Airbnb Berlin Price Prediction

Seong Woo AHN, Paul BÉRARD, Leila BERRADA, Tanguy BLERVACQUE

1. Introduction

This project aims to predict the price of a night's accomodation offered on the short-term rental website Airbnb in the city of Berlin, Germany. It is derived from a data challenge proposed by `dphi.tech`.

After preprocessing the data through various techniques (handling missing values, categorical data, feature scaling, etc.), we applied all Machine Learning approaches taught in the course (Linear Regression, Decision Trees, Random Forests, AdaBoost, etc.) to predict the target variable *Price* with the best accuracy possible. We then compared the performances of these models using the following accuracy metrics: *MSE*, *RMSE*, *MAE*, *Adjusted-R^2*.

See code on `github.com/paulberard/airbnb-price-prediction-model` or click here.

See Annex for visualization of obtained results.

2. Data preprocessing

2.1. Dropping irrelevant features

Hereafter is the list of variables that were dropped from the dataset, as well as a short explanation of why they were removed. Please check the code on `GitLab` for the details of the justifications.

Listing ID. This feature identifies each unique instance, but is not relevant for our model.
Listing Name. This variable represents the name of the listing, which has been anonymized and therefore does not bring any information to the dataset.
City, Country Code, Country. These features were dropped because all instances had the same categorical values, respectively `Berlin`, `DE` and `Germany`, and were therefore irrelevant to train a price prediction model.
Square Feet. More than 98% of the instances had missing values for this feature, making it unusable.
First Review, Last Review. These variables, representing the date of the first and last review were dropped because imputing the missing values would be impractical and dropping instances with missing values could create a bias.
Business Travel Ready. This feature was dropped because all instances had the same categorical value `f`, which would be useless to train our model.
Postal Code. Brought no more information than what was already in *Latitude*, *Longitude* and *Neighbourhood Groups*.
Neighborhood. At first we wanted to keep this feature. However, there are 63 neighborhoods, which would have created 63 more features after a one-hot encoding. That would have been too many variables for an information that is already present in *Latitude*, *Longitude* and *Neighborhood Group*, which led us to drop that feature.

2.2. Handling missing values

We used different strategies depending on the feature and the category of missing values for each feature. See **Figure 1** for a global visualization of missing data in the features.

Listwise deletion
We took a very conservative approach and only used this strategy when a very small number of instances had missing values for a certain feature, which could then be considered to be Missing Completely At Random (MCAR).

Price. Only 9 out of the 15 692 instances had missing values for the target variable, and were therefore removed.
Accomodates, Bathrooms, Bedrooms, Beds, Guests Included, Min Nights, Host Name, Host Since, Is Superhost, Property Type. We dropped the instances with a missing value in one of these features as they represented, in total, less than 3% of the entire dataset.

Imputing missing values
At first, we intended to impute some features like *Neighbourhood* and *Postal Code* by using the rest of the information in the dataset (closest point based on *Longitude* and *Latitude*). However, we decided to drop those columns at the end. The choice of imputation was then let free, so that each one of us could run some tests and see what worked better for each algorithm (mean, median, most frequent value imputations).

2.3. Handling outliers

We purposely did not do anything about outliers, as some models are immune to them, and also because we wanted to keep the information of high prices that might characterize a certain neighborhood or area of Berlin.

2.4. Feature scaling

Just like for imputing missing values, the choice of scaling was let free to choose the best one for each algorithm. Therefore, some used *StandardScaler* and others Min-Max Scaling.

2.5. Handling categorical data

For categorical data, the common strategy was to one-hot encode the following features: *Host Response Time*, *Property Type*, *Room Type*. The missing values in these features were dealt with by creating an additional `unknown` class for each categorical feature. These values were furthermore not scaled, as it is common best practice to leave categorical one-hot encoded data as is.

2.6. Feature engineering

We engineered 2 features that we considered might be a nice addition to our variables:
Gender. We used the feature *Host Name* and an NLP model (LSTM cells and a Dense layer) to get the "gender" based on the name of each host. For this study, we only considered 2 genders: Female and Male. (# No hate on the others.)
Distance From Barycenter. We first computed the spatial barycenter (b_{lat}, b_{long}) of all accomodations by averaging on *Latitude* and *Longitude*, and then computed the euclidean distance from each accomodation's position to the barycenter, which feeds the data of this new feature.

