AirBnb Amsterdam Listings

Imports

In [ ]:
```
#disable some annoying warnings
import warnings
warnings.filterwarnings('ignore', category=FutureWarning)

#plots the figures in place instead of a new window
%matplotlib inline

import pandas as pd
import numpy as np

from scipy import stats

from sklearn.cluster import KMeans, Birch, MiniBatchKMeans, Agglomerative
from sklearn.decomposition import PCA
from sklearn.mixture import GaussianMixture
from sklearn.preprocessing import StandardScaler

import matplotlib.pyplot as plt
import matplotlib as mtl
import altair as alt
import seaborn as sns

from ipywidgets import interact

import ipywidgets as widgets

# pd.set_option('display.max_columns', None)
alt.data_transformers.disable_max_rows()
pd.set_option("display.max_rows", None, "display.max_columns", None)

# figure sizes
standard_fig_size = (21,6)
boxplot_fig_size = (15,3)
heatmap_fig_size = (24,12)
countplot_fig_size = (15,6)
```

About the dataset

What is the dataset about?

- The dataset is about detailed Airbnb Listings in Amsterdam.

Where did you get this dataset from (i.e., source of the dataset)?

- The dataset is from the activist project (this is how they call themselves) Inside Airbnb. The data they provide is publicly available data from Airbnb, it can be found at http://insideairbnb.com/get-the-data.html

Loading Data

Since 2008, guests and hosts have used Airbnb to expand on traveling possibilities and present a more unique, personalized way of experiencing the world. Today, Airbnb became one of a kind service that is used and recognized by the whole world. Data analysis on millions of listings provided through Airbnb is a crucial factor for the company. Here we have some above over 5400 number of listings in the city of Amsterdam that generate a lot of data — data that can be analyzed.

Data feature explanation:
https://docs.google.com/spreadsheets/d/2iWCNJcSutYqpULSQHlNyGInUvHg2BoUGoNR

In [ ]:
```
#use a standard dataset of heterogenous data
listings = pd.read_csv('data/listings.csv')
listings.head()
```

Out[ ]:

	id	listing_url	scrape_id	last_scraped	name
0	2818	https://www.airbnb.com/rooms/2818	20211104024252	2021-11-04	Quiet Garden View Room & Super Fast WiFi
1	20168	https://www.airbnb.com/rooms/20168	20211104024252	2021-11-04	Studio with private bathroom in the centre 1
2	27886	https://www.airbnb.com/rooms/27886	20211104024252	2021-11-04	Romantic, stylish B&B houseboat in canal district
3	28871	https://www.airbnb.com/rooms/28871	20211104024252	2021-11-04	Comfortable double room
4	29051	https://www.airbnb.com/rooms/29051	20211104024252	2021-11-04	Comfortable single room

Data cleanup

Dropping superflous features.

In [ ]:
```
# List of the features that are not important for us.
misc_features = [
    'listing_url',
    'scrape_id',
    'name',
    'description',
    'neighborhood_overview',
    'picture_url',
    'host_url',
    'host_location',
    'host_about',
    'host_thumbnail_url',
    'host_picture_url',
    'neighbourhood',
    'neighbourhood_group_cleansed',
    # 'latitude',
    # 'longitude',
    'calendar_updated',
    'calendar_last_scraped',
    'license'
]

# dropping the columns from df
listings = listings.drop(misc_features, axis=1)
```

Data adjustments

For example the list of 'host_verification' options is not that important, what is important is how many different options the host provides/requires for verification.

Features to adjust:

- bathroom_text, ditch text (only number is useful), move then to column bathrooms
- host_verification as an amount of options
- amenities as an amount of options
- districts, 7 districts form 26 neighborhoods

In [ ]:
```
import re
regex = re.compile('[a-z A-Z]')

# doing some float adjustemnts
listings['bathrooms'] = listings['bathrooms_text'].fillna('NaN').str.repl
listings['host_verifications'] = listings['host_verifications'].map(lambd
listings['amenities'] = listings['amenities'].map(lambda x: len([_x.strip
listings['price'] = listings['price'].astype('string').str.strip('$').str
listings['host_response_rate'] = listings['host_response_rate'].str.repla
listings['host_acceptance_rate'] = listings['host_acceptance_rate'].str.r

# doing some data adjustments
# --> use dataype 'timedelta' to have the datatype as a reference between
listings['last_scraped'] = listings['last_scraped'].astype('datetime64')
listings['host_since'] = listings['host_since'].astype('datetime64')
listings['first_review'] = listings['first_review'].astype('datetime64')
listings['last_review'] = listings['last_review'].astype('datetime64')
```

```
neighbourhood_district = {
    'Centre District': ['Centrum-Oost', 'Centrum-West'],
    'Nieuw-West District': ['Geuzenveld - Slotermeer', 'Slotervaart', 'De
    'Noord District': ['Noord-Oost', 'Noord-West', 'Oud-Noord'],
    'Oost District': ['Oostelijk Havengebied - Indische Buurt', 'IJburg -
    'West District': ['De Baarsjes - Oud-West', 'Westerpark', 'Bos en Lom
    'Zuid District': ['Buitenveldert - Zuidas', 'De Pijp - Rivierenbuurt'
    'Zuidoost District': ['Bijlmer-Centrum', 'Gaasperdam - Driemond', 'Bi
}

d = {k: oldk for oldk, oldv in neighbourhood_district.items() for k in ol

listings['neighbourhood_district'] = listings['neighbourhood_cleansed'].m
# drops
listings = listings.drop('bathrooms_text', axis=1)
```

In [ ]:
```
listings.tail()
```

Out[ ]:

	id	last_scraped	host_id	host_name	host_since	host_r
5397	53121415	2021-11-04	303405414	Jm	2019-10-20	witl
5398	53124758	2021-11-04	300888539	WanderlustSolutionbooking	2019-10-08	
5399	53127475	2021-11-04	6606794	Bart	2013-05-27	
5400	53131052	2021-11-04	430112294	Bao	2021-11-02	
5401	53140075	2021-11-04	303405414	Jm	2019-10-20	witl

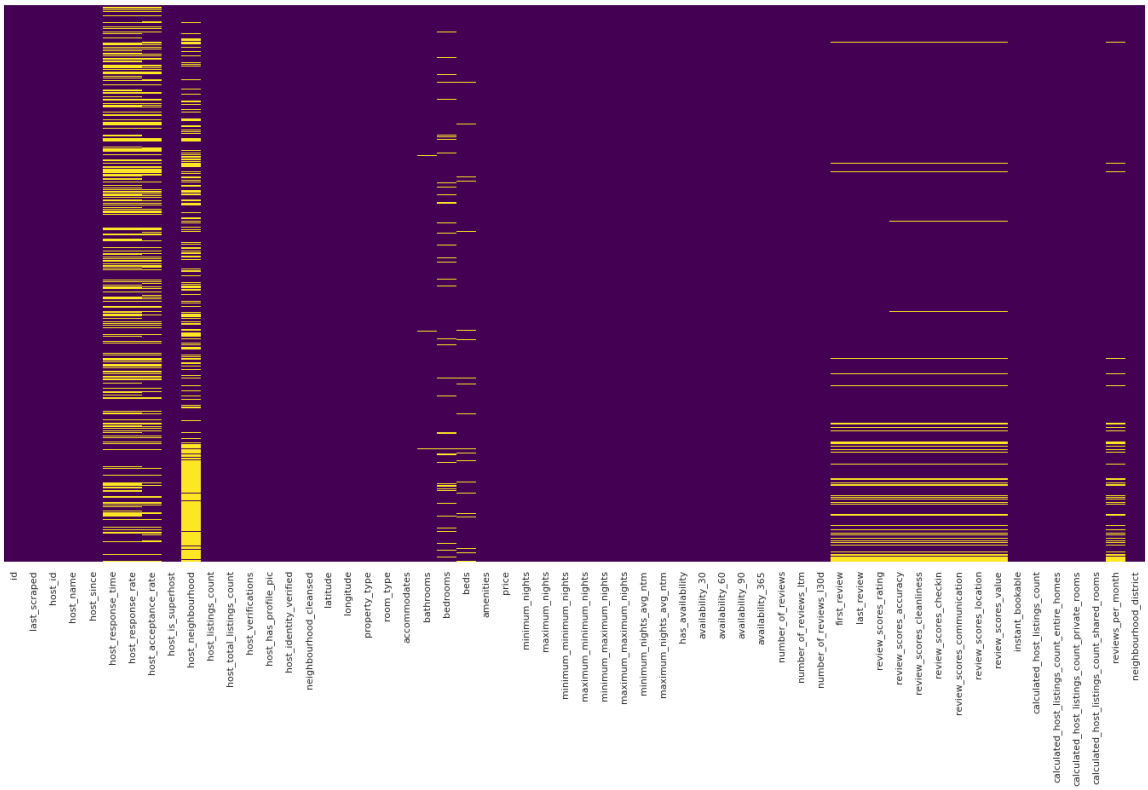
Deeper look into missing data

In [ ]:
```
# figure setup fro following figures
sns.set(rc={'figure.figsize':heatmap_fig_size})
```

To get a deeper understanding of data gathering and the issues that come with missing data, a heatmap displaying null/NaN values is very powerful to find them fast and maybe eliminate some of the features, not only for data visualization and interpretation but also for the possible future training of an agent on the data.

In [ ]:
```
sns.heatmap(listings.isnull(), yticklabels=False, cbar=False, cmap='virid
```

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



What already seems striking is that the lower data entries (i suspect newer, more recently created, ones, which sounds intuitive, but we'll have at look at that later) more often don't seem to have any entries in the regarding review features:

- *features first_review*
- *last_review*
- *review_scores_rating*
- *review_scores_accuracy*
- *review_scores_cleanliness*
- *review_scores_checking*
- *review_scores_communication*
- *review_scores_location*
- *review_scores_value*
- *reviews_per_month*.

As already said we must have a look into this funny and peculiar observation later on.

To do some artificial data filling here would probably obscure the data to much, because about $\frac{1}{4}$ of the entries are effected, so it would be more reasonable to look at why they are effected, then making (!) them whole.

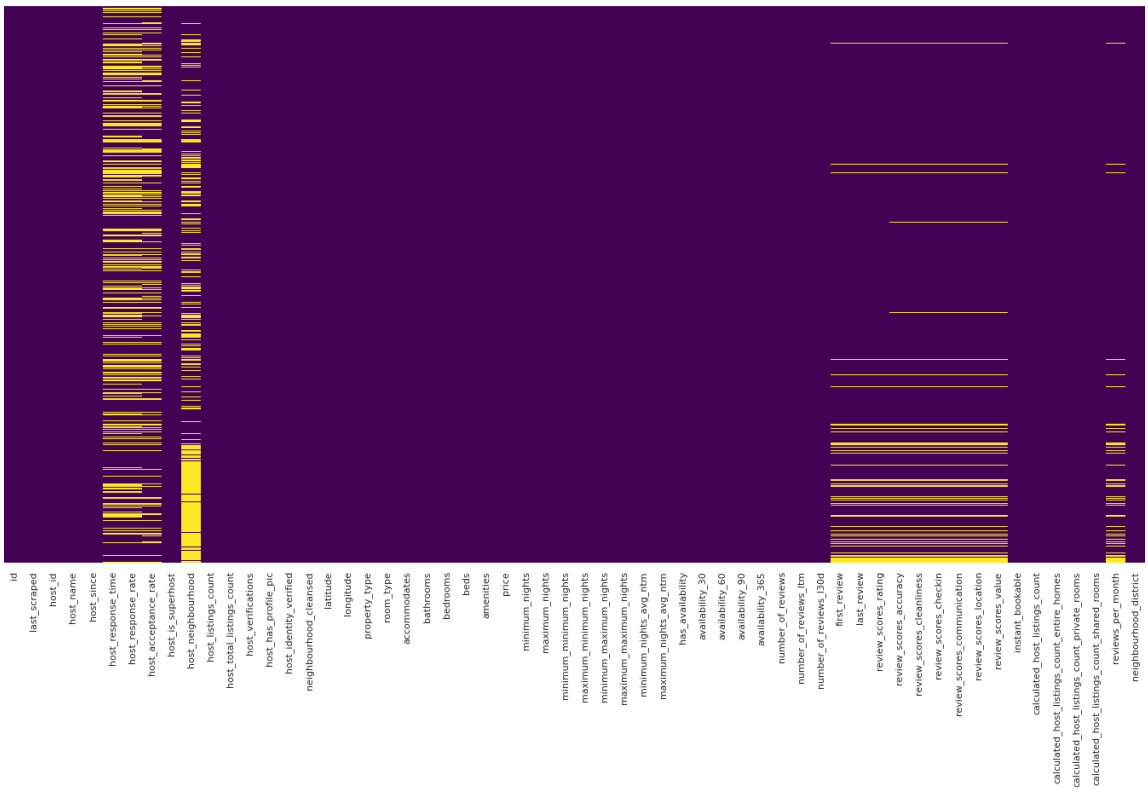
One can also observe that the bathrooms, bedrooms and beds feature (all discrete/categorical variables) have sparse null values in them, we can now just fill these null values with the mean value in order to have a full dataset where no entries have to be dropped. For features like host_neighborhood where there are alot on null values we can consider dropping the feature, but due to the fact that we are not training an ML agent it is not that important to have a very clean dataset. Maybe we can find something interesiting regarding missing host_neighborhood values!

In [ ]:
```
listings['bathrooms'] = listings['bathrooms'].fillna(round(listings['bath
listings['bedrooms'] = listings['bedrooms'].fillna(round(listings['bedroo
listings['beds'] = listings['beds'].fillna(round(listings['beds'].mean())
```

One more look on the clean data to see if we missed something.

In [ ]:
```
sns.heatmap(listings.isnull(), yticklabels=False, cbar=False, cmap='virid
```

Out[ ]: <AxesSubplot:>



Descriptive Statistics on individual Attributes

In [ ]:
```
# figure setup for following figures
sns.set(rc={'figure.figsize':boxplot_fig_size})
```

Price Feature

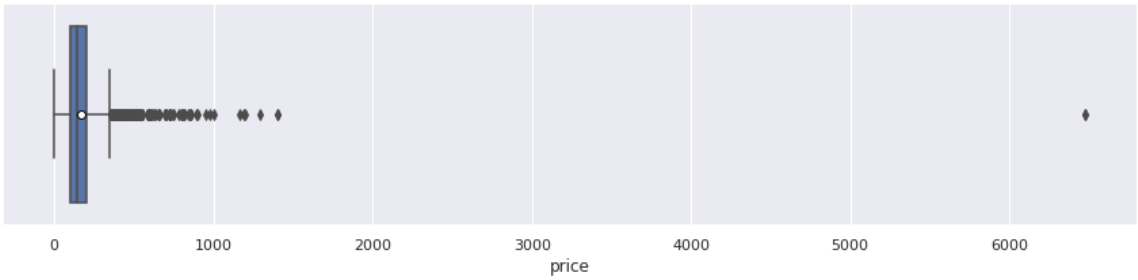
Here we will have a look at the *price* feature which is a continous feature.

Starting with a boxplot of the price to get a grip of the summary statistics of the price attribute, the mean is marked as a white dot.

In [ ]:
```
sns.boxplot(
    listings['price'],
    showmeans=True,
    meanprops={
        "marker":"o",
        "markerfacecolor":"white",
        "markeredgecolor":"black",
        "markersize":"6"
```

```
    }
)
```

Out[ ]: <AxesSubplot:xlabel='price'>



We see that there is one outlier that is siginificantly bigger then the rest, so our mean is affected by this to a certain extent that we will investigate later.

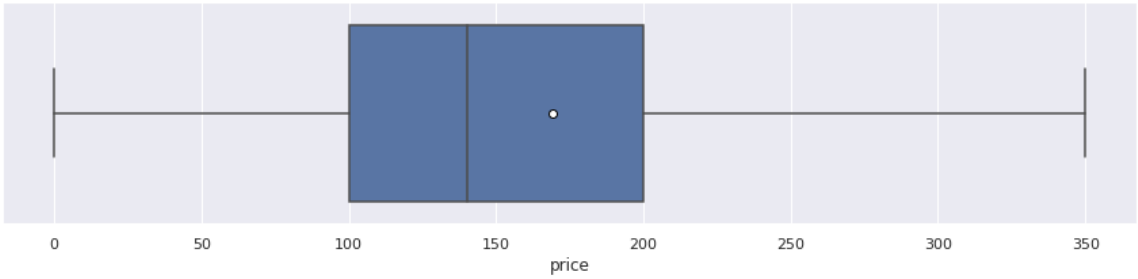
In [ ]:
```
# this has to be the biggest outlier
print(f"Biggest Outlier: {listings['price'].max()}")
```

```
Biggest Outlier: 6477.0
```

Inorder to get a better look at the boxplot, we plot it without the outliers.

In [ ]:
```
sns.boxplot(
    listings['price'],
    showfliers=False,
    showmeans=True,
    meanprops={
        "marker":"o",
        "markerfacecolor":"white",
        "markeredgecolor":"black",
        "markersize":"6"
    }
)
```

Out[ ]: <AxesSubplot:xlabel='price'>



We can immediately see that the mean is pulled to the right (higher values) of the median.

To show the summary statistics needed for the boxplot not in a plot but in raw numbers.

In [ ]:
```
_min = listings['price'].min()
q1 = listings['price'].quantile(0.25)
median = listings['price'].quantile(0.5)
q3 = listings['price'].quantile(0.75)
_max = listings['price'].max()
```

```
print(f"Minimum: {_min}")
print(f"Quartile 1: {q1}")
print(f"Quartile 2/Median: {median}")
print(f"Quartile 3: {q3}")
print(f"Maximum: {_max}")
```

```
Minimum: 0.0
Quartile 1: 100.0
Quartile 2/Median: 140.0
Quartile 3: 200.0
Maximum: 6477.0
```

We can already see from the boxplot that there is indeed a (or mulitple) listings with price 0, this is more or less an issue, so we'll have a short look into said listings/entries to determine if this effects more then one column and possibly replace these (preferable with the median, because the mean is so greatly effected by the big outlier, so it would not be representative).

In [ ]:
```
listings[listings['price'] == 0]
```

Out[ ]:

	id	last_scraped	host_id	host_name	host_since	host_response_time
4306	42430386	2021-11-04	308694260	Dutchies Hostel	2019-11-11	within an hour
4308	42431273	2021-11-04	309119467	Hotel Abba	2019-11-13	within an hour
4309	42431450	2021-11-04	311321908	Melrose Hotel	2019-11-22	within an hour
4366	43095925	2021-11-04	311323273	Acostar Hotel	2019-11-22	within an hour
4370	43148414	2021-11-04	318649852	Luxury Suites	2019-12-18	within an hour
4591	45478851	2021-11-04	367980459	The Delphi	2020-09-17	NaN
4716	47110768	2021-11-04	380676508	Sircle	2020-12-18	NaN

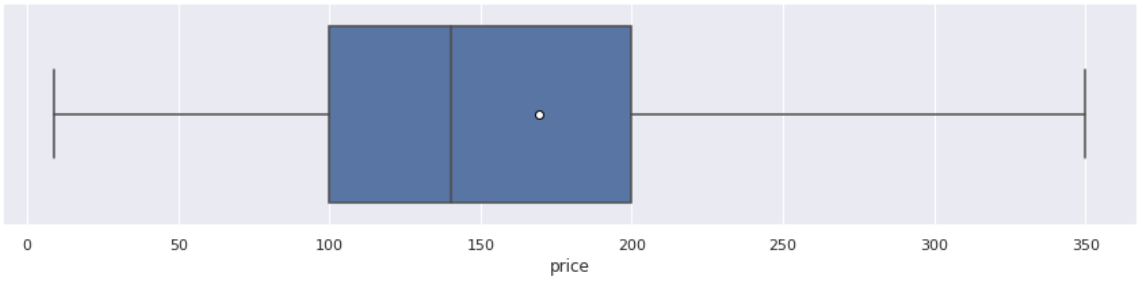
Due to these Datasets not having a lot of representation (7 entries out of 5402) we can delete them without having a loss of representation and have a better look at the price boxplot again. Further some of the data entries are very sparse populated and not useful for any other furter usage.

In [ ]:
```
listings = listings.drop(listings[listings['price'] == 0].index)
```

In [ ]:
```
sns.boxplot(
    listings['price'],
    showfliers=False,
    showmeans=True,
    meanprops={
        "marker":"o",
        "markerfacecolor":"white",
        "markeredgecolor":"black",
        "markersize":"6"
```

```
    }
)
```

Out[ ]: <AxesSubplot:xlabel='price'>



In [ ]:
```
_min = listings['price'].min()
q1 = listings['price'].quantile(0.25)
median = listings['price'].quantile(0.5)
q3 = listings['price'].quantile(0.75)
_max = listings['price'].max()

print(f"Minimum: {_min}")
print(f"Quartile 1: {q1}")
print(f"Quartile 2/Median: {median}")
print(f"Quartile 3: {q3}")
print(f"Maximum: {_max}")
print(f"Interquartile Range: {q3 - q1}")
```

```
Minimum: 9.0
Quartile 1: 100.0
Quartile 2/Median: 140.0
Quartile 3: 200.0
Maximum: 6477.0
Interquartile Range: 100.0
```

Now we have a realistic minimum of 9 (Euros). My guess for the 0 Euro price is that these rooms were from Hotels (as one can guess from the feature room_type) or the name of the listing. They probably wanted to promote the hotel rooms somehow and did not set a price for the listing to lure the user onto their website instead of booking the room via AirBnb. As also already described, the data in some of these listings is pretty sparse!

Distribution and Summary Statistics of price feature

In [ ]:
```
# figure setup for following figures
sns.set(rc={'figure.figsize':standard_fig_size})
```

Now a look into the distribution of the price feature.

In [ ]:
```
# helper
prices = listings.copy()['price']
```

Calculating Skewness, the measure of asymmetry (skew) of a normal or probability distribution, and Kurtosis, the shape, height and steepness of a normal or probability distribution.

In [ ]:
```
print(f"Skewness: {prices.skew()}")
print(f"Kurtosis: {prices.kurtosis()}")
```

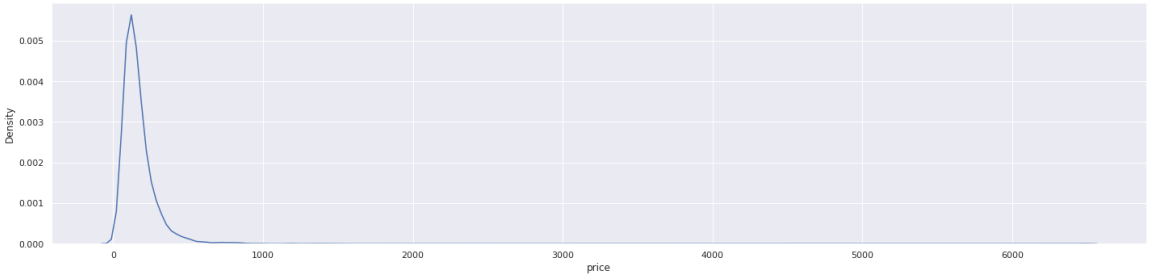
```
Skewness: 22.11819735315031
Kurtosis: 816.7769977189903
```

From the skewness value of 22.12 we can see that the data is skewed (really really) heavily to the left and from the kurotisis value of 816.78 one can conclude that the peak is very high and steep. Following these observations one can also conclude that the data is most likely (kind of) skewed normally distributed.

Plotting the probability distribution of the *price*.

In [ ]:
```
sns.displot(
    prices,
    kind='kde',
    height=5,
    aspect=4
)
```

Out[ ]:
```
<seaborn.axisgrid.FacetGrid at 0x7f362abaa9b0>
```



The fact that the price is a (kinda wobbly) skewed normally distribution is also backed by the plot of the data.

Calculating the mean, variance, median and mode so we can see how the data is effected, rather poluted, by outliers.

