energy_price_prediction

October 28, 2018

1 Machine Learning Engineer Nanodegree

1.1 Capstone Project

Monish Ananthu October 3rd, 2018 ## ***Energy Price Prediction***

1.2 I. Introduction

The Bundesnetzagentur's electricity market information platform "SMARD" is an abbreviation of the German term "Strommarktdaten", which translates to electricity market data. Data that is published on SMARD's website gives an up-to-date and in-depth overview of what is happening on the German electricity market.

The SMARD Website offers real time data for analysis. This data is available for download in different formats. Link to SMARD Website

The following electricity market data categories can be accessed/downloaded:

- Electricity generation
  - Actual generation
  - Forecasted generation
  - Installed capacity

- Electricity consumption
  - Realised consumption
  - Forecasted consumption
- The market
  - Wholesale market price
  - Commercial exchanges
  - Physical flows
- System stability
  - Balancing energy
  - Total costs
  - Primary balancing capacity
  - Secondary balancing capacity
  - Tertiary balancing reserve
  - Exported balancing energy
  - Imported balancing energy

The above data is available from 2015 onwards. The statistical data available is visuallized and limited to a specific subcategory (for example: Electricity generation --> Actual generation). The visualization does not convey how the data is correlated to one another and also the correlation of data between different catagories like "Actual generation" and "Wholesale market price" would be a very interesting to determine.

What makes SMARD Data so interesting?

- Data is already consolidated from different transmission system operators in a standard format
- High frequency of data (in 15 minute / hourly intervals) provides a good basis for data analysis
- Data available from 2015 ist constantly updated

1.3 II. Problem Statement

The problem to be solved is the prediction of the wholesale market price of energy [Euro/MWh] using the data available above. The problem at hand is a supervised learning problem in the field of Machine Learning. From the **Datasets and Inputs** section below, we have the following input data:

a. Actual generation
b. Realized Consumption
c. Balancing energy

It is important to find correlations among the above input features and use this information to predict the wholesale market price of energy.

All data is available in CSV format

1.4 III. Analysis and Preprocessing

The data sets can be downloaded at https://www.smard.de/en/downloadcenter/download_market_data Select category, sub-category, country = Germany, Dates: 01/01/2015 - 31/12/2015, Filetype: CSV and download file.

We will consider a Data sets for the years 2015 and 2016.

The subcategories below refer to feature sets. If a sub-category is not relevant, all features in the feature set can be discarded. Partial relevance means that a part of the the features need to be considered.

Category	Sub-category	Relevant?	Data frequency	Details
Electricity generation	Actual generation	yes	15 mins	Amount of energy generated by different sources at a specific time period
	Forecasted generation	no	15 mins	Forecasted features are not relevant
	Installed Capacity	no	NA	Not enough data

Category	Sub-category	Relevant?	Data frequency	Details
Electricity consumption	Realized consumption	yes	15 mins	Energy consumption at specific time period
	Forecasted consumption	no	15 mins	Forecasted features are not relevant
The Market	Wholesale market price	yes (partially)	15 mins	Energy price per MWh. Only data for Germany is relevant
	Commercial exchanges	no	60 mins	Energy Imports and Exports are out of scope for price prediction
	Physical flows	no	60 mins	Energy Imports and Exports are out of scope for price prediction

Category	Sub-category	Relevant?	Data frequency	Details
System stability	Balancing energy	yes	15 mins	Overall energy balancing volumes and balancing price
	Total costs	no	monthly	Not enough data
	Primary balancing capacity	no	15 mins	Energy balancing efforts and resulting costs are not in scope
	Secondary bancing capacity	no	15 mins	Energy balancing efforts and resulting costs are not in scope
	Tertiary balancing reserve	no	15 mins	Energy balancing efforts and resulting costs are not in scope

Category	Sub-category	Relevant?	Data frequency	Details
	Exported balancing energy	no	NA	No data available for 2015
	Imported balancing energy	no	NA	No data available for 2015

After the initial screening we have the following features:

Category	Sub-category	Feature	Data frequency	Comments
		Date		Date starting 01/01/2015 - 31/12/2015
		Time of day		Timestamps in 15 min intervals for the date range specified above
Electricity generation	Actual generation	Hydropower[MWh]	15 mins	Generated energy in MWh
		Wind offshore[MWh]	15 mins	Generated energy in MWh
		Wind onshore[MWh]	15 mins	Generated energy in MWh
		Photovoltaics[MWh]	15 mins	Generated energy in MWh
		Other renewable[MWh]	15 mins	Generated energy in MWh
		Nuclear[MWh]	15 mins	Generated energy in MWh
		Fossil brown coal[MWh]	15 mins	Generated energy in MWh

Category	Sub-category	Feature	Data frequency	Comments
		Fossil hard coal[MWh]	15 mins	Generated energy in MWh
		Fossil gas[MWh]	15 mins	Generated energy in MWh
		Hydro pumped storage[MWh]	15 mins	Generated energy in MWh
		Other conventional[MWh]	15 mins	Generated energy in MWh
Electricity consumption	Actual consumption	Total[MWh]	15 mins	Feature name needs to be modified to Total consumption for simplicity
The Market	Wholesale market price	Germany/Austria/Luxembourg[Euro/MWh]	60 mins	Market prices of other countries are not relevant and need not be considered. The feature name will be renamed to Price Germany for simplicity
System stability	Balancing energy	Balancing energy volume[MWh]	15 mins	Balancing energy in MWh
		Balancing energy price[Euro/MWh]	15 mins	Price for balancing energy Euro/MWh

Each of the features in the dataset contains a value for a particular time period/interval. The feature we like to predict "Wholesale market price" is available every 60 minutes. This implies that the input features which are currently available every 15 minutes need to be reduced to once every 60 minutes to correspond with the predicted feature.

Why is this data set relevant for the problem?

From the information presented in [2], we see clearly

The electricity market brings supply and demand together.

The main element to control the market is the Price. We already have supply data i.e. energy supply data from different sources and demand data which is the consumption data. We also have the energy price for any give time period. If supply and demand are key factors which influence the price, we already have the relevant data to analyse using Supervised Machine Learning.

```
In [1]: # coding=utf-8
```

```
In [2]: # Import necessary libraries
        import source_data_helper as sc
        import pandas as pd
        import numpy as np
        import os
        import matplotlib.pyplot as plt
        from sklearn import preprocessing

        %matplotlib inline
```

Read CSV Data We start by reading data which have a frequency of 15 mins:

1. Actual generation
2. Actual consumption
3. Balancing energy

```
In [3]: cwd = os.getcwd()
        source_path = os.path.join(cwd, 'datasets', '2015-2017')

        # Read all csv data with a time series frequency of 15 mins. data_freq_15min is a list of dataframes
        data_freq_15min = sc.read_multiple_csv(source_path, ['DE_Actual generation_2015-2017.csv',
```

```
                                                'DE_Actual consumption_2015-2017.csv',
                                                'DE_Balancing energy_2015-2017.csv'])
```

```
dataset has 70271 samples with 14 features each.
dataset has 70271 samples with 3 features each.
dataset has 70271 samples with 4 features each.
```

Actual generation (Frequency = 15 mins)

```
In [4]: data_freq_15min[0].head()
```

```
Out[4]:          Date Time of day  Biomass[MWh]  Hydropower[MWh]  Wind offshore[MWh]  \
        0 2015-01-01    12:00 AM       1005.50           288.25              130.00
        1 2015-01-01    12:15 AM       1007.00           287.75              129.25
        2 2015-01-01    12:30 AM       1006.50           292.75              128.50
        3 2015-01-01    12:45 AM       1005.25           289.50              128.75
        4 2015-01-01     1:00 AM        998.75           295.25              128.75

           Wind onshore[MWh]  Photovoltaics[MWh]  Other renewable[MWh]  Nuclear[MWh]  \
        0            2028.25                 0.0                  14.5       2685.50
        1            2023.00                 0.0                  14.5       2646.25
        2            2040.25                 0.0                  14.5       2660.75
        3            2036.50                 0.0                  14.5       2718.00
        4            2045.75                 0.0                  14.5       2772.25

           Fossil brown coal[MWh]  Fossil hard coal[MWh]  Fossil gas[MWh]  \
        0                 3976.25                 686.00           263.00
        1                 3963.25                 721.25           261.75
        2                 3924.75                 695.75           260.50
        3                 3871.75                 664.75           241.50
        4                 3899.00                 520.50           202.25

           Hydro pumped storage[MWh]  Other conventional[MWh]
        0                     192.50                  1521.75
```

```
1                      149.75                  1498.00
2                      173.25                  1503.25
3                       95.00                  1518.75
4                       67.50                  1491.75
```

The amount of energy in MWh(Megawatt hour) from different energy sources for each time period is listed above

In [5]:
```
# Fix column names with Whitespaces and Uppercases
data_freq_15min[0].columns = data_freq_15min[0].columns.str.strip().str.lower().str.replace(' ', '_').
                                 str.replace('(', '').str.replace(')', '').str.replace('[', '_').str.replace(']', '')
data_freq_15min[0].head()
```

Out[5]:
```
         date time_of_day  biomass_mwh  hydropower_mwh  wind_offshore_mwh  \
0 2015-01-01    12:00 AM      1005.50          288.25             130.00
1 2015-01-01    12:15 AM      1007.00          287.75             129.25
2 2015-01-01    12:30 AM      1006.50          292.75             128.50
3 2015-01-01    12:45 AM      1005.25          289.50             128.75
4 2015-01-01     1:00 AM       998.75          295.25             128.75

   wind_onshore_mwh  photovoltaics_mwh  other_renewable_mwh  nuclear_mwh  \
0           2028.25                0.0                 14.5      2685.50
1           2023.00                0.0                 14.5      2646.25
2           2040.25                0.0                 14.5      2660.75
3           2036.50                0.0                 14.5      2718.00
4           2045.75                0.0                 14.5      2772.25

   fossil_brown_coal_mwh  fossil_hard_coal_mwh  fossil_gas_mwh  \
0                3976.25                686.00          263.00
1                3963.25                721.25          261.75
2                3924.75                695.75          260.50
3                3871.75                664.75          241.50
4                3899.00                520.50          202.25

   hydro_pumped_storage_mwh  other_conventional_mwh
0                    192.50                 1521.75
```

```
1                     149.75                      1498.00
2                     173.25                      1503.25
3                      95.00                      1518.75
4                      67.50                      1491.75
```

Actual consumption (Frequency = 15 mins)

```
In [6]: data_freq_15min[1].head()
```

```
Out[6]:         Date Time of day  Total[MWh]
        0 2015-01-01    12:00 AM    10606.25
        1 2015-01-01    12:15 AM    10505.25
        2 2015-01-01    12:30 AM    10517.00
        3 2015-01-01    12:45 AM    10468.50
        4 2015-01-01     1:00 AM    10307.50
```

The amount of energy consumed in MWh(Megawatt hour) for each time period is listed above

```
In [7]: # Fix column names with Whitespaces and Uppercases
        data_freq_15min[1].columns = data_freq_15min[1].columns.str.strip().str.lower().str.replace(' ', '_').
                                     str.replace('(', '').str.replace(')', '').str.replace('[', '_').str.replace(']', '')
```