2.7. Vizualisations

Vizualisation of our instances on the map of Berlin on **Figure 2**. Vizualisation of prices in the neighborhood groups on **Figure 3**.

3. Price prediction models

After preprocessing the data, we decided to train 8 different models in order to compare their performance. For this comparison to make sense, we all applied the same train-test split. As you can see for example in the notebook *notebooks/Tanguy/LinearRegression*, we chose a 80%/20% split between train and test sets. We also made sure our split preserved the distributions of all variables in both train and test data.

3.1. Linear Regression

The first model we decided to train was a linear regression. The only additional preprocessing done for this model was to eliminate the *Neighborhood* feature, as one-hot encoding it would have doubled the number of features and we believe the information for this variable is already present in *Latitude*, *Longitude* and *Neighborhood Group*.

After train/test splitting, we decided to standardise all numerical features using *StandardScaler*. The reason for this is mainly that it seemed to work best.

After looking at the features, it was clear that some might be much more informative than others. In addition, there were still many at this stage (58). For those reasons, we decided to apply two different data dimension reduction techniques and compare their results in order to choose the best one.

We started by performing forward and backward stepwise feature selections. While forward selection suggested we use a 33 feature model, backward selection opted for 41 features. As the MSEs seemed very close in both cases, we chose to go for forward selection, as it allowed us to reduce the dimension even further (**Figure 4**).

We then tried to reduce dimension using PCA. According to **Figure 5**, no number of components clearly stood out, so we decided to try out 3, 11, 36 and 51 components, as they correspond to the discontinuity points in the plot. This decision was also made by looking at the cumulative explained variance plot which you can find on the notebook (36 and 51 validate the at least 80% explained variance rule). The model with the best performances (MSE, RMSE, MAE and Adjusted R^2) is the 51-feature model, nevertheless it is interesting to note that the 3 feature model has a very good predictive power given its complexity.

According to all our performance metrics (MSE, MAE and Adjusted-R^2), the model given by forward stepwise selection was the best one. Nevertheless, let us note that the features chosen by stepwise selection were nearly the same as the ones that drove the first few main principal components of the PCA (see notebook for details). This is a double proof that they accounted for the most explained variance in the model.

To conclude on the linear regression, we lead a quick feature analysis using SHAP. Shapley values are a tool that gives us insight on how each feature influences the final prediction (for a single instance on average). By looking at the average influence of features (**Figure 6**), we can see that *Room Type Entire home/apt*, *Accomodates*, *Neighborhood Groupe Mitte*, *Room Type Private room*, *Bedrooms*, *Neighborhood Groupe Pankow*, *Bathrooms*, *Neighborhood Groupe Friedrischain-Kreuzberg* and *Guests Included* have the most clear-cut effect. When looking at the correlations table a bit higher in the notebook, we can see that theses features are also the ones with the highest correlation to the prediction.

3.2. Decision Tree Regression

A Decision Tree is a weak supervised learner whose goal is to predict a dependent variable by learning simple decision rules inferred from the data features. It is a form of binary tree that recursively segments the feature space $\mathcal{X}$ to create K hyper-rectangles R_k, $k \in \{1, ..., K\}$, where each of them is associated with a constant value γ_k. For regression tasks, γ_k is determined as the average value of the instances in R_k. In *scikit-learn*, the corresponding estimator is found in *sklearn.tree.DecisionTreeRegressor*.

The preliminary data preprocessing did not include the imputation of missing values so that each model could use the most appropriate imputation strategy. Therefore, we created different pipelines combining mean, median and most frequent value imputation strategies with min-max and standardization scaling strategies. We tested them all on a default-parameter decision tree and determined, by choosing the method with the lowest *RMSE*, that the best approach was {median imputation + standard normalization}.

To give our decision tree a better chance of finding discriminative features, we performed dimensionality reduction through several methods: feature selection (forward and backward stepwise selection) and PCA. See **Figure 7** and **Figure 8** for the evolution of the MSE as a function of the number of features selected by each method. While the latter did not yield satisfying results at all, forward stepwise selection outperformed all other reduction methods, as evidenced by the cross-validation score calculated on 10 folds at each selection step.

