

Apartment rental offers in Germany

Saeid Cheshmi – 98222027

1. Data Exploration

The data was scraped from Immoscout24, the biggest real estate platform in Germany. The data set contains most of the important properties, such as living area size, the rent, both base rent as well as total rent (if applicable), the location (street and house number, if available, ZIP code and state), type of energy etc. the dataset has 268850 rows and 49 columns which they are properties of home. Our goal here is to predict total rent of a home. The dataset has the following attributes:

- **“regio1”**: Bundesland
- **“serviceCharge”**: aucilliary costs such as electricity or internet in €
- **“heatingType”**: Type of heating
- **“telekomTvOffer”**: Is payed TV included if so which offer
- **“telekomHybridUploadSpeed”**: how fast is the hybrid inter upload speed
- **“newlyConst”**: is the building newly constructed
- **“balcony”**: does the object have a balcony
- **“picturecount”**: how many pictures were uploaded to the listing
- **“pricetrend”**: price trend as calculated by Immoscout
- **“telekomUploadSpeed”**: how fast is the internet upload speed
- **“totalRent”**: total rent
- **“yearConstructed”**: construction year
- **“scoutId”**: immoscout Id
- **“noParkSpaces”**: number of parking spaces
- **“firingTypes”**: main energy sources, separated by colon
- **“hasKitchen”**: has a kitchen
- **“geo_bln”**: bundesland (state), same as regio1
- **“cellar”**: has a cellar
- **“yearConstructedRange”**: binned construction year, 1 to 9
- **“baseRent”**: base rent without electricity and heating
- **“houseNumber”**: house number
- **“livingSpace”**: living space in sqm

- **“geo_krs”**: district, above ZIP code
- **“condition”**: condition of the flat
- **“interiorQual”**: interior quality
- **“petsAllowed”**: are pets allowed, can be yes, no or negotiable
- **“street”**: street name
- **“streetPlain”**: street name (plain, different formatting)
- **“lift”**: is elevator available
- **“baseRentRange”**: binned base rent, 1 to 9
- **“typeOfFlat”**: type of flat
- **“geo_plz”**: ZIP code
- **“noRooms”**: number of rooms
- **“thermalChar”**: energy need in kWh/(m²a), defines the energy efficiency class
- **“floor”**: which floor is the flat on
- **“numberOfFloors”**: number of floors in the building
- **“noRoomsRange”**: binned number of rooms, 1 to 5
- **“garden”**: has a garden
- **“livingSpaceRange”**: binned living space, 1 to 7
- **“regio2”**: District or Kreis, same as geo krs
- **“regio3”**: City/town
- **“description”**: free text description of the object
- **“facilities”**: free text description about available facilities
- **“heatingCosts”**: monthly heating costs in €
- **“energyEfficiencyClass”**: energy efficiency class (based on binned thermalChar, deprecated since Feb 2020)
- **“lastRefurbish”**: year of last renovation
- **“electricityBasePrice”**: monthly base price for electricity in € (deprecated since Feb 2020)
- **“electricityKwhPrice”**: electricity price per kwh (deprecated since Feb 2020)
- **“date”**: time of scraping

According to above features, there are some uninformative features, they should be removed in preprocessing step. Also, our target, totalRent, is continuous number we are dealing with regression problem.

2. Preprocessing

We preprocess the dataset in following steps

- 1) First we delete columns which over 50% of their values are null. The columns, **'telekomHybridUploadSpeed'**, **'noParkSpaces'**, **'heatingCosts'**, **'energyEfficiencyClass'**, **'lastRefurbish'**, **'electricityBasePrice'**, **'electricityKwhPrice'**, were removed.
- 2) According to last section, there are some uninformative features in the dataset, following features were removed from the dataset: **'date'**, **'description'**, **'scoutId'**, **'picturecount'**, **'facilities'**, **'houseNumber'**, **'livingSpaceRange'**, **'yearConstructedRange'**, **'baseRentRange'**, **'noRoomsRange'**, **'street'**, **'streetPlain'**.
- 3) There are some duplicated records in the dataset, so we should remove them, and keep last one. They were 2247 records!!
- 4) Our target, totalRent, has too many null values, so we can remove them or filling them with appropriate values. There are over 40k records which their totalRent feature is null, so removing them it's not a good deal. It makes sense that we fill them with median of their values.
- 5) Also, we delete records where baseRent is higher than or equal to totalRent, because it doesn't make sense. We delete records where totalRent is equal to 0, or higher than 10K.
- 6) Some of records have living space of 0, we should remove them, too.
- 7) There are 7 numerical columns which they have null values, we fill the null values with mean of the feature. Figure 1 shows these columns.
- 8) Also, there are 7 categorical columns which they have null values, we fill the null values with mode of the feature. Figure 2 shows these columns.
- 9) We detect outlier by computing z-score and then delete them. This method considers 99.7% of records as normal and rest of them are outliers.
- 10) Some of categorical features have too many states, so we should remove them. Figure 3 shows categorical columns with number of their states. We delete **'firingTypes'**, **'geo_krs'**, **'regio2'**, **'regio3'**, **'geo_bln'**.

serviceCharge	3929
newlyConst	0
balcony	0
pricetrend	1676
telekomUploadSpeed	29017
totalRent	0
yearConstructed	52004
hasKitchen	0
cellar	0
baseRent	0
livingSpace	0
lift	0
geo_plz	0
noRooms	0
thermalChar	96182
floor	44484
numberOfFloors	86549
garden	0

Figure 1. Numerical columns
with number of null values

regio1	0
heatingType	38736
telekomTvOffer	28348
newlyConst	0
balcony	0
firingTypes	51001
hasKitchen	0
geo_bln	0
cellar	0
geo_krs	0
condition	61513
interiorQual	100996
petsAllowed	103536
lift	0
typeOfFlat	31956
garden	0
regio2	0
regio3	0

Figure 2. Categorical columns
with number of null values

regio1	16
heatingType	13
telekomTvOffer	3
newlyConst	2
balcony	2
firingTypes	118
hasKitchen	2
geo_bln	16
cellar	2
geo_krs	419
condition	10
interiorQual	4
petsAllowed	3
lift	2
typeOfFlat	10
garden	2
regio2	419
regio3	8526

Figure 3. Categorical columns with number of their states.

