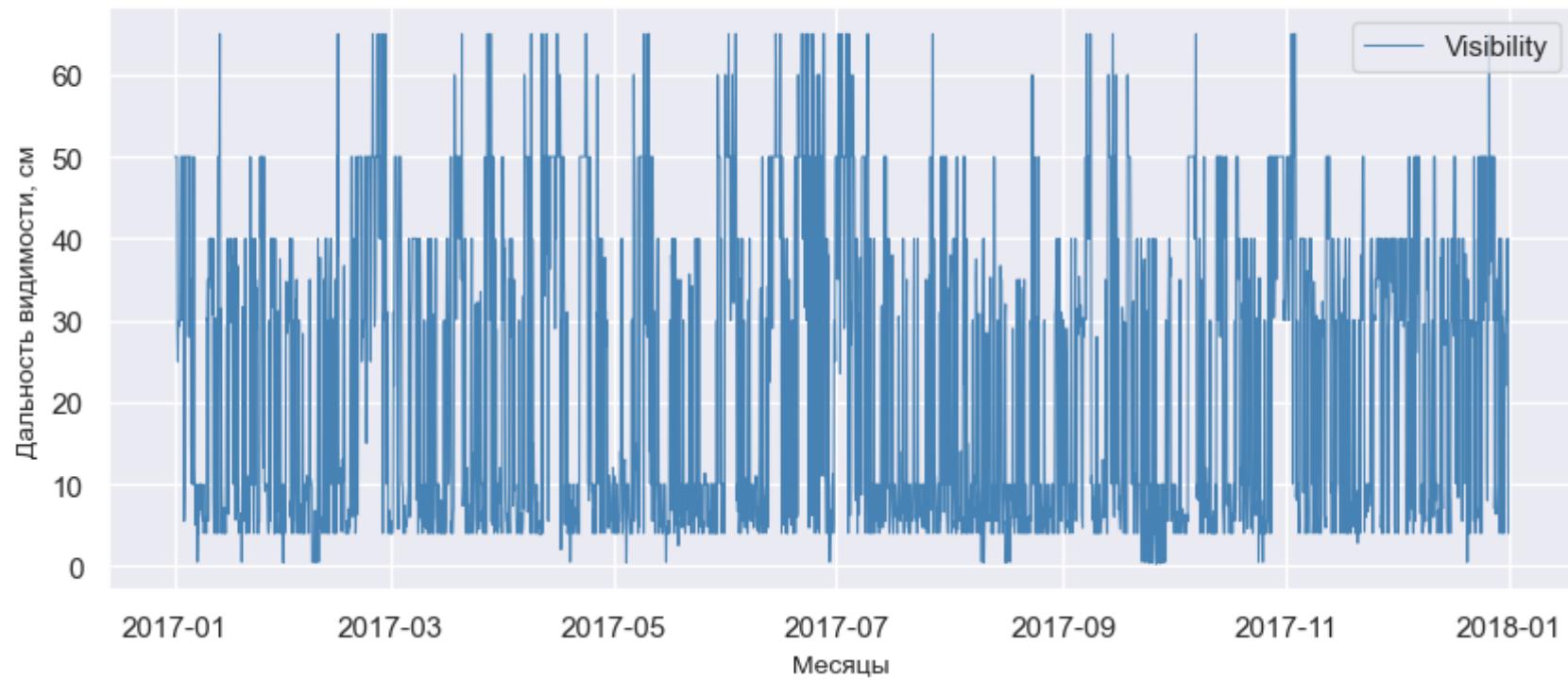
Чашниково: Ежедневная динамика параметра  
 дальности видимости Visibility, 2019 год



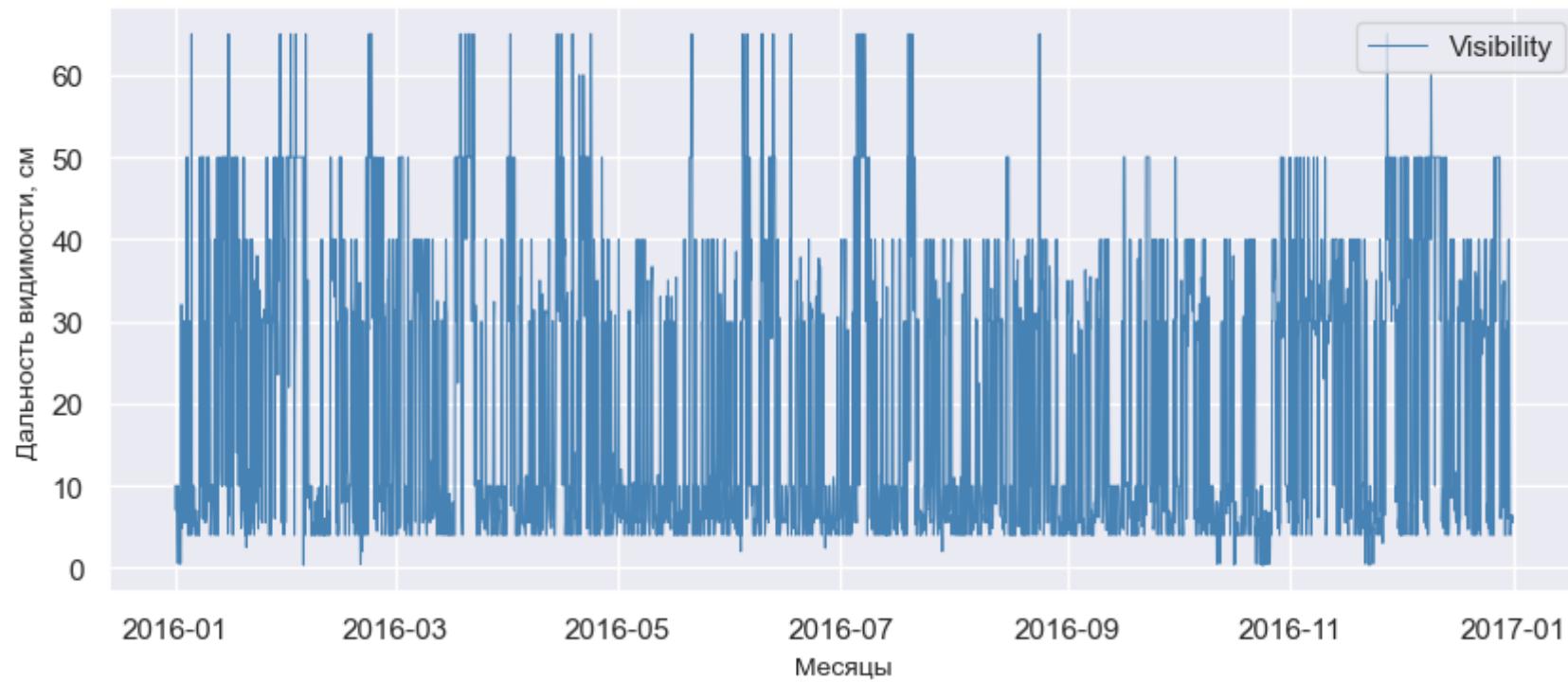
Чашниково: Ежедневная динамика параметра  
 дальности видимости Visibility, 2018 год



