```
#Overall task of this assignment

'''
You use weather forecasts (wind speed and wind direction)
to predict how much power (in megawatts) the wind turbines will generate. '''
```

```
'\nYou use weather forecasts (wind speed and wind direction)\nto predict how much power (in megawatts) the wind turbines
will generate. '
```

```
#goal of this notebook are step 1 and 2 of the data analytics cycle
```

```
!pip install scikit-learn
import pandas as pd
import numpy as np
import os
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.linear_model import Ridge
```

```
Requirement already satisfied: scikit-learn in /opt/conda/lib/python3.12/site-packages (1.7.2)
Requirement already satisfied: numpy>=1.22.0 in /opt/conda/lib/python3.12/site-packages (from scikit-learn) (2.3.5)
Requirement already satisfied: scipy>=1.8.0 in /opt/conda/lib/python3.12/site-packages (from scikit-learn) (1.16.3)
Requirement already satisfied: joblib>=1.2.0 in /opt/conda/lib/python3.12/site-packages (from scikit-learn) (1.5.2)
Requirement already satisfied: threadpoolctl>=3.1.0 in /opt/conda/lib/python3.12/site-packages (from scikit-learn) (3.6.0)
```

```
#Step 1 Discovery
```

```
#loading the data
df_future  = pd.read_csv("../datasets/future.csv")
df_power   = pd.read_csv("../datasets/power.csv")
df_weather = pd.read_csv("../datasets/weather.csv")
```

```
#discovery future for predicting
df_future.head() #data from UK governmental weather service -->weather forecasts for the FUTURE
'''interpretation of columns
time refers to target time --> the time we want to forecast
lead hours -->difference between target time and source time,i.e., for calculating Source time: target time -3
direction needs to be converted into numeric later on and speed is obviously speed
'''
```

```
'interpretation of columns\ntime refers to target time --> the time we want to forecast\nlead hours -->difference between
target time and source time,i.e., for calculating Source time: target time -3\ndirection needs to be converted into numeric
later on and speed is obviously speed\n'
```

```
df_future.head()
```

	target_time	Direction	Lead_hours	Source_time	Speed
0	2022-03-11 15:00:00+00:00	SE	3	1646996400	15.19936
1	2022-03-11 18:00:00+00:00	SE	6	1646996400	16.09344
2	2022-03-11 21:00:00+00:00	SE	9	1646996400	16.98752
3	2022-03-12 00:00:00+00:00	SSE	12	1646996400	16.09344
4	2022-03-12 03:00:00+00:00	SSE	15	1646996400	12.96416

```
#discovering weather, data from ukmetoffice
df_weather.head() #same starting hour, but DIFFERENT date -->weather forecasts for the past

'''interpretation of columns
# similar to df_future with the difference that source is 1 hour less
'''
```

```
'interpretation of columns\n# similar to df_future with the difference that source is 1 hour less\n'
```

```
#dicovering power
df_power.head() # real electricity outcome in the past
''' interpretation of columns
time is time of measurement
```

```
Total is the renewable generation in Megawatts
    --> the measurement is recorded every minute, representing the instantaneous power output at that timestamp
ANM and Non-ANM are not relevant for this assignment
(same date as weather, but earlier and each minute)
'''
```

```
' interpretation of columns\ntime is time of measurement\nTotal is the renewable generation in Megawatts\n    --> the
measurement is recorded every minute, representing the instantaneous power output at that timestamp\nANM and Non-ANM are
not relevant for this assignment\n(same date as weather, but earlier and each minute)\n'
```

```
#df_power and df_weather is focusing on the past, whereas df_ future is focusing on the future
```

```
#DATA Cleaning
```

```
#handling Null Values and blanks etc
```

```
#for df_future
# Check for Null values
null_counts = df_future.isnull().sum()

# Check for blank or whitespace-only strings
blank_counts = df_future.astype(str).map(lambda x: x.strip() == "").sum()

print("Null values:\n", null_counts)
print("\nBlank/whitespace values:\n", blank_counts)
```

```
Null values:
 time           0
Direction      0
Lead_hours     0
Source_time    0
Speed          0
dtype: int64