In [ ]:
```
print(f"Mean: {prices.mean()}")
print(f"Variance: {prices.var()}")
print(f"Median: {prices.median()}")
print(f"Mode: {prices.mode()[0]} (Value), {len(prices[prices == prices.mo
```

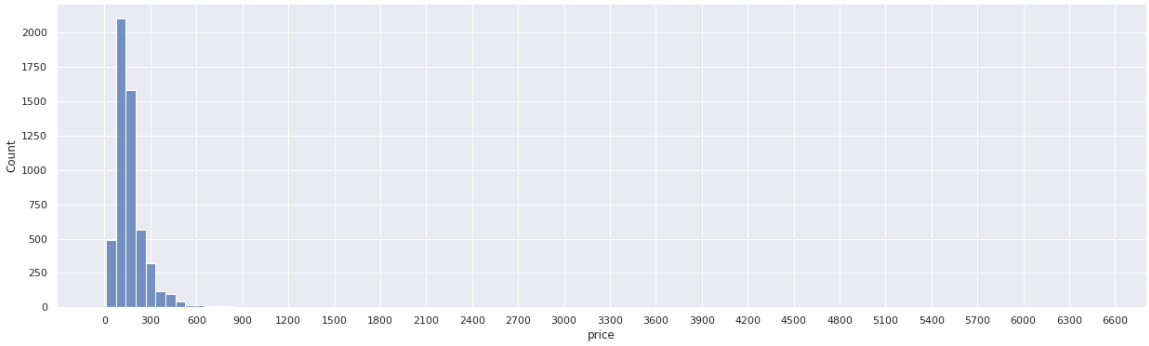
```
Mean: 169.22205746061167
Variance: 26846.766031731964
Median: 140.0
Mode: 150.0 (Value), 225 /  4.17% (Occurences)
```

Looking at the mean one can see a pretty representative value, altough it is biased by the rather big values (what one can kind of confirm by the noticeably smaller median). The variance suffers a little bit more from this, which also leads to the quick assumption that the data is pretty wide spread, but it is not what we can see from the distribution plot above, where the density peaks around the mean/median and therefore very low in the spectrum, which leads us to the conclusion that the data is indeed not very wide spread, despite the variance suggesting so, but rather concentrated at around 50 to 300. Looking at the mode we see that we have a value

of 150 and an occurence count of 225 / 4.17% which is rather a lot if one considers the price as an arbitrarly chooseable number in 5395 data entries.

In [ ]:
```
graph = sns.histplot(prices, bins=100)
_ = graph.set_xticks([300*i for i in range(100)])
graph.plot()
```

Out[ ]:
```
[]
```



This plot confirms the belief that the data is in fact mostly centered around 50 to 300.

Distribution and Summary Statistics of price feature w/o outliers

Now a look into the distribution of the price feature but only considering prices < 1000 to account for the majority, the really big, of outlier.

In [ ]:
```
prices = listings.copy()[listings['price'] < 1000]['price']
```

Again some summary statistics.

In [ ]:
```
print(f"Mean: {prices.mean()}")
print(f"Variance: {prices.var()}")
print(f"Median: {prices.median()}")
print(f"Mode: {prices.mode()[0]} (Value), {len(prices[prices == prices.mo
```

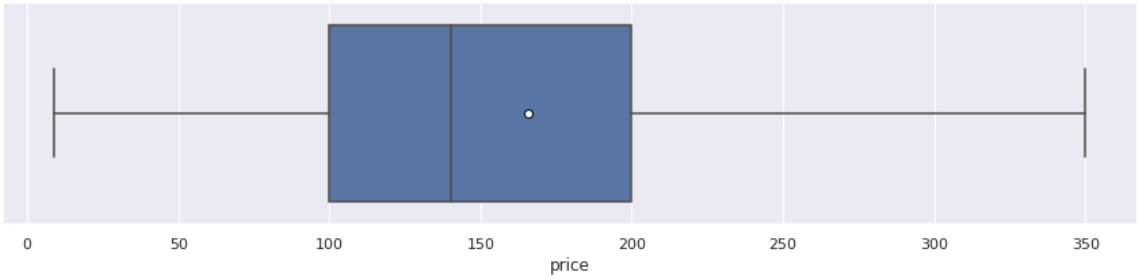
```
Mean: 165.6497122702803
Variance: 10728.90976736985
Median: 140.0
Mode: 150.0 (Value), 225 / 4.18% (Occurences)
```

The mean still is far away from the median, which tells us that there are still (much bigger) outliers left in the data, this spread is also backed by the still high variance (altough it dropped by half). Obviously the data is still concentrated around 50 to 300. Median and Mode stayed unchanged.

In [ ]:
```
fig, ax = plt.subplots(figsize=boxplot_fig_size)

sns.boxplot(
    prices,
    showfliers=False,
    showmeans=True,
    meanprops={
        "marker":"o",
        "markerfacecolor":"white",
```

```
        "markeredgecolor":"black",
        "markersize":"6"
    }
)
```

Out[ ]: <AxesSubplot:xlabel='price'>



The boxplot visually confirmes that the mean is still pulled to the right, but not as heavy as before.

Calculating Skewness and Kurtosis for Distribution.

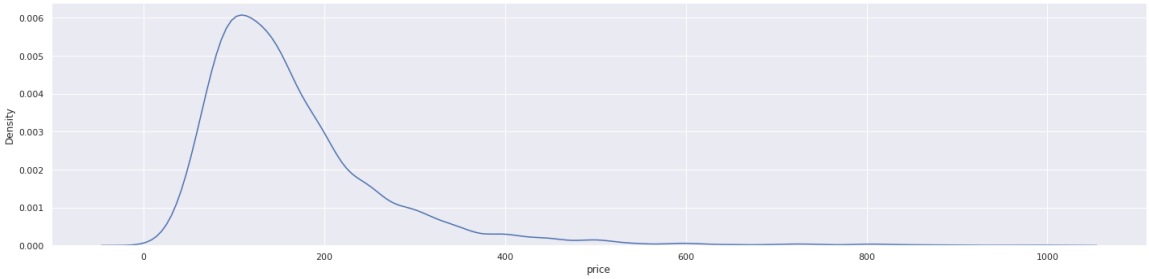
In [ ]:
```
print(f"Skewness: {prices.skew()}")
print(f"Kurtosis: {prices.kurtosis()}")
```

```
Skewness: 2.5069575758616547
Kurtosis: 10.222914563863492
```

One observes that all of a sudden the skewness and kurtosis drop significantly to a skewness value of 2.51 and a kurtosis value of 10.22. The data is still highly skewed to the right (value above +1) and the peak (kurtosis) is still high and steep, but both values are nearly not as extreme as before. Still one can conclude that the data is kinda skewed normally distributed, but now much nicer for sure.

In [ ]:
```
sns.displot(
    prices,
    kind='kde',
    height=5,
    aspect=4
)
```

Out[ ]: <seaborn.axisgrid.FacetGrid at 0x7f362a8c0400>

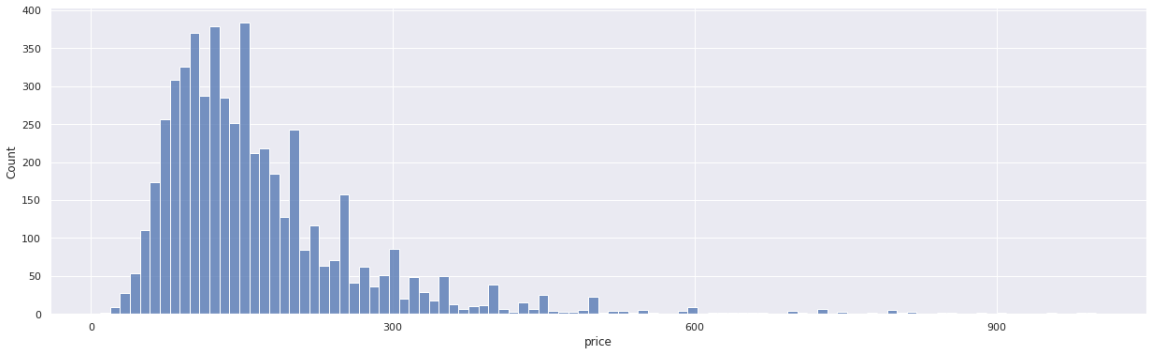


The distribution plot still shows that the *price* is a (much cleaner and not that extremely) skewed (kinda wobbly) normal distribution.

Calculating the mean, variance and median of the cleansed prices.

In [ ]:
```
graph = sns.histplot(prices, bins=100)
_ = graph.set_xticks([300*i for i in range(100)])
graph.plot()
```

Out[ ]:
```
[]
```



This plot confirms the belief that the data is in fact mostly centered around 50 to 300.

Interactive Distribution and Boxplot of price feature (upper bound and boxplot outliers)

Here we can play around with the upper bound of the price feature to plot a distribution plot and some summary statistics.

In [ ]:
```
from ipywidgets import interact, interactive, fixed, interact_manual
import ipywidgets as widgets
from ipywidgets import IntSlider

@interact(
    cutoff=IntSlider(
        min=0,
        max=listings['price'].max() + 10,
        step=10,
        description="Price upper bound: ",
        continous_update=True
    ),
    outliers=widgets.Checkbox(
        value=True,
        description='Show outliers in Boxplot',
        disabled=False,
        indent=False
    )
)
def _(cutoff, outliers):
    prices = listings[listings['price'] < cutoff]['price']

    if prices.count() != 0:
        occurences = prices[prices == prices.mode()[0]].count()
        mode = prices.mode()[0]
    else:
        occurences = 'nan'
        mode = 'nan'

    print('-----------------------')
    print(f"| Mean: {prices.mean():.3f}")
    print(f"| Variance: {prices.var():.3f}")
```

```
    print(f"| Median: {prices.median():.3f}")
    print(f"| Mode: {mode} (Value), {occurences} (Occurences)")
    print('-----------------------')
    print(f"| Skewness: {prices.skew():.3f}")
    print(f"| Kurtosis: {prices.kurtosis():.3f}")
    print('-----------------------')

    sns.displot(
        prices,
        kind='kde',
        height=5,
        aspect=4
    )
    plt.show()

    sns.boxplot(
        prices,
        showfliers=outliers,
        showmeans=True,
        meanprops={
        "marker":"o",
        "markerfacecolor":"white",
        "markeredgecolor":"black",
        "markersize":"6"
        }
    )
    plt.show()
```

```
interactive(children=(IntSlider(value=0, description='Price upper bound:
', max=6487, step=10), Checkbox(value…
```

Bathrooms/Bedrooms/Beds Features

All three of these features (bathrooms, bedrooms, beds) are basically Categorical/Discrete features.

In [ ]:
```
# figure setup for following figures
sns.set(rc={'figure.figsize':countplot_fig_size})
```

The following are some helper functions to make everything nicer.

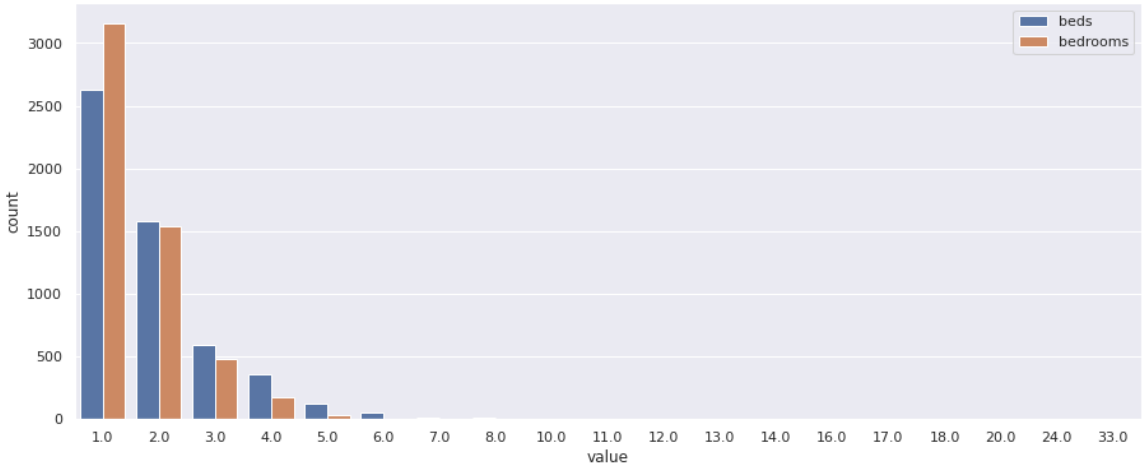
In [ ]:
```
"""
    Plotting a countplot for given feature.
      - Ascending has to be set to False in case of a string feature
"""
def countplot(keyword: str, ascending=True):

    graph = sns.countplot(
        x=keyword,
        data=listings
    )

    counts = listings[keyword].value_counts().sort_index(ascending=ascend

    for i, p in enumerate(graph.patches):
        height = p.get_height()
        graph.text(p.get_x() + p.get_width() / 2., height + 0.1, f"{(coun
```

```
"""
    Plotting a countplot for given (multiple) features in one plot.
"""
def countplot_multiple(keywords: [str], ascending=True):
    graph = sns.countplot(
        x='value',
        hue='variable',
        data=pd.melt(listings[keywords])
    )
    graph.legend(loc='upper right')
```

Next are some countplots inorder to get a grip of the distribution of the categorical/discrete features of bathrooms, bedrooms and beds using the occurence counts of the individual categorical/discrete values.

Due to the fact that bed and bedrooms share the same scale one can draw them togehter in a single countplot.

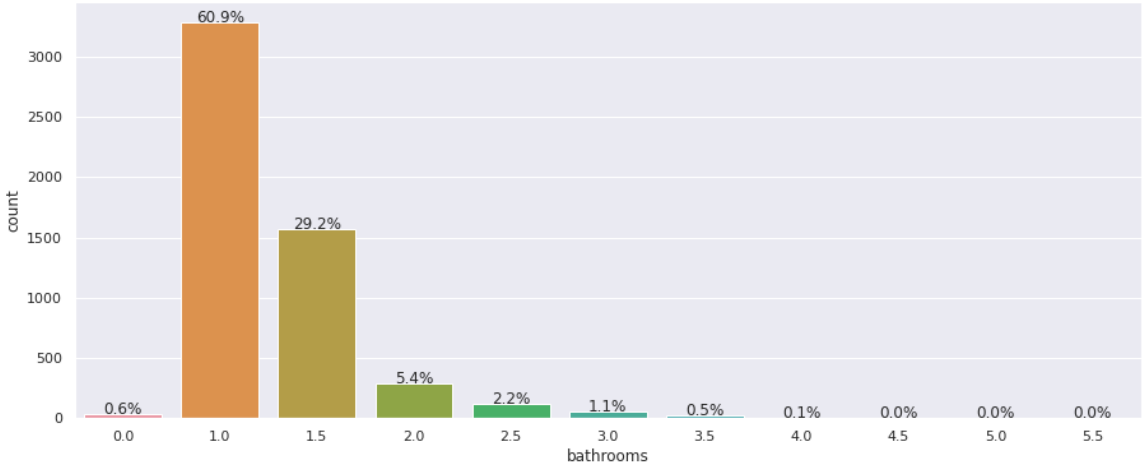
In [ ]:
```
countplot_multiple(['beds', 'bedrooms'])
```



This plot does not hold that much statistical information, rather information about their correlation. To briefly sum it up: They (obviously) correlate with each other, altough there are a lot more single bedroom apartments then apartmnts with a single bed, i'll leave it at that, the point of this section is not to find correlations, thus the next plots are also countplots with a percentage.

Countplot of *bathrooms*.

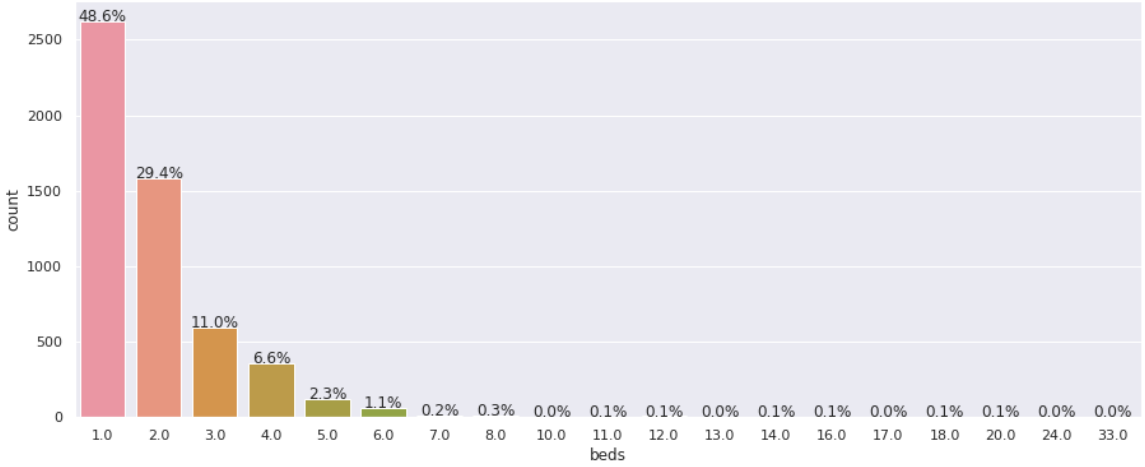
In [ ]:
```
countplot('bathrooms')
```



It was kinda predictable that the most listings (60.9%) would be single bedrooms, because even in a three bedroom apartment it is ethically reasonable to have a single bathroom. What is interesting is that 0.6% of the listings have 0 bathrooms, this could be just 0 values, so the host didn't bother with putting the info in or there is some kind of shared bathroom stuff going on, which is rather unlikely. Besides that the amount of listings (except for 0 bathrooms) drops off exponentially with encreasing steps/bathrooms.

Countplot of *beds*.

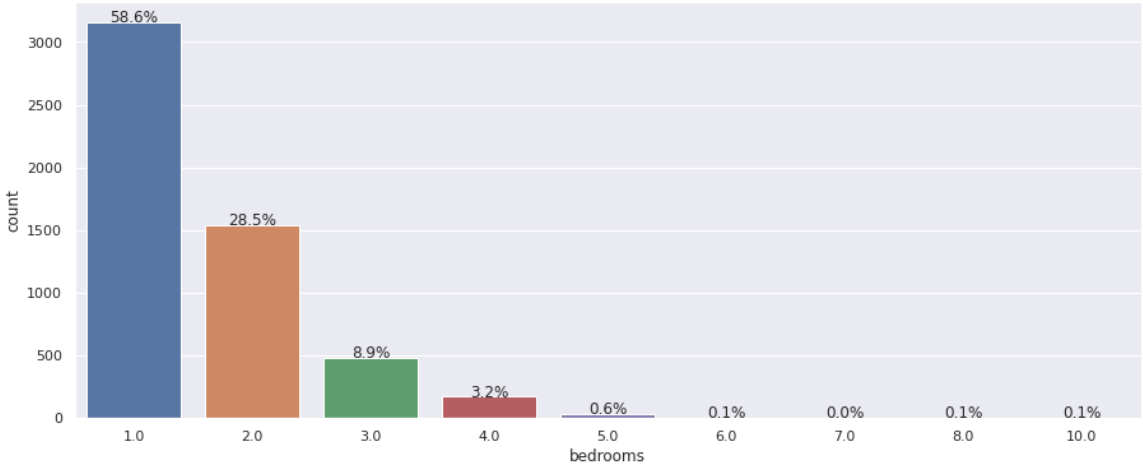
In [ ]:
```
countplot('beds')
```



Again nearly about half (48.6%) of the listings are single bed apartments, after that again the count/percentage is dropping off exponentially. Remark: there are apartemnts with e.g 24 or 33 beds, but due to rounding and displaying the plot nicely the percentage is 0.0% and obviously the bar is not visible.

Countplot of *beds*.

In [ ]:
```
countplot('bedrooms')
```



More than half (58.6%) of the listings are single bedroom apartments, after that again the count/percentage is dropping off exponentially. Remark: there are apartemnts with e.g 7 bedrooms, but due to rounding and displaying the plot nicely the percentage is 0.0% and obviously the bar is not visible.

Response Times Feature

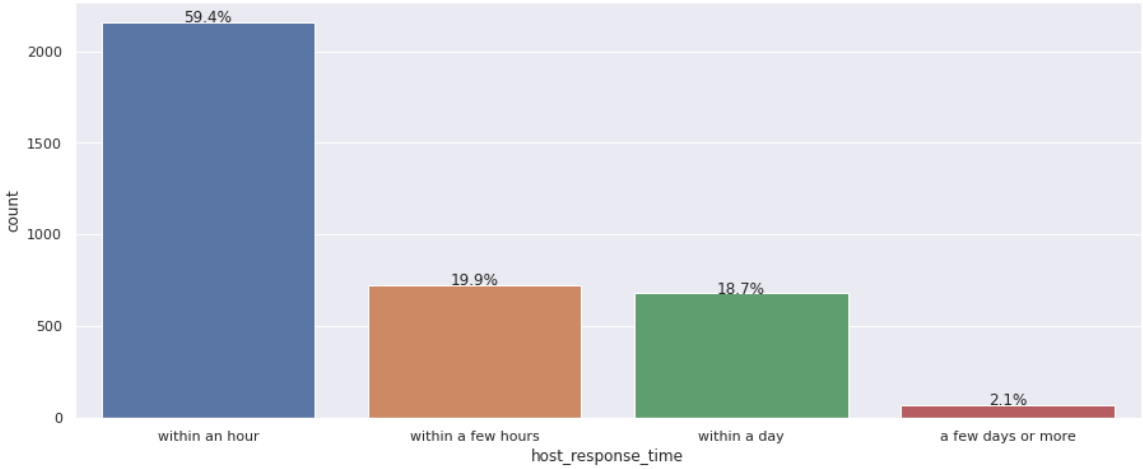
Short look at the categroical feature of response times.

From the heatmap seen in section one, one can observe that here one finds a lot of missing values, just as a remark.

In [ ]:
```
# figure setup for following figures
sns.set(rc={'figure.figsize':countplot_fig_size})
```

Again a countplot is the plot to go for categorical features like *host_response_time*.

In [ ]:
```
countplot('host_response_time', ascending=False)
```



One can see that in 59.4% of the time the host resonds "within an hour", "responding within a few hours" and "within a day" are with 19.9% and 18.7% respectively pretty close, this might be due to the classification process e.g. the mean of response times falls into the category "within a few hours" between 2 and 5 hours as *host_response_time*. Further the subjective time difference between immediately and 1 hour is not that much less then e.g. between 2 hours annd 5 hours,

in the latter the time streches much longer from an absolute point of view, but in a subjective one this could be, at least for some, not the case.

Reviews Features

All the features this is referring to are continous features.

These Features all analyzed in this section span:

- review_scores_rating
- review_scores_accuracy
- review_scores_cleanliness
- review_scores_checkin
- review_scores_communication
- review_scores_location
- review_scores_value

Here again one can observe a lot of missing values seen in the boxplot in section 1.

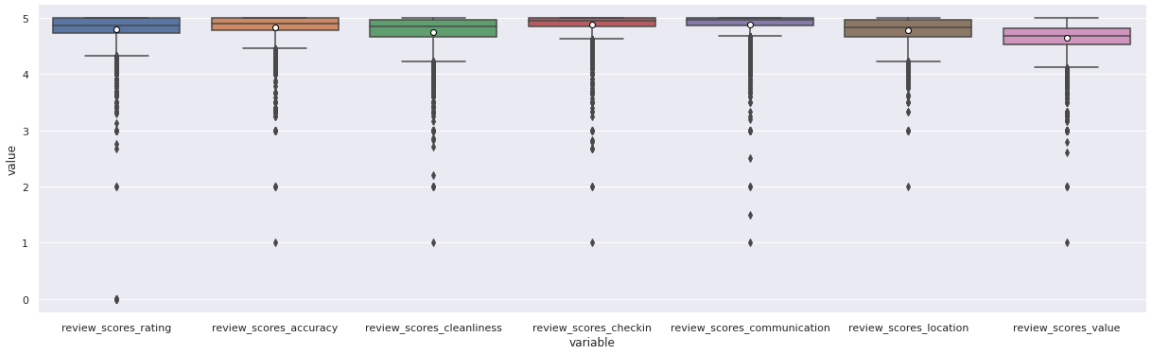
In [ ]:
```
# figure setup for following figures
sns.set(rc={'figure.figsize':standard_fig_size})
```

Helper function and helper variable to make everything nicer and easier.