Balancing energy (Frequency = 15 mins)

```
In [8]: data_freq_15min[2].head()
```

```
Out[8]:         Date Time of day  Balancing energy volume[MWh]  \
        0 2015-01-01    12:00 AM                        -475.0
        1 2015-01-01    12:15 AM                        -181.0
        2 2015-01-01    12:30 AM                         154.0
        3 2015-01-01    12:45 AM                         137.0
        4 2015-01-01     1:00 AM                         463.0

           Balancing energy price[Euro/MWh]
        0                            -49.41
        1                            -19.69
```

```
2                              74.70
3                              62.28
4                              63.71
```

```
In [9]: # Fix column names with Whitespaces and Uppercases
        data_freq_15min[2].columns = data_freq_15min[2].columns.str.strip().str.lower().str.replace(' ', '_').
                                    str.replace('(', '').str.replace(')', '').str.replace('[', '_').str.replace(']', '')
```

Wholesale market price (Frequency = 60 mins)

Wholesale market price/Day ahead price has a frequency of 60 mins. Therfore, we will read this data separately to merge them later with the 15 min data set, which will be reduced to a frequency of 60 minutes.

```
In [10]: prices_freq_60min = sc.read_csv(os.path.join(source_path,
                                              "DE_Day-ahead prices_2015-2017.csv"))
```

```
dataset has 17568 samples with 14 features each.
```



```
In [11]: prices_freq_60min.head()
```

```
Out[11]:         Date Time of day  Germany/Austria/Luxembourg[Euro/MWh]  \
         0 2015-01-01    12:00 AM                                   NaN
         1 2015-01-01     1:00 AM                                   NaN
         2 2015-01-01     2:00 AM                                   NaN
         3 2015-01-01     3:00 AM                                   NaN
         4 2015-01-01     4:00 AM                                   NaN

            Denmark 1[Euro/MWh]  Denmark 2[Euro/MWh]  France[Euro/MWh]  \
         0                25.02                27.38               NaN
         1                18.29                18.29               NaN
         2                16.04                16.04               NaN
         3                14.60                14.60               NaN
         4                14.95                14.95               NaN

            Northern Italy[Euro/MWh]  Netherlands[Euro/MWh]  Poland[Euro/MWh]  \
         0                      NaN                    NaN               NaN
```

```
1                        NaN                     NaN                 NaN
2                        NaN                     NaN                 NaN
3                        NaN                     NaN                 NaN
4                        NaN                     NaN                 NaN

   Sweden 4[Euro/MWh]  Switzerland[Euro/MWh]  Slovenia[Euro/MWh]  \
0               27.38                  44.94               27.30
1               23.37                  43.43               23.25
2               19.33                  38.08               22.20
3               17.66                  35.47               19.56
4               17.53                  30.83               18.88

   Czech Republic[Euro/MWh]  Hungary[Euro/MWh]
0                     26.48              45.07
1                     24.20              44.16
2                     22.06              39.17
3                     20.27              26.93
4                     19.17              20.94
```

```
In [12]: # Fix column names with Whitespaces and Uppercases
         prices_freq_60min.columns = prices_freq_60min.columns.str.strip().str.lower().str.replace(' ', '_').
                                         str.replace('(', '').str.replace(')', '').str.replace('[', '_').str.replace(']', '')
```

Reduce frequency of all features to 60 minutes and merge to a single data set

```
In [13]: data_freq_60min = sc.convert_multiple_to_hourly(data_freq_15min)
```

```
Modified dataset has 17568 samples with 14 features each.
Modified dataset has 17568 samples with 3 features each.
Modified dataset has 17568 samples with 4 features each.
```

```
In [14]: # view reduced data for actual generation
         data_freq_60min[0].head()
```

Out[14]:
```
         date time_of_day  biomass_mwh  hydropower_mwh  wind_offshore_mwh  \
0 2015-01-01    12:00 AM      1005.50          288.25             130.00
1 2015-01-01     1:00 AM       998.75          295.25             128.75
2 2015-01-01     2:00 AM      1001.00          293.25             129.00
3 2015-01-01     3:00 AM      1008.00          284.25             128.50
4 2015-01-01     4:00 AM      1008.75          279.25             129.75

   wind_onshore_mwh  photovoltaics_mwh  other_renewable_mwh  nuclear_mwh  \
0           2028.25                0.0                 14.5      2685.50
1           2045.75                0.0                 14.5      2772.25
2           2134.50                0.0                 14.5      2774.00
3           2149.50                0.0                 14.5      2759.25
4           2184.00                0.0                 14.5      2766.50

   fossil_brown_coal_mwh  fossil_hard_coal_mwh  fossil_gas_mwh  \
0                3976.25                686.00          263.00
1                3899.00                520.50          202.25
2                3774.50                449.25          101.00
3                3574.00                483.50          101.00
4                3540.25                469.50          101.25

   hydro_pumped_storage_mwh  other_conventional_mwh
0                    192.50                 1521.75
1                     67.50                 1491.75
2                    167.00                 1480.25
3                    136.50                 1537.00
4                    142.25                 1476.00
```


In [15]:
```
# join new dataframes list with the previous join to get master dataframe
data_freq_60min.extend([prices_freq_60min])
```

In [16]:
```
# Merge all dataframes in the list iteratively with keys= date, time_of_day
master_data = sc.join(data_freq_60min)
```

In [17]:
```
# Columns containing price data for other countries are not relevant for the current problem. Drop them
master_data.drop(['denmark_1_euro/mwh','denmark_2_euro/mwh', 'france_euro/mwh',
```

```
                           'northern_italy_euro/mwh', 'netherlands_euro/mwh', 'poland_euro/mwh',
                           'sweden_4_euro/mwh', 'switzerland_euro/mwh', 'slovenia_euro/mwh',
                           'czech_republic_euro/mwh', 'hungary_euro/mwh'], 1, inplace=True, errors='ignore')
```

In [18]:
```
# Rename columns for better readability
master_data.rename(columns={'total_mwh': 'total_consumption_mwh',
                            'germany/austria/luxembourg_euro/mwh': 'price_germany_euro/mwh'}
                               , inplace=True)

# Dataset characteristics
print("Number of instances in dataset = {}".format(master_data.shape[0]))
print("Total number of columns = {}".format(master_data.columns.shape[0]))
# List number of null values pro Feature
print("Column wise count of null values:-")
print(master_data.isnull().sum())
```

```
Number of instances in dataset = 17596
Total number of columns = 18
Column wise count of null values:-
date                              0
time_of_day                       0
biomass_mwh                     203
hydropower_mwh                  135
wind_offshore_mwh               113
wind_onshore_mwh                117
photovoltaics_mwh               146
other_renewable_mwh             212
nuclear_mwh                      96
fossil_brown_coal_mwh           196
fossil_hard_coal_mwh            168
fossil_gas_mwh                  150
hydro_pumped_storage_mwh        121
other_conventional_mwh         1017
total_consumption_mwh             0
balancing_energy_volume_mwh       0
```

```
balancing_energy_price_euro/mwh        0
price_germany_euro/mwh               120
dtype: int64
```

As we can see above, 12 features contain missing or NaN values. Let us explore the data in detail to determine how to deal with NaNs.

1.4.1 Data Exploration

We have 4 types of data and we have to explore them separately.

a. Actual generation - contains values of individual energy sources
b. Realized Consumption - total energy consumption (Germany)
c. Balancing energy - Energy required for balancing and price Euro/Mwh
d. Wholesale energy price - target variable

In [19]:
```
# columns for actual generation
actual_generation = ['biomass_mwh','hydropower_mwh','wind_offshore_mwh',
                     'wind_onshore_mwh','photovoltaics_mwh','other_renewable_mwh',
                     'nuclear_mwh','fossil_brown_coal_mwh','fossil_hard_coal_mwh',
                     'fossil_gas_mwh','hydro_pumped_storage_mwh', 'other_conventional_mwh']

#actual_consumption = ['50hz_consumption','amprion_consumption','tennet_consumption','transnetbw_consumption']
actual_consumption = ['total_consumption_mwh']

balancing_energy = ['balancing_energy_volume_mwh','balancing_energy_price_euro/mwh']

target = ['price_germany_euro/mwh']
```

In [20]:
```
master_data[actual_generation].describe()
```

Out[20]:
```
        biomass_mwh  hydropower_mwh  wind_offshore_mwh  wind_onshore_mwh  \
count  17393.000000    17461.000000       17483.000000      17479.000000
mean    1059.656126      453.039030         285.107447       1907.624707
std       89.666552      128.472593         231.897884       1606.903028
```

```
min      660.500000     202.250000          0.000000         28.000000
25%      986.250000     354.250000         85.750000        726.250000
50%     1080.250000     423.750000        210.250000       1412.250000
75%     1147.000000     530.750000        481.750000       2616.250000
max     1206.000000     785.250000        915.500000       7738.250000

       photovoltaics_mwh  other_renewable_mwh   nuclear_mwh  \
count       17450.000000         17384.000000  17500.000000
mean          985.667722            23.507248   2345.017114
std          1511.748379            15.589565    360.398637
min             0.000000             6.500000   1130.500000
25%             0.000000            16.250000   2156.250000
50%            17.250000            25.000000   2457.000000
75%          1579.000000            27.750000   2630.250000
max          6563.000000           518.250000   2868.500000

       fossil_brown_coal_mwh  fossil_hard_coal_mwh  fossil_gas_mwh  \
count           17400.000000          17428.000000    17446.000000
mean             3762.063707           2337.405899      362.094334
std               576.352412           1231.525012      311.543847
min              1332.500000            103.750000        7.250000
25%              3453.000000           1250.062500      153.312500
50%              3836.750000           2393.125000      235.125000
75%              4186.500000           3350.562500      455.187500
max              4807.500000           5169.500000     2364.750000

       hydro_pumped_storage_mwh  other_conventional_mwh
count              17475.000000            16579.000000
mean                 175.073462             1437.941764
std                  226.936886              609.139779
min                    0.000000              174.750000
25%                    1.250000             1021.750000
50%                   74.500000             1334.500000
75%                  275.000000             1716.750000
```

```
max                    1598.250000            8127.500000
```

Actual generation depicts real time energy generation data. If we have missing values here, it implies that the particular energy source did not produce energy for the time period. We can replace the missing values with 0s.