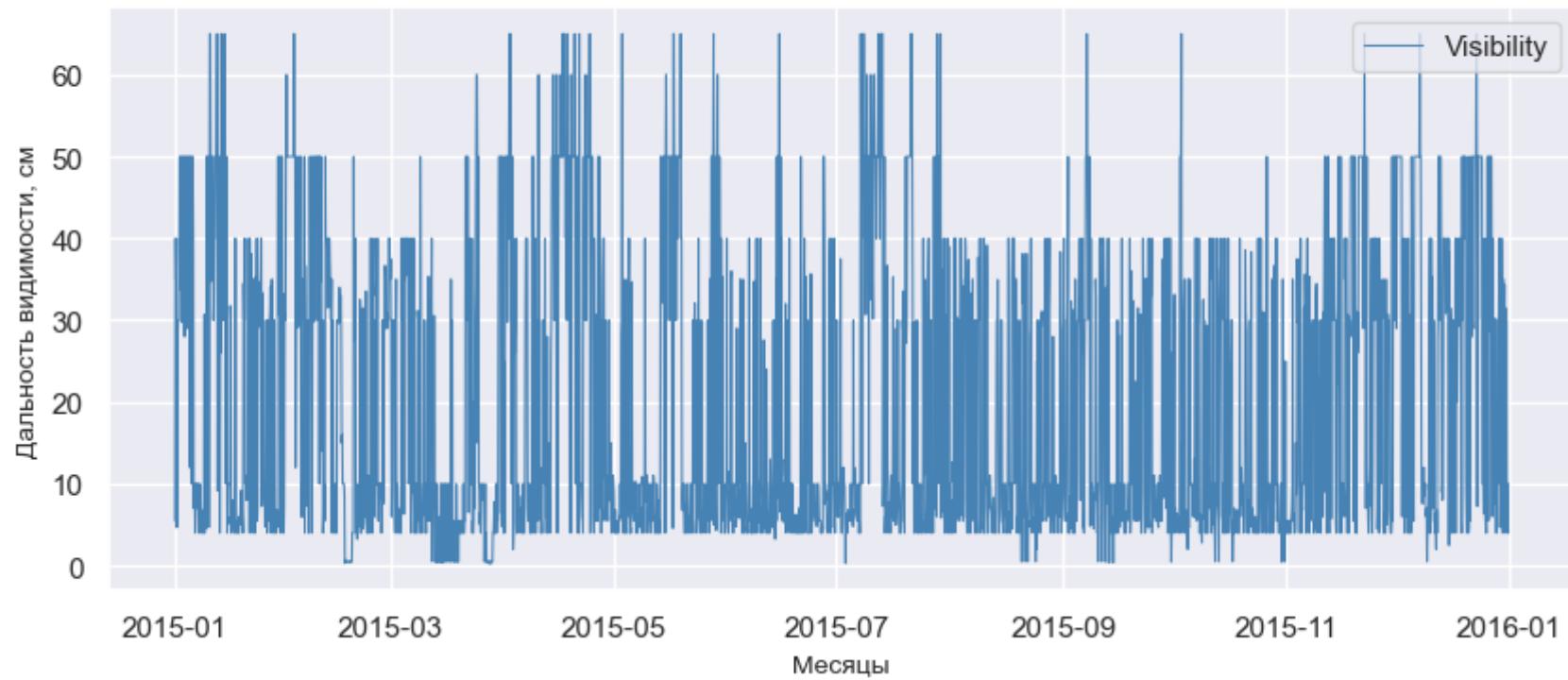
Чашниково: Ежедневная динамика параметра  
 дальности видимости Visibility, 2017 год