In [ ]:
```
# a variable for convenience
rating_features = [
    'review_scores_rating',
    'review_scores_accuracy',
    'review_scores_cleanliness',
    'review_scores_checkin',
    'review_scores_communication',
    'review_scores_location',
    'review_scores_value'
]

"""
    Plotting a boxplot for given (multiple) features in one plot.
"""
def boxplot_multiple(keywords: [str], outliers=True):
    graph = sns.boxplot(
        y='value',
        x='variable',
        data=pd.melt(listings[keywords]),
        showfliers=outliers,
        showmeans=True,
        meanprops={
            "marker":"o",
            "markerfacecolor":"white",
            "markeredgecolor":"black",
            "markersize":"6"
        }
    )
```

First looking at the continous data as a boxplot, again here with the mean as a white dot.

In [ ]:
```
boxplot_multiple(rating_features)
```



One can only observe that all the different review features are concentrated in the top values (4 to 5), looking at the boxplots without outliers one might get a better look.

In [ ]:
```
boxplot_multiple(rating_features, outliers=False)
```

Here one can see clearly now that the outliers pull the mean below the median, some stronger then others, for example the mean of *review_scores_value* is further at the median as for *review_scores_cleanliness*, 4 out of the 7 have their maximum value as quartile1, 2 of theses 4 have a very high median, but a mean close to quartile2 and therefore the data is concentrated at the very very top. The boxplots for *review_scores_cleanliness* and *review_scores_location* here look pretty equal except for the mean, which is higher in the *review_scores_location*, which is due to the outliers one can observe in the plot above. The only boxplot beeing different then all the others is the one for *review_scores_value* with generally way lower values.

Calculating the skewness and kurtosis of the features, maybe the data is somehow normally distributed data.

In [ ]:
```
for rf in rating_features:
    print(f"{rf}:")
    print(f"    Skewness: {listings[rf].skew()}")
    print(f"    Kurtosis: {listings[rf].kurtosis()}")
    print(f"    Mean: {listings[rf].mean()}")
    print(f"    Variance: {listings[rf].var()}")
    print(f"    Median: {listings[rf].median()}")
```

```
    print(f"    Mode: {listings[rf].mode()[0]} (Value), {len(listings[lis
    print()
```

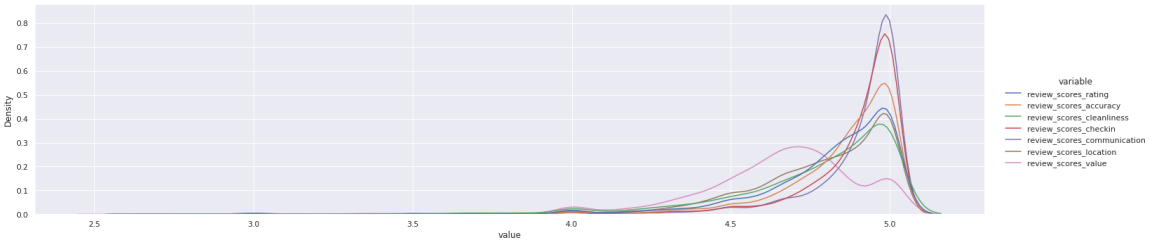
```
review_scores_rating:
    Skewness: -7.654542987675809
    Kurtosis: 94.32220773423532
    Mean: 4.793984992902048
    Variance: 0.11372008009928493
    Median: 4.87
    Mode: 5.0 (Value), 1254 / 25.43% (Occurences)

review_scores_accuracy:
    Skewness: -4.514907584512385
    Kurtosis: 38.09214152382168
    Mean: 4.837973577235773
    Variance: 0.056222097289091416
    Median: 4.9
    Mode: 5.0 (Value), 1386 / 28.17% (Occurences)

review_scores_cleanliness:
    Skewness: -2.9517704398764764
    Kurtosis: 15.586312617733448
    Mean: 4.754432926829268
    Variance: 0.09713108718557707
    Median: 4.85
    Mode: 5.0 (Value), 1117 / 22.70% (Occurences)

review_scores_checkin:
    Skewness: -5.567780072654971
    Kurtosis: 52.975528108514226
    Mean: 4.878662601626016
    Variance: 0.048626468777942515
    Median: 4.95
    Mode: 5.0 (Value), 1740 / 35.37% (Occurences)

review_scores_communication:
    Skewness: -5.704993366735886
    Kurtosis: 56.083778758355955
    Mean: 4.885837398373984
    Variance: 0.05069468386230925
    Median: 4.96
    Mode: 5.0 (Value), 1987 / 40.39% (Occurences)

review_scores_location:
    Skewness: -2.065602113093645
    Kurtosis: 9.389020793239467
    Mean: 4.78174593495935
    Variance: 0.05527312511053043
    Median: 4.84
    Mode: 5.0 (Value), 1107 / 22.50% (Occurences)

review_scores_value:
    Skewness: -2.49868420439041
    Kurtosis: 14.73358308405459
    Mean: 4.6427906504065035
    Variance: 0.0841288441492008
    Median: 4.68
    Mode: 5.0 (Value), 589 / 11.97% (Occurences)
```

These summary statistics, skew and kurtosis do not let one observe something out of the ordinary. The review features generally have a rather low variance, meaning that the data is closely distributed around the mean. The review features also have a general negative skewness to variying degrees, meaning all the feature's distributions are skewed to the right, altough the kurtosis varies more then the skewness, meaning some distributions are lower and flatter then others. Altough what is interesting is that the mode is a very common (in terms of amount) value and the maximum value, for example the mode of 5.0 is 40.39% of the values for *review_scores_communication*.

Distribution Plot of all the review features.

In [ ]:
```
sns.displot(
    x='value',
    hue='variable',
    data=pd.melt(listings[listings[rating_features] > 2.5][rating_feature
    kind='kde',
    height=5,
    aspect=4
)
```

Out[ ]:
```
<seaborn.axisgrid.FacetGrid at 0x7f362050dc30>
```

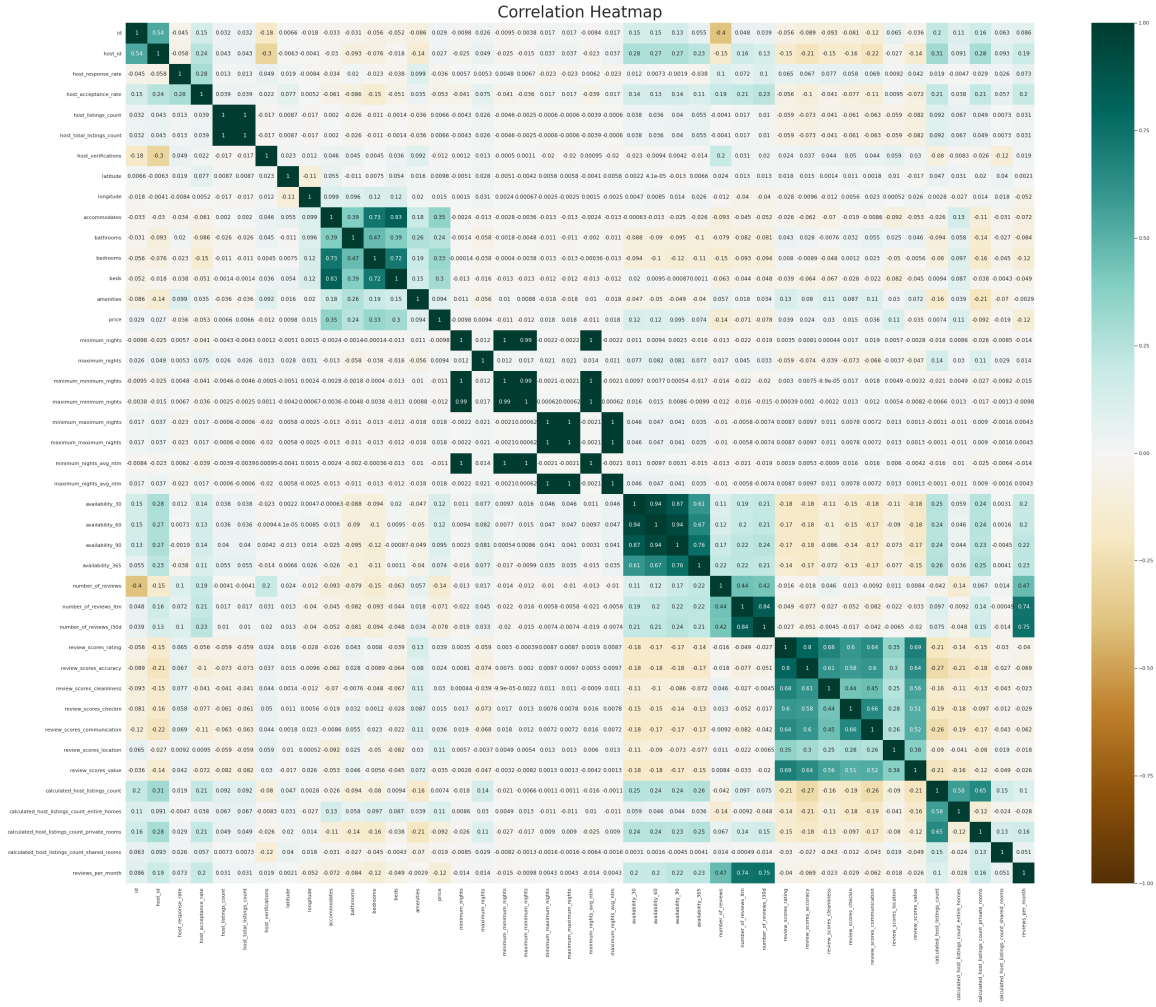


One can again see what was prophesied by the skewness and kurtosis. The Distributions commly look very woobly with a small hill at 4.0. The one that already stood out in the boxplots, *review_scores_value*, has a stranger distribution then the rest with its peak at around 4.7 and a small hill at 5.0. Looking at *review_scores_checkin* one can see that it is way less wobbly then the rest and peaking second highest at nearly 5.0. The feature *review_scores_communication* peaks highest, also its boxplot looked pretty one sided.

Correlations between Attributes

To get an overview of how the attributes in the dataset are correlated, a correlation heatmap might be of great help.

In [ ]:
```
# set figure size
plt.figure(figsize=(45, 35))
# define plot with appropriate colormap
heatmap = sns.heatmap(listings.corr(), vmin=-1, vmax=1, annot=True, cmap=
# set title of heatmap
heatmap.set_title('Correlation Heatmap', fontdict={'fontsize':35}, pad=12
```



Obvious relationships can already be seen for the different types of reviews, furnishing and listing counts (greenish blocks). Correlations with higher positive and negative values seem to be appropriate to focus on (excluding values close to *-1* or *1* as they are mostly obvious).

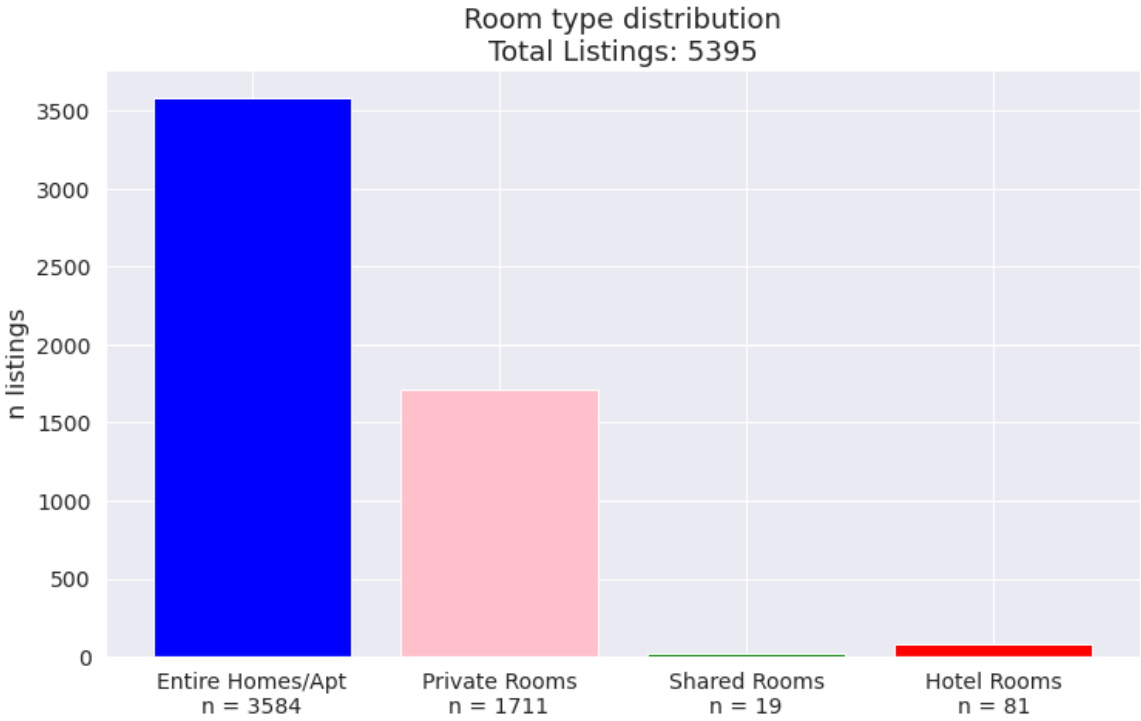
Correlations of Neighbourhood / Room-type Distributions / Rating / Price

For now, the focus will be on correlations between the different *neighbourhoods* and the corresponding *location-ratings*, *prices* and *room-type* distributions. Taking a look into the distribution of the *neighbourhoods* and *room-types* in general will give a good intuition for further correlations and/or findings.

(short introduction) Descriptive statistics

In [ ]:
```
# Count and print room_types
total_all = len(listings.index)
homes_all = len(listings[listings['room_type']=='Entire home/apt'].index)
private_rooms_all = len(listings[listings['room_type']=='Private room'].i
shared_rooms_all = len(listings[listings['room_type']=='Shared room'].ind
hotel_rooms_all = len(listings[listings['room_type']=='Hotel room'].index

#print(f'Total listings: {total_all}\nHomes: {homes_all}\nPrivate rooms:
# Plotting room type distribution
_ = plt.figure(figsize=(12, 7))
room_type_distribution = plt.bar([f"Entire Homes/Apt\nn = {homes_all}", f
```

```
_ = plt.ylabel("n listings", fontsize = 16)
_ = plt.title(f'Room type distribution\nTotal Listings: {total_all}', fon
_ = plt.xticks(fontsize=14)
_ = plt.yticks(fontsize=14)
```

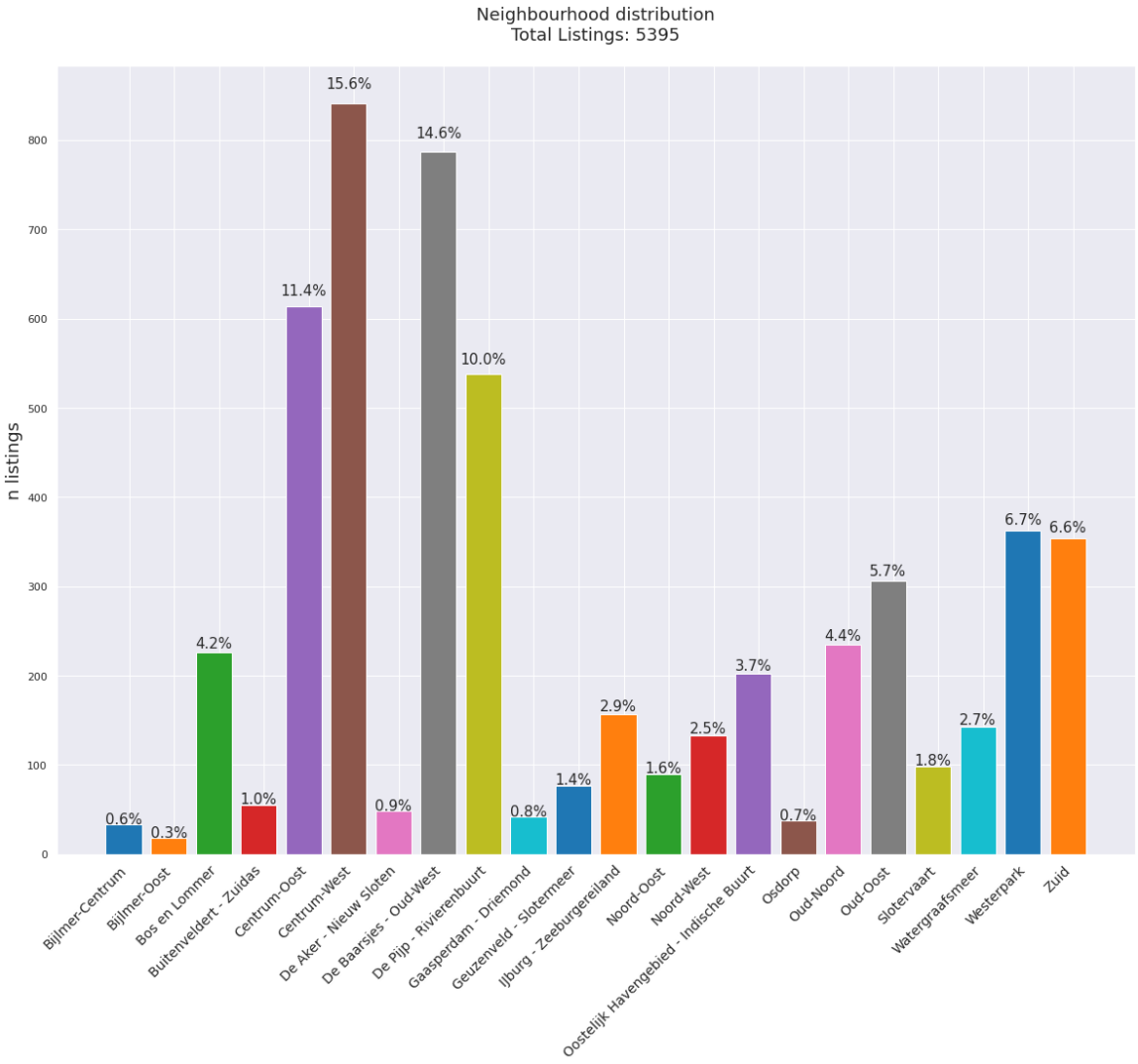


More than half of the listings are **entire homes/apartments**. **Private rooms** are about 1/3 of the listings. Interestingly, **shared rooms** and **hotel rooms** make up only a really small amount in listings.

In [ ]:
```
## Get neighbourhood counts
listings_neighbourhood = listings["neighbourhood_cleansed"].groupby(listi
for index, neighb in enumerate(listings_neighbourhood.index):
    ## better formatting
    if len(neighb) == 6:
        tabs = "\t\t\t\t\t\t"
    elif len(neighb) == 4:
        tabs = "\t\t\t\t\t\t"
    elif len(neighb) < 15:
        tabs = "\t\t\t\t\t"
    elif len(neighb) < 23:
        tabs = "\t\t\t\t"
    elif len(neighb) > 30:
        tabs = "\t\t"
    else:
        tabs = "\t\t\t"
    #print(f'{neighb}:{tabs}{listings_neighbourhood[index]}')
#print()
#print(f'Total amount of listings: {listings_neighbourhood.sum()}')

def plot_nd():
    # Figsize
    _ = plt.figure(figsize=(20, 15))

    # Get colormap
    cmap = plt.get_cmap("tab20")
    color = cmap([i*2 if i < 11 else i*2 + 1 for i in range(10)])
```

```
    ## Plotting
    neighbourhood_distribution = plt.bar(listings_neighbourhood.index, li

    ## Title and Labels
    _ = plt.ylabel("n listings", fontsize = 18)
    _ = plt.title(f'Neighbourhood distribution\nTotal Listings: {total_al

    for p in neighbourhood_distribution.patches:
        width = p.get_width()
        height = p.get_height()
        percentage = height*100/total_all
        x, y = p.get_xy()
        plt.annotate(f'{percentage:.1f}%', (x + width/2, y + height*1.02)

    # Rotate labels to be readable
    xticks_pos = [0.65*patch.get_width() + patch.get_xy()[0] for patch in
    _ = plt.xticks(xticks_pos, listings_neighbourhood.index,  ha='right',

    return neighbourhood_distribution, color
```

In [ ]:
```
neighbourhood_distribution, color = plot_nd()
```



For the *neighbourhoods* one can clearly see a trend where most of the listings are located at. Namely **Centrum-Oost**, **Centrum-West**, **De Baarsjes - Oud-West** and **De Pijp - Rivierenbuurt** have the highest amount of listings. Some striking low

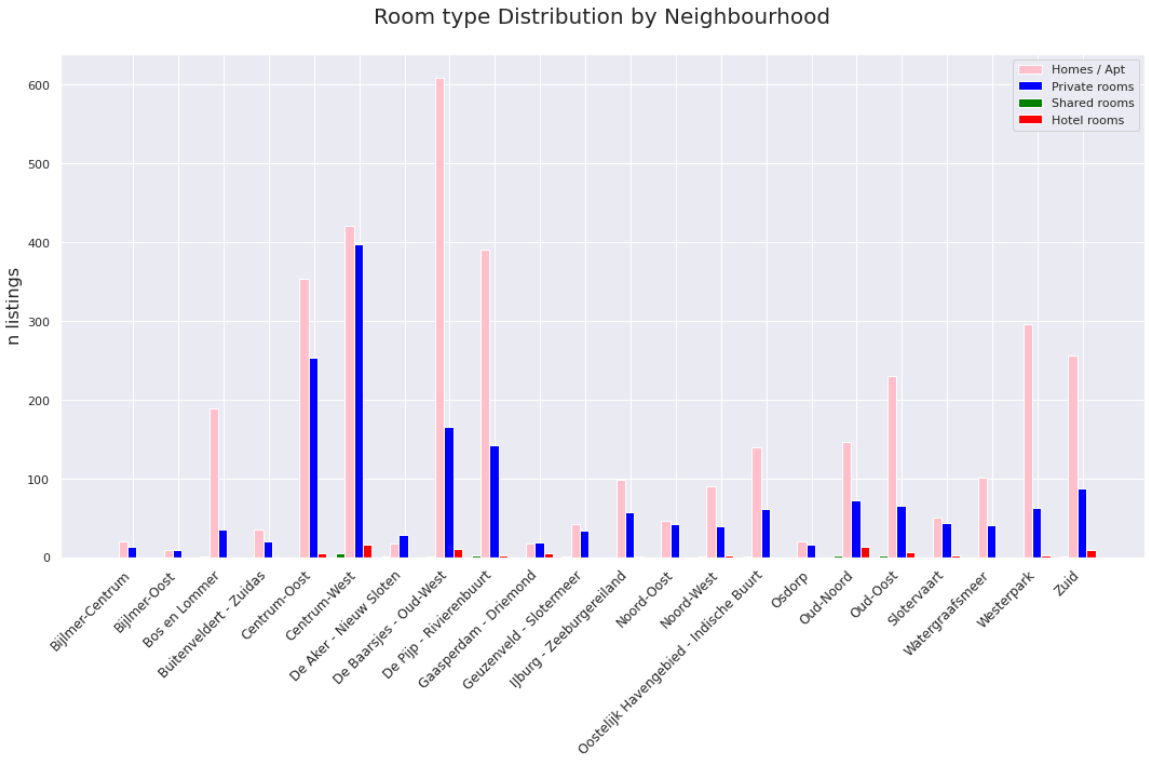
numbers of listings are recognizable in some few neighbourhoods. These might be interesting to investigate further for corresponding *ratings* and *prices*.

Neighbourhood / Room-type distributions

The next chapter focuses on the *room-type* distributions for each neighbourhood, visualized in a respective histogram.