Blank/whitespace values:
 time           0
Direction      0
Lead_hours     0
Source_time    0
Speed          0
dtype: int64
```

```
#check for df_power
#for df_future
# Check for Null values
null_counts = df_power.isnull().sum()

# Check for blank or whitespace-only strings
blank_counts = df_power.astype(str).map(lambda x: x.strip() == "").sum()

print("Null values:\n", null_counts)
print("\nBlank/whitespace values:\n", blank_counts)
```

```
Null values:
 time       0
ANM        0
Non-ANM    0
Total      0
dtype: int64

Blank/whitespace values:
 time       0
ANM        0
Non-ANM    0
Total      0
dtype: int64
```

```
#for df_weather
# Check for Null values
null_counts = df_weather.isnull().sum()

# Check for blank or whitespace-only strings
blank_counts = df_weather.astype(str).map(lambda x: x.strip() == "").sum()

print("Null values:\n", null_counts)
print("\nBlank/whitespace values:\n", blank_counts)
```

```
Null values:
 time           0
```

```
Direction      0
Lead_hours     0
Source_time    0
Speed          0
dtype: int64

Blank/whitespace values:
 time           0
Direction      0
Lead_hours     0
Source_time    0
Speed          0
dtype: int64
```

```
df_power.head()
```

	time	ANM	Non-ANM	Total
0	2021-12-11 13:39:00+00:00	10.195558	17.102	27.297558
1	2021-12-11 13:40:00+00:00	10.043559	17.582	27.625559
2	2021-12-11 13:41:00+00:00	10.961558	17.139	28.100558
3	2021-12-11 13:42:00+00:00	11.240559	16.194	27.434559
4	2021-12-11 13:43:00+00:00	10.672559	16.566	27.238559

```
#no null values or blank --> "nothing" to handle
```

```
#check if Lead_hours in df_weather is consistent the same number
df_weather[df_weather['Lead_hours'] != 1]
```

```
  time  Direction  Lead_hours  Source_time  Speed
```

```
#aligning the time lines/ examing them more
'''
Challenge:
The power data (df_power) is recorded at a one-minute resolution, whereas the weather data (df_weather)
is only available every three hours.

Solution:
Aggregate the power data to a 3-hour frequency (e.g., using resampling) so that both datasets share the same temporal reso

Important note: When aligning the datasets, the power values will be aggregated using the mean for the respective 3 hours,
ensuring that all minute-level measurements within each 3-hour window are preserved. Simply discarding the minute-level dat
keeping only the exact 3-hour timestamps would lose valuable information and reduce the representativeness of the power mea
'''
```

```
'\nChallenge:\nThe power data (df_power) is recorded at a one-minute resolution, whereas the weather data (df_weather) \nis
only available every three hours.\n\nSolution:\nAggregate the power data to a 3-hour frequency (e.g., using resampling) so
that both datasets share the same temporal resolution.\n\nImportant note: When aligning the datasets, the power values will
be aggregated using the mean for the respective 3 hours,\nensuring that all minute-level measurements within each 3-hour
window are preserved. Simply discarding the minute-level data and\nkeeping only the exact 3-hour timestamps would lose
valuable information and reduce the representativeness of the power measurements.\n'
```

```
#actual code
df_power['time'] = pd.to_datetime(df_power['time'])
df_power = df_power.set_index('time')

# resampling of 3h intervals
df_power_3h = df_power.resample('3H').mean()

#just using data from 3pm on 2021-12-11 onwards, since df_weather starts there
df_power_3h = df_power_3h[df_power_3h.index.hour >= 15]

#seeing if it worked
df_power_3h.head()
```

```
/tmp/ipykernel_2294/3213745544.py:6: FutureWarning: 'H' is deprecated and will be removed in a future version, please use 'h
  df_power_3h = df_power.resample('3H').mean()
```

	ANM	Non-ANM	Total
time			
2021-12-11 15:00:00+00:00	10.956392	16.664656	27.621048
2021-12-11 18:00:00+00:00	6.927587	14.207956	21.135542
2021-12-11 21:00:00+00:00	6.049698	14.566511	20.616209
2021-12-12 15:00:00+00:00	2.003770	7.424600	9.428370
2021-12-12 18:00:00+00:00	8.335842	13.690172	22.026014

```
'''
Before we can merge the datasets, we need to verify that they have the same number of entries and identical timestamps.
