



UNIVERSITÀ DI PISA

Project Report:

Apartment rental offers in Germany

Distributed Data Analysis and Mining

- Luca Coda-Giorgio [665948] l.codagiorgio@studenti.unipi.it
- Alberto La Piccirella [636680] a.lapiccirella@studenti.unipi.it
- Marco Di Cristo [636814] m.dicristo@studenti.unipi.it
- Alberto Mura [666650] a.mura9@studenti.unipi.it
- Payam Firouzfar [667114] p.firouzfar@studenti.unipi.it

Academic Year 2023/2024

1. Introduction

In the world of real estate analytics, access to comprehensive and curated datasets is paramount for deriving meaningful insights. This report centers around a dataset compiled from Germany's real estate online platform: Immoscout24. The dataset exclusively captures rental property offers, reflecting the dynamic landscape of apartment rentals across Germany. The dataset was obtained through a scraping process in a dedicated GitHub repository, ensuring transparency and reproducibility. In particular, we have 49 attributes and 268,850 rows.

This comprehensive dataset encompasses a wealth of information pivotal for real estate analysis. Key attributes include living area size, rent (both base and total if applicable), location details (street, house number, ZIP code, and state), energy type, and more. In a first sight we spotted a few interesting target variable as:

- **totalRent** (sum of base rent, service charge and heating cost)
- **yearConstructed** (apartment's construction year)
- **serviceCharge** (auxiliary costs, such as electricity or internet, in €)
- **typeOfFlat** (apartment's type, i.e. ground floor, terraced flat)

Soon after we realized most of these attributes were incomplete or not trustworthy.


| ▲ regio1 | ▲ serviceCharge | ▲ heatingType | ▲ telekomTvOffer | ▲ telekomHybridUp... | ✓ newlyC |
|--------------------------|--|--------------------------------|---------------------------------------|---|---|
| Bundesland | auxiliary costs such as electricity or internet in € | Type of heating | Is paid TV included if so which offer | how fast is the hybrid inter upload speed | is the building constructed |
| Nordrhein-Westfal... 23% | 150 5% | central_heating 48% | ONE_YEAR_FREE 85% | NA 83% |  |
| Sachsen 22% | 100 5% | NA 17% | NA 12% | 10 17% | |
| Other (147833) 55% | Other (241562) 90% | Other (95017) 35% | Other (8599) 3% | | |
| Nordrhein-Westfalen | 245 | central_heating | ONE_YEAR_FREE | NA | FALSE |
| Rheinland-Pfalz | 134 | self_contained_central_heating | ONE_YEAR_FREE | NA | FALSE |
| Sachsen | 255 | floor_heating | ONE_YEAR_FREE | 10 | TRUE |
| Sachsen | 58.15 | district_heating | ONE_YEAR_FREE | NA | FALSE |
| Bremen | 138 | self_contained_central_heating | NA | NA | FALSE |
| Schleswig-Holstein | 142 | gas_heating | NONE | NA | FALSE |

Figure 1.1: Dataset

2. Data Preparation

Exploring this dataset was not an easy task due to many reasons, but the main reason is Data quality. The dataset is haunted by rampant inconsistencies, errors, and missing values. There are no standardized formats for variables, leading to confusion about the meaning of certain data points. No effort was made to validate or clean the data, resulting in a high degree of noise and inaccuracies. Another problem was the mole of missing values. Even if we have 49 attributes, a big chunk of those were half empty.

To be more clear and precise we will enumerate the various problems of the dataset:

1. Csv - commas and break line problems
2. Different type of Null values. Presented in 7 different way:
NA, NULL, no_information, NO_INFORMATION Nan, None or just empty
3. Too many missing values of many attributes
4. Different data type in the same attributes

The Csv format gave some problem in the way it was written, due to the presence of comma, other punctuation and also the presence of long text features. The result was a dataset with different value attributes because they were swapped, left blank, or with splitted information. To resolve this problem we started by looking the first rows in a txt file, in order to understand the composition, and then try to find the best parameters to instantiate the reading csv Spark method. After that, due to the discrepancy between nature of data and the type schema, we decide to force the data type of each column.

2.1 Data cleaning

2.1.1 First round of the Data cleaning: attributes

We made more action to delete uncleaned data due to Missing values. To have a better idea about the content of each column and with the aim of take wise decisions, we perform a deep analysis of the distinct values of each column.

- The first step was the elimination of all columns with more than 120 000 missing values: {*telekomHybridUploadSpeed*, *noParkSpaces*, *heatingCosts*, *energyEfficiencyClass*, *lastRefurbish*, *electricityBasePrice*, *electricityKwhPrice*}
- elimination of the exactly duplicate variables: {*geobl_n = regio1*, *geokrs = regio2*}
- elimination of column with the same value. We put a treshold to the variance for this task: {*telekomTvOffer*, *newlyConst*}
- elimination of non informative features and pre-calculated binned columns (we decided that, if needed, we would re-created by ourselves): {*pricetrend*, *telekomUploadSpeed*, *scoutId*, *yearConstructedRange*, *noRoomsRange*, *date*, *houseNumber*, *livingSpaceRange*}.

In this step we only kept "*baseRentRange*" to impute the missing values since it seems relevant and doesn't have any missing values

The size of the dataframe after dropping the attributes is (268850, 29), while the following are the columns' names: {*regio1*, *serviceCharge*, *heatingType*, *balcony*, *picturecount*, *totalRent*, *yearConstructed*, *firingTypes*, *hasKitchen*, *cellar*, *baseRent*, *livingSpace*, *condition*, *interiorQual*, *petsAllowed*, *streetPlain*, *lift*, *baseRentRange*, *typeOfFlat*, *geo_plz*, *noRooms*, *thermalChar*, *floor*, *numberOfFloors*, *garden*, *regio2*, *regio3*, *description*, *facilities*}. As final action for this particular step we substitute NA and Nan with the NULL in each column to have the same representation of NULL values.

2.1.2 Second round of Data cleaning: rows

In this section we made some action to clear the rows of the dataframe. We went from this size (268850, 29) to (196440, 29).

- We put a threshold in each rows: if the row has more than 6 attribute out of 29 NULL, the row is deleted. (To do this action we created a new column with the count of Null values. If the count is bigger than 6 we delete the row)
- Due to the importance of the attribute "*yearConstructed*" we decided to drop every row without a value inside it.

In this step, we also performed a manual task of outlier detection by dropping all the row containing strange values in some columns. This analysis was done by listing the top N values of a column and looking if very big values were present. The columns involved in this part, and the range of values that we decide to keep, were:

| yearConstructed | noRooms | floor | numberOfFloors | serviceCharge | totalRent | baseRent | livingSpace |
|-----------------|---------|-------|----------------|---------------|-----------|----------|-------------|
| >=1700 <2023 | <=20 | <=30 | <=30 | <=1000 | <=15000 | <=20000 | <=1000 |

2.1.3 Imputation of variables

After all the cleaning there is the section where we impute the variable in the place of NULL values. This part more than every other one is subjective due to the decision of the value used for the replacement. We divided the decision by type of the data.