In [ ]:
```
# Get individual neighbourhood-data

total = []
homes = []
private_rooms = []
shared_rooms = []
hotel_rooms = []

for i in listings_neighbourhood.index:
    listings_current_neighbourhood = listings[listings["neighbourhood_cle

    total.append(len(listings_current_neighbourhood.index))
    homes.append(len(listings_current_neighbourhood[listings_current_neig
    private_rooms.append(len(listings_current_neighbourhood[listings_curr
    shared_rooms.append(len(listings_current_neighbourhood[listings_curre
    hotel_rooms.append(len(listings_current_neighbourhood[listings_curren


labels = listings_neighbourhood.index

### Bar plot
# figsize
plt.rcParams["figure.figsize"] = (15,10)

# the label locations
x = np.arange(len(labels))
# the width of the bars
width = 0.20

# 4 different subplots for room types
fig, ax = plt.subplots()
rects1 = ax.bar(x - width/2, homes, width, label='Homes / Apt', color=['p
rects2 = ax.bar(x + width/2, private_rooms, width, label='Private rooms',
rects3 = ax.bar(x - width - width/2, shared_rooms, width, label='Shared r
rects4 = ax.bar(x + width + width/2, hotel_rooms, width, label='Hotel roo

# Add some text for labels, title and custom x-axis tick labels, etc.
ax.set_ylabel('n listings', fontsize=16)
ax.set_title(f'Room type Distribution by Neighbourhood\n', fontsize = 20)
ax.legend()

# Rotate labels to be readable
xticks_pos = [0.65*patch.get_width() + patch.get_xy()[0] for patch in nei
_ = plt.xticks(xticks_pos, listings_neighbourhood.index,  ha='right', rot

# Set layout
_ = fig.tight_layout()

plt.show()
```

```
# reset figsize for further plots
plt.rcParams["figure.figsize"] = plt.rcParamsDefault["figure.figsize"]
```



The highest amount of homes is found in **De Baarsjes - Oud-West**, whereas **Centrum-West** has the highest number of private rooms. Notably, even though both neighbourhoods have a pretty similar amount of listings, their room type distribution is very different.

When only focusing on *neighbourhoods* with more than 300 listings, **Centrum-West** and **Centrum West** have by far the highest distributions of **private rooms**.

To see if there are any clear correlations and to get a good overview, the next two chapters will visualize neighbourhood ratings and price differences.

Neighbourhood / Rating

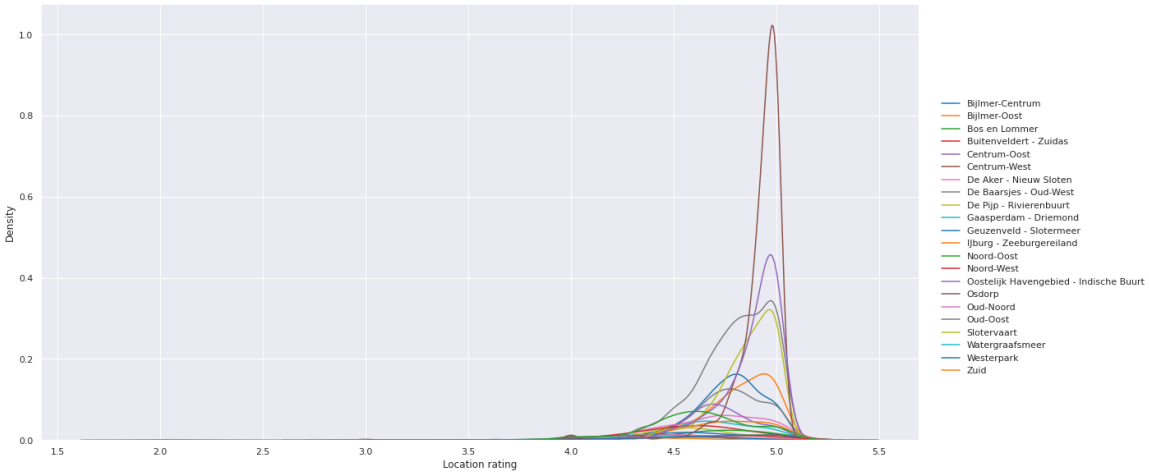
As there are a few entries with price values equal to zero and some few entries with a notably high price, dropping corresponding outliers for the following plots gives a better picture.

In [ ]:
```
# get individual neighbourhood data for plotting

neighbourhood_ratings = []
neighbourhood_prices = []


for i in listings_neighbourhood.index:
    listings_current_neighbourhood = listings[listings["neighbourhood_cle

    neighbourhood_prices.append(pd.DataFrame(listings_current_neighbourho
    # drop entries where price == 0 or > 1000
    neighbourhood_prices[-1] = neighbourhood_prices[-1][neighbourhood_pri
    neighbourhood_prices[-1] = neighbourhood_prices[-1][neighbourhood_pri
```

```
    neighbourhood_ratings.append(pd.DataFrame(listings_current_neighbourh

neighbourhood_prices_df = pd.concat(neighbourhood_prices, ignore_index=Tr
neighbourhood_ratings_df = pd.concat(neighbourhood_ratings, ignore_index=
```

To directly compare the rating-distribution over all neighbourhoods, a density-plot seems to be a good start.

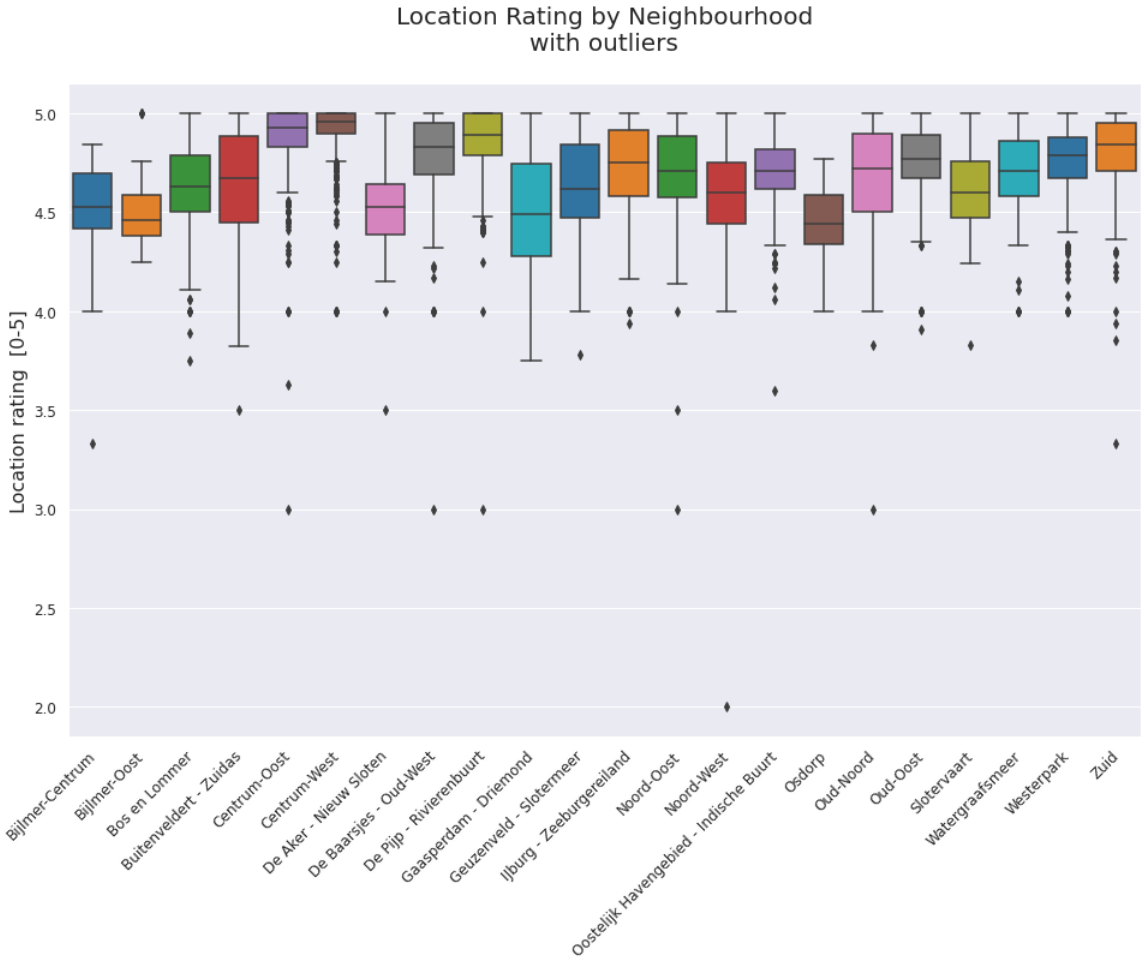
In [ ]:
```
# Plotting
plot = sns.displot(
    data=neighbourhood_ratings_df,
    kind='kde',
    height=8,
    aspect=2,
    palette=color
)
_ = sns.set(font_scale = 2)
_ = plot.set(xlabel='Location rating')
```



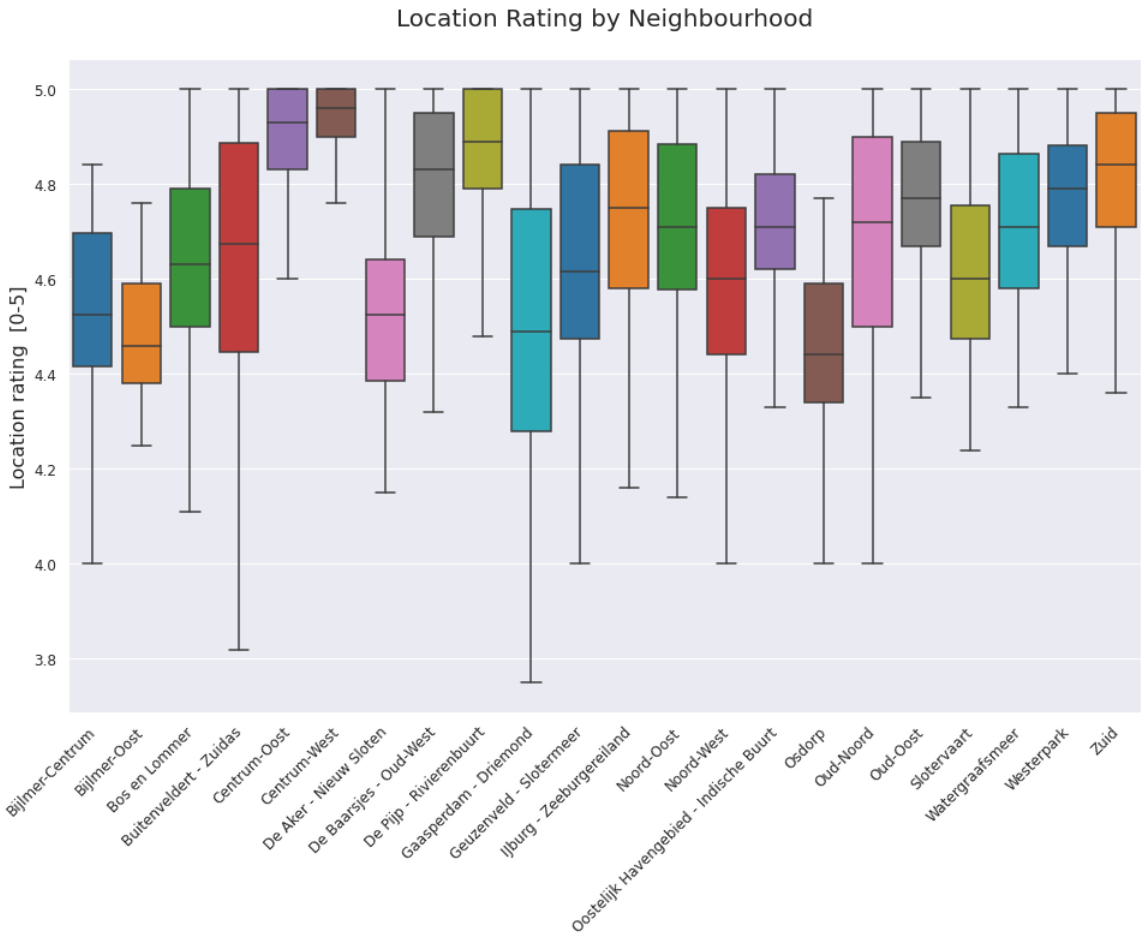
The density plot already highlights 4 different neighbourhoods which have the majority of their *ratings* mostly centered around 5. A further boxplot will let one compare the different areas much easier.

In [ ]:
```
def plot6(outliers = False):
    # Plotting
    plt.rcParams["figure.figsize"] = (16,10)

    graph = sns.boxplot(
        showfliers=outliers,
        data=neighbourhood_ratings_df,
        palette=color)

    if outliers:
        graph.axes.set_title("Location Rating by Neighbourhood\nwith outl
    else:
        graph.axes.set_title("Location Rating by Neighbourhood\n",fontsiz
    graph.set_ylabel("Location rating  [0-5]",fontsize=16)
    graph.tick_params(labelsize=12)

    # Rotate labels to be readable
    xticks_pos = [0.65*patch.get_width() + patch.get_xy()[0] for patch in
    _ = plt.xticks(xticks_pos, listings_neighbourhood.index,  ha='right',
```

```
    _ = plt.rcParams["figure.figsize"] = plt.rcParamsDefault["figure.figs

plot6(True)
```



In [ ]:
```
plot6(False)
```



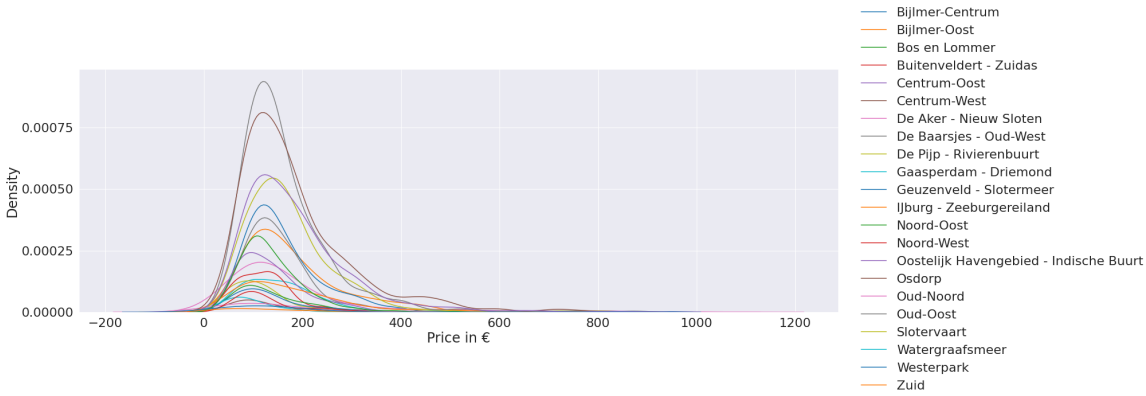
Clearly, **Centrum-Oost** and **Centrum-West** are the highest rated locations. This also makes sense when looking at the amount of listings in the corresponding areas. Especially the latter, **Centrum-West**, seems to be the highest rating neighbourhood. **De Pijp - Rivierenbuurt** is worth mentioning as well.

Furthermore, the ratings for **Gaasperdam-Driemond** vary the most - it has the lowest boxplot-minimum while also having its maximum at 5. Remarkably, the lowest median values are found in **Bijlmer-Oost** and **Osdorp**. It may be noted that these three neighbourhoods only have a small amount of listings available, which certainly could be the reason for their extraordinary numbers.

Neighbourhood / Price

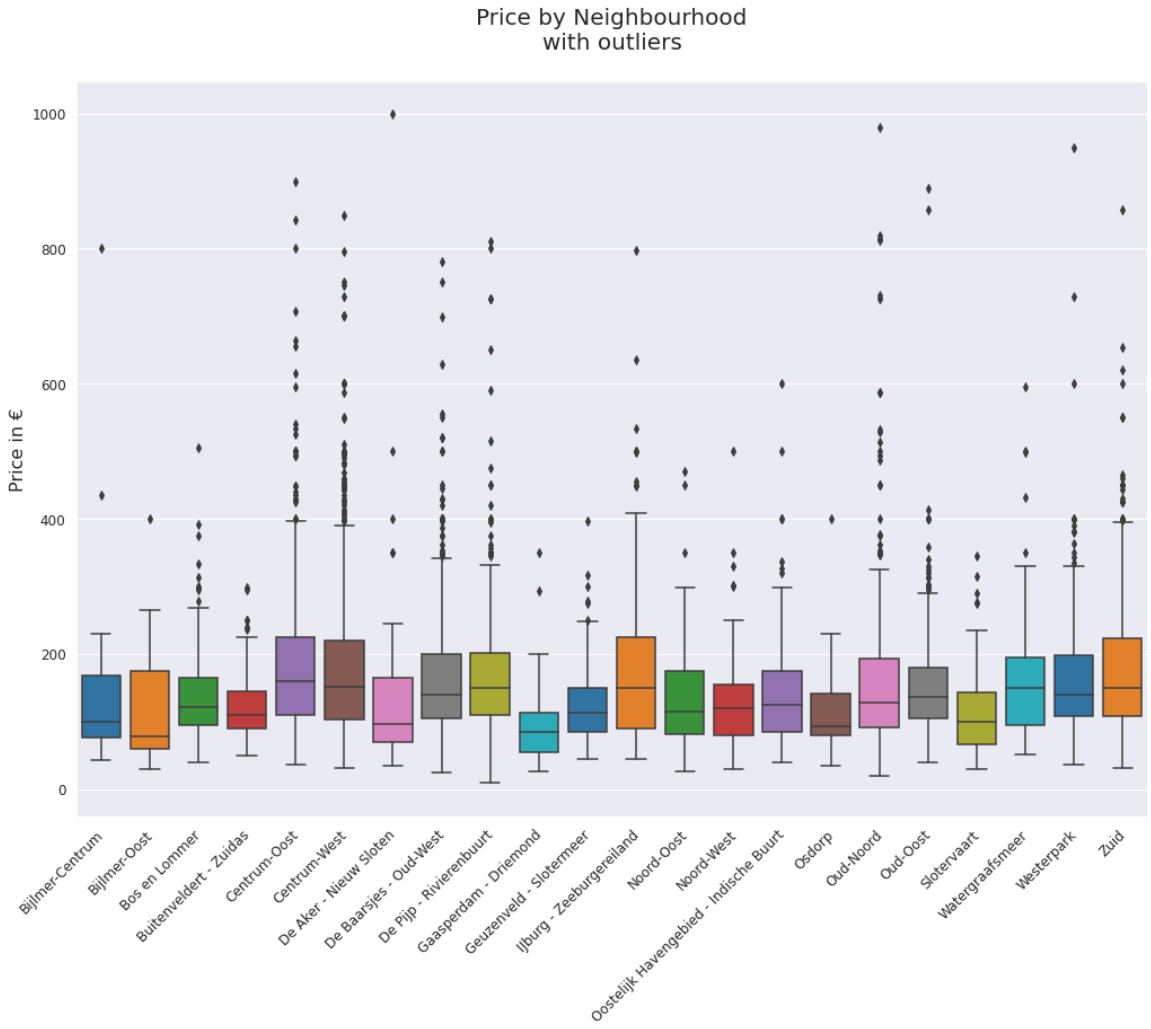
The focus now switches to the relationship between the neighbourhoods and their respective price distributions. The approach stays the same.

In [ ]:
```
plot = sns.displot(
    data=neighbourhood_prices_df,
    kind='kde',
    height=7,
    aspect=3,
    palette=color
)
sns.set(font_scale = 4)
_ = plot.set(xlabel='Price in €')
```

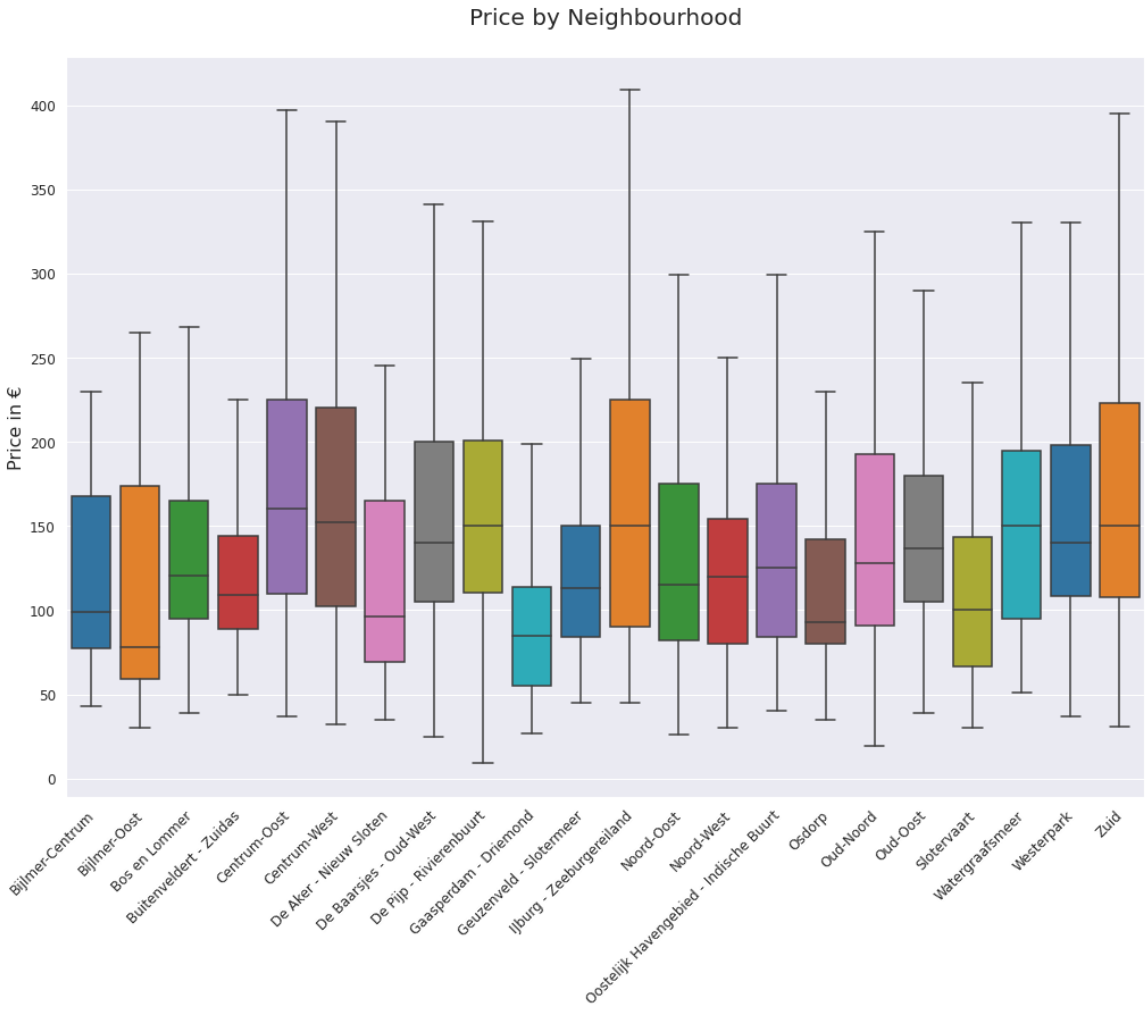


The density plot already indicates, that most locations have similar prices on the lower end. A few neighbourhoods have remarkably larger price ranges than the remaining locations.

In [ ]:
```
def plot7(outliers = False):
    # Plotting
    plt.rcParams["figure.figsize"] = (17,12)
    graph = sns.boxplot(
        showfliers=outliers,
        data=neighbourhood_prices_df,
        palette=color)

    if outliers:
        graph.axes.set_title("Price by Neighbourhood\nwith outliers\n",fo
    else:
        graph.axes.set_title("Price by Neighbourhood\n",fontsize=20)

    graph.set_ylabel("Price in €",fontsize=16)
    graph.tick_params(labelsize=12)

    # Rotate labels to be readable
    xticks_pos = [0.65*patch.get_width() + patch.get_xy()[0] for patch in
    _ = plt.xticks(xticks_pos, listings_neighbourhood.index,  ha='right',
    _ = plt.rcParams["figure.figsize"] = plt.rcParamsDefault["figure.figs

plot7(True)
```



In [ ]:
```
plot7(False)
```



For the price-distribution, it seems obvious that there are many outliers present. Focusing on the plot with hidden outliers should give a good enough picture about the different distributions. The **minimum-prices** of the neighbourhoods **vary between 10-50€**. The **maximum-prices** of the neighbourhoods vary between **200-420€**

Gaasperdam - Driemond has the lowest and also smallest price range, even though its neighbourhood has the highest variation in its location ratings. Again, it may be noted that only 0.8% of listings are located there.

