Modeling air pollution

Let us try to predict Beijing's air pollution, especially PM2.5 values in advance!

Inspiration comes from here.

Dataset

Beijing PM2.5 Data Data Set

Columns of the dataset:

No: row number

year: year of data in this row

month: month of data in this row

day: day of data in this row

hour: hour of data in this row

pm2.5: PM2.5 concentration

DEWP: Dew Point

TEMP: Temperature

PRES: Pressure

cbwd: Combined wind direction

Iws: Cumulated wind speed

Is: Cumulated hours of snow

Ir: Cumulated hours of rain

```
!wget  https://raw.githubusercontent.com/jbrownlee/Datasets/master/pollution.csv
```

显示隐藏的输出项

```
!pip  install  seglearn
```

显示隐藏的输出项

```
import  pandas  as  pd
import  csv
import  numpy  as  np
import  matplotlib.pyplot  as  plt

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

```

```
def  sniff_format(location):
        with  open(location,  newline='')  as  csvfile:
                sniffer  =  csv.Sniffer()
                sample  =  csvfile.read(1024)
                dialect  =  sniffer.sniff(sample)
                header  =  sniffer.has_header(sample)
                if  header:
                        header=0
                else:
                        header=None

        return  {"dialect":dialect,  "header":header}

def  describe_full(df):
        #pd.options.display.float_format  =  '{:.2f}'.format
        dtypes_description=pd.DataFrame(dict(df.dtypes),["dtypes"])
        na_description  =  pd.DataFrame(dict(df.isna().sum()),["NA-s"])
        na_percent  =  ((pd.DataFrame(dict(df.isna().sum()),["NA%"])/len(df))*100).round(decimals=2)
        description  =  df.describe(include='all')
        full_description  =  dtypes_description.append(na_description).append(na_percent).append(description).replace(np.nan,  '',  regex=True)

        mask  =  full_description.loc["freq",:]==1
        full_description.at[["top"],mask.index[mask]]=""
        #TODO:  scientific  notation  -  could  be  nicer

        return  full_description
```

```
csv_format  =  sniff_format("pollution.csv")

df  =  pd.read_csv("pollution.csv",header=csv_format["header"],dialect=csv_format["dialect"])

#There  is  a  warning  that  would  be  worth  investigationg,  but  for  now,  let's  ignore  it
import  warnings
warnings.filterwarnings("ignore")

describe_full(df)
```

	No	year	month	day	hour	pm2.5	DEWP	TEMP	PRES	cbwd	Iws	Is	Ir
dtypes	int64	int64	int64	int64	int64	float64	int64	float64	float64	object	float64	int64	int64
NA-s	0	0	0	0	0	2067	0	0	0	0	0	0	0
NA%	0	0	0	0	0	4.72	0	0	0	0	0	0	0
count	43824	43824	43824	43824	43824	41757	43824	43824	43824	43824	43824	43824	43824
unique										4			
top										SE			
freq										15290			
mean	21912.5	2012	6.52355	15.7278	11.5	98.6132	1.81725	12.4485	1016.45		23.8891	0.0527337	0.194916
std	12651	1.41384	3.44857	8.79942	6.92227	92.0504	14.4334	12.1986	10.2687		50.0106	0.760375	1.41587
min	1	2010	1	1	0	0	-40	-19	991		0.45	0	0
25%	10956.8	2011	4	8	5.75	29	-10	2	1008		1.79	0	0
50%	21912.5	2012	7	16	11.5	72	2	14	1016		5.37	0	0
75%	32868.2	2013	10	23	17.25	137	15	23	1025		21.91	0	0
max	43824	2014	12	31	23	994	28	42	1046		585.6	27	36

```
#Safety  assertions  to  ensure,  that  No  is  as  it  seems  an  index  column  sorted,  series
pd.testing.assert_series_equal(df.No,df.No.sort_values())
np.testing.assert_array_equal((df.No-df.No[0]).values,  np.arange(0,df.shape[0],1))
#Don't  really  need  these  if  we  decide  to  go  for  date  based  indexing
```

```
df.drop("No",axis=1,  inplace=True)
df["date"]=  pd.to_datetime(df['year'].astype(str)+'-'+df['month'].astype(str)+"-"+df["day"].astype(str)+"T"+df["hour"].astype(str).apply(lambda  x:  x.zf
df.set_index(df.date,  inplace=True)
df.drop("date",  axis=1,  inplace=True)
df.head(10)
```

	year	month	day	hour	pm2.5	DEWP	TEMP	PRES	cbwd	Iws	Is	Ir
date												
2010-01-01 00:00:00	2010	1	1	0	NaN	-21	-11.0	1021.0	NW	1.79	0	0
2010-01-01 01:00:00	2010	1	1	1	NaN	-21	-12.0	1020.0	NW	4.92	0	0
2010-01-01 02:00:00	2010	1	1	2	NaN	-21	-11.0	1019.0	NW	6.71	0	0
2010-01-01 03:00:00	2010	1	1	3	NaN	-21	-14.0	1019.0	NW	9.84	0	0
2010-01-01 04:00:00	2010	1	1	4	NaN	-20	-12.0	1018.0	NW	12.97	0	0
2010-01-01 05:00:00	2010	1	1	5	NaN	-19	-10.0	1017.0	NW	16.10	0	0
2010-01-01 06:00:00	2010	1	1	6	NaN	-19	-9.0	1017.0	NW	19.23	0	0

