In [1]:
```
show_gridsearch = False
```

Inleiding

Dit is een Jupyter Notebook. Het kan gebruikt worden om beschrijvingen te maken in 'markdown' (zie deze link (https://github.com/adam-p/markdown-here/wiki/Markdown-Cheatsheet)). Daarnaast kun je code gebruiken (bijvoorbeeld Python) en daarvan de output verwerken in het notebook. Jupyter Notebook wordt daarom ook vaak bij Data Science/Engineering gebruikt. Een Jupyter Notebook kan met diverse programma's worden gelezen en is ook supported door bijvoorbeeld GitHub en Kaggle. Je kunt ook naar pdf printen en daarbij de code wel of niet zichtbaar maken. Door gebruik te maken van secties/hoofdstukken wordt het document overzichtelijk

Ik gebruik Jupyter Notebook in een Anaconda omgeving. Dit is een tool waarmee je diverse 'environments' kunt aanmaken, om er zo voor te zorgen dat bijvoorbeeld versies van libraries niet met elkaar clashen. Anaconda (dat ook Jupyter Notebook bevat) is hier (https://www.anaconda.com/distribution/) te downloaden. Het is aan te raden om de 64-bits versie met **Python 3.7** te nemen.

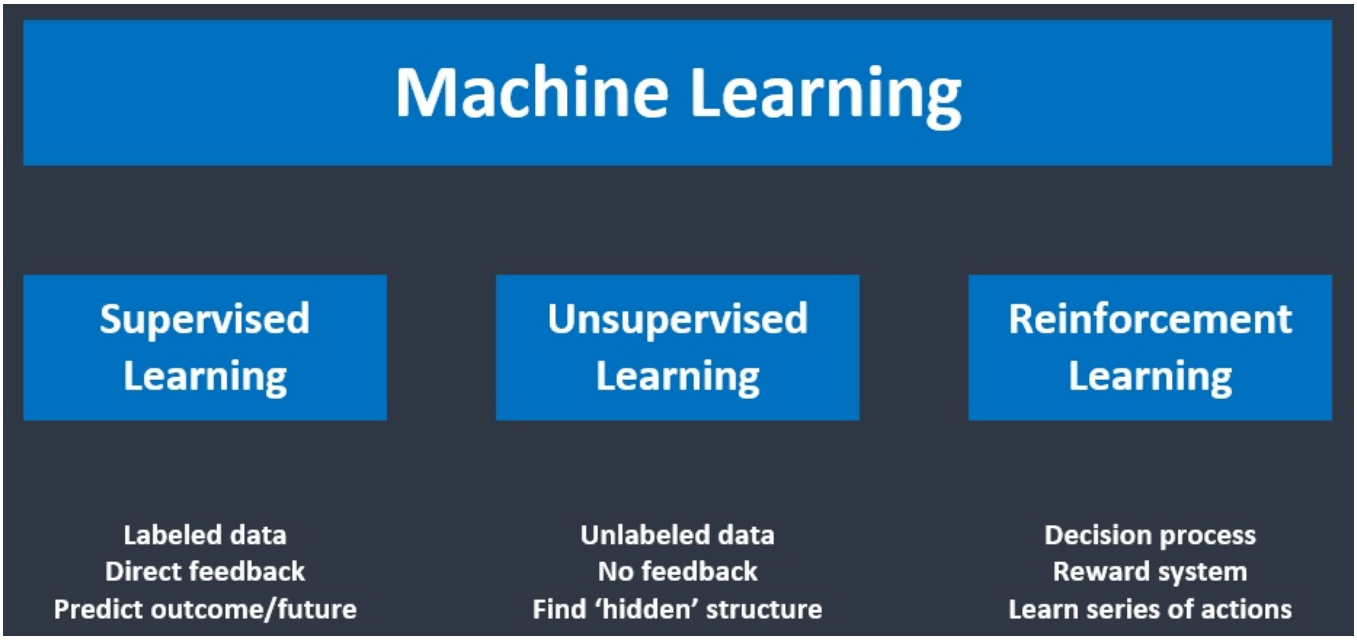
Door te dubbelklikken op een markdown veld, kom je in edit mode. Je gaat weer naar de view mode door op *ctrl-enter* te klikken.

Hier (https://www.dataquest.io/blog/jupyter-notebook-tutorial/) is een tutorial te vinden voor het gebruik van Jupyter Notebook.

Machine Learning

In dit hoofdstuk probeer ik op eenvoudige wijze de aanpak van machine learning te laten zien. Omdat machine learning in vrij omvangrijk vakgebied is, probeer ik de focus te leggen op technieken waarvan ik denk dat ze voor toepassingen bij klanten van Magion handig zouden kunnen zijn.

Eerst nemen we even een stapje terug en kijken we welke vormen van machine learning er zijn:

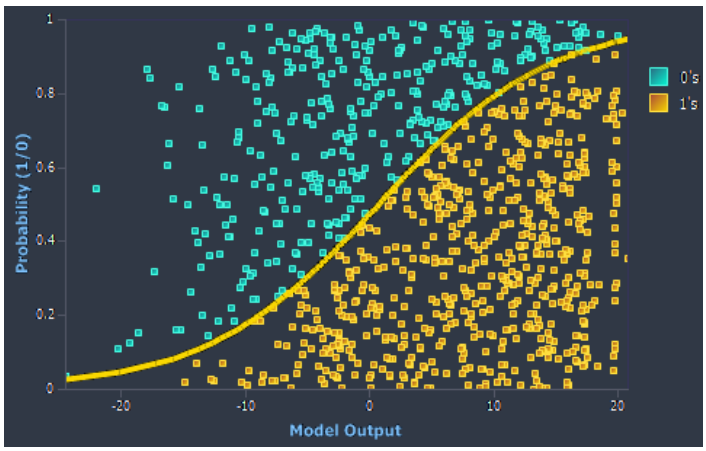


Voor Magion zal supervised learning het meest interessante zijn.

Supervised Learning

Bij supervised learning zijn 2 types, die we nader zullen bekijken:

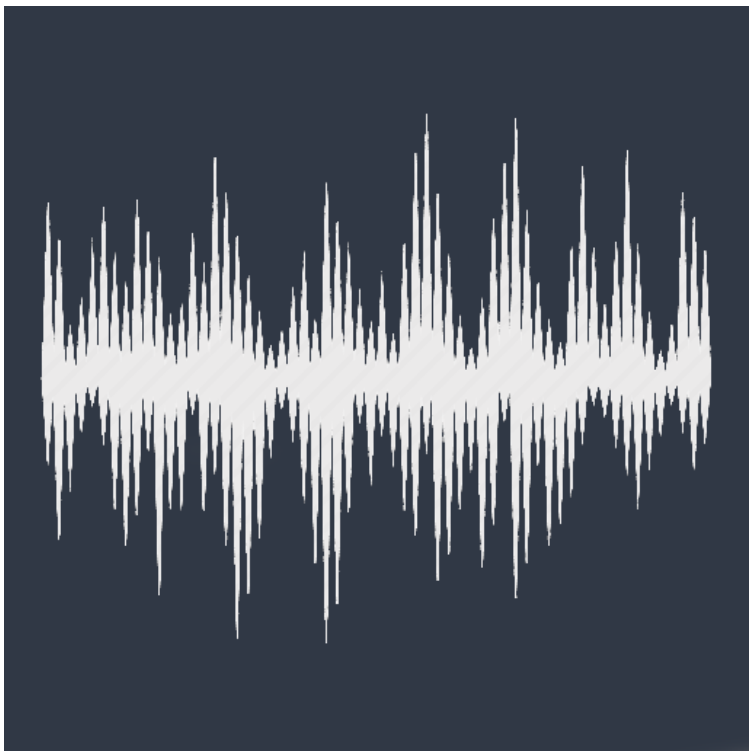
- classification: de input labels en de output zijn een categorie. Voorbeelden:
  - Een (fabrieks)proces is healthy of niet onder bepaalde condities. Dit wordt ook wel binary classification genoemd.



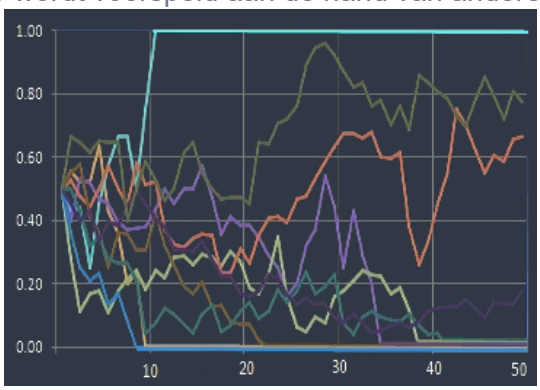
  - Het voorspellen of een moedervlek kwaadraardig is of niet is een ander voorbeeld van binary classification.

  - Het herkennen van een gezicht uit een vaste database mensen. Dit wordt ook wel mutli-class classification genoemnd.

  - Het voorspellen van het type muziek aan de hand van een audio-file is ook een voorbeeld van multi-class classification.



- regression: de input labels en de output zijn een analoge waarde. Voorbeelden:
  - De waarde van een analyser wordt voorspeld aan de hand van andere proceswaardes.



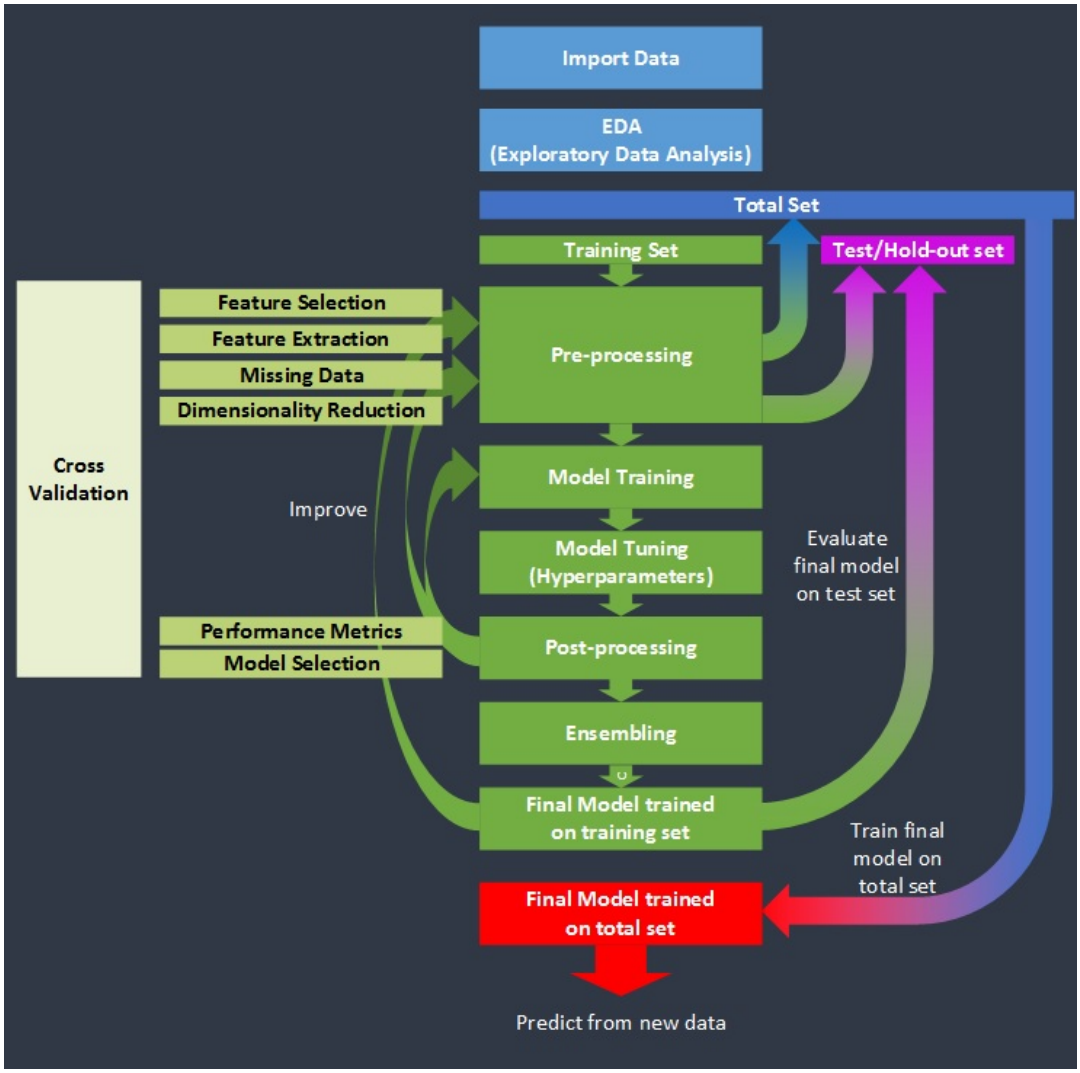
  - Het voorspellen van aandeel-koersen.

Zowel classificatie als regressie is interessant voor ons. Binnen supervised learning zijn nog enkele specialisaties, zoals bijvoorbeeld:

- image recognition (vindt enorm veel toepassing in de medische wereld)
- natural language processing (wordt bijvoorbeeld door zoekmachines gebruikt)
- audio recognition (bijvoorbeeld stem herkenning) en zo zijn er nog meer gebieden... Dit zijn allemaal specialismen die bovenop het "normale" machine learning komen.
- neurale netwerken: deze kunnen op zich op vele gebieden worden toegepast, gebruikelijk bijvoorbeeld bij image recognition, maar ik noem deze toch even apart omdat het bouwen van een neural netwerk wat anders is dan een standaard classificatie of regressie model.

Hetzelfde geldt voor **time-series**, dit is ook een vak apart, waarbij gekeken wordt naar ARIMAs (autoregressive integrated moving average), manieren om met datum/tijd/weekdagen/ploegenschema's om

Flowchart



Python

Waarom Python?

De taal die we gaan gebruiken is Python. Waarom? Er zijn grofweg 3 talen die veel gebruikt worden voor machine learning:

- Matlab
- R
- Python

Matlab leunt heel erg op matrixrekenen en wordt daardoor vooral door academicy gebruikt.

R zit een beetje tussen matlab en Python in en was een of 2 jaar geleden nog net zo populair als Python voor het gebruik in machine learning.

Tegenwoordig zie je echter dat er steeds meer mensen zich richten op Python, waardoor ook de community en dus de support veel groter is dan bij R. Dat houdt vaak in dat je daardoor betere libraries hebt en snellere updates. Daarnaast is het gewoon handig om Python te kennen omdat het een goede basis-programmeer taal is (dat is R zeker niet). Als je op ML competitie sites als Kaggle kijkt, zie je daar dat zo'n 10% alleen R gebruikt, 60% alleen Python en 30% een mix van die twee (bron Kaggle 2018 (https://www.kaggle.com/erikbruin/r-vs-python-and-kmodes-clustering-2018-survey)).

Python leren

Er is een enorm groot aanbod voor het online leren van Python. Zie bijvoorbeeld deze link (https://hackernoon.com/10-free-python-programming-courses-for-beginners-to-learn-online-38312f3b9912) voor een overzicht van enkele gratis cursussen. Ook op onlineacademy.nl (http://onlineacademy.nl) staat een basiscursus python.

Als je het leren van Python meteen wilt combineren met machine learning, heeft Kaggle (https://www.kaggle.com/learn/python) ook diverse gratis cursussen voor je.

Van veel oefenen en google (http://www.google.nl) gebruiken als je er niet uitkomt leer je overigens ook enorm veel!

Python en Jupyter Notebook

Binnen Jupyter Notebook kun je elke cell afzonderlijk uitvoeren. Je voert dan alleen dat stukje code uit. Dat is soms heel handig, soms ook vervelend. Als je bijvoorbeeld een notebook opnieuw start, moet je vaak je hele programma (dus alle cells) opnieuw doorlopen.

Enkele tips om een en ander toch aangenaam te maken:

- Start altijd met een cell met libraries. Deze libraries heb je meestal nodig, hoe kort je stukje code ook is. Door de cell te "Runnen" heb je meteen alle libraries geladen en kun je elk stukje code uitvoeren.
- Soms heb je stukken code waarvan de uitvoering veel tijd kost, maar die je liever zou overslaan als je eenmaal de uitkomst kent. Zet aan het begin van de cel dan bijvoorbeeld:

```
if longtest:
    rest van code
```

zodat de code alleen wordt uitgevoerd als je de parameter longtest `True` hebt gemaakt. Als je run 'all cells' uitvoert en de parameter longtest is `False`, dan zal deze cell overgeslagen worden. En als je weet dat bijvoorbeeld een gridsearch voor hyperparameters soms wel 10 uur duurt, ben je blij dat je de cel even overslaat als je het resultaat toch al weet!