We also notice that the mean and median values of the different features vary to a large extent. Hence, in a later stage we should scale our data.

```
In [21]: # Fill missing values with zeros
         master_data['biomass_mwh'].fillna(0,inplace=True)
         master_data['hydropower_mwh'].fillna(0,inplace=True)
         master_data['wind_offshore_mwh'].fillna(0,inplace=True)
         master_data['wind_onshore_mwh'].fillna(0,inplace=True)
         master_data['photovoltaics_mwh'].fillna(0,inplace=True)
         master_data['other_renewable_mwh'].fillna(0,inplace=True)
         master_data['nuclear_mwh'].fillna(0,inplace=True)
         master_data['fossil_brown_coal_mwh'].fillna(0,inplace=True)
         master_data['fossil_hard_coal_mwh'].fillna(0,inplace=True)
         master_data['fossil_gas_mwh'].fillna(0,inplace=True)
         master_data['hydro_pumped_storage_mwh'].fillna(0,inplace=True)
         master_data['other_conventional_mwh'].fillna(0,inplace=True)
```

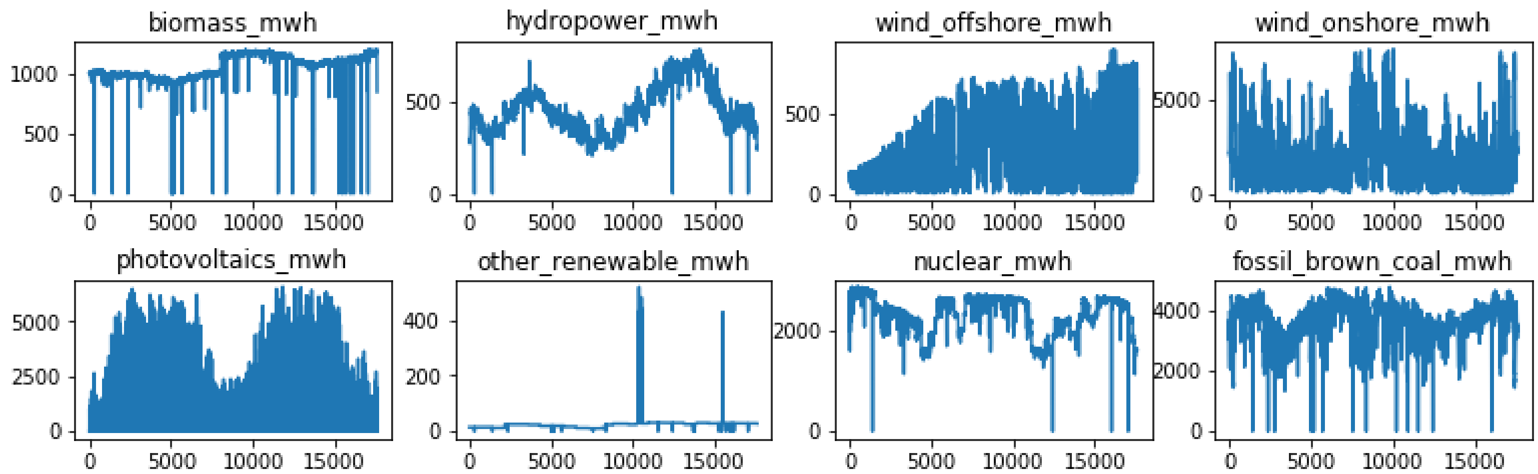

1.4.2 Data Visualization

1. Detecting yearly patterns As the data is spread over 2 years, we visualize the data to discover yearly patterns.

```
In [22]: # Plot actual generation
         fig = plt.figure(figsize=(13,8))
         fig.subplots_adjust(hspace=.5)

         plt.subplot(4,4,1)
         master_data['biomass_mwh'].plot()
         plt.title('biomass_mwh')

         plt.subplot(4,4,2)
         master_data['hydropower_mwh'].plot()
```

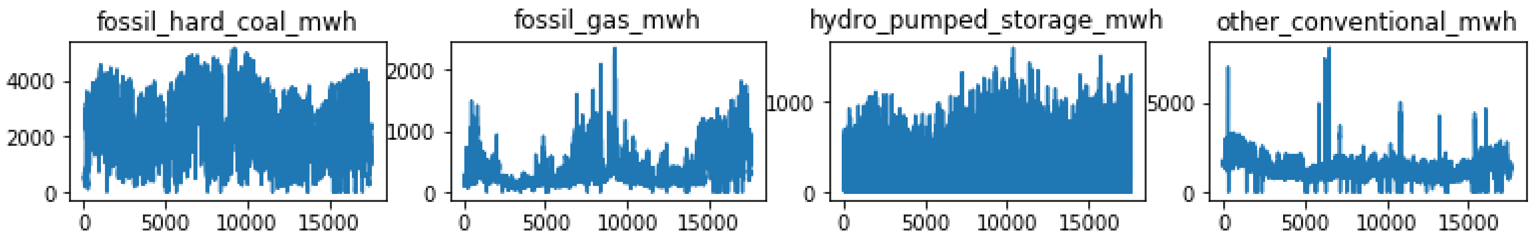
```
plt.title('hydropower_mwh')

plt.subplot(4,4,3)
master_data['wind_offshore_mwh'].plot()
plt.title('wind_offshore_mwh')

plt.subplot(4,4,4)
master_data['wind_onshore_mwh'].plot()
plt.title('wind_onshore_mwh')

plt.subplot(4,4,5)
master_data['photovoltaics_mwh'].plot()
plt.title('photovoltaics_mwh')

plt.subplot(4,4,6)
master_data['other_renewable_mwh'].plot()
plt.title('other_renewable_mwh')

plt.subplot(4,4,7)
master_data['nuclear_mwh'].plot()
plt.title('nuclear_mwh')

plt.subplot(4,4,8)
master_data['fossil_brown_coal_mwh'].plot()
plt.title('fossil_brown_coal_mwh')
```

Out[22]: Text(0.5,1,'fossil_brown_coal_mwh')




```
In [23]: fig = plt.figure(figsize=(13,8))
         fig.subplots_adjust(hspace=.5)

         plt.subplot(4,4,9)
         master_data['fossil_hard_coal_mwh'].plot()
         plt.title('fossil_hard_coal_mwh')

         plt.subplot(4,4,10)
         master_data['fossil_gas_mwh'].plot()
         plt.title('fossil_gas_mwh')

         plt.subplot(4,4,11)
         master_data['hydro_pumped_storage_mwh'].plot()
         plt.title('hydro_pumped_storage_mwh')

         plt.subplot(4,4,12)
         master_data['other_conventional_mwh'].plot()
         plt.title('other_conventional_mwh')
```

Out[23]: Text(0.5,1,'other_conventional_mwh')



We currently have 17596 hourly dataset rows over a persiod of 2 years 2015-2016 and 2016-2017 i.e. 8798 per year.

Looking at the above data we can observe that non-renewable sources of energy have a pattern which can be predicted if we were to divide the graphs right in the middle. This is true for nuclear, fossil brown coal, fossil hard coal, fossil gas, hydro pumped storage and other conventional energy sources.

The same is not consistent for renewable sources. Wind energy is weather dependent which does not have set patterns. If we observe biomass and hydropower, they reflect similar patterns but in varying sizes. This could be a result of increasing yearly investments in these energy sources. Photovoltaics appears to be consistent with a predictable pattern.

```
In [24]: master_data[actual_consumption].describe()
```

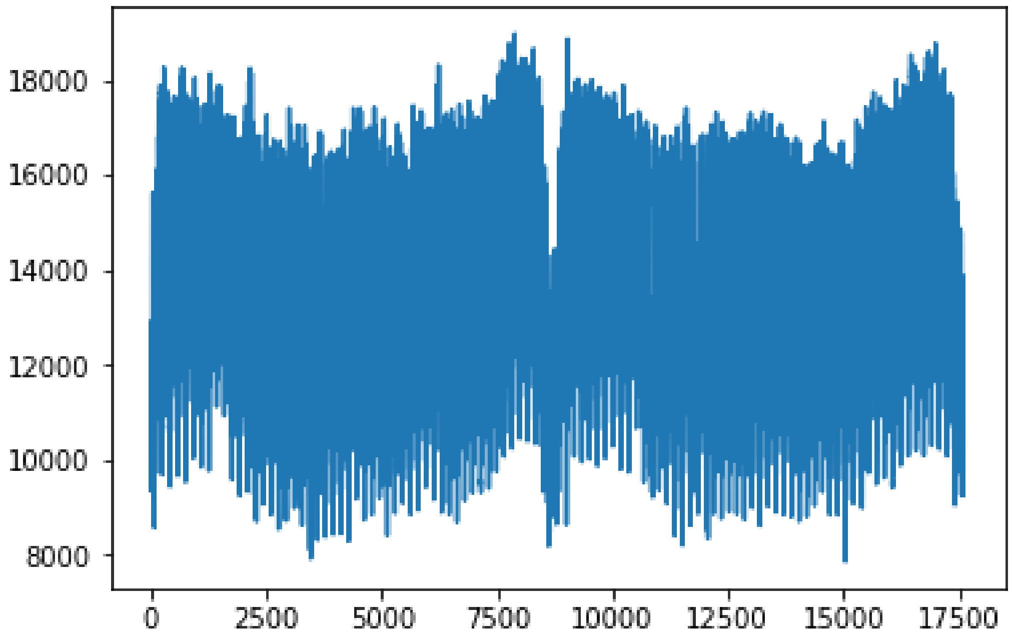
Out[24]:

	total_consumption_mwh
count	17596.000000
mean	13663.896255
std	2477.577389
min	7856.750000
25%	11586.750000
50%	13582.500000
75%	15906.437500
max	18975.250000

The mean and median of 'total_consumption_mwh' varies to a large extent when compared to features grouped under 'actual_generation'. Scaling is therefore necessary.

```
In [25]: master_data['total_consumption_mwh'].plot()
```

Out[25]: <matplotlib.axes._subplots.AxesSubplot at 0xa8efb70>



The consumption data for the second year seems to follow a similar pattern as the first year. This implies that the energy consumption for a certain time of year is consistent to the previous year.

In [26]: master_data[balancing_energy].describe()

Out[26]:

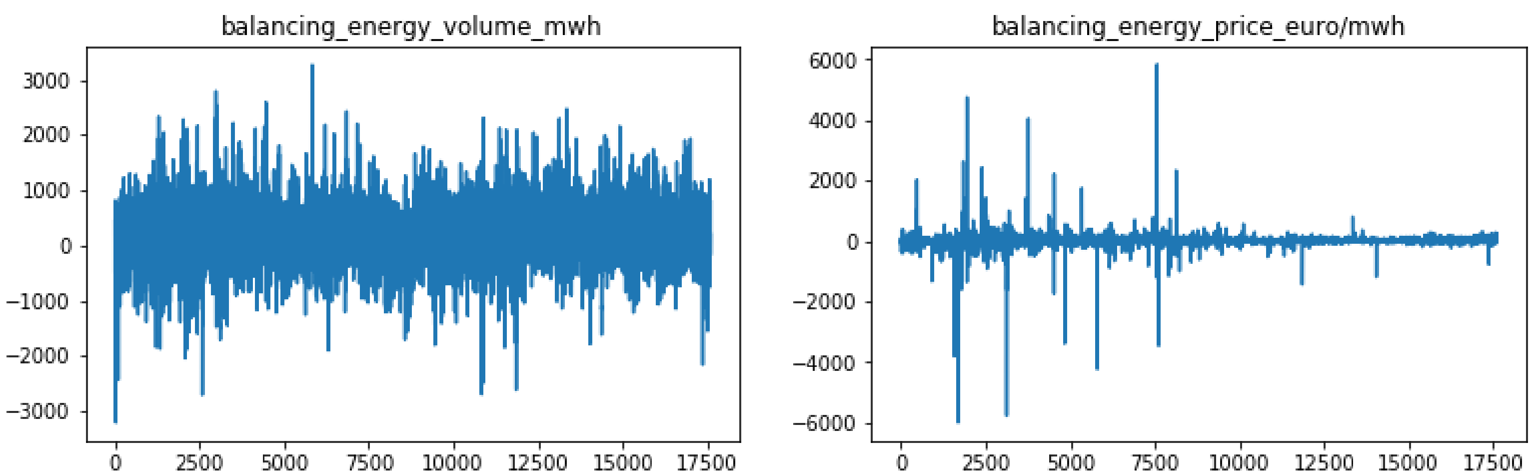
	balancing_energy_volume_mwh	balancing_energy_price_euro/mwh
count	17596.000000	17596.000000
mean	165.372471	31.917856
std	464.695247	150.874693
min	-3210.000000	-5997.420000
25%	-102.250000	1.890000

50%	160.000000	39.940000
75%	437.000000	60.882500
max	3271.000000	5824.630000

Similar to 'total_consumption_mwh', Data Scaling is also required.