Finally, we performed a 10-fold cross-validated Grid Search to determine the optimal hyperparameters of our decision tree (namely, *max_depth* and *min_samples_split*), which allowed us to considerably improve the performance of our model, as shown by our accuracy metrics. See notebook for the details.

3.3. Bayesian Ridge Regression

In Bayesian Ridge Regression (BRR), the goal is to determine the posterior distribution of the model parameters rather than finding one single optimal value for each of them. Such techniques include regularization parameters (α and λ) tuned to the data in the estimation procedure. To do so, we introduce uninformative priors over the hyperparameters of the model. The regularization in BRR is equivalent to finding the posterior distribution under a Gaussian prior over the model's coefficients. The priors over α and λ are chosen to be Gamma distributions.

After testing all {imputation + scaling} pipelines to finish the preprocessing, we opted for the one with the lowest RMSE: {most frequent value imputation + a mix of min-max scaling and standard normalization}. Standard normalization was used for numeric features thought to follow a Gaussian distribution.

Regarding the dimensionality reduction, we performed a forward stepwise selection, a backward stepwise selection and a PCA. See **Figure 9** and **Figure 10** for *MSE* $= f(k)$, where k is the number of features in the model $\mathcal{M}_k$ kept by each feature selection method. We compared their results and backward selection came out as the winning strategy: not only did it have a better 10-fold cross-validated score, but it also returned a lower number of features than FSS.

After reducing the dimension of the feature space, we executed a cross-validated Grid Search to tune the model's hyperparameters α and λ to improve its performance.

3.4. KNN Regression

Details concerning this part can be found in the notion page under "KNN Regressor".

KNN regression which stands for K-Nearest Neighbors regression is weak learner that aims to attribute a value to a given data point by looking at this point's k nearest neighbors, and evaluating the mean value amongst these k values. The estimator that is available in *scikit-learn* is called *sklearn.neighbors.KneighborsRegressor*.

For our task at hand, the strategy employed to proceed in training such an estimator was the following:

1. Determine the best imputation and scaling strategy for the regressor.
2. Select the best features for KNN by using forward and backward stepwise selection.
3. Evaluate the performance of KNN with PCA.
4. Tune the hyperparameters through a grid search in the selected/reduced feature space.

For the first step, different pipelines combining mean, median, and most frequent value univariate imputation strategies with min-max and standardization scaling strategies were created. After quickly testing these pipelines on a default-parameter KNN regressor on the test data set, it was concluded that {median imputation + standard scaling} (for numeric values) was the best strategy for the estimator at hand. Multivariate stochastic regression imputation was tested as well, but as some imputed values didn't make any sense (ratings above /10, frequencies above 100%), this imputation method was discarded. See **Figure 11** and **Figure 12** for distribution comparison between scaled imputed numeric values of train and test set.

Out of the feature selection/dimensionality reduction methods, backward selection yielded both an overall best result in terms of performance (cross validation score was calculated on 10-folds at each selection step), and also returned a relatively low number of features (17). Further details on these selected features can be found on the Notion page. You can look at **Figure 13** and **Figure 14** for the evolution of the MSE for the forward and backward selection processes. See also **Figure 15** for the evolution of the MSE for the PCA analysis. Most of the variance are captured in the 20 or so variables, but clearly performance is worse than feature selection methods.

Finally, with the selected features, a cross-validated grid search on the training set was performed to determine which hyperparameters were optimal. The evaluated parameters were *weights*, *n_neighbors*, *leaf_size*. After an extensive exploration of the hyperparameter space the best results were yielded for a *leaf_size* of 1, *n_neighbors* equal to around 50 and "distance" weight function prediction strategy (closer neighbors of a query point will have a greater influence than neighbors which are further away). See last table for details on the performance.