Een kleine oefening

In dit hoofdstuk doen we eerst nog een kleine oefening om te laten zien waarom visualisatie zo belangrijk is in Data Science!

Libraries

Zoals gezegd kun je het beste beginnen met de libraries. In de volgende cel gaan we deze libraries laden. Normaal zou ik hier alle libraries inzetten die ik in de hele code zou gebruiken, maar nu beperk ik me tot enkele standaard libraries en zal de overige libraries pas laden zodra deze gebruikt gaan worden, zodat het een geheel stukje code wordt.

Omdat een library een stukje code is, heeft het ook een versienummer. Het versienummer kan van invloed zijn op de werking van de library. Ook kan het zijn dat een library niet standaard bij de installatie van Python is bijgesloten. In dat geval moet je de library zelf downloaden. Binnen Anaconda kan dat bijvoorbeeld via de user interface, maar ook via een command line (bijvoorbeeld `conda install -c anaconda numpy`).

In [2]:

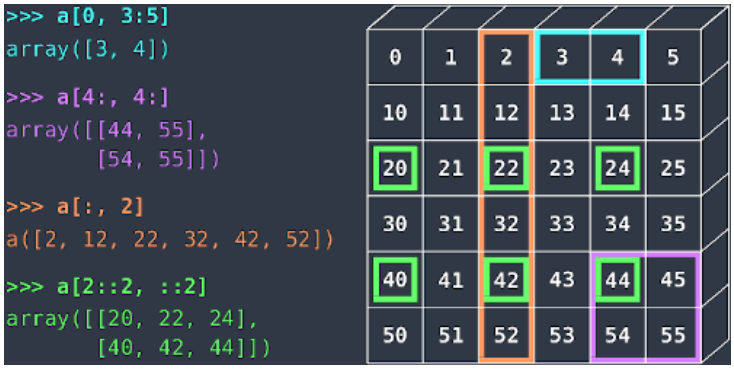
```
# Het laden van libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# Onderstaande code gebruik ik omdat de plots dan matchen met
# de 'dark mode'
from jupyterthemes import jtplot
jtplot.style(theme='oceans16')
```

Als je bovenstaande code uitvoert gebeurt er niets zichtbaars. Soms krijg je warnings die laten zien dat er in een nieuwe versie van een library een wijziging gaat plaatsvinden, maar dat kan verder geen kwaad). Wat je in principe doet is dat je een stuk code naar het geheugen laadt via `import` en deze toekent aan een alias via `as` . Dat laatste is gedaan zodat je minder tikwerk hebt tijdens het gebruik van een library: In plaats van `matplotlib.pyplot.show()` kan ik nu `plt.show()` gebruiken.

De libraries die ik in de vorige cell geladen heb, zijn libraries die veel gebruikt worden bij machine learning.

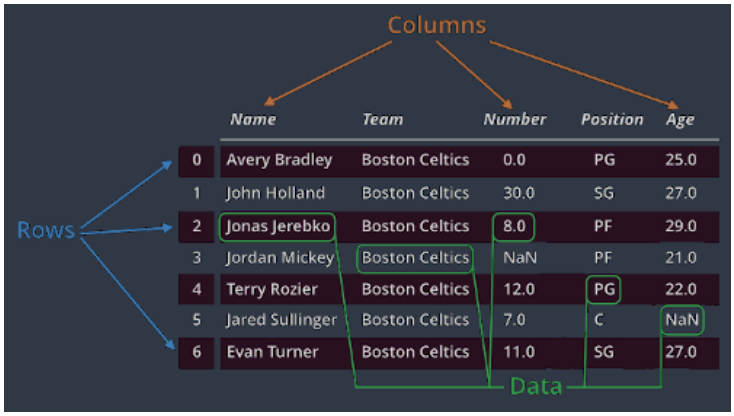
NumPy

NumPy is een library die het mogelijk maakt om snelle berekeningen te maken met multi-dimensionale arrays/matrices. Op de NumPy website (https://numpy.org/) is alle informatie te vinden.



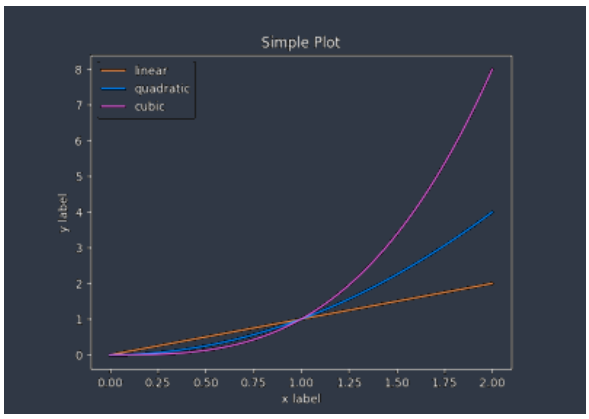
Pandas

ePandas is een uitbreiding op NumPy, waarbij de multi-dimensionale arrays (DataFrames) voorzien zijn van kolomnamen en indexen, om zo eenvoudig elementen binnen het dataframe te benaderen. Ook biedt de library de mogelijkheid om data te importeren van bijvoorbeeld URLs of CSV-bestanden en om dataframes te mergen/sorteren/filteren etc. Onmisbaar voor Machine Learning! Ook voor Pandas is alle informatie op de Pandas website (https://pandas.pydata.org/pandas-docs/stable/) te vinden.



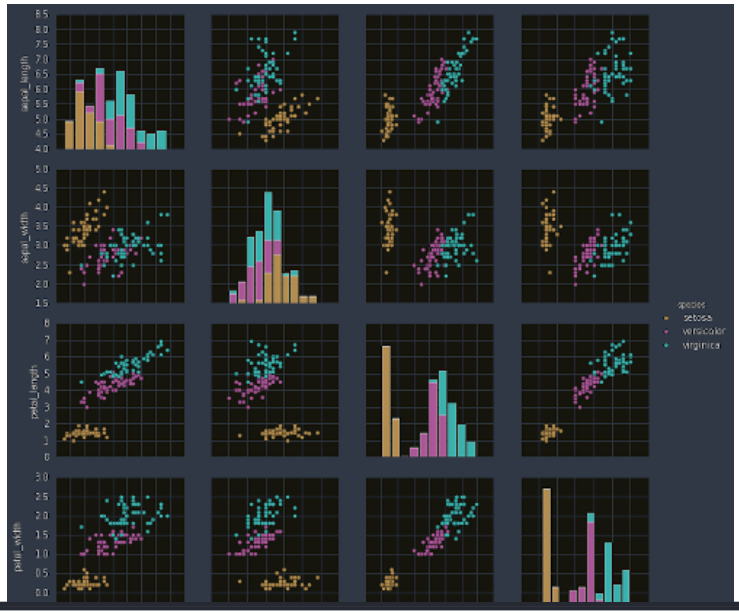
Matplotlib

Matplotlib is een library voor het maken van plots. Visualisatie van data is een van de belangrijkste elementen binnen Data Science en daarom is deze library ook onmisbaar. Meestal maak je maar van een zeer beperkt deel van matplotlib gebruik, zie ook volgende paragraaf. Documentatie is te vinden op de Matplotlib website (https://matplotlib.org/users/index.html).



Seaborn

Seaborn is een library die bovenop Matplotlib is gebouwd en met korte commando's de meest mooie plots kan maken van je data. De [Example Gallery (https://seaborn.pydata.org/examples/index.html)](https://seaborn.pydata.org/examples/index.html) van de website laat enkele voorbeelden zien.



Statistieken...

In de volgende cel gaan we data laden vanuit een csv file. Daarna bekijken we de data en laat ik meteen zien waarom visualisatie zo belangrijk is. Dit is nog niet de set waarop we machine learning gaan loslaten!

In [3]:

```
# load csv to dataframe df_quartet
path = 'https://raw.githubusercontent.com/pvdwijdeven/Magion/master/Data/'
file = 'quartet.csv'
df_quartet = pd.read_csv(path+file)

datasets=[]

# met .describe() kun je wat standaard parameters laten zien.
# Omdat ik alleen de eerste 4 regels wil laten zien, doe ik dat door er
# [0:3] achter te zetten (de eerste regel is 0)
# Afronden naar 2 decimalen kan door er .round(2) achter te zetten
print(df_quartet.describe()[0:3].round(2))
```

```
            x1     y1     x2     y2     x3     y3     x4     y4
count   11.00  11.00  11.00  11.00  11.00  11.00  11.00  11.00
mean     9.00   7.50   9.00   7.50   9.00   7.50   9.00   7.50
std      3.32   2.03   3.32   2.03   3.32   2.03   3.32   2.03
```

We hebben hier 4 sets, waarvan het aantal variabelen (11), het gemiddelde en de standaard deviatie gelijk zijn.

Laten we kijken hoe het zit met het verband tussen x en y bij elke dataset. Hiervoor ga ik gebruik maken van de scipy library (https://docs.scipy.org/doc/scipy/reference/) waarin veel statische functies te vinden zijn. Omdat ik niet de gehele library wil laden, maak ik gebruik van de vorm:

`from library import function .`

In [4]:

```
# import pearson correlation function
from scipy.stats import pearsonr
# import linear regression function
from scipy.stats import linregress

# loop over all 4 datasets
for i in range(0,4):
    # maak 4 x,y datasets
    x = df_quartet.iloc[:,i*2]
    y = df_quartet.iloc[:,i*2+1]
    print('dataset',i+1)
    # bereken daarvan de person correlatie (afgerond op 2 decimalen)
    print("x y correlation:",pearsonr(x,y)[0].round(2))
    # bereken a,b voor y=ax+b
    a, b, _, _, _ = linregress(x, y)
    print("best fit: y = %.2fx+%.2f\n" %(a,b))
```

```
dataset 1
x y correlation: 0.82
best fit: y = 0.50x+3.00

dataset 2
x y correlation: 0.82
best fit: y = 0.50x+3.00

dataset 3
x y correlation: 0.82
best fit: y = 0.50x+3.00

dataset 4
x y correlation: 0.82
best fit: y = 0.50x+3.00
```

Het lijkt er sterk op dat de 4 datasets gelijk zijn: de gemiddelden, standaard deviaties en correlaties zijn gelijk, en zelfs een lineaire regressie geeft hetzelfde resultaat.

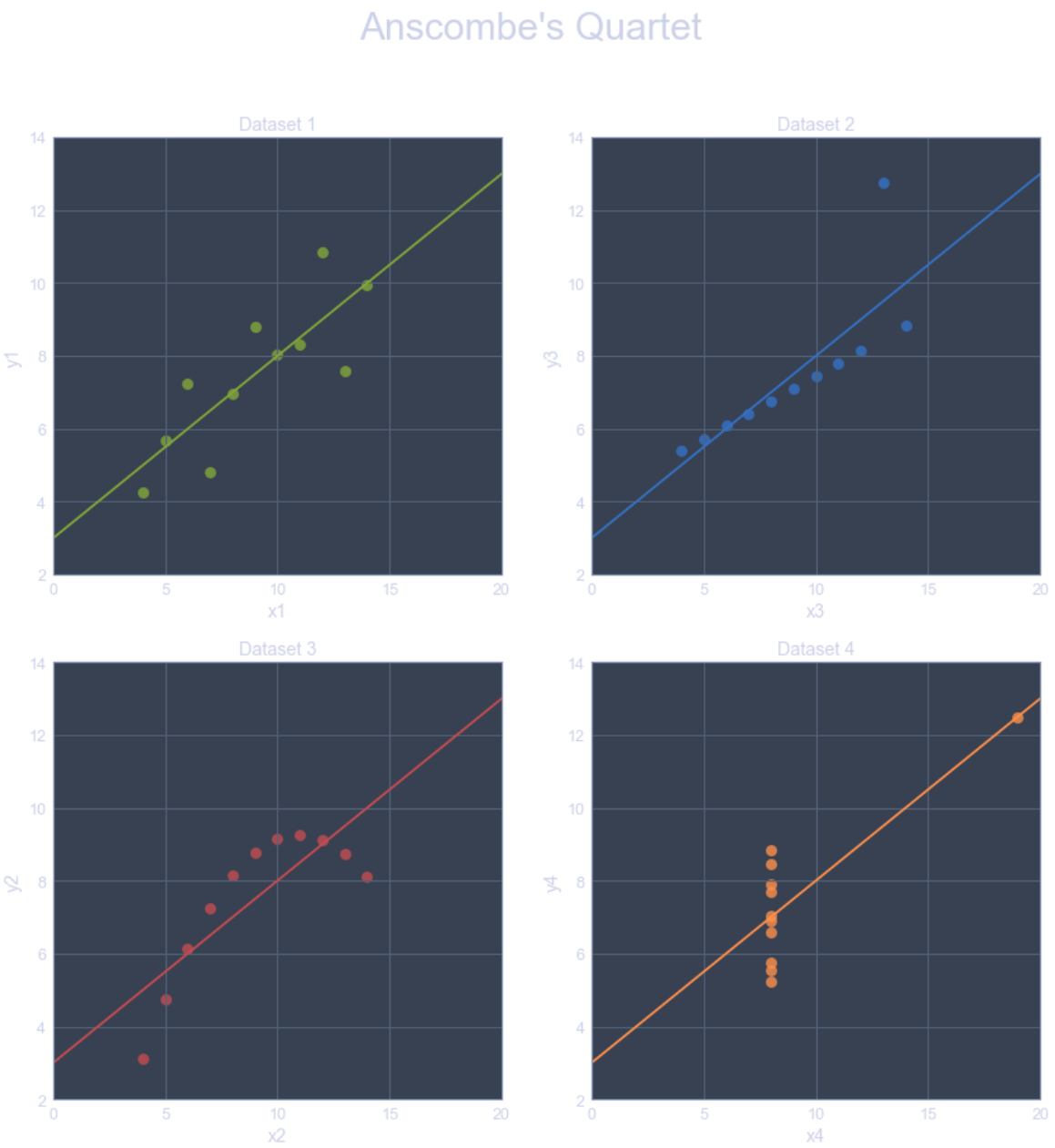
Natuurlijk is er meer aan de hand. We gaan nu zien waarom visualisatie zo belangrijk is.

In de volgende cel gaan we de 4 datasets plotten, met daarin meteen de lineaire regressie weergegeven.

... en visualisatie

In [5]:

```
# Maak een figuur met 4 subplots erin (2 x 2)
fig, axs = plt.subplots(2, 2, figsize=(14, 14))
# Voor de eerste subplot (positie 0,0), maak een regressieplot
axs[0, 0].set(xlim=(0, 20), ylim=(2, 14), xticks=(0, 5, 10, 15, 20),
              yticks=(2, 4, 6, 8, 10, 12, 14),title="Dataset 1")
sns.regplot(x=df_quartet.x1, y=df_quartet.y1, color="g",
            scatter_kws={'s': 80}, ci=None, ax=axs[0, 0])
# Voor de tweede subplot (positie 0,1), maak een regressieplot
axs[0, 1].set(xlim=(0, 20), ylim=(2, 14), xticks=(0, 5, 10, 15, 20),
              yticks=(2, 4, 6, 8, 10, 12, 14),title="Dataset 2")
sns.regplot(x=df_quartet.x3, y=df_quartet.y3, color="b",
            scatter_kws={'s': 80}, ci=None, ax=axs[0, 1])
# Voor de derde subplot (positie 1,0), maak een regressieplot
axs[1, 0].set(xlim=(0, 20), ylim=(2, 14), xticks=(0, 5, 10, 15, 20),
              yticks=(2, 4, 6, 8, 10, 12, 14),title="Dataset 3")
sns.regplot(x=df_quartet.x2, y=df_quartet.y2, color="r",
            scatter_kws={'s': 80}, ci=None, ax=axs[1, 0])
# Voor de vierde subplot (positie 1,1), maak een regressieplot
axs[1, 1].set(xlim=(0, 20), ylim=(2, 14), xticks=(0, 5, 10, 15, 20),
              yticks=(2, 4, 6, 8, 10, 12, 14),title="Dataset 4")
sns.regplot(x=df_quartet.x4, y=df_quartet.y4, color="y",
            scatter_kws={'s': 80}, ci=None, ax=axs[1, 1])
fig.suptitle("Anscombe's Quartet",size=30)
# laat de plot zien
plt.show()
```



Zoals je in bovenstaande plots kunt zien, zijn de datasets geheel verschillend, terwijl de statistieken dit niet lieten zien. Visualisatie geeft meteen een heel ander inzicht. Dit geldt niet alleen voor jezelf, om inzichtelijk te maken met wat voor data je te maken hebt, maar nog veel meer voor degene (bijvoorbeeld klant) aan wie je wilt laten zien hoe de data in elkaar zit. We gaan verderop kijken welke mogelijkheden er allemaal zijn om dit te bewerkstelligen.