11) We reduce number of states in some columns. For **'regio1'** feature, we put states with lower than 3% into other state. Figure 4 shows **'regio1'** states with their percentages. For **'heatingType'** feature, we put states with lower than 1% into other state. Figure 5 shows **'heatingType'** states with their percentages. For **'condition'** feature, we put states with lower than 1% into other state. Figure 6 shows **'condition'** states with their percentages. For **'typeOfFlat'** feature, we put states with lower than 1% into other states. Figure 7 shows **'typeOfFlat'** states with their percentages.

Nordrhein_Westfalen	24.182465
Sachsen	23.703093
Sachsen_Anhalt	8.402672
Bayern	6.686044
Niedersachsen	6.230532
Hessen	5.678279
Baden_Württemberg	5.021474
Thüringen	3.490087
Rheinland_Pfalz	3.018958
Brandenburg	2.824606
Mecklenburg_Vorpommern	2.753460
Berlin	2.611600
Schleswig_Holstein	2.592946
Hamburg	1.151360
Bremen	1.142250
Saarland	0.510173

Figure 4. **'regio1'** states

central_heating	65.199774
district_heating	9.520628
gas_heating	7.637847
self_contained_central_heating	6.954145
floor_heating	5.733808
oil_heating	1.950458
heat_pump	0.889766
combined_heat_and_power_plant	0.710598
night_storage_heater	0.533599
wood_pellet_heating	0.351829
electric_heating	0.337946
stove_heating	0.121036
solar_heating	0.058566

Figure 5. **'heatingType'** states

well_kept	51.392998
refurbished	10.796061
fully_renovated	10.082426
mint_condition	7.345885
first_time_use	6.715544
modernized	6.663485
first_time_use_after_refurbishment	5.572860
negotiable	0.881957
need_of_renovation	0.547048
ripe_for_demolition	0.001735

Figure 6. 'condition' states

apartment	62.982951
roof_storey	13.408963
ground_floor	11.893627
other	3.616763
maisonette	2.921348
raised_ground_floor	2.183419
terraced_flat	1.140948
penthouse	0.796061
half_basement	0.781745
loft	0.274175

Figure 7. 'typeOfFlat' states

3. Exploratory Data Analysis

3.1. Target Distribution

As figure 8 shows target distribution, the distribution is skewed and majority of records in the dataset have total rent below 1000. Also, there are small number of total rent higher than 2000.

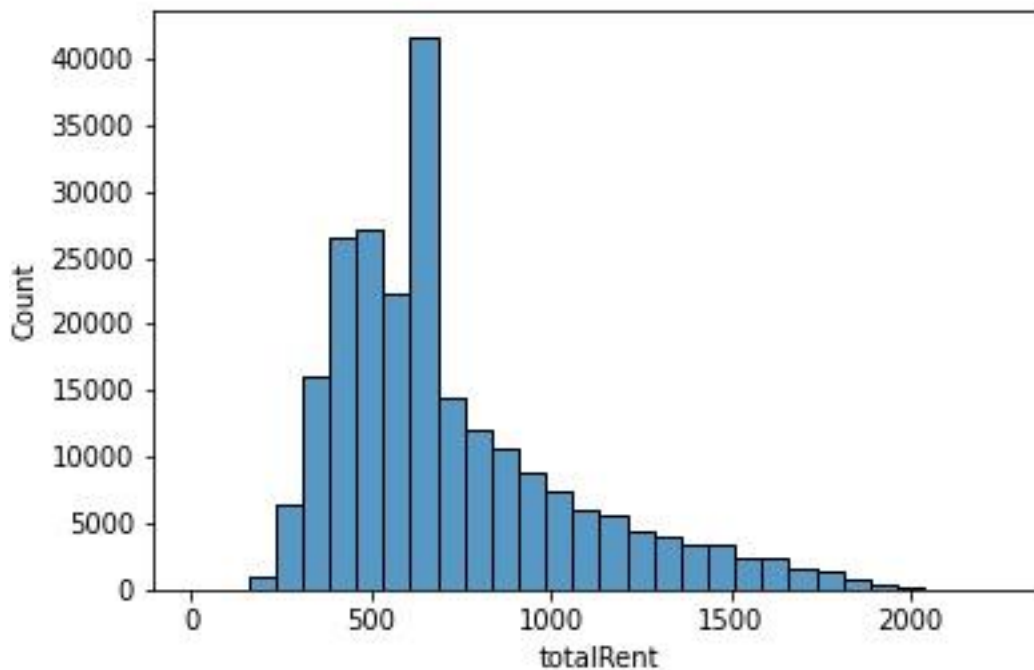


Figure 8. Total rent distribution

If we look at base rent distribution, we find out interesting issue, base rent distribution is similar to total rent. So, we can assume there is a strong relation between them. Figure 9 shows the base rent distribution.

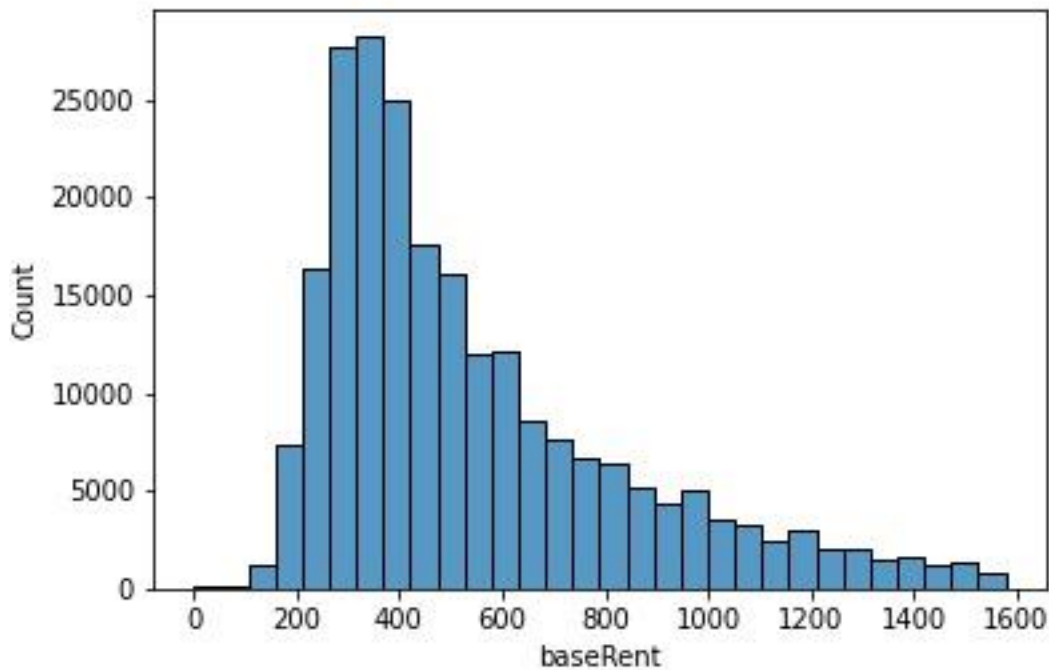


Figure 9. Base rent distribution

3.2. Distributions of Numerical Features

According to figure 10, price_trend, livingSpace, serviceCharge, thermalChar, floor and numberOfFloors have almost normal distribution. Lots of records in the dataset have 40 Mbps internet upload speed and other speeds have much smaller share.