In [27]:
```
fig = plt.figure(figsize=(13,8))

plt.subplot(2,2,1)
master_data['balancing_energy_volume_mwh'].plot()
plt.title('balancing_energy_volume_mwh')

plt.subplot(2,2,2)
master_data['balancing_energy_price_euro/mwh'].plot()
plt.title('balancing_energy_price_euro/mwh')
```

Out[27]:
```
Text(0.5,1,'balancing_energy_price_euro/mwh')
```



The balancing energy volume contains a consistent yearly pattern. This does not apply to the balancing energy price which appears more random.

```
In [28]: master_data[target].describe()
```

```
Out[28]:        price_germany_euro/mwh
         count             17476.000000
         mean                 30.386641
         std                  12.551637
         min                -130.090000
         25%                  23.460000
         50%                  29.660000
         75%                  37.060000
         max                 104.960000
```

We still have 120 missing values for **price_germany**. The reason for missing values could be an error in the logging system. In other cases if we look at the data above, we notice negative values. In certain rare periods particularly at the end of the year, there is a such a surplus of wind energy produced that the prices go below zero i.e the consumer gets paid for consuming energy. In the energy market, there is always supply and demand and hence, a price for each time period. Earlier for energy generation we replaced all missing values with zeroes. The case here is different and a price of zero has a false implication and would have a negative influence on the prediction.

The best strategy here would be to either use the Median or the mean.

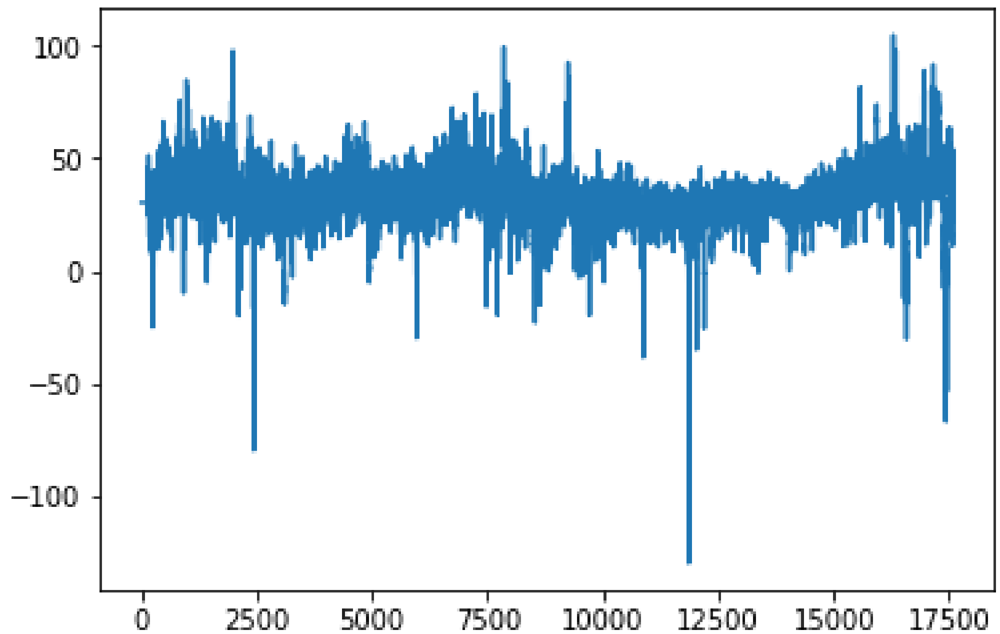
Median = 29.66 Mean = 30.386641

Since there isn't a large difference between Median and Mean, let us consider the Mean value to fill the missing values

```
In [29]: master_data['price_germany_euro/mwh'].fillna(master_data['price_germany_euro/mwh']
                                                      .mean(),inplace=True)
```

```
In [30]: master_data['price_germany_euro/mwh'].plot()
```

```
Out[30]: <matplotlib.axes._subplots.AxesSubplot at 0xa94cf28>
```



From the above figure we can observe a pattern with a lot of activity at start and end of the year. As the year proceeds, it appears more constant. The pattern cannot be reproduced the next year 100% but the tendency and value ranges remain the same. Moreover, pricing is a complex issue which not only depends on demand and supply but other factors which we haven't considered in our case. One factor could be the increase of investment every year in renewable energies increasing output but also at the same time making renewable energy expensive in the initial phases.

Since the machine learning models cannot handle timeseries data, we will delete time related data like 'date' and 'time_of_day', however perserving the order of recorded data as the order of data is essential for training a supervised regression model for time series.

```
In [31]: # Drop date and time
         master_data.drop(['date','time_of_day'], axis=1, inplace=True)
```

Our final list of features can be seen below.

```
In [32]: list(master_data)
```

```
Out[32]: ['biomass_mwh',
          'hydropower_mwh',
          'wind_offshore_mwh',
          'wind_onshore_mwh',
          'photovoltaics_mwh',
          'other_renewable_mwh',
          'nuclear_mwh',
          'fossil_brown_coal_mwh',
          'fossil_hard_coal_mwh',
          'fossil_gas_mwh',
          'hydro_pumped_storage_mwh',
          'other_conventional_mwh',
          'total_consumption_mwh',
          'balancing_energy_volume_mwh',
          'balancing_energy_price_euro/mwh',
          'price_germany_euro/mwh']
```

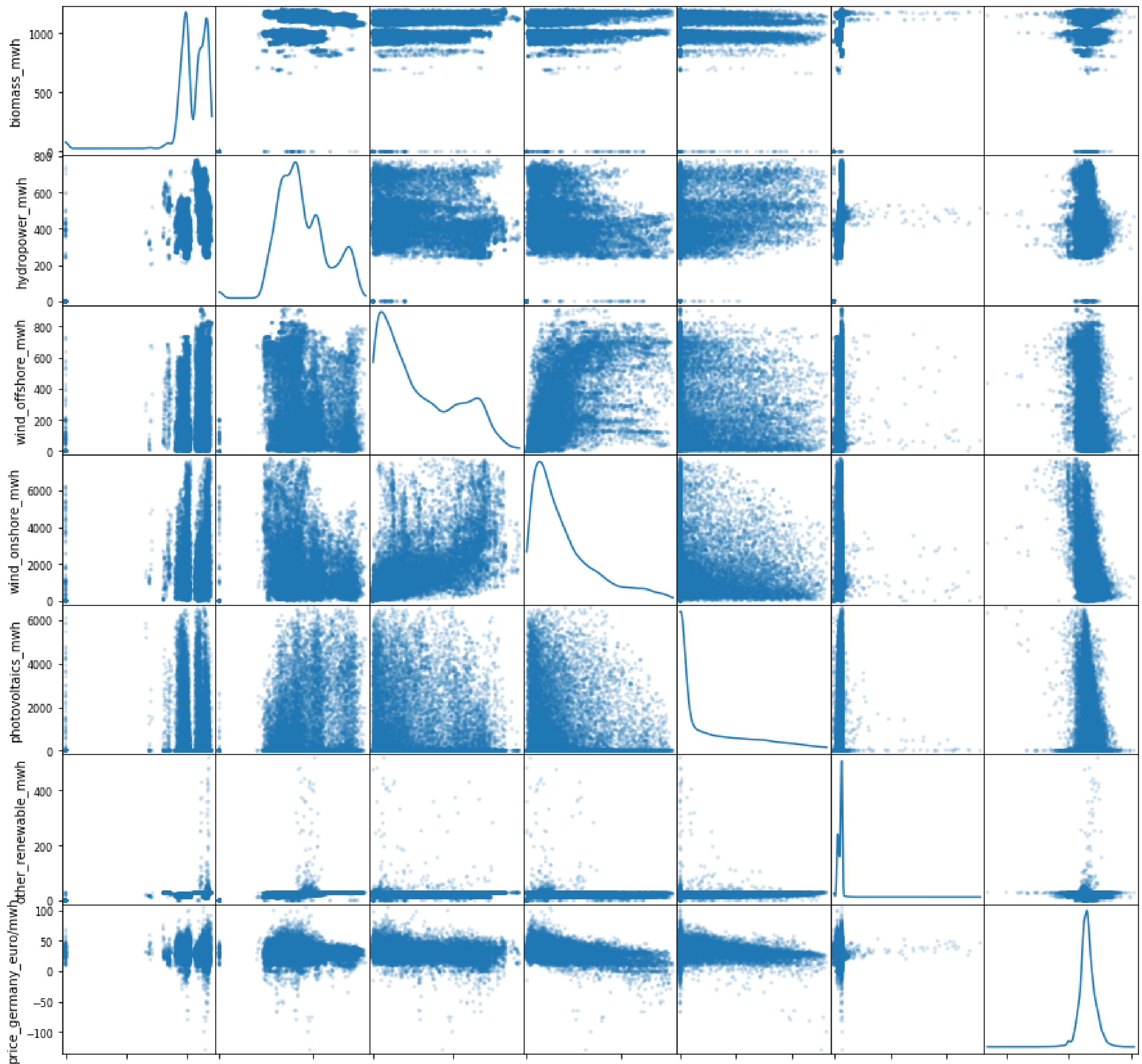

2. Linear correlation among input features and target variable In the scatter plots below, we attempt to find linear correlations between the input features and the target variable.

```
In [33]: from pandas.plotting import scatter_matrix
         scatter_matrix(master_data[['biomass_mwh','hydropower_mwh','wind_offshore_mwh',
                                     'wind_onshore_mwh','photovoltaics_mwh','other_renewable_mwh'
                                     ,'price_germany_euro/mwh']], alpha=0.2, figsize=(15,15), diagonal='kde')
```

```
Out[33]: array([[<matplotlib.axes._subplots.AxesSubplot object at 0x000000000ACE81D0>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x000000000AD18940>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x000000000AD3EF28>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x000000000AD6B5F8>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x000000000AD92C88>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x000000000AD92CC0>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x000000000ADEC9E8>],
                [<matplotlib.axes._subplots.AxesSubplot object at 0x000000000AE1F0B8>,
```