Nu wordt het tijd voor het echte werk, we gaan ons volledig storten op een dataset!

Het voorspellen van de huizenmarkt.

De dataset die we gaan gebruiken is afkomstig van Kaggle (http://kaggle.com), een machine learning competitie website. Ik heb de dataset wat ingekrompen wat betreft aantal features, om zo wat sneller door deze oefening heen te kunnen. Hierdoor zullen de resultaten wat minder goed zijn dan wat je zult aantreffen bij Kaggle zelf.

Je kunt de originele (volledige) dataset + competitie hier (https://www.kaggle.com/c/house-prices-advanced-regression-techniques/overview) terugvinden.

Import data

We beginnen met het inlezen van de data. Dit doen we met pd.read_csv, een pandas functie dus. Ik heb hier opzettelijk wat parameters bij gezet, ook al zijn de meeste al per default OK:

- `sep` : geeft aan wat de separator van de csv file is, in dit geval een ','.
- `header` : geeft aan welke rij de header bevat, dus de kolomnamen.
- `index_col` : geeft aan welke kolom de index bevat. In dit geval is dat 'Id'
- `na_filter` : Indien True worden alle waardes zoals "NA" omgezet naar een voor Python herkenbare waarde die "Not Available" representeert. We lezen de csv file in een DataFrame.

Met `print(df_huis.shape)` laten we de grootte van het dataframe zien (rijen, kolommen).

Met `print(df_huis.head(5))` laten we de eerste 5 rows van de DataFrame zien.

In [6]:

```
# inlezen van de data
path = 'https://raw.githubusercontent.com/pvdwijdeven/Magion/master/Data/'
df_huis = pd.read_csv(path+'Huisprijzen_train.csv', sep=',', header=0,
                      index_col='Id', na_filter=True)

print('Grootte van df_huis:')
print(df_huis.shape)

print('\nEerste 5 rijen van df_huis:')
print(df_huis.head())
```

```
Grootte van df_huis:
(1460, 19)

Eerste 5 rijen van df_huis:
    OppPerceel  OppBG  Opp1e  Opp2e  OppKelder  OppGarage  OppOprit  \
Id
1         8450   1710    856    854        856        548      65.0
2         9600   1262   1262      0       1262        460      80.0
3        11250   1786    920    866        920        608      68.0
4         9550   1717    961    756        756        642      60.0
5        14260   2198   1145   1053       1145        836      84.0

    OppTerras  OppBaksteen  Slaapkamers  OpenHaard  Bouwjaar  Verbouwja
ar  \
Id
1           0        196.0            3          0      2003         20
03
2         298          0.0            3          1      1976         19
76
3           0        162.0            3          1      2001         20
02
4           0          0.0            3          1      1915         19
70
5         192        350.0            4          1      2000         20
00

    VerkoopJaar  VerkoopMaand  Kwaliteit  Conditie VerkoopOmst  Verkoop
Prijs
Id
1          2008             2          7         5     Normaal        2
08500
2          2007             5          6         8     Normaal        1
81500
3          2008             9          7         5     Normaal        2
23500
4          2006             2          7         5   Abnormaal        1
40000
5          2008            12          8         5     Normaal        2
50000
```

Voordat we beginnen met Explorartory Data Analysis (EDA), eerst nog even een overzicht wat welke kolom voorstelt. We hebben geluk dat we deze gegevens hebben, dat is namelijk lang niet altijd het geval!

Feature	Beschrijving
OppPerceel	Oppervlakte Perceel
OppWoon	Woonoppervlakte (nto vloerop.)
Opp1e	Oppervlakte 1e verdieping
Opp2e	Oppervlakte 2e verdieping
OppKelder	Oppervlakte kelder
OppGarage	Oppervlakte garage
OppOprit	Oppervlakte oprit
OppTerras	Oppervlakte terras
OppBaksteen	Oppervlakte bakstenen
Slaapkamers	Aantal slaapkamers
OpenHaard	Aantal openhaarden
Bouwjaar	Bouwjaar
Verbouwjaar	Verbouwjaar
VerkoopMaand	Maand van verkoop
VerkoopJaar	Jaar van verkoop
Kwaliteit	Kwaliteit van het huis (1-10)
Conditie	Conditie van het huis (1-10)
VerkoopOmst	Verkoopomstandigheden
VerkoopPrijs	Verkoopprijs

Het doel van de machine learning is het bepalen van de VerkoopPrijs. De *VerkoopPrijs* is dus het `label` , de rest van de parameters zijn de `features` .

EDA

We gaan nu beginnen met de exploratory data analysis:

> *In statistics, exploratory data analysis (EDA) is an approach to analyzing data sets to summarize their main characteristics, often with visual methods.*

Er zijn 2 standaard functies die een eerste inzicht geven in een dataset: `dataframe.info()` en `dataframe.describe()` . Laten we daar mee beginnen.

In [7]:

```
print(df_huis.info())
print(df_huis.describe())
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 1460 entries, 1 to 1460
Data columns (total 19 columns):
OppPerceel      1460 non-null int64
OppBG           1460 non-null int64
Opp1e           1460 non-null int64
Opp2e           1460 non-null int64
OppKelder       1460 non-null int64
OppGarage       1460 non-null int64
OppOprit        1201 non-null float64
OppTerras       1460 non-null int64
OppBaksteen     1452 non-null float64
Slaapkamers     1460 non-null int64
OpenHaard       1460 non-null int64
Bouwjaar        1460 non-null int64
Verbouwjaar     1460 non-null int64
VerkoopJaar     1460 non-null int64
VerkoopMaand    1460 non-null int64
Kwaliteit       1460 non-null int64
Conditie        1460 non-null int64
VerkoopOmst     1460 non-null object
VerkoopPrijs    1460 non-null int64
dtypes: float64(2), int64(16), object(1)
memory usage: 228.1+ KB
None
          OppPerceel        OppBG        Opp1e        Opp2e    OppKelde
r  \
count    1460.000000  1460.000000  1460.000000  1460.000000  1460.00000
0
mean    10516.828082  1515.463699  1162.626712   346.992466  1057.42945
2
std      9981.264932   525.480383   386.587738   436.528436   438.70532
4
min      1300.000000   334.000000   334.000000     0.000000     0.00000
0
25%      7553.500000  1129.500000   882.000000     0.000000   795.75000
0
50%      9478.500000  1464.000000  1087.000000     0.000000   991.50000
0
75%     11601.500000  1776.750000  1391.250000   728.000000  1298.25000
0
max    215245.000000  5642.000000  4692.000000  2065.000000  6110.00000
0

         OppGarage     OppOprit    OppTerras  OppBaksteen  Slaapkamers
\
count  1460.000000  1201.000000  1460.000000  1452.000000  1460.000000
```

```
mean    472.980137    70.049958    94.244521   103.685262     2.866438
std     213.804841    24.284752   125.338794   181.066207     0.815778
min       0.000000    21.000000     0.000000     0.000000     0.000000
25%     334.500000    59.000000     0.000000     0.000000     2.000000
50%     480.000000    69.000000     0.000000     0.000000     3.000000
75%     576.000000    80.000000   168.000000   166.000000     3.000000
max    1418.000000   313.000000   857.000000  1600.000000     8.000000

         OpenHaard     Bouwjaar  Verbouwjaar  VerkoopJaar  VerkoopMaand
\
count  1460.000000  1460.000000  1460.000000  1460.000000   1460.000000
mean      0.613014  1971.267808  1984.865753  2007.815753      6.321918
std       0.644666    30.202904    20.645407     1.328095      2.703626
min       0.000000  1872.000000  1950.000000  2006.000000      1.000000
25%       0.000000  1954.000000  1967.000000  2007.000000      5.000000
50%       1.000000  1973.000000  1994.000000  2008.000000      6.000000
75%       1.000000  2000.000000  2004.000000  2009.000000      8.000000
max       3.000000  2010.000000  2010.000000  2010.000000     12.000000

         Kwaliteit     Conditie   VerkoopPrijs
count  1460.000000  1460.000000    1460.000000
mean      6.099315     5.575342  180921.195890
std       1.382997     1.112799   79442.502883
min       1.000000     1.000000   34900.000000
25%       5.000000     5.000000  129975.000000
50%       6.000000     5.000000  163000.000000
75%       7.000000     6.000000  214000.000000
max      10.000000     9.000000  755000.000000
```

Vanuit de `.info` kun je zien dat de meeste features 1460 elementen bevatten, maar blijkbaar missen we wat data bij *OppOprit* en *OppBaksteen*. We zullen deze later bij de pre-processing fase gaan aanvullen.

De `.describe` waardes komen wat overweldigend over, maar geven behoorlijk wat informatie. Het is echter wat lastig om al die getallen te 'lezen'. Daarom is visualisatie ook zo belangrijk. Dat gaan we nu dus ook doen.

Eerst maak ik een functie waarmee we voor 1 feature meerdere plots tegelijk kunnen genereren. De `dropna()` code zorgt ervoor dat alle NA waardes verwijderd worden. Dit voorkomt fouten in de graph. Later zullen we deze missende waardes gaan vervangen met zo goed mogelijke schattingen.

In [8]:

```
# maak een functie met als parameter 'feature'
def plot_feature(feature):
    # maak een figuur met 3 subplots (1 x 3), dus 3 naast elkaar
    fig, axs = plt.subplots(1, 3, figsize=(15, 5))
    # het linker figuur is een scatterplot
    sns.scatterplot(x=feature, y='VerkoopPrijs',
                    data=df_huis[[feature, 'VerkoopPrijs']].dropna(),
                    ax=axs[0])
    # het middelste figuur is een distributieplot
    sns.distplot(df_huis[feature].dropna(), ax=axs[1], kde=False, bins=20)
    # het rechter figuur is een boxplot
    df_huis[[feature]].dropna().boxplot(sym='b.', ax=axs[2])
    # laat de plots zien
    plt.show()
```

Oppervlakte features

We beginnen met het bestuderen van alle oppervlakte features. Ik maak het mezelf even makkelijk door een lijst van alle oppervlakte features te maken. Dat kan via een ouderwetse for loop, maar ook op een wat meer 'Pythonische wijze': via list comprehension. Beide voorbeelden heb ik hieronder uitgewerkt.

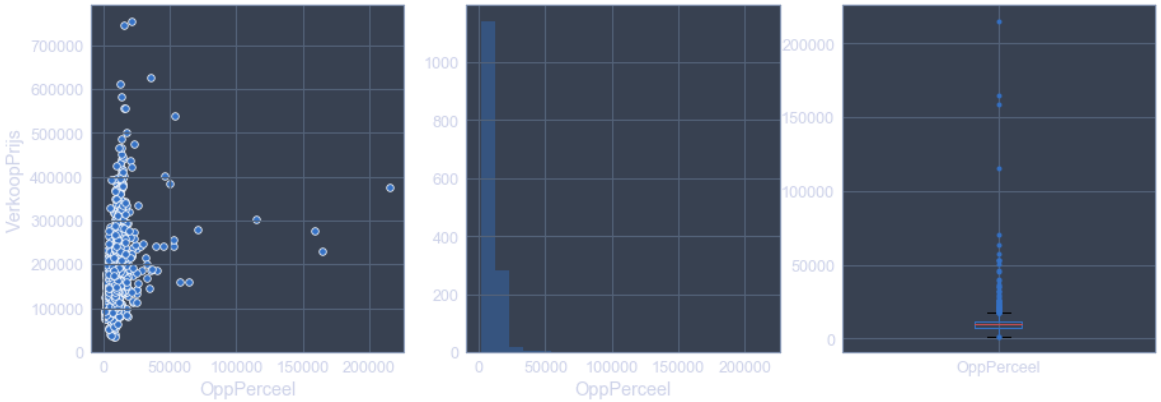
In [9]:

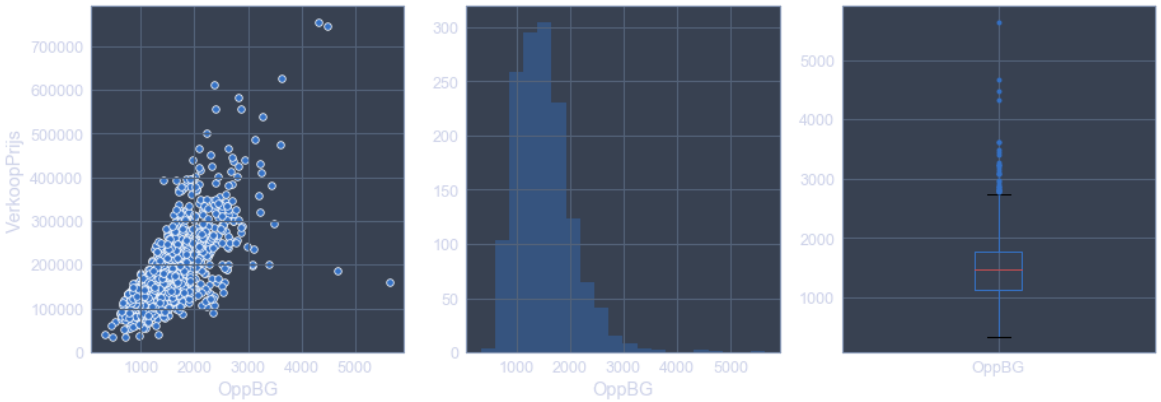
```
# traditionele manier
columns_opp = []
for column in df_huis.columns:
    if column[:3]=='Opp':
        columns_opp.append(column)

# list comprehension
columns_opp = [column for column in df_huis.columns if column[:3] == 'Opp']

print(columns_opp)
```

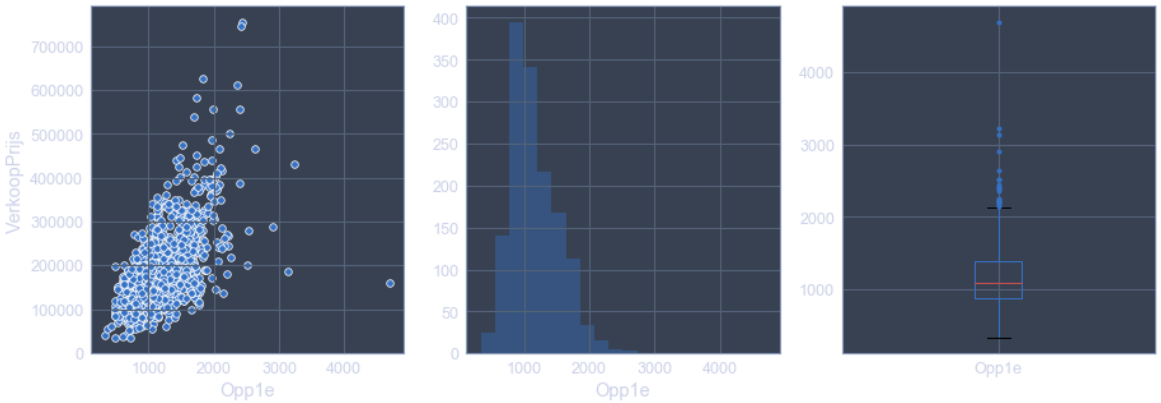
```
['OppPerceel', 'OppBG', 'Opp1e', 'Opp2e', 'OppKelder', 'OppGarage', 'Op
pOprit', 'OppTerras', 'OppBaksteen']
```