Encoding day of week

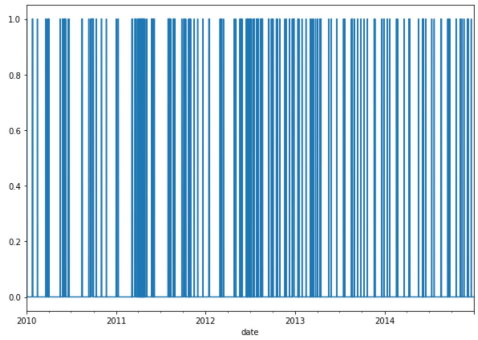
We explicitly encode the day of week, since we assume that weekends and workdays behave differently.

```
df["dayofweek"]=df.index.dayofweek+1
```

Decision about NaN-s

```
fig = plt.gcf()
fig.set_size_inches(10,7)
df["pm2.5"].isnull().astype(float).plot()
```

<matplotlib.axes._subplots.AxesSubplot at 0x7f7c46465d90>




```
def isnullsum(df):
       return df.isnull().sum()
print("1-----")
print("% NaN datapoints per year:",(df.groupby(df.index.year)["pm2.5"].apply(isnullsum)/df.groupby(df.index.year)["pm2.5"].count())*100.0)

print("2------------------------------")


print("% NaN datapoints per month:",(df.groupby(df.index.month)["pm2.5"].apply(isnullsum)/df.groupby(df.index.month)["pm2.5"].count())*100.0)

print("3------------------------------")

print("% NaN datapoints per dayofweek:",(df.groupby(df.index.dayofweek)["pm2.5"].apply(isnullsum)/df.groupby(df.index.dayofweek)["pm2.5"].count())*100

print("4------------------------------")
print("% NaN datapoints per hour:",(df.groupby(df.index.hour)["pm2.5"].apply(isnullsum)/df.groupby(df.index.hour)["pm2.5"].count())*100.0)

```

显示隐藏的输出项

After examining the NaN values in pm2.5, we see no obvious temporal pattern. This is cause for worry, since by simply dropping the rows with NaN values, we can destroy the temporal coherence of the data, hence **data imputation is desirable.**

The autocorrelation charts below imply, that it is not unreasonable to take the previous value to fill NaN-s (high autocorrelation with the previous timestep).

```

df.fillna(method='ffill', inplace=True)

print(df.isnull().sum())

df.dropna(inplace=True)

print(df.isnull().sum())

```

显示隐藏的输出项

Examining autocorrelations

```
from statsmodels.graphics.tsaplots import plot_pacf

#columns = [] #use this for speedup
columns = ["pm2.5","DEWP","TEMP","PRES","Iws","Is","Ir"]

for col in columns:
       plt.figure()
       plot_pacf(df[col].dropna(), lags=200, zero=False)

plt.show()

```

显示隐藏的输出项

What do we see?

Well, the fact, that we don't see.

Or more precisely: smog (and weather) is slow to move, it is extremely strongly autocorrelated with itself one-two hours before, so in order to at least be able to see some autocorrelation structure beyond this, we need to filter out the first some hours from our autocorrelation analysis.

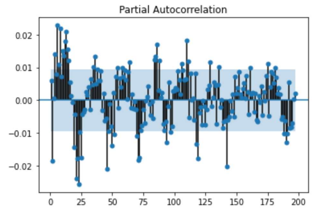
(By the way, that's why we don't stick to the prediction of the next hour as in the original "inspiration" blogpost. Would not be too relevant...)

```
from statsmodels.graphics.tsaplots import _prepare_data_corr_plot, _plot_corr
import statsmodels.graphics.utils as utils
from statsmodels.tsa.stattools import pacf

def plot_pacf_drop(x, ax=None, lags=None, alpha=.05, method='ywunbiased',
                        use_vlines=True, title='Partial Autocorrelation', zero=True,
                        vlines_kwargs=None, drop_no=0, **kwargs):

       lags_orig=lags
       fig, ax = utils.create_mpl_ax(ax)
       vlines_kwargs = {} if vlines_kwargs is None else vlines_kwargs
```