Otherwise, one dataset might contain time points (e.g., on weekends) that the other does not, which would lead to misalignm
'''

print("Power 3h count:", df_power_3h.index.nunique())
print("Weather count:", df_weather['time'].nunique())
```

```
Power 3h count: 270
Weather count: 716
```

```
#--> The timestamps are not identical. To understand the misalignment, we should compare the date ranges covered by both da
last_power_day = df_power_3h.index.max()
last_weather_day = df_weather['time'].max()

print("Last timestamp in Power (3h): ", last_power_day)
print("Last timestamp in Weather:     ", last_weather_day)
```

```
Last timestamp in Power (3h):  2022-03-10 21:00:00+00:00
Last timestamp in Weather:      2022-03-11 12:00:00+00:00
```

```
#one/ half a day difference but does not explaint the huge difference above
#comparing the amount of days
# Tage aus Power (3h)
power_days = df_power_3h.index.normalize().nunique()

# Tage aus Weather
weather_days = pd.to_datetime(df_weather['time']).dt.normalize().nunique()

print("Number of days in Power (3h):  ", power_days)
print("Number of days in Weather:     ", weather_days)
```

```
Number of days in Power (3h):   90
Number of days in Weather:      91
```

```
#one day does not explain the huge difference
power_counts_per_day = df_power_3h.index.to_series().dt.date.value_counts().sort_index()
print("Power timestamps per day:\n", power_counts_per_day)

weather_counts_per_day = pd.to_datetime(df_weather['time']).dt.date.value_counts().sort_index()
print("\nWeather timestamps per day:\n", weather_counts_per_day)
```

```
Power timestamps per day:
 time
2021-12-11    3
2021-12-12    3
2021-12-13    3
2021-12-14    3
2021-12-15    3
             ..
2022-03-06    3
2022-03-07    3
2022-03-08    3
2022-03-09    3
2022-03-10    3
Name: count, Length: 90, dtype: int64

Weather timestamps per day:
 time
2021-12-11    3
2021-12-12    8
2021-12-13    8
2021-12-14    8
2021-12-15    8
```

```
               ..
2022-03-07    8
2022-03-08    8
2022-03-09    8
2022-03-10    8
2022-03-11    4
Name: count, Length: 91, dtype: int64
```

```
#difference is on the timestamps per day
#Solution: "only" merge the timestamps from df_power, which are on df_weather. The goal is to augment df_weather
#with the column Total from df_power, inner join on time
df_weather['time'] = pd.to_datetime(df_weather['time'])

# Merge auf gemeinsame timestamps
df_merged = df_weather.merge(
    df_power_3h[['Total']],
    left_on='time',
    right_index=True,
    how='inner'              # nur gemeinsame Timestamps
)

df_merged.head()
```

	time	Direction	Lead_hours	Source_time	Speed	Total
0	2021-12-11 15:00:00+00:00	SSE	1	1639227600	11.17600	27.621048
1	2021-12-11 18:00:00+00:00	SSW	1	1639238400	8.04672	21.135542
2	2021-12-11 21:00:00+00:00	WSW	1	1639249200	11.17600	20.616209
8	2021-12-12 15:00:00+00:00	SSW	1	1639314000	4.91744	9.428370
9	2021-12-12 18:00:00+00:00	S	1	1639324800	4.91744	22.026014

```
df_merged.shape
```

```
(268, 6)
```

```
#double checking to ensure no Null Values or blanks etc
null_counts = df_merged.isnull().sum()

# Check for blank or whitespace-only strings
blank_counts = df_merged.astype(str).map(lambda x: x.strip() == "").sum()

print("Null values:\n", null_counts)
print("\nBlank/whitespace values:\n", blank_counts)
```

```
Null values:
 time           0
Direction      0
Lead_hours     0
Source_time    0
Speed          0
Total          3
dtype: int64

Blank/whitespace values:
 time           0
Direction      0
Lead_hours     0
Source_time    0
Speed          0
Total          0
dtype: int64
```

```
#appearently 3 null values in column Total

null_rows = df_merged[df_merged['Total'].isna()]
print(null_rows)
```

```
                         time Direction  Lead_hours  Source_time    Speed  \
112 2021-12-25 15:00:00+00:00       SSE           1   1640437200  7.15264
113 2021-12-25 18:00:00+00:00        SE           1   1640448000  8.04672
114 2021-12-25 21:00:00+00:00        SE           1   1640458800  8.94080

     Total
112    NaN
113    NaN
114    NaN
```

```
'''
Since these three missing values all occur on the very first day of measurement,
removing them does not break the temporal continuity of the dataset.