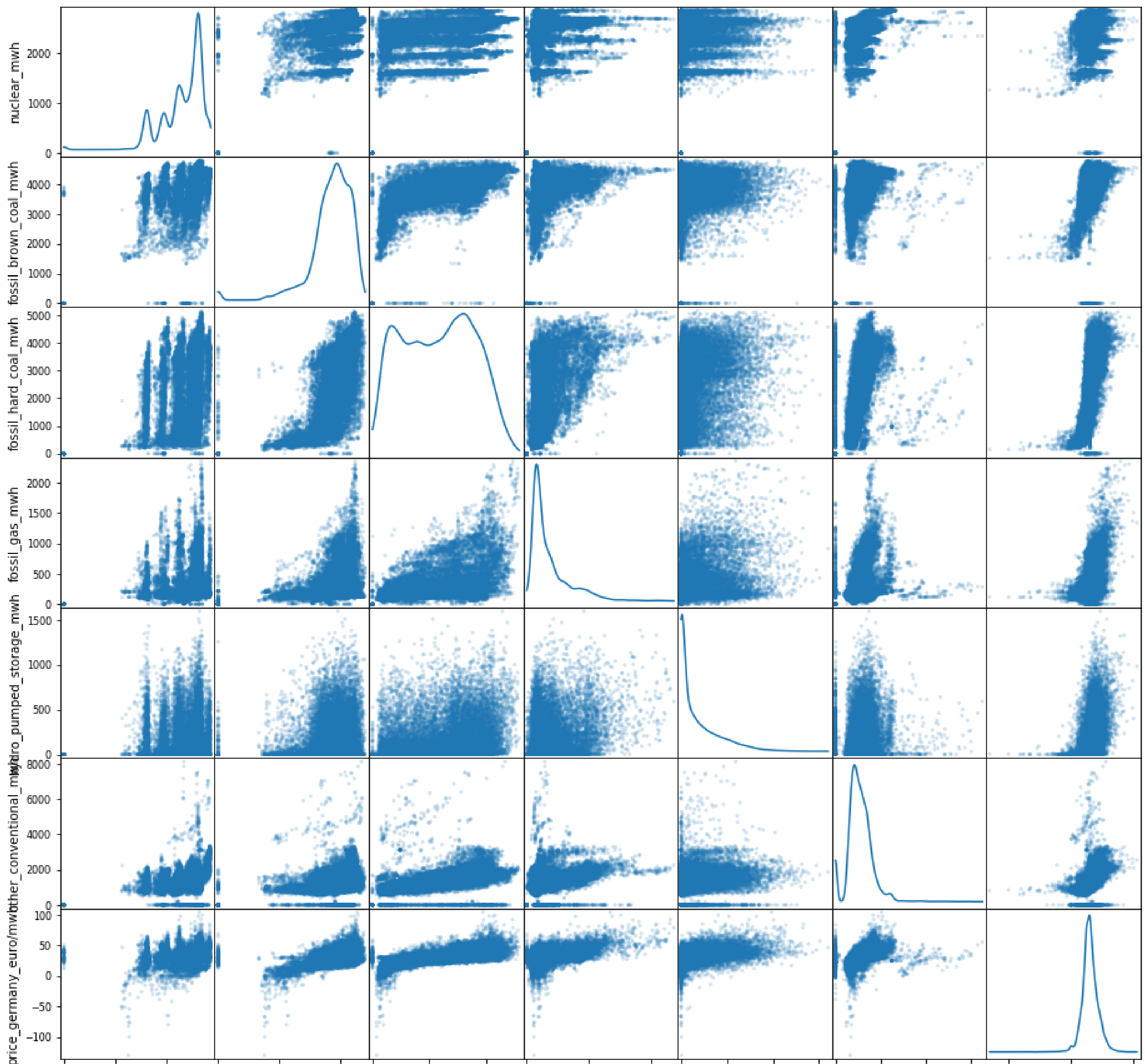
```
 <matplotlib.axes._subplots.AxesSubplot object at 0x000000000AE46748>,
 <matplotlib.axes._subplots.AxesSubplot object at 0x000000000AE6FDD8>,
 <matplotlib.axes._subplots.AxesSubplot object at 0x000000000B1804A8>,
 <matplotlib.axes._subplots.AxesSubplot object at 0x000000000B1A9B38>,
 <matplotlib.axes._subplots.AxesSubplot object at 0x000000000B1DB208>,
 <matplotlib.axes._subplots.AxesSubplot object at 0x000000000B204898>],
[<matplotlib.axes._subplots.AxesSubplot object at 0x000000000B22AF28>,
 <matplotlib.axes._subplots.AxesSubplot object at 0x000000000B25D5F8>,
 <matplotlib.axes._subplots.AxesSubplot object at 0x000000000B285C88>,
 <matplotlib.axes._subplots.AxesSubplot object at 0x000000000B2B5358>,
 <matplotlib.axes._subplots.AxesSubplot object at 0x000000000B2DE9E8>,
 <matplotlib.axes._subplots.AxesSubplot object at 0x000000000B3100B8>,
 <matplotlib.axes._subplots.AxesSubplot object at 0x000000000B337748>],
[<matplotlib.axes._subplots.AxesSubplot object at 0x000000000B3D1DD8>,
 <matplotlib.axes._subplots.AxesSubplot object at 0x000000000B4044A8>,
 <matplotlib.axes._subplots.AxesSubplot object at 0x000000000B42AB38>,
 <matplotlib.axes._subplots.AxesSubplot object at 0x000000000C56F208>,
 <matplotlib.axes._subplots.AxesSubplot object at 0x000000000C703898>,
 <matplotlib.axes._subplots.AxesSubplot object at 0x000000000C72EF28>,
 <matplotlib.axes._subplots.AxesSubplot object at 0x000000000CBFF5F8>],
[<matplotlib.axes._subplots.AxesSubplot object at 0x000000000CC26C88>,
 <matplotlib.axes._subplots.AxesSubplot object at 0x000000000CCB9358>,
 <matplotlib.axes._subplots.AxesSubplot object at 0x000000000CCE09E8>,
 <matplotlib.axes._subplots.AxesSubplot object at 0x000000000CD120B8>,
 <matplotlib.axes._subplots.AxesSubplot object at 0x000000000D229748>,
 <matplotlib.axes._subplots.AxesSubplot object at 0x000000000D252DD8>,
 <matplotlib.axes._subplots.AxesSubplot object at 0x000000000D2854A8>],
[<matplotlib.axes._subplots.AxesSubplot object at 0x000000000D2ADB38>,
 <matplotlib.axes._subplots.AxesSubplot object at 0x000000000D2DF208>,
 <matplotlib.axes._subplots.AxesSubplot object at 0x000000000D308898>,
 <matplotlib.axes._subplots.AxesSubplot object at 0x000000000D37FF28>,
 <matplotlib.axes._subplots.AxesSubplot object at 0x000000000D3AF5F8>,
 <matplotlib.axes._subplots.AxesSubplot object at 0x000000000D41AC88>,
 <matplotlib.axes._subplots.AxesSubplot object at 0x000000000D44B358>],
```

```
 [<matplotlib.axes._subplots.AxesSubplot object at 0x000000000D4729E8>,
  <matplotlib.axes._subplots.AxesSubplot object at 0x000000000D4A40B8>,
  <matplotlib.axes._subplots.AxesSubplot object at 0x000000000D4CE748>,
  <matplotlib.axes._subplots.AxesSubplot object at 0x000000000D4F3DD8>,
  <matplotlib.axes._subplots.AxesSubplot object at 0x000000000D5264A8>,
  <matplotlib.axes._subplots.AxesSubplot object at 0x000000000D54FB38>,
  <matplotlib.axes._subplots.AxesSubplot object at 0x000000000D582208>]],
dtype=object)
```

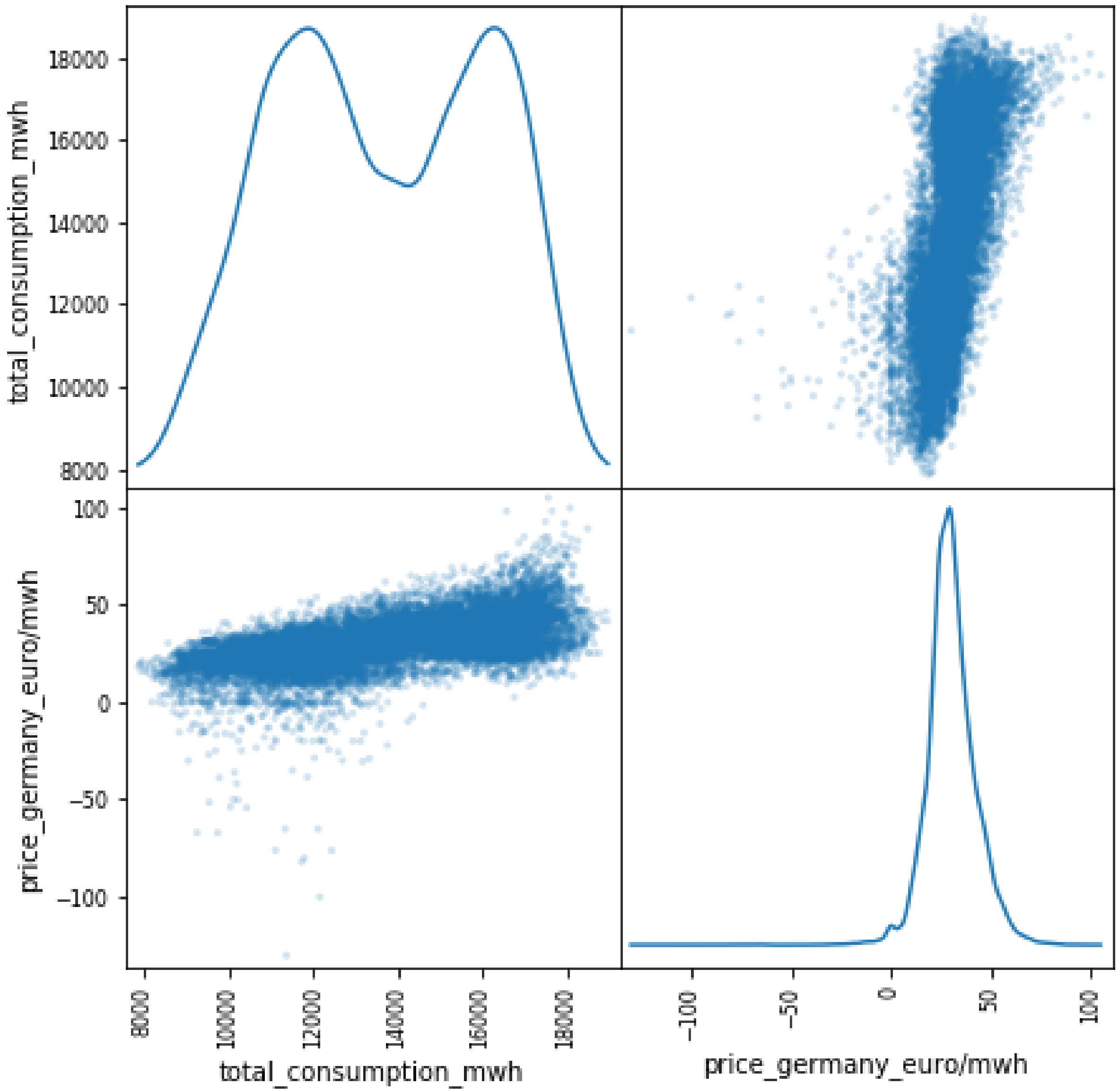

biomass_mwh
hydropower_mwh
wind_offshore_mwh
wind_onshore_mwh
photovoltaics_mwh
other_renewable_mwh
price_germany_euro/mwh
0
500
1000
800
600
400
200
6000
4000
2000
100
50
−50
−100

```
In [34]: scatter_matrix(master_data[['nuclear_mwh','fossil_brown_coal_mwh','fossil_hard_coal_mwh',
                                     'fossil_gas_mwh','hydro_pumped_storage_mwh', 'other_conventional_mwh',
                                            'price_germany_euro/mwh']], alpha=0.2, figsize=(15.5,15.5), diagonal='kde')

Out[34]: array([[<matplotlib.axes._subplots.AxesSubplot object at 0x000000000CAD04E0>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x000000000CABB5C0>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x000000000CAF4A90>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x000000000CB180F0>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x000000000CB3F780>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x000000000CB3F7B8>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x000000000CB984E0>],
                [<matplotlib.axes._subplots.AxesSubplot object at 0x000000000DA6EB38>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x000000000DAA1208>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x000000000DAC9898>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x000000000DAF3F28>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x000000000DB235F8>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x000000000DB4CC88>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x000000000DFAE358>],
                [<matplotlib.axes._subplots.AxesSubplot object at 0x000000000DFD59E8>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x000000000E0070B8>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x000000000E02F748>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x000000000E058DD8>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x000000000E08A4A8>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x000000000E0B0B38>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x000000000E526208>],
                [<matplotlib.axes._subplots.AxesSubplot object at 0x000000000E54A898>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x000000000E574F28>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x000000000E5A75F8>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x000000000E5CFC88>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x000000000E600358>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x000000000E62C9E8>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x000000000E65C0B8>],
```