After a careful study of each attribute by the construction of a table (as an example Table 2.1 which shows the percentage of discrete variable values, we made some other tasks:

- We used:
 - The **mode** for the type string: {*heatingType*, *firingTypes*, *condition*, *typeOfFlat*}
 - The **median**, grouping by *baseRentRange* for the continuous variables: {*serviceCharge*, *baseRent*, *totalRent*, *thermalChar*, *livingSpace*}
 - The **median** for: {*noRooms*, *floor*, *numberOfFloors*}
- Because of the huge number of missing values in the attributes *petsAllowed* and *interiorQual*

Table 2.1: Top 10 distinct values, count, and percentage in column 'heatingType'

| HeatingType | Count | Percentage |
|----------------------|--------|-------------------|
| central_heating | 120848 | 61.46146960696558 |
| district_heating | 21335 | 10.85065912604768 |
| floor_heating | 15152 | 7.70607860688421 |
| gas_heating | 14388 | 7.31751973309464 |
| self_contained_ce... | 14244 | 7.24428350557409 |
| oil_heating | 3637 | 1.84972332980714 |
| heat_pump | 2397 | 1.21907803726910 |
| combined_heat_and... | 1812 | 0.92155586296688 |
| night_storage_heater | 1005 | 0.51112783790382 |
| wood_pellet_heating | 814 | 0.41398811945642 |

respectively with "*negotiable*" and "*normal*". The possible values became:

- petsAllowed: {negotiable, no, yes}
- interiorQual: {luxury, normal, sophisticated, simple}

We still have not treated the free text features, even in terms of missing values. And also the "streetPlain" (the address with German characters) has still missing values, since we can not impute the address of an apartment

2.2 Anomaly detection

Next part is Outlier detection, fundamental for this dataset. We based the analysis on the attributes of the following two datatypes *'integer'*, *'double'*. Then we created a new column where for each numeric column, a function calculates the lower and upper bounds for outliers using the first and third percentile. The function adds a new column indicating whether each value is an outlier (1) or not (0).

Then we dropped the outliers from the dataset, using this equivalence:

$$IQR = interquartile3 - interquartile1$$

$$outlier_if_the_value_is_less_than = interquartile1 - 1.5 * IQR$$

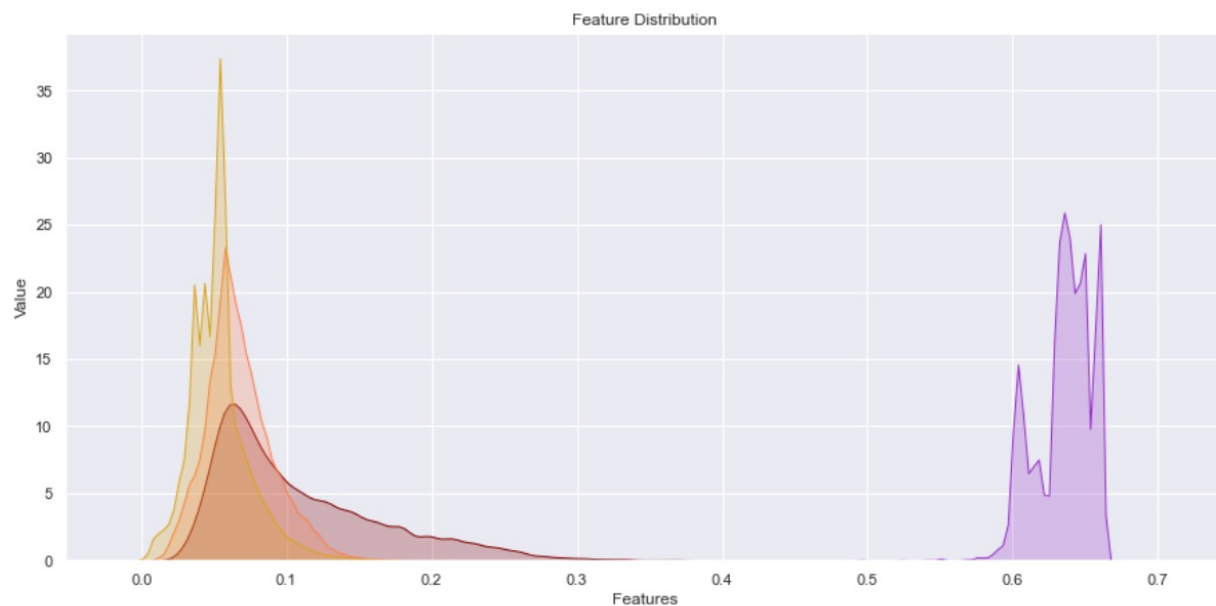
$$outlier_if_the_value_is_more_than = interquartile3 + 1.5 * IQR$$

In the end the size went to: (188417, 28)

2.3 Normalization

After we select the column to normalize, we made a couple of attempts. In the end we decided to use the **MinMax scaler** among all those available. Then we transformed the distribution with the log function because of the the **skewness** of data, paying attention to the negative values.

Now we are ready for last task before having the full cleaned dataset.



`log_baseRent = maroon`

`log_yearConstructed = darkorchid`

`log_livingSpace = coral`

`log_thermalChar = goldenrod`

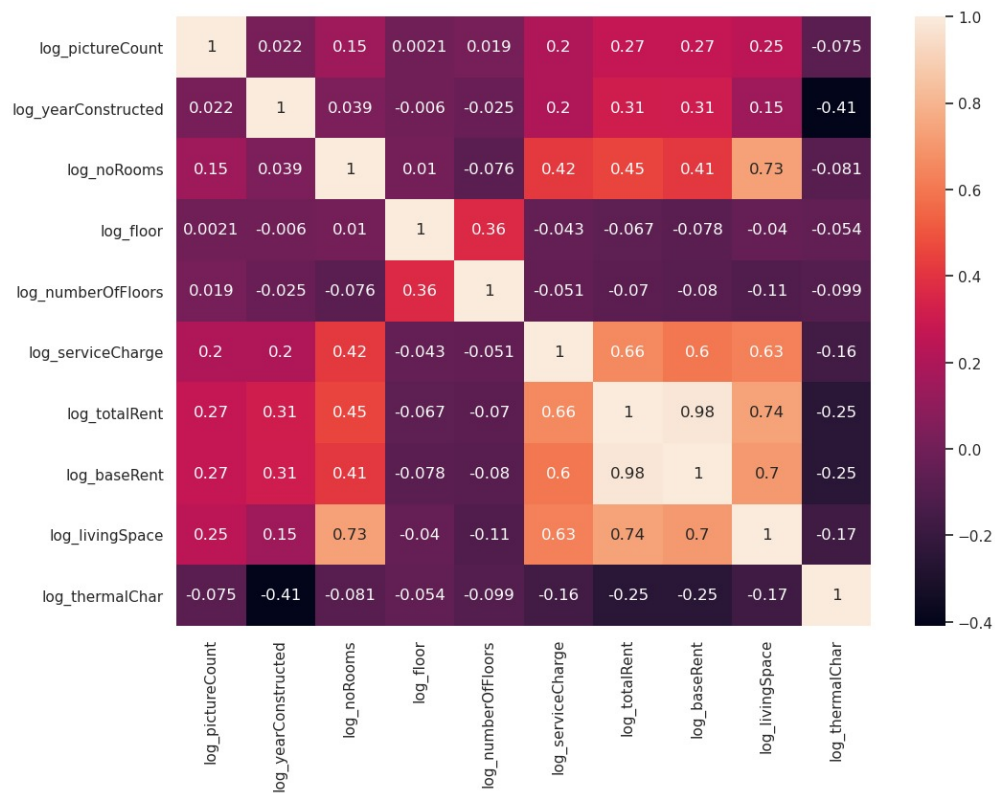
2.4 Third round of Data cleaning: Correlations