Therefore, dropping these rows is safe and does not affect the sequential structure of the data.
'''
df_merged = df_merged.dropna(subset=['Total'])

# checking if it worked
print(df_merged['Total'].isna().sum())
```

```
0
```

```
#renaming column time to target_time for both df_merged and df_future
df_merged = df_merged.rename(columns={'time': 'target_time'})
df_future = df_future.rename(columns={'time': 'target_time'})
```

```
#now encoding for column direction

'''
The assignment mentions three possible methods for encoding wind direction:
(1) One-Hot Encoding of the cardinal direction strings,
(2) Mapping each direction to an angle (degrees or radians),
(3) Converting wind speed and direction into u/v vector components.

I choose to use the u/v vector representation.
Although this approach introduces two additional numerical features and slightly increases model complexity,
the main advantage is that it reflects the physical nature of wind as a 2D vector and avoids the artificial
discontinuity at 0°/360°. Therefore, the benefits clearly outweigh the drawbacks for this forecasting task.
'''
```

```
'\nThe assignment mentions three possible methods for encoding wind direction:\n(1) One-Hot Encoding of the cardinal
direction strings,\n(2) Mapping each direction to an angle (degrees or radians),\n(3) Converting wind speed and direction
into u/v vector components.\n\nI choose to use the u/v vector representation.\nAlthough this approach introduces two
additional numerical features and slightly increases model complexity,\nthe main advantage is that it reflects the physical
nature of wind as a 2D vector and avoids the artificial \ndiscontinuity at 0°/360°. Therefore, the benefits clearly
outweigh the drawbacks for this forecasting task.\n'
```

```
# encoding direction into vector components



dir_map = {
    "N": 0, "NNE": 22.5, "NE": 45, "ENE": 67.5,
    "E": 90, "ESE": 112.5, "SE": 135, "SSE": 157.5,
    "S": 180, "SSW": 202.5, "SW": 225, "WSW": 247.5,
    "W": 270, "WNW": 292.5, "NW": 315, "NNW": 337.5
}

df_merged["dir_deg"] = df_merged["Direction"].map(dir_map)
df_merged["dir_rad"] = np.deg2rad(df_merged["dir_deg"])

df_merged["u"] = df_merged["Speed"] * np.cos(df_merged["dir_rad"])
df_merged["v"] = df_merged["Speed"] * np.sin(df_merged["dir_rad"])
```

```
df_merged.head()
```

	target_time	Direction	Lead_hours	Source_time	Speed	Total	dir_deg	dir_rad	u	v
0	2021-12-11 15:00:00+00:00	SSE	1	1639227600	11.17600	27.621048	157.5	2.748894	-10.325278	4.276870e+00
1	2021-12-11 18:00:00+00:00	SSW	1	1639238400	8.04672	21.135542	202.5	3.534292	-7.434200	-3.079346e+00
2	2021-12-11 21:00:00+00:00	WSW	1	1639249200	11.17600	20.616209	247.5	4.319690	-4.276870	-1.032528e+01

```
'''
Explination: u = East–West component (horizontal)
    u > 0 → The wind comes from the west and blows toward the east
    u < 0 → The wind comes from the east and blows toward the west
v = North–South component (vertical)
    v > 0 → The wind comes from the south and blows toward the north
    v < 0 → The wind comes from the north and blows toward the south
'''
# columns Direction, dir_reg and dir_rag can be deleted
#df_merged = df_merged.drop(columns=['Direction', 'dir_deg', 'dir_rad'])
```

```
df_merged.head()
```

	target_time	Direction	Lead_hours	Source_time	Speed	Total	dir_deg	dir_rad	u	v
0	2021-12-11 15:00:00+00:00	SSE	1	1639227600	11.17600	27.621048	157.5	2.748894	-10.325278	4.276870e+00
1	2021-12-11 18:00:00+00:00	SSW	1	1639238400	8.04672	21.135542	202.5	3.534292	-7.434200	-3.079346e+00
2	2021-12-11 21:00:00+00:00	WSW	1	1639249200	11.17600	20.616209	247.5	4.319690	-4.276870	-1.032528e+01

```
# before data transforming splitting the dataset to prevent data leakage

'''
Note about "sequential" data:
Although the dataset consists of time-indexed measurements and is therefore sequential in nature,
the assignment does not require classical time-series forecasting. Instead, we model the problem as a supervised regressio
predicting future power output from weather features at the same timestamp. Nevertheless,
because the data is sequential, temporal ordering must be preserved when splitting into train/validation/test sets
to avoid data leakage.