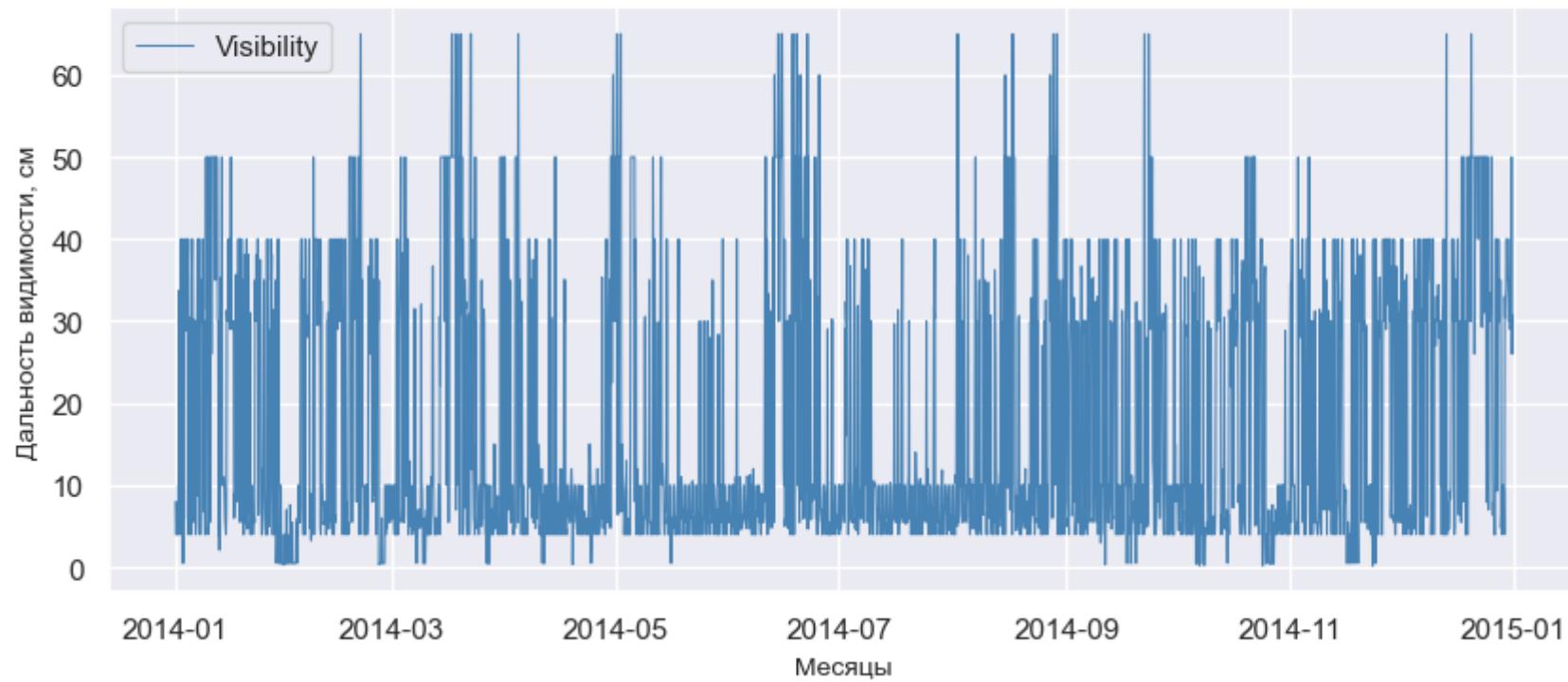
Чашниково: Ежедневная динамика параметра  
 дальности видимости Visibility, 2016 год



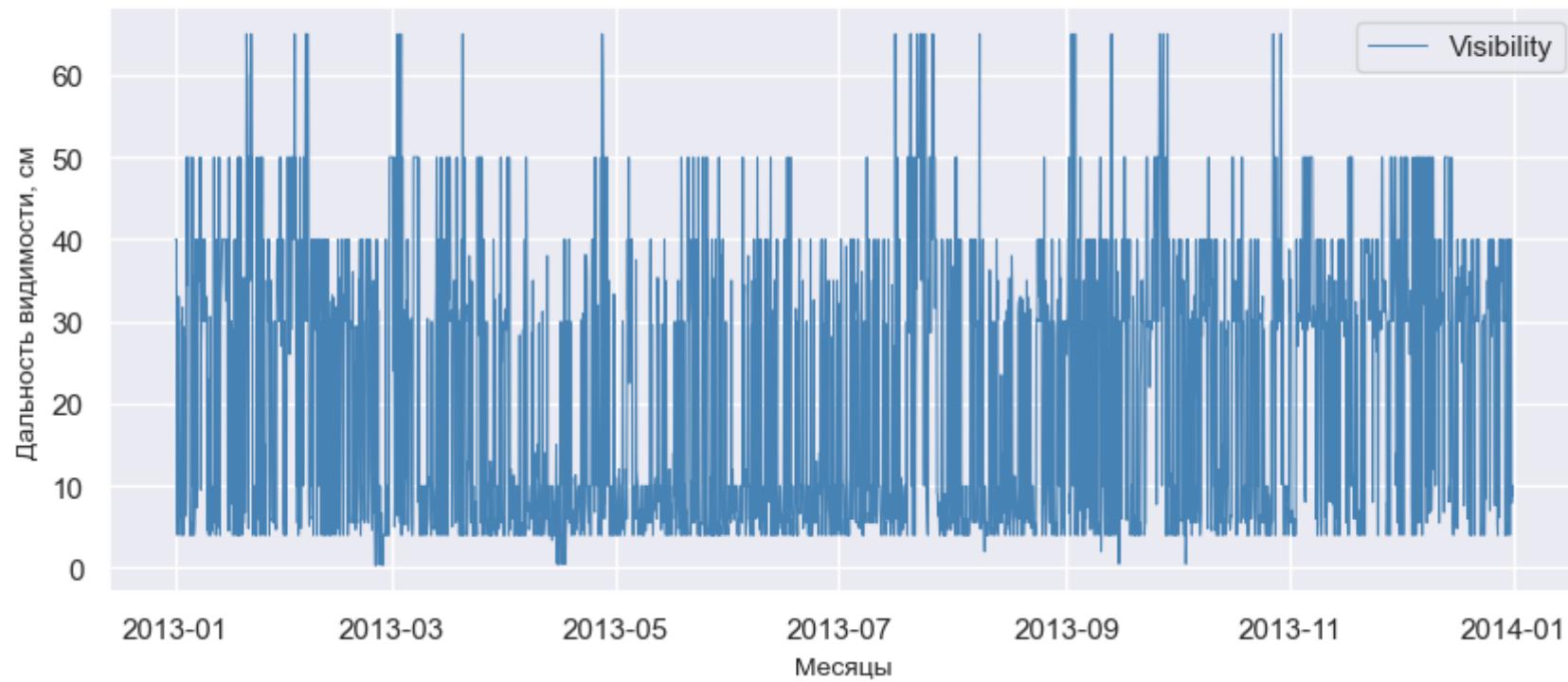
Чашниково: Ежедневная динамика параметра  
 дальности видимости Visibility, 2015 год