At the end we performed a correlation analysis between log transformed continous attributes.

As we can see in the image, there were two attributes very correlated: log_baseRent and log_totalRent with a ratio of 0.98.

We decide to drop log_totalRent, because originally it had some missing values, differently from log_baseRent.

At this moment, we had a total cleaned dataframe, where the data were both scaled, transformed and cleaned.



2.5 Exploration of variables

To visualize the attributes, we printed a couple of attributes per data type before the normalization.

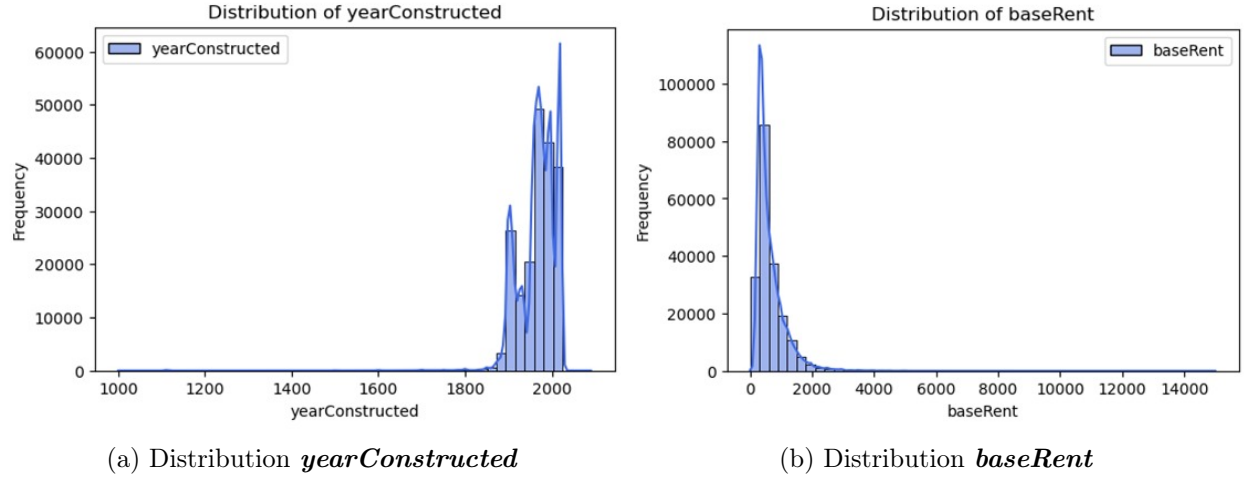


Figure 2.1: Plots of Continuous attributes

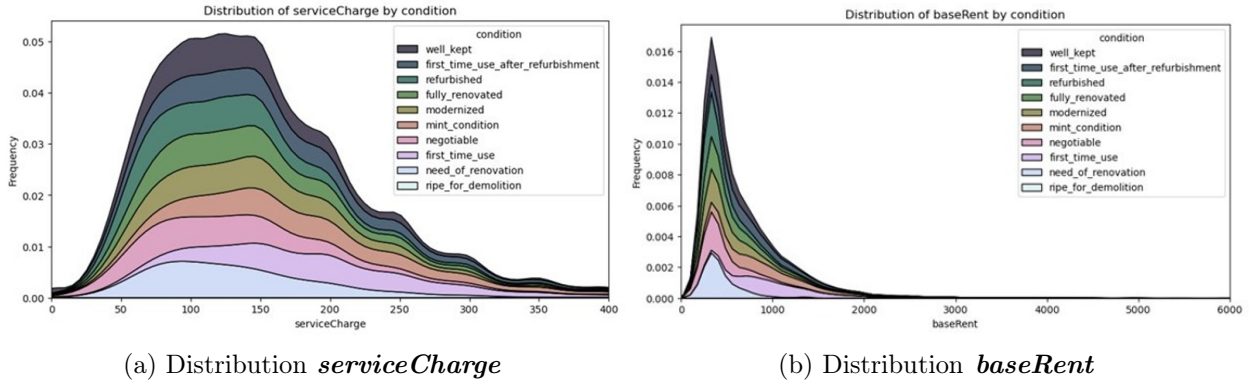


Figure 2.2: Plots of attributes

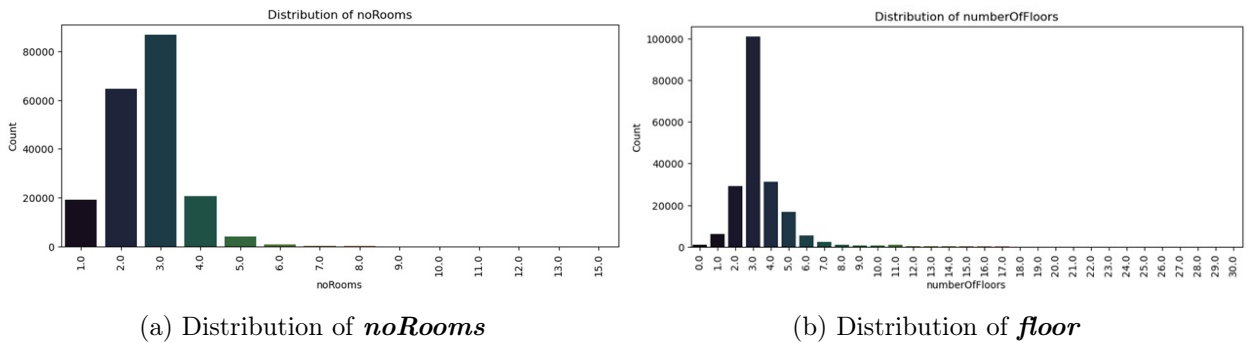


Figure 2.3: Plots of Discrete attributes

3. Machine Learning

3.1 Clustering

In the clustering part we aim to identify well defined clusters (distinct groups) in the dataset.