By looking at yearConstructed distribution, we find out most of records aren't newly built. Also, most of homes in the dataset have 2 or 3 room and minority of homes have more than 4 rooms.

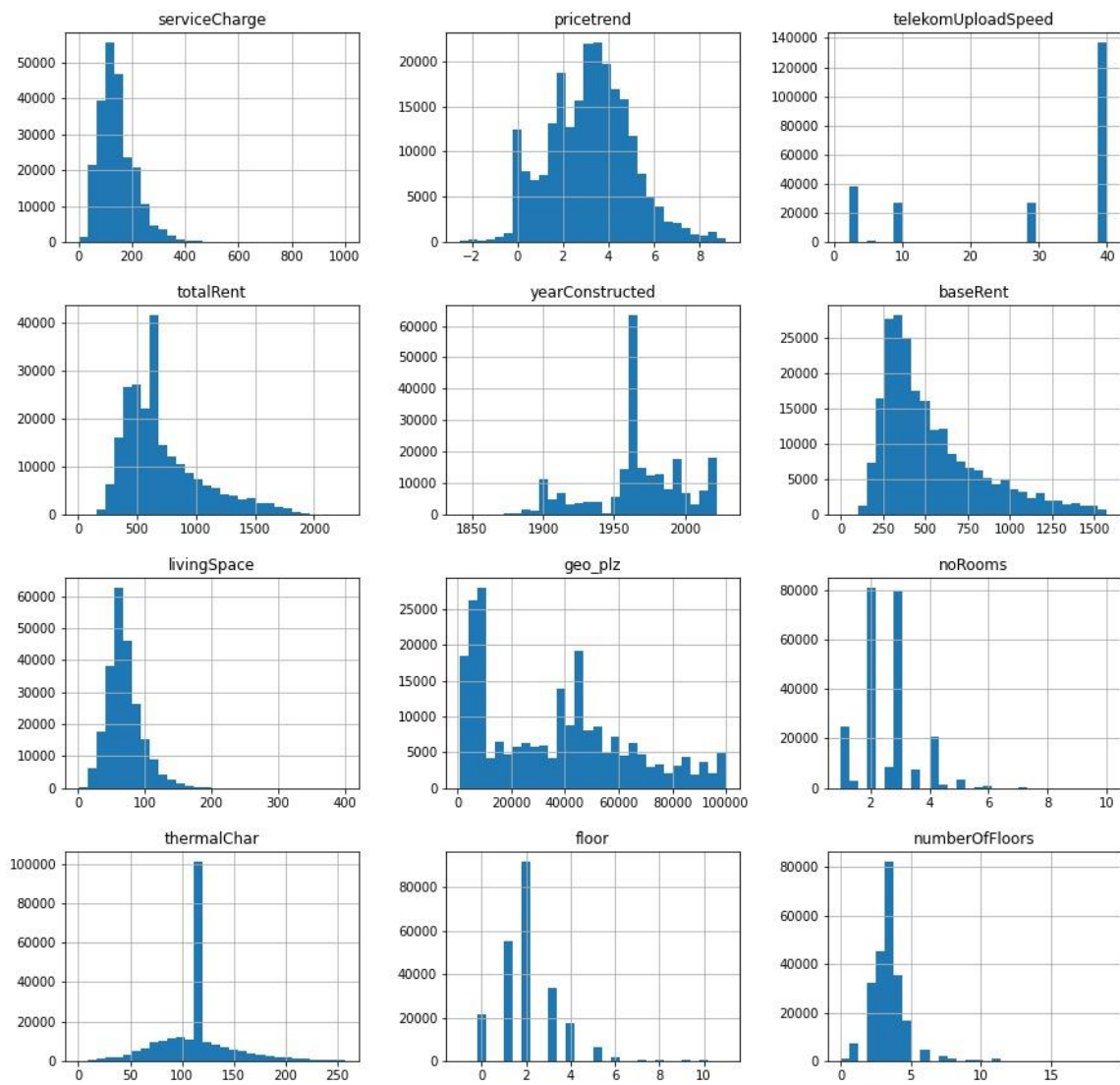


Figure 10. Distributions of numerical features

3.3. Distributions of Categorical Features

As figure 11 shows, Boolean features of the dataset aren't distributed equally, majority of homes aren't newly built or don't have garden. Also, minority of homes have lift or kitchen. But in cellar and balcony, we encounter with better situation, number of positive and negative states are closer to each other.

According to figure 12, most of homes in the dataset have normal interior quality and a few of them are luxury or simple. In most of home you can negotiate for having pet. A majority of homes in the dataset are apartment and other types are much smaller than this group in comparing number of records. Most of home are located in Nordrhein_westfalen and Sachsen states, and other homes are distributed in another states.

By looking at figure 13, we find out that over 200k of records have one-year free offer for tv and only a small number of them doesn't have the offer. Majority of homes use central heating for heating the home. As it can be seen number of well-kept homes are much higher than other groups.

