1.5em 0pt

Using Inside AirBnb Data for Price Prediction with Deep Learning Methods

Tobias Rinnert, Juan Sebastián Aristizábal Ortiz
Chair of Statistics, University of Göttingen

March 5, 2022

Abstract

Price prediction is one of the classical fields for the application and evaluation of modern innovative methods of statistical learning. Even if now better established, Deep Neural Networks (DNN) are still very promising and empirical research on their usage for regression tasks is relevant. As a part of the statistics department's empirical seminar "Statistical and Deep Learning" of the University of Göttingen, we use Inside Airbnb's public available data of the city of Berlin to use DNN and further statistical models for prediction of the price of individual listings. Firstly, we shortly describe the available data and then the process of cleaning and preparing the data for modeling. Thereafter, we present how we analyzed images of the flats to generate further variables. This includes the detection of multiple objects on the the images using convolutional neural networks (CNN) as well as the analysis of the color temperature (CT) and perceived brightness (PB) of the images. Consequently, we describe our DNN and further modeling techniques used for price prediction and the analysis of variable importance. Finally, we present, interpret and discuss the attained results using classical performance measures. Further architectural considerations and insights for DNN are presented on the appendix.

Keywords: Deep Neural Networks (DNN), Price Prediction, Inside Airbnb

1 Introduction

1.1 Problem

The central problem we attempt to address is to effectively predict the price of single Airbnb listings using public available Airbnb data stemming both from Airbnb itself and Inside Airbnb data bank. DNN are the method type of interest to be analyzed and used. Besides delivering good predictive performance further project goals to be attained are interpretability i.e. identifying which variables contribute the most to predictive performance and generalization i.e. being able to extrapolate model results to other cities without loss of performance and interpretational power.

1.2 Data Cleaning

The data in questions comes from the Inside Airbnb, an activist project that attempts to record and provide data relevant to the analysis, evaluation and the overall political discourse regarding Airbnb's impact on both the housing market and the urban community as a whole where it operates on a short term scope (1). The city of Berlin was assigned to us for analysis. Berlin's raw data came into 3 main clusters.

- *Listings* consists of 74 variables of different level of measurement and type describing the properties' type, size and amenities, booking metrics e.g nights, dates and day count of availability and actual booked days, metric reviews of single listings, information about the host e.g. name, owned listings, text descriptions as well as the listings' price.
- *Neighborhoods* consists of 2 variables with broad and specific legal neighborhoods names.
- *Reviews* contains 6 variables regarding the name, the time and the actual conceived text based user review given to a single listing after booking.

We saw a high degree of overlapping between the *listings* data set and the other two available sets. Moreover, *listings* contained the highest amount of relevant variables for our analysis. Subsequently, we focused mainly on the former and argue about why we did or did not consider using the variables contained in the latter two sets for each specific case.

Aiming to clean the data with the lowest possible information loss, we firstly followed the approach of identifying and removing variables that were repeated, absent or irrelevant. The only repeated variable we found in the *listings* set was "host listings count" which equated to the variable "host total listings count". The variable "bathrooms", "license", "calendar updated" were completely empty. Considered as irrelevant were "scrape id" that contained the identifier assigned to each listing by Inside Airbnb after picture scraping. We decided against variables "longitude" and "latitude" firstly because the exploratory data analysis done by the seminar's lead and by us did not delivered insight for believing that the listing's location decisively affects the price, considering Berlin's polycentric nature. A short analysis regarding this spatial data can be found in the appendix Most importantly, even if we could have tested this empirically, we judged the inclusion of these variables to be detrimental towards model generalizability. Fitting a model including such variables would have overfitted the model towards Berlin's specific geo-economic characteristics limiting its application to cities with different characteristics. The text-based available "neighborhood" variable also further provided for a geographical based price differentiation, if needed.

The second data cleaning step aimed to correctly identify and re-code missing and non-available values in each variable. Raw data came with different codings for NAs and NANs. R-Package *Naniar* (2) proved useful for this task. Subsequently, we calculated the missing rate for each variable and discarded those with a rate higher than 50%. Those variables were i.e. "neighborhood", "neighborhood overview", "host neighborhood", "host response rate","host acceptance rate" and "host response time" respectively. After discarding these, the next highest NA rate variable was 20.61% which we considered as acceptable. A cleansed complete (lacking of NAs) "neighborhood" variable was also available in the *listings* set and showed a high overlap with those variables included in the *neighborhoods* sub set. Thus, the removal of this variables did not seem much of a loss for us.

Trustfulness variables i.e. variables that indicate how much a tenant can be trusted or relied upon, came in both metric and written form. After booking and usage, Airbnb demands both the service and the host to be rated on several dimensions. This is taking place metrically on a scale from 1 to 5 (1 being the worst and 5 being the best) and in written form with a minimum cap of words to be written (3). We thought it is therefore sensible to assume a high degree of correlation between these variables set i.e. the metric and written reviews. Given the assumed correlation and time constraints, we decided against analyzing the reviews data set and instead focused on analyzing images of the flats, as they were not analyzed in any variable in the original data set. Initial steps taken to analyze the reviews can be found in the appendix in section A.12. Finally, we re-coded several metric variables that were in string form and extracted date based variables from the listings set. These required separate handling to re-code, which was aimed and attempted but fell outside the scope of the project.

1.3 Image analysis

To add further variables to our analysis we decided to analyze the flat images posted by the hosts on the Airbnb website. Before analyzing the pictures, they first had to be downloaded from the Airbnb website. For this we used the package selenium.

To analyze the pictures of the Airbnb flats, we used three methods. First, we detected which objects can be seen inside the pictures. Second, we measured the colour temperature of the pictures and third we measured the brightness.
All these analyses try to explore the same hypothesis: A host that lists a high price for a flat, would also try to make the flat as comfortable as possible, and show the quality of the flat in the pictures. Thus, certain objects, such as a fireplace, should be found more often in pictures of a high priced flat. Such flats should also be photographed with higher care by the host, resulting in brighter and warmer pictures.

1.3.1 Multi object detection with RetinaNet

To identify objects in the pictures we used RetinaNet with a ResNet50 as its backbone. The architecture of this Network will be explained in the following section.

The basic architecture of the RetinaNet can be seen in Figure 1

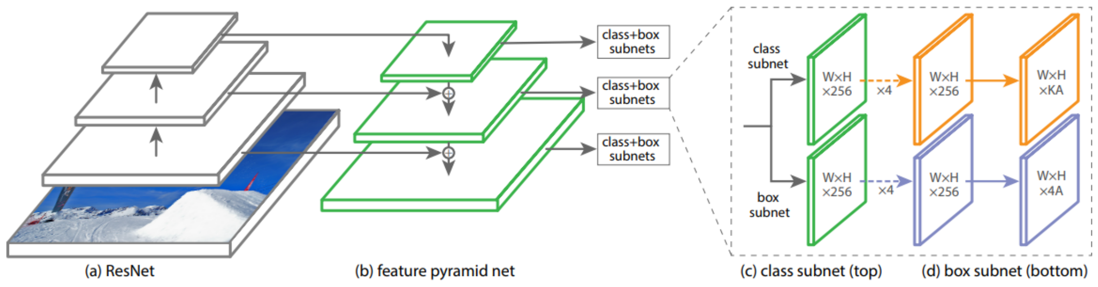

Figure 1: Architecture of RetinaNet

The first step of the RetinaNet is to extract feature maps by using a ResNet. In our case a ResNet50 is used, which means that the ResNet consists of 50 layers. Generally speaking, deeper layers of the Resnet50 represent feature maps of lower resolution. The architecture of the ResNet50 can be seen in Figure 2. In order to stay inside the scope of this paper, ResNet50 will not be described in more detail here.