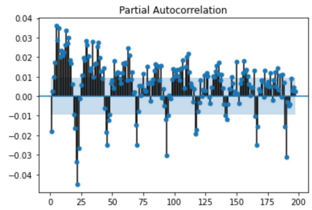
```
    lags, nlags, irregular = _prepare_data_corr_plot(x, lags, zero)
    confint = None
    if alpha is None:
        acf_x = pacf(x, nlags=nlags, alpha=alpha, method=method)
    else:
        acf_x, confint = pacf(x, nlags=nlags, alpha=alpha, method=method)

    if drop_no:
        acf_x = acf_x[drop_no+1:]
        confint = confint[drop_no+1:]
        lags, nlags, irregular = _prepare_data_corr_plot(x, lags_orig=drop_no, zero)

    _plot_corr(ax, title, acf_x, confint, lags, False, use_vlines,
               vlines_kwargs, **kwargs)

    return fig
```

```
import matplotlib.pyplot as plt

#columns = [] #use this for speedup
columns = ["pm2.5","DEWP","TEMP","PRES","Iws","Is","Ir"]

for col in columns:
    plt.figure()
    plot_pacf_drop(df[col].dropna(), lags=200, drop_no=3, zero=False)

plt.show()

```

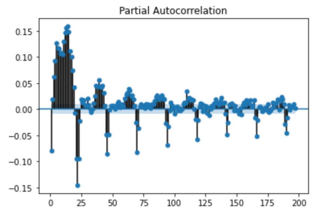
<Figure size 432x288 with 0 Axes>



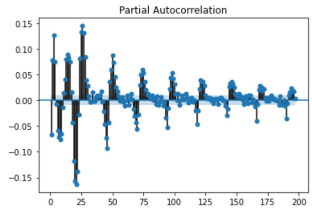
<Figure size 432x288 with 0 Axes>



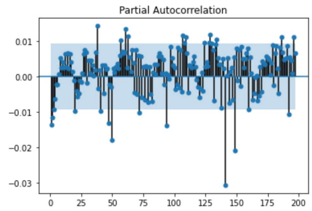
<Figure size 432x288 with 0 Axes>



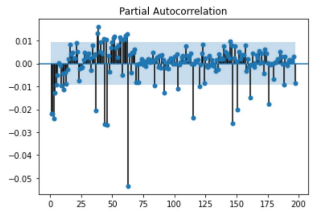
<Figure size 432x288 with 0 Axes>



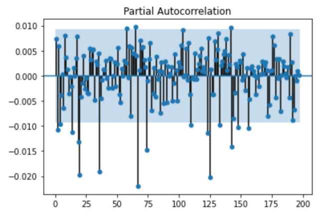
<Figure size 432x288 with 0 Axes>



<Figure size 432x288 with 0 Axes>



<Figure size 432x288 with 0 Axes>



Studying even the filtered charts leaves us in doubt about the possible window for modeling (in case of the classical models), so we will keep 100 as the modeling window (nearly two weeks). This is a parameter that is worth empirically studying later on.

It is worth mentioning, that the `pacf` charts would definitely change drastically if we would use some differencing. Since down below we decide

Seasonal decomposition and the question of trends

```
from statsmodels.tsa.seasonal import seasonal_decompose
from statsmodels.tsa.tsatools import freq_to_period
import matplotlib.pyplot as plt

analysis = seasonal_decompose(df["pm2.5"], freq=freq_to_period(df.index.inferred_freq))

analysis.plot()
plt.show()
```

显示隐藏的输出项

Well, the default setting (infer periods - hourly) is rather uninformative, so it is maybe worth using some domain knowledge here, and use yearly frequency.

```
analysis = seasonal_decompose(df["pm2.5"], freq=24*365)

analysis.plot()
plt.show()
```

显示隐藏的输出项

We do get the first impression, that there is no overarching simple trend, as well as there are non-trivial seasonal patterns. At a later stage we should investigate differencing regimes, but for now, we leave the data as is.

Train, valid, test split - before normalization

Contamination by the normalization values is a distant possibility, but let's stick to paranoid practices.

```
VALID_AND_TEST_SIZE=0.1
```

```
from sklearn.model_selection import train_test_split

X_train, X_else, y_train, y_else = train_test_split(df, df["pm2.5"], test_size=VALID_AND_TEST_SIZE*2, shuffle=False)
X_valid, X_test, y_valid, y_test = train_test_split(X_else, y_else, test_size=0.5, shuffle=False)

```

We could have used `temporal_split` from `seglearn`, but that would have cast everything to numpy, so it was more convenient this way for now. Using `seglearn` is encouraged - if we would like to go into classical modeling.