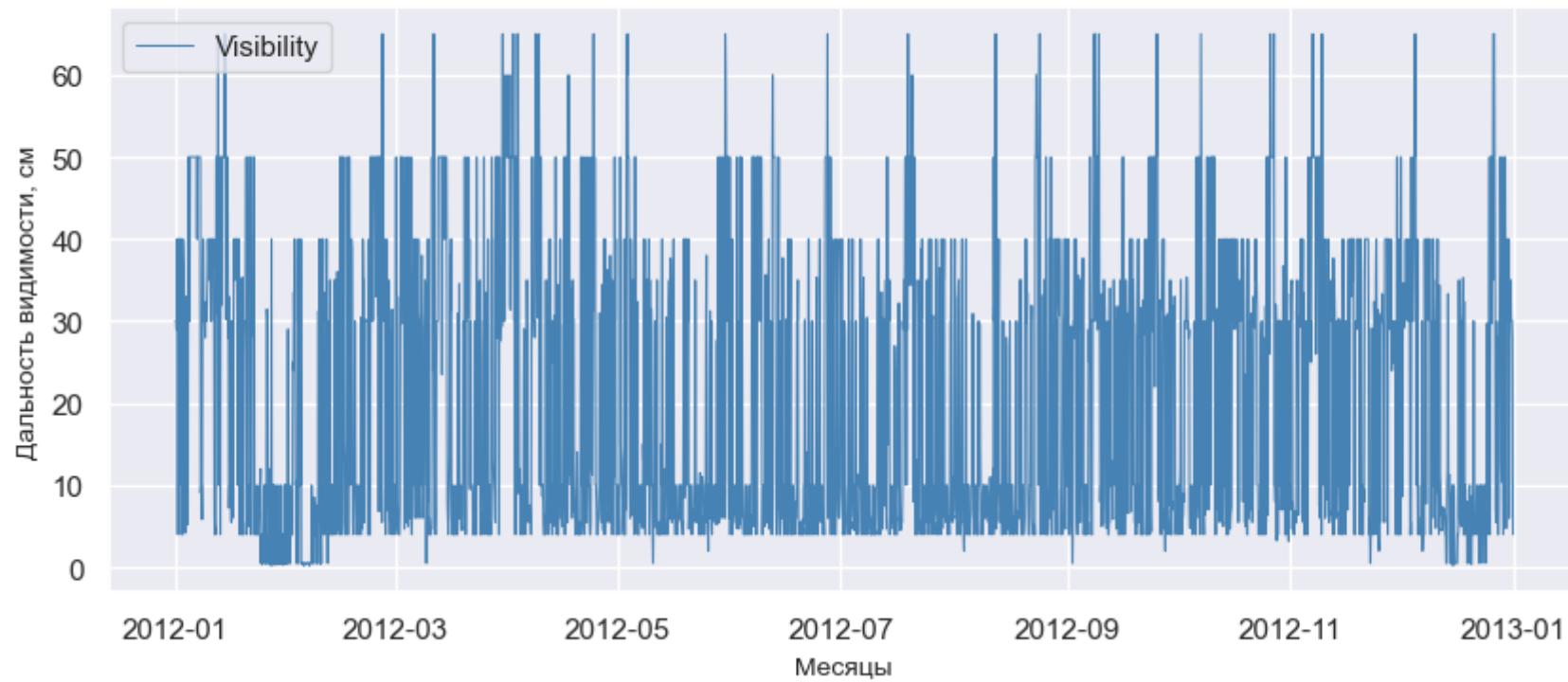
Чашниково: Ежедневная динамика параметра  
 дальности видимости Visibility, 2014 год



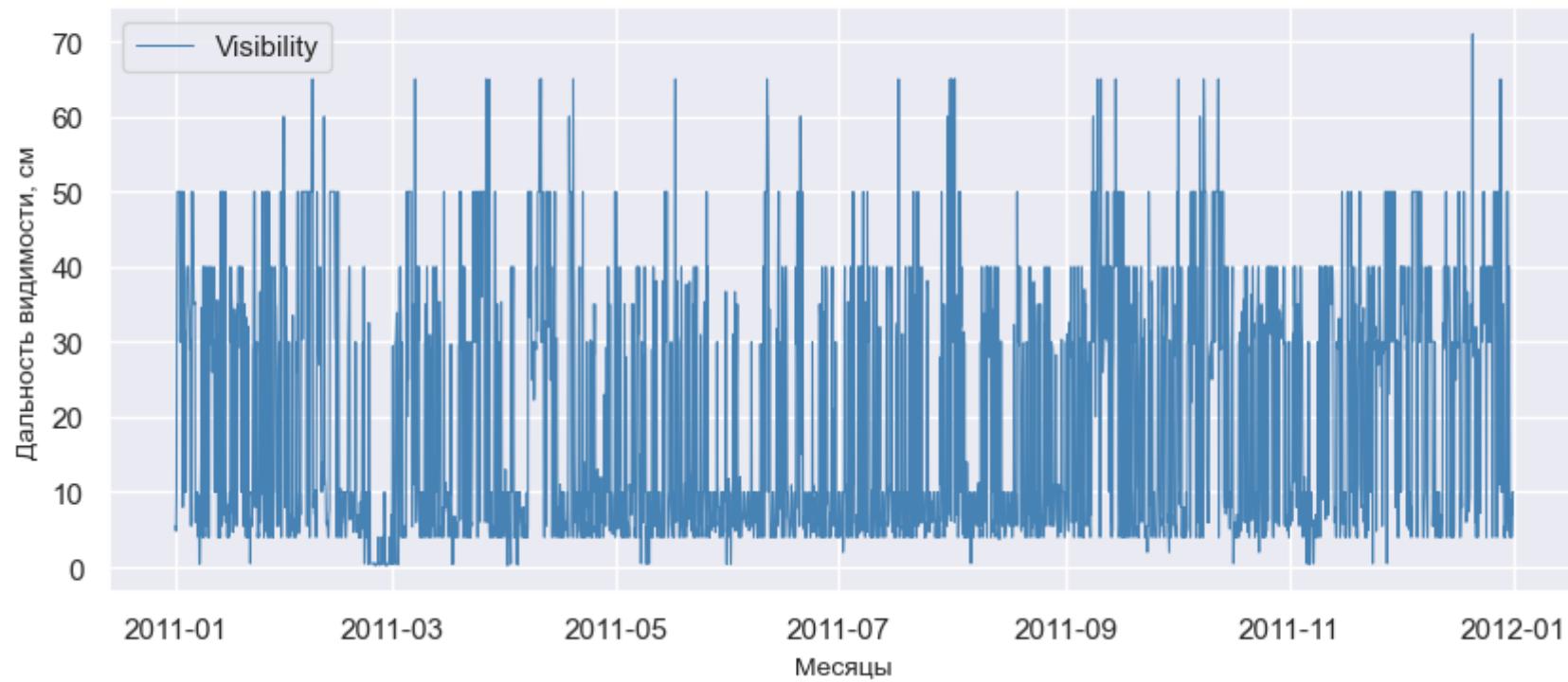
Чашниково: Ежедневная динамика параметра  
 дальности видимости Visibility, 2013 год