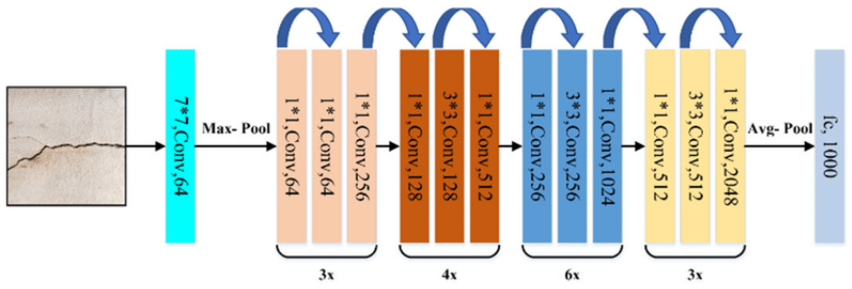

Figure 2: Architecture of ResNet50 (4)

RetinaNet does not, however, use the final output of the ResNet50. Instead, it adopts the Feature Pyramid Network (FPN) described by Lin, Dollar, et al. (2017). For each level of resolution represented in the feature maps inside the ResNet50, the FPN extracts the last feature map for each level. This can be seen in part a of Figure 1. (5)
The final feature map is then upscaled to the scale of the second last feature map, using the nearest neighbor upscaling, and merged with the second last feature map. This is done iteratively for each extracted feature map. In total, there are five levels of resolution from which feature maps are extracted in the RetinaNet, forming five levels of the "pyramid".

Three of these can be seen in part b of Figure 1. (5)

Because of this architecture, the FPN represents an arbitrary sized image with several proportional sized feature maps. In addition, each resulting feature map can give sufficient information for detecting large and small objects. This is the results of the merging of the extracted feature maps, described above. Feature maps with a bigger scale are formed earlier in the ResNet. Therefore their grid cells represent smaller objects better. Accordingly, grids of feature maps formed later in the ResNet represent larger objects better, as their grids represent a larger region of the initial image. By upscaling and merging the different extracted levels of feature maps, the feature maps used for predication in the RetinaNet can give sufficient information for identifying both large and small objects. (5)

This is very important when we look at the flat pictures listed on Airbnb. While objects are often sufficiently lit etc., the pictures show the room and not the objects. Thus, the size of the objects varies drastically. The architecture of the RetinaNet can cope with such data.
To form predictions, the RetinaNet first forms anchor boxes which were introduced by Ren et al. (2017). For a given feature map with n grid cells, n anchor boxes of different sizes and ratios are created. The sizes and ratios are chosen in such a way that most of the area of the feature map is covered by each box. For each anchor box, the existence, class, size and shape of an object is predicted.

This is done by two fully convolutional networks for each of the five resulting feature maps. Both networks run simultaneously. Therefore, the RetinaNet is a one-step detection model and not a two-step detection model such as the R-CNN. The "box" sub network predicts the size and shape of an object in terms of the center, the height and width of the according anchor box, while the "class" sub network predicts the existence and class of the object. An anchor box is now classified as a match in training if its Intersection-over-Union (IOU) with the box of the labeled images is above 0.5. If the IoU is between 0.4 and 0.5, the labels for the box sub network and the class sub network are ignored in the loss function. (6)

The loss function of the RetinaNet consists of the weighted sum of the loss functions for the two sub networks.

$$L = \lambda L_{loc} + L_{cls}$$

For the location and the shape of the anchor box, the loss function can be described as follows:

$$L_{loc} = \sum_{j \in \{x,y,w,h\}} smooth_{L1}(P_j^i - T_j^i) \quad \text{with} \quad smooth_{L1}(x) = \begin{cases} 0.5x^2 & |x| < 1 \\ |x| - 0.5 & |x| \geq 1 \end{cases}$$

$$T_x^i = (G_x^i - A_x^i)/A_w^i \quad T_y^i = (G_y^i - A_y^i)/A_h^i \quad T_w^i = log(G_w^i/A_w^i) \quad T_h^i = log(G_h^i/A_h^i)$$

Where x and y denote the coordinates of the center and h and w the height and width of anchor, A. P denotes the prediction probability.

For the class and existence of the object, the loss function can be described as follows:

$$L_{cls} = -\sum_{i=1}^{K} (y_i log(p_i)(1-p_i)^\gamma \alpha_i + (1-y_i)log(1-p_i)p_i^\gamma(1-\alpha_i))$$

For K classes, yi being a dummy denoting whether the prediction of the class for right, pi the prediction probability. With gamma = 0 and alpha = 1, the function would be equal to categorical cross entropy. Alpha however can vary to account for class imbalance, and gamma can vary to punish easily classified objects, which makes the network focus more on objects which are harder to detect. (6)
Predictions are then conducted by only selecting anchor boxes from each FPN level which have a confident score for its respected class of above 0.05. From the five FPN levels, the level with the highest confident score is selected. Redundant classifications, for each image and class, any anchor boxes of the same class, which overlap with the anchor box with the highest confident level with an IoU of 0.5 are deleted. Finally, the regression sub network is used to improve the shape and size of the final boxes. (6)

In order to utilize the RetinaNet, we used Monk AI and the open-source pipeline by Tessellate Imaging. The repository included training data of indoor images taken from the Open Image data set with which the RetinaNet was trained on. [1] The RetinaNet was trained to detect the objects that can be found inside a flat, such as beds drawers or tables. The full list is attached in the appendix.

While the RetinaNet finds solutions for many problems, especially one problem remained. For some pictures one object was still detected several times. Such an image can be seen in Figure 3. One can see that the bed as well as the curtains are detected multiple times. Here the mechanism of RetinaNet that should delete redundant detections described above, does not work. The reason for this can also be seen in Figure 3. For both classes, one anchor box is relatively large. The smaller anchor boxes do not overlap with the large anchor box enough so that the anchor box with the lower confident level is not deleted. Hence, we had to find a way to deal with multiple detections per object.
Rather than increasing the threshold at which the RetinaNet saves the respected anchor box, which results in fewer detections all together, we decided to only save one object per class per picture. This means that for the picture above we would only detect one curtain even though there are clearly two. However we were able to keep the threshold relatively low and thus detect a greater variety of objects. We felt that for our use case this was a

[1] The repository can be found here: `https://github.com/Tessellate-Imaging/Monk_Object_Detection.git` Further information on why we used this repository can be found in the appendix in A.6

reasonable compromise.

The resulting data was collected in a large data frame of dummy variables. The maximal number of pictures a host listed was 18. Thus, the data frame hold 18 × 24 = 432 columns indicating for each picture and object, if an objects was detected in the picture. For hosts that listed less than 18 pictures, the respected columns were filled with zeros and thus treated as if no object were detected. Further columns were created to track the number of pictures per host and the sum of detections per class per host.

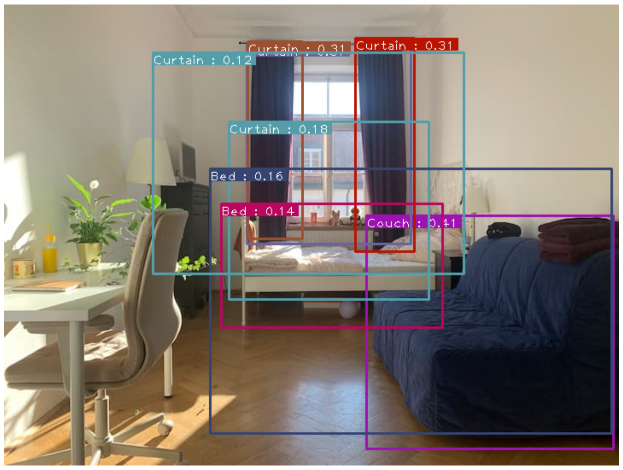

Figure 3: Exampe Output RetinaNet

1.3.2 Color temperature

In our second analysis we looked at the color temperature for each picture. We did so by using the measurement for correlated color temperature (CCT) described by Hernandez, Lee and Romero (1999). Since this paper deals with the deep learning part of the analysis we will not go into further detail here. The color temperature is a measurement for how yellow or blue light appears. It is measured in kelvin units per RGB value and ranges usually between 2200 and 6500 kelvin. (7)

Instead of determining the CCT for each pixel and taking the mean over these values for each picture, we instead determined the CCT of only three colors per picture. These colors were chosen by applying a k-means cluster analysis available in the cv2 package. K-means clustering is an unsupervised machine learning algorithm that determines clusters by iteratively calculating the Euclidean distance between the data points and centroids and minimizing this distance for each centroid. Thus, the three colors should represent the “dominant” colors in the respected image. (8)
We then took the mean over these three values to determine our approximation of the CCT of the image. For missing images, we filled the missing values with the CCT-mean over all pictures per host to avoid values of zero and a bias in the price prediction.