We did make use of K-Means as clustering algorithm. This algorithm aims to choose centroids that minimise the inertia, or within-cluster sum of squares criterion:

$$\sum_{i=0}^n \min_{\mu_j \in C} (\|x_i - \mu_j\|^2)$$

Three different analysis have been performed. First of all we performed a Silhouette analysis on a few selected variables (double type):

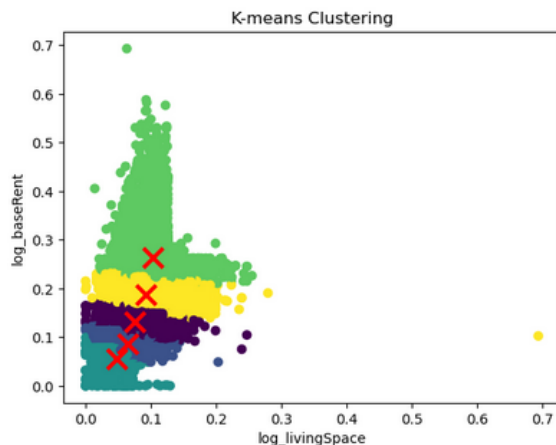
- "log_yearConstructed", "log_noRooms", "log_floor", "log_numberOfFloors",
"log_serviceCharge", "log_baseRent", "log_livingSpace", "log_thermalChar"

We found that the number of clusters that better leads to a reduction on inter cluster distance is 3.

Afterwards, we did perform a more specific analysis on 2 target variables:

- "log_baseRent", "log_livingSpace"

Here we performed a Silhouette analysis that suggested to make use of 5 clusters. The following result has been found:



(a) K-Mean Clustering (K=5)

Two factors have been noticed:

First of all, there is not clear separation among the different clusters. Another thing to notice regards the light green cluster. It can be noticed that, for some observations, the increase in baseRent is not accompanied by an increase in the living space, delivering a relatively surprising result. The cause might be related to the apartment location. For example, living in a 50 ms² apartment in

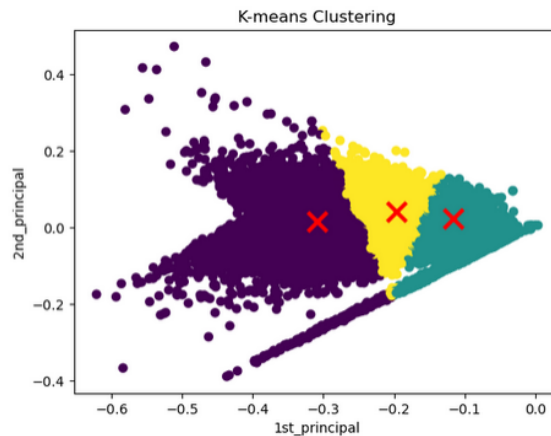
Prenzlauer Berg district might have a well higher cost than living in a similar size apartment in Neu-Koeln.

The latest performed analysis aims to better capture the clusters based on a few more features, hence apart from the two mentioned above, we added the following in the analysis:

- "log_serviceCharge", "log_thermalChar"

After having performed the Silhouette analysis we opted to use 3 clusters for the analysis.

We then performed a PCA test in order to reduce the dimensions to 2 and hence plot the analysis.



(a) K-Mean Clustering (K=3), after PCA

3.2 Regression

Then we moved to perform Regression. In this part, our aim was to try to predict the target variable *log_baseRent*. In order to do that, we had to apply some transformation before and follow a clear path: **transformations**, **feature importance** (for having a smaller dataframe to fit), define a **baseline** to compare results, choice of the **models** to try, **hyperparameter tuning**, **apply models** to test set, model **evaluation**. In this section we'll explore all these steps, with the objective of clearly explain how we reached our conclusions and results.

3.2.1 Transformation and Feature Importance

Starting from our cleaned dataframe (*dataframe_totalCleaned*), we began to handle text features, with the aim to include them during the feature selection's phase. We followed two different approach, based on the lengthy of the texts inside the column:

- For short text features, we used **StringIndexer** (*'regio2'*, *'geo_plz'*, *'regio3'*, *'regio1'*, *'heatingType'*, *'firingTypes'*, *'condition'*, *'interiorQual'*, *'petsAllowed'*, *'typeOfFlat'*) so that we had a numerical representation of them.
- For long text features (*'description'*, *'facilities'*, *'streetPlain'*), we used a combination of two methods: **Tokenizer**, which splits long sentences in a list of word, and **HashingTF + IDF**, which maps a sequence of words to their term frequency representations in a vector space and then computes the Inverse Document Frequency. By doing this, we obtained similar data both for short and long text features.

After that we put all together in a vector obtained with **VectorAssembler** and ran a Feature Importance based on a DecisionTreeRegressor with *log_baseRent* as target variable. By applying it, we remained with just 5 features, i.e. '*geo_plz*', '*regio3*', '*log_yearConstructed*', '*log_livingSpace*', '*log_thermalChar*', '*log_baseRent*'.

Once having this list of features, it came to our mind that there could be some kind of correlation between '*geo_plz*' and '*regio3*', that we ignored in the previous steps because they were not numerical. We applied a simple correlation between the StringIndexer representations of these two variables, and we discovered that they were correlated at 0.78, so we decided to maintain just '*geo_plz*' because of a better granularity. We have reduced a lot our dataframe, from 29 attributes to just 5 attributes. This big "cutting" was also due to the fact that with a lot of features we encountered many Memory Overflow errors and connection reset error, maybe correlated to the node's memory of the distributed environment.

3.2.2 Baseline and models' choice

Once we had chosen the features, we started to setting up the environment for models' application: First of all, we needed to have a "dummy result" in order to be able to compare the models. We decided to run a simple Linear Regression, so that we could take note of the results. With Linear Regression we obtained not very good evaluations (Table 3.1):

Table 3.1: Performance Metrics

| Metric | RMSE | MSE | R ² | MAE |
|--------|--------|--------|----------------|--------|
| Value | 0.0497 | 0.0025 | 0.3163 | 0.0382 |

By analyzing these results, we immediately noticed that, for a better comprehension of model's evaluation, we should have as reference's metric **R²** because we had our values log-transformed and scaled between 0 and 1. This setting could lead us to think that the error is low. Otherwise, R² indicates the variance in the dependent variable (the target) that is predictable from the independent variables.