Ijburg - Zeeburgereiland reaches the highest maximum at about **420€** with a large price range, while having an average number of listings available. Its rating is situated rather average. **Centrum--Oost** and **Centrum-West** are very similar, but with much more listings and higher ratings.

Conclusion / Correlations

To see if there is any relationship between the *location-rating* and *price* for each neighbourhood, plotting it as histogramms collectively might already highlight any correlation. The *ratings* and *prices* will be averaged for each neighbourhood.

The average *rating* values vary between 4 and 5, whereas the average *prices* vary between approx. 100 and 180. When plotting on the same graph, one needs to

"norm" respective y-axes.

Furthermore, to highlight neighbourhoods with good *ratings* and low *prices*, a stronger color opacity will indicate respective areas.

In [ ]:
```
# Get individual neighbourhood-data

prices = []
ratings = []


for i in listings_neighbourhood.index:
    listings_current_neighbourhood = listings[listings["neighbourhood_cle

    prices.append(listings_current_neighbourhood["price"])
    # drop entries where price == 0 or > 500
    prices[-1] = prices[-1][prices[-1]!=0]
    prices[-1] = prices[-1][prices[-1]<500]

    prices[-1] = prices[-1].mean()

    ratings.append(listings_current_neighbourhood["review_scores_location
    ratings[-1] = ratings[-1].mean()



labels = listings_neighbourhood.index

### calculations to center the mean values
min_price = min(prices)
max_price = max(prices)

min_rating = min(ratings)
max_rating = max(ratings)

avg_price = sum(prices)/len(prices)
avg_rating = sum(ratings)/len(ratings)
```

In [ ]:
```
# plot1
def plot1(figsize = (15,10)):
    # Figsize
    _ = plt.figure(figsize=figsize)

    # Get colormap
    cmap = plt.get_cmap("Blues")
    color = cmap(0.5)

    ## Plotting
    neighbourhood_distribution = plt.bar(labels, prices, color=color, wid

    ## Title and Labels
    _ = plt.ylabel("Average Price in €", fontsize=16)
    _ = plt.title(f'Average Price by Neighbourhood\n', fontsize=20)
    _ = plt.yticks(fontsize=14)

    # Set background and grid color
    ax.set_facecolor('0.9')
    ax.grid(color='0.85', linewidth=0.7)
```

```
    # Rotate labels to be readable
    xticks_pos = [0.65*patch.get_width() + patch.get_xy()[0] for patch in
    _ = plt.xticks(xticks_pos, listings_neighbourhood.index,  ha='right',
```

In [ ]:
```
# plot2
def plot2(figsize = (15,10)):
    # Figsize
    _ = plt.figure(figsize=figsize)

    # Get colormap
    cmap = plt.get_cmap("Greens")
    color = cmap(0.5)

    ## Plotting
    neighbourhood_distribution = plt.bar(labels, ratings, color=color, wi

    ## Title and Labels
    _ = plt.ylabel("Average Location Rating [0-5]", fontsize=16)
    _ = plt.title(f'Average Rating by Neighbourhood\n', fontsize=20)
    _ = plt.yticks(fontsize=14)

    # Set background and grid color
    ax.set_facecolor('0.9')
    ax.grid(color='0.85', linewidth=0.7)

    # Rotate labels to be readable
    xticks_pos = [0.65*patch.get_width() + patch.get_xy()[0] for patch in
    _ = plt.xticks(xticks_pos, listings_neighbourhood.index,  ha='right',
```

In [ ]:
```
# plot3
def plot3(figsize = (15,10)):

    ### Bar plot
    # figsize
    plt.rcParams["figure.figsize"] = figsize

    # the label locations
    x = np.arange(len(labels))
    # the width of the bars
    width = 0.275

    # 2 different subplots for Price and Rating
    fig, ax = plt.subplots()
    ax2 = ax.twinx()

    # Colors
    # We want to distinguish low prices with high ratings

    # get individual colormaps
    cmap_price = mtl.cm.get_cmap('Blues')
    cmap_rating = mtl.cm.get_cmap('Greens')

    # get individual color-values as list of colors
    colors_price = cmap_price(0.5)
    colors_rating = cmap_rating(0.5)

    rects1 = ax.bar(x - width/2, prices, width, color=colors_price)
    rects2 = ax2.bar(x + width/2, ratings, width, color=colors_rating)
```

```
    ax.set_ylim(bottom=0, top=max_price + 0.1*avg_price)
    ax2.set_ylim(bottom=0, top=max_rating + 0.1*avg_rating)

    # Add some text for labels, title and custom x-axis tick labels, etc.
    legend = plt.legend([rects1, rects2],["Price", "Rating"], loc="upper

    ax.set_title(f'Average Price/Rating by Neighbourhood\n', fontsize=20)
    price_label = ax.set_ylabel("Average Price in €", fontsize=16)
    price_label.set_color(cmap_price(0.6))
    rating_label = ax2.set_ylabel("Average Location Rating [0-5]", fontsi
    rating_label.set_color(cmap_rating(0.6))

    # Rotate labels to be readable
    xticks_pos = [0.65*patch.get_width() + patch.get_xy()[0] for patch in
    _ = ax.set_xticks(xticks_pos, listings_neighbourhood.index,  ha='righ
    _ = ax2.set_xticks(xticks_pos, listings_neighbourhood.index,  ha='rig

    _ = plt.yticks(fontsize=14)

    # Set background and grid color
    ax.set_facecolor('0.95')
    ax2.set_facecolor('0.95')
    ax.grid(color='0.85', linewidth=0.7)
    ax2.grid(color='0.85', linewidth=0.7)

    ax.yaxis.set_tick_params(labelsize=14)
    ax2.yaxis.set_tick_params(labelsize=14)

    # Set layout
    _ = fig.tight_layout()

    plt.show()

    # reset figsize for further plots
    plt.rcParams["figure.figsize"] = plt.rcParamsDefault["figure.figsize"
```

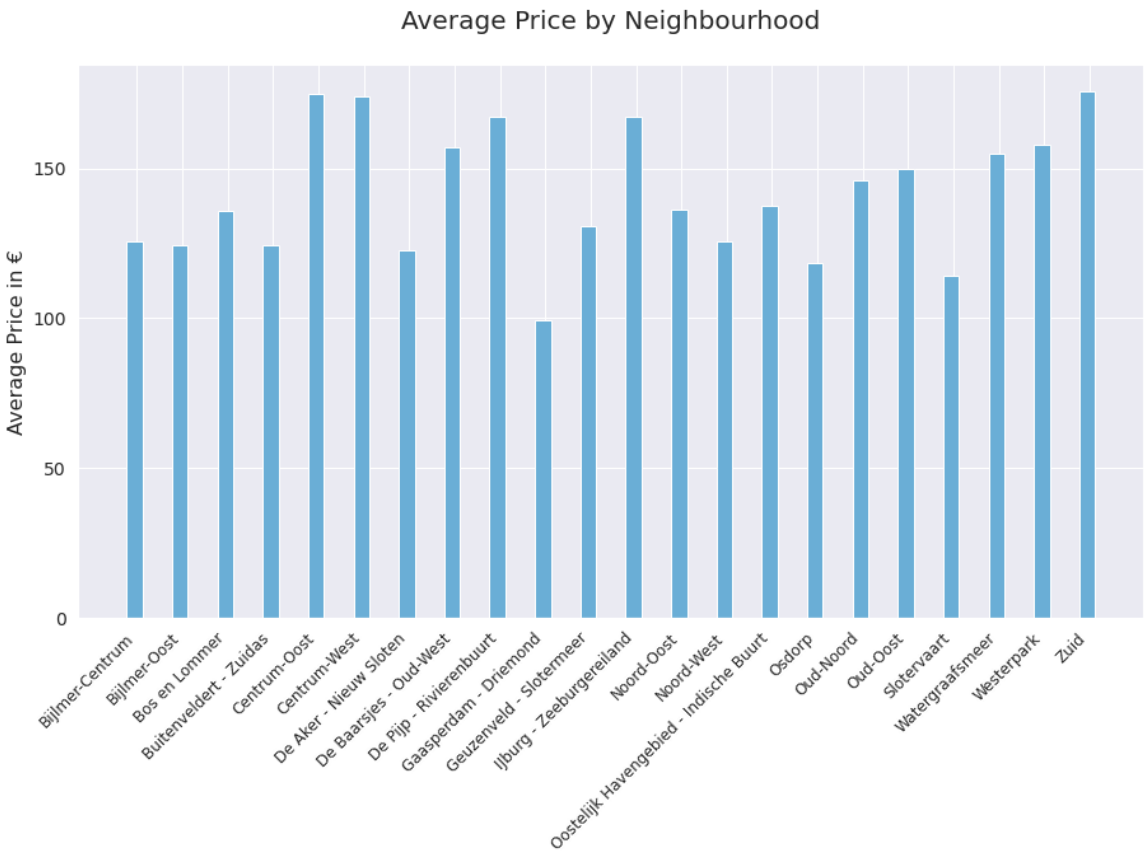
In [ ]:

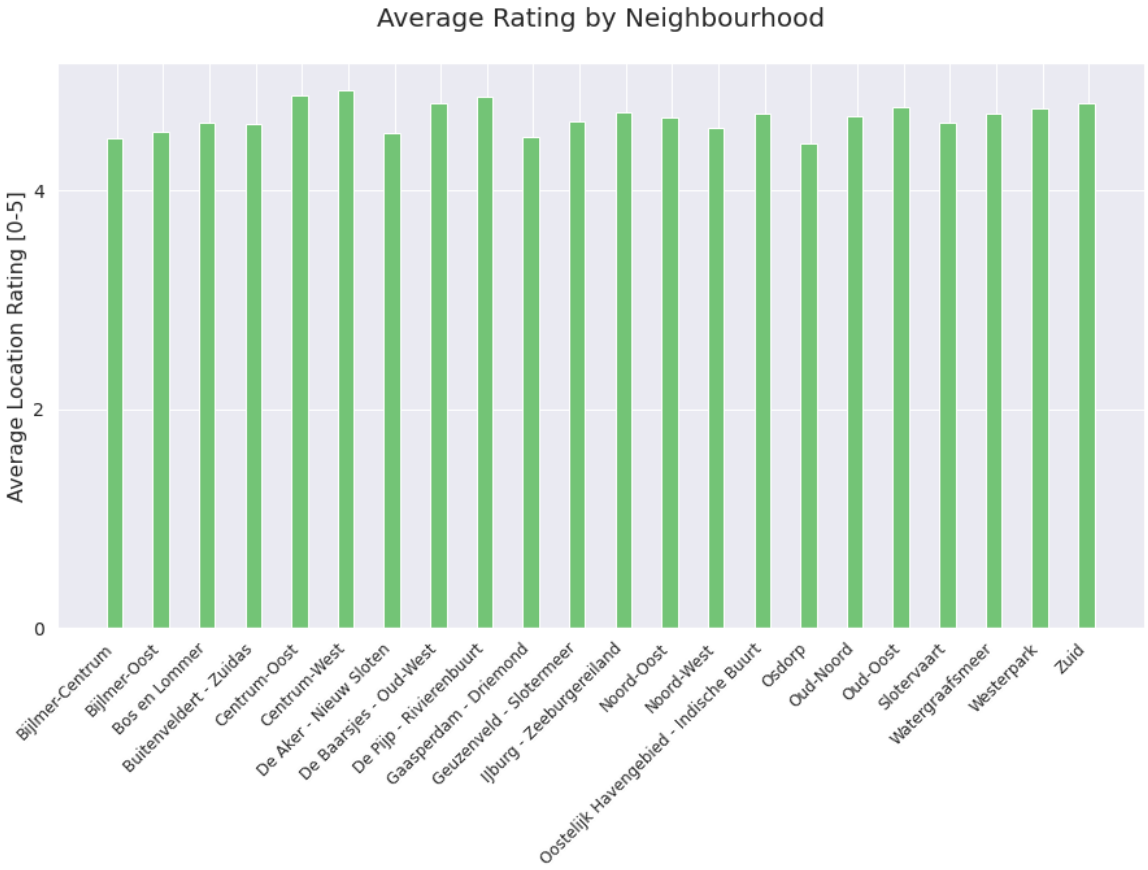
```
def plot4(figsize = (15,10)):
    ### Bar plot
    # figsize
    plt.rcParams["figure.figsize"] = figsize

    # the label locations
    x = np.arange(len(labels))
    # the width of the bars
    width = 0.3

    # 2 different subplots for Price and Rating
    fig, ax = plt.subplots()
    ax2 = ax.twinx()

    # Colors
    # We want to distinguish low prices with high ratings

    # get individual colormaps
    cmap_price = mtl.cm.get_cmap('Blues')
    cmap_rating = mtl.cm.get_cmap('Greens')

    # norm values from 0 - 1
    norm_price = mtl.colors.Normalize(vmin=min_price, vmax=max_price)
```

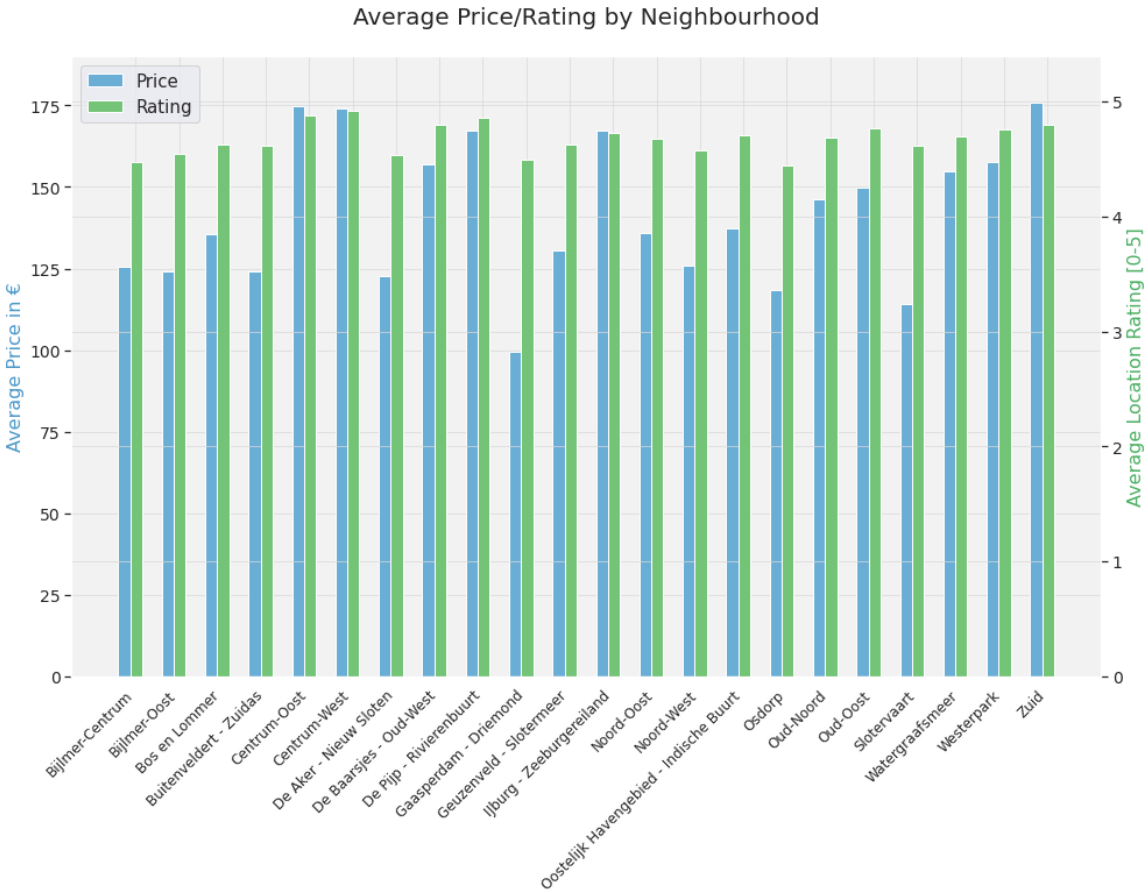
```
    norm_rating = mtl.colors.Normalize(vmin=min_rating, vmax=max_rating)
    price_normalized = norm_price(prices)
    rating_normalized = norm_rating(ratings)

    # get comparision value, new intervall [0, 2]
    comp_values = [1 + rating_normalized[i] - price for i, price in enume

    # best_value, if rating is high and price is low
    best_value = max(comp_values)
    # worst_value, if rating is low and price is high
    worst_value = min(comp_values)
    # avg_value for checking (by looking at the dataset, should be center
    avg_value = sum(comp_values)/len(comp_values)
    if 0.9 > avg_value or 1.1 < avg_value:
        print("Error in calculation")


    # set min value and max value with equal offset (we do not want white
    norm_values = mtl.colors.Normalize(vmin=worst_value*0.4, vmax=2-worst

    # normalize again to be between 0 and 1 for colormaps
    color_values = norm_values(comp_values)

    # check again
    if 0.9 > sum(comp_values)/len(comp_values) or 1.1 < sum(comp_values)/
        print("Error in calculation")

    # get individual color-values as list of colors
    colors_price = cmap_price(color_values)
    colors_rating = cmap_rating(color_values)

    rects1 = ax.bar(x - width/2, prices, width, color=colors_price)
    rects2 = ax2.bar(x + width/2, ratings, width, color=colors_rating)

    ax.set_ylim(bottom=min_price - 0.03*avg_price, top=max_price + 0.03*a
    ax2.set_ylim(bottom=min_rating - 0.03*avg_rating, top=max_rating + 0.

    # Add some text for labels, title and custom x-axis tick labels, etc.
    legend = plt.legend([rects1, rects2],["Price", "Rating"], loc="upper

    ax.set_title(f'Average Price/Rating by Neighbourhood\n', fontsize=20)
    price_label = ax.set_ylabel("Average Price in €", fontsize=16)
    price_label.set_color(cmap_price(0.6))
    rating_label = ax2.set_ylabel("Average Location Rating [0-5]", fontsi
    rating_label.set_color(cmap_rating(0.6))

    # Rotate labels to be readable
    xticks_pos = [0.65*patch.get_width() + patch.get_xy()[0] for patch in
    _ = ax.set_xticks(xticks_pos, labels,  ha='right', rotation=45, fonts
    _ = ax2.set_xticks(xticks_pos, labels,  ha='right', rotation=45, font

    # Set background and grid color
    ax.set_facecolor('0.96')
    ax2.set_facecolor('0.96')
    ax.grid(color='0.85', linewidth=0.7)
    ax2.grid(color='0.85', linewidth=0.7)

    ax.yaxis.get_label().set_fontsize(14)
    ax2.yaxis.get_label().set_fontsize(14)
```

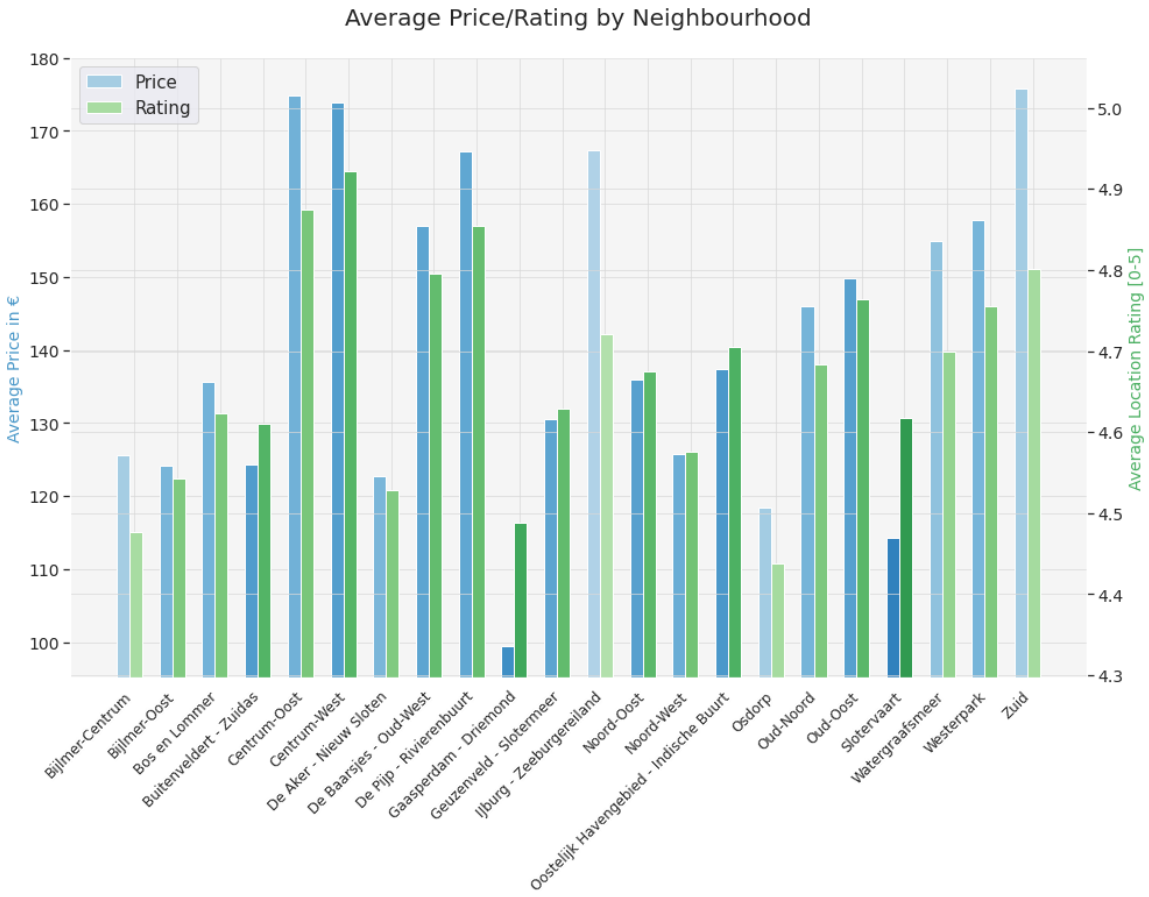
```
    ax.yaxis.set_tick_params(labelsize=14)
    ax2.yaxis.set_tick_params(labelsize=14)

    # Set layout
    _ = fig.tight_layout()

    plt.show()

    # reset figsize for further plots
    plt.rcParams["figure.figsize"] = plt.rcParamsDefault["figure.figsize"
```

In [ ]:
```
figsize = (15, 8)
figsize2 = (15, 12)
plot1(figsize)
plot2(figsize)
plot3(figsize2)
plot4(figsize2)
```




Average Rating by Neighbourhood
Average Location Rating [0-5]
0
2
4
Bijlmer-Centrum
Bijlmer-Oost
Bos en Lommer
Buitenveldert - Zuidas
Centrum-Oost
Centrum-West
De Aker - Nieuw Sloten
De Baarsjes - Oud-West
De Pijp - Rivierenbuurt
Gaasperdam - Driemond
Geuzenveld - Slotermeer
IJburg - Zeeburgereiland
Noord-Oost
Noord-West
Oostelijk Havengebied - Indische Buurt
Osdorp
Oud-Noord
Oud-Oost
Slotervaart
Watergraafsmeer
Westerpark
Zuid