'''

def create_splits(df, seed=42):
    """
    Creates reproducible train/validation/test splits for sequential (time-series-like) data.
    - No shuffling → preserves chronological order (avoids data leakage)
    - First 80% → temp_train
    - Last 20% → test
    - temp_train again split: 80% → train, 20% → validation
      (resulting in 64% train, 16% val, 20% test)
    """

    # Ensure correct time ordering
    df = df.sort_values(by="target_time").reset_index(drop=True)

    n = len(df)

    # 1️⃣ Train/Test split (80/20)
    split_1 = int(n * 0.80)
    temp_train = df.iloc[:split_1]
    test_df = df.iloc[split_1:]

    # 2️⃣ Train/Validation split (80/20 of temp_train)
    split_2 = int(len(temp_train) * 0.80)
    train_df = temp_train.iloc[:split_2]
    val_df = temp_train.iloc[split_2:]

    return train_df, val_df, test_df


# --- Create the splits ---
df_merged = pd.read_csv("/work/bda2/datasets/df_merged.csv")

train_df, val_df, test_df = create_splits(df_merged, seed=42)

print(len(train_df), len(val_df), len(test_df))
```

```
169 43 53
```

```
#check if Lead_hours in df_merged is consistent the same number
df_merged[df_merged['Lead_hours'] != 1]
```

target_time	Direction	Lead_hours	Source_time	Speed	Total	dir_deg	dir_rad	u	v

```
#normalisation
'''
Since my numerical features (Speed, u, v) have highly different sizes, I apply standard scaling
within the pipeline to support better model training and to avoid bias toward features with larger numerical ranges.
Otherwise the "model" could simply think, the higher the number, the more important the feature
'''
```

```
# deciding for a standard scaler
```

```
numeric_features = ["Speed", "u", "v"]

# define preprocessing: scale numerical columns
preprocessor = ColumnTransformer(
    transformers=[
        ("num", StandardScaler(), numeric_features)
    ],
    remainder="passthrough"
)

# developing pipeline so I can use it for train, val and test independently to avoid data leakage
pipeline = Pipeline(steps=[
    ("preprocess", preprocessor),
    ("model", Ridge())
])

# applying fit for train
pipeline.fit(train_df[numeric_features + ["Lead_hours"]], train_df["Total"])

# applying predict for val, test
val_preds = pipeline.predict(
    val_df[numeric_features + ["Lead_hours"]]
)

# Predict on test set
test_preds = pipeline.predict(
    test_df[numeric_features + ["Lead_hours"]]
)

'''
why fit for train and predict for val, test? (both are from library scikit-learn)
We use fit() only on the training set because it teaches the model
how to learn patterns; applying it to validation or test data would cause data leakage.

We use predict() for validation and test because these sets must only be used
to evaluate the already-trained model, never to teach it.
'''
```

```
'\nwhy fit for train and predict for val, test? (both are from library scikit-learn)\nWe use fit() only on the training set
because it teaches the model \nhow to learn patterns; applying it to validation or test data would cause data
leakage.\n\nWe use predict() for validation and test because these sets must only be used \nto evaluate the already-trained
model, never to teach it.\n'
```

```
#saving df_merged for further uswe
output_path = "/work/bda2/datasets/df_merged.csv"


df_merged.to_csv(output_path, index=False)

print(f"df_merged successfully saved to: {output_path}")
```