1.3.3 Perceived Brightness

To determine the brightness of each picture we used the weighted distance in the 3d-color space of RGB. The input for the formula given below was the mean of each R, G and B value of each picture.

$$PB = \sqrt{0.241R^2 + 0.691G^2 + 0.068B^2}$$

This method was suggested by Herbst (2006) as an alternative to HSV and HSL, which are both colour spaces used to determine brightness. The weights try to replicate how humans perceive brightness depending how the RGB values vary. For example, because the color yellow is perceived as bright in almost all its variations in the RGB color space, it receives a high weight. (9)

While this method is not tested scientifically, it was the best approach we found and gave reasonable results as can be seen in later sections. In particular we found that when considering the mean brightness per host, the unweighted method to determine the brightness per picture, results in a decrease of variable importance of 1.5 percent. The variable importance was determined with a Lasso model described in a later section. For missing images, we again filled the missing values with the brightness-mean over all pictures per host.

1.4 Further Considerations and Final Data Set

The initial available data for Berlin consisted of 17290 observations. After dropping all NAs, 12175 observations remained. This corresponded to an overall information loss of 29.58%. Even if it seemed high, our sample size remained big enough to partition for modeling and validation purposes. At the end it had a total of 64 variables.

Another important aspect of data preprocessing was normalization, either by means of standard score or feature scaling. Caret, the implementation for modeling and hyper-parameter tuning we eventually orbited towards, offered a diverse range of data preprocessing options including different normalization types (10). Because of this, normalization was addressed while modeling. Standard Scoring was the chosen normalization procedure for our whole work presented here.

The news of Munich data being used as a test set also demanded for cleaning and analysis of the data by the same criteria as described for Berlin. The situation was quite similar with minor exceptions on the *listings* data set for Munich, which contained a variable not present in the Berlin set. The reduced size of the Munich data relative to the Berlin data set, allowed for a quicker image scraping and data analysis process. Specifically, the Munich data set consisted of 4995 observations in its raw state. After cleaning, its size was reduced to 3222 observations, which corresponded to a lost rate of information of 0.3549%. In this case, we also had a total of 64 variables.

2 Methods

2.1 Data splitting

We cared about constructing a homogeneous framework not just for training and finding the best hyper-parameters but also for results comparison. Initially, we thought we would be presenting our test results on our Berlin data. Thus, we split this set into three parts consisting of 80% for training, 10% for validating and 10% for the final model test. The latter subset was carefully isolated in our GitHub Repo to be used just once.

Eventually, the news of using the Munich data set as a test set changed the way we partitioned the data. We proceeded then to split the Berlin set into 90% for training and 10% for validation leaving the Munich set fully for test purposes.

2.2 Model Choice

The choice of modeling methods came into question after having achieved a fully working cleaned data set. The way we answered this was dynamic. Having worked thoroughly with neural networks for image recognition and analysis, we thought of using Lasso for delivering interpretability by means of its implicit regularization and subsequent variable selection while considering computational constraints and time left available. Eventually we considered to also model the price with a deep neural net in line with the seminar goals . We further expanded our scope by including other "competitive" models for the task in hand such as Generalized Boosted Trees and Random Forests. These chosen methods are described as follows:

2.2.1 Lasso

As already mentioned, we thought of Lasso as a natural way to deliver interpretability. By means of the $\ell 1$ penalty, the lasso shrinks some of the coefficient estimates to be exactly equal to zero when the tuning parameter λ is sufficiently large (11). This offers implicit variable selection as the trained model delivers non-zero coefficients for those variables that considerably contribute toward predictive performance aiding towards the easier identification of relationship between variables. If the λ parameter is set to zero, the resulting model equates the classical ordinal least squares (OLS) model, which we thought of a natural reference model for comparison in a statistical context (12). We also considered to approach the normalization of the data empirically i.e. training models with and without normalized data to look for difference in performance. The same approach was also taken when considering the logarithmization of the price variable for some models. These preprocessing steps regard logical and sensible distributional considerations but their absence or presence proved interesting for us to investigate. Thus, for the lasso we trained 4 models in total:

1. OLS
2. No data preprocessing
3. Normalized i.e location and scale
4. Normalized i.e location and scale with log (price)

The approach of hyper-parameter tuning followed the considerations taken in both (11) and (13). The only hyper-parameter to tune is the shrinkage parameter λ. A tuning grid of values from from 10^{10} to $0,01$ was chosen, as this considers all possible cases of excluded variables (11).
For OLS, λ was set to 0. In this case, we used 10-Fold cross validation for hyper-parameter tuning using the Caret implementation's option *repeatedcv* (10). We validated the results

of Berlin in the previously described validation sub set and tested on the Munich set. Computationally, training times for Lasso on Berlin data were relatively fast with a mean training time of 3 minutes.

2.2.2 Deep Neural Network

Before describing the final model used to predict the price of the test set for Berlin, we will shortly describe how and why we decided to drop the data for the individual images for the final predictions.
Our first approach was to only include the data for the individual pictures and the cleaned data described in section one, precluding the number of detected objects per class and per host due to the obvious linear dependencies between the two data formats. However, no form of deep neural net could be trained with this data. This is most likely due to vanishing gradient descent. Because relatively few hosts listed more than 10 pictures, the final columns describing the detections in the pictures 10-18 hold very little information. After deleting the information for the last 8 images however, training was still not possible. Further descriptions of our attempts to train a neural net with this data can be found in the appendix.

In our final model we only considered the following data from the image analysis per host: The number of objects detected per class, the number of pictures per host, the mean CCT and the mean of the perceived brightness. Together with the data described in section one, we receive 63 variables used for price prediction. We standardized each variable using the z-transformation. We started by using adaptive resampling to determine the hyper parameters of our model. We used a multi layer perceptron model with dropout available in the caret package as an approximation for our deep learning model. Unfortunately, we could not use ”mxnet” or the neural network provided by caret as both resulted in errors.

We used adaptive resampling to find the best dropout rate, activation function and batch size. Adaptive resampling first processes a minimum number of hyperparameter combinations and then forms a probability distribution over all combinations. It then resamples the grid of hyperparameter combinations according to this probability and tests the combinations with higher probability first. This allows for reduced computational costs compared to simple “grid search” which tries out each predefined combination of hyperparameters. (14)

To use this method, we used the “adaptive cv” method, available on caret. The following hyper parameters were suggested: A dropout of 0.17 and the hyperbolic tangent activation function (tanh) as the activation function.
Dropout is used to simulate different architectures of the model while training, by “dropping” a given percentage or rate of nodes in the respected layer. This helps prevent overfitting, as it prevents the dropped units to adapt too much to the data. (15) The hyperbolic tangent activation function (tanh) and as comparison the sigmoid function can be seen in figure 4. It reaches from -1 to 1, going through 0. This is beneficial as negative or neutral

influences on the predicted price are mapped as such.