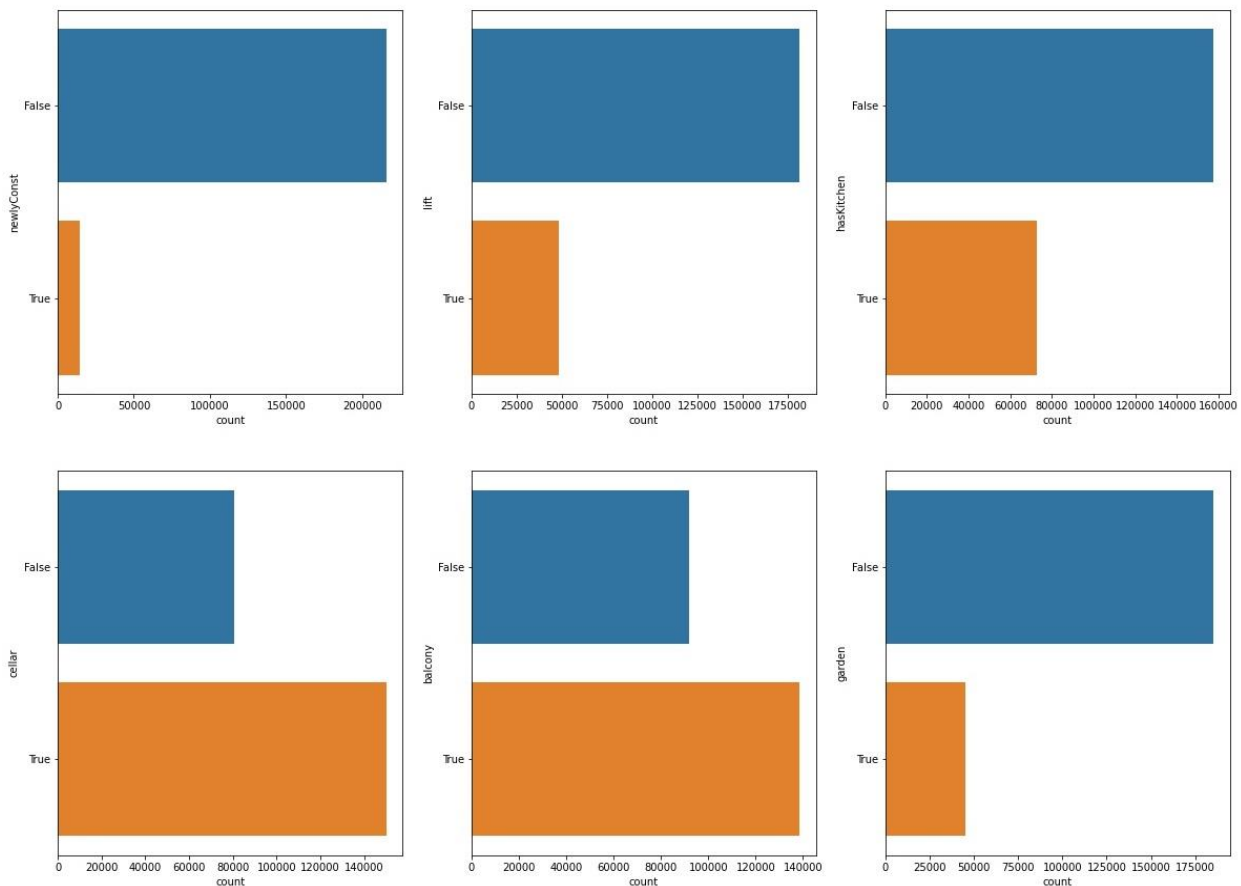


Figure 11. Distributions of Boolean features

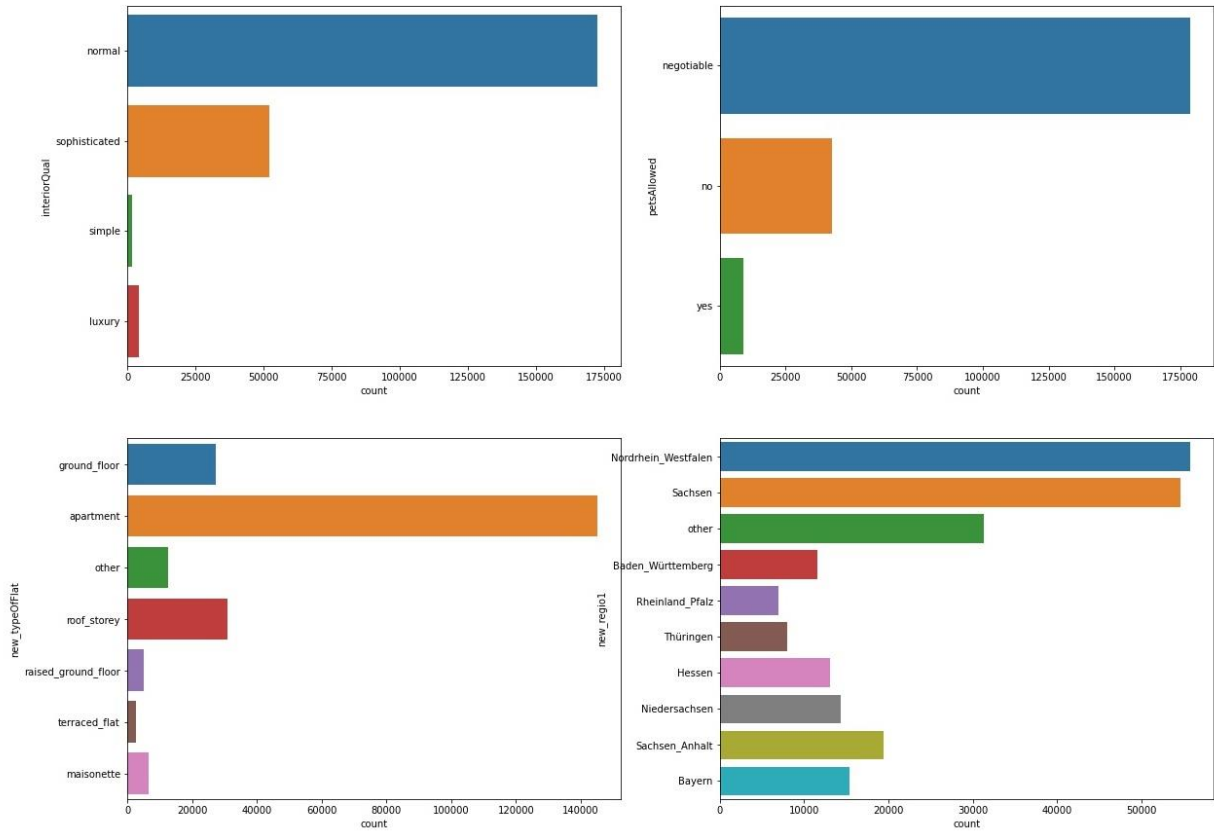


Figure 12. Distributions of Categorical Features part-1

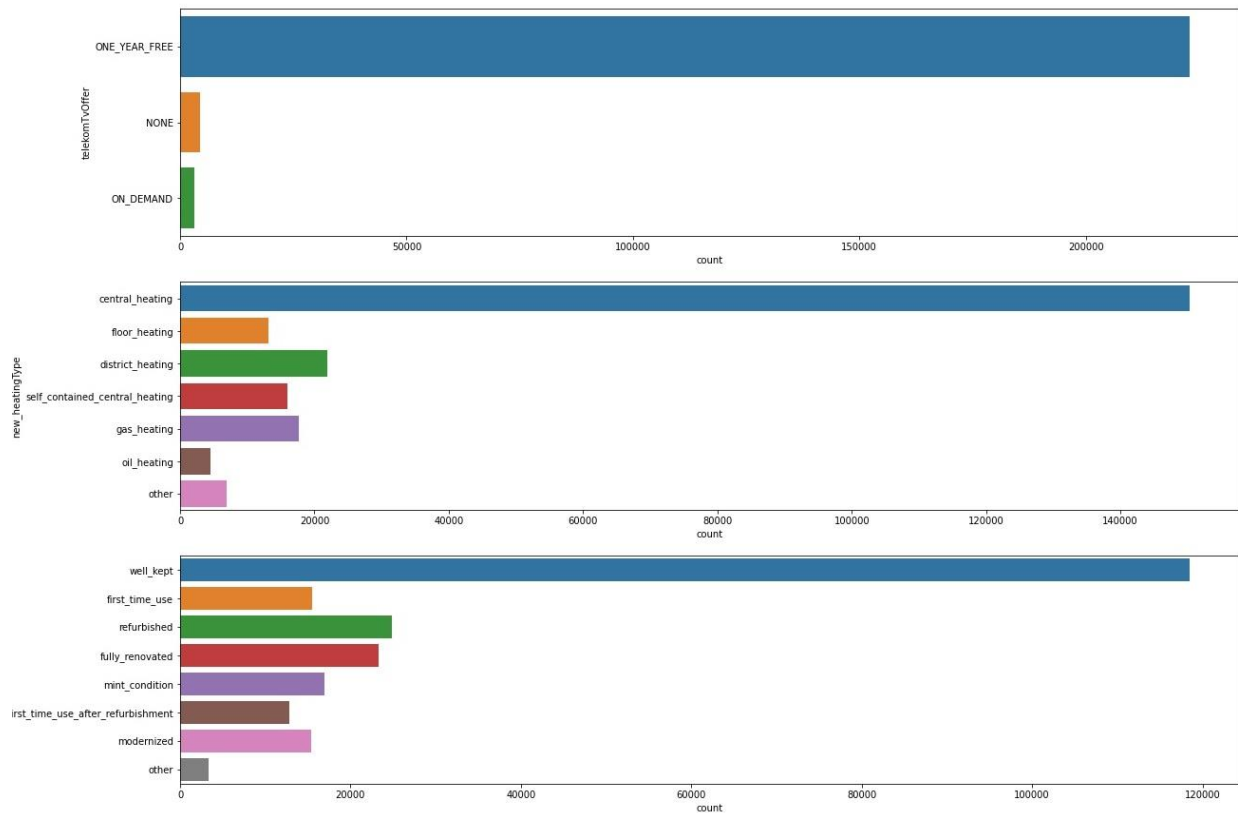


Figure 13. Distributions of Categorical Features

3.4. Base Rent Mean In Each State

According to figure 14, we can see that Baden_Württemberg, Bayern, and Hessen have highest average base rent in the Germany and Sachsen_Anhalt and Thüringen have lowest average in the country.