As we figured out that a simple model as Linear Regression could not suit with our context, we decided to apply more complex model with the aim of reaching satisfactory outcomes. We switch our attention to : **Decision Tree Regressor**, **Gradient Boosting Regressor** and **Random Forest Regressor**

3.2.3 Hyperparameter tuning

First of all, we decided to perform a hyperparameter tuning in order to give proper parameters to our models. After having initially splitted our dataset in train and test set (80%/20%) we decided to take a sample of 30% of the train set and split it in train and validation set (80%/20%). After that we ran a GridSearch with Cross-validation (3-folds): the choice of this algorithm gave us some computational issues because it gave us Overflow memory error during the tuning of all the parameter. In order to avoid it, we perform a "parameter-by-parameter" tuning, even if we know that maybe this methodologies could not return us the best outcome. Here (Table 3.2), the parameters chosen:

As done for Linear Regression, we tuned the parameter by optimizing the R² metric, in order to

Table 3.2: Regressor Hyperparameters

| Regressor | Hyperparameter | Options |
|-------------------|---------------------|-----------------|
| Decision Tree | maxDepth | 5, 10, 15 |
| | minInstancesPerNode | 1, 5, 10 |
| | minInfoGain | 0.0, 0.1 |
| Gradient Boosting | maxDepth | 5, 10, 15 |
| | maxIter | 10, 20 |
| | minInstancesPerNode | 1, 5, 10 |
| | stepSize | 0.1, 0.05, 0.01 |
| | subsamplingRate | 0.8, 0.9 |
| Random Forest | numTrees | 20 |
| | minInstancesPerNode | 50, 100, 150 |

have better understanding. Once we had tune all the parameter, we obtain the options in 3.3 as best results:

Table 3.3: Regressor Best Hyperparameters

| Regressor | Hyperparameters |
|---------------|--|
| DT Regressor | maxDepth = 10, minInstancesPerNode = 10, minInfoGain = 0.0 |
| GBT Regressor | maxDepth = 10, maxIter = 30, minInstancesPerNode = 10, stepSize = 0.1, subsamplingRate = 0.9 |
| RF Regressor | maxDepth = 10, numTrees = 20, minInstancesPerNode = 50 |

Finally, we took note of this parameter and we switch to test our models.

3.2.4 Model's application and evaluation

Table 3.4: Regressor Evaluation Metrics (RMSE, MSE, R2, MAE)

| Regressor | RMSE | MSE | R2 | MAE |
|-------------------|--------|--------|--------|--------|
| Decision Tree | 0.0326 | 0.0011 | 0.7065 | 0.0212 |
| Gradient Boosting | 0.0283 | 0.0008 | 0.7792 | 0.0183 |
| Random Forest | 0.0321 | 0.0010 | 0.7160 | 0.0210 |

As we can see in Table 3.4, we obtained pretty good result in each of the models that we tried. The best model was Gradient Boosting Regressor, which obtained an R2 of 0.7792:

Table 3.5: Data Representation

| log_baseRent | prediction |
|---------------------|---------------------|
| 0.06765864847381486 | 0.10742668170619389 |
| 0.0466011032562937 | 0.04476442388755942 |
| 0.07417939817425147 | 0.06704596252044018 |
| 0.07510747248680548 | 0.0652128055743308 |
| 0.06391332574365285 | 0.05920736107852727 |

3.2.5 Alternative Path

We also tried to do a regression by following another path, with other transformations (involved more intensively text features) in order to understand if we could obtain better results.

In this approach we start from the beginning by another way of data preparation, we make the two text columns and use **TF-IDF** and word-embedding methods after **tokenizing** them also we have done decimal reduction of the values. So the steps that we have taken is as below: First, We identified certain columns as having ordered categorical values. For each of these columns, we created a new column to the original column name. This new column contains numeric indices corresponding to the original categorical values. Then We identified columns which assumed to have boolean values. We combined these boolean values into a new column by converting True to 1 and False to 0. Furthermore, We have selected columns for one-hot encoding. For each of these columns, we created an index and then applied one-hot encoding to generate new columns with binary values indicating the presence of each category. Then we Combined All Steps into a **Pipeline**, to streamline and automate the data transformation process as proceeded steps: 1. Fitting the Model: We applied the pipeline to the original data to learn and fit the transformations. 2. Transforming the Data: We applied the trained pipeline to the original data to produce a new DataFrame with all the transformations applied. 3. Column Selection for Removal: We identified a list of columns that are not needed for further analysis or modeling. These include various location-related, condition-related, and indexing columns. for the part as explained above we have done some processing on the text analysis which shown to be effective and change the results, we performed the following text processing steps:

1. **Lowercasing:** We created a new column in the DataFrame We converted the text to lowercase for uniformity. The purpose is likely to standardize the text, making it case-insensitive for future analysis or modeling tasks.

2. **Tokenization** Function Definition: We defined a tokenization function for German text. The function is named “german-tokenize” and takes a text input and returns a list of tokens (individual words) extracted from the text. The function handles cases where the input text is None by returning an empty list.

3. **Tokenization UDF** (User-Defined Function) **Creation:** We created a user-defined function (UDF) named “german-tokenize-udf” using the previously defined tokenization function. This UDF is designed to operate on a column of text data and apply the tokenization process.

4. **Tokenization Applied to DataFrame:** We added a new column to the dataframe, This column is generated by applying the UDF (german-tokenize-udf) to the existing column.

5. **Stopwords Removal:** We defined a list of German stopwords (common words like “der,” “die,” “das,” etc.). Stopwords are words that are often common and do not contribute much to the meaning of a text. We used a StopWordsRemover to remove these stopwords from the tokenized description column. The result is a new column where common words have been removed.

6. **TF-IDF** (Term Frequency-Inverse Document Frequency) **Transformation:** We transformed the preprocessed text data using TF-IDF to represent the importance of words. HashingTF is applied to convert the filtered tokens into raw features using a hashing trick. IDF (Inverse Document Frequency) is applied to scale down the impact of frequently occurring words.

7. **Word Embeddings with Word2Vec:** We created word embeddings using Word2Vec, representing words as vectors in a continuous space. A Word2Vec model is defined with a vector size of 100, a minimum count of 5 (ignores words with a count less than 5).

8. **Rounding Numeric Columns:** We specified a list of columns containing numeric values and rounded each specified column in the to 2 decimal places.

Then We specified a list of columns that will be used as features (feature-columns) including the rounded numeric columns, one-hot encoded columns, word embeddings, and the newly created boolean columns. The target column for prediction (log-baseRent) is also included. And We removed rows with missing values in specific columns (log-pictureCount, log-yearConstructed, log-noRooms, etc.) from the DataFrame.

In summary, we performed the following tasks: 1. Rounds specified numeric columns to a certain number of decimal places. 2. Splits and casts boolean values into separate columns. 3. Selects a subset of columns deemed relevant for further analysis or modeling. 4. Handles missing values in specific columns by removing rows with missing values in those columns.