To control the learning rate of the training process we use the root mean square optimizer (RMSprop), an adaptive learning rate method that was proposed by Geoff Hinton. The RMSprop is an improved version of Rprop. The basic Rprop algorithm has two main functions: First, it only considers the sign of the gradient and not the actual value to deal with small gradients. Secondly, if the last two gradients are of the same sign and thus a theoretical minimum was not yet reached, the step size of the respected weight is increased. Thus, it builds up momentum towards the loss-minima. (16) However, since it does not take the actual value of the gradient into account, Rprop does not work for mini batches. In contrast to Rprop, RMSprop can be used for mini batches, which are useful when dealing with large redundant data like we deal with in this paper. RMSprop tracks the moving mean average of the squared gradient for each weight and adjusts the previous learning rate by dividing it by the square root of this moving average. If the steps suggested by RMSprop are getting too large, RMSprop also decreases the step sizes, which helps to avoid overshooting the minima. (17)

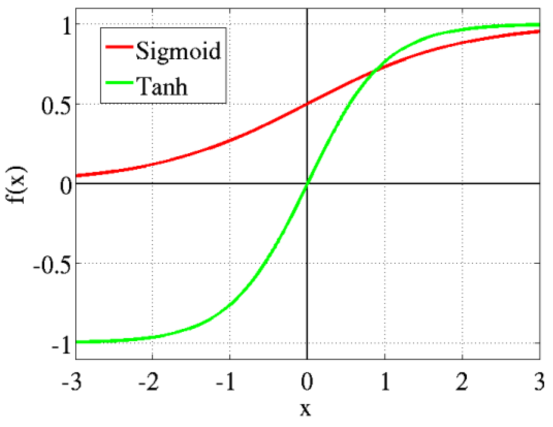

Figure 4: The tnah activiation function and the sigmoid function as an comparison

As our loss function we chose the widely used mean squared error.

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y_i})^2$$

The function ensures that the loss will not become negative due to the squared errors. This also leads to a large punishment for large errors, leading to a model that is less likely to predict extreme values. This makes the model more robust against outliers. (18)
After a lot of experimentation, which is described in more detail in the appendix, we settled for a very simple architecture, that none the less performed very well relative to our comparison models.
Varying the batch size for training did not influence the results. As a low mini-batch size is suggested generally in Bengio 2012, we followed this recommendation. Mini-batch sizes have the advantage of stabilizing the training process. (19), (20)

The validation set was set to be 40 percent of the training data, which accounts for 36 percent of the full data set. The results of the validation process can be seen in Figure 4:

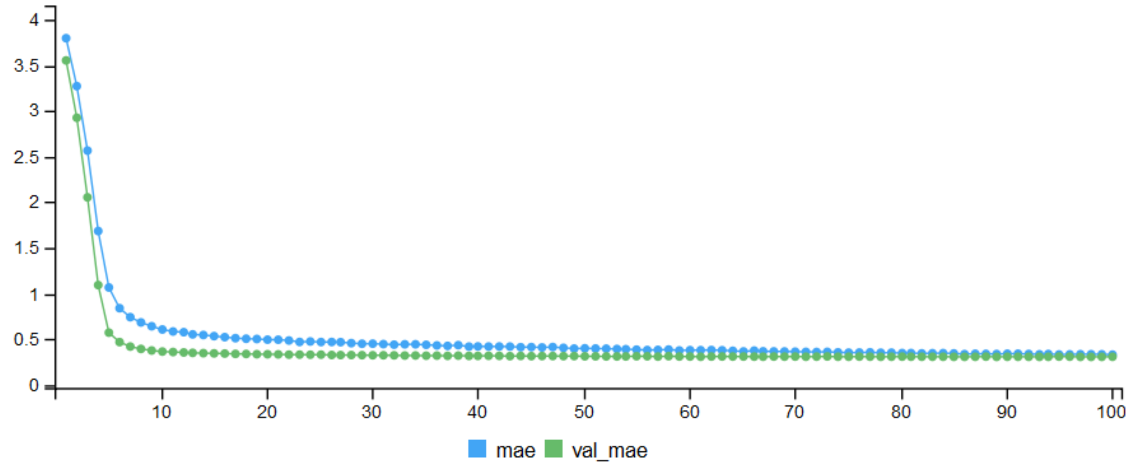

Figure 5: Training process of the DNN

One can see that both training set and validation set converge nicely. There is no improvement for the validation set after an epoch of 70. Therefore, the final model was trained with the full training set consisting of 90 percent of the data, for 70 epochs.
The final deep neural network holds 12 nodes per layer with two layers in total. Each layer is activated by the tanh function and is trained with a dropout rate of 0.17. This leads to a model with 937 trainable parameters.

2.2.3 Random Forests

The consideration of training a Random Forest (RF) arose as a response to Lasso's linearity. Classification and Regression Trees (CART) based methods partition a given predictor space into a set of non-overlapping rectangle-shaped regions (21). Methods that aggregate CART's, such as RF seemed to us as a good counterargument to address the problem at hand. RF grows single CART's on each calculated bootstrap set of samples. For each tree, it randomly considers a subset of variables of size m for each split the tree performs. Then all grown trees are aggregated by means of averaging in the regression case. The hyper-parameter controlling the value of m is called *mtry* and is the central tuning parameter of this method as stated by (12). On its conception mtry $= \frac{p}{3}$ and node.size $= 5$ were proposed as ideal values for the tuning parameters of RF (22). These rules of thumb were subsequently included as default parameters for most RF implementations such as *Ranger*, which we used together with *caret* (23). As already mentioned, further tuning parameters for RF, offered in the Caret package are *node.size*, which indicates the amount of observations left on each node i.e. in each partition of the predictor space. Smaller node sizes lead to overfitting on the training data set. [2]

[2]The partition of the predictor space is best understood visually. Recursive binary splitting and Bootstrapping are both central concepts that fell outside the scope of this report. The reader is referred to (21) (12) and (24) respectively for detailed visualizations and explanations.

We approached data preprocessing similarly to Lasso when modeling the listing's price with RF by training 3 models:

1. No data preprocessing
2. Normalized i.e location and scale
3. Normalized i.e location and scale with logarithmized price

In this case, we also searched for the best parameters with a 10-Fold cross validation using *Caret* as with Lasso. The initial tuning grid of values was chosen as follows:

- *mtry* from from 1 to 64 to consider all possible cases of excluded variables
- node.size $= \text{seq}(1, 1001, 10)$

This tuning grid proved to be computationally prohibitive and after 16 hours the process was abruptly ended. We lacked of the computational resources to search along the grid for the best hyper-parameters. After fixing *node.size* to the recommended value of 5 (12), the search over the ideal value for *mtry* proved also to be prohibitive while still performing a 10-fold cross validation. Thus, with limited temporal and computational resources we decided consider three values for *mtry*:

- $14 = \frac{p}{3} - 7$
- $\frac{p}{3} = 21$ which, as already mentioned, is the recommended value for *mtry* (22)
- $28 = \frac{p}{3} + 7$

The rationale was to consider 2 main deviations from the default parameter value to see if there is a tendency towards lower or higher *mtry* values relative to the default. Training time lasted 6 hours in this case. mtry $= \frac{p}{3}$ proved to be the best performing value and we thus kept it. With the identified best value of 21 for *mtry*, we validated the three models with different preprocessing considerations described above on the Berlin subset and tested once on the Munich set.

2.2.4 Generalized Boosting Trees

We further considered building a set of weak CART learners into a strong ensemble by means of gradient boosting. To boost trees was therefore in line with the rationale behind using random forest[3]. At first, we attempted to tune all 4 available parameters in caret (10) with the 10-Fold cross validation framework used for all other models. The initial tune grid looked like this:

- interaction.depth = seq(1,10,1)
- shrinkage seq(0.001, 0.202, 0.04). Best: 0.001
- n.trees = seq(500, 5000, 500)

[3]Gradient decent and further inner workings of GBM fell outside the scope of this report. We refer the reader to (12) and (25) for detailed explanations on the matter.