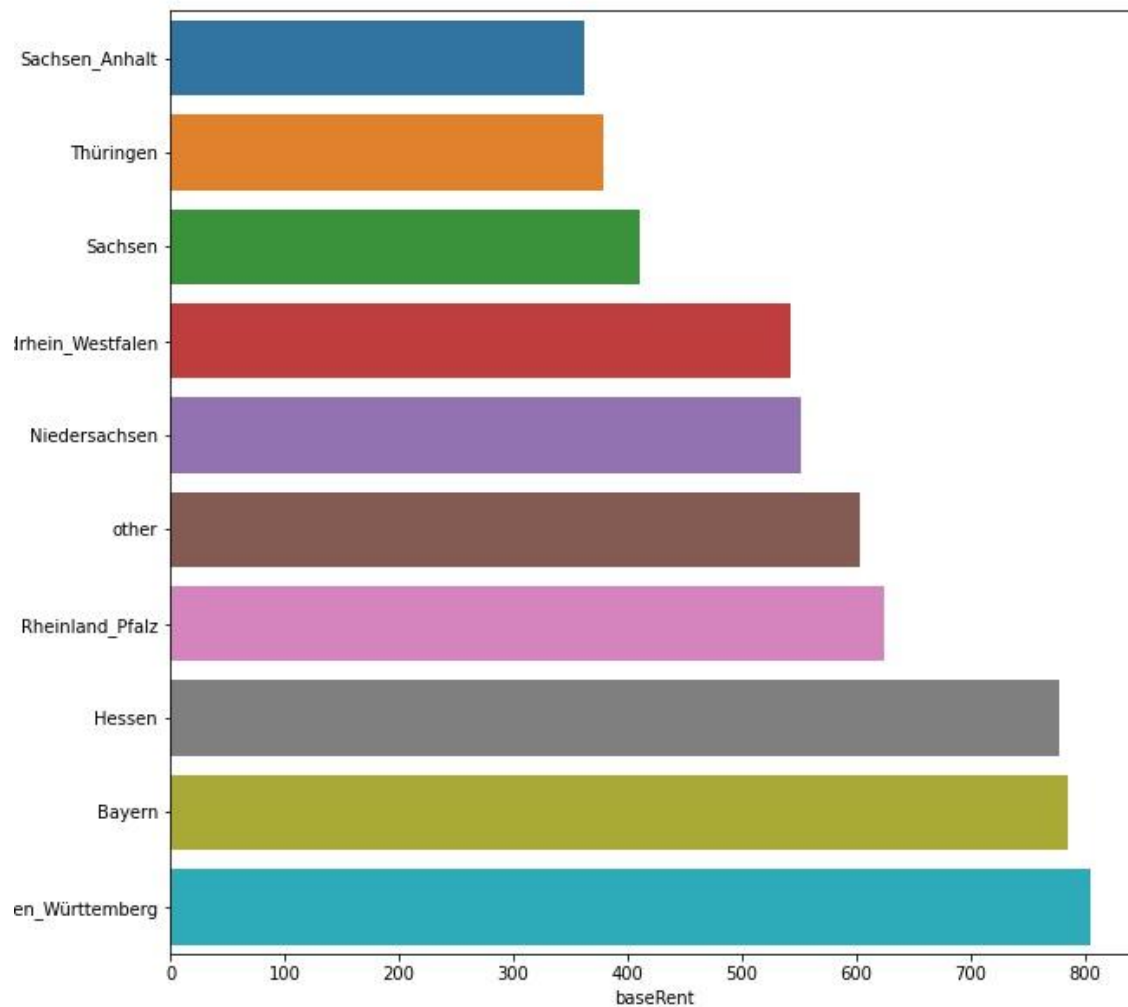


Figure 14. Base rent average based on states

3.5. Correlation matrix

As it can be seen from correlation matrix there are some strong correlations between total rent and some other features. Base rent is highly correlated with total rent, also living space and service charge are also have high correlation with total rent. As a result, applying a linear model might be led to good results, but at first, we delete base rent from dataset, it can be caused data leakage, so it doesn't make sense predicting total rent with base rent, but after that we will train another model with base rent feature to compare with first one.

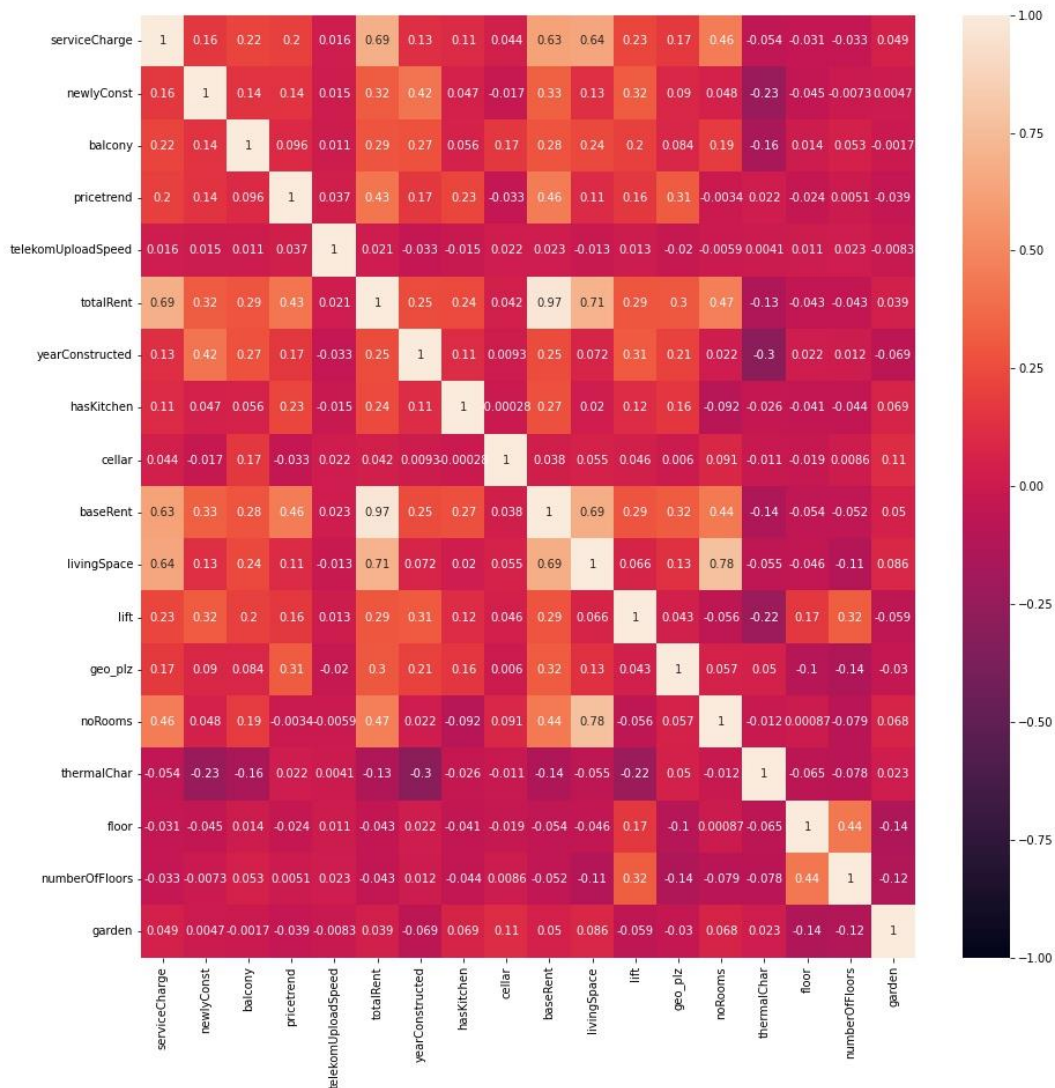


Figure 15. Correlation matrix

4. Model Training

4.1. Splitting train and test set

We keep 80% of our dataset for training and rest of it for testing our model. These two sets aren't overlapping.

4.2. Pipeline

We make a pipeline consist of two main steps, preprocessing and main model. In first part, we use `StandardScaler` for numerical columns and `OneHotEncoder` for categorical columns. For simplicity we define `ColumnTransformer` which is given numerical columns and categorical columns, it applies mentioned transformer on specific columns. After that, our preprocessed dataset given to linear regression model and model will be trained.

StandardScaler: subtracts feature mean from each feature and divides on standard deviation, after that sample mean is 0 and feature has unit variance, by doing so we center and scale each feature.

OneHotEncoding: one-hot encoding is the process by which categorical data are converted into numerical data. Categorical features are turned into binary features that are “one-hot” encoded, meaning that if a feature is represented by that column, it receives a 1, otherwise it receives a 0.

4.3. Results of first model

After training our model, we got the following results on the test dataset. First, we evaluated our model with default metric of model, R2, then we used mean absolute error for evaluating it.

```
[111] r2_score = lr_model.score(data_test,target_test)
      print(f'R2 score of linear regression model on test dataset is {r2_score:.3f} .')

R2 score of linear regression model on test dataset is 0.776 .

[112] y_predicted = lr_model.predict(data_test)
      print(f'Mean absolute error of linear regression model on test dataset is {mean_absolute_error(y_predicted,target_test):.3f} ')

Mean absolute error of linear regression model on test dataset is 117.298
```

Figure 16. Results of first model

According to above figure, we have R2 score of 77.6% and MAE of 117.298. Mean absolute error, measures mean of absolute difference between predicted and true value for every observation. R2 score works by measuring the amount of variance in the predictions explained by the dataset, it is the difference between the samples in the dataset and the predictions made by the model. It seems it isn't a very good performance but it's not bad!!