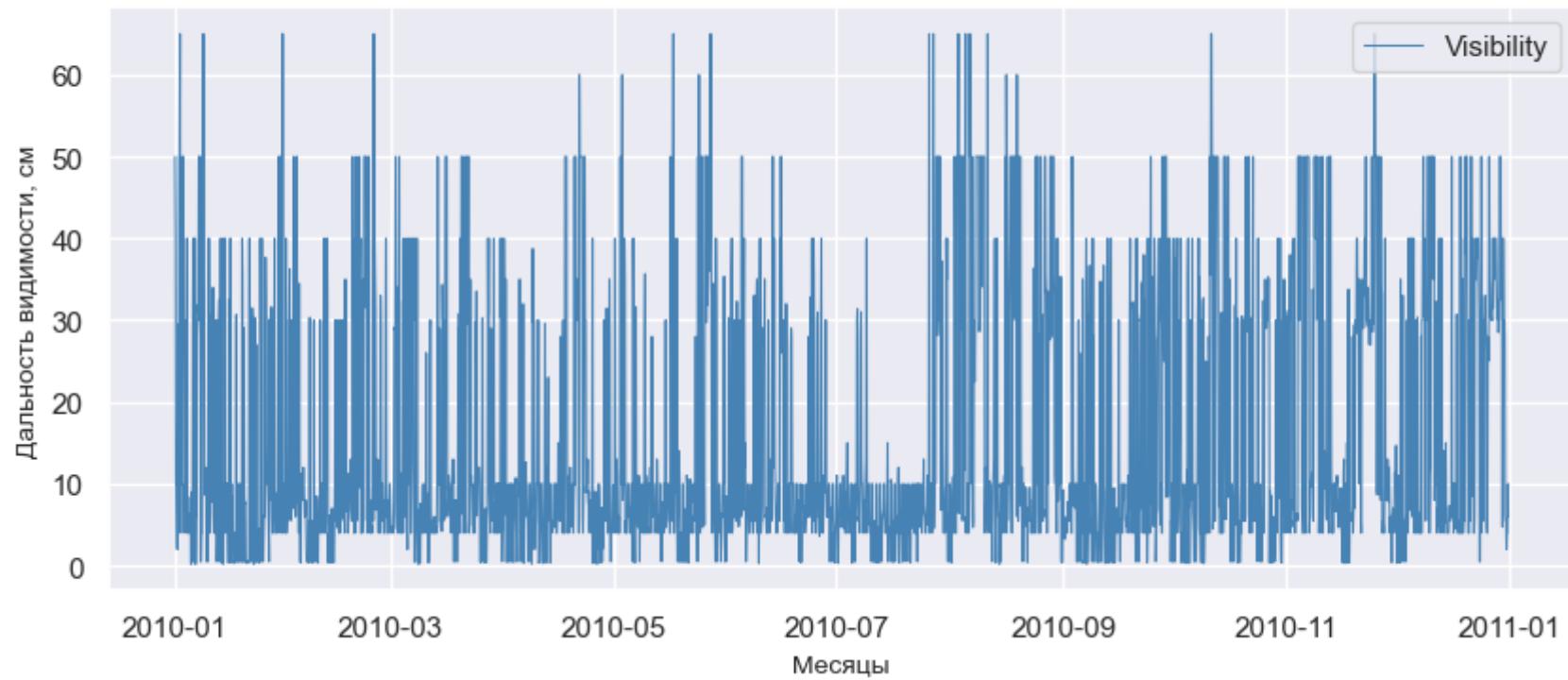
Чашниково: Ежедневная динамика параметра  
 дальности видимости Visibility, 2012 год



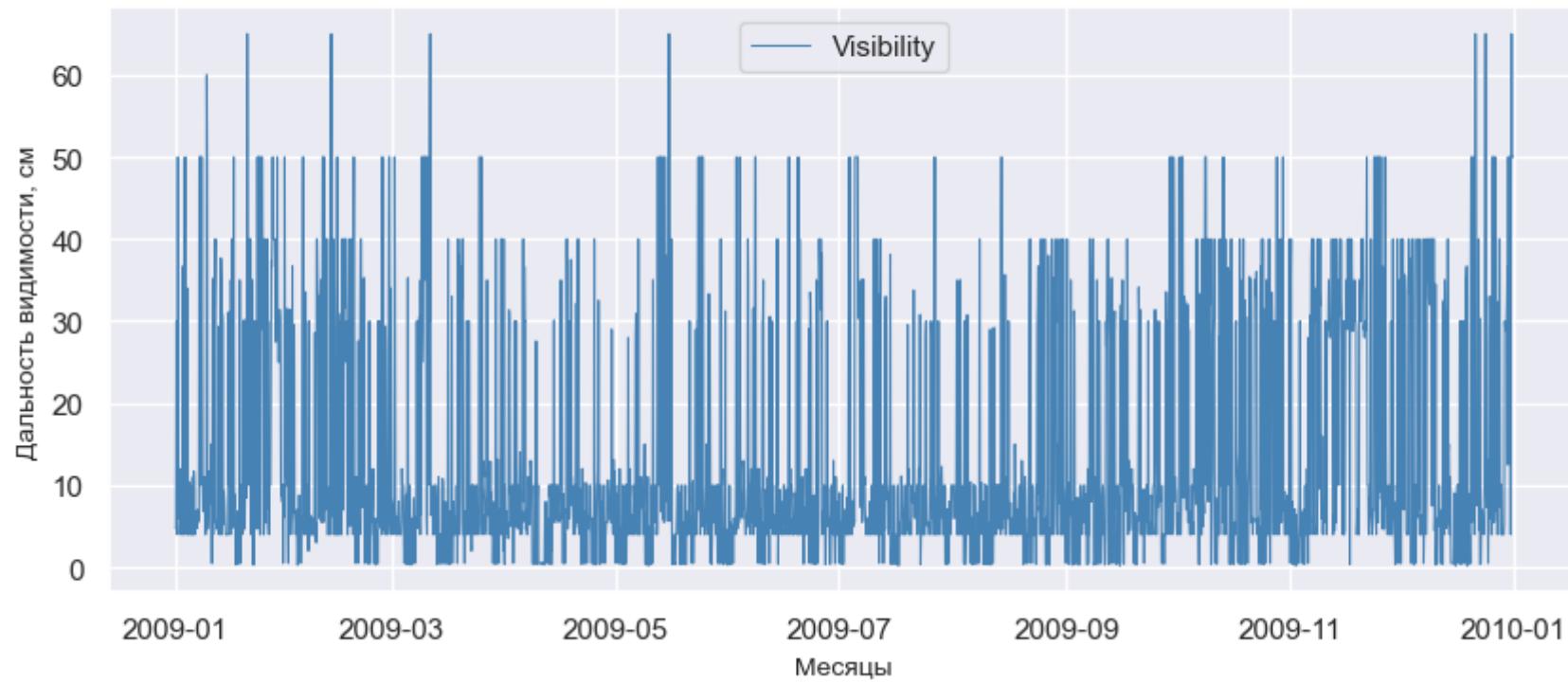
Чашниково: Ежедневная динамика параметра  
 дальности видимости Visibility, 2011 год