- n.minobsinnode seq(1,100,10)

As with RF, the process proved to be computationally prohibitive and failed after 24 hours CPU time. To troubleshoot in the same manner as with RF, we fixed those parameters deemed in the literature to be less influential (12) by using the prespecified parameter values and while tuning only *shrinkage*.

The resulting grid looked as follows:

- interaction.depth = 1
- shrinkage seq(0.001, 0.202, 0.04).
- n.trees = 5000
- n.minobsinnode 10

Even when fixing all other three parameters, training time on Berlin set took circa 16 hours. Because of these high computational costs, we just tuned for the simple GBM model without any kind data preprocessing and used the best values for *shrinkage* to train the three intended models. Specifically, with the identified parameters 0.001 for *shrinkage*, we trained 3 GBM models, validated on the Berlin sub set and tested on the Munich test set.

3 Results

3.1 Validation

Table 1: Validation Results Berlin

Model	RMSE	R Squared	MAE
OLS	0.5547540	0.3818374	0.4107917
Lasso	0.4935600	0.4465568	0.3757873
Lasso N	0.4935600	0.4465568	0.3757873
Lasso N + log(price)	0.4436540	0.5181270	0.3442663
DNN	0.3998783	0.6069826	0.3068217
RF	0.4085395	0.6140226	0.3083759
RF N	0.4087117	0.6139556	0.3084462
RF N + log(price)	0.3802465	0.6473361	0.2889315
Boost	0.5786111	0.3622141	0.4219792
Boost N	0.6139971	0.3502980	0.4415882
Boost N + log(price)	0.4221685	0.5688760	0.3249057

Table 1 shows the validation results on the Berlin subset for all trained models described in the previous section. Given that in several cases we preprocessed the dependent variable differently, we carefully adjusted prediction to a log scale before calculating performance metrics. This means that all results shown in table 1 are in log. When considering performance, our best performing model was our RF with normalization of the data over a logarithmic price closely followed by our DNN. This high performance of RF was surprising to us, when considering the lousiness of the hyper-parameter tuning. Recall that RF actually run on prespecified parameter values. It is therefore probable that performance could increase with a more throughout best parameter search. GBM performed poorly. When considering both normalized data and a logarithmic price, GBM performed competitively, even better than Lasso. Lasso delivered consistent results and neither under- nor over-performed relative to the other models. Lasso's regularization did actually lead to a performance increase relative to the unregulated OLS case, which provides evidence to assume that several variables do not contribute to the overall performance.

We also see that normalization does not lead to noticeable increases in performance with exception of GBM where normalized data actually led to worst results. We cannot yet make sense of this behavior. Logarithmizing the price variable, on the other hand, led to a considerable performance increase in all reference models. Recall that our DNN was trained considering normalization and a logarithmic price, so a decrease in performance could sensibly be assumed, if these two prepossessing aspects were missing when trained.

3.2 Test

Table 2: Test Results Munich

Model	RMSE	R Squared	MAE
OLS	0.6330120	0.2460199	0.4687229
Lasso	0.5868852	0.2998020	0.4312133
Lasso N	0.5868852	0.2998020	0.4312133
Lasso N +log(price)	0.6404690	0.2869786	0.4614091
DNN	0.6419999	0.0632258	0.4594905
RF	0.5270313	0.3860209	0.3767017
RF N	0.5304562	0.3782364	0.3798950
RF N + log(price)	0.5456279	0.4245355	0.3867661
Boost	0.6560061	0.2191735	0.4775103
Boost N	0.6575854	0.2427723	0.4705026
Boost N + log(price)	0.5896727	0.3410397	0.4201564

Table 2 shows the validation results on the Munich set for all trained models described in the previous section. All results shown in table 2 are also in log scale. As it was expected, performance dropped considerably on the test set. We had high expectations of generalizability, as both Berlin and Munich are located in Germany. Such a considerable performance decrease was therefore disappointing. Our best performing model was again RF, this time without any sort of preprocessing. Against the validation results, data preprocessing proved in this case to be detrimental to performance. Our DNN performed poorly just on par with the poor performance of GBMs and OLS. With a mean performance drop of circa 8.5 RMSE units, Lasso based models proved to be the most generalizable relative to the mean performance drop of 0.12, 0.24 and 0.16 RMSE units for RF, DNN and GBM respectively. The latter refers to the overall variation of performance between the validation and the test set.

3.3 Interpretability

With regard to interpretability, we calculated the variable importance for the 20 most influential variables. We did initially so for all lasso models only, as their variable selection is intrinsic to the model architecture. Later, we did the same for GBM based models using the *caret* package (10). Interesting is how data preprocessing affects the variable importance as shown in figure 6 and figure 7 in the appendix. In all cases, variables "bedroom" and "accommodates" showed the highest importance. All sorts of metric "review scores" also proved to be very influential. To our delight, although differently for each considered plot, several variables arisen from our image analysis showed a high level of influence on our models, most interestingly the "mean_brightness" of the pictures, a variable present in almost all variable importance plots in figure 6 and figure 7 in the appendix. Several objects identified and the correlated color temperature also show a high degree of importance as shown in the plots.

4 Conclusions

There are several steps that can be taken to improve the quality of the data taken from the image analysis. These steps were not taken by us as this would have go beyond the available time for this project and beyond the computational power we had in our disposal.

As already mentioned, one could gain more insight from the available raw data by re-coding text and time variables into working metric ones. Insights of how this could be conceived are, as already mentioned, discussed in section A.12 in the appendix.

Scraped images could be pre-processed before detecting objects on them. This could be done by classifying the pictures as either pictures showing the inside of the flat or not. Such and deep neural net is available on *monk ai* and linked in our repository. Images which do not show the inside of the flat should not be considered in the analysis we described here. Furthermore a few images were rotated, which could influence the performance of the RetinaNet for those images. Rotating the images to the right angle would further improve the data. "Cleaning" the image data in these ways would improve the noise in the data and should increase the significance of the different variables created from the picture analysis.

Furthermore, the analysis of the room temperature could be improved. The number of clusters created per picture representing the dominant colors of the image could be raised. We were not able to do this, as this would have required a lot of computational power. Also, by weighting the mean of the CCT per dominant color with the size of the cluster for each dominant color, the approximation of the room temperature could be improved.

Moreover, the cluster analysis done to analysis the room temperature with the suggested improvements could also be used to approximate the perceived brightness of the image. It should also be noted that the analysis was fractured by using two programming languages. A first step when improving the work described in this paper, would be to translate the code written in R to Python.

When modeling, we could also profit, again, from higher computational power to perform a more extensive search of best values for all of our model's hyper-parameters. With regard to computational constrains and being well aware that a complete grid search for tuning parameters could deliver better results, we pondered about how to make the most from what we had available. The computational problems, concerning the various comparison models, were probably exacerbated by the underlying usage of 10-fold cross validation. The later framework was still kept for comparison purposes between models, as we deemed that an homogeneous training framework could aid comparison. A more sensible hyper-parameter tuning approach would be to use the already calculated Out-of-Bag (OBB) error while dropping the usage of 10-Fold CV to reduce the computational burden (12), (22).