4.4. Ridge model

We will also train another linear model, ridge regression, ridge is a regression model which it has l2 regularization term. We can tune alpha hyperparameter for this model. Alpha hyperparameter is coefficient of l2 norm in cost function. We use previous preprocessing matters and we will tune alpha by GridSearchCV. After tuning we got the following results:

	mean_fit_time	mean_score_time	param_regressor__alpha	params	mean_test_score	std_test_score	rank_test_score
0	1.073391	0.200306	0.001	{'regressor__alpha': 0.001}	117.045032	0.293849	1
1	1.015256	0.184574	0.01	{'regressor__alpha': 0.01}	117.045032	0.293849	2
2	1.032773	0.188528	0.1	{'regressor__alpha': 0.1}	117.045036	0.293849	3
4	1.057771	0.193960	1	{'regressor__alpha': 1}	117.045072	0.293845	4
5	1.026338	0.194524	10	{'regressor__alpha': 10}	117.045481	0.293810	5
6	1.021735	0.185323	100	{'regressor__alpha': 100}	117.053929	0.293779	6
3	3.053817	0.214210	0	{'regressor__alpha': 0}	117.233490	0.398493	7
7	0.959233	0.170331	1000	{'regressor__alpha': 1000}	117.330708	0.302168	8

Figure 17. results of tuning alpha

There isn't huge difference between different values of alpha, we select first one and evaluate the model with this parameter on test set. Figure 18 shows the results, unfortunately we didn't get much better MAE.

```

mae = grid_search.score(data_test,target_test)
mae = -mae
print(f'MAE of ridge model : {mae:.3f}')

MAE of ridge model : 117.237

```

Figure 18. Result of tuned ridge model

4.5. Evaluate model with baseRent feature!!

As we mentioned earlier, having baseRent in the dataset feature might be led to data leakage and doesn't make sense in reality, but for comparing with the previous results we add base rent to dataset and compare its performance to previous one.

```

[121] dataset_br= pd.concat([dataset,baseRent],axis=1)

[122] data= dataset_br.drop(['totalRent'],axis=1)
      target = dataset_br.totalRent

      data_train , data_test,target_train , target_test = train_test_split(data,target,test_size=0.2,random_state=42)

      cat_cols = cat_selector(data_train)
      numeric_cols = num_selector(data_train)
      preprocessor = ColumnTransformer([('cat_preprocessor',OneHotEncoder(),cat_cols),
                                      ('num_preprocessor',StandardScaler(),numeric_cols)])

      ridge = make_pipeline(preprocessor,Ridge())
      ridge.fit(data_train,target_train)
      y_predicted = ridge.predict(data_test)

      print(f'MAE (with baseRent feature) : {mean_absolute_error(y_predicted,target_test):.3f}')

MAE (with baseRent feature) : 42.124

```

Figure 19. Model with baseRent

As expected, by adding baseRent we got much better MAE, but this model isn't appropriate in real world scenarios.

5. Multiprocessing

5.1. Multiprocessing by python

We can use multiprocessing utilities of python for doing our preprocess parts. Here for comparing purpose, between multiprocessing and ordinary manner, we gather all preprocessing steps except, two parts of them into one function and then apply it on dataset, single core and multiprocessing. Also, we define another function which split dataset into some parts then creates a pool and maps the function on the parts, at last concatenates them.

The following figures show the results:

```
[98] %%time
      new_df = df.copy()
      new_df = preprocess(new_df)

CPU times: user 1.8 s, sys: 285 ms, total: 2.08 s
Wall time: 2.08 s
```

Figure 20. Single core preprocessing

```
[104] %%time
      multiprocessing_df = df.copy()
      multiprocessing_df = multiprocessing_preprocessing(multiprocessing_df,preprocess,4)

CPU times: user 1.51 s, sys: 906 ms, total: 2.42 s
Wall time: 5.27 s
```

Figure 21. Multiprocess preprocessing, n_cores=4

```
[106] %%time
      multiprocessing_df = df.copy()
      multiprocessing_df = multiprocessing_preprocessing(multiprocessing_df,preprocess,2)

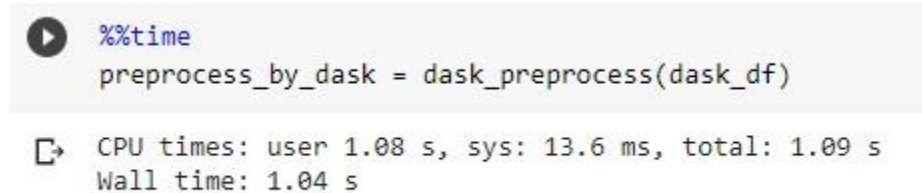
CPU times: user 1.47 s, sys: 1.01 s, total: 2.48 s
Wall time: 4.89 s
```

Figure 22. Multiprocess preprocessing, n_cores = 2

As you can see by using multiple cores, preprocessing time grows 2X to 2.5X times, this issue may be due to the cost of initialization and diving. N_cores here, determines number of pool workers and number of dataframe parts which we want to split.

5.2. Multiprocessing by dask

Like previous section, we want to preprocess our dataset with dask and compare its runtime with preprocessing with single core. We gather all preprocessing steps into one function and call it. Below figure shows the result:



```
%%time
preprocess_by_dask = dask_preprocess(dask_df)

CPU times: user 1.08 s, sys: 13.6 ms, total: 1.09 s
Wall time: 1.04 s
```

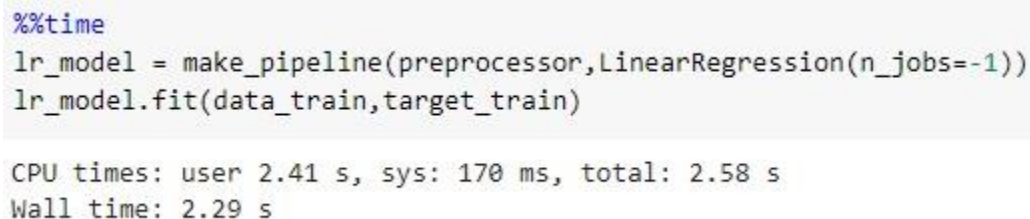
Figure 23. dask preprocessing

As you can see by using dask, preprocessing time was halved, as a result we can use dask for faster preprocessing!!