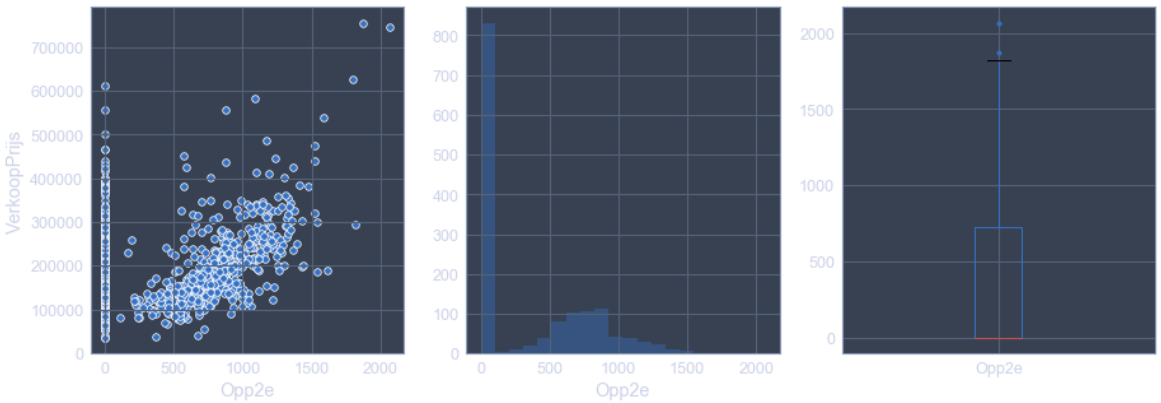
In [10]:

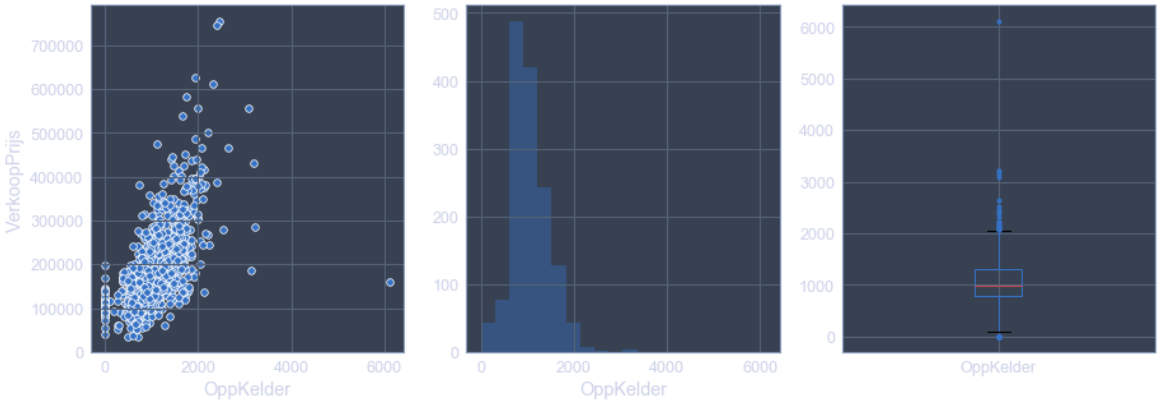
```
for feature in columns_opp:
    plot_feature(feature)
```

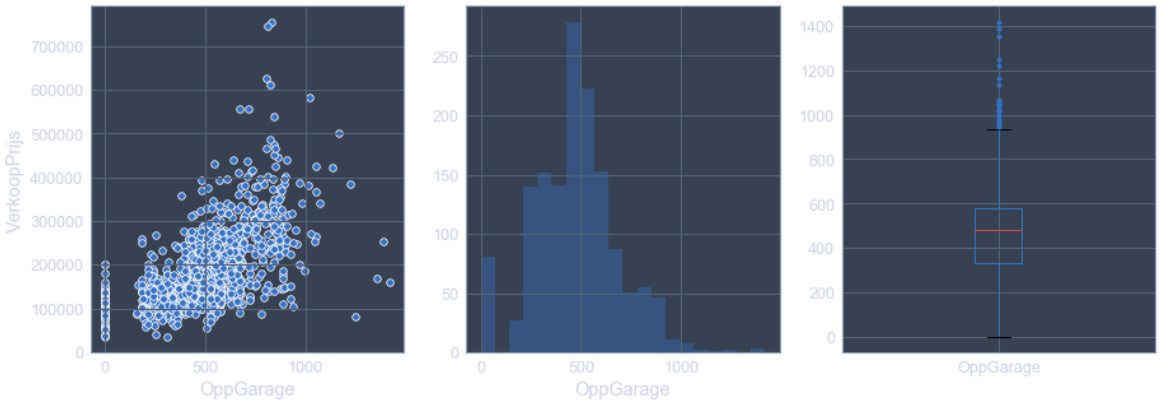

VerkoopPrijs
OppPerceel
OppPerceel
OppPerceel

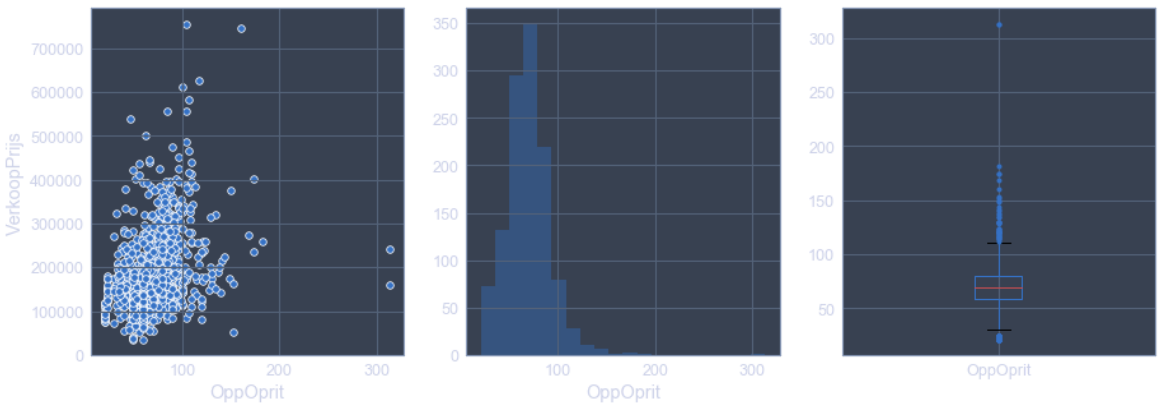

VerkoopPrijs
OppBG
OppBG
OppBG

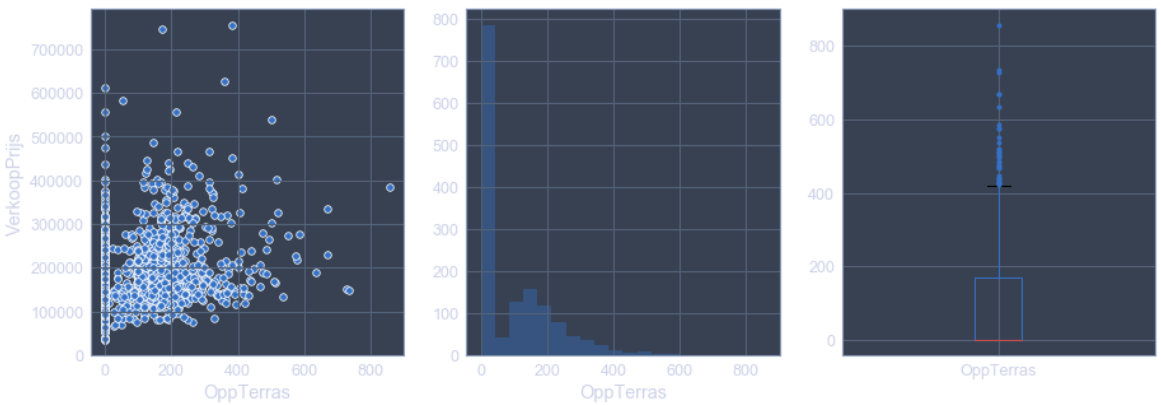

VerkoopPrijs
Opp1e
Opp1e
Opp1e

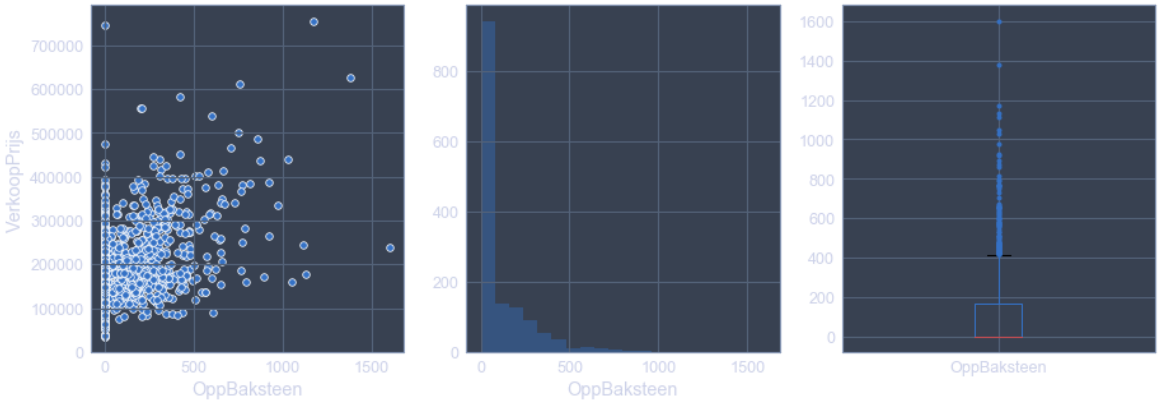

VerkoopPrijs
Opp2e
Opp2e
Opp2e


VerkoopPrijs
OppKelder
OppKelder
OppKelder


VerkoopPrijs
OppGarage
OppGarage
OppGarage


VerkoopPrijs
OppOprit
OppOprit
OppOprit


VerkoopPrijs
OppTerras
OppTerras
OppTerras



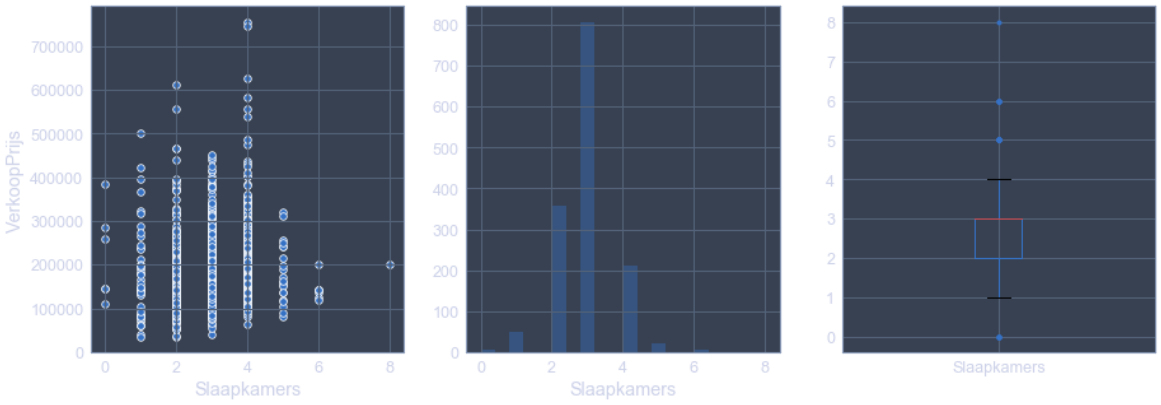
We zien hier enkele dingen:

- Zoals al een beetje verwacht, lijkt het erop dat voor alle oppervlakte features geldt: hoe meer oppervlakte, hoe hoger de huisprijs. Dit valt af te leiden uit de scatterplot. Natuurlijk hebben we het hier over een algemene trend: omdat de huizenprijs een samenstelling is van allerlei parameters, kan het best zijn dat een grotere kelder, met een kleinere woonoppervlak uiteindelijk op een lagere huisprijs uitkomt.
- Zowel de distributieplot als de boxplot laten zien dat er veel hoge 'outliers' zijn. Een grote groep uitschieters kan een machine learning algoritme behoorlijk verstoren. Hier moeten we met feature engineering handig mee omgaan, wellicht door bins (https://scikit-learn.org/stable/auto_examples/preprocessing/plot_discretization.html) te gaan gebruiken.
- We zien ook dat er een aantal features zijn die veel 0-waardes hebben. Bijvoorbeeld Oppervlakte baksteen (blijkbaar zijn er geen baksteen muren bij dat huis), oppervlakte 2e verdieping (die is er dan blijkbaar niet) etc.. Tijdens feature engineering zouden we daar mee om kunnen gaan door ook hier bins te gebruiken, en/of het toevoegen van een extra feature, zoals wel of geen 2e verdieping (een binaire waarde dus).

Aantal slaapkamers

De volgende feature die we gaan bekijken is het aantal slaapkamers.

In [11]:
```
plot_feature('Slaapkamers')
```



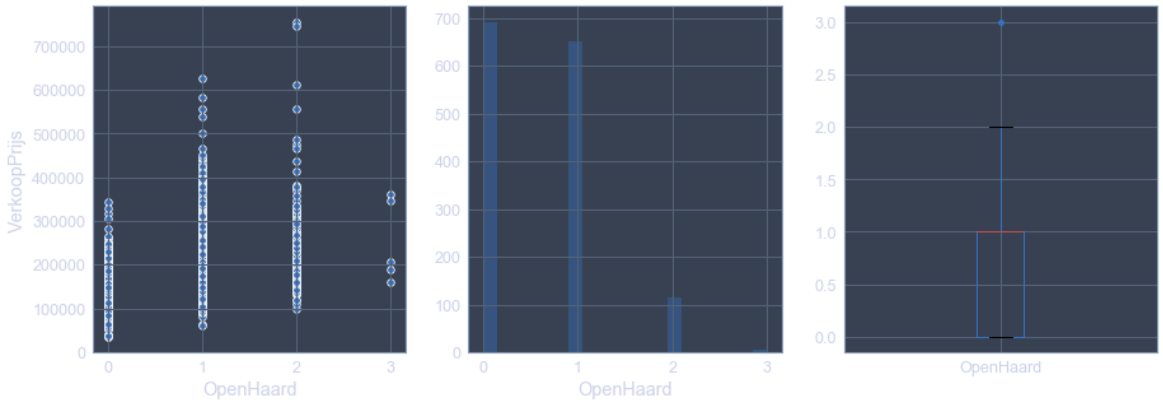
Het is duidelijk te zien dat de meeste huizen 2 of 3 slaapkamers hebben.

Het is moeilijk om te zeggen of er echt een verband zit tussen de verkoopprijs en het aantal slaapkamers. Dat komt waarschijnlijk ook doordat er nog een verband zit tussen slaapkamers en het totale woon oppervlak.

Aantal openhaarden

In [12]:

```
plot_feature('OpenHaard')
```



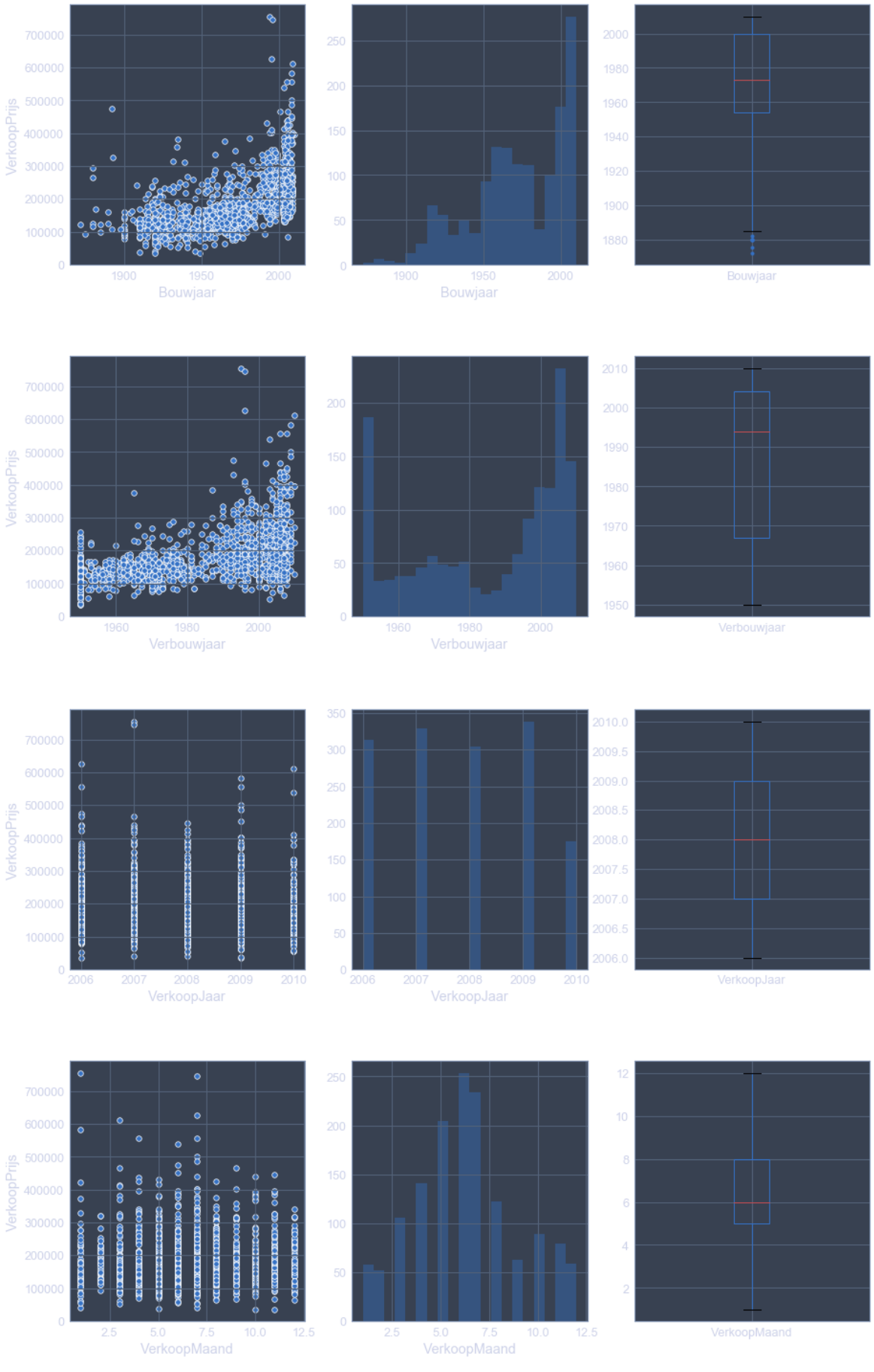
Het aantal openhaarden is meestal 0 of 1, slechts 5% heeft 2 of meer openhaarden. We zouden deze feature daarom kunnen vereenvoudigen door er een boolean feature van te maken: wel of geen openhaard.