Overall, we were able to show that our DNN based image analysis, specifically object identification, CCT and PB analysis delivered a high degree of useful information to competitive modeling performance on the seminar scale. Furthermore we showed that the predictive power for of an relatively small DNN can be compared to the predictive power of regression models

A Appendix

A.1 Variable Importance Plots

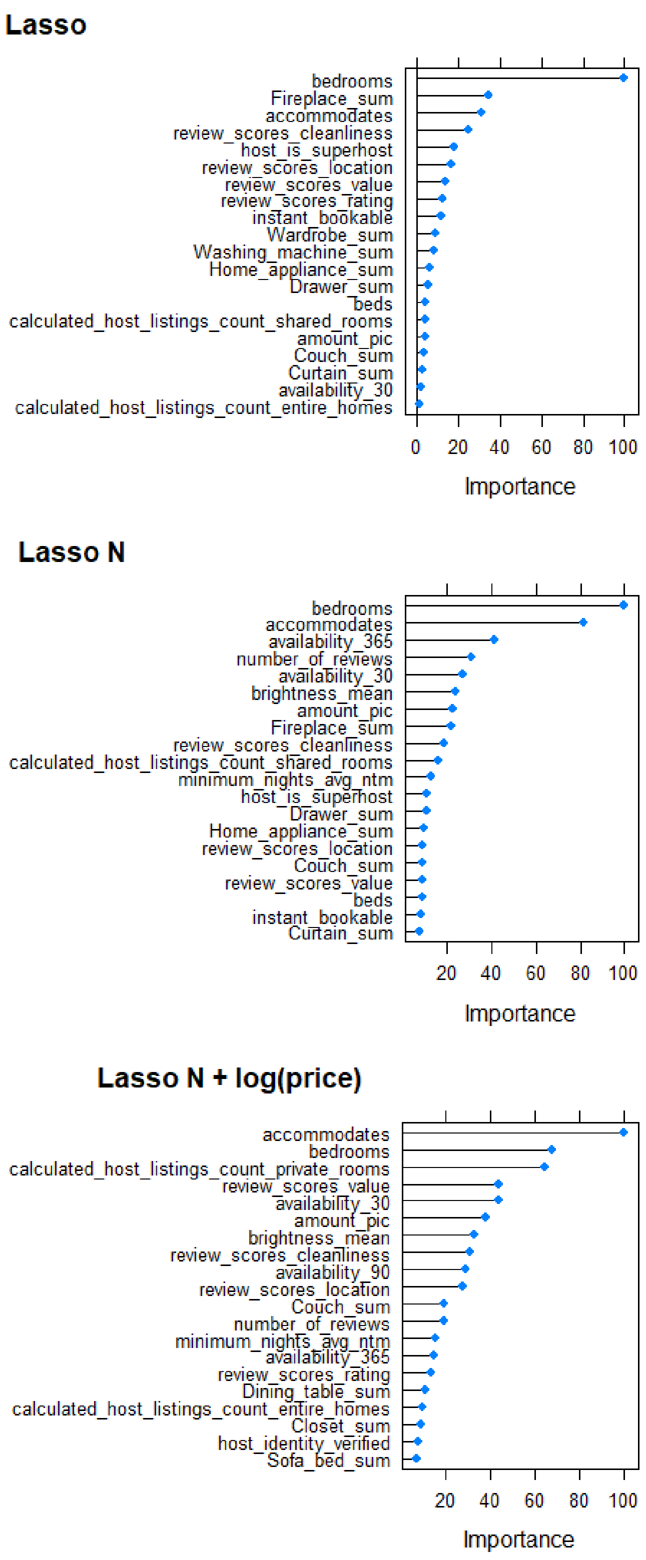

Figure 6: Variable Importance for different Lasso models

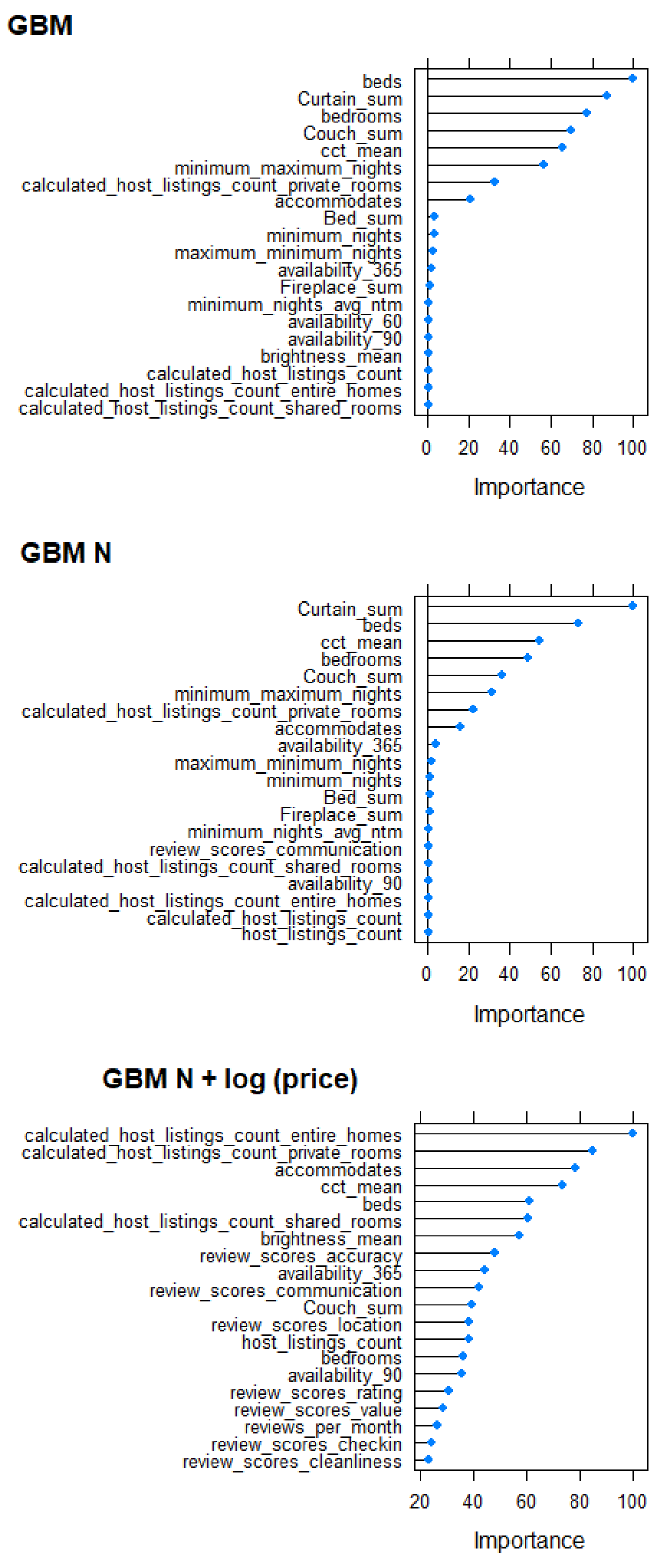

Figure 7: Variable Importance for different GBM models

A.2 Full list of classes used to identify objects

Alarm clock, Bathroom cabinet, Bathtub, Bed, Bookcase, Ceiling fan, Chest of drawers, Clock, Closet, Computer monitor, Couch, Curtain, Dining table, Dishwasher, Drawer, Fireplace, Gas stove, Home appliance, Infant bed, Microwave oven, Sofa bed, Toilet, Wardrobe, Washing machine

A.3 Scraping the pictures for the image analysis

While we quickly found a way to scrape one picture per host from the Airbnb website, we had trouble finding a way to scrape all pictures from a host. This was due to the fact, that we had no previous experiences with html or java. We had to find a way to locate each picture and download it in its best quality. This meant that we had to navigate automatically over the website of Airbnb. While we tried to do this with the Selenium package, we in fact did not succeed to do so. The base structure of the scraper we use was given to us by the group of Dominik and Lars. For this we want to thank them here again. We managed to make the scraper work on Colab, enabling us to scrape the images directly into our google drive.

A.4 Picture analysis in R

As we wanted to write our project in R, we naturally first tried to implement a multi detection model in R. However, it should be noted that we found this to be very difficult. It seems that R is not very popular for analyzing images. Not only are most deep learning libraries natively written in python, but also sources for extracting training data such as OpenImages can be accessed more easily with python. After a time of unsuccessful trying, we therefore abandoned our initial approach and decided to implement our Image Analysis in Python.