3.5. Random Forest Regression

Random Forests (RFs) are an ensemble learning method. Indeed, they use Bagging and Aggregation of weak learner (Decision Trees) in order to perform classification or regression tasks. For regression, the prediction is the mean or average value of all the trees in the forest.
In our study, we wanted to see how efficient such an ensemble learning method was for predicting prices in Berlin. To do so, we went through multiple steps to get the best predictions possible with this model.
We first tried to see what kind of imputation would be interesting for such a model, and the result was that a median imputation would give a better RMSE.
Then, even if a Random Forest is not so much affected by the lack of normalization, we compared the RMSE of a pipeline with no normalization with one with a *StandardScaler*, and one with min-max scaling. The scores proved min-max scaling to be the most efficient approach.
Then, we performed a Random Search to tune the hyperparameters of the model which allowed us to obtain better scores.
Also, even though Random Forests are not affected by the curse of dimensionality, we wanted to make a test: compare the scores of the RF if we used the features returned by a forward feature selection (**Figure 16**) and the features that the Random Forest itself considered as relevant (given that Random Forests can be used for feature selection): see **Figure 17**. By performing these tests each time with the features returned by each method, we realized that the best RMSE was obtained when we kept all the features. Our optimized Random Forest model had an RMSE of : 31.7.
Finally, even if Random Forests are hard to interpret, we can plot some trees of the forest (see **Figure 18**) or use Shapley Values to help us understand how the different features impacted the model. See **Figure 19** and **Figure 20**. Here for instance, we can see that the price is higher when the listing can accommodate more people. However, if the number of rooms is low the price will be lower too.

3.6. SVR

Support Vector Regression are the way to perform a regression task using the Support Vector Machine method. They use the same principles as SVM classification, for instance choosing maximal margin. The main differences however is that since we want to perform regression, there are no separable classes, therefore there is a strong need of having a tolerance value ϵ.

Therefore, we aimed to see if such a technique would be interesting and efficient for price prediction.

We first did a quick test of the SVR algorithm and noticed that the R^2 score was negative, meaning that if we had predicted the mean value of the prices, we would have made a better prediction than our model. This was not ok! Our data and the use of the algorithm were not adapted for our predictions. Let's change that!

First, we tested 2 types of imputations and compared them to see which one would give us the best results. Although for this part, the RMSE were not that different, we got a better result with imputing all the missing values with the median value. Then, in order to get the R^2 score to be positive, we scaled our data with *StandardScaler* from *sklearn*.

After that, we wanted to deal with the curse of dimensions. To do so, we tested if a PCA would give us a better score. However it was not effective. The RMSE were higher, so we dropped that method. We also tried to perform feature selection and a Grid Search but it required too much computational time.

After searching how to have a model that would train faster and get better results we switched to use the sklearn model: Linear SVR. Linear SVR applies linear kernel and works well with large datasets. The use of a linear Kernel proved to be more effective. We performed a Grid Search on it to tune the hyperparameters. We also did a forward feature selection and found that 38 features were needed for this algorithm (**Figure 21**).

We thus ran, as our best algorithm for this method, a pipeline with a median imputation, a standard scaling, 38 features and the tuned Linear SVR model. We ended up with an RMSE of 31.92, much better then at the start of our study on this method.

Finally we ended up this part with the use of Shapley Values to help us understand how the different features impacted the model. See **Figure 22** and **Figure 23**. Here we can see the type of accommodation lead the price to be higher while location tends to get it down.

3.7. AdaBoost

For the AdaBoost model, the inital approach was to employ the same strategy as the one employed for the KNN regression model. However, an issue with doing so was that feature selection was being done on a *sklearn.ensemble.AdaBoostRegressor* with its default parameters (base estimator being a Decision Tree of maximal depth 3). What wasn't realized at this moment was how bad initial hyperparameters can strongly affect the feature selection phase. In fact, following the same order of Feature Selection then Hyperparameter tuning, the results at the end were ultimately very bad relatively to other models.

The revised strategy plan for AdaBoost model was therefore changed to the following in a second iteration:

1. Determine the best imputation and scaling strategy for the regressor.

2. First iteration on a hyperparameter search. Try to converge to good parameters before feature selection.

3. Select the best features for the adjusted AdaBoost model by using forward and backward stepwise selection.

4. Further tune the hyperparameters through a grid search in the selected feature space for optimal values.

For the imputation and scaling strategies, by fixing the AdaBoostRegressor's random_state (this was crucial because of the stochastic nature of the estimator), the best resulting configurations were for mean imputation with a standard scaling for numeric features that were deemed Gaussian in distribution, and min-max scaling for the rest of numeric features. The selection of the so-called Gaussian features was done purely approximately, looking only at the visual distributions: these were **Host Since, Latitude, and Longitude**.