After the above steps we have done the regression on the processed data and without hyperparameter tuning it has a better results than not analysing the text. As above we chose 3 error metrics for the regression analysis

RMSE and **MAE:** Both are low, which is generally good. It means the model's predictions are close to the actual values on average.

R-squared: A high R2 value (close to 1) indicates that the model explains a significant portion of the variability in the target variable. An R2 of 0.815 is a strong result and suggests that our model is capturing a substantial amount of the variance in the data.

Table 3.6: Regression Evaluation Metrics

| Metric | Value |
|-----------|--------|
| RMSE | 0.0197 |
| MAE | 0.0130 |
| R-squared | 0.8150 |

Also we have plotted the plots of residuals and the plot of the prediction v.s our data as below that shown in Figure 3.3 . The first plot shown the difference between the data and the predicted data and as shown they are near the line so they shown a good result In the second plot we have shown the residuals that are low and most of them are near the zero value so it again shown that the regression model has done good.

3.3 Classification

In parallel to the regression task explained in Section 3.2, we decided to implement also the discrete-target counterpart, classification. We chose to perform this type of machine learning task on two different target variables: *log_livingSpace*, with the goal of discovering how well selected features

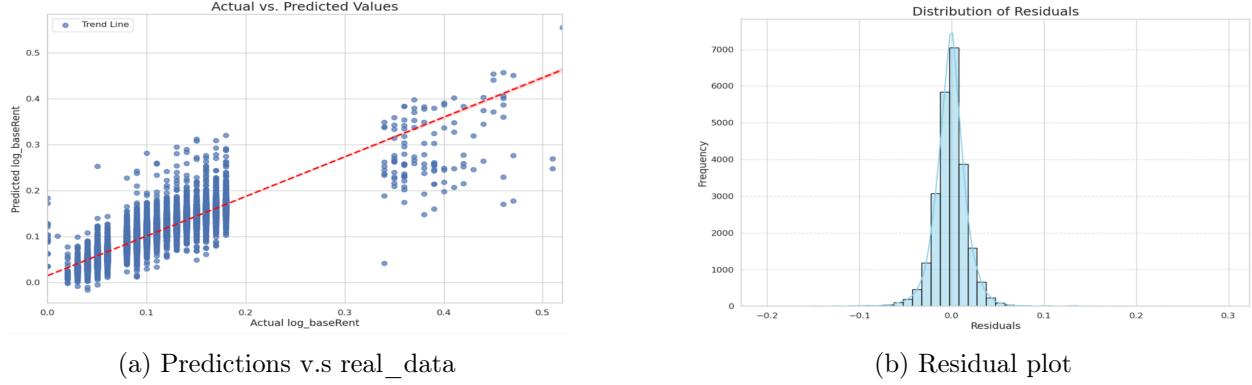


Figure 3.3: Residuals and prediction plots

would be able to predict the extension of a given apartment. In this case, since such variable was continuous, we decided to discretize it in three bins representing small, medium and large sizes; **condition**, this time with the goal of using selected features to make predictions about how good the structural condition of a given apartment was. In the following subsections, we will detail every part of the workflow we used to obtain the final classifiers and assess each result, therefore reaching our final conclusions. Since we have two separate target variables, for every upcoming task we will mention the differences between the two implementations (resulting in different datasets). The **workflow** can be summarized in the following steps: Classification-specific processing of the data, feature selection, hyperparameter tuning of the chosen classifiers and final training, testing and results assessment.

3.3.1 Data processing

In this section, we will explain every pre-processing step we used to make the data coherent and suitable for the subsequent tuning and implementation of the classifiers. First we loaded the dataframe, *dataframe_totalCleaned.csv*, resulting from the global data cleaning tasks described in Section 2.1. Once ensured this was done correctly, we proceeded to remove two attributes, *geo_plz* (ZIP code) and *streetPlain* (the address of the apartment), since the first was found to be highly correlated with the attribute *regio1* and the second was too specific of an information to keep. In the case of the target **condition** specifically, we checked the number of records representing each of its 10 distinct classes and we found three of them to be so under-represented (one of them had only one record, Table 3.7) that we decided to remove their corresponding records, obtaining 7 final classes as result of it. Therefore, this dataframe was shrunk to 185 568 records.

Then, we defined the number of classes for each target variable: in the case of **condition**, this was done by taking all of remaining 7 classes, while for *log_livingSpace*, since continuous, we set 3 as the number of bins in which to discretize it, as mentioned above. Next, we decided to **shuffle** the records in the dataset, since we wanted to avoid any ordering bias in the classifiers training phase, and proceeded to perform "**global**" **data transformations**, intended as transformations regarding the whole data before splitting it in training and test sets. These were equally performed on both the datasets for *log_livingSpace* and **condition**. The transformation was done to **encode** the independent features with three distinct approaches based on their data type and their number of distinct values:

- For the **categorical** features, we first applied a **StringIndexer** on them, mapping each class

Table 3.7: Number of records by *condition*'s class

| <i>condition</i> | count |
|---------------------------------|--------------|
| first_time_use | 17223 |
| first_time_use_after_renovation | 10729 |
| fully_renovated | 19268 |
| negotiable | 1818 |
| well_kept | 88986 |
| need_of_renovation | 1030 |
| ripe_for_demolition | 1 |
| refurbished | 19468 |
| mint_condition | 16538 |
| modernized | 13356 |

to a numerical value, and then we divided them in two groups to treat them differently: For features with more than 10 distinct values, we kept the **StringIndexer** values as they were; for the others, we decided to apply **OneHotEncoder** on them, creating one binary (dummy) feature for every distinct value of the already indexed ones.

- For the **boolean** features, we decided to map their values from {true, false} to integers {1, 0} respectively.

These transformations were done to make all the non-numeric original features numeric, and the reasoning behind the one-hot encoding of only the categorical features with less than a user defined threshold of distinct values was the fact that, since this process creates a new feature for every distinct value, we didn't want to create thousands of new ones, which would have resulted in a more time consuming and computationally expensive workflow.

Once these "global" transformations were done, we **split** the data into **train** and **test** sets using a **randomSplit** of 70%/30%, in order to perform the following ML tasks without the risk of incurring in the *Data Leakage* phenomena. Since several features resulting from the aforementioned process had not the same value-scale, we implemented a Min-Max scaling on only the indexed attributes, mapping their values in the (0, 1) range. To do so, we made use of the **VectorAssembler** followed by a **MinMaxScaler** both applied iteratively to each of these last. The scaler used to transform the training and test sets was fitted exclusively on the training set. Thereafter, we decided to not consider the free text attributes *description* and *facilities* for this task, given the computational-heavy process it would have entailed, which would have made more difficult the analyses. Nonetheless, we appreciated how it would have helped to retrieve some German words both identifying dimensional characteristics for the classification on *log_livingSpace* and structural-integrity notions for the tasks on *condition*.