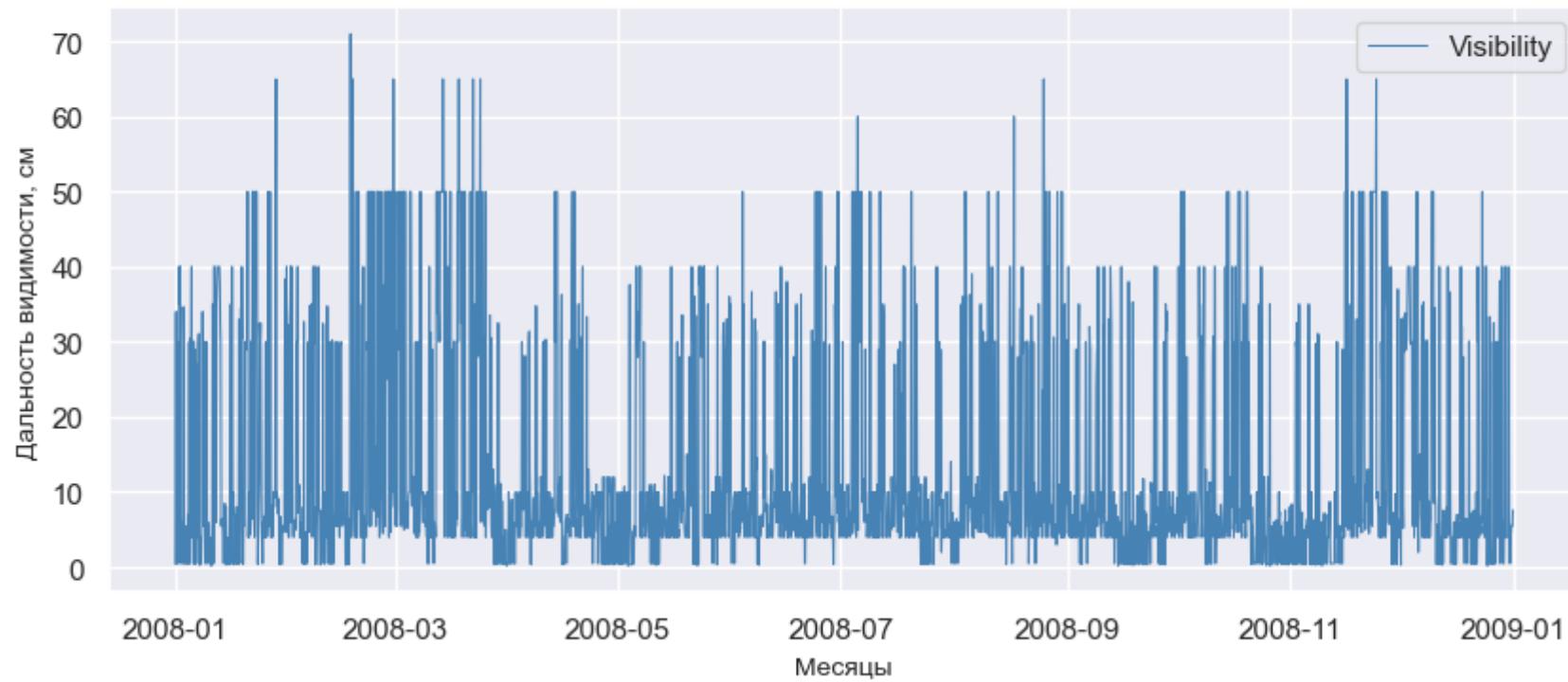
Чашниково: Ежедневная динамика параметра  
 дальности видимости Visibility, 2010 год