6. EXTRA

6.1. Comparing model training by sklearn and dask

We can train model by dask_ml library, too. In this section, we want to compare runtimes of sklearn model training and dask_ml. Figure 24 shows training runtime of linear regression pipeline by sklearn and figure 25 shows training runtime of the same model by dask_ml.



```
%%time
lr_model = make_pipeline(preprocessor, LinearRegression(n_jobs=-1))
lr_model.fit(data_train, target_train)

CPU times: user 2.41 s, sys: 170 ms, total: 2.58 s
Wall time: 2.29 s
```

Figure 24. Model training by sklearn

```

▶ %%time
lr_dask = make_pipeline(preprocessor, LinearRegression(n_jobs=-1))
lr_dask.fit(dd_train, dd_target_train)

CPU times: user 11.5 s, sys: 595 ms, total: 12 s
Wall time: 10.5 s

```

Figure 25. Model training by `dask_ml`

As you can see for this simple model `dask_ml`'s runtime is longer than `sklearn` one, so it isn't good choose for these simple pipelines. We also try `dask_ml` for ridge model tuning, we use the same pipeline for both of them, and measure runtimes.

```

[189] %%time

ridge_model = Pipeline([('preprocessor', preprocessor), ('regressor', Ridge())])
param_grid = {
    'regressor__alpha' : [0.001, 0.01, 0.1, 0.1, 0.1, 1, 10, 100, 1000]
}
grid_search = GridSearchCV(ridge_model, param_grid= param_grid, cv=5, n_jobs=-1, scoring = 'neg_mean_absolute_error')
grid_search.fit(data_train, target_train)

CPU times: user 5.51 s, sys: 532 ms, total: 6.05 s
Wall time: 35.8 s

```

Figure 26. Ridge hyperparameter tuning by `sklearn`

```

▶ %%time

ridge_dask = Pipeline([('preprocessor', preprocessor), ('regressor', Ridge())])
param_grid = {
    'regressor__alpha' : [0.001, 0.01, 0.1, 0.1, 0.1, 1, 10, 100, 1000]
}
grid_search = GridSearchCV(ridge_dask, param_grid= param_grid, cv=5, n_jobs=-1, scoring = 'neg_mean_absolute_error')
grid_search.fit(dd_train, dd_target_train)

CPU times: user 40.5 s, sys: 6.76 s, total: 47.3 s
Wall time: 30.7 s

```

Figure 27. Ridge hyperparameter tuning by `dask_ml`

As above figures show, this time using `dask_ml` was a good deal!! We got 5 second less than `sklearn` runtime.

6.2. Feature engineering

In this section we want to make some changes respect to previous preprocessing steps to improve our model performance. We apply almost the same preprocessing steps except, we don't reduce states of categorical feature which were reduced in section 2. Also, we create new 'rent_per_meter' which is "baseRent" divide on "livingSpace". Figure 28 shows rent_per_meter distribution, which is similar to baseRent distribution.

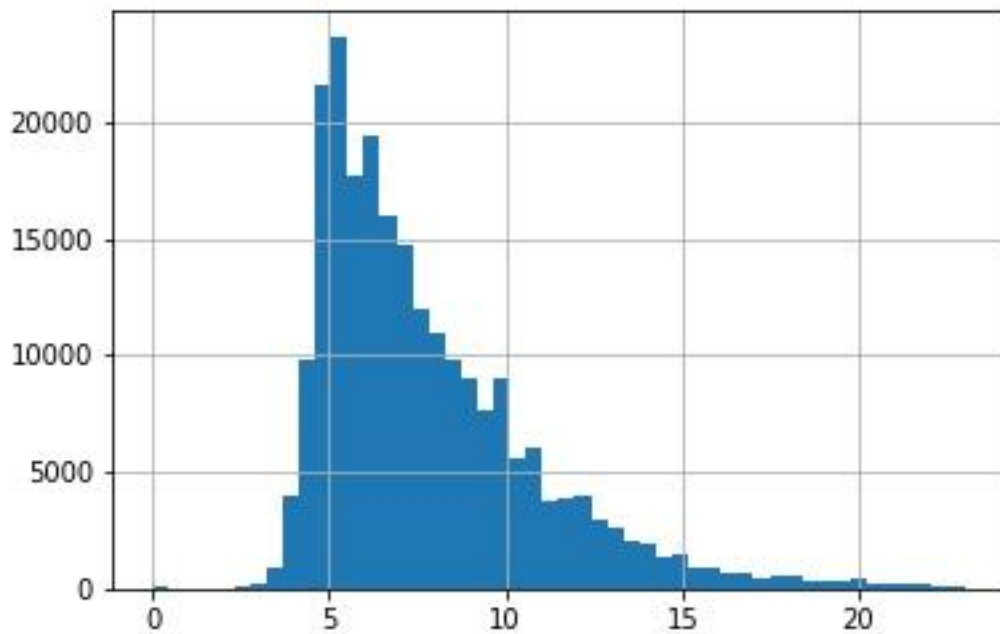


Figure 28. rent_per_meter distribution

After preprocessing is completed, we will train a ridge model and a KNN model.

```
[250] ridge_score = ridge.score(data_test,target_test)
      print(f'ridge score : {ridge_score:.3f}')

      ridge score : 0.905

[251] ridge_predicted = ridge.predict(data_test)
      print(f'MAE of ridge model : {mean_absolute_error(target_test,ridge_predicted):.3f}')

      MAE of ridge model : 68.692
```

Figure 29. R2 score and MAE of ridge model

As it can be seen model's performance improves by adding the new feature to dataset. We got higher R2 score and lower mean absolute error.

We also trained a KNN model on the dataset, with k=7, but as figure 30 shows, we got higher MAE respect to ridge model.

```
[257] knn_predicted = knn.predict(data_test)
      print(f'MAE of KNN : {mean_absolute_error(target_test,knn_predicted)} ')

      MAE of KNN : 84.65399709550394
```

Figure 30. MAE of KNN model

7. References

1. Apartment rental offers in Germany
[<https://www.kaggle.com/datasets/corrieaar/apartment-rental-offers-in-germany>]
2. Make your Pandas apply functions faster using Parallel Processing
[<https://towardsdatascience.com/make-your-own-super-pandas-using-multiproc-1c04f41944a1>]