Average Price/Rating by Neighbourhood
Price
Rating
Average Price in €
Average Location Rating [0-5]
0
25
50
75
100
125
150
175
0
1
2
3
4
5
Bijlmer-Centrum
Bijlmer-Oost
Bos en Lommer
Buitenveldert - Zuidas
Centrum-Oost
Centrum-West
De Aker - Nieuw Sloten
De Baarsjes - Oud-West
De Pijp - Rivierenbuurt
Gaasperdam - Driemond
Geuzenveld - Slotermeer
IJburg - Zeeburgereiland
Noord-Oost
Noord-West
Oostelijk Havengebied - Indische Buurt
Osdorp
Oud-Noord
Oud-Oost
Slotervaart
Watergraafsmeer
Westerpark
Zuid



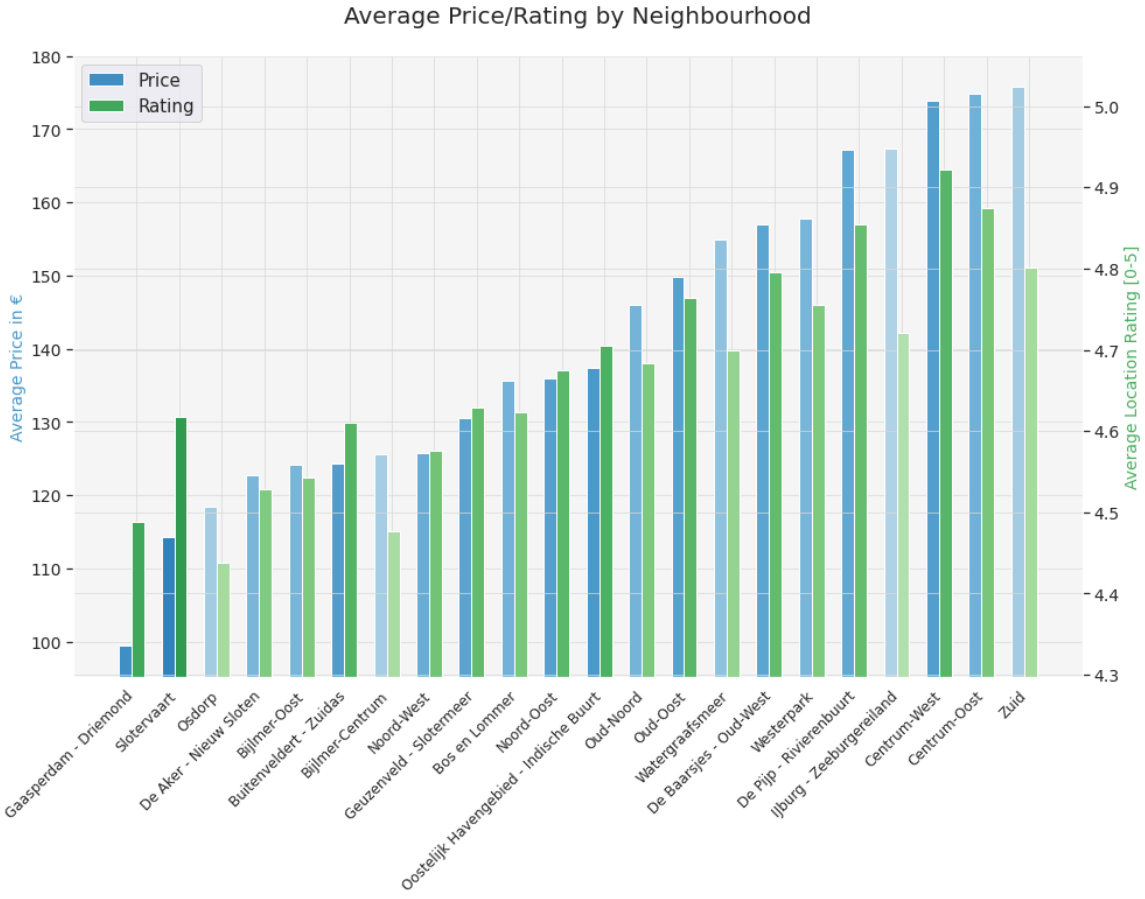
As already seen in previous plots and now highlighted by darker colors, **Gaasperdam - Driemond** and **Slotervaart** have pretty high rating-vales while also maintaining low prices.

In contrast, **Ijburg - Zeeburgereiland** has pretty high avg. prices and a relatively low avg-rating.

As most neighbourhoods seem to be around the same spectrum, plotting the same graph again ordered by prices may yield a nice correlation.

In [ ]:
```
# Get individual neighbourhood-data ORDERED

_prices = []
ratings = []


for i in listings_neighbourhood.index:
    listings_current_neighbourhood = listings[listings["neighbourhood_cle

    _prices.append(listings_current_neighbourhood["price"])
    # drop entries where price == 0 or > 500
    _prices[-1] = _prices[-1][_prices[-1]!=0]
    _prices[-1] = _prices[-1][_prices[-1]<500]

    _prices[-1] = _prices[-1].mean()

    ratings.append(listings_current_neighbourhood["review_scores_location
    ratings[-1] = ratings[-1].mean()

labels = list(listings_neighbourhood.index)
```

```
prices, ratings = (list(t) for t in zip(*sorted(zip(_prices, ratings))))

idx = np.argsort(_prices)

_, labels = (list(t) for t in zip(*sorted(zip(_prices, labels))))

### calculations to center the mean values
min_price = min(prices)
max_price = max(prices)

min_rating = min(ratings)
max_rating = max(ratings)

avg_price = sum(prices)/len(prices)
avg_rating = sum(ratings)/len(ratings)

plot4(figsize2)
```



Interestingly, the best *price/rating* scores can be found for the cheapest neighbourhoods. Centered neighbourhoods maintain an average *price/rating*, whereas the worst scoring neighbourhoods are on the more expensive(right hand) side.

One can clearly see its correlation, as there are genereally higher ratings on the right hand side.

Note: The best *price/rating* score is reached, when a neighbourhood has the highest (relative) *rating* and the lowest (relative) *price*.

Correlation: Response-rate/Review-scores for superhosts and non-superhosts

For the next chapter, the difference between **superhosts** and **non-superhosts** will be analyzed. By looking at the qualifications for superhosts on https://www.airbnb.com/help/article/829/how-to-become-a-superhost, superhosts have to maintain a higher *response-rate* than 90% and an *overall-rating* higher than 4.8.

To get an overview on how many **superhosts** and **non-superhosts** are in the database, their corresponding amount is printed and their respective correlation plots will be plotted further below.

In [ ]:
```
# Print counts
n_superhosts = listings['host_is_superhost'].value_counts()[1]
n_nsuperhosts = listings['host_is_superhost'].value_counts()[0]
print(f'Number of superhosts: {n_superhosts}\nNumber of non-superhosts: {

# split dataset into two separate datasets
is_sh = listings['host_is_superhost'] == "t"
is_nsh = listings['host_is_superhost'] == "f"
listings_sh = listings[is_sh]
listings_nsh = listings[is_nsh]
```

```
Number of superhosts: 1414
Number of non-superhosts: 3980
```

As there are more than twice as many non-superhosts compared to superhosts, reducing the transparency of non-superhost entries will yield a more readable scatterplot.

In [ ]:
```
def plot5(regression = False):
    fig, ax = plt.subplots(figsize=(17,10))

    # Plotting of scatter-plot for superhosts and non-superhosts

    if not(regression):
        _ = plt.scatter('host_response_rate', 'review_scores_rating', c =
        _ = plt.scatter('host_response_rate', 'review_scores_rating', c =
    else:
        _ = plt.scatter('host_response_rate', 'review_scores_rating', c =

    if not(regression):
        _ = ax.set_title(f'Response rate / Review scores scatterplot\n',
    else:
        _ = ax.set_title(f'Response rate / Review scores scatterplot\n wi

    if regression:
        sns.regplot(x='host_response_rate', y='review_scores_rating', dat

    if not(regression):
        # Legend
        lgnd = plt.legend(['is superhost', 'is not a superhost'], loc='lo
        colors=['green', 'red']
        for i, j in enumerate(lgnd.legendHandles):
```

```
            j.set_color(colors[i])
            j.set_alpha(1)
    _ = plt.ylabel("Review scores rating  [0-5]", fontsize=15)
    _ = plt.xlabel("% Host response rate", fontsize=15)

    ax.xaxis.set_tick_params(labelsize=14)
    ax.yaxis.set_tick_params(labelsize=14)

plot5(True)
plot5()
```

There are extrems on both ends. Clearly, **superhosts** generally maintain a higher *response-rate* and *ratings*. Yet when looking at the plot, one can detect some **superhosts** that should not have the qualifications to be one.

Furthermore, what seems unusual are the number of hosts that have a *response-rate* of 0% and exactly 50%. Also interesting to interpret are the corresponding *ratings* that are mostly higher than 4. This might be an indication that the dataset's rating values are not scraped accurately for all the listings.

With the help of the **blue regression line** one can validate its correlation.

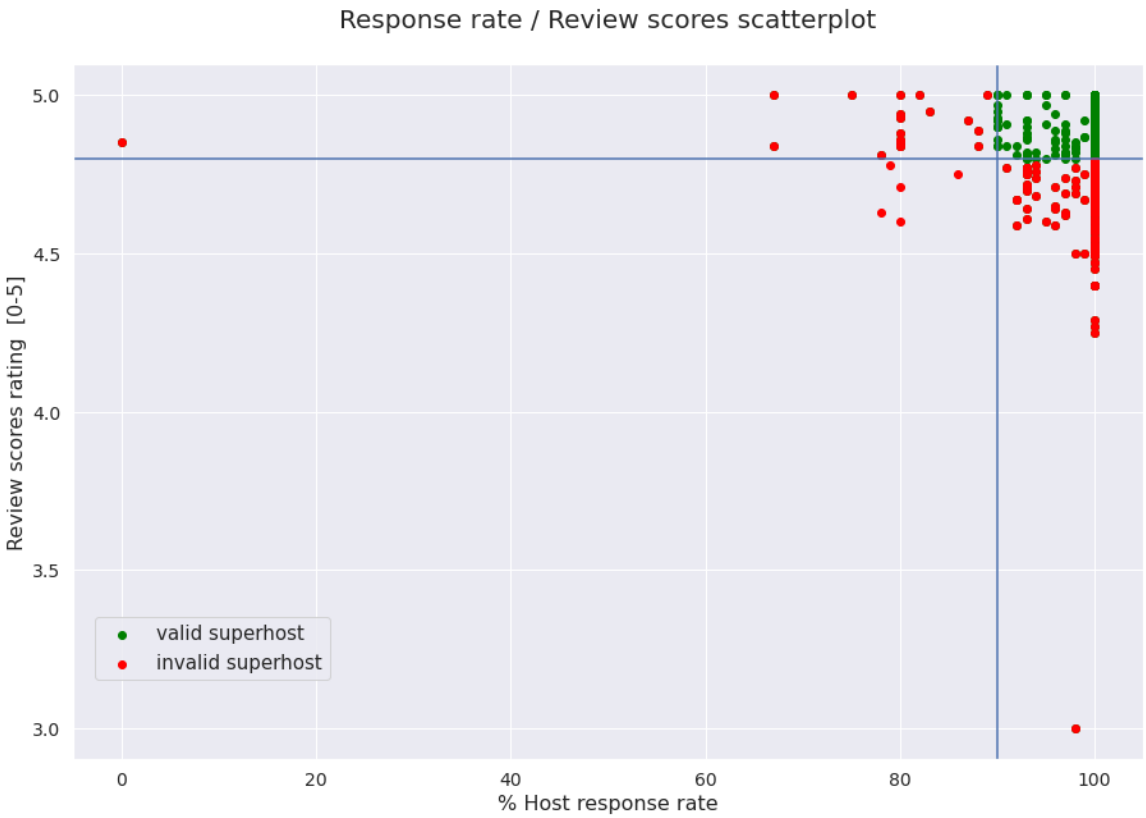
To confirm that there are indeed **unqualified superhosts** present in the dataset, the total number of **superhosts** that have either a *response-rate* < 90% or a *rating* < 4.8 are calculated, printed and plotted below.

In [ ]:
```
number_of_invalid_sh = listings_sh[(listings_sh["review_scores_rating"] <
print(f'Number of superhosts: {listings_sh.count()[0]}\nNumber of unquali
```

```
Number of superhosts: 1414
Number of unqualified superhosts: 297
```

Approximately **17% of superhosts** should not be eligible to be one. This might indicate that airbnb does not enforce their own superhost-qualifications as written on their site. The following plot additionally confirms the observation.

In [ ]:
```
# splitting into seperate datasets
invalid_sh = listings_sh[(listings_sh["review_scores_rating"] < 4.8) | (l
valid_sh = listings_sh[(listings_sh["review_scores_rating"] >= 4.8) | (li

# Plotting
_ = plt.rcParams["figure.figsize"] = (15,10)
_ = plt.ylabel("Review scores rating  [0-5]", fontsize=16)
_ = plt.xlabel("% Host response rate", fontsize=16)
_ = plt.title(f'Response rate / Review scores scatterplot\n', fontsize=20

plot = plt.scatter('host_response_rate', 'review_scores_rating', c = ["gr
plot = plt.scatter('host_response_rate', 'review_scores_rating', c = ["re
# Legend
lgnd = plt.legend(['valid superhost', 'invalid superhost'], loc='lower ce
colors=['green', 'red']
for i, j in enumerate(lgnd.legendHandles):
    j.set_color(colors[i])
    j.set_alpha(1)

# horizontal and vertical lines
ynew = 4.8
xnew = 90
plt.axhline(ynew, color='b')
plt.axvline(xnew, color='b')

plt.xticks(fontsize=14)
plt.yticks(fontsize=14)

# Reset figsize for further plots
_ = plt.rcParams["figure.figsize"] = plt.rcParamsDefault["figure.figsize"
```



Notably, the number of listings with a *response-rate* of 0% and 50% shrunk immensely. Still, one listing remained with a *response-rate* of 0%.

Correlation: Host-verifications vs Number-of-Reviews/Availability

Hosts on airbnb can have different kind of verifications. The details of what specific verification a host has aquired have been omitted and only the amount of verifications has been counted. When looking at the **correlation-plot**, one can clearly see that there is a:

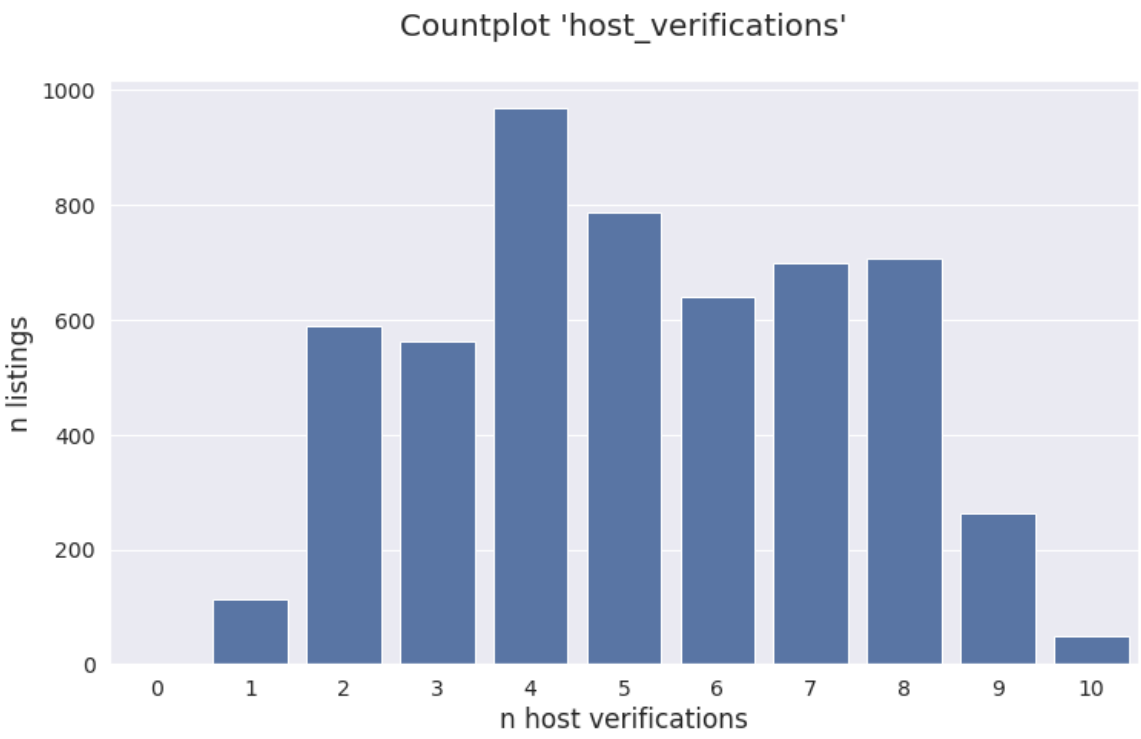
- **positive correlation** for *host-verifications* and *number-of-reviews*
- **negative correlation** for *host-verifications* and *availability*

This might indicate, that hosts with more verifications get booked more often and therefore yield a higher amount of reviews while having a lower availabilty.

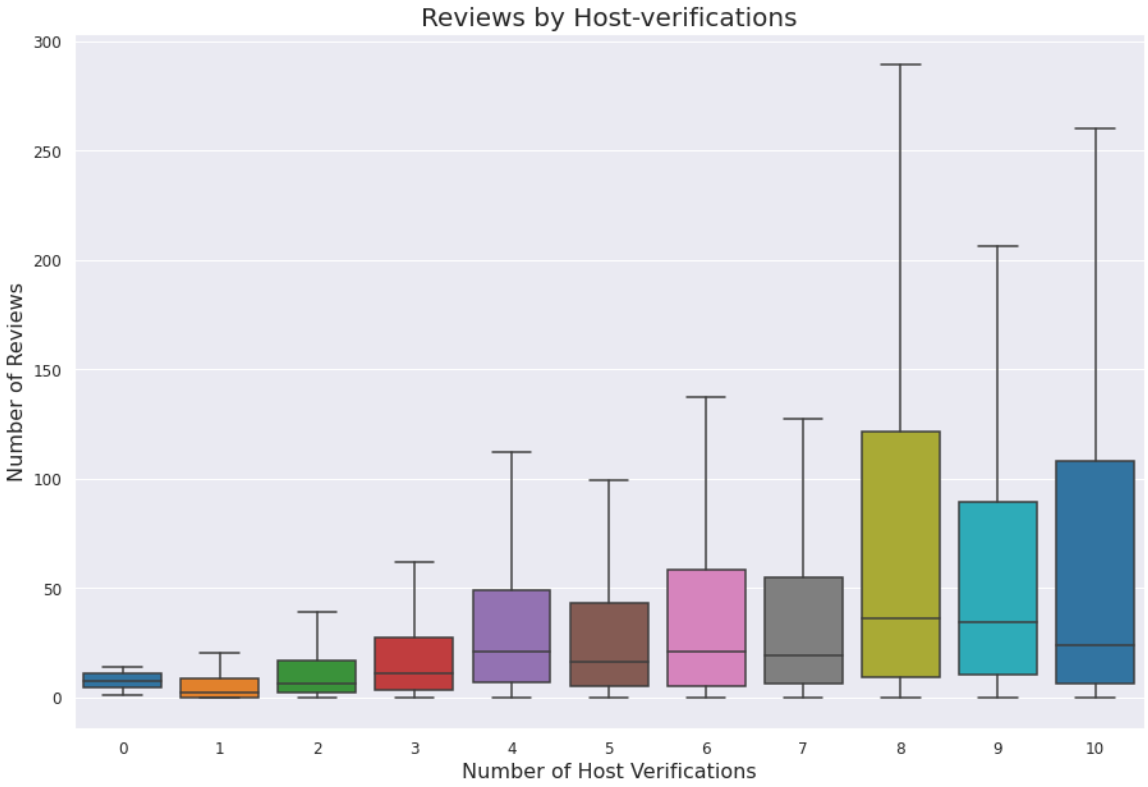
Since the focus is on a correlation between one categorical and one continuos attribute, a multi-boxplot will give a good overview to inspect the links between the two attributes. Furthermore, since the **sample sizes** for each number of *host-verifications* might invalidate some bins, they will simultanously be checked.

In [ ]:
```
# Get dataset splitup into n host verification lists

number_of_reviews = []
host_ver_count = []
columns = []

plt.rcParams["figure.figsize"] = (12,7)
```

```
print()
for i in range(listings["host_verifications"].max()):
    host_ver = listings[listings["host_verifications"] == i]
    host_ver_count.append(host_ver["availability_365"].count())
    number_of_reviews.append(pd.DataFrame(host_ver["number_of_reviews"].t
    columns.append(i)

df_number_of_reviews = pd.concat(number_of_reviews, ignore_index=True)

ax = sns.countplot(data = df_number_of_reviews, color='b')
_ = ax.set_title(f'Countplot \'host_verifications\'\n', fontsize=20)
_ = ax.set_xlabel('n host verifications', fontsize=17)
_ = ax.set_ylabel('n listings', fontsize=17)
ax.xaxis.set_tick_params(labelsize=14)
ax.yaxis.set_tick_params(labelsize=14)


plt.rcParams["figure.figsize"] = plt.rcParamsDefault["figure.figsize"]
```



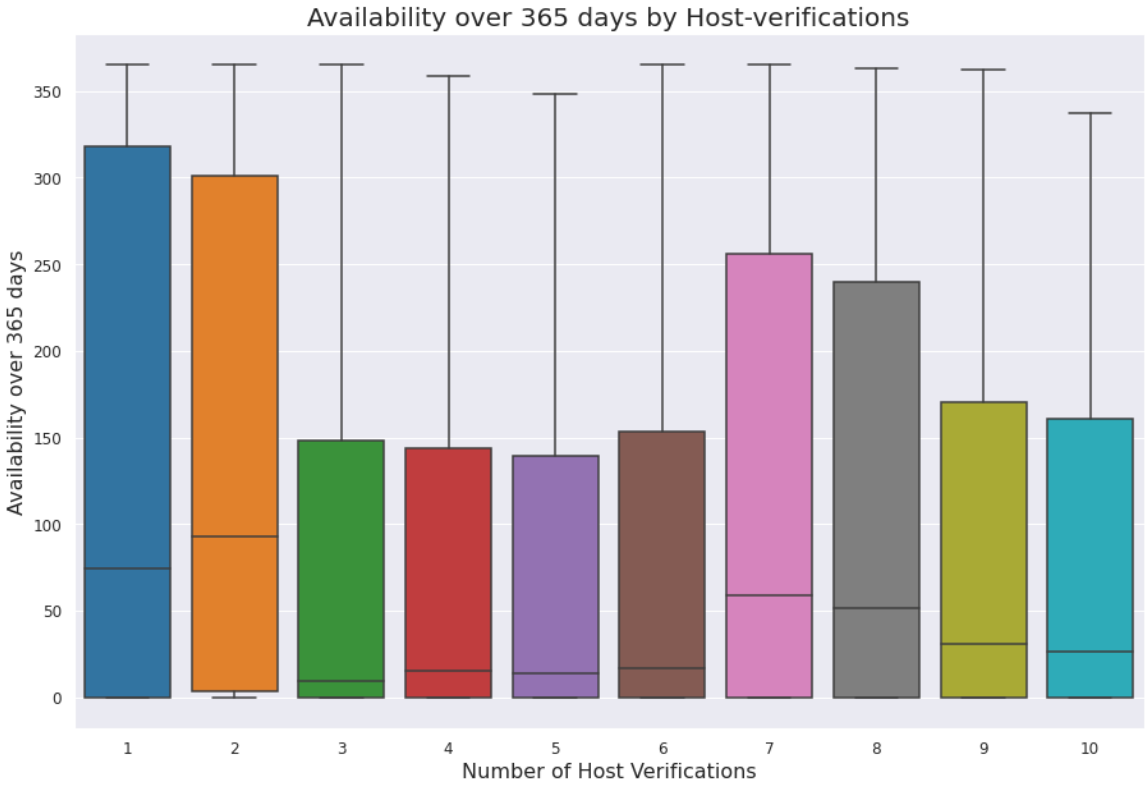
In [ ]:
```
# Plotting
plt.rcParams["figure.figsize"] = (15,10)
graph = sns.boxplot(
    showfliers=False,
    data=df_number_of_reviews,
    palette=color)

graph.axes.set_title("Reviews by Host-verifications",fontsize=20)
graph.set_xlabel("Number of Host Verifications",fontsize=16)
graph.set_ylabel("Number of Reviews",fontsize=16)
graph.tick_params(labelsize=12)

# set labels
_ = plt.xticks(df_number_of_reviews.columns)
# reset figsize
_ = plt.rcParams["figure.figsize"] = plt.rcParamsDefault["figure.figsize"
```



Clearly visible, a higher amount of *host-verifications* generally yields higher *number-of-reviews* which strengthens the claim of having more bookings.