Next, we proceeded for both the classification tasks and respective data to assemble the features using **VectorAssembler** and perform the processing of the target variable, to make it suitable for the subsequent steps of the workflow (it is important to notice that, also in this case, the fitting of these transformers were done exclusively on the training instances and then applied to the test set):

- for *log_livingSpace*, as said before, we performed a **natural binning** of the variable in 3 classes, exploiting the **QuantileDiscretizer** class. We did not perform a equal-width binning

since we still had extreme values which would have been grouped into singleton bins (or small frequency bins nonetheless). The resulting classes were discrete floats in the range (0, 2) and the discretized target was called *log_livingSpace_label*;

- for *condition*, instead, we applied a `StringIndexer` on it, mapping the classes to discrete floats in the range (0, 6) and the discretized target was called *condition_label*.

3.3.2 Feature selection

At this point in the workflow, we had a dataframe that could be used to perform a dimensionality reduction technique, such as the feature selection based on the **features' importance** retrievable from a non-tuned `Decision Tree Classifier`, since we wanted to reduce the generalization errors for the classifiers. For both the classification tasks, we first sampled the training data: for *log_livingSpace_label*, since the bins were already balanced, we sampled 40% of the data, while for *condition_label*, since the classes were imbalanced with one being much more frequent than the others (Table 3.7), we performed a stratified sampling (20% of the majority class, 100% of the minority classes, Table 3.8).

Table 3.8: Number of records in the stratified sample based on *condition_label*

| <i>condition_label</i> | 0.0 | 1.0 | 4.0 | 3.0 | 2.0 | 6.0 | 5.0 |
|------------------------|-------|-------|-------|-------|-------|------|------|
| count | 12491 | 13579 | 11721 | 12023 | 13565 | 7428 | 9266 |

Thereafter, we proceeded to `randomSplit` the sampled training data (70%/30%) in two sets, one for training and one for evaluating the features' importances. The evaluations in Table 3.9 also gave us results that could be used as benchmarks for the final and tuned classification models.

Table 3.9: Non-tuned `DecisionTreeClassifier` evaluation metrics

| Metric | <i>log_livingSpace_label</i> | <i>condition_label</i> |
|--------------------|------------------------------|------------------------|
| Accuracy | 0.756 | 0.484 |
| Weighted Precision | 0.761 | 0.488 |
| Weighted Recall | 0.75 | 0.486 |
| Weighted F-Measure | 0.759 | 0.477 |

Through the analysis of the features' importances, for each target, were kept the following ones: for *log_livingSpace_label*, we set the importance threshold to 0.002, resulting in 11 features {*balcony*, *hasKitchen*, *lift*, *log_pictureCount*, *log_yearConstructed*, *log_noRooms*, *log_floor*, *log_numberOfFloors*, *log_serviceCharge*, *log_baseRent*, and *log_thermalChar*}, while for *condition_label*, we set the importance threshold to 0.003, resulting in 13 features {*hasKitchen*, *cellar*, *lift*, *log_pictureCount*, *log_yearConstructed*, *log_floor*, *log_numberOfFloors*, *log_serviceCharge*, *log_baseRent*, *log_livingSpace*, *log_thermalChar*, *interiorQual*, and *regio1*}.

Once the most important features were selected, we restructured the dataframe in order to contain in the assembled *features* column only these last (separately by target variables) and this was done using `VectorAssemblers`, as described before. The target variables, instead, were once again taken discretized as in the previous steps. Up to this point, we had the non-sampled training set at our disposal, but only with the target and the remaining features. As soon as the dimensionality reduction was performed, we began with the classifiers' selection and hyperparameter tunings.

3.3.3 Hyperparameter tuning

Before performing the hyperparameter tuning, we did a `randomSplit` of the training set (70% train /30% validation) and we decided to **sample** with the same strategy as in Section 3.3.2, but with respect to the target *condition_label* with different fractions for the majority and minority classes, to ensure a less computationally demanding tuning: for the **majority** class 10% of the records, while for the **minority** classes 60% of the records were selected.

As for the models to use as classifiers, we decided to implement a **Decision Tree Classifier**, a **Random Forest Classifier** and a **Multilayer Perceptron Classifier** (Artificial Neural Network). For the tuning of these models' hyperparameters, we performed a **3-fold Cross Validation** using `CrossValidator` and as performance evaluation metric the **weightedFMeasure** (trying to bypass any imbalance) to retrieve the best models' parameters. This was done equally for both the classification tasks. Since the parameters for each model could not be passed simultaneously due to the lengthy durations of this procedure, we decided to circumvent such constraint tuning the hyperparameters **pair-by-pair**, starting from what we thought had precedence over the others, selecting 2/3 values for each at every iteration. Due to space limitations, in this part we will not place every parameter's tried values and combinations. The prioritized objectives in this phase were **avoiding overfitting** and **ensuring generalization** of the models, keeping in mind the **trade-off** between **complexity** and **performance**.

While the best parameters for each model and each target label aren't shown due to space constraints, the performances for the best configurations, based on the weighted F-Measure, are displayed in Table 3.10.

Table 3.10: **Weighted F-Measure** by model, for each target variable

| Model | <i>log_livingSpace_label</i> | <i>condition_label</i> |
|--------------------------------|------------------------------|------------------------|
| DecisionTreeClassifier | 0.747 | 0.412 |
| RandomForestClassifier | 0.73 | 0.35 |
| MultilayerPerceptronClassifier | 0.746 | 0.342 |

3.3.4 Testing and results assessment

Once we had found optimal hyperparameter configurations for each model, we trained these last using all the records in the aforementioned training sets and to make predictions on the respective test sets: for *log_livingSpace_label*, we had 131 946 records in the training set and 56 439 records in the test set, while for *condition_label*, we had 129 936 records in the training set and 55 595 records in the test set.

Table 3.11: **Performance** for DecisionTree, RandomForest and MultilayerPerceptron Classifiers

| Target | Model | Accuracy | Weighted Precision | Weighted F-Measure |
|------------------------------|--------------|----------|--------------------|--------------------|
| <i>log_livingSpace_label</i> | DecisionTree | 0.755 | 0.757 | 0.75 |
| | RandomForest | 0.729 | 0.726 | 0.73 |
| | MLP | 0.745 | 0.741 | 0.742 |
| <i>condition_label</i> | DecisionTree | 0.47 | 0.462 | 0.463 |
| | RandomForest | 0.472 | 0.472 | 0.459 |
| | MLP | 0.392 | 0.397 | 0.375 |

The **performances** for the **final testing** of the classifiers are displayed in Table 3.11. From such results, we can tell that the models have been, in our opinion, **reasonably good** at classifying the different classes of each target variable. It is also important, especially in the context of the *condition_label* classification task, to highlight the fact that the final models were trained on original-**imbalanced** data. This, probably, has led to worst results, but since oversampling the minority classes would have been too expensive and undersampling (as we did in the tasks before) would have led to information loss, we decided to follow this methodology.