(Ver)bouwjaar en Verkoopjaar/maand

Het volgende onderzoek richt zich op het bouwjaar, het jaar van de laatste verbouwing, het verkoopjaar en de bijbehorende verkoopmaand.

In [13]:

```
columns_time=['Bouwjaar','Verbouwjaar','VerkoopJaar','VerkoopMaand']
for feature in columns_time:
    plot_feature(feature)
```



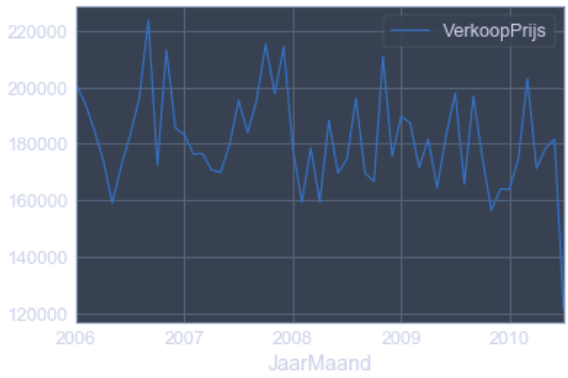
Zowel bouwjaar als verbouwjaar laten een duidelijke stijging van de verkoopprijs zien naarmate de bouw/verbouwing van het huis nieuwer is.

We zouden deze jaartallen nog af kunnen trekken van het verkoopjaar om zo de leeftijd van het gebouw/verbouwing mee te kunnen nemen.

Het verkoopjaar en de verkoopmaand zijn momenteel niet echt handig. Deze twee features moeten duidelijk gecombineerd worden om een beter beeld te krijgen. We zouden kunnen kijken of dat helpt door bijvoorbeeld het gemiddelde van de huisprijzen per maand/jaar te bekijken.

In [14]:

```
# maak een test dataframe aan
df_test=pd.DataFrame()
# De column JaarMaand wordt van het type datetime
df_test['JaarMaand']=pd.to_datetime(df_huis['VerkoopJaar'].astype(str)  +
                                    df_huis['VerkoopMaand'].astype(str),
                                    format='%Y%m')
df_test['VerkoopPrijs']=df_huis['VerkoopPrijs']
# groupeer op JaarMaand en neem daarbij het gemiddelde per maand.
df_test=df_test.groupby('JaarMaand').mean()
df_test.plot()
plt.show()
```



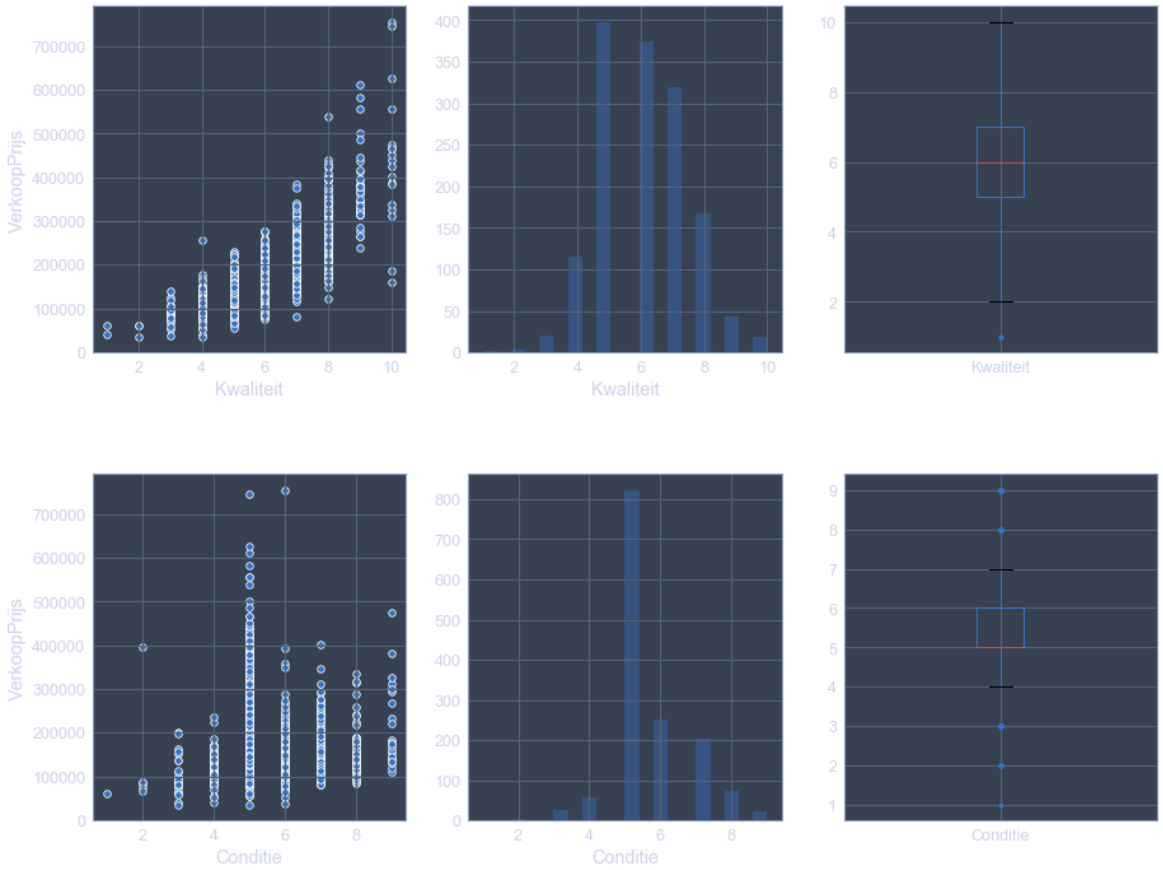
Er lijkt hier een lichte daling in de huisprijs te zijn in de loop van de jaren, maar dat is moeilijk te bepalen en waarschijnlijk statistisch gezien marginaal of zelfs verwerpelijk. Gelukkig hoeven we ons daar geen zorgen over te maken omdat de data gewoon meegenomen zal worden in ons model.

Kwaliteit en conditie

Deze twee features geven allebei een cijfer (1-10) aan de status van het huis, waarbij 1 slecht is en 10 heel goed. Laten we eens kijken wat we hiervan kunnen gebruiken.

In [15]:

```
plot_feature('Kwaliteit')
plot_feature('Conditie')
```



De kwaliteit laat duidelijk een stijgende lijn zien in de verkoopprijs bij een hogere kwaliteit. Voor conditie is dit wat minder zichtbaar. Dit wordt vooral veroorzaakt door het cijfer 5 wat blijkbaar nogal vaak wordt gegeven voor conditie.

Verkoop omstandigheden

De verkoopOmst is een bijzondere feature. Zagen we voorheen overal intergers of floats, nu zien we ineens tekst staan. Laten we eerst eens kijken wat voor verschillende waardes er zijn.

In [16]:

```
# met DataFrame.unique() worden alle unieke waardes weergegeven
print(df_huis['VerkoopOmst'].unique())
```

```
['Normaal' 'Abnormaal' 'NietAf' 'AanlLand' 'Allocatie' 'Familie']
```

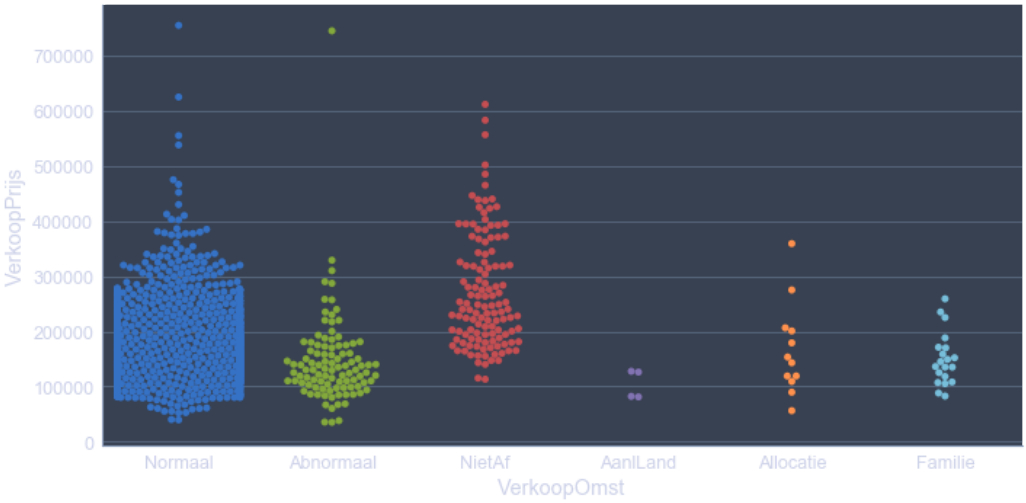
Blijkbaar zijn er 6 verschillende verkoop omstandigheden. Gelukkig zaten er beschrijvingen hiervan bij de dataset:

Status	Beschrijving
Normaal	Normale verkoop
Abnormaal	Abnormale verkoop (handel, executie, snelle verkoop)
AanlLand	Aankoop naastliggend land
Allocatie	2 gelinkte eigendommen mmet verschillende doelen
Familie	Verkoop binnen familie
NietAf	Huis was nog gereed (nieuwbouw, onafgemaakt)

Dit klinkt als een feature die behoorlijke invloed op de verkoopprijs kan hebben. Laten we eens kijken wat we hierover kunnen vinden.

In [17]:

```
sns.catplot(x="VerkoopOmst", y="VerkoopPrijs", kind='swarm',
            data=df_huis,height=5,aspect=2);
```



We zien hier wel dat vooral 'Abnormaal' wat goedkoper uit lijkt te vallen. Van de overige verkoopomstandigheden is de spreiding dusdanig groot en de frequentie zo klein, dat er weinig over te zeggen valt.

Wat wel heel erg opvalt is dat de 'NietAf' huizen redelijk hoog in de verkoop zitten. De reden daarvan is zeer waarschijnlijk omdat dit allemaal nieuwbouw betreft.

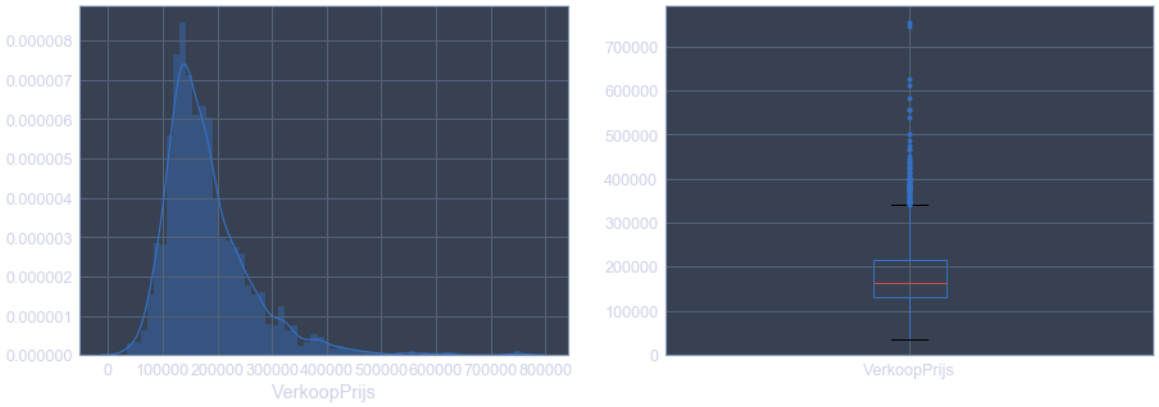
Omdat dit categorische data betreft (tekst of strings), kunnen de meeste algoritmes hier niet mee om gaan. In de pre-processing fase (almost there!) zullen we deze data dan ook gaan omzetten naar numerieke data.

Verkoopprijs

Dan rest natuurlijk nog de waarde waar alles om draait: de verkoopprijs.

In [18]:

```
feature='VerkoopPrijs'
fig, axs = plt.subplots(1, 2, figsize=(15, 5))
sns.distplot(df_huis[feature].dropna(), ax=axs[0], kde=True, bins=60)
# met dropna() worden alle NaN (Not a Number) waarder verwijderd
df_huis[[feature]].dropna().boxplot(sym='b.', ax=axs[1])
plt.show()
```



De plots laten een zogenaamd ***'right skewed'*** normale verdeling zien. Right skewed wil zeggen dat de 'bult' van de normale verdeling wat meer naar links zit, maar er een flinke uitloper naar rechts is. Een kenmerk hierbij is dat het gemiddelde groter is dan de modus:

In [19]:

```
print(df_huis['VerkoopPrijs'].mean())
print(df_huis['VerkoopPrijs'].mode()[0])
```

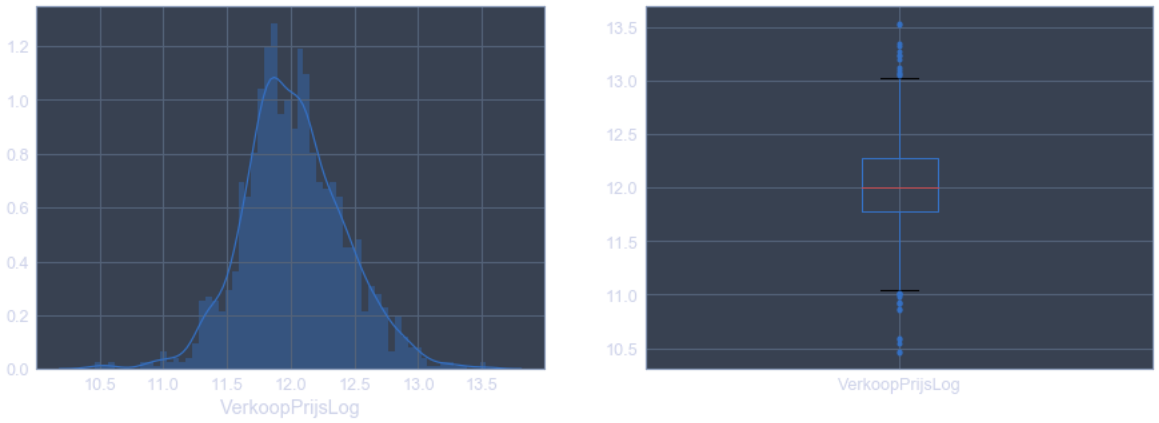
```
180921.19589041095
140000
```

Omdat sommige (zeker niet alle!) algoritmes er vanuitgaan dat de parameters een normale verdeling volgen, kan het zinvol zijn de waarde dusdanig te wijzigen dat het meer op een normale verdeling lijkt.

Voor right skewed data kan dat bijvoorbeeld door een log te nemen van de data. Laten we eens kijken wat voor invloed dat hier heeft.