A.5 Setting up tensorflow in R

It turned out to be very cumbersome to set up *keras* and *tensorflow* in R. The only way we found worked, was to run R over anaconda. This was necessary because R needs a connection to python to run *keras* in R. We therefore recommend anyone who wants to program in the field of deep learning to use python. We further faced critical problems in Sebastians interface, as it did not support the usage of several advanced vector extensions (AVX) considered as a precondition to run *tensorflow* (26). Therefore the areas of responsibility for this project were allocated as follows: Sebastian concentrated on cleaning the data as well as preparing and training the various comparison models, crucial for interpreting and evaluating the results. Tobias focused on the image analysis and the DNN for price prediction, as well as the initial analysis of the spatial and text data. This resulted in an equal contribution to the final paper.

A.6 Multi object detection

As described shortly, finding resources for multi object detection in R was also very difficult. For the reasons described above, almost all resources for deep learning are written in python. While there were some resources, it quickly became apparent that debugging deep learning code in R was very difficult as very few people use R for deep learning and thus very few questions about errors were answered on the internet. Furthermore, retrieving labeled images for training an multi object detection DNN is almost exclusively done in python. We downloaded a train set of images several times from OpenImages. This cost a lot of time and storage. We ultimately deleted the downloaded images again after finding that a training set of images was already available in the repository of Tessellate Imaging that we used in the end. The repository can be found under: `https://github.com/Tessellate-Imaging/Monk_Object_Detection`. For the reasons named above we conducted our image analysis on python, using Colab pro plus, that we connected to a Google Drive account.
While trying to detect objects in the pictures we also looked at different models and the possibility to mplement them on R: Yolo (You only look once) and MASK-RCNN. While Mask-RCNN would have been a viable alternative, YOLO was designed for object detection in real time. It therefore is not as accurate as Mask-RCNN or RetinaNet. (27), (28)

Considering the size and complexity of the RetinaNet, our overall goal to predict the price of an flat with an DNN and our goal to create variables from the image analysis that could have an impact when predicting flat prices, we feel it would have been contra productive to create our own multi object detection model. Such an model would surely not perform as well as already created architectures for multi object detection. While trying to improve the RetinaNet and thus creating an own version of an established multi object detection model would be very interesting, it was not possible in the given time. Another possibility would have been to use other implementations of RetinaNet e.g. from keras, and train the network with labeled data from OpenImage. However we would have still have done the same as the repository we eventually used, more or less coping each step. We did not feel comfortable with the decision to use the repository and build upon it, but we hope that given the work and results presented in the paper, it is understandable that we ultimately stuck with this decision in order to have a working multi object detection model. The infrastructure of the repository and the use of Colab and Drive enabled us to detect objects on the over 120000 pictures scraped from Airbnb, and generate variables which then could be interpreted easily and which added predictive power to our data set.

A.7 Problem with the analysis of room temperature

After the correlated colour temperature analysis was completed, some values were very large. Hosts with these large values had to be removed. Also, the analysis took very long even on Colab pro plus. The analysis ran 8 days.

A.8 Attempts to train a deep neural network with a data frame containing data for every single image

While our attempts were not successful, we will shortly describe the steps we took when we tried to train a DNN with data for every image. We assume that we problem we faced was that of vanishing descent. As a variable was created for each image and class, there was not much information inside each of these variables. To deal with this, we first deleted the information for the last 8 images. Secondly, we varied the dropout rate, the batch size, and the architecture. We also introduced the leaky Relu activation function. However, none of these attempts were successful. This problem could be dealt with by cleaning the image data with the approaches suggested in the paper.

A.9 Our initial approach of k-fold-cross-validation and its problems

Initially we used Ten-fold-cross-validation to validate our deep neural net for price prediction. However, since for our data set the mean average error does not rise for most models after a given number of epochs, it was difficult to determine the point at which our model overfits. We began to use bigger models as these seemed to work better when predicting the test set. Also with larger models, the mean average error did increase after approximately 50 epochs. We therefore presented such a model in the presentation, as here we were more confident that we had stopped before the models overfits and could visually show this.

But because we used k-fold-cross validation we overlooked, that the model fluctuated strongly while training. Therefore, the performance of the model varied strongly when training the same model again. This was however identified after the presentation and thus k-fold-cross validation was not used, but rather the validation set described in the paper. This resulted in a better validation process and a different architecture of the model. Since we had a large data set for Berlin, k-fold cross validation was also not as necessary as it would have been for smaller data sets.

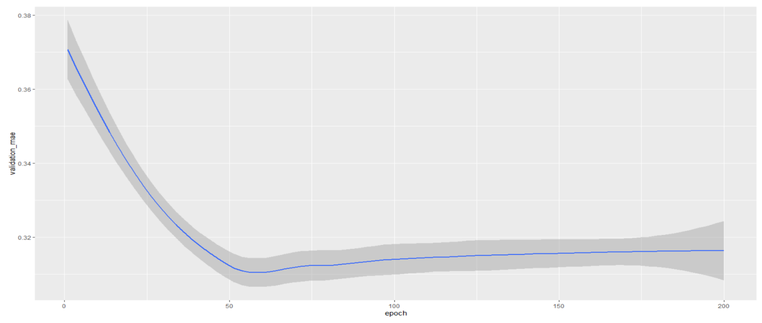

Figure 8: Training process of the DNN with k-means-clustering and large architecture

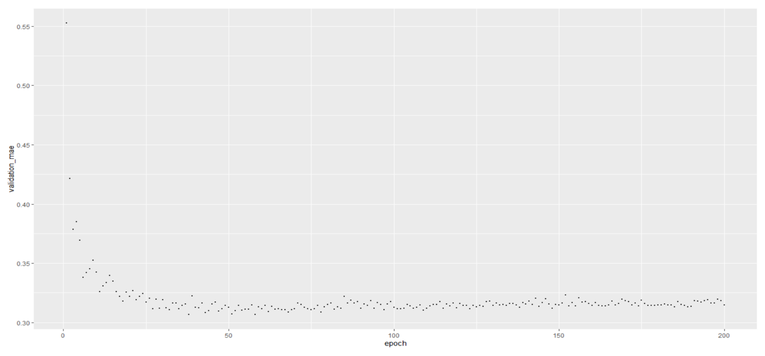

Figure 9: Training process of the DNN with k-means-clustering and large architecture

A.10 Results of different architectures of the DNN

Results of different architectures of the dnn for Berlin and Munich, using a dropout rate of 0.17, the tanh activation function in each layer and the means squared error as the loss function. The architecture is given as sequence, denoting the number of nodes for each layer

Table 3: Results of different architectures of the DNN

Architecture	Rmse for the test set t	Rmse for the train set	Rmse for the Munich data set
6, 6	0.4112	0.4042	0.6307
12, 12	0.3972	0.395	0.6148
12, 12, 12, 12	0.4127	0.3915	0.6168
64, 32, 16, 8	0.4036	0.3697	0.6232
124, 64, 32, 16	0.4049	0.3852	0.6353
189, 126, 63	0.4266	0.3665	0.8549

A.11 Further ideas for image analysis

The cluster algorithm to determine the dominant colors could also be replaced by a convolutional net. Instead of using a k-mean clustering a convolutional neural network could be used to create a feature map of the image. This feature map could then be used to calculate the perceived brightness or CCT for each cell of the feature map. The final perceived brightness or CCT of the image would again be the mean over the determined values. However, images labeled with an accurate CCT and perceived brightness values would be necessary to train such networks. We therefore settled for the simpler approach. A working model trained with such data would be useful to quickly determine brightness or CCT.

A.12 Text and spatial analysis

We also started to analyze the text and coordinates of the Airbnb data. We stemmed the text of the reviews and created word clouds. However as already described, we did not include the text in our analysis.