Data normalization

Our default assumption is to use Scikit's minmax scaler for easier learning by neural models.

But there are some exceptions:

How to normalize dates?

For the year it is more tricky, it is basically an ordinal. Subtracting the first year is nice, but how to handle the normalization to 0,1?

We could use 2018 as a max, but **WE WOULD HAVE TO WRITE A BIG CAVEAT MESSAGE FOR DEPLOY PEOPLE!**

So it should be something like `(df.year - (df.year.min())-1)/((df.year.max()-df.year.min())*2)` (-1 is for avoiding zero, making the life of the network more easy...)

For now we stick to the minmax scaler (living risky... :-)

For month, day, hour default assumption is, scikit's minmax scaler could work, but we will choose a more elaborate solution from [here](). This capitalizes on the circular nature of these quasi ordinals.

```
from sklearn.preprocessing import MinMaxScaler
import warnings
from sklearn.exceptions import DataConversionWarning
warnings.filterwarnings(action='ignore', category=DataConversionWarning)
# I literally hate when a standard Scikit function throws big bunches of warnings
# - though suppressing them is a dangerous practice. Hence this comment.

def minmax_scale(df_x,series_y, normalizers=None):
    features_to_minmax = ["year","pm2.5","DEWP","TEMP","PRES","Iws","Is","Ir"]

    if not normalizers:
        normalizers = {}

    for feat in features_to_minmax:
        if feat not in normalizers:
            normalizers[feat] = MinMaxScaler()
            normalizers[feat].fit(df_x[feat].values.reshape(-1, 1))

        df_x[feat] = normalizers[feat].transform(df_x[feat].values.reshape(-1, 1))

    series_y=normalizers["pm2.5"].transform(series_y.values.reshape(-1, 1))

    return df_x, series_y, normalizers
```

```

X_train_norm, y_train_norm, normalizers = minmax_scale(X_train, y_train)
X_valid_norm, y_valid_norm, _ = minmax_scale(X_valid, y_valid, normalizers=normalizers)
X_test_norm, y_test_norm, _ = minmax_scale(X_test, y_test, normalizers=normalizers)

```

```
normalizers
```

```
{'DEWP': MinMaxScaler(copy=True, feature_range=(0, 1)),
 'Ir': MinMaxScaler(copy=True, feature_range=(0, 1)),
 'Is': MinMaxScaler(copy=True, feature_range=(0, 1)),
 'Iws': MinMaxScaler(copy=True, feature_range=(0, 1)),
 'PRES': MinMaxScaler(copy=True, feature_range=(0, 1)),
 'TEMP': MinMaxScaler(copy=True, feature_range=(0, 1)),
 'pm2.5': MinMaxScaler(copy=True, feature_range=(0, 1)),
 'year': MinMaxScaler(copy=True, feature_range=(0, 1))}
```

```
X_train_norm
#y_train_norm
```

	year	month	day	hour	pm2.5	DEWP	TEMP	PRES	cbwd	Iws	Is	Ir	dayofweek
date													
2010-01-02 00:00:00	0.0	1	2	0	0.129779	0.278689	0.250000	0.527273	SE	0.002290	0.000000	0.0	6
2010-01-02 01:00:00	0.0	1	2	1	0.148893	0.295082	0.250000	0.527273	SE	0.003811	0.000000	0.0	6
2010-01-02 02:00:00	0.0	1	2	2	0.159960	0.360656	0.233333	0.545455	SE	0.005332	0.000000	0.0	6
2010-01-02 03:00:00	0.0	1	2	3	0.182093	0.426230	0.233333	0.563636	SE	0.008391	0.037037	0.0	6

Encoding of ordinals

The encoding of `cbwd` is interesting, since it is an ordinal again, or better to say not even that, it has a nice circular topology, so we will use the same sin-cos solution.