In [20]:

```
df_test = pd.DataFrame()
df_test['VerkoopPrijsLog']=np.log(df_huis['VerkoopPrijs'])
feature='VerkoopPrijsLog'
fig, axs = plt.subplots(1, 2, figsize=(15, 5))
sns.distplot(df_test[feature].dropna(), ax=axs[0], kde=True, bins=60)
df_test[[feature]].dropna().boxplot(sym='b.', ax=axs[1])
plt.show()

print(df_test['VerkoopPrijsLog'].mean())
print(df_test['VerkoopPrijsLog'].mode()[0])
```




```
12.024050901109373
11.84939770159144
```

Dat ziet er al een stuk beter uit! Als we dit toepassen moeten we er natuurlijk wel rekening mee houden dat de op voorspellingen op het einde weer een geinverteerde log moet worden toegepast:

$$VerkoopPrijs = e^{VerkoopPrijsLog}$$

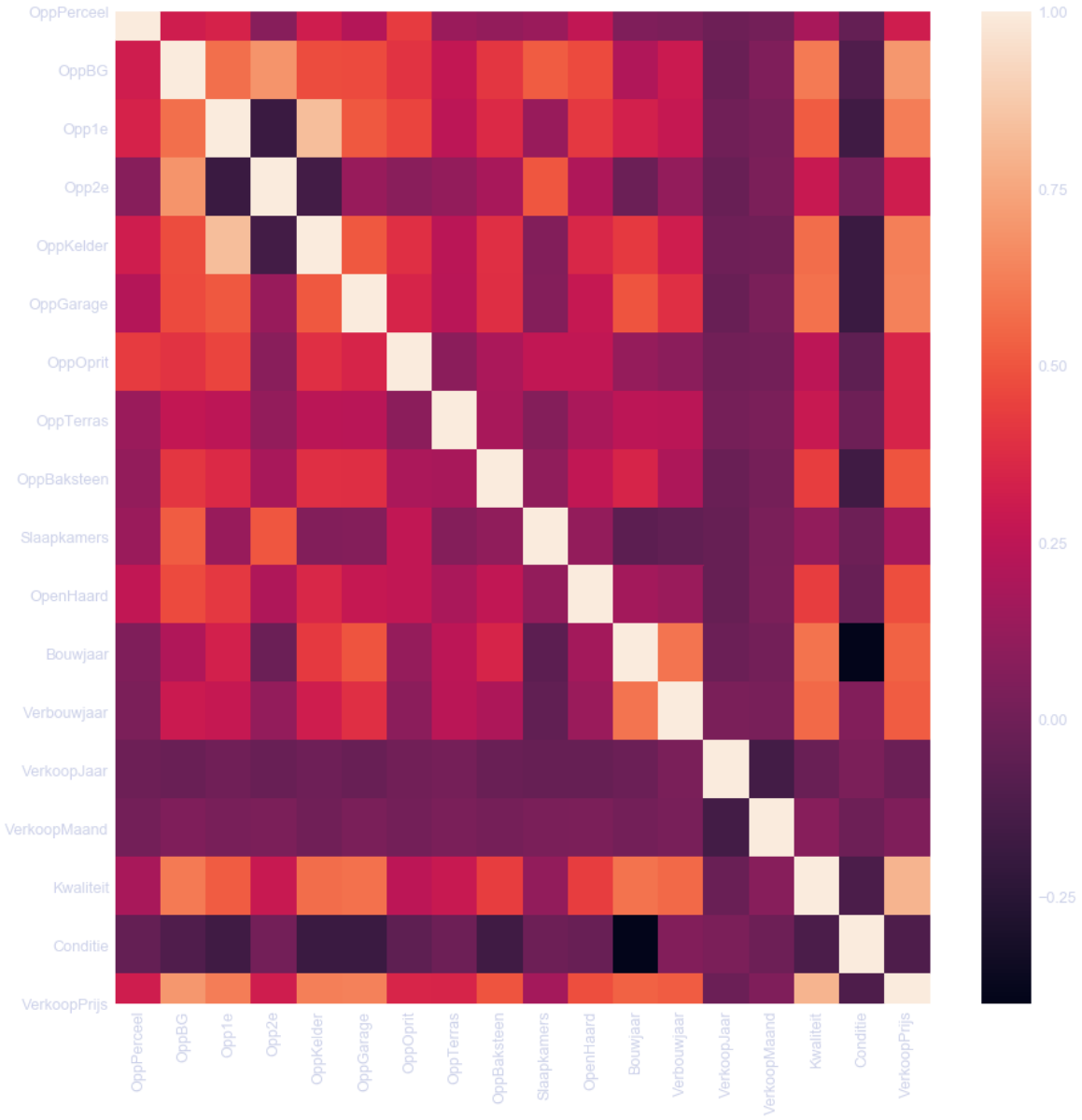
Dat was een mooi moment om te laten zien dat je ook $\LaTeX$ kunt gebruiken in markdown.

Correlaties

Wat belangrijk is in machine learning is om inzichtelijk te maken welke relaties variabelen hebben. Niet alleen met de target waarde (VerkoopPrijs), maar ook onderling. Als features namelijk een onderling verband hebben, kan dat in het machine learning model een negatief effect hebben, omdat dit het model teveel zou kunnen versterken.

In [21]:

```
# we maken hier gebruik van de functie .corr() waardoor een correlatie
# matrix ontstaat. Omdat er nog onbekende (NaN) waardes in de data staan
# negeer ik de rows die deze waardes bevatten (.dropna())
fig, axs = plt.subplots(figsize=(15, 15))
sns.heatmap(df_huis.dropna().corr())
plt.show()
```



Het is niet vreemd om te zien dat de meeste features een vrij hoge correlatie hebben met de verkoopprijs. Ik heb immers een voorselectie gemaakt en daarbij de meest invloedrijke features uit de originele dataset gehaald.

WoonOppervlak en Kwaliteit lijken met veel andere features gecorreleerd te zijn. Het is zeker de moeite waard om te kijken of deze features niet teveel drukken op het model, bijvoorbeeld door te kijken of verwijdering of een lager gewicht geven, een positief resultaat geeft.

Het splitsen van data

Hoewel er vast nog wel meer te verkennen valt (EDA), gaan we toch maar verder. De volgende stap in het flowchart is het splitsen van de data.

Waarom nu al, en niet pas wanneer we het model gaan trainen?

Dit doe je om de invloed van de test-set geheel uit de train-set te halen. Tijdens het pre-processen zou het bijvoorbeeld kunnen dat je de missende waardes wilt vullen met het gemiddelde van de set. Als de test-set daarbij zit, beinvloed je dus de train-set met waardes van de test-set. Dit is ook de reden dat de test-set vaak de hold-out-set wordt genoemd.

We maken hier ook kennis met [sklearn (scikit-learn) (https://scikit-learn.org/stable/)](https://scikit-learn.org/stable/), een library met enorm veel functies voor machine learning. Je importeert dan ook niet de hele library, maar alleen die functies die je nodig hebt.

In [22]:

```
from sklearn.model_selection import train_test_split
# Het aanmaken van een train en test set (resp. 80% en 20%)
X_total = df_huis.drop('VerkoopPrijs',axis=1)
y_total = df_huis['VerkoopPrijs']
# Voor X, de features, neem ik alleen de index, zodat ik die later kan
# gebruiken als 'masker'
train_mask, test_mask, y_train, y_test = \
    train_test_split(X_total.index, y_total, test_size=0.2, random_state=42)
# X_train en X_test maken met het masker. Dit lijkt wat omslachtig, maar
# komt vaker van pas.
X_train = X_total.loc[train_mask]
X_test = X_total.loc[test_mask]
print('De trainset is nu:',X_train.shape)
print('De testset is nu: ',X_test.shape)
```

```
De trainset is nu: (1168, 18)
De testset is nu:  (292, 18)
```

We hebben nu 4 nieuwe variabelen:

- `X_train` : bevat 80% van de originele set, alle feature kolommen
- `y_train` : bevat dezelfde 80% van de originele set, de labels (VerkoopPrijs)
- `X_test` : bevat 20% van de originele set, alle feature kolommen
- `y_test` : bevat dezelfde 20% van de originele set, de labels (VerkoopPrijs)

De trainset zal gebruikt worden om het model te maken en trainen. De test set zal gebruikt worden om dit model te testen, alsof het nieuwe data is.

Wat misschien opvalt is `random_state=42` . Hiervoor kan elk getal gebruikt worden, maar door hier altijd hetzelfde getal te gebruiken zorg je ervoor dat je telkens dezelfde split hebt. Dat is nodig om modellen te kunnen vergelijken. Ook in algoritmes komt vaak een random_state voor en zul je deze voor test-doeleinden ook constant houden.

Pre-processing

Dit is de fase waarin de volgende zaken plaatsvinden:

- invullen van ***missing data***: dit is een must omdat de meeste algoritmes niet kunnen omgaan met ontbrekende data.
- ***feature extraction***: met behulp van de huidige features proberen we features aan te passen, of zelfs nieuwe features van te maken
- ***feature selection***: een selectie van de aanwezige features wordt gemaakt
- ***dimensionality reduction***: afhankelijk van de hoeveelheid data is het soms nodig/handig om het aantal features te 'comprimeren' tot minder features. Dit noemt men dimensionality reduction, met als grote voordeel dat er bij veel data minder opslagruimte en rekencapaciteit nodig is. In ons geval is dit echter niet nodig.

Missing data

Laten we beginnen met de ontbrekende data. Hiervoor zijn diverse manieren om daar vanaf te komen:

- het weglaten van de rijen waarin data ontbreekt.
- proberen de ontbrekende data te interpoleren.
- het gemiddelde of de mediaan nemen van de overige data en daarmee de ontbrekende data vervangen.
- de modus nemen van de overige data en daarmee de ontbrekende data vervangen. Dit werkt ook bij categorische data!
- Met behulp van machine learning de ontbrekende data proberen te schatten (via KNN, MICE of deep learning).

Om het training model zo 'clean' mogelijk te houden, raadt ik aan om hiervoor alleen de training-data te gebruiken.

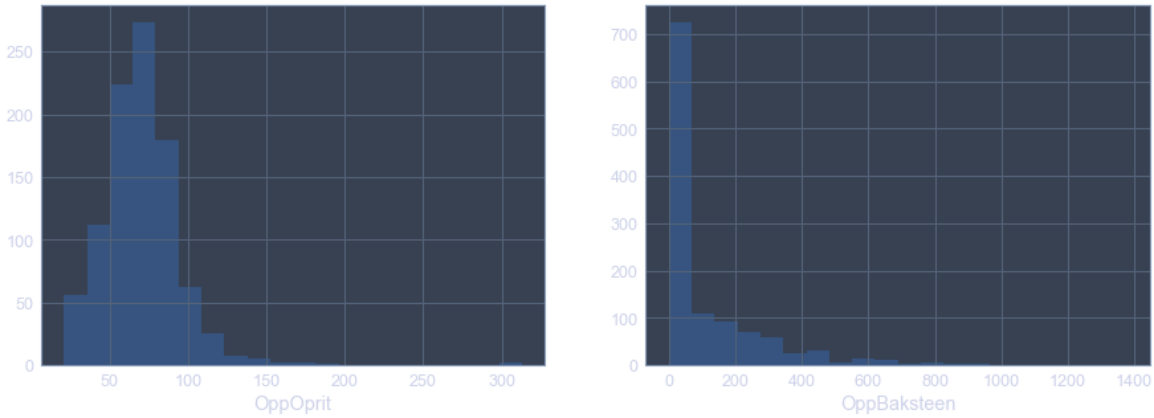
We hebben in dit geval 2 features waarbij data ontbreekt:

- OppOprit
- OppBaksteen

In [23]:

```
fig,axs=plt.subplots(1,2,figsize=(15,5))
sns.distplot(X_train['OppOprit'].dropna(), ax=axs[0], kde=False, bins=20)
sns.distplot(X_train['OppBaksteen'].dropna(), ax=axs[1], kde=False, bins=20)
plt.plot()

print('OppOprit gemiddelde:', X_train['OppOprit'].mean())
print('OppOprit ontbrekende data:', X_train['OppOprit'].isna().sum())
print('OppBaksteen modus:', X_train['OppBaksteen'].mode()[0])
print('OppBaksteen ontbrekende data:', X_train['OppBaksteen'].isna().sum())
```

```
OppOprit gemiddelde: 70.34384858044164
OppOprit ontbrekende data: 217
OppBaksteen modus: 0.0
OppBaksteen ontbrekende data: 6
```



Als we kijken naar de verdeling van OppOprit, lijkt het een goed idee om hier het gemiddelde te nemen om de ontbrekende data mee te vullen. Gezien het hoge aantal ontbrekende data zou het misschien beter zijn om de ontbrekende data via machine learning te achterhalen, maar laten we het ons niet te moeilijk maken de eerste keer.

Voor OppBaksteen missen we maar 6 waardes. Bovendien zien we hier dat er heel veel huizen zijn die helemaal geen baksteen hebben. Laten we daarom deze waarde 0 maar gebruiken om de ontbrekende waardes mee te vullen.

In [24]:

```
def pre_process(data):
    # vullen van ontbrekende data
    data['OppOprit'].fillna(data['OppOprit'].mean(), inplace=True)
    data['OppBaksteen'].fillna(0, inplace=True)
    return(data)

X_train_pp = pre_process(X_train)
print('OppOprit ontbrekende data:', X_train_pp['OppOprit'].isna().sum())
print('OppBaksteen ontbrekende data:', X_train_pp['OppBaksteen'].isna().sum())
```

```
OppOprit ontbrekende data: 0
OppBaksteen ontbrekende data: 0
```

Categorical data

Zoals al eerder gezegd, kunnen de meeste algoritmes niet omgaan met categorische data zoals teksten. Wij hebben 1 feature met categorische data: VerkoopOmst.

Er zijn 2 manieren om met categorische data om te gaan:

- de strings vervangen door getallen. Dit doe je als er heel veel verschillende categorische waardes zijn, of als er weinig categorische waardes zijn, maar deze waardes een logische reeks vertegenwoordigen die goed door getallen kunnen worden uitgedrukt, zoals bijvoorbeeld: "Voldoende", "Ruim Voldoende", "Goed", "Heel goed", "Prima".
- Voor elke categorische waarde een feature aanmaken waarin staat of de waarde wel of niet waar is voor de huidige waarneming (rij). Dit noemt met one-hot-encoding.

We gaan nu hot-encoding toepassen. Aan de hand van de bewerkte data kun je zien wat er gebeurd.

In [25]:

```
df_test = pd.get_dummies(data=X_train['VerkoopOmst'], prefix='Omst')
df_test['VerkoopOmst']=X_train['VerkoopOmst']
print(df_test.head(10))
```

```
      Omst_AanlLand  Omst_Abnormaal  Omst_Allocatie  Omst_Familie  \
Id
255               0               0               0             0
1067              0               0               0             0
639               0               0               0             0
800               0               0               0             0
381               0               0               0             0
304               0               1               0             0
87                0               0               0             0
1386              0               0               0             0
266               0               0               0             0
794               0               0               0             0

      Omst_NietAf  Omst_Normaal VerkoopOmst
Id
255             0             1     Normaal
1067            0             1     Normaal
639             0             1     Normaal
800             0             1     Normaal
381             0             1     Normaal
304             0             0   Abnormaal
87              0             1     Normaal
1386            0             1     Normaal
266             0             1     Normaal
794             1             0      NietAf
```

We gaan dit nu toevoegen aan onze pre-processing functie.

In [26]:

```
def pre_process(data):
    # vullen van ontbrekende data
    data['OppOprit'].fillna(data['OppOprit'].mean(), inplace=True)
    data['OppBaksteen'].fillna(0, inplace=True)
    # one-hot encoding van categorische data
    data = pd.get_dummies(data=data, prefix='Omst', columns=['VerkoopOmst'])
    return(data)

X_train_pp = pre_process(X_train)
```

Baseline model

Nu we geen ontbrekende data meer hebben, wil ik een baseline model maken. Dat is het meest eenvoudige model, zonder dat er wat aan de features is aangepast (behalve de ontbrekende data, maar anders werkt het model niet)...

Voor het baseline model kies ik voor een heel eenvoudig algoritme: Lineare regressie (https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html).

Om te kunnen vergelijken met andere modellen, moeten we natuurlijk een score hebben om te vergelijken. In het geval van regressie, is het een goed idee om te kijken naar RMSE ofwel Root Mean Squared Error (https://en.wikipedia.org/wiki/Root-mean-square_deviation). Hoe groter de RMSE, hoe groter de afwijking tussen de voorspelde waarde en de eigenlijke waarde. De originele Kaggle competie gebruikt de RMSE van de logaritme van de verkoopwaarde als score. Het logaritme is genomen om zo het gewicht in verschil tussen hoge en lage prijzen meer in verhouding te krijgen.