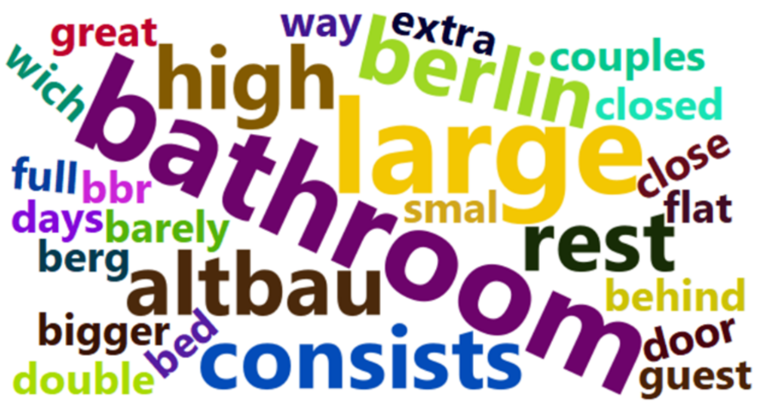

Figure 10: Example: wordcloud

We also analyzed the spatial data in Berlin. Since the data seemed to be randomly distributed, we did not include the coordinates of the flats in our price prediction.

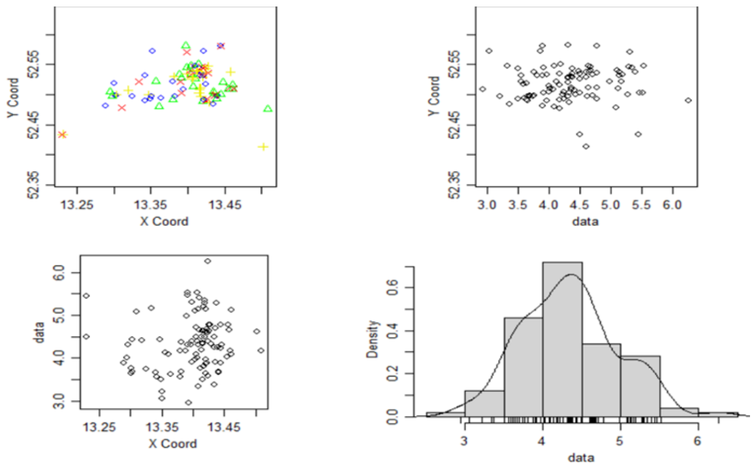

Figure 11: Distribution of spatial data

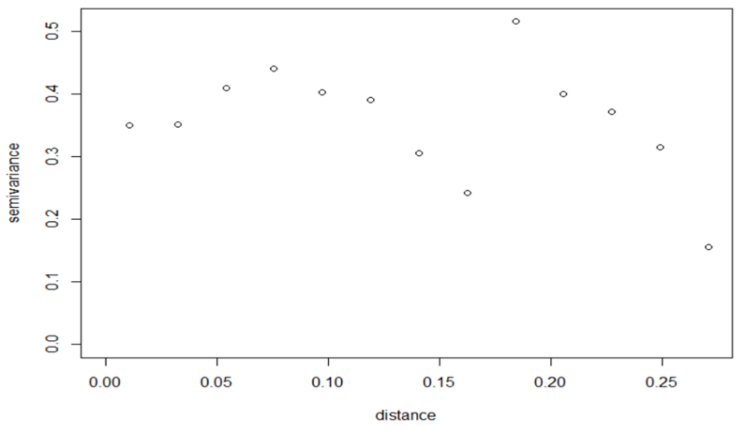

Figure 12: Variogram of Berlin

References

[1] Inside Airbnb, "Inside airbnb. adding data to the debate," 2022.

[2] N. Tierney, D. Cook, M. McBain, and C. Fay, *naniar: Data Structures, Summaries, and Visualisations for Missing Data*, 2021. R package version 0.6.1.

[3] Airbnb, "Airbnb: einzigartige unterkünfte und aktivitäten," 02.03.2022.

[4] L. alAli, F. Alnajjar, H. A. Jassmi, M. Gocho, W. Khan, and M. A. Serhani, "Performance evaluation of deep cnn-based crack detection and localization techniques for concrete structures," *Sensors (Basel, Switzerland)*, vol. 21, no. 5, 2021.

[5] T.-Y. Lin, P. Dollár, R. Girshick, K. He, B. Hariharan, and S. Belongie, "Feature pyramid networks for object detection."

[6] T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollár, "Focal loss for dense object detection."

[7] J. Hernández-Andrés, R. L. Lee, and J. Romero, "Calculating correlated color temperatures across the entire gamut of daylight and skylight chromaticities," *Applied optics*, vol. 38, no. 27, pp. 5703–5709, 1999.

[8] I. Pavan Kumar, V. P. Hara Gopal, S. Ramasubbareddy, S. Nalluri, and K. Govinda, "Dominant color palette extraction by k-means clustering algorithm and reconstruction of image," in *Data Engineering and Communication Technology* (K. S. Raju, R. Senkerik, S. P. Lanka, and V. Rajagopal, eds.), vol. 1079 of *Advances in Intelligent Systems and Computing*, pp. 921–929, Singapore: Springer Singapore, 2020.

[9] "Hsp color model - alternative to hsv (hsb) and hsl," 01.11.2019.

[10] M. Kuhn, *caret: Classification and Regression Training*, 2021. R package version 6.0-90.

[11] G. James, D. Witten, T. Hastie, and R. Tibshirani, *An Introduction to Statistical Learning: With Applications in R*. Springer eBook Collection, New York, NY: Springer US and Imprint: Springer, 2nd ed. 2021 ed., 2021.

[12] T. Hastie, R. Tibshirani, and J. H. Friedman, *The elements of statistical learning: Data mining, inference, and prediction / Trevor Hastie, Robert Tibshirani, Jerome Friedman*. Springer series in statistics, New York: Springer, 2nd ed. ed., 2009.

[13] J. Friedman, T. Hastie, and R. Tibshirani, "Regularization paths for generalized linear models via coordinate descent," *Journal of statistical software*, vol. 33, no. 1, pp. 1–22, 2010.

[14] N. D. Lewis, *Deep learning made easy with R : a gentle introduction for data science*. Manning Publications, February 2016.

[15] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *J. Mach. Learn. Res.*, vol. 15, no. 1, pp. 1929–1958, 2014.

[16] H. B. Martin Riedmiller, *RPROP - A Fast Adaptive Learning Algorithm.* 1992.

[17] V. Bushaev, "Understanding rmsprop — faster neural network learning," *Towards Data Science*, 02.09.2018.

[18] G. Seif, "Understanding the 3 most common loss functions for machine learning regression," *Towards Data Science*, 21.05.2019.

[19] Y. Bengio, "Practical recommendations for gradient-based training of deep architectures," in *Neural Networks: Tricks of the Trade* (G. Montavon, G. B. Orr, and K.-R. Müller, eds.), vol. 7700 of *Lecture Notes in Computer Science*, pp. 437–478, Berlin, Heidelberg: Springer Berlin Heidelberg, 2012.

[20] D. Masters and C. Luschi, *Revisiting Small Batch Training for Deep Neural Networks.* arXiv, 2018.

[21] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone, *Classification And Regression Trees.* Routledge, 2017.

[22] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.

[23] M. N. Wright, S. Wager, and P. Probst, *ranger: A Fast Implementation of Random Forests*, 2021. R package version 0.13.1.

[24] B. Efron and R. J. Tibshirani, *An introduction to the bootstrap.* New York: Chapman & Hall, 1993.

[25] R. E. Schapire, "Explaining adaboost," in *Empirical Inference*, pp. 37–52, Springer, Berlin, Heidelberg, 2013.

[26] Stack Overflow, "python - your cpu supports instructions that this tensorflow binary was not compiled to use: Avx avx2 - stack overflow," 04.03.2022.

[27] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," pp. 779–788, 2016.

[28] K. He, G. Gkioxari, P. Dollar, and R. Girshick, "Mask r-cnn," in *2017 IEEE International Conference on Computer Vision (ICCV)*, IEEE, 2017.