Problem is, that there is a valid "zero" value, marked "cv" in there. We are tempted to replace that with 0.

```

def encode_cyclicals(df_x):
    #"month","day","hour", "cdbw", "dayofweek"

    DIRECTIONS = {"N":1.0,"NE":2.0, "E":3.0, "SE":4.0, "S":5.0, "SW":6.0, "W":7.0, "NW":8.0, "cv":np.nan}

    df_x['month_sin'] = np.sin(2*np.pi*df_x.month/12)
    df_x['month_cos'] = np.cos(2*np.pi*df_x.month/12)
    df_x.drop('month', axis=1, inplace=True)

    df_x['day_sin'] = np.sin(2*np.pi*df_x.day/31)
    df_x['day_cos'] = np.cos(2*np.pi*df_x.day/31)
    df_x.drop('day', axis=1, inplace=True)

    df_x['dayofweek_sin'] = np.sin(2*np.pi*df_x.dayofweek/7)
    df_x['dayofweek_cos'] = np.cos(2*np.pi*df_x.dayofweek/7)
    df_x.drop('dayofweek', axis=1, inplace=True)

    df_x['hour_sin'] = np.sin(2*np.pi*df_x.hour/24)
    df_x['hour_cos'] = np.cos(2*np.pi*df_x.hour/24)
    df_x.drop('hour', axis=1, inplace=True)

    df_x.replace({'cbwd': DIRECTIONS}, inplace=True)
    df_x['cbwd'] = df_x['cbwd'].astype(np.float64)

    df_x['cbwd_sin'] = np.sin(2.0*np.pi*df_x.cbwd/8.0)
    df_x['cbwd_sin'].replace(np.nan, 0.0, inplace=True) #Let's handle the case with no wind specially
    df_x['cbwd_cos'] = np.cos(2.0*np.pi*df_x.cbwd/8.0)
    df_x['cbwd_cos'].replace(np.nan, 0.0, inplace=True) #Let's handle the case with no wind specially
    df_x.drop('cbwd', axis=1, inplace=True)

    return df_x
```

```
X_train_norm = encode_cyclicals(X_train_norm)
X_valid_norm = encode_cyclicals(X_valid_norm)
X_test_norm = encode_cyclicals(X_test_norm)
```

```
X_train_norm
```

	year	pm2.5	DEWP	TEMP	PRES	Iws	Is	Ir	month_sin	month_cos	day_sin	day_cos	dayofweek_sin	dayofweek_cos	hour_sin	hour_cos	cbwd_sin	cbwd_cos
date																		
2010-01-02 00:00:00	0.0	0.129779	0.278689	0.250000	0.527273	0.002290	0.000000	0.0	5.000000e-01	0.866025	3.943559e-01	0.918958	-0.781831	0.623490	0.000000	1.000000	1.224647e-16	-1.0
2010-01-02 01:00:00	0.0	0.148893	0.295082	0.250000	0.527273	0.003811	0.000000	0.0	5.000000e-01	0.866025	3.943559e-01	0.918958	-0.781831	0.623490	0.258819	0.965926	1.224647e-16	-1.0
2010-01-02 02:00:00	0.0	0.159960	0.360656	0.233333	0.545455	0.005332	0.000000	0.0	5.000000e-01	0.866025	3.943559e-01	0.918958	-0.781831	0.623490	0.500000	0.866025	1.224647e-16	-1.0
2010-01-02 03:00:00	0.0	0.182093	0.426230	0.233333	0.563636	0.008391	0.037037	0.0	5.000000e-01	0.866025	3.943559e-01	0.918958	-0.781831	0.623490	0.707107	0.707107	1.224647e-16	-1.0
2010-01-02 04:00:00	0.0	0.138833	0.426230	0.233333	0.563636	0.009912	0.074074	0.0	5.000000e-01	0.866025	3.943559e-01	0.918958	-0.781831	0.623490	0.866025	0.500000	1.224647e-16	-1.0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
2013-12-31 19:00:00	1.0	0.022133	0.229508	0.433333	0.400000	0.195540	0.000000	0.0	-2.449294e-16	1.000000	-2.449294e-16	1.000000	0.974928	-0.222521	-0.965926	0.258819	-2.449294e-16	1.0
2013-12-31 20:00:00	1.0	0.018109	0.196721	0.433333	0.418182	0.203948	0.000000	0.0	-2.449294e-16	1.000000	-2.449294e-16	1.000000	0.974928	-0.222521	-0.866025	0.500000	-2.449294e-16	1.0
2013-12-31 21:00:00	1.0	0.023139	0.196721	0.433333	0.418182	0.213877	0.000000	0.0	-2.449294e-16	1.000000	-2.449294e-16	1.000000	0.974928	-0.222521	-0.707107	0.707107	-2.449294e-16	1.0
2013-12-31 22:00:00	1.0	0.020121	0.196721	0.416667	0.418182	0.222285	0.000000	0.0	-2.449294e-16	1.000000	-2.449294e-16	1.000000	0.974928	-0.222521	-0.500000	0.866025	-2.449294e-16	1.0
2013-12-31 23:00:00	1.0	0.023139	0.213115	0.433333	0.418182	0.234504	0.000000	0.0	-2.449294e-16	1.000000	-2.449294e-16	1.000000	0.974928	-0.222521	-0.258819	0.965926	-2.449294e-16	1.0

35040 rows × 18 columns

```
#Just in case to ensure we did everything right
assert all(x==np.float64 for x in list(X_train_norm.dtypes))
```

It would be worth checking with some assertions that the manual normalizers work well. Let's leave it to later work.

It is also worth noting, that the normalizers should be saved and used in production.