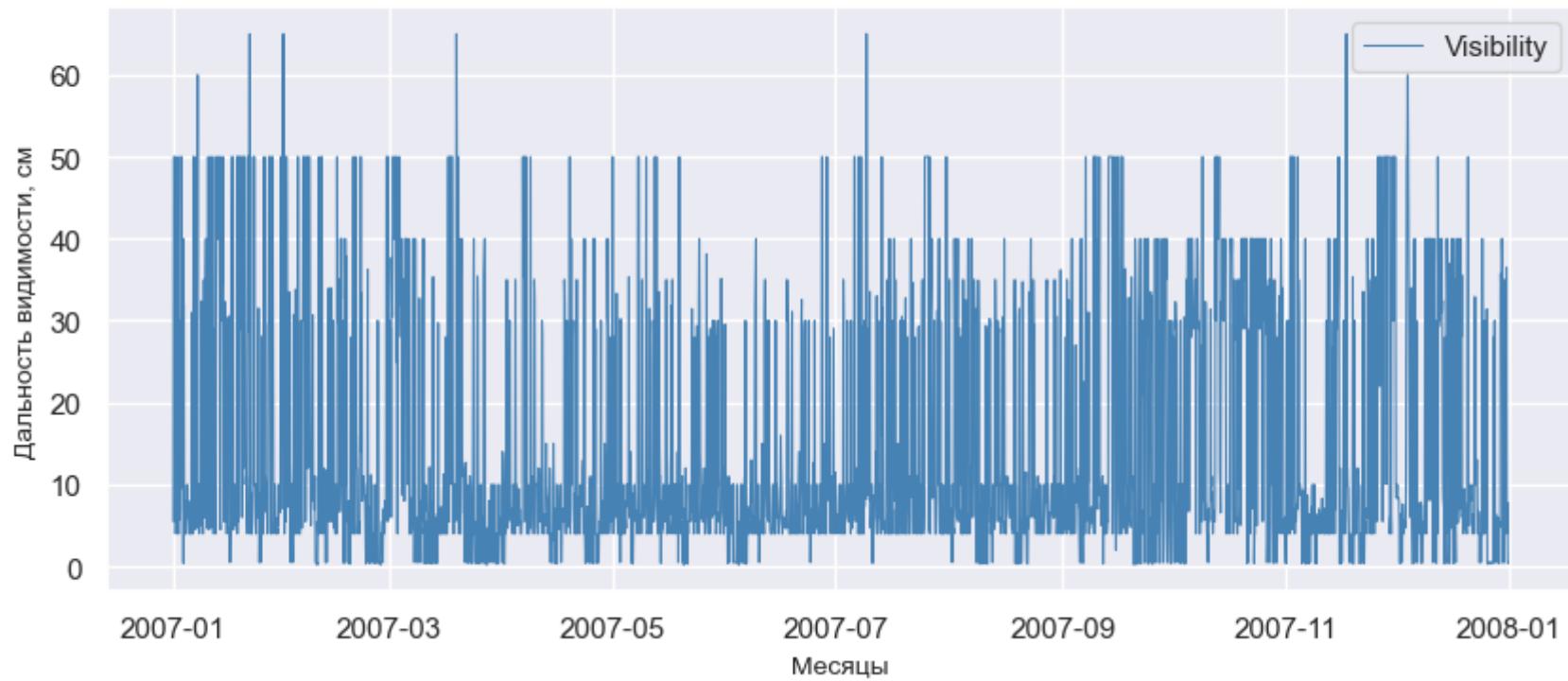
Чашниково: Ежедневная динамика параметра  
 дальности видимости Visibility, 2009 год



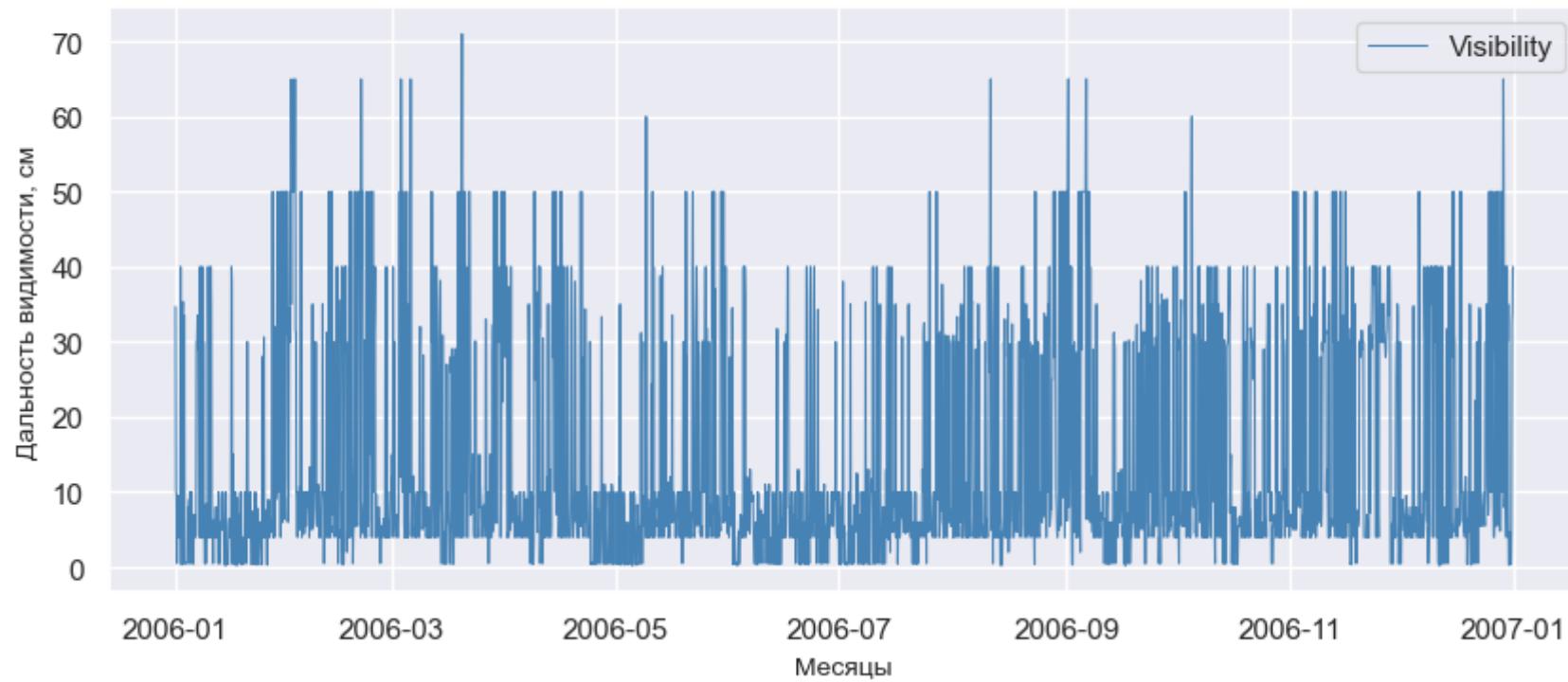
Чашниково: Ежедневная динамика параметра  
 дальности видимости Visibility, 2008 год