Om niets aan het toeval over te laten, splitsen we de training data 5 keer in een verschillende train/validatie set, telkens op een andere manier. Dat noemt met k-fold validatie, in dit geval dus 5-fold. Je gaat het model dus 5 keer trainen en evalueren en neemt uiteindelijk het gemiddelde van de resultaten.

Hoewel er al veel kant-en-klare functies voor zijn, gaan we om het concept te begrijpen de functies zelf maken:

In [27]:
```
from sklearn.model_selection import KFold

# Eerst definieren we de rmse van de log.
# Om inf te vermijden heb ik het minimum op 0.001 geclipped.
def rmse_log(y,y_pred):
    return np.sqrt(np.mean((np.log(y_pred.clip(0)+1) - np.log(y+1))**2))

# Een herbruikbare functie voor toepassing van KFOLD scoring
def KFold_score(X,y,model,scoring,folds=5,output=True):
    # Eerst het aantal folds definieeren
    # Natuurlijk een random_state=42 om altijd hetzelfde resultaat te krijgen
    kf = KFold(n_splits=folds, shuffle=True, random_state=42)
    # Nu per fold het model fitten en de score berekenen.
    score=[]
    for train_index, val_index in kf.split(X,y):
        X_train = X.iloc[train_index]
        X_val =  X.iloc[val_index]
        y_train = y.iloc[train_index]
        y_val = y.iloc[val_index]
        model.fit(X_train,y_train)
        y_pred=model.predict(X_val).clip(1)
        score.append(scoring(y_val,y_pred))
    score=pd.Series(score)
    if output:
        #print('Model:',model)
        print ('Gemiddelde score:', score.mean())
        print ('Standaard deviatie score:', score.std())
    return(score)
```

Dit gaan we nu testen op een heel simpel model: we nemen het gemiddelde van de training resultaten als predctie-waarde. Dit kunnen we doen met een dummy-regressor:

In [28]:
```
from sklearn.dummy import DummyRegressor
model=DummyRegressor('mean')
result = KFold_score(X_train_pp,y_train,model,rmse_log,10)
```

```
Gemiddelde score: 0.3971761023465816
Standaard deviatie score: 0.03090394580495271
```

Dat is een vrij slechte score, maar dat is niet vreemd. Laten we nu een iets minder simpel model nemen: lineaire regressie

In [29]:
```
from sklearn.linear_model import LinearRegression
model=LinearRegression()
result = KFold_score(X_train_pp,y_train,model,rmse_log,10)
```

```
Gemiddelde score: 0.34537008999527374
Standaard deviatie score: 0.3240268166995933
```

Nu hebben we dus een baseline model, een baseline score en een functie om andere modelen te testen. Het huidige model is zo onnauwkeurig dat de invloed van verbeteringen in features wellicht lastig te zien zijn. Dit is ook te zien in de grote verschillen tussen de verschillende folds: de standaard deviatie is relatief hoog.

Daarom test ik nog een paar andere modellen om een wat betere baseline te hebben.

Ik ga er 2 gebruiken:

- KNN, ofwel K-Nearest Neighbors
- Random Forest Regressor.

Het K-Nearest Neighbors algoritme kijkt zoekt de K dichtstbijzijnde punten in de n-dimensionale ruimte (n= aanal features), en rekent daarbij het gemiddelde resultaat uit.

In [30]:

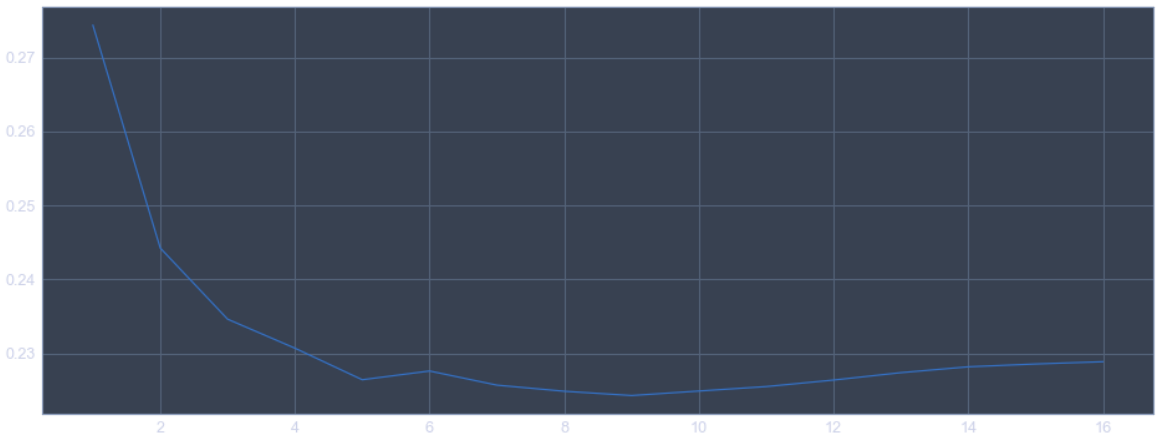
```
from sklearn.neighbors import KNeighborsRegressor
model=KNeighborsRegressor()
result = KFold_score(X_train_pp,y_train,model,rmse_log,10)
```

```
Gemiddelde score: 0.22643952825784922
Standaard deviatie score: 0.029896849919332853
```

Dit resultaat is al een stuk beter en je ziet ook dat er tussen de verschillende folds minder afwijking zit. Een van de belangrijkste parameters van het KNN algoritme is het aantal neighbours (`n_neighbors`). Laten we eens kijken wat de invloed op de score is van deze parameter.

In [31]:

```
score={}
for k in range(1,17):
    model=KNeighborsRegressor(n_neighbors=k)
    result = KFold_score(X_train_pp,y_train,model,rmse_log,10,False)
    score[k]=result.mean()
plt.figure(figsize=(16, 6))
plt.plot(list(score.keys()),list(score.values()))
plt.show()
```



Zoals je in bovenstaande plot ziet, is in dit geval 9 neighbours het optimale getal. Minder levert een overfit op, en meer levert een underfit op. Laten we de score voor n_neighbour=9 aannemen als score voor dit model:

In [32]:
```
model=KNeighborsRegressor(n_neighbors=9)
result = KFold_score(X_train_pp,y_train,model,rmse_log,10)
```

```
Gemiddelde score: 0.22430621657322386
Standaard deviatie score: 0.029991189054013188
```

Het volgende model is en random forest. Een random forest is een verzameling van decision trees. Een decision tree is een algoritme met de nare eigenschap dat je deze zwaar kunt overfitten. Overfitten doe je eigenlijk door te goed te trainen. Door nu iets minder goed te trainen (eigenlijk dus door fouten te introduceren), maar dat voor heel veel verschillende trees op verschillende manieren te doen, krijg je een random forest, die nauwkeurig is, maar met veel minder kans op overfitting.

Het aantal estimators (aantal trees in het random forest) heb ik al ge-optimaliseerd door een vergelijkbare methode als bij KNN. Bij 25 heb je een redelijk goede score en nog steeds een snel model. 125 zou beter zijn, maar maakt het moodel een stuk trager, wat niet handig is bij veelvuldig testen van features.

In [33]:
```
from sklearn.ensemble import RandomForestRegressor
model=RandomForestRegressor(random_state=42, n_estimators=25)
result = KFold_score(X_train_pp,y_train,model,rmse_log,10)
```

```
Gemiddelde score: 0.15135929309698565
Standaard deviatie score: 0.019391545646093047
```

Dat is een hele mooie score, met een lage standaard afwijking. RandomForest wordt dus ons baseline model voor de pre-processing fase.

Bins voor oppervlaktes

Zoals al eerder aangegeven zagen we diverse outliers in de oppervlakte features, maar ook

In [34]:
```
def pre_process(data):
    # vullen van ontbrekende data
    data['OppOprit'].fillna(data['OppOprit'].mean(), inplace=True)
    data['OppBaksteen'].fillna(0, inplace=True)
    # one-hot encoding van categorische data
    data = pd.get_dummies(data=data, prefix='Omst', columns=['VerkoopOmst'])
    # bins en daarna one-hot encoding van alle oppervlakte data
    for column in columns_opp:
        data[column]=pd.cut(data[column],bins=30,labels=False)
        data = pd.get_dummies(data=data, prefix=column)
    return data

X_train_pp = pre_process(X_train)
result = KFold_score(X_train_pp,y_train,model,rmse_log,10)
```

```
Gemiddelde score: 0.15091779008272338
Standaard deviatie score: 0.019886212080746046
```

We zien hier een kleine verbetering in zowel de score als de standaard deviatie tussen de scores van de diverse folds. Laten we deze verbetering dus maar aanhouden!

Normalisering

Een volgende stap zou kunnen zijn het normaliseren van de data: in ons voorbeeld is het aantal slaapkamers een vrij klein getal vergeleken met bijvoorbeeld het perceel-oppervlak. Hierdoor maak je kans dat sommige algoritmes het aantal slaapkamers niet genoeg gewicht geven in de berekeningen. Door alle waardes te normaliseren, bijvoorbeeld door ervoor te zorgen dat ze allemaal tussen 0 en 1 liggen, voorkom je issues hiermee.

In [35]:

```
def pre_process(data):
    # vullen van ontbrekende data
    data['OppOprit'].fillna(data['OppOprit'].mean(), inplace=True)
    data['OppBaksteen'].fillna(0, inplace=True)
    # one-hot encoding van categorische data
    data = pd.get_dummies(data=data, prefix='Omst', columns=['VerkoopOmst'])
    # bins en daarna one-hot encoding van alle oppervlakte data
    for column in columns_opp:
        data[column]=pd.cut(data[column],bins=30,labels=False)
        data = pd.get_dummies(data=data, prefix=column)
    # Het normaliseren van de data
    data = ((data-data.min())/(data.max()-data.min()))
    return data

X_train_pp = pre_process(X_train)
result = KFold_score(X_train_pp,y_train,model,rmse_log,10)
```

```
Gemiddelde score: 0.15057253571488285
Standaard deviatie score: 0.019822907360514917
```

In [36]:

```
# Het gemiddelde van alle min waardes is 0 en van alle max waardes is 1:
print(X_train_pp.min().mean())
print(X_train_pp.max().mean())
```

```
0.0
1.0
```

Zoals je kunt zien heeft de normalisatie weinig effect op de score, maar dat komt ook door het type algoritme. Bij decision trees is normalisatie meestal overbodig, maar bijvoorbeeld bij lineaire regressie is het zeer wenselijk. In het kader van 'baat het niet dan schaadt het niet' laten we de normalisatie gewoon meedoen. Het beste is om deze op het einde van alle feature-manipulaties te

Simplification

Het aantal openhaarden is een feature waarvan we gezien hebben dat de meeste huizen 0 of 1 open haard hebben. Soms zijn er echter 2 openhaarden. We kunnen proberen of we dit kunnen samenvoegen tot een feature die eenvoudig aangeeft of er wel of geen openhaard is in het huis. In dit geval kan dat door het getal 2 te wijzigen naar een 1. Zo hebben we van een Integer een Boolean gemaakt.

In [37]:
```
def pre_process(data):
    # vullen van ontbrekende data
    data['OppOprit'].fillna(data['OppOprit'].mean(), inplace=True)
    data['OppBaksteen'].fillna(0, inplace=True)
    # one-hot encoding van categorische data
    data = pd.get_dummies(data=data, prefix='Omst', columns=['VerkoopOmst'])
    # bins en daarna one-hot encoding van alle oppervlakte data
    for column in columns_opp:
        data[column]=pd.cut(data[column],bins=30,labels=False)
        data = pd.get_dummies(data=data, prefix=column)
    # Maak van openhaard een boolean ipv 0,1,2:
    data['OpenHaard'].loc[data['OpenHaard']==2] = 1
    # Het normaliseren van de data
    data = ((data-data.min())/(data.max()-data.min()))
    return data

X_train_pp = pre_process(X_train)
result = KFold_score(X_train_pp,y_train,model,rmse_log,10)
```

```
C:\Users\pvdwi\Anaconda3\envs\ML\lib\site-packages\pandas\core\indexin
g.py:205: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: http://pandas.pydata.org/pandas-d
ocs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  self._setitem_with_indexer(indexer, value)


Gemiddelde score: 0.1515063074183396
Standaard deviatie score: 0.019866997995415473
```

In [38]:
```
print(X_train_pp['OpenHaard'].max())
```

```
1.0
```

Hoewel we nu inderdaad een boolean hebben gemaakt van de reeks, levert het niets op, sterker nog, de score is een beetje hoger geworden zelfs... Blijkbaar was dit dus geen goed idee, dus dit gaan we niet toepassen. Ik doe dit door een commentaar-teken # te zetten voor de betreffende regel in de functie.

NB: soms is het ook afhankelijk van het algoritme of een idee wel of niet aanslaat!

Jaartallen en maanden

Er zijn een 4-tal kolommen met jaren en maanden. Hier kunnen we nog nuttige dingen mee doen:

- verkoop maand en jaar samenvoegen
- leeftijd van huis sinds bouw of sinds laatste verbouwing

Laten we de resultaten 1 voor 1 bekijken:

In [39]:

```
def pre_process(data):
    # vullen van ontbrekende data
    data['OppOprit'].fillna(data['OppOprit'].mean(), inplace=True)
    data['OppBaksteen'].fillna(0, inplace=True)
    # one-hot encoding van categorische data
    data = pd.get_dummies(data=data, prefix='Omst', columns=['VerkoopOmst'])
    # bins en daarna one-hot encoding van alle oppervlakte data
    for column in columns_opp:
        data[column]=pd.cut(data[column],bins=30,labels=False)
        data = pd.get_dummies(data=data, prefix=column)
    # Maak van openhaard een boolean ipv 0,1,2: (NIET toegepast)
    # data['OpenHaard'].loc[data['OpenHaard']==2] = 1
    # Voeg verkoop jaar en maand samen
    data['VerkoopDatum'] = pd.to_datetime(data['VerkoopJaar'].astype(str)  +
                                  data['VerkoopMaand'].astype(str),
                                  format='%Y%m')
    data.drop(['VerkoopJaar','VerkoopMaand'], axis=1,inplace = True)
    # Het normaliseren van de data
    data = ((data-data.min())/(data.max()-data.min()))
    return data

X_train_pp = pre_process(X_train)
result = KFold_score(X_train_pp,y_train,model,rmse_log,10)
```

```
Gemiddelde score: 0.14931637708208484
Standaard deviatie score: 0.019557277567812958
```

Het samenvoegen van verkoop maand en jaar (en het droppen van de afzonderlijke kolommen) heeft duidelijk een goede invloed op ons model. Laten we kijken wat de bouw/verbouwleeftijd doet. Ik neem hiervoor het verschil tussen bouwjaar en verkoopjaar als het verbouwjaar 1950 is, anders neem ik het verschil tussen verbouwjaar en verkoopjaar. Het lijkt er namelijk op dat als er geen verbouwing was, simpelweg 1950 is ingevuld voor het verbouwjaar.

In [40]:

```
def pre_process(data):
    # vullen van ontbrekende data
    data['OppOprit'].fillna(data['OppOprit'].mean(), inplace=True)
    data['OppBaksteen'].fillna(0, inplace=True)
    # one-hot encoding van categorische data
    data = pd.get_dummies(data=data, prefix='Omst', columns=['VerkoopOmst'])
    # bins en daarna one-hot encoding van alle oppervlakte data
    for column in columns_opp:
        data[column] = pd.cut(data[column], bins=30, labels=False)
        data = pd.get_dummies(data=data, prefix=column)
    # Maak van openhaard een boolean ipv 0,1,2: (NIET toegepast)
    # data['OpenHaard'].loc[data['OpenHaard']==2] = 1
    # Voeg verkoop jaar en maand samen
    data['VerkoopDatum'] = pd.to_datetime(data['VerkoopJaar'].astype(str) +
                                          data['VerkoopMaand'].astype(str),
                                          format='%Y%m')
    data['VerbouwBouwJaar'] = data['Verbouwjaar']
    data['VerbouwBouwJaar'].loc[data['VerbouwBouwJaar'] == 1950] = \
        data['Bouwjaar']
    data['Leeftijd'] = data['VerkoopJaar'] - data['VerbouwBouwJaar']
    data.drop(['VerkoopJaar', 'VerkoopMaand', 'VerbouwBouwJaar',
               'Verbouwjaar', 'Bouwjaar','VerkoopDatum'], axis=1, inplace=True)
    # Het normaliseren van de data
    data = ((data-data.min())/(data.max()-data.min()))
    return data


X_train_pp = pre_process(X_train)
result = KFold_score(X_train_pp, y_train, model, rmse_log, 10)
```

```
C:\Users\pvdwi\Anaconda3\envs\ML\lib\site-packages\pandas\core\indexin
g.py:205: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: http://pandas.pydata.org/pandas-d
ocs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  self._setitem_with_indexer(indexer, value)


Gemiddelde score: 0.15533332667984312
Standaard deviatie score: 0.015234560848917486
```

Dit was geen goed idee. We halen dit definitief uit de functie. Ook het toevoegen van bins bleek hier niets toe te voegen...