Creating target (y) and "windows" (X) for modeling

By default we use the next 24 hour value of "pm2.5" for prediction, that is, I would like to predict what the pm2.5 will be like **at this hour 24 hours from now.**

We use the quite handy **seglearn** package for this.

Because of computational reasons, we **use the window of 100 hours** to predict. Classical models would have hard time to accommodate substantially (like 5-10x) context windows, LSTM-s would suffer from the challenge of long term memory. After a basic run of modeling the next big challenge would be to investigate PACF structure more and use eg. stateful LSTM modeling to try to accommodate the large "lookback".

```
TIME_WINDOW=100
FORECAST_DISTANCE=24
```

```
from seglearn.transform import FeatureRep, SegmentXYForecast, last

segmenter = SegmentXYForecast(width=TIME_WINDOW, step=1, y_func=last, forecast=FORECAST_DISTANCE)

X_train_rolled, y_train_rolled,_=segmenter.fit_transform([X_train_norm.values],[y_train_norm.flatten()])
```

```
X_train_rolled[:1]
```

```
array([[[ 0.00000000e+00,  1.29778672e-01,  2.78688525e-01, ...,
          1.00000000e+00,  1.22464680e-16, -1.00000000e+00],
        [ 0.00000000e+00,  1.48893360e-01,  2.95081967e-01, ...,
          9.65925826e-01,  1.22464680e-16, -1.00000000e+00],
        [ 0.00000000e+00,  1.59959759e-01,  3.60655738e-01, ...,
          8.66025404e-01,  1.22464680e-16, -1.00000000e+00],
        ...,
        [ 0.00000000e+00,  7.74647887e-02,  1.31147541e-01, ...,
          9.65925826e-01,  1.00000000e+00,  6.12323400e-17],
        [ 0.00000000e+00,  5.03018109e-02,  1.14754098e-01, ...,
          8.66025404e-01,  1.00000000e+00,  6.12323400e-17],
        [ 0.00000000e+00,  4.42655936e-02,  1.14754098e-01, ...,
          7.07106781e-01,  1.00000000e+00,  6.12323400e-17]]])
```

For non-sequence models

We have to "flatten" the data to be able to use classical, non-sequence regression models from Scikit.

We only need to do this for X, any transformation of y is unnecessary.

```
X_train_rolled.shape
```

```
(34917, 100, 18)
```

```
shape = X_train_rolled.shape
X_train_flattened = X_train_rolled.reshape(shape[0],shape[1]*shape[2])
X_train_flattened.shape
```

```
(34917, 1800)
```

```
X_valid_rolled, y_valid_rolled,_=segmenter.fit_transform([X_valid_norm.values],[y_valid_norm.flatten()])

shape = X_valid_rolled.shape
X_valid_flattened = X_valid_rolled.reshape(shape[0],shape[1]*shape[2])
```

Evaluation helper

Use this function to evaluate your models **on validation data.**

This assumes that your model has the `predict()` function, which is true for **Scikit-learn, XGBoost** and **Keras**, so you can can hand over any of those.

A special issue by models optimized by iterative methods is to **get the final model**. **Early stopping and / or model save and reload** can help there.

WARNING: This is just a basic evaluation scheme, more thorough investigation needed in the future!

```
from sklearn.metrics import mean_squared_error
from math import sqrt

def evaluate_model(model, X_valid, y_valid_true):
    predictions = model.predict(X_valid)
    rms = sqrt(mean_squared_error(y_valid_true, predictions))
    print("Root mean squared error on valid:",rms)
    normalized_rms = normalizers["pm2.5"].inverse_transform(np.array([rms]).reshape(1, -1))[0][0]
    print("Root mean squared error on valid inverse transformed from normalization:",normalized_rms)
    return normalized_rms
```

Classical modeling

In "classical" modeling we assume a multiple regression case, so we **DO NOT USE time series as such, but the "flat" versions of the data** as input. Output is the same.

Baseline - DummyPredictor

TASK Create a dummy predictor as a baseline. Use Scikit-learn's builtin capability to do dummy models in regression case. Use the default setting, that is the prediction of the mean value.

```
from sklearn.dummy import DummyRegressor
dummy_model = DummyRegressor(strategy="mean")

dummy_model.fit(X_train_flattened,y_train_rolled)
```

```
DummyRegressor(constant=None, quantile=None, strategy='mean')
```

Evaluation

```
result = evaluate_model(dummy_model,X_valid_flattened,y_valid_rolled)
```

```
Root mean squared error on valid: 0.0986950073704243
Root mean squared error on valid inverse transformed from normalization: 98.10283732620175
```

Fitting a RandomForest on raw data

TASK: Fit a RandomForest from Scikit. Please be aware, that the number of trees in the model is having a strong influence on training time.