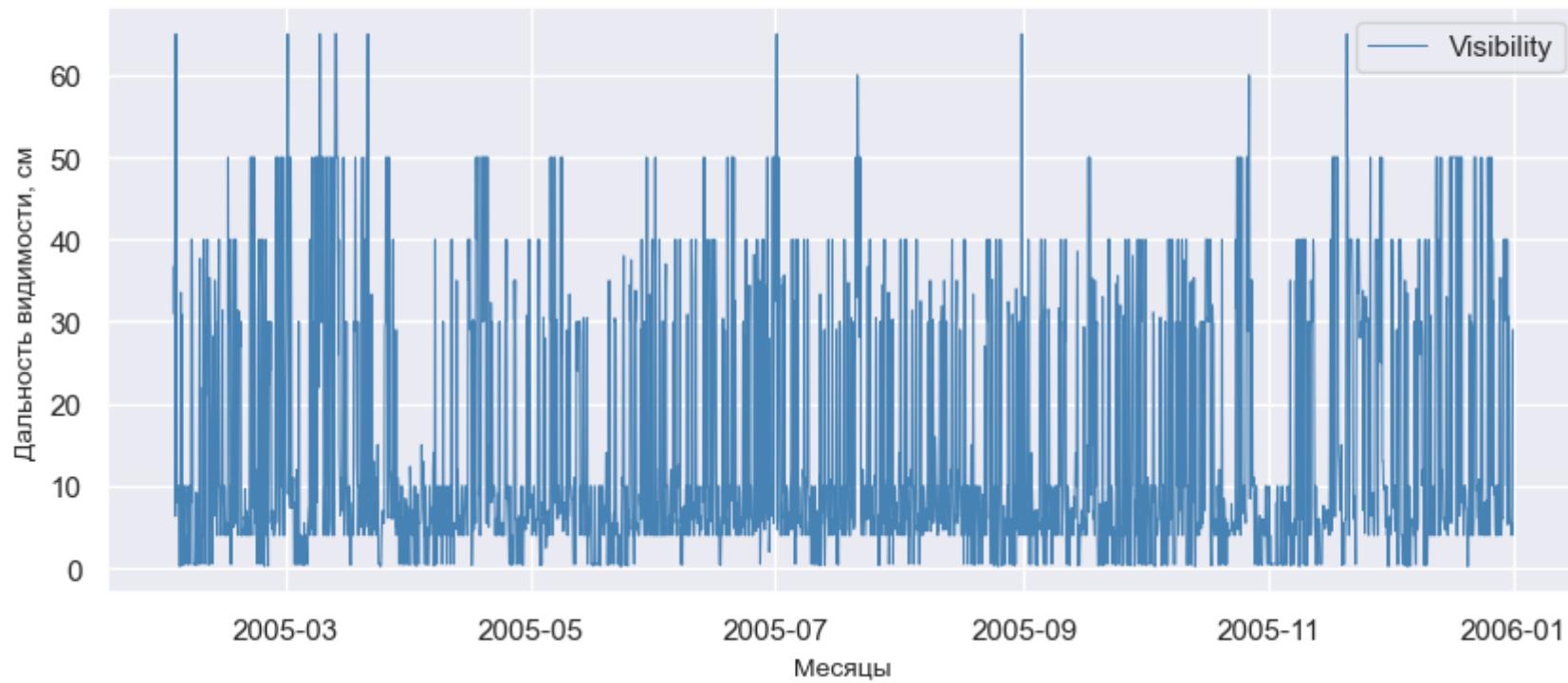
Чашниково: Ежедневная динамика параметра  
 дальности видимости Visibility, 2007 год



Чашниково: Ежедневная динамика параметра  
 дальности видимости Visibility, 2006 год



Чашниково: Ежедневная динамика параметра дальности видимости Visibility, 2005 год



### 7.1.6. Сохранение полученных данных в файлы

```
In [178...]: # # Определённые выше пути к файлам данных:  
# path  
# raw_path1  
# raw_path2  
  
# Создадим новые значения директорий  
predict_path1 = f'{path}predict/{PARAMETER71}/locations/'  
predict_path2 = f'{path}predict/{PARAMETER71}'/  
  
makedirs(predict_path1, exist_ok=True)  
makedirs(predict_path2, exist_ok=True)  
  
# Запишем текущие данные в файлы
```

```

for name in dict_df_locations.keys():
    print(name + '.csv ->', end=' ')
    dict_df_locations[name].to_csv(
        path_or_buf=f'{predict_path1}{name}.csv'
    )
    print('DONE!')

print('df_'+PARAMETER71 + '.csv ->', end=' ')
dict_df_parameters['df_'+PARAMETER71].to_csv(
    path_or_buf=f'{predict_path2}df_{PARAMETER71}.csv'
)
print('DONE!')

```

df\_Chashnikovo.csv -> DONE!  
df\_Dmitrov.csv -> DONE!  
df\_Kashyn.csv -> DONE!  
df\_Klin.csv -> DONE!  
df\_Mozhaisk.csv -> DONE!  
df\_Naro\_Fominsk.csv -> DONE!  
df\_Nemchinovka.csv -> DONE!  
df\_N\_Jerusalem.csv -> DONE!  
df\_Rfrnce\_point.csv -> DONE!  
df\_Serpukhov.csv -> DONE!  
df\_Staritsa.csv -> DONE!  
df\_Tver.csv -> DONE!  
df\_Volokolamsk.csv -> DONE!  
df\_V\_Volochev.csv -> DONE!  
df\_Visibility.csv -> DONE!

## ПРОДОЛЖЕНИЕ В ТЕТЕРАДИ 6

## Разное

Вернём значения отображения к настройкам по умолчанию

In [179...]

```

pd.set_option('display.max_rows', 60) # Восстановим значение по умолчанию максимума отображаемых строк
pd.set_option('display.max_columns', 20) # Восстановим значение по умолчанию максимума отображаемых столбцов

```

In [180...]

```

InteractiveShell.ast_node_interactivity = "last_expr" # Отображение результата только последней строки кода

```