An initial hyperparameter search was done to make sure the features were going to be selected in a parameter space where the model performed decently. Afterwards, for the feature selection, once again backward selection was the winning strategy. Around 30 features were selected for this phase, and you can look at the evolution of the MSE in **Figure 24**.

Finally, a final cross-validated grid search was executed to get the very best performance out of the model.The resulting model used a Decision Tree with a *max_depth* of 10 and a *min_sample_leaf* of 10 as a base estimator, and the the learning rate was set at 0.001 with 200 estimators for the boosting parameters.

One thing that should be noted in all of this was how time consuming AdaBoost was, with the backward selection process lasting more than 3 hours, and the final grid search over 5 hours, despite efforts to optimize the whole process (e.g. by lowering the number of cross validation folds).

3.8. XGBoost

The last model we decided to experiment on was the XGBoost. For this model we use the initial preprocessing and we delete the *Neighborhood* feature as we believe the information it holds is already present in other features. After the same train/test split as in the other models we decided to standardise all the numerical features using *StandardScaler*.

The first step here was to fine-tune our model's parameters. For this, we start by fixing the learning rate *eta* = 0.1 and looking for the best *n_estimators* given this. Once we have found *n_estimators* = 46, we fix this parameter as well and look for the best (*max_depth*, *min_child_weight*) combination. We find (*max_depth*, *min_child_weight*) = (4, 2) and finish the process by looking for the best *gamma*, which is 0.

Once the model's parameters have been fine-tuned, we want to look for the best model given our features. As in previous notebooks we consistently observed that stepwise subset selection was more effective, we use forward stepwise selection. Using **Figure 25**, we select three possible models. First, the minimum MSE is obtained for 44 features so we will build this model. Second, we observe that the MSE stabilizes around 15 features so we will also try a 15 features model. Finaly, because boosted gradient trees are powerfull even with many features, we will try out the full features model.

Looking at the results of the three models on the test set, we can see that the full features model is the best on all metrucs except for adjusted R-square, where the 15 features model is the best. Nevertheless, as they have nearly the same adjusted R-square, we will select the full feature model. We are not very surprised by this, as boosting usually handles models with a lot of features individually not very explicative but explicative all together pretty well.

To conclude on XGBoost, we lead a quick feature analysis using SHAP (**Figure 26**). We can see that the main features driving the predictions are the same as for the other models, even though the impact of a feature in a given instance is not always the same, as XGBoost is capable of understanding non-linear patterns (this was not the case of the linear regression for instance).

4. Conclusion

After the preprocessing and optimization of our 8 models, we can compare their performance using the metrics we chose, in the following table :

Table 1. Performance of models

Model	MSE	RMSE	MAE	Adjusted-R^2
Linear Regression	1039.5	32.2	20.4	0.47
SVR	1033.62	32.14	20.31	0.47
Bayesian Ridge	1449.62	38.07	20.33	0.38
KNN	1074.76	33.32	19.94	0.45
Decision Tree	1373.01	37.05	21.12	0.34
Random Forest	1003.91	31.68	19.64	0.48
AdaBoost	1085.56	32.95	19.39	0.45
XGBoost	1010.2	31.8	19.6	0.48

In this study we tried 8 different method to predict prices in Berlin. For each method we did our best to optimize it as much as possible in order to get the best result. By comparing the scores gotten from the Decision Tree method and the one from the Random Forest we see the power of the ensemble learning. Also the Boosting method both gave us really good scores. Ultimately the best result in terms of MSE and Adjusted-R^2 are yielded by Random Forest and XGBoost, while in terms of MAE it's AdaBoost.
However we are still far from a perfect prediction: we manage to get the error below 20 euros on average but there's still room for progress. We have several of hypotheses to explain this:

1. The prices don't present a large enough variance for an estimator to have an accurate prediction.
2. The features themselves aren't descriptive enough of host behaviour in the decision of price. Deeper information on his/her background (financial, social, professional) could give better insight on the way the price would be set.