Suggestion: use couple of tens of trees, definitely << 100 to be able to wait it out...

Pro tip: To utilize all the CPU cores, use the right setting of n_jobs. That speeds things up.

```
#from sklearn...
from sklearn.ensemble import RandomForestRegressor
N_ESTIMATORS = 5
RANDOM_STATE = 452543634
```

```
RF_base_model = RandomForestRegressor(n_estimators=N_ESTIMATORS,random_state=RANDOM_STATE,n_jobs=-1)

RF_base_model.fit(X_train_flattened,y_train_rolled)
```

```
RandomForestRegressor(bootstrap=True, ccp_alpha=0.0, criterion='mse',
                      max_depth=None, max_features='auto', max_leaf_nodes=None,
                      max_samples=None, min_impurity_decrease=0.0,
                      min_impurity_split=None, min_samples_leaf=1,
                      min_samples_split=2, min_weight_fraction_leaf=0.0,
                      n_estimators=5, n_jobs=-1, oob_score=False,
                      random_state=452543634, verbose=0, warm_start=False)
```

Evaluation

```
result = evaluate_model(RF_base_model,X_valid_flattened,y_valid_rolled)
```

```
Root mean squared error on valid: 0.09564886485904463
Root mean squared error on valid inverse transformed from normalization: 95.07497166989035
```

Fitting a RandomForest on feature transformed data

TASK: Since we use `seglearn` , we can try to capitalize on it's functionality to calculate features from the time time series. Use `FeatureRep` from `seglearn` to transform features, fit a RandomForest and hope for the best!

```
#from seglearn.pipe import Pype
#from seglearn.transform import FeatureRep, Segment
#from seglearn.pipe import Pype
#RF_feature_model =Pype([('features', FeatureRep()),
                       # ('rf', RandomForestRegressor(n_estimators=N_ESTIMATORS,random_state=RANDOM_STATE,n_jobs=-1))])
RF_feature_model=RandomForestRegressor(n_estimators=10,random_state=RANDOM_STATE,n_jobs=-1)
feature_converter = FeatureRep()

RF_feature_model.fit(feature_converter.fit_transform(X_train_flattened),y_train_rolled)

```

```
RandomForestRegressor(bootstrap=True, ccp_alpha=0.0, criterion='mse',
                      max_depth=None, max_features='auto', max_leaf_nodes=None,
                      max_samples=None, min_impurity_decrease=0.0,
                      min_impurity_split=None, min_samples_leaf=1,
                      min_samples_split=2, min_weight_fraction_leaf=0.0,
                      n_estimators=10, n_jobs=-1, oob_score=False,
                      random_state=452543634, verbose=0, warm_start=False)
```

Evaluation

WARNING: This is just a basic evaluation scheme, more thorough investigation needed!

```
result = evaluate_model(RF_feature_model,feature_converter.fit_transform(X_valid_flattened),y_valid_rolled)
```

```
Root mean squared error on valid: 0.10491616432748488
Root mean squared error on valid inverse transformed from normalization: 104.28666734151996
```

XGBoost for speedup

Use the XGBoost library to fit gradient boosted trees to the problem. They are usually way quicker to learn and many times at least on par with RandomForests, or better. Let's see!

```
import xgboost as xgb
# If in trouble, use !pip install xgboost

# XGBoost needs it's custom data format to run quickly
dmatrix_train = xgb.DMatrix(data=X_train_flattened,label=y_train_rolled)
dmatrix_valid = xgb.DMatrix(data=X_valid_flattened,label=y_valid_rolled)
```

```
params = {'objective': 'reg:linear', 'eval_metric': 'rmse', 'n_estimators': N_ESTIMATORS}

evallist = [(dmatrix_valid, 'eval'), (dmatrix_train, 'train')]
#num_round = 2
#bst = xgb.train(param, dtrain, num_round)
num_round = 12 #Can easily overfit, experiment with it!

xg_reg = xgb.train(params,dmatrix_train,num_round)
```

```
[08:35:32] WARNING: /workspace/src/objective/regression_obj.cu:152: reg:linear is now deprecated in favor of reg:squarederror.
```

```
result = evaluate_model(xg_reg,dmatrix_valid,y_valid_rolled)
```

```
Root mean squared error on valid: 0.09036746035938585
Root mean squared error on valid inverse transformed from normalization: 89.82525559722953
```

Building an LSTM model

Modeling assumptions

TASK: We believe, that the time dependent structure of this dataset is complex, so we try to use LSTM models from Keras. We are not explicitly utilizing **statefulness**, that is a **major area to be investigated later on**.

More information on statefulness can be found here.

Fit an LSTM model **on the time series - non-flat - data!**.