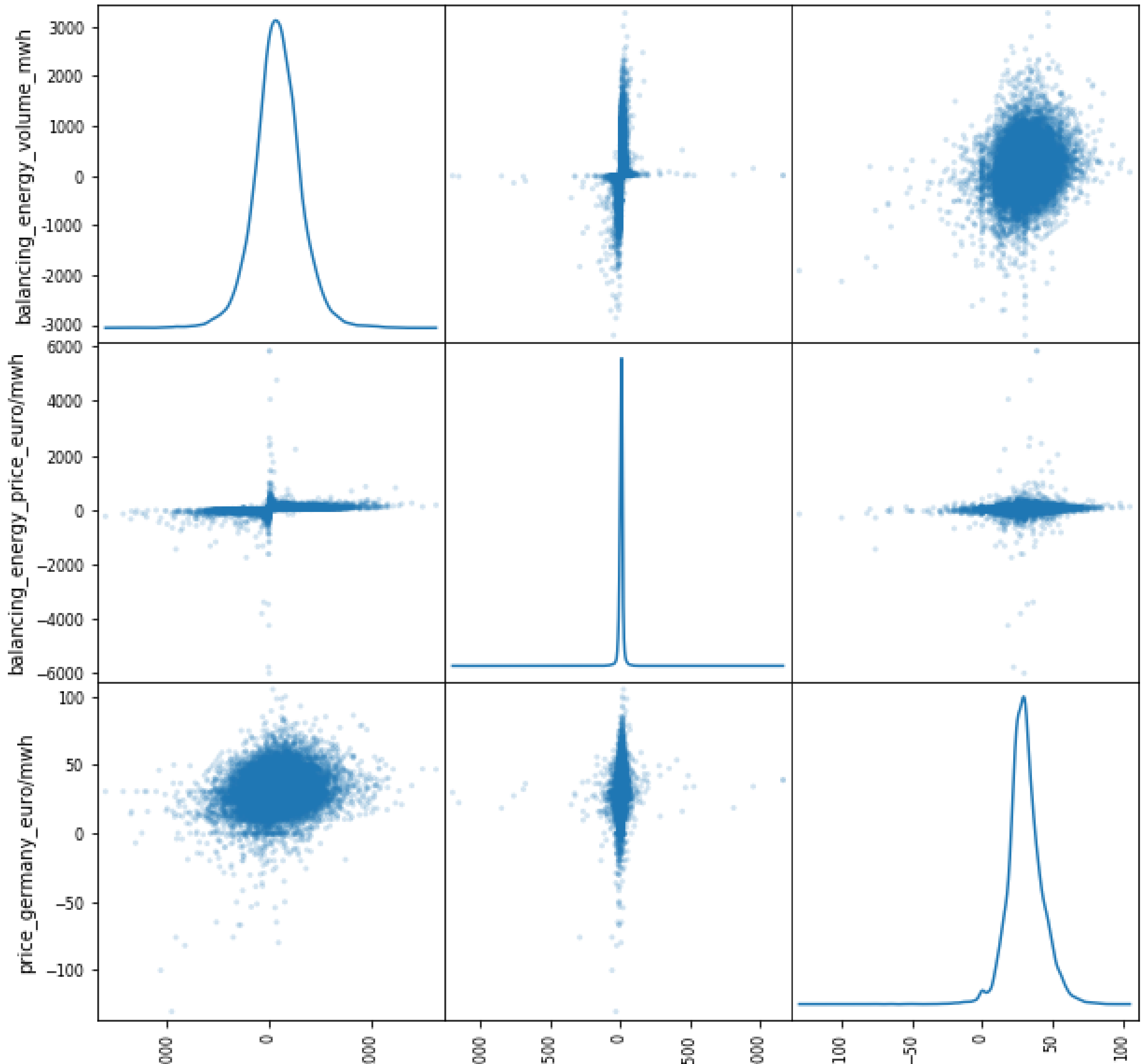
```
 [<matplotlib.axes._subplots.AxesSubplot object at 0x000000000E683748>,
  <matplotlib.axes._subplots.AxesSubplot object at 0x000000000E6AADD8>,
  <matplotlib.axes._subplots.AxesSubplot object at 0x000000000E8FC4A8>,
  <matplotlib.axes._subplots.AxesSubplot object at 0x000000000E925B38>,
  <matplotlib.axes._subplots.AxesSubplot object at 0x000000000E955208>,
  <matplotlib.axes._subplots.AxesSubplot object at 0x000000000ECAE898>,
  <matplotlib.axes._subplots.AxesSubplot object at 0x000000000ECD5F28>],
 [<matplotlib.axes._subplots.AxesSubplot object at 0x000000000ED085F8>,
  <matplotlib.axes._subplots.AxesSubplot object at 0x000000000ED32C88>,
  <matplotlib.axes._subplots.AxesSubplot object at 0x000000000ED63358>,
  <matplotlib.axes._subplots.AxesSubplot object at 0x000000000ED8A9E8>,
  <matplotlib.axes._subplots.AxesSubplot object at 0x000000000EDBB0B8>,
  <matplotlib.axes._subplots.AxesSubplot object at 0x000000000EDE3748>,
  <matplotlib.axes._subplots.AxesSubplot object at 0x000000000EE0BDD8>],
 [<matplotlib.axes._subplots.AxesSubplot object at 0x000000000EE404A8>,
  <matplotlib.axes._subplots.AxesSubplot object at 0x000000000FE67B38>,
  <matplotlib.axes._subplots.AxesSubplot object at 0x000000000FE98208>,
  <matplotlib.axes._subplots.AxesSubplot object at 0x000000000FEC1898>,
  <matplotlib.axes._subplots.AxesSubplot object at 0x000000000FEE9F28>,
  <matplotlib.axes._subplots.AxesSubplot object at 0x000000000FF1A5F8>,
  <matplotlib.axes._subplots.AxesSubplot object at 0x000000000FF44C88>]],
dtype=object)
```


nuclear_mwh
fossil_brown_coal_mwh
fossil_hard_coal_mwh
fossil_gas_mwh
hydro_pumped_storage_mwh
other_conventional_mwh
price_germany_euro/mwh
0
1000
2000
3000
4000
5000
500
1500
2000
6000
8000
100
50
−50
−100

```
In [35]: scatter_matrix(master_data[['total_consumption_mwh','price_germany_euro/mwh']],
                        alpha=0.2, figsize=(7,7), diagonal='kde')

Out[35]: array([[<matplotlib.axes._subplots.AxesSubplot object at 0x00000000106F5860>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x00000000107BF860>],
                [<matplotlib.axes._subplots.AxesSubplot object at 0x0000000010B2F7B8>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x0000000010B50AC8>]],
               dtype=object)
```


total_consumption_mwh
18000
16000
14000
12000
10000
8000
price_germany_euro/mwh
100
50
0
−50
−100
8000
100000
120000
140000
160000
180000
total_consumption_mwh
−100
−50
0
50
100
price_germany_euro/mwh

```
In [36]: scatter_matrix(master_data[['balancing_energy_volume_mwh','balancing_energy_price_euro/mwh',
                                     'price_germany_euro/mwh']], alpha=0.2, figsize=(10,10), diagonal='kde')

Out[36]: array([[<matplotlib.axes._subplots.AxesSubplot object at 0x0000000010BCBB70>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x0000000010CB67B8>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x0000000010CDEE10>],
                [<matplotlib.axes._subplots.AxesSubplot object at 0x0000000010D104E0>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x0000000010D38B70>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x0000000010D38BA8>],
                [<matplotlib.axes._subplots.AxesSubplot object at 0x0000000010D918D0>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x0000000010DBBF60>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x0000000010DEA630>]],
               dtype=object)
```


balancing_energy_volume_mwh
balancing_energy_price_euro/mwh
price_germany_euro/mwh

As we can observe we cannot identify a direct linear correlation between the input features and the target variable.

1.4.3 Outlier Detection and Elimination

Using Tukey's method of Outlier detection [], we look for datasets which are 1.5 times below the 1st quartile (25th percentile) and 1.5 times above the 3rd quartile (75th percentile) and delete them from the master dataset.

```
In [37]: temp_frame = master_data.copy()
         # get only source features for scaling
         x_features = ['biomass_mwh','hydropower_mwh','wind_offshore_mwh','wind_onshore_mwh',
                       'photovoltaics_mwh','other_renewable_mwh','nuclear_mwh',
                       'fossil_brown_coal_mwh','fossil_hard_coal_mwh','fossil_gas_mwh',
                       'hydro_pumped_storage_mwh','other_conventional_mwh','total_consumption_mwh',
                       'balancing_energy_volume_mwh']
```

```
In [38]: from collections import Counter

         out_counter = Counter()


         # For each feature find the data points with extreme high or low values
         for index,feature in temp_frame[x_features].T.iterrows():

             # TODO: Calculate Q1 (25th percentile of the data) for the given feature
             Q1 = np.percentile(feature,25)

             # TODO: Calculate Q3 (75th percentile of the data) for the given feature
             Q3 = np.percentile(feature,75)

             # TODO: Use the interquartile range to calculate an outlier step (1.5 times the interquartile range)
             step = (Q3 - Q1)*1.5

             # Display the outliers
```

```
            new_data = temp_frame[~((temp_frame[index] >= Q1 - step)
                                    & (temp_frame[index] <= Q3 + step))]

            for index in new_data.index.get_values():
                #print(index)
                out_counter[index] += 1

        # OPTIONAL: Select the indices for data points you wish to remove
        outliers  = [list(out_counter.elements())]

        print (f'Detected {len(list(set(out_counter.elements())))} Outliers')
```

```
Detected 5326 Outliers
```

```
In [39]: # Remove the outliers, if any were specified
         temp_frame.drop(temp_frame.index[outliers], axis=0, inplace=True)
         print(f'{len(temp_frame)} data sets after outlier removal')
```

```
12270 data sets after outlier removal
```

1.4.4 Feature Scaling

From our data analysis we observed that each of the features are on a different scale. There are significant differences in the mean values of each of the individual features. We cannot use this data to train the machine learning model without scaling them. In our case, we will use the MinMaxScaler to scale all the features as some features also include negative values. The values will be scaled between -1 and 1 [5].

```
In [40]: # Drop 'balancing_energy_price_euro/mwh' based on domain knowledge as this feature is not required
         y = temp_frame['price_germany_euro/mwh']
         X = temp_frame.drop(['price_germany_euro/mwh',
                              'balancing_energy_price_euro/mwh'], axis=1)

         # Use MinMaxScaler to scale
```

```
scaler = preprocessing.MinMaxScaler()

X = scaler.fit_transform(X)
X = pd.DataFrame(X, columns=['biomass_mwh','hydropower_mwh','wind_offshore_mwh',
                             'wind_onshore_mwh','photovoltaics_mwh','other_renewable_mwh',
                             'nuclear_mwh','fossil_brown_coal_mwh','fossil_hard_coal_mwh',
                             'fossil_gas_mwh','hydro_pumped_storage_mwh','other_conventional_mwh',
                             'total_consumption_mwh','balancing_energy_volume_mwh'])
```

1.4.5 Avoiding Look ahead bias and cross validation for the energy Time Series data set

The available data set is a time series data set on an hourly basis. Regular cross validation methods like kfold, holdout are not appropriate due to the look ahead bias they generate. Considering the current time 't', we are predicting data for a future time period 't + n', where x is the difference in time to reach the target time for our prediction. Here it is important to not use future data for out training set. We need a special function 'TimeSeriesSplit' to generate training and testing data sets without look forward bias. 'TimeSeriesSplit' ensures that the future data is always used as the test set for prediction to simulate learning under real conditions.