Results: *log_livingSpace_label*

In order to better appreciate and understand the models, we can observe from Table 3.12 the top 3 most important features retrieved from both the **DecisionTree** and **RandomForest** classifiers. This type of information is consistent with our initial thought process. In fact it seems logical that, to predict the extensional aspects of an apartment, the most important information to know would be its number of rooms. Furthermore, usually the higher are the rent and auxiliary expenses (i.e. electricity), the bigger the apartment will be and vice-versa.

Table 3.12: **Feature importance** in **DecisionTree** and **RandomForest** models for *log_livingSpace_label*

| Feature | DecisionTree | RandomForest |
|--------------------------|--------------|--------------|
| <i>log_noRooms</i> | 0.4778 | 0.5159 |
| <i>log_baseRent</i> | 0.3889 | 0.1819 |
| <i>log_serviceCharge</i> | 0.1206 | 0.2639 |

Noting the **confusion matrices** in Figure 3.4, we can also see how the errors are made mostly when the real size of the apartment is small, since it gets frequently classified by the three models as intermediate. In contrast, the misclassifications between small and large apartments are minimal especially in the **DecisionTree**. For this classification task, the **worst performances** were obtained by the **RandomForest** classifier, while (in our opinion, strangely) the **best performances** were achieved by the "single" **DecisionTree** classifier, followed by the very-close performing **MultilayerPerceptron**.

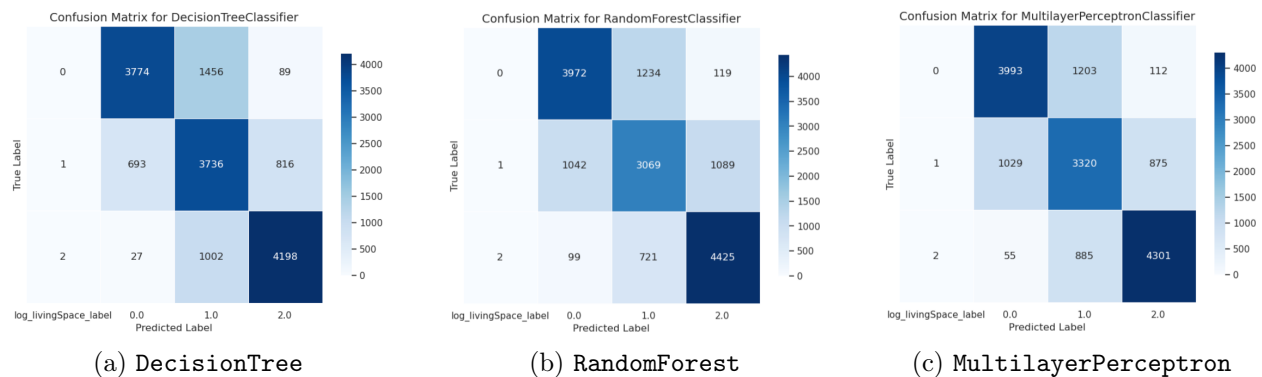


Figure 3.4: **Confusion matrices** of the classifiers for *log_livingSpace_label*

Results: *condition_label*

Even in this case, looking at Table 3.13, we can say that these **features' importances** seem to be reasonable and consistent with our first impressions. The **year of construction** of a house

or apartment should be a very good indicator of how well the structural integrity is at the time of the web-scraping from which the data was extracted. In our opinion, this could be also a key attribute, given its importance, influencing the misclassification error achieved by the models, since the apartments could be renewed and refurbished but also dated. Therefore, the models could have given "too much" importance to the *yearConstructed* feature, leading to errors.

Table 3.13: **Feature importance** in *DecisionTree* and *RandomForest* models for *condition_label*

| Feature | DecisionTree | RandomForest |
|----------------------------|--------------|--------------|
| <i>log_yearConstructed</i> | 0.7396 | 0.5919 |
| <i>regio1</i> | 0.0823 | - |
| <i>interiorQual</i> | 0.0462 | 0.0890 |
| <i>log_thermalChar</i> | - | 0.0865 |

The other top features, *regio1* and *interiorQual*, might tell us that there may be newly-built housing areas of Germany, therefore improving the classification of the apartments in terms of their condition. The interior quality of the property may be connected to the condition of the whole building itself.

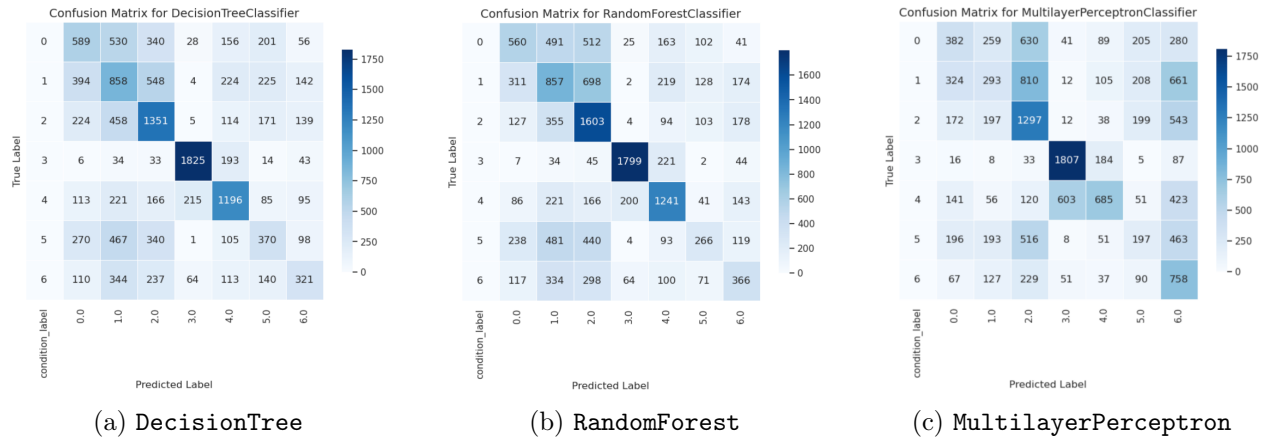


Figure 3.5: **Confusion matrices** of the classifiers for *condition_label*

From the **confusion matrices** in Figure 3.5, we can notice how the models do are good at classifying the classes {1, 2, 3 and 4}, while for the others there are large amounts of misclassifications. As said before, this problem could also be enhanced by the class imbalance in the data. Finally, for this classification task we have that the **worst performances** were achieved by the *MultilayerPerceptron* classifier, which could be due to the fact that each layer is forced to have a *sigmoid* activation function and *softmax* output activation function. Instead, the **best performances** were obtained by the *RandomForest* classifier.