Log van verkoopprijs

Zoals al eerder besproken zouden we de logaritme van de verkoopprijs kunnen nemen om een betere verdeling te kunnen krijgen. Laten we eens kijken wat voor effect dat op het model heeft.

Hiervoor moeten we niet alleen de log van y_train nemen, maar ook de log verwijderen uit de score-formule!

In [41]:

```
def rmse(y,y_pred):
    return np.sqrt(np.mean((y_pred-y)**2))

def pre_process(data):
    # vullen van ontbrekende data
    data['OppOprit'].fillna(data['OppOprit'].mean(), inplace=True)
    data['OppBaksteen'].fillna(0, inplace=True)
    # one-hot encoding van categorische data
    data = pd.get_dummies(data=data, prefix='Omst', columns=['VerkoopOmst'])
    # bins en daarna one-hot encoding van alle oppervlakte data
    for column in columns_opp:
        data[column] = pd.cut(data[column], bins=30, labels=False)
        data = pd.get_dummies(data=data, prefix=column)
    # Maak van openhaard een boolean ipv 0,1,2: (NIET toegepast)
    # data['OpenHaard'].loc[data['OpenHaard']==2] = 1
    # Voeg verkoop jaar en maand samen
    data['VerkoopDatum'] = pd.to_datetime(data['VerkoopJaar'].astype(str) +
                                          data['VerkoopMaand'].astype(str),
                                          format='%Y%m')
    data.drop(['VerkoopJaar', 'VerkoopMaand'], axis=1, inplace=True)
    # Het normaliseren van de data
    data = ((data-data.min())/(data.max()-data.min()))
    return data

X_train_pp = pre_process(X_train)
result = KFold_score(X_train_pp, np.log(y_train), model, rmse, 10)
```

```
Gemiddelde score: 0.15248801828406086
Standaard deviatie score: 0.018576712497425457
```

Weer een helaas... Ook hier heeft het geen goede invloed op het model. Toch is dit er eentje om in het achterhoofd te houden als we andere types modellen erop loslaten.

Hyperparameter tuning

We hebben nu redelijk veel energie gestoken in de features, laten we nu eens kijken of we het model kunnen fine-tunen. Dat doen we met behulp van een GridSearch. Dit is een algoritme waarin je een aantal reeksen hyperparameters invoert. Het algoritme probeert dan elke mogelijke combinatie uit en geeft aan wat de beste combi is. Omdat je per set parameters ook nog eens een KFold kunt toepassen, kan het aantal testen vrij hoog zijn, dus houdt rekening met een lange wachttijd!

Laten we dat voor ons model eens uit proberen.

In [42]:

```
from sklearn.model_selection import GridSearchCV

def pre_process(data):
    # vullen van ontbrekende data
    data['OppOprit'].fillna(data['OppOprit'].mean(), inplace=True)
    data['OppBaksteen'].fillna(0, inplace=True)
    # one-hot encoding van categorische data
    data = pd.get_dummies(data=data, prefix='Omst', columns=['VerkoopOmst'])
    # bins en daarna one-hot encoding van alle oppervlakte data
    for column in columns_opp:
        data[column] = pd.cut(data[column], bins=30, labels=False)
        data = pd.get_dummies(data=data, prefix=column)
    # Maak van openhaard een boolean ipv 0,1,2: (NIET toegepast)
    # data['OpenHaard'].loc[data['OpenHaard']==2] = 1
    # Voeg verkoop jaar en maand samen
    data['VerkoopDatum'] = pd.to_datetime(data['VerkoopJaar'].astype(str) +
                                          data['VerkoopMaand'].astype(str),
                                          format='%Y%m')
    data.drop(['VerkoopJaar', 'VerkoopMaand'], axis=1, inplace=True)

    # Het normaliseren van de data
    data = ((data-data.min())/(data.max()-data.min()))
    return data

X_train_pp = pre_process(X_train)

# We nemen het model, zonder parameters (behalve de random_state)
model = RandomForestRegressor(random_state=42)

# De volgende parameters gaan we testen.
params = {'n_estimators': [120, 130, 140],
          'max_features': [18, 17, 16]
          }
GSCV = GridSearchCV(model, param_grid=params, cv=5, verbose=10)
if show_gridsearch:
    GSCV.fit(X_train_pp, y_train)
```

Zoals je ziet zijn er 5 x 3 x 3 = 45 testen gedaan. De uitkomst is opgeslagen in het model. Laten we die uitkomst eens bekijken:

In [43]:

```
if show_gridsearch:
    print(GSCV.best_params_ )
```

In het model gaan we deze parameters toepassen. Dit geeft het volgende resultaat:

In [44]:

```
model = RandomForestRegressor(random_state=42, max_features=17,
                              n_estimators=130)
print('Resultaat zonder log van verkoopprijs:')
result = KFold_score(X_train_pp,y_train, model, rmse_log, 10)
print('\nResultaat met log van verkoopprijs:')
result = KFold_score(X_train_pp,np.log(y_train), model, rmse, 10)
```

```
Resultaat zonder log van verkoopprijs:
Gemiddelde score: 0.14781203239749557
Standaard deviatie score: 0.021343404367987352

Resultaat met log van verkoopprijs:
Gemiddelde score: 0.1471180912877655
Standaard deviatie score: 0.020685147699726115
```

Ensemble models

Zoals je ziet is de score waarbij we de log van y_train nemen tot nu toe het beste van allemaal.

Toch kunnen we nog een stap verder gaan: we stappen in de wereld van de ensemble modellen.

Een ensemble model is een combinatie van modellen/algoritmes waardoor heel veel optimalisatietaken al zijn ingebouwd. Je moet daarbij denken aan handige methodes om het volume training data te vergroten (bootstrapping (https://machinelearningmastery.com/a-gentle-introduction-to-the-bootstrap-method/)), het combineren van modellen via voting, en het optimaliseren via boosting (https://www.analyticsvidhya.com/blog/2015/11/quick-introduction-boosting-algorithms-machine-learning/), waarbij de resultaten (lees: de foute voorspellingen) van een training wordt gebruikt om een volgende training weer te optimaliseren.

XGBoost

We gaan nu een vrij nieuw algoritme toepassen, wat momenteel erg populair is, vooral omdat het gewoon heel vaak een heel goed resultaat geeft: xgboost (https://xgboost.readthedocs.io/en/latest/python/python_api.html#module-xgboost.sklearn). Dit is een library die niet bij sci-kit-learn zit, dus die moet je afzonderlijk installeren. Via conda (zoek binnen windows naar anaconda prompt) kan dit bijvoorbeeld met:

```
conda install -c conda-forge xgboost
```

In [45]:

```
from xgboost import XGBRegressor
model = XGBRegressor(objective='reg:squarederror', random_state=42)
print('Resultaat zonder log van verkoopprijs:')
result = KFold_score(X_train_pp,y_train, model, rmse_log, 10)
print('\nResultaat met log van verkoopprijs:')
result = KFold_score(X_train_pp,np.log(y_train), model, rmse, 10)
```

```
Resultaat zonder log van verkoopprijs:


C:\Users\pvdwi\Anaconda3\envs\ML\lib\site-packages\xgboost\core.py:587:
FutureWarning: Series.base is deprecated and will be removed in a futur
e version
  if getattr(data, 'base', None) is not None and \


Gemiddelde score: 0.14267688566893566
Standaard deviatie score: 0.022360208592407577

Resultaat met log van verkoopprijs:
Gemiddelde score: 0.1398309589715458
Standaard deviatie score: 0.024607487325559327
```

Tuning XGBoost

Zelfs zonder tuning ziet dit er al een stuk beter uit dan het getunede randomforest-model! Zo zie je dat de algoritmes steeds beter worden: de trucks die mensen uithalen om te optimaliseren worden steeds meer verwerkt in de algoritmes. De meeste winst valt dan ook vaak te behalen in de feature-engineering (pre-processing fase). En bij heel veel data door neurale netwerken toe te passen, maar dat is nog een vak apart....

Laten we eens kijken of we de xgboost nog wat scherper kunnen krijgen! Ik ga hier gebruik maken van alle 4 de kernen van de cpu. Daardoor gaan de berekeningen een stuk sneller. Dit doe je door (als het algoritme de parameter heeft) `n_jobs=4` aan te geven.

In [46]:

```
params = {'n_estimators': [200, 225, 250],
          'learning_rate': [0.04, 0.05, 0.06],
          'max_depth': [2,3,4]
          }
model = XGBRegressor(objective='reg:squarederror', random_state=42,n_jobs=4)
GSCV = GridSearchCV(model, param_grid=params, cv=5, verbose=10, n_jobs=4)
GSCV.fit(X_train_pp, y_train)
```

```
Fitting 5 folds for each of 27 candidates, totalling 135 fits


[Parallel(n_jobs=4)]: Using backend LokyBackend with 4 concurrent worke
rs.
[Parallel(n_jobs=4)]: Done   5 tasks      | elapsed:    2.4s
[Parallel(n_jobs=4)]: Done  10 tasks      | elapsed:    2.6s
[Parallel(n_jobs=4)]: Done  17 tasks      | elapsed:    3.1s
[Parallel(n_jobs=4)]: Done  24 tasks      | elapsed:    3.5s
[Parallel(n_jobs=4)]: Done  33 tasks      | elapsed:    4.4s
[Parallel(n_jobs=4)]: Done  42 tasks      | elapsed:    5.3s
[Parallel(n_jobs=4)]: Done  53 tasks      | elapsed:    5.9s
[Parallel(n_jobs=4)]: Done  64 tasks      | elapsed:    6.5s
[Parallel(n_jobs=4)]: Done  77 tasks      | elapsed:    7.6s
[Parallel(n_jobs=4)]: Done  90 tasks      | elapsed:    8.8s
[Parallel(n_jobs=4)]: Done 105 tasks      | elapsed:    9.6s
[Parallel(n_jobs=4)]: Done 120 tasks      | elapsed:   10.7s
[Parallel(n_jobs=4)]: Done 135 out of 135 | elapsed:   12.2s finished


GridSearchCV(cv=5, error_score='raise-deprecating',
             estimator=XGBRegressor(base_score=0.5, booster='gbtree',
                                    colsample_bylevel=1, colsample_byno
de=1,
                                    colsample_bytree=1, gamma=0,
                                    importance_type='gain', learning_ra
te=0.1,
                                    max_delta_step=0, max_depth=3,
                                    min_child_weight=1, missing=None,
                                    n_estimators=100, n_jobs=4, nthread
=None,
                                    objective='reg:squarederror',
                                    random_state=42, reg_alpha=0, reg_l
ambda=1,
                                    scale_pos_weight=1, seed=None, sile
nt=None,
                                    subsample=1, verbosity=1),
             iid='warn', n_jobs=4,
             param_grid={'learning_rate': [0.04, 0.05, 0.06],
                         'max_depth': [2, 3, 4],
                         'n_estimators': [200, 225, 250]},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=Fa
lse,
             scoring=None, verbose=10)
```

In [47]:
```
print(GSCV.best_params_ )
```

```
{'learning_rate': 0.06, 'max_depth': 3, 'n_estimators': 250}
```

In [48]:
```
model = XGBRegressor(objective='reg:squarederror', random_state=42,
                     learning_rate=0.05, n_estimators=225, max_depth=3)
result = KFold_score(X_train_pp,np.log(y_train), model, rmse, 10)
```

```
Gemiddelde score: 0.13964929903597695
Standaard deviatie score: 0.02492658991332524
```

Een overzicht van de scores

Helemaal in het begin zijn we begonnen met een simpel model (het gemiddelde van de verkoopprijs) waarbij de score `0.3972` was. Daarna vonden we een beter algoritme, randomforestregressor, wat een score van `0.1527` opleverde. Door vervolgens met de features te spelen, kwamen we op `0.1540` uit. Toen we het randomforest algoritme gingen tunen was de minimale score nog maar `0.1478` . Tenslotte kwamen we uit op een ensemble algoritme, XGBoost, die, na tuning, een score van `0.1393` opleverde.

Het testen van het model op de hold-out set

Dit is een mooi moment om eens te kijken wat voor score de hold-out set opleverd. Houdt daarbij altijd rekening dat deze wat slechter uit zal vallen dan de validatie score, omdat je totale train-set waarop kfold validation is uitgevoerd altijd beinvloed is met de eigen data.

Laten we beginnen met het trainen van het model op de totale trainingset en daarna te kijken wat de voorspelling op de test-set is. We moeten nu dus niet alleen de totale training set pre-processen, maar ook de test-set.

In [49]:

```
def pre_process(data):
    # vullen van ontbrekende data
    data['OppOprit'].fillna(data['OppOprit'].mean(), inplace=True)
    data['OppBaksteen'].fillna(0, inplace=True)
    # one-hot encoding van categorische data
    data = pd.get_dummies(data=data, prefix='Omst', columns=['VerkoopOmst'])
    # bins en daarna one-hot encoding van alle oppervlakte data
    for column in columns_opp:
        data[column] = pd.cut(data[column], bins=30, labels=False)
        data = pd.get_dummies(data=data, prefix=column)
    # Maak van openhaard een boolean ipv 0,1,2: (NIET toegepast)
    # data['OpenHaard'].loc[data['OpenHaard']==2] = 1
    # Voeg verkoop jaar en maand samen
    data['VerkoopDatum'] = pd.to_datetime(data['VerkoopJaar'].astype(str) +
                                          data['VerkoopMaand'].astype(str),
                                          format='%Y%m')
    data.drop(['VerkoopJaar', 'VerkoopMaand'], axis=1, inplace=True)

    # Het normaliseren van de data
    data = ((data-data.min())/(data.max()-data.min()))
    return data

train_mask=X_train.index
test_mask=X_test.index
X_total_pp = pre_process(X_total)
X_train_pp = X_total_pp.loc[train_mask]
X_test_pp = X_total_pp.loc[test_mask]
X_train_pp = pre_process(X_total)
y_train = y_total
model = XGBRegressor(objective='reg:squarederror', random_state=42,
                     learning_rate=0.05, n_estimators=225, max_depth=3)

model.fit(X_train_pp,np.log(y_train))
y_pred = model.predict(X_test_pp)
print(rmse(y_pred,np.log(y_test)))
```

```
0.11087972053343313
```

Het resultaat is, zoals verwacht, wat minder goed dan de resultaten van de train-set zelf, maar nog steeds zitten we met `0.1444` vrij goed!

Meedoen met een competitie

Zoals gezegd komt de originele database van Kaggle. Nu hebben ik heel veel data weggelaten, waardoor de score een stuk lager zal zijn dan bij de volledige dataset. Om een submission voor Kaggle te doen moeten we een csv file maken met de resultaten die bij de (competitie)test-file hoort.

Omdat we van de verkoopprijs het logaritme hebben genomen, moeten we dit weer ongedaan maken op de einduitkomst door het exponent te nemen.

In [50]:

```
# het inlezen van de competitie data
X_Kaggle = pd.read_csv(path + "Huisprijzen_test.csv", sep=',', header=0,
                       index_col='Id', na_filter=True)

# door een mask te maken kunnen we later de competitie-data weer terugvinden
Kaggle_mask = X_Kaggle.index

# de competitie data wordt samen met de originele trainingdata door de pre-
# processor gehaald
X_train_test = pre_process(X_total.append(X_Kaggle))

# tenslotte wordt de voorspelling berekend (en exponent toegepast)
submission = pd.read_csv(path + "sample_submission.csv")
submission.iloc[:,1] = (np.expm1(model.predict(X_train_test.loc[Kaggle_mask])))

# Wegschrijven van de data
submission.to_csv("submissionMagion.csv", index=False)
```

Hier stopt dit notebook. Dit zal nog allemaal overweldigend en vaag zijn, maar door veel te oefenen wordt het allemaal steeds logischer!