1.5 Benchmark Model

Since energy price prediction is a classic regression problem, we will start with Linear Regression as our benchmark model.

```
In [41]: # import relevant libraries
         from sklearn.model_selection import TimeSeriesSplit
         from sklearn import metrics, preprocessing
         from sklearn.preprocessing import MinMaxScaler
         import time

In [42]: from sklearn.linear_model import LinearRegression, Ridge, Lasso
         # Perform Linear Regression using TimeSeriesSplit
         l_tscv = TimeSeriesSplit(n_splits=200)
         l_cv = l_tscv.split(X)

         l_reg = LinearRegression()
         train_list = []
         test_list = []
```

```python
    rmse_list = []

    # set start time
    start = time.time()

    # generate train and test indices and train each set
    for train_index, test_index in l_cv:
        #print("TRAIN:", train_index, "TEST:", test_index)
        X_train, X_test = X.iloc[train_index], X.iloc[test_index]
        y_train, y_test = y.iloc[train_index], y.iloc[test_index]
        l_reg.fit(X_train,y_train)
        y_pred = l_reg.predict(X_test)
        train_score = l_reg.score(X_train,y_train)
        test_score = l_reg.score(X_test,y_test)
        train_list.append(train_score)
        test_list.append(test_score)
        rmse = np.sqrt(metrics.mean_squared_error(y_test, y_pred))
        rmse_list.append(rmse)

    # set end time
    end = time.time()

    print("Average RMSE Score = ",np.mean(rmse_list))
    print("Average Train Prediction Accuracy =",np.mean(train_list))
    print("Average Test Prediction Accuracy =",np.mean(test_list))
    print("Execution time: ",end - start)
```

```
Average RMSE Score =  4.953005879681328
Average Train Prediction Accuracy = 0.758279100701416
Average Test Prediction Accuracy = 0.4243483359932581
Execution time:  0.7720000743865967
```

1.5.1 Considered Regression Models for prediction

Regularized versions of Linear Regression

1) Ridge
2) Lasso

Ensemble models

3) Random Forest Regression
4) Gradient Boosting Regression

Tree Based Gradient Boosting

5) Light GBM Regression

Gradient Boosted Decision Tree

6) XGBoost Regression

1.5.2 Build Data Pipeline

```
In [43]: from sklearn.model_selection import GridSearchCV
         from sklearn.ensemble import GradientBoostingRegressor, RandomForestRegressor
         from sklearn.svm import SVR
         from sklearn.tree import DecisionTreeRegressor
         from sklearn.ensemble import GradientBoostingRegressor
         from sklearn import preprocessing
         import lightgbm as lgb
         from lightgbm.sklearn import LGBMRegressor
         from xgboost.sklearn import XGBRegressor
         import xgboost as xgb
         from matplotlib import pyplot
```

```
In [44]: # result lists
```

```python
# simple pipeline function to run mutiple regression models sequentially
def run_pipeline(estimators, splits, X, y):
    perf_summary = []
    for est in estimators:
        tscv = TimeSeriesSplit(n_splits = splits)
        cv = tscv.split(X)
        # Initialize regressor
        perf = execute_regressor(est,X,y,cv)
        #print(perf)
        perf_summary.append(perf)
    return perf_summary

# function to execute a specific regression model provided earlier in the pipeline
def execute_regressor(reg, X, y, cv, **kwargs):
    reg_perf = {}
    rmse_l = []
    train_l = []
    test_l = []

    # record start
    start = time.time()

    regressor = reg(**kwargs)

    for train_index, test_index in cv:

        X_train, X_test = X.iloc[train_index], X.iloc[test_index]
        y_train, y_test = y.iloc[train_index], y.iloc[test_index]
        regressor.fit(X_train,y_train)
        y_pred = regressor.predict(X_test)
        train_score = regressor.score(X_train,y_train)
        test_score = regressor.score(X_test,y_test)
        train_l.append(train_score)
        test_l.append(test_score)
```

```
            rmse = np.sqrt(metrics.mean_squared_error(y_test, y_pred))
            rmse_list.append(rmse)

        #record end time
        end = time.time()

        reg_perf["name"] = reg.__name__
        reg_perf["train_time"] = end - start
        reg_perf["train_score"] = np.mean(train_l)
        reg_perf["test_score"] = np.mean(test_l)
        reg_perf["rmse"] = np.mean(rmse_list)
        return reg_perf
```

In [45]:
```
# define regression functions
estimators = [Ridge, Lasso, RandomForestRegressor, SVR,
              GradientBoostingRegressor, LGBMRegressor, XGBRegressor]
# Run pipeline and get performance of regressors
perf_list = run_pipeline(estimators, 70, X, y)
```

In [46]:
```
# Construct dataframe from dictionary list
model_summary = pd.DataFrame.from_dict(perf_list)
model_summary.rename(index={0:'Ridge',1:'Lasso', 2:'Random Forest',
                            3:'SVR', 4:'Gradient Boosting', 5:'LGBM',
                            6:'XGBoost'}, inplace=True)
model_summary
```

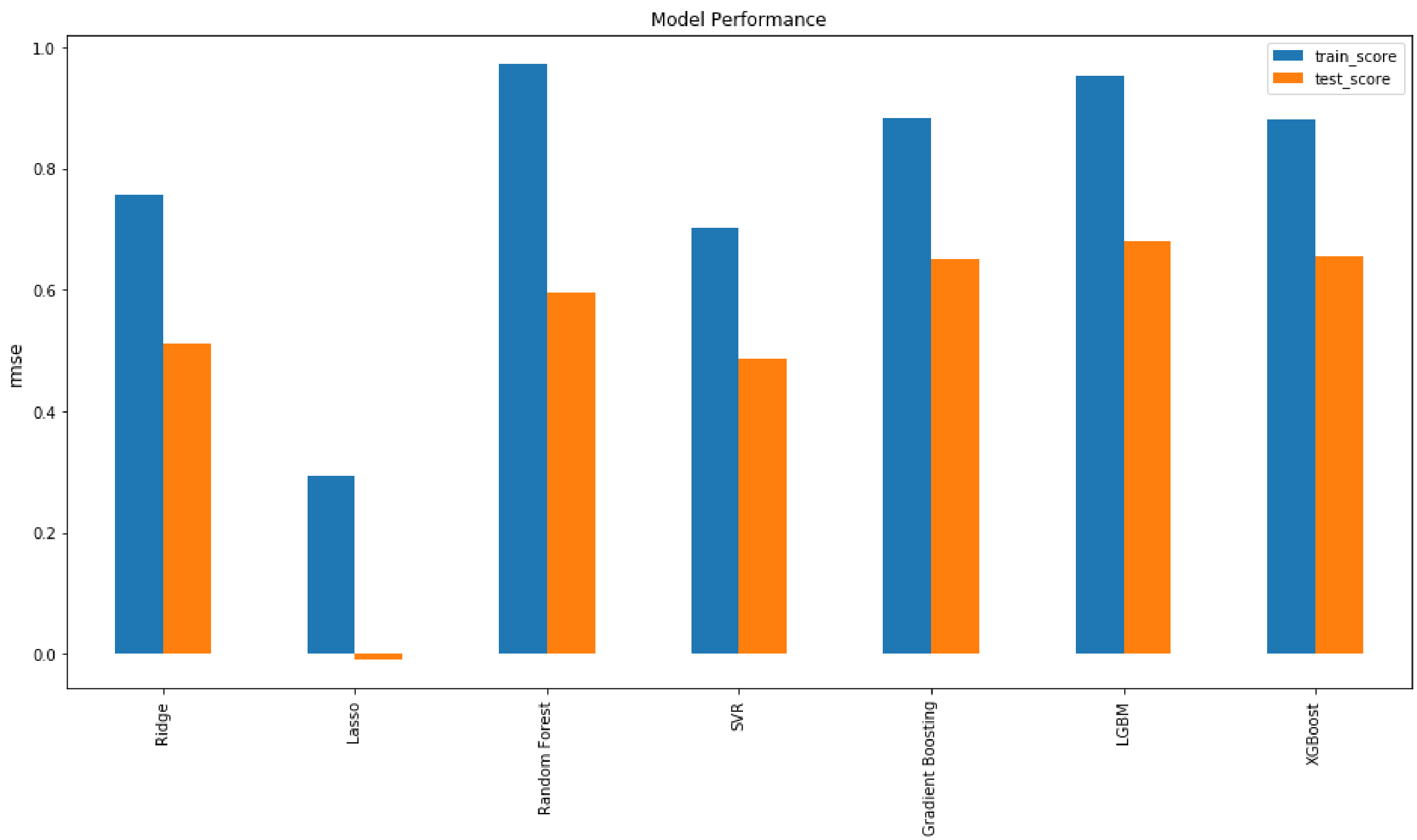
Out[46]:

	name	rmse	test_score	\
Ridge	Ridge	4.976745	0.511779	
Lasso	Lasso	5.529965	-0.008370	
Random Forest	RandomForestRegressor	5.401900	0.596863	
SVR	SVR	5.387532	0.486393	
Gradient Boosting	GradientBoostingRegressor	5.260943	0.649664	
LGBM	LGBMRegressor	5.150525	0.679183	
XGBoost	XGBRegressor	5.070604	0.655381	

```
                     train_score  train_time
Ridge                   0.758339       0.268
Lasso                   0.292662       0.260
Random Forest           0.972212      39.046
SVR                     0.701419     259.357
Gradient Boosting       0.883455      40.815
LGBM                    0.953913      20.458
XGBoost                 0.882269      33.593
```