Next up is the same approach, focusing on the *availability* of the listings. Furthermore, since the **sample size** of **0 *host-verifications* listings is 2** and therefore much too low, it will be excluded from the graph.

In [ ]:
```
# Get dataset splitup into n host verification lists

availability = []

for i in range(1, listings["host_verifications"].max()):
    host_ver = listings[listings["host_verifications"] == i]
    availability.append(pd.DataFrame(host_ver["availability_365"].tolist(

df_availability = pd.concat(availability, ignore_index=True)


# Plotting
plt.rcParams["figure.figsize"] = (15,10)
graph = sns.boxplot(
    showfliers=False,
    data=df_availability,
    palette=color)

graph.axes.set_title("Availability over 365 days by Host-verifications",f
graph.set_xlabel("Number of Host Verifications",fontsize=16)
graph.set_ylabel("Availability over 365 days",fontsize=16)
graph.tick_params(labelsize=12)

# set labels
_ = plt.xticks(range(10))
# reset figsize
_ = plt.rcParams["figure.figsize"] = plt.rcParamsDefault["figure.figsize"
```



Also this looks accurate when comparing the individual median values, even though the correlation doesn't look as clean as the one above. It definitely looks like that hosts with more than 2 *verifications* get more bookings than any below. A further interesting finding is that for every number of *host-verifications*, one can find listings with almost an *availability* of 365 days. This means, that there are several individual listings with an *availability* of 365days/year.

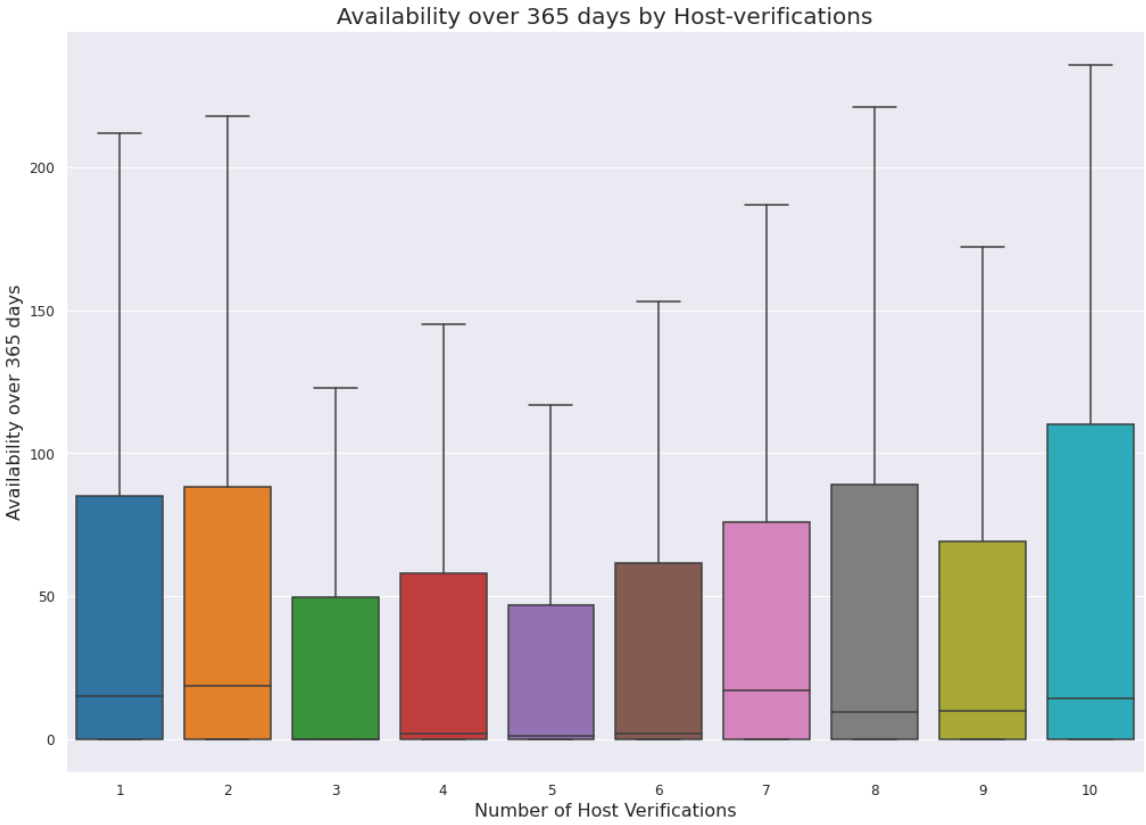
The median values are all situated pretty low, and the variation of the availability is very large for each bin.

To get a better representation, excluding all listings with high availability throughout the year might help out.

In [ ]:
```
# Get dataset splitup into n host verification lists

availability = []
availability_excl = 240

for i in range(1, listings["host_verifications"].max()):
    host_ver = listings[listings["host_verifications"] == i]
    availability.append(pd.DataFrame(host_ver[host_ver["availability_365"]

df_availability = pd.concat(availability, ignore_index=True)
print()
print(f'Excluded {total_all - df_availability.count().sum() - 2} listings


# Plotting
plt.rcParams["figure.figsize"] = (17,12)
graph = sns.boxplot(
    showfliers=False,
    data=df_availability,
    palette=color)
```

```
graph.axes.set_title("Availability over 365 days by Host-verifications",f
graph.set_xlabel("Number of Host Verifications",fontsize=16)
graph.set_ylabel("Availability over 365 days",fontsize=16)
graph.tick_params(labelsize=12)

# set labels
_ = plt.xticks(range(10))
# reset figsize
_ = plt.rcParams["figure.figsize"] = plt.rcParamsDefault["figure.figsize"
```

```
Excluded 1186 listings from a total of 5393 listings with availability >
240 for plotting.
```



Even though a slight correlation is recognizable, having more than 2 verifications seems to be enough for hosts to get the most bookings. In contrast, a high *number-of-verifications* yields more *reviews*. Considering the small sample sizes on both ends (0, 1, 9, 10 host verifications bins), the plots of those bins might not be that accurate.

It might have something to do with special types of verifications that yield the most bookings.

Correlation: Price vs Amenities

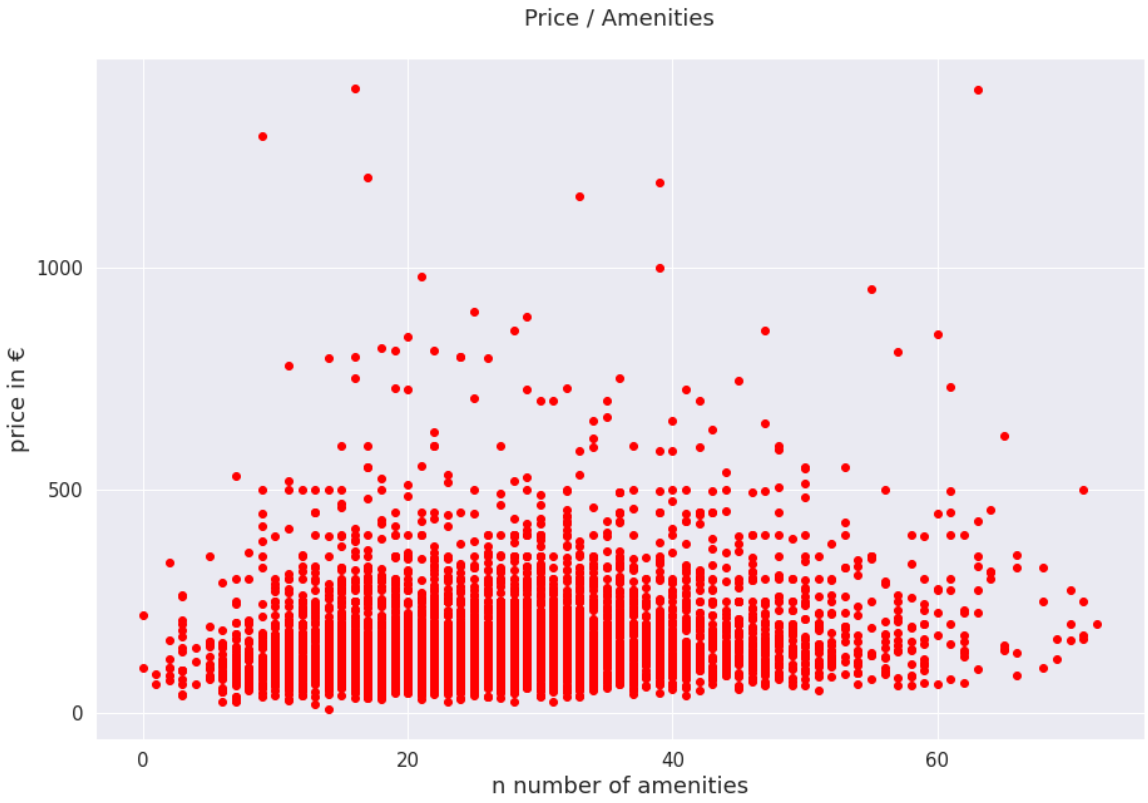
Further promising correlation found in the **correlation-heatmap**:

- **positive correlation** for *amenities* and *price*

There is a strong-relationship between the *amount-of-amenities* (e.g. extras) and their respective *prices*. Even though it might be an obvious correlation, the extend of

its relationship remains to be showed.

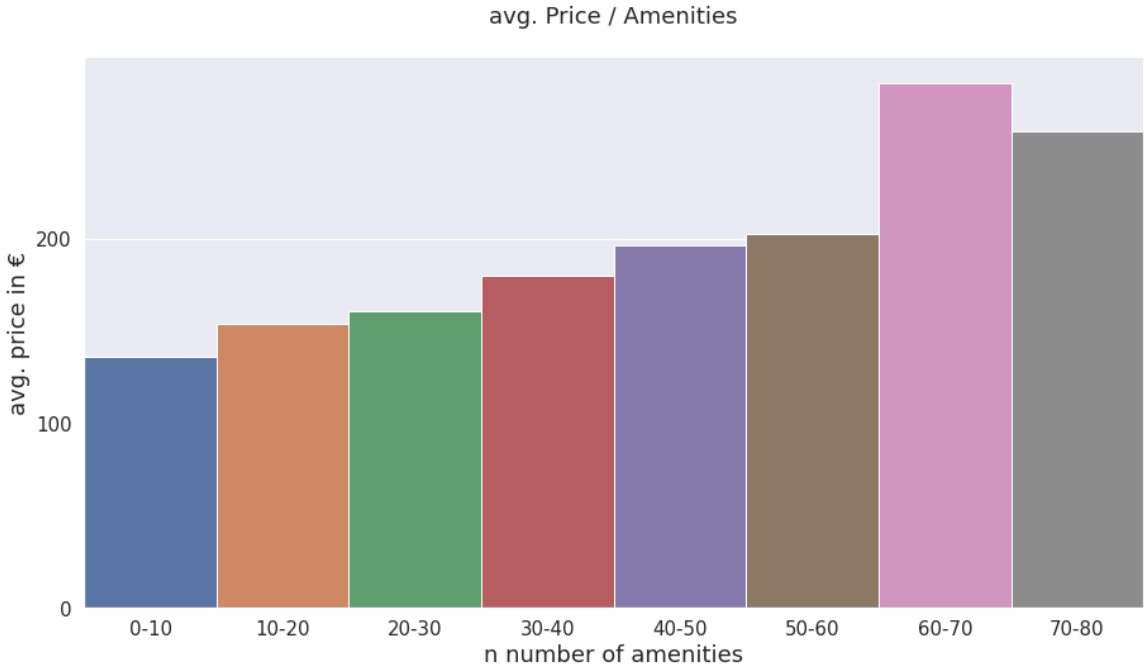
In [ ]:
```
## exclude extraordinary price values
new_listings = listings[listings["price"] < 2000]

# Plotting of scatter-plot for superhosts and non-superhosts
_ = plt.rcParams["figure.figsize"] = (15,10)
_ = plt.xticks(fontsize = 15)
_ = plt.yticks(fontsize = 15)
_ = plt.xlabel("n number of amenities", fontsize = 18)
_ = plt.ylabel("price in €", fontsize = 18)
_ = plt.title(f'Price / Amenities\n', fontsize = 18)

plot = plt.scatter('amenities', 'price', c = ["red"], data=new_listings)

# Reset figsize for further plots
_ = plt.rcParams["figure.figsize"] = plt.rcParamsDefault["figure.figsize"
```



Since it is hard to evaluate any correlation within this scatterplot, a visualization via a barplot and corresponding bins is tried below. It remains to be noted from the scatterplot above, that there might be **very low sample sizes for the latter bins**.

In [ ]:
```
df = pd.DataFrame({'number_of_amenities':new_listings["amenities"], 'pric
df['number_of_amenities'] = pd.cut(df['number_of_amenities'], bins=range(

fig,ax = plt.subplots(figsize=(15,8))
plot = sns.barplot(x='number_of_amenities', y='price', data=df, ax=ax, ci

_ = plt.xticks(fontsize = 15)
_ = plt.yticks(fontsize = 15)
_ = plt.xlabel("n number of amenities", fontsize = 18)
_ = plt.ylabel("avg. price in €", fontsize = 18)
_ = plt.title(f'avg. Price / Amenities\n', fontsize = 18)
```

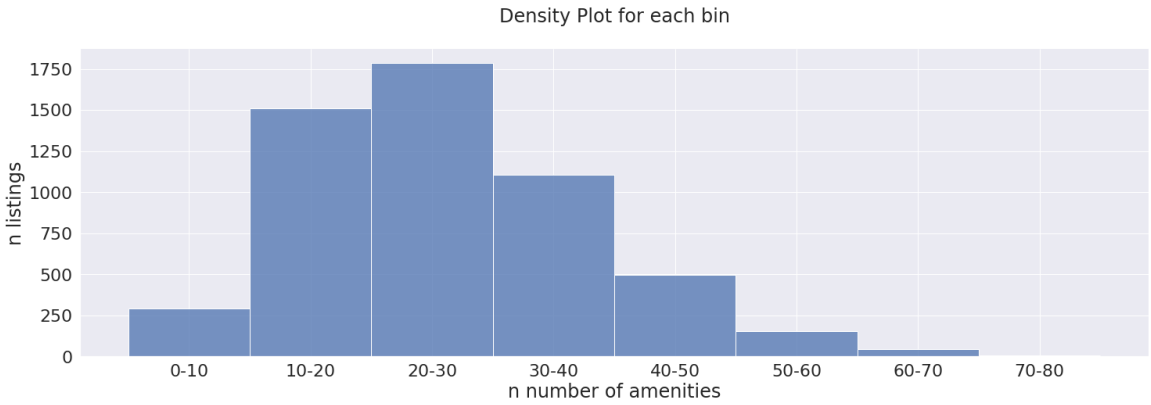
```
def change_width(ax, new_value) :
    for patch in ax.patches :
        current_width = patch.get_width()
        diff = current_width - new_value

        # we change the bar width
        patch.set_width(new_value)

        # we recenter the bar
        patch.set_x(patch.get_x() + diff * .5)


change_width(ax, 1.)
```



This perfectly illustrates the clean correlation between the *average prices* for each bin of *n-amenities*. Interestingly, the *average price* is lower than the previous for the last bin. Furthermore, **the *averaged price* approximately doubles from 0-10 to 60-70 *amenities***. The biggest increase of the price can be found for the pink bin.

To visualize the sample-sizes for each bin, the distribution of listings over all bins is quickly checked below.

In [ ]:
```
_ = sns.set(font_scale = 2)
plot = sns.displot(x='number_of_amenities', data=df, height=7, aspect=3)
_ = plot.set(xlabel='n number of amenities', ylabel="n listings", title=f
```



The big differences for the last two bins might come from the fact that there are very low sample sizes for respective listings. All in all, the *averaged price* increases monotonically and linear, over an increase in *amenities*.

Correlations with missing-reviews in listings

As there has been a lot of missing values for the feature-block *ratings*, this chapter focuses on correlating it to any other attribute to possibly find any reasoning behind it.

Firstly, a new feature in the dataset is going to be inserted, indicating whether a listing *has a rating* or not to investigate if there are any obvious relations with other attributes.

In [ ]:
```
# insert new boolean feature that indicates if the listing has a rating
null_values_listings = listings
null_values_listings["has_rating"] = ~null_values_listings["review_scores
```

In [ ]:
```
null_values_listings[null_values_listings["has_rating"]==False].head(10)
```

Out[ ]:

	id	last_scraped	host_id	host_name	host_since	host_response_time	h
200	917955	2021-11-04	589967	Silvy	2011-05-15	NaN	
356	1658094	2021-11-04	8788562	Elaine	2013-09-12	within a day	
683	4162988	2021-11-04	20157581	Rob	2014-08-17	within a few hours	
708	4339471	2021-11-04	3374197	Bibi	2012-08-26	NaN	
739	4610966	2021-11-04	21453001	Robbert	2014-09-17	NaN	
806	5273691	2021-11-04	27300591	Marit	2015-02-06	a few days or more	
929	6375033	2021-11-04	22159410	Anja	2014-10-05	within a day	
1132	8265818	2021-11-04	15519914	Iris	2014-05-14	within a day	
1214	9254675	2021-11-04	22490099	Viv	2014-10-13	NaN	
1451	12373532	2021-11-04	30890942	Zoku Amsterdam	2015-04-09	within an hour	

At first glance, one can't see any correlation to other attributes.

As it could be in correlation to being a superhost, its entries will be statistically checked below.

In [ ]:
```
null_values_listings[null_values_listings["has_rating"]==False]["host_is_
```

Out[ ]:
```
count     464
unique      2
top         f
freq      430
Name: host_is_superhost, dtype: object
```

This can't be the case, since both, **superhosts** and **non-superhosts**, have null values present.

To find out if there might be any other relevant correlations, calculating the **point biserial correlation** sounds reasonable, taking every continuos value present in the dataset and the binominal attribute *has_rating*. Missing entries will be replaced with mean values.

In [ ]:
```
columns=["host_acceptance_rate",
         "host_listings_count",
         "host_total_listings_count",
         "host_verifications",
         "price",
         "minimum_nights",
         "maximum_nights",
```

```
        "availability_365",
        "calculated_host_listings_count"]


corr_list = []

y = null_values_listings['has_rating'].astype(float)

for column in columns:
    # fill NaN values with mean
    x=null_values_listings[column].fillna(null_values_listings[column].me
    # apply pointbiserial
    corr = stats.pointbiserialr(list(x), list(y))
    corr_list.append(corr[0])

print(f'Point Biserial Correlation\nBinominal Feature = \'has_rating\'\n'
for i in range(len(columns)):
    print(f'Continuous feature: {columns[i]}\nCalculation: {corr_list[i]}
```

```
Point Biserial Correlation
Binominal Feature = 'has_rating'

Continuous feature: host_acceptance_rate
Calculation: 0.05604744916776344
Continuous feature: host_listings_count
Calculation: -0.005886325329899029
Continuous feature: host_total_listings_count
Calculation: -0.005886325329899029
Continuous feature: host_verifications
Calculation: 0.11065388479456437
Continuous feature: price
Calculation: -0.10201035221436199
Continuous feature: minimum_nights
Calculation: -0.00367703423541322
Continuous feature: maximum_nights
Calculation: 0.04002729351340432
Continuous feature: availability_365
Calculation: -0.0423981765482739
Continuous feature: calculated_host_listings_count
Calculation: -0.06636454831613735
```

The values indicate that there is no strong relation between *has_ratings* and the continuos features listed above. Unfortunately, this chapter is concluded without any findings about why there are missing ratings in the dataset.

Correlations: Districts / Price / Roomtype / Apartments

Data preperation

In [ ]:
```
# listings_without_outliers = listings without 2 most expensive apartment
listings_without_outliers = listings.copy()
listings_without_outliers = listings_without_outliers[listings_without_ou

# changing bedrooms to 1, 2, 3+
listings_without_outliers['bedrooms'] = listings_without_outliers['bedroo
```

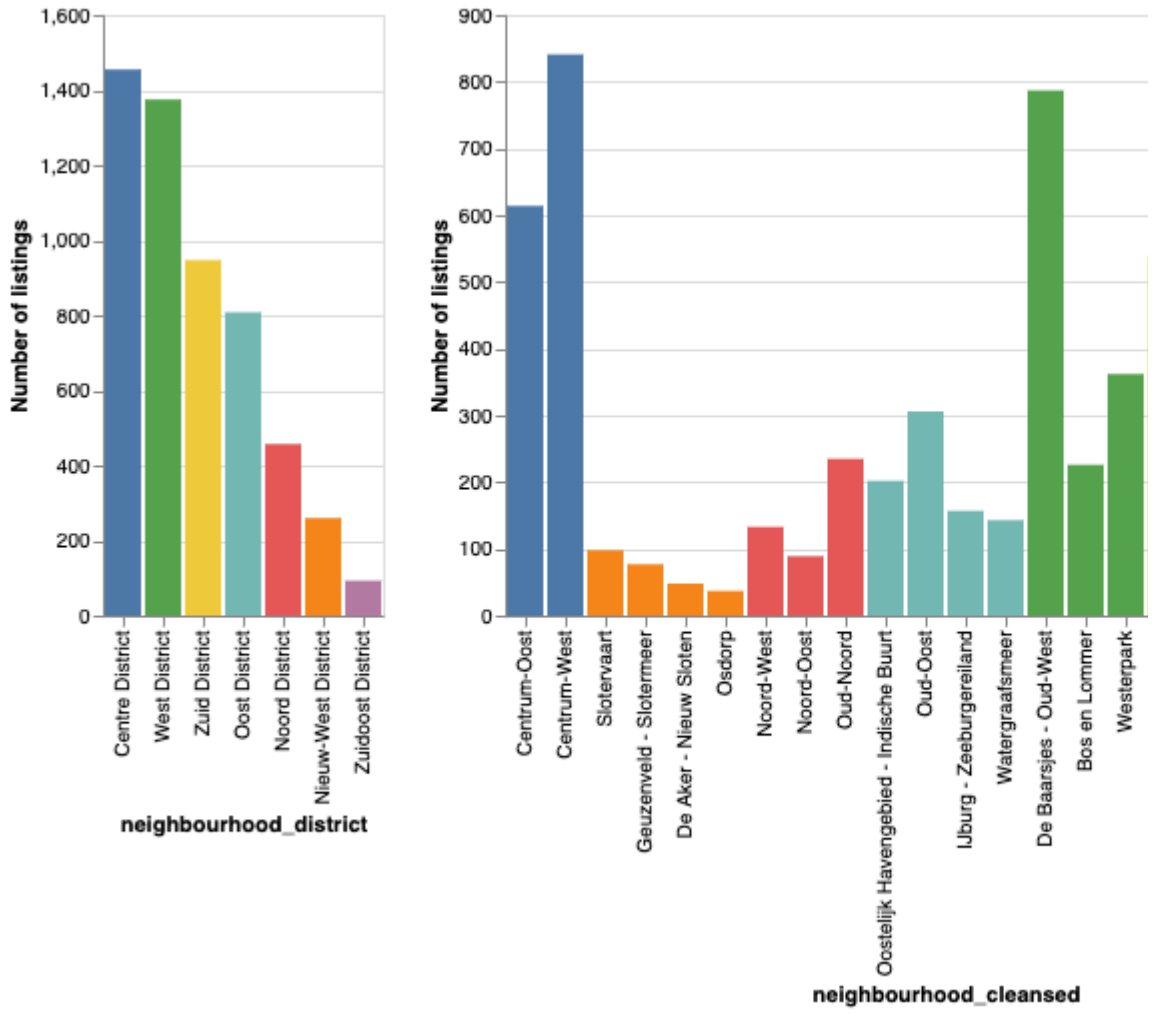
Interactive linking between neighbourhoods and districts

Firstly, to get an overview the number of listings in each neighbourhood and each district. The result tells that the most listings are in the Centre District (1458 listings) followed by West District (1377) and the most linkings in specific neighbouhood is the Centrum-West neighbourhood (842) followed by De Baarsjes - Oud-West neighbourhood (788) from West District.