Use:

1. At least 1 LSTM layer
2. A dense layer for output - think about activation! This is a regression case!

Very advisable - but optional - to use Dropout. You can not use it everywhere, though... Experiment!

You are allowed to use functional API, but for this **Sequential API is sufficient.**

You **can use LeraningRateScheduler** if you like.

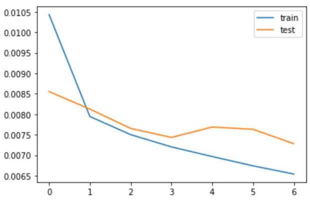
```
LSTM_CELL_SIZE= 18
BATCH_SIZE = 250
EPOCHS = 7
DROPOUT_RATE=0.3
```

```
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense, Dropout, LSTM
from tensorflow.keras import backend as be
from tensorflow.keras.callbacks import ModelCheckpoint, LearningRateScheduler

column_count=len(X_train_norm.columns) #Remember,column count before rolling...

be.clear_session()

# You might very well be needing it!
# Remeber to save only what is worth it from validation perspective...
# model_saver = ModelCheckpoint(...)

# If you need it...
#def schedule(epoch, lr):
#       ...
#       return lr

#lr_scheduler = LearningRateScheduler(schedule)

# Build your whole LSTM model here!
model =Sequential()
model.add(LSTM(units = LSTM_CELL_SIZE, return_sequences = True, input_shape =X_train_rolled.shape[1:]))
model.add(Dropout(DROPOUT_RATE))
model.add(LSTM(units = LSTM_CELL_SIZE, return_sequences = True))
model.add(Dropout(DROPOUT_RATE))

model.add(LSTM(units = LSTM_CELL_SIZE, return_sequences = False))
model.add(Dropout(DROPOUT_RATE))

model.add(Dense(units = 1))
#For shape remeber, we have a variable defining the "window" and the features in the window...

model.compile(loss='mean_squared_error', optimizer='adam')
# Fit on the train data
# USE the batch size parameter!
# Use validation data - warning, a tuple of stuff!
# Epochs as deemed necessary...
# You should avoid shuffling the data maybe.
# You can use the callbacks for LR schedule or model saving as seems fit.
history = model.fit(X_train_rolled,y_train_rolled, epochs=EPOCHS, batch_size=BATCH_SIZE,validation_data =(X_valid_rolled,y_valid_rolled))
```

```
Epoch 1/7
140/140 [==============================] - 30s 186ms/step - loss: 0.0104 - val_loss: 0.0086
Epoch 2/7
140/140 [==============================] - 24s 175ms/step - loss: 0.0079 - val_loss: 0.0081
```

```
Epoch 3/7
140/140 [==============================] - 25s 176ms/step - loss: 0.0075 - val_loss: 0.0077
Epoch 4/7
140/140 [==============================] - 25s 175ms/step - loss: 0.0072 - val_loss: 0.0074
Epoch 5/7
140/140 [==============================] - 24s 174ms/step - loss: 0.0070 - val_loss: 0.0077
Epoch 6/7
140/140 [==============================] - 25s 177ms/step - loss: 0.0067 - val_loss: 0.0076
Epoch 7/7
140/140 [==============================] - 25s 175ms/step - loss: 0.0065 - val_loss: 0.0073
```

```
plt.plot(history.history['loss'],  label='train')
plt.plot(history.history['val_loss'],  label='test')
plt.legend()
plt.show()
```




```
#  You  can  use  the  early  stopped  model  OR  load  it.
#  For  that  you  have  to  import  the  load  function...
#  IF  AND  ONLY  IF  loading,  it  is  good  practice  to  throw  out  the  trash  from  the  graph...
#bc.clear_session()


result  =  evaluate_model(model,X_valid_rolled,y_valid_rolled)
```

```
Root mean squared error on valid: 0.08532126707552125
Root mean squared error on valid inverse transformed from normalization: 84.81232147306811
```

```
assert  result  <  86.0
```

Things that should be improved

- More conclusive investigation of PACF for better time window estimate
  - It can well be, that long windows do not add that much to the performance
- More interesting features for XGBoost (like from tsfresh), since present features are a disaster
- MOST IMPORTANT: **More thorough error / prediction analysis!!!**
- LSTM with **Custom iterator with stateful model**
- Investigation of different loss function (eg. MAE) for training. (And with it, think abut the importance of extreme values: do we think they are outliers? Are they interesing to predict?)
- Investigation of "teacher forcing" for LSTM-s in Keras (if it makes sense)

Conclusion

Even with decent amount of struggle, the "dummy" of always using the mean is very appealing, so it seems, this is not that easy of a task 24 hours in advance. Further investigation of classical as well as neural models remains open!

Final test

We did not use the final test, since our investigations are not concluded yet. Remember: using it once before project "go live" is a good practice!