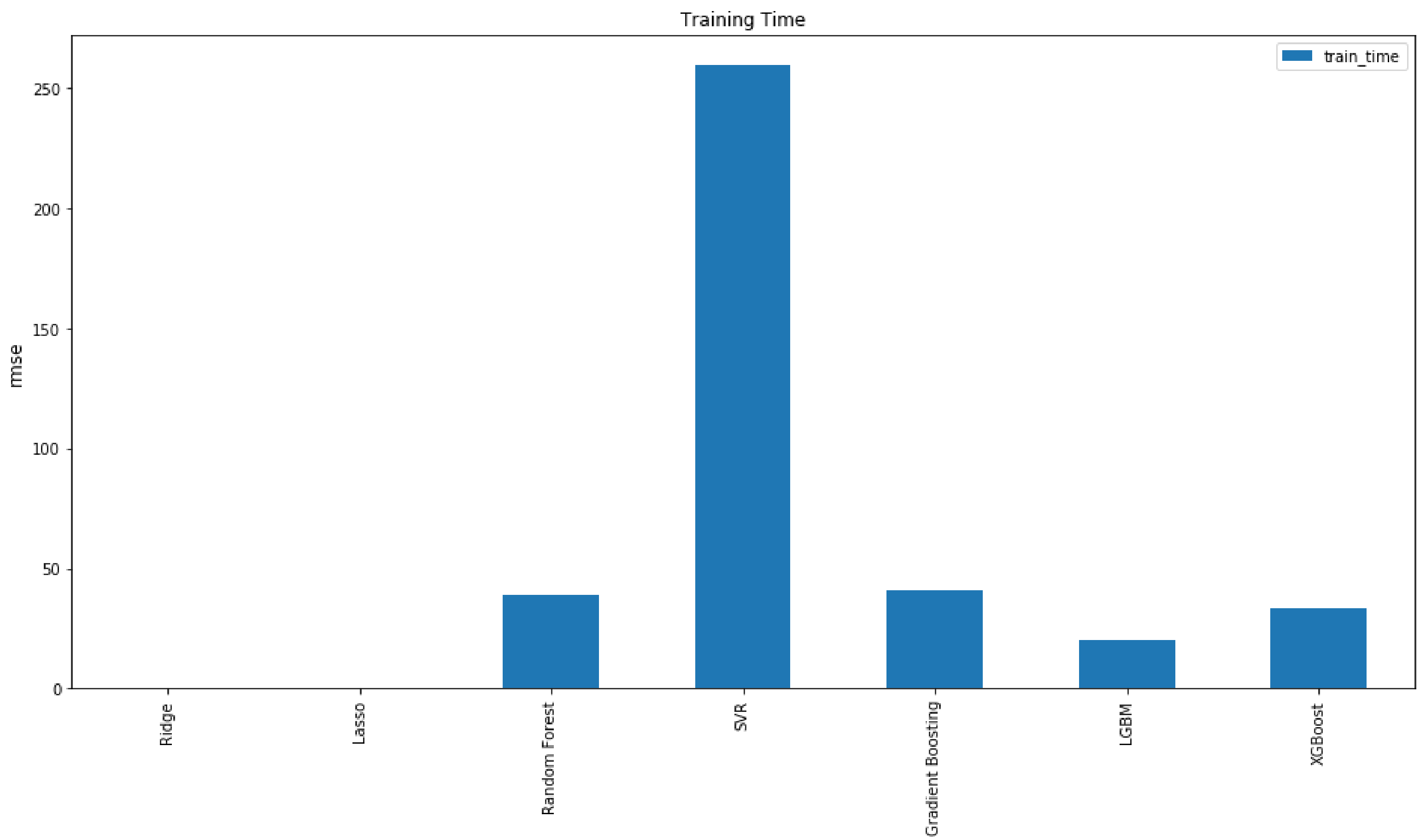
In [47]:
```
# Plot train and test scores
axis = model_summary[["train_score", "test_score"]].plot(kind="bar",
                     title="Model Performance", figsize=(16, 8))
axis.set_ylabel("rmse", fontsize="large")
```

Out[47]:
```
Text(0,0.5,'rmse')
```



The best model according to test scores is LGBM. Although XGBoost and Gradient Boosting Regressor perform quite well.

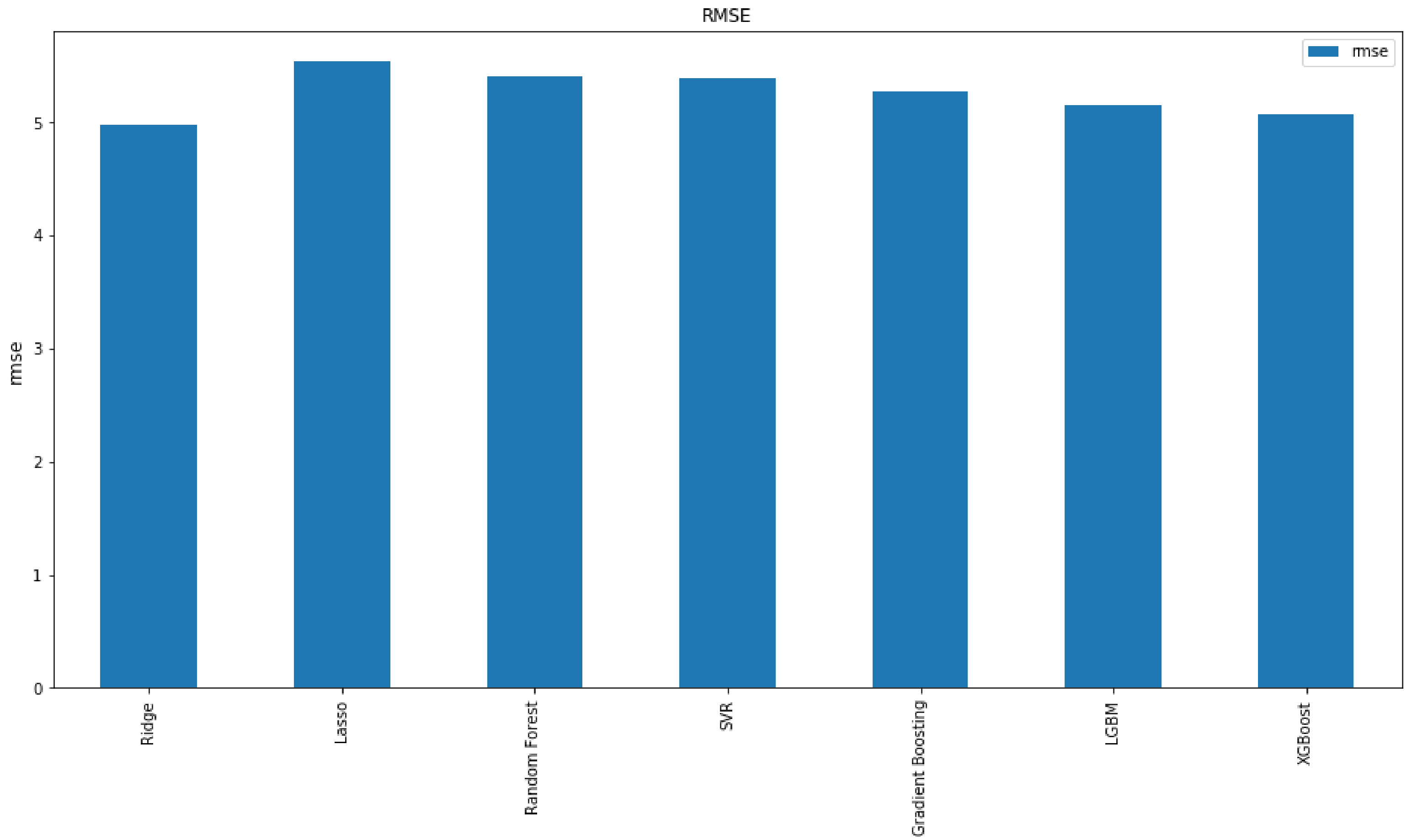
```
In [48]: # Plot train and test scores
         axis = model_summary[["train_time"]].plot(kind="bar", title="Training Time", figsize=(16, 8))
         axis.set_ylabel("rmse", fontsize="large")

Out[48]: Text(0,0.5,'rmse')
```



LGBM doesn't only have the best accuracy, but also has the least training time. Accuracy and training time are 2 important factors if we consider to deploy this model in a productive environment.

```
In [49]: # Plot train and test scores
         axis = model_summary[["rmse"]].plot(kind="bar", title="RMSE", figsize=(16, 8))
         axis.set_ylabel("rmse", fontsize="large")
```

Out[49]: Text(0,0.5,'rmse')



The RMSE Scores for all considered models are on a similar level and LGBM has a lower RMSE compared to most of the models.
Best Model : LGBM Regression **Worst Model**: Lasso

1.5.3 Hyper Parameter Tuning

The hyper parameters of LGBM Regressor can now be tuned using GridSearch CV for better fitting leading to higher accuracy of prediction.

Accuracy: 67.91% Error rate (RMSE): 4.36

```
In [50]: lgbm = LGBMRegressor(min_data=1)
         lgbm_param = {'min_data': [1],'min_data_in_bin':[1],'num_leaves': [50,60],
                       'min_data_in_leaf': [50,60,70],'max_depth': [30,50,70]}
         tscv = TimeSeriesSplit(n_splits=70)
         cv = tscv.split(X)
         lgbm_cv = GridSearchCV(lgbm, lgbm_param, cv=cv, verbose=1)
         lgbm_cv.fit(X,y)
         lgbm_cv.best_params_
         lgbm_result = lgbm_cv.cv_results_
         print("Best LGBM Parameters :",lgbm_cv.best_params_)
         print("LGB Score = ",lgbm_cv.best_score_)
```

```
Fitting 70 folds for each of 18 candidates, totalling 1260 fits


[Parallel(n_jobs=1)]: Done 1260 out of 1260 | elapsed:  7.2min finished


Best LGBM Parameters : {'max_depth': 30, 'min_data': 1, 'min_data_in_bin': 1, 'min_data_in_leaf': 60, 'num_leaves': 50}
LGB Score =  0.682210714016851
```

1.6 Reflection

Based on the results obtained from real world energy data, an accuracy of 68.22% is a good result when we take into account the factors that we haven't accounted for in our problem.

Hypothesis 1 We have only considered the energy produced and consumed inside Germany. Germany also exports and imports energy from its neighbours. These factors have not been accounted for in this dataset. If the exports or import needs have changed between 2015 and 2016, they have not been considered.

Hypothesis 2 Some of the energy generation features are dependent on weather, for example wind or water. We have to calculate certain amount of loss with these features.

Hypothesis 3 The information available does not account for the local energy distribution within Germany and the network structures. If a certain amount of energy is available at a certain point of time, this does not guarantee that the energy can be supplied throughout Germany when demand arises.

For any given market, supply and demand define the price. The energy data that was analyzed shows the complexity of an energy market and multiple dependencies that determine prices. Therefore a greater increase in prediction accuracy can only be achieved when we increase the complexity of the current model.

1.6.1 References

[1] http://colingorrie.github.io/outlier-detection.html (Tukey's Method for outlier detection) [2] https://medium.com/apteo/avoid-time-loops-with-cross-validation-aa595318543e (Look ahead bias) [3] https://machinelearningmastery.com/gentle-introduction-xgboost-applied-machine-learning/ (XGBoost) [4] https://smard.de [5] http://benalexkeen.com/feature-scaling-with-scikit-learn/ (MinMaxScaler)