Combining those two charts below with the use of interactivity to show clearly each neighbourhood in each district.

In [ ]:
```
brush = alt.selection_interval(encodings=['x'])

chart = alt.Chart(listings).mark_bar().encode(
    y = alt.Y('count()', axis=alt.Axis(title='Number of listings')),
    tooltip=['count()', 'neighbourhood_district', 'average(price)'],
    color=alt.condition(brush, 'neighbourhood_district', alt.value('light
).add_selection(
    brush
)

chart.encode(x = alt.X('neighbourhood_district', sort='-y')) | chart.enco
```

Out[ ]:



Average price of each neighbourhood sorted by districts

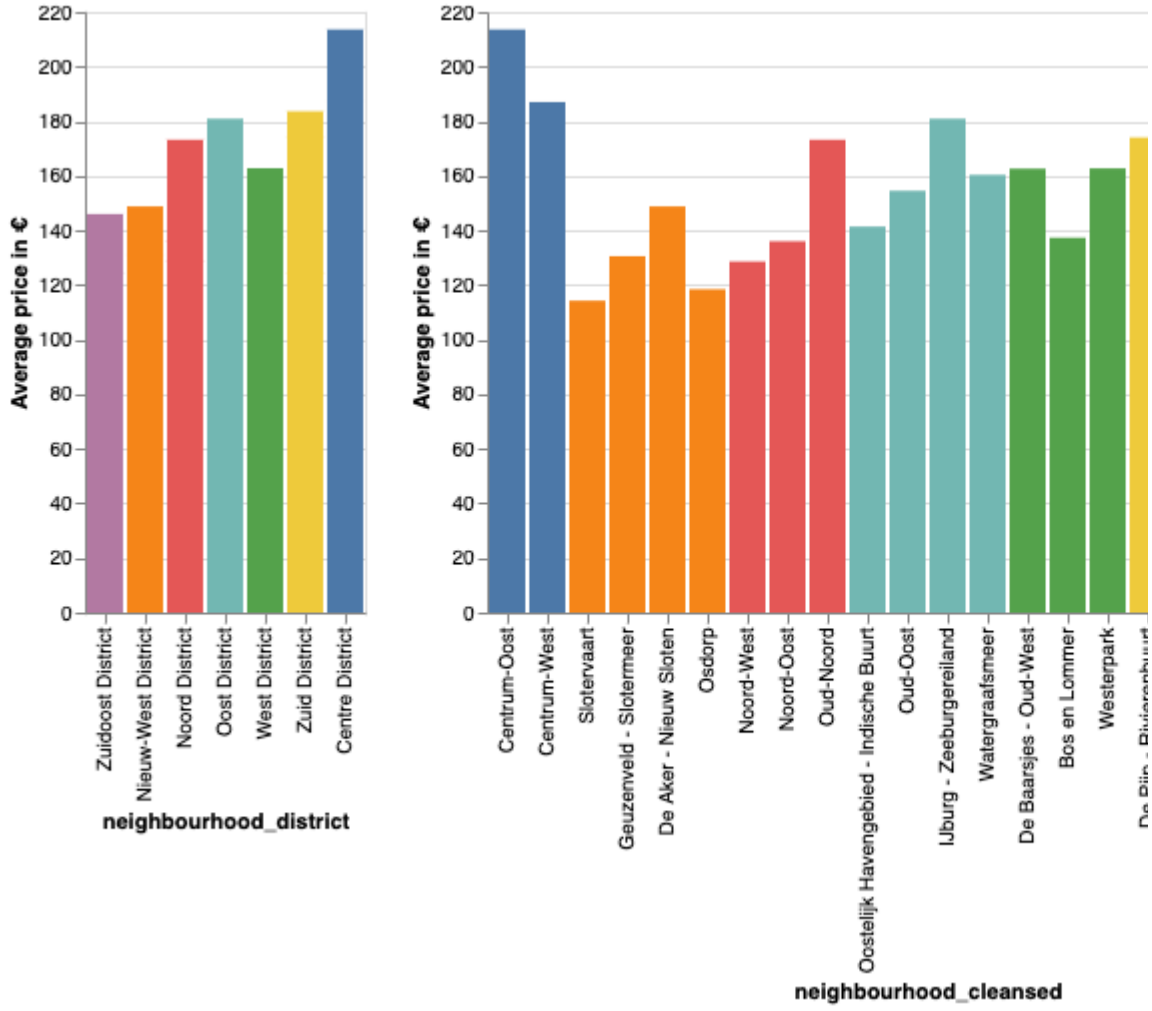
Secondly, to get an overview of average prices throughout each neighbourhood and district. As a result one can see that on average obviously Centre District is the most

expensive followed by Zuid and West district. And the most expensive neighbourhoods go to Centrum-Oost, Centrum-West and Zuid.

Taking everything into account with numbers of listings one could find that the most listings and most expensive is the Centre District but that does not apply for second place. In second place the most listings are in West District but the second most expensive is Zuid District.

In [ ]:
```
brush = alt.selection_interval(encodings=['x'])

chart = alt.Chart(listings).mark_bar().encode(
    y = alt.Y('average(price)', axis=alt.Axis(title='Average price in €')
    tooltip=['neighbourhood_district', 'neighbourhood_cleansed', 'average
    color=alt.condition(brush, 'neighbourhood_district', alt.value('light
).add_selection(
    brush
)

chart.encode(x = alt.X('neighbourhood_district', sort='y')) | chart.encod
```

Out[ ]:

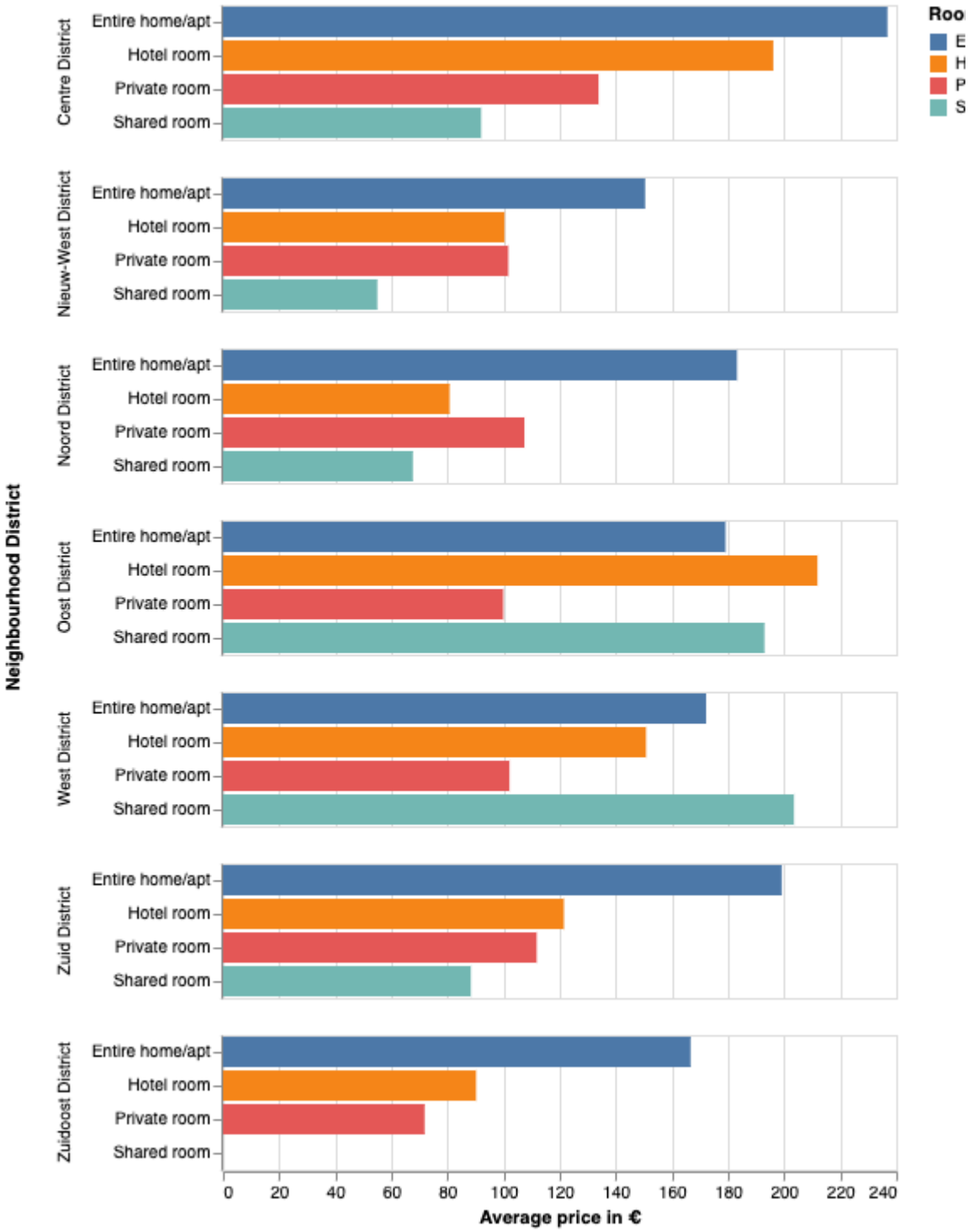


Average price based on room type in each district

The focus here is on room type. Let's pick the Centre District. Another factor of interest would be that average price of an apartment in the Centre is 237€ , but there is considerable difference when choosing private room (around 134€ on average).

In [ ]:
```
alt.Chart(listings_without_outliers).mark_bar().encode(
    x = alt.X('average(price)', axis=alt.Axis(title='Average price in €')
    y = alt.Y('room_type', axis=alt.Axis(title='')),
    color = alt.Color('room_type', legend=alt.Legend(title="Room type:"))
    row = alt.Row('neighbourhood_district', title='Neighbourhood District
    tooltip = ['neighbourhood_district', 'room_type', 'average(price)', '
)
```

Out[ ]:



Average price based on room type in each district as a stacked histogram

Another example would be Stacked histogram to additionally visualize average price of apartments in each district with certain room type. Using the normalize

interactivity one could see the major difference in Zuidost District where average price of entire home/apartment is twice as much as hotel or private room.

In [ ]:
```
@interact(normalize=False)
def stacked_hist(normalize):
    if normalize == False:
        return alt.Chart(listings).mark_bar().encode(
            x = alt.X('average(price)', axis=alt.Axis(title='Average pric
            y = alt.Y('neighbourhood_district', axis=alt.Axis(title='Neig
            color = alt.Color('room_type', legend=alt.Legend(title="Room
            tooltip = ['neighbourhood_district', 'room_type', 'average(pr
        )
    else:
        return alt.Chart(listings).mark_bar().encode(
            x = alt.X('average(price)', stack='normalize', axis=alt.Axis(
            y = alt.Y('neighbourhood_district', axis=alt.Axis(title='Neig
            color = alt.Color('room_type', legend=alt.Legend(title="Room
            tooltip = ['neighbourhood_district', 'room_type', 'average(pr
        )
```

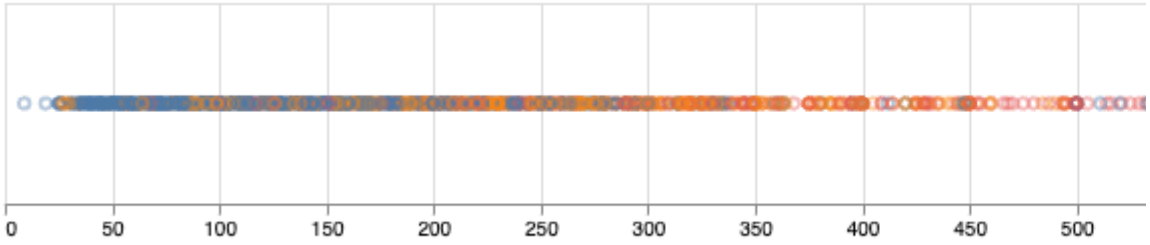
```
interactive(children=(Checkbox(value=False, description='normalize'), Ou
tput()), _dom_classes=('widget-interac…
```

The popularity of 1, 2 and 3+ apartments

Lets add another encoding which will be on the x axis and showing the superiority of one and two bedrooms apartments supplemented by a price. One can see where most airbnb offers are distributed by price.

In [ ]:
```
alt.Chart(listings_without_outliers).mark_point(opacity=0.4).encode(
    x = alt.X('price', axis=alt.Axis(title='Price')),
    color='bedrooms',
    tooltip=['host_name', 'price', 'bedrooms']
).properties(width=1500, height=100)
```

Out[ ]:



The focus here is to show that the most listings are 1 bedroom apartments. For better readability there is a Mark Type button which one could switch between point/circle/square.

In [ ]:
```
baseChart = alt.Chart(listings_without_outliers);

@widgets.interact(mark_type = widgets.Dropdown(
    options = [('point', baseChart.mark_point()),
               ('circle', baseChart.mark_circle()),
               ('square', baseChart.mark_square()),
               ('tick', baseChart.mark_tick()),
               ('line', baseChart.mark_line())],
```

```
    description = 'Mark Type:'))

def show_plot(mark_type):
    return mark_type.encode(
        y = alt.Y('number_of_reviews', axis=alt.Axis(title='Number of rev
        x = alt.X('price', axis=alt.Axis(title='Price')),
        color = alt.Color('bedrooms', legend=alt.Legend(title="Number of
        tooltip = ['price', 'number_of_reviews', 'bedrooms']
    )
```

```
interactive(children=(Dropdown(description='Mark Type:', options=(('poin
t', alt.Chart(...)), ('circle', alt.Ch…
```

Clustering Similar Items

Data Preprocessing

In [ ]:
```
listings.shape
```

Out[ ]:
```
(5395, 59)
```

Until now the dataset preserved its high dimensionality with 57 features. To increase the interpretability and plot them all in 2D, a dimensionality reduction technique which preserves the information is used, namely Principal Component Analysis.
Before applying PCA some more data manipulation is required, especially since our features consist both of numerical and categorical data. The latter are encoded in a series of distinct int64 via the method `panda.factorize()`, which transform them in *category*. A categorical variable takes on a limited, and usually fixed, number of possible values.
Furthermore all *Nan*, *inf* and missing cells must be removed.

In [ ]:
```
#select all non-numeric data
obj = listings.select_dtypes(exclude="number")
obj.dtypes
```

Out[ ]:
```
last_scraped              datetime64[ns]
host_name                         object
host_since                datetime64[ns]
host_response_time                object
host_is_superhost                 object
host_neighbourhood                object
host_has_profile_pic              object
host_identity_verified            object
neighbourhood_cleansed            object
property_type                     object
room_type                         object
has_availability                  object
first_review              datetime64[ns]
last_review               datetime64[ns]
instant_bookable                  object
neighbourhood_district            object
has_rating                          bool
dtype: object
```

In [ ]:
```
# pandas.factorize() encode the array values in a series of distinct int6

listings['last_scraped'] = pd.Categorical(pd.factorize(listings['last_scr
listings['host_name'] = pd.Categorical(pd.factorize(listings['host_name']
listings['host_response_time'] = pd.Categorical(pd.factorize(listings['ho
listings['host_is_superhost'] = pd.Categorical(pd.factorize(listings['hos
listings['host_neighbourhood'] = pd.Categorical(pd.factorize(listings['ho
listings['host_has_profile_pic'] = pd.Categorical(pd.factorize(listings['
listings['host_identity_verified'] = pd.Categorical(pd.factorize(listings
listings['neighbourhood_cleansed'] = pd.Categorical(pd.factorize(listings
listings['property_type'] = pd.Categorical(pd.factorize(listings['propert
listings['room_type'] = pd.Categorical(pd.factorize(listings['room_type']
listings['has_availability'] = pd.Categorical(pd.factorize(listings['has_
listings['instant_bookable'] = pd.Categorical(pd.factorize(listings['inst
listings['first_review'] = pd.Categorical(pd.factorize(listings['first_re
listings['last_review'] = pd.Categorical(pd.factorize(listings['last_revi
listings['neighbourhood_district'] = pd.Categorical(pd.factorize(listings
```

In [ ]:
```
#drop datetime64 features (not supported in pandas even if legit in numpy
date_feat = ["last_review", "first_review", "host_since", "last_scraped"]
listings = listings.drop(date_feat, axis=1)
```

In [ ]:
```
### clean the dataset from NaN, Inf, and missing cells.

def clean_dataset(df):
    assert isinstance(df, pd.DataFrame), "df needs to be a pd.DataFrame"
    df.dropna(inplace=True)
    indices_to_keep = ~df.isin([np.nan, np.inf, -np.inf]).any(1)
    return df[indices_to_keep].astype(np.float64)


listings = clean_dataset(listings.copy())
```

In [ ]:
```
listings.head()
```

Out[ ]:

	id	host_id	host_name	host_response_time	host_response_rate	host_accept
0	2818.0	3159.0	0.0	0.0	100.0	
1	20168.0	59484.0	1.0	1.0	100.0	
2	27886.0	97647.0	2.0	0.0	100.0	
3	28871.0	124245.0	3.0	0.0	100.0	
4	29051.0	124245.0	3.0	0.0	100.0	

In [ ]:
```
features= [i for i in listings.columns]
```

Since we introduced new vector spaces with the encoding from categorical to numerical data, this often leads the variables to be in very different scale, which ultimately results in clustering even between uncorrelated datapoints. Thus, to avoid that, scaling is crucial.

In [ ]:
```
# Standardizing the features
# Separating out the features
x = listings.loc[:, features].values
```

```
x = listings.drop(["host_acceptance_rate","neighbourhood_cleansed", "pric

y = listings.loc[:,['host_acceptance_rate',"neighbourhood_cleansed","pric

st_listings = StandardScaler().fit_transform(listings)
```

In [ ]:
```
pca = PCA(n_components=2)
principalComponents = pca.fit_transform(st_listings)
principalDf = pd.DataFrame(data = principalComponents
             , columns = ['principal component 1', 'principal component 2
```

In [ ]:
```
principalDf.shape
```

Out[ ]:
```
(3247, 2)
```

In [ ]:
```
finalDf = pd.concat([principalDf, listings[['host_acceptance_rate','neigh
```

In [ ]:
```
pca.explained_variance_ratio_.cumsum() #this should add up to 1 but doesn
```

Out[ ]:
```
array([0.11662389, 0.19709629])
```

In [ ]:
```
final= clean_dataset(finalDf)
final.head()
```

Out[ ]:

	principal component 1	principal component 2	host_acceptance_rate	neighbourhood_cleansed	price
0	-0.250808	-3.046095	100.0	0.0	59.0
1	0.493972	-0.764274	100.0	1.0	106.0
2	-0.343165	-2.893822	100.0	2.0	135.0
3	0.562436	-4.200943	99.0	2.0	75.0
4	0.980985	-4.462891	99.0	1.0	55.0

Clustering

In the following an interactive plot is implemented. The user has free choice between 5 different clustering algorithms and the number of clusters.

K-Means:

- Probably the most common, it assigns datasamples to a cluster with the aim of minimizing the variance within each cluster.

BIRCH:

- Balanced Iterative Reducing and Clustering using Hierarchies uses a height-balanced tree data structure, it doesn't cluster directly the dataset but works with smaller partition/summary which are added to bigger ones, step by step.

Mini-Batch K-Means:

- Mini-Batch K-Means is a modified version of k-means that makes updates to the cluster centroids using mini-batches of samples rather than the entire dataset, which can make it faster for large datasets, and perhaps more robust to statistical noise.

Agglomerative Clustering:

- Agglomerative clustering involves merging examples until the desired number of clusters is achieved.

Gaussian mixture:

- Gaussian Mixture models assume that there are a certain number of Gaussian distributions, and each of these distributions represent a cluster. Hence, a Gaussian Mixture Model tends to group the data points belonging to a single distribution together.

In [ ]:
```
@interact(n_clusters=(1,10), algorithm=["Kmeans", "Birch", "MiniBatchKMea
def draw_plot(n_clusters, algorithm):
    if algorithm =="Kmeans":
        kmeans = KMeans(n_clusters=n_clusters, random_state = 102)
        final["predict"] = kmeans.fit_predict(final)

    elif algorithm == "Birch":
        birch = Birch(threshold=0.01, n_clusters=n_clusters)
        final["predict"] = birch.fit_predict(final)

    elif algorithm == "MiniBatchKMeans":
        miniBatchKMeans = MiniBatchKMeans(n_clusters=n_clusters)
        final["predict"] = miniBatchKMeans.fit_predict(final)

    elif algorithm == "AgglomerativeClustering":
        agglomerativeClustering= AgglomerativeClustering(n_clusters=n_clu
        final["predict"] = agglomerativeClustering.fit_predict(final)

    elif algorithm == "GaussianMixture":
        gaussianMixture = GaussianMixture(n_components=n_clusters)
        final["predict"] = gaussianMixture.fit_predict(final)

    #adjust for plotting
    selector = alt.selection_single(empty="all", fields=['predict']) #sor


    base = alt.Chart(final).properties(
        width=250,
        height=250
    ).add_selection(selector)

    first = base.mark_point(filled=True, size=200).encode(
        x='principal component 1',
        y='principal component 2',
        color=alt.condition(selector, 'predict:N', alt.value('lightgray')
    ).interactive()

    second = base.mark_point().encode(
        x='price',
```

```
        y=alt.Y('neighbourhood_cleansed', scale=alt.Scale(domain=(-15, 15
        color=alt.Color('neighbourhood_cleansed:O')
    ).transform_filter(
        selector
    ).interactive()


    return first.properties(title='PCA') | second.properties(title='Price
    ######################
```

```
interactive(children=(IntSlider(value=5, description='n_clusters', max=1
0, min=1), Dropdown(description='algor…
```

In the first graph all previous 57 features are summarized and plotted in 2D, they are not distinguishable any more, notwithstanding clusters mantains some differences with respect to neighbourhood and prices.
If we take the example with setting **n_clusters: 5** ald **algorithm: K-Means** we see that most of the datapoints are in (1)the red cluster (as well as the outtermost outlier), which entails all the 21 neighbourhoods and has a price range 0,~120 €, (2) the blue cluster is one of the most dense group, and only goes from ~125 to 225€, (3)the light sky from ~225,370 €, (4)the orange one instead entails ~ 350,630 € but span only over a half of the neighbourhood, as well as the (5) the green, with only a few instances, from ~650 to 1,200 €.
This continuous correspondence between cluster and price range is mantained with any setting of number of clusters; of course the more clusters the narrower their range.
Similar results are visible with **BIRCH**, **Minibatch-K-Means**, **Minibatch-K-Means** and **Agglomerative Clustering** which suggests a similarity in the operating of the algorithms.
Very different is the scenario when using **Gaussian Mixture**.
Taking into accounts the most outlier datapoint, which in K-Means belongs to the first red clusters, with Gaussian Mixture it not only denotes fewer neighbourhoods (only 10 out of 21) but the datapoints in this cluster cover the whole range of prices.
It is worth mentioning that all the previous algorithm put the two outter most outliers in different clusters, whereas in the Gaussian Mixture they pertain to the same one, even when setting the number of cluster to the max: 10. This difference is probably explained by the more flexible, probabilistic nature of this last algorithm, in contrast with the data-driven approach of the prior ones.
An optimal solution would be to use an hybrid approach to get the best result.

Sources:

https://machinelearningmastery.com/clustering-algorithms-with-python/
https://analyticsindiamag.com/guide-to-birch-clustering-algorithmwith-python-codes/
https://www.analyticsvidhya.com/blog/2019/10/gaussian-mixture-models-clustering/
https://towardsdatascience.com/gaussian-mixture-models-vs-k-means-which-one-to-choose-62f2736025f0

In [ ]: