Airbnb Price Prediction in New York City

Machine Learning Project GitHub Repository

Gabriel Maccione Cyrille Malongo Julien Maronne

December 23, 2025

Abstract

This report presents a machine learning project aiming at predicting the nightly price of Airbnb listings in New York City using the public `AB_NYC_2019` dataset. After an exploratory analysis and a careful preprocessing of the data, several regression algorithms are compared, including XGBoost, Random Forest, Ridge Regression, Gradient Boosting and LightGBM. The target variable is modeled in the logarithmic scale to reduce skewness and stabilize the variance. The final model is selected based on standard regression metrics and its ability to generalize. We also discuss the business implications of the results and the main limitations of the study.

Contents

1 Business scope

Airbnb hosts face the recurring problem of choosing an appropriate nightly price for their listings. An overestimated price may lead to low occupancy, while an underestimated price reduces potential revenue and may distort competition. A data-driven pricing tool can therefore:

- help hosts set an *optimal* price given the attributes of their listing;
- improve transparency and consistency of prices across similar listings;
- support strategic decisions such as investing in specific neighbourhoods or amenities.

The scope of this project is restricted to New York City listings in 2019. The objective is to build a predictive model that, given a listing's characteristics (location, room type, reviews, availability, etc.), outputs an estimate of its nightly price. From a business perspective, such a model can be integrated into a recommendation system, a dashboard for hosts, or used as a benchmarking tool by Airbnb itself.

The work is implemented in Python and organized around a main Jupyter notebook Projet_MachineLearning_Vfinal(2).ipynb and a companion script `Projet_Machine_Learning.py`. The dataset `AB_NYC_2019.csv` contains real Airbnb listings, and several open-source libraries are used, such as `pandas`, `scikit-learn`, `xgboost`, `lightgbm`, `geopandas` and `contextily`.

2 Problem formulation and methods

We formulate the task as a supervised regression problem. For each listing i, we observe a feature vector $\mathbf{x}_i$ (describing the listing) and a scalar response y_i (its observed nightly price). Our goal is to learn a function f such that

$$\hat{y}_i = f(\mathbf{x}_i)$$

is as close as possible to the true price y_i for unseen listings.

In practice, we model the transformed target

$$z_i = \log(1 + y_i)$$

instead of y_i directly, and we learn a function g such that $\hat{z}_i = g(\mathbf{x}_i)$. The final price prediction is then obtained as $\hat{y}_i = \exp(\hat{z}_i) - 1$.

2.1 Algorithm description

The general pipeline implemented in the notebook and script is the following:

1. **Data loading and exploration.** The dataset `AB_NYC_2019.csv` is loaded using `pandas`. Basic inspections (`head`, `info`, `describe`, histogram of prices, scatter plots) are used to understand the structure and quality of the data.
2. **Feature engineering and cleaning.**
   - Categorical variables (e.g., `neighbourhood_group`, `room_type`, `host_name`) are first filled with a "Missing" category when necessary and then encoded using their empirical frequency in the dataset. This corresponds to a simple *frequency encoding*.
   - Listings with non-positive prices (`price` $= 0$) are removed since they are considered outliers or invalid records.
   - Remaining missing values are handled directly inside the machine learning pipeline: numerical features are imputed using the median through a SimpleImputer, while categorical features are imputed with the most frequent category using SimpleImputer before being encoded with a OneHotEncoder.

- A new target variable `price_log` is created as $\log(1+$`price`$)$, and the original `price` column is dropped.
- Several additional features are engineered to capture important business and spatial effects:
  - `distance_center`, defined as the Euclidean distance between each listing and a fixed city centre reference point (Times Square, approximately at latitude 40.7580 and longitude -73.9855), computed from the latitude and longitude;
  - `neighbourhood_rank`, which maps each `neighbourhood_group` (Bronx, Staten Island, Queens, Brooklyn, Manhattan) to an ordinal score from 1 to 5 reflecting its typical price level;
  - `is_entire_home`, a binary indicator equal to 1 when the `room_type` is "Entire home/apt" and 0 otherwise, isolating the effect of full-home rentals.
- Identifier columns (`host_id`, `id`) are dropped from the final feature set, as they either pose difficulties for the models or are not directly interpretable features.
- High-cardinality text features such as `name` and `neighbourhood` are also dropped to reduce dimensionality.

3. **Modeling.** The main algorithms evaluated are:
   - **XGBoost** (`XGBRegressor`), a gradient boosting model based on decision trees.
   - **Random Forest**, an ensemble of decision trees trained with bootstrap aggregation.
   - **Ridge Regression**, a linear regression with L_2 regularization.
   - **Gradient Boosting Regressor**, the classic gradient boosting implementation from `scikit-learn`.
   - **LightGBM**, a highly efficient gradient boosting framework optimized for speed and performance.

   All models are wrapped into `scikit-learn` `Pipeline`s that include preprocessing steps and the regressor itself.

4. **Evaluation.** The dataset is split into training and validation subsets (80% / 20%). Models are evaluated mainly using the Root Mean Squared Error (RMSE) and the coefficient of determination R^2 on the log-price scale. Additional diagnostic plots such as residual histograms and predicted vs. true values are used to visually assess the fit.

2.2 Limitations

The chosen methodology is subject to several limitations:

- **Single train/validation split.** The evaluation relies on a single 80/20 split, which may not be fully representative of the true generalization performance. Cross-validation would provide more stable estimates.
- **Frequency encoding of categorical variables.** Mapping string categories to their global frequency is simple and compact, but it may discard useful information and introduce subtle target leakage if not carefully controlled. A more standard approach would be one-hot encoding or target encoding with proper regularization.
- **Limited hyperparameter tuning.** While most models were initially evaluated using basic hyperparameters (e.g., `n_estimators = 100`), we later introduced a more advanced configuration for XGBoost, incorporating parameters such as `max_depth = 10`, `n_estimators = 800`, `learning_rate = 0.01`, `min_child_weight = 3`,

`subsample = 0.6`, `colsample_bytree = 0.8`, `gamma = 0.1`, `reg_alpha = 0.1` and `reg_lambda = 2`. Although this represents a significant improvement in tuning for XGBoost, a systematic and automated approach (grid search, random search, or Bayesian optimization) could further optimize performance across all models.

- **Temporal and market dynamics.** The dataset is static (2019 snapshot) and restricted to New York City. Seasonal effects, regulatory changes and market dynamics are not modeled, which limits the temporal validity and geographical generality of the conclusions.
- **Unobserved factors.** Important determinants of price such as photos, exact quality of the apartment, host response time or demand shocks are not included. The model therefore captures only part of the true price formation process.

3 Methodology

3.1 Data description and exploration

The `AB_NYC_2019` dataset contains close to 49 000 Airbnb listings in New York City and 16 variables, including:

- unique identifiers for listing and host (`id`, `host_id`);
- textual information (`name`, `host_name`);
- location variables (`neighbourhood_group`, `neighbourhood`, `latitude`, `longitude`);
- characteristics of the listing (`room_type`, `minimum_nights`, `availability_365`);
- social proof variables (`number_of_reviews`, `last_review`, `reviews_per_month`);
- the target variable `price` (nightly price in USD).

Exploratory analysis in the notebook includes:

- descriptive statistics of all numerical features and counts of categorical levels;
- a histogram of prices showing a highly right-skewed distribution with many low-priced listings and a long tail of expensive ones;
- a scatter plot of index vs. price highlighting outliers;
- boxplots of price by `neighbourhood_group` and by `room_type`;
- a map of New York City where listings are colored by price or log-price using `geopandas` and `contextily`.

Geographical visualization confirms that:

- Manhattan and parts of Brooklyn concentrate the most expensive listings;
- outer boroughs tend to present more moderate prices;
- price patterns are spatially clustered, justifying the importance of location features.

3.1.1 Missing values

The exploratory phase reveals several missing values:

- `last_review` and `reviews_per_month` contain many missing entries because not all listings have received reviews;
- some textual fields such as `name` or `host_name` may also contain a small number of missing values.

The handling of missing values is twofold:

- At an early stage, categorical variables are filled with a "Missing" category before being frequency-encoded.
- Later, in the modeling pipelines, numerical variables are imputed with the median and (potential) categorical variables with the most frequent category using `SimpleImputer`.

For some experiments, rows with remaining missing values after basic preprocessing are simply dropped to simplify the pipeline.

3.1.2 Imbalanced data

The data presents several forms of imbalance:

- **Price distribution.** Prices are strongly right-skewed: most listings are relatively affordable, with a few extremely expensive ones. This is problematic for models that assume homoscedasticity and symmetric errors.
- **Neighbourhood groups.** The proportion of listings is not homogeneous across neighbourhood groups: Manhattan and Brooklyn dominate the dataset, while Staten Island has comparatively few listings.
- **Room types.** Entire homes/apartments and private rooms represent the majority of listings; shared rooms and other categories are rare.

The main mitigation strategy is the log transformation of the target, which reduces the impact of very expensive listings on the loss function. No specific rebalancing is applied to neighbourhood or room type because the task is regression and the models used (tree-based ensembles) are robust to such distributional imbalances.

3.1.3 Outliers

The analysis reveals several types of outliers:

- Listings with `price` equal to 0, which are considered invalid or placeholder entries.
- Extremely high prices, which may correspond to luxury properties, multi-night pricing mistakes, or data errors.

The main cleaning step is to remove listings with price ≤ 0. The remaining extreme prices are not explicitly removed, but their effect is mitigated through the log transformation of the target. Tree-based models such as Random Forest and gradient boosting are also relatively robust to a small number of extreme observations.

3.1.4 Engineered features

Beyond simple cleaning and encoding, we introduce several engineered features that make domain knowledge more explicit:

- **Distance to city centre (`distance_center`).** Using the geographic coordinates, we compute an approximate Euclidean distance between each listing and a fixed reference point in Midtown Manhattan (around Times Square, latitude 40.7580, longitude -73.9855). This variable summarizes how central or peripheral a listing is, which is strongly related to price.
- **Neighbourhood rank (`neighbourhood_rank`).** Instead of using raw neighbourhood group labels, we map each `neighbourhood_group` to an ordinal score from 1 (cheapest, e.g. Bronx) to 5 (most expensive, e.g. Manhattan). This ranking reflects the typical price gradient across boroughs and provides the models with a simple monotonic signal.
- **Entire home indicator (`is_entire_home`).** We add a binary indicator that equals 1 if the listing is an "Entire home/apt" and 0 otherwise. This separates full-home rentals from private or shared rooms, which have very different price levels and demand profiles.

These engineered features help the models better capture spatial and structural effects (centrality, borough hierarchy, type of accommodation) without relying solely on raw coordinates or categorical labels.

3.1.5 Correlation analysis

To better understand the relationship between the engineered features and the target variable, we compute the Pearson correlation between all numerical variables and `price_log`. The resulting correlation plot is shown in Figure 1.

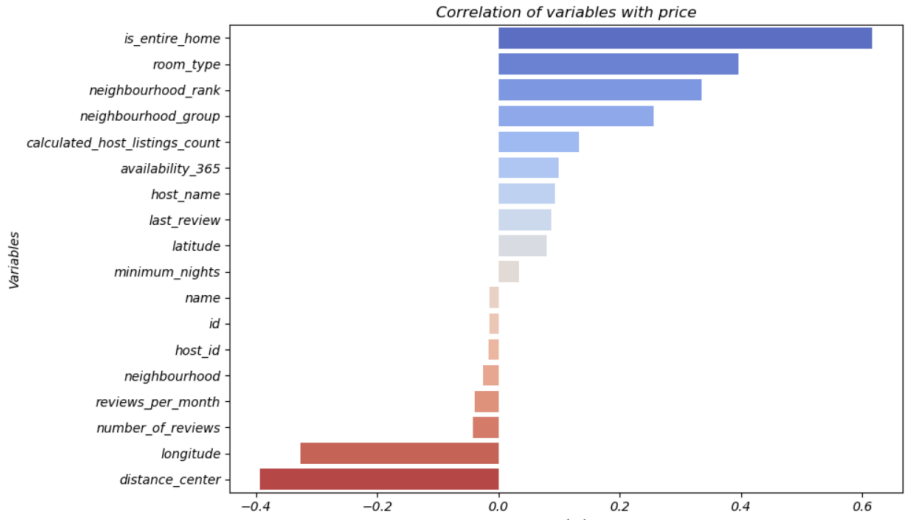

Figure 1: Correlation of numerical and engineered features with the target price.

The analysis highlights several important trends:

- **is_entire_home** shows the strongest positive correlation with price. Listings offering an entire apartment or house are significantly more expensive than rooms.
- **room_type** and the engineered **neighbourhood_rank** also show strong positive correlations, confirming that both the type of accommodation and borough hierarchy are major determinants of price.

- **distance_center** presents a clear negative correlation: as listings move farther away from Midtown Manhattan, their prices tend to decrease.
- Variables such as **reviews_per_month**, **number_of_reviews**, and **neighbourhood** show very weak correlations with price, suggesting limited predictive potential.
- **id** and **host_id** appear in the lower part of the ranking with near–zero correlation, confirming that these identifiers carry no meaningful information about pricing. They are therefore removed from the modeling pipeline.

This correlation study validates the relevance of several engineered variables (`distance_center`, `neighbourhood_rank`, `is_entire_home`) and supports the decision to exclude features with no predictive value.

3.2 Data splitting for train/test

After preprocessing, the cleaned dataset contains fewer rows than the original (only listings with strictly positive prices and complete information are kept). We split the data into:

- a training set (80% of the listings), used to fit the models;
- a validation set (20%), used to estimate performance and to compare models.

The split is random but controlled by a fixed `random_state` to ensure reproducibility. Formally, if $\mathcal{D}$ denotes the full dataset, it is partitioned into:

$$\mathcal{D}_{\text{train}}, \mathcal{D}_{\text{valid}} \subset \mathcal{D}, \quad \mathcal{D}_{\text{train}} \cap \mathcal{D}_{\text{valid}} = \emptyset, \quad |\mathcal{D}_{\text{train}}| \approx 0.8|\mathcal{D}|, \quad |\mathcal{D}_{\text{valid}}| \approx 0.2|\mathcal{D}|.$$

This simple hold-out strategy avoids information leakage between training and evaluation, at the cost of some variance in the performance estimates.

3.3 Algorithm implementation and hyperparameters

For each model, a dedicated `scikit-learn` pipeline is defined. A typical example is:

```
numeric_features = [...]
categorical_features = [...]

numeric_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='median'))
])

categorical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='most_frequent')),
    ('encoder', OneHotEncoder(handle_unknown='ignore'))
])

preprocessor = ColumnTransformer(
    transformers=[
        ('num', numeric_transformer, numeric_features),
        ('cat', categorical_transformer, categorical_features)
    ])

model = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('regressor', XGBRegressor(n_estimators=100, random_state=42))
])
```

In practice, because categorical variables are already frequency-encoded at an earlier stage, the list `categorical_features` is often empty and only numerical preprocessing is effectively applied.

The main hyperparameters chosen for the baseline models are:

- **XGBoost:** number of estimators $n_{\text{estimators}} = 100$, default learning rate and tree depth, `random_state = 42`.
- **Random Forest:** $n_{\text{estimators}} = 100$, `random_state = 42`.
- **Ridge Regression:** regularization parameter $\alpha = 1.0$.
- **Gradient Boosting Regressor:** $n_{\text{estimators}} = 100$, `random_state = 42`.
- **LightGBM:** $n_{\text{estimators}} = 100$, `random_state = 42`.

3.3.1 Final XGBoost training pipeline

For the final XGBoost experiment, we implemented a more structured pipeline that closely follows the steps of the Python code.

Train/validation split. Starting from the engineered dataset, we define the target as `price_log` and the feature matrix as all remaining columns:

$$y = \texttt{price\_log}, \quad X = \texttt{df} \setminus \{\texttt{price\_log}\}.$$

We then perform an 80/20 split:

$$(X_{\text{train}}, X_{\text{valid}}, y_{\text{train}}, y_{\text{valid}}) = \texttt{train\_test\_split}(X, y, \texttt{test\_size} = 0.2, \texttt{random\_state} = 42).$$

Automatic feature type detection. We automatically detect numerical and categorical features using the data types:

- *Numeric features*: all columns in X_{train} with a numeric data type.
- *Categorical features*: all columns in X_{train} with a string data type.

Preprocessing. For the tuned XGBoost pipeline, we focus on the numeric subset and apply a simple, robust preprocessing:

- numerical features are imputed with the median using `SimpleImputer(strategy='median')`;
- a `ColumnTransformer` applies this numeric transformer to all numeric features, with `remainder='drop'`, so that only the cleaned numeric variables are passed to the regressor.

Formally, the preprocessing block is:

```
numeric_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='median'))
])

preprocessor = ColumnTransformer(
    transformers=[
        ('num', numeric_transformer, numeric_features),
    ],
    remainder='drop'
)
```

Tuned XGBoost regressor. The final regressor is an `XGBRegressor` with parameters obtained from a manual search around reasonable configurations:

```
xgb = XGBRegressor(
    subsample=0.6,
    reg_lambda=2,
    reg_alpha=0.1,
    n_estimators=800,
    min_child_weight=3,
    max_depth=10,
    learning_rate=0.01,
    gamma=0.1,
    colsample_bytree=0.8,
    objective='reg:squarederror',
    tree_method='hist',
    random_state=42
)
```

Full pipeline and training. The full training pipeline is then defined as:

```
pipeline = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('regressor', xgb)
])

pipeline.fit(X_train, y_train)
```

After training, we predict on the validation set:

```
y_valid_pred = pipeline.predict(X_valid)
```

and export the predictions to a CSV file:

```
pd.DataFrame({'price_log': y_valid_pred}).to_csv(
    'predictions_xgb_targetenc.csv', index=False
)
```

This allows us to keep a trace of the tuned XGBoost predictions for further analysis and comparison.

4 Results

4.1 Metrics

To evaluate model performance, we use mainly the Root Mean Squared Error (RMSE) and the coefficient of determination R^2 on the log-price scale.

Given a set of n observations with true targets z_i and predictions $\hat{z}_i$, the RMSE is defined as:

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(z_i - \hat{z}_i)^2}.$$

RMSE is expressed in the same units as the target (here, in terms of $\log(1 + \text{price})$) and penalizes large errors more strongly.

The R^2 score is defined as:

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(z_i - \hat{z}_i)^2}{\sum_{i=1}^{n}(z_i - \bar{z})^2},$$

where $\bar{z}$ is the mean of the true targets. An R^2 close to 1 indicates that the model explains most of the variance in the data, while an R^2 close to 0 means that the model does not perform better than predicting the mean. Negative values indicate that the model is worse than the naive mean predictor.

In addition to these scalar metrics, the notebook reports:

- histograms of prediction errors $z_i - \hat{z}_i$ (residuals);
- scatter plots of $\hat{z}_i$ vs. z_i with a 45-degree reference line.

The residual distribution is centered around zero, with approximately 99% of errors between -1 and 1 in the log-price scale, meaning that most predictions are within a multiplicative factor of roughly e of the true price.

4.2 Final XGBoost performance

For the tuned XGBoost pipeline described above, we obtain the following quantitative results on the log-price scale:

- a Root Mean Squared Error on the training set of

  $$\mathrm{RMSE}_{\mathrm{train}} \approx 0.430,$$

  which indicates that, on average, the prediction error in $\log(1 + \text{price})$ remains moderate;
- a validation R^2 score of

  $$R^2_{\mathrm{valid}} \approx 0.615,$$

  showing that the model explains about 61% of the variance of the log-price on unseen data.

On the original price scale, an RMSE of 0.430 in the log domain roughly corresponds to a typical multiplicative error of $\exp(0.43) \approx 1.54$, meaning that predicted prices are often within a factor of 1.5 of the true prices. Given the high noise and heterogeneity of Airbnb prices, this level of accuracy is consistent with other models tested in the project.

4.3 Overfitting / underfitting / imbalance

The models exhibit different degrees of bias and variance, but their training performances are relatively close:

- On the training set, LightGBM, Random Forest and XGBoost reach very similar scores, with R^2 values around 0.59 and RMSE between 0.438 and 0.440. This indicates that all three non-linear ensemble models are able to capture a comparable amount of structure in the data.
- Gradient Boosting performs slightly worse, with a train R^2 of about 0.57 and a higher RMSE (around 0.45), which suggests a bit more bias but still reasonable capacity to fit the training data.
- Ridge Regression clearly underfits the problem: its train R^2 remains below 0.50 and its RMSE is significantly larger (around 0.49), which is expected for a purely linear model on a highly non-linear task.

- The tuned XGBoost configuration improves the trade-off between bias and variance: its train R^2 is close to that of LightGBM and Random Forest, and the validation R^2 around 0.615 shows that this capacity effectively transfers to unseen data rather than only memorizing the training set.
- Residual plots show that most predictions align with the reference line, but some points are scattered far away, corresponding to outliers or rare configurations (for example, extremely high prices or atypical locations). These observations are more difficult to model and contribute to heavier tails in the residual distribution.
- Because the distribution of neighbourhood groups and room types is imbalanced, models may fit better the majority patterns (e.g., Manhattan entire apartments) than the minority ones (e.g., shared rooms in Staten Island). However, given the regression setting and the relatively large dataset, this imbalance is not critical.

Overall, the best-performing models (LightGBM, Random Forest and XGBoost) strike a reasonable compromise between underfitting and overfitting: they capture substantial structure in the data without perfectly memorizing the training set, and the tuned XGBoost shows good generalization on the validation set.

4.4 Evaluation, comparison with the baseline

As a reference, a naive baseline model can be defined as predicting the mean of `price_log` for all listings. This baseline has $R^2 = 0$ by construction and an RMSE equal to the standard deviation of the log-price.

The machine learning models clearly outperform this baseline. On the training set, their performances are summarized as follows:

Model	Train R^2 (approx.)	Train RMSE (approx.)
LightGBM	0.599	0.439
Random Forest	0.598	0.440
XGBoost	0.597	0.440
Gradient Boosting	0.574	0.452
Ridge Regression	0.497	0.491

From this table, we observe that:

- LightGBM obtains the best training performance, with the highest R^2 and the lowest RMSE, although the margin compared to Random Forest and XGBoost is very small.
- Random Forest and XGBoost are essentially tied with LightGBM on the training set, confirming that all three tree-based ensembles have very similar capacity on this problem.
- Gradient Boosting is slightly behind the top three methods, but still clearly better than Ridge Regression.
- Ridge Regression is the weakest model, with substantially lower R^2 and higher RMSE, which is consistent with its linear structure on a non-linear task.

When taking into account the validation performance, the tuned XGBoost model stands out with a validation R^2 of about 0.615 and a train RMSE around 0.43 in the log domain, showing that it not only fits the training data well but also generalizes correctly to unseen listings. LightGBM and Random Forest remain very strong alternatives, but XGBoost offers a particularly good compromise between flexibility, performance and stability on the held-out data.

5 Encountered issues

During the development of the notebook, we faced a few important issues that influenced the final design of the pipeline and our understanding of the results.

5.1 Evaluating on the training set instead of the test set

In an intermediate version of the notebook, when comparing several models in a loop, we mistakenly evaluated the performance on the *training* data instead of on the held-out validation set. Concretely, after fitting each pipeline on `X_train`, we computed predictions on `X_train` again and then calculated R^2 and RMSE using $(y\_train, \hat{y}\_\text{train})$. This produced very optimistic scores that did not reflect the true generalization ability of the models.

This error is a classic pitfall in machine learning: a model can always perform very well on the data it has seen during training, especially for flexible models such as Random Forest or gradient boosting, but this says little about its performance on new, unseen listings.

The issue was detected when we started analysing residuals and scatter plots on the validation set and noticed a clear gap between the apparent performance (based on the training metrics) and the behaviour on unseen data. We then corrected the code so that, for the model comparison, metrics are computed on `X_valid` and `y_valid`. After this correction, the scores became lower but more realistic, and we could fairly compare the models and select the best candidates.

5.2 Need for richer feature engineering

At the beginning of the project, we trained models on a relatively raw version of the dataset, keeping many columns that were noisy or weakly informative (such as identifiers or high-cardinality text fields) and without transforming the target variable. The resulting performance was disappointing, with high errors and unstable behaviour across splits.

To address this, we had to significantly enrich and refine the feature engineering:

- removing listings with zero price and dropping the raw `price` in favour of its logarithm `price_log`;
- dropping `id` and `host_id`, which either carried little predictive signal for our models or introduced unnecessary complexity;
- frequency-encoding the main categorical variables and carefully handling missing values;
- adding informative engineered variables such as `distance_center`, `neighbourhood_rank` and `is_entire_home` to better capture spatial and structural effects;
- analysing correlations with `price_log` to keep only the most relevant variables.

Only after this more thorough feature engineering did the models start to achieve acceptable R^2 and RMSE on the validation set. This experience highlighted the importance of data preprocessing: improving the quality and representation of the features was at least as impactful as changing the machine learning algorithm itself.

5.3 Attempt to switch dataset: AB_US_2023 and degraded performance

During the project, we attempted to improve our results by training the models on a much larger dataset: the `AB_US_2023.csv` file from the Kaggle "US Airbnb Open Data" collection. This dataset aggregates listings from multiple cities and states in the United States and contains far more entries than the original `AB_NYC_2019.csv`. In principle, a larger dataset should allow the model to learn richer patterns and generalize better.

However, our experiments showed that using the US-wide 2023 dataset actually **degraded performance**, even after adapting the preprocessing steps and feature engineering pipeline. Several factors explain this surprising outcome:

- **Lack of geographical homogeneity.** The 2023 dataset merges listings from dramatically different markets (New York, Los Angeles, Denver, rural areas, small towns, etc.). Price dynamics, neighbourhood effects, demand patterns, and host behaviours vary enormously across regions. A single model struggles to capture such heterogeneous structures, resulting in higher variance and poorer generalization.
- **Different feature distributions and missingness patterns.** The US-wide dataset introduces missing values in variables that were complete in the NYC dataset, and the distribution of core features (e.g., room types, review frequencies, availability) differs substantially between cities. Our pipeline, originally optimized for NYC, was not directly transferable.
- **Temporal shift and post-pandemic market changes.** The 2023 dataset reflects a very different Airbnb market compared to 2019. Pricing trends, remote work patterns, regulatory environments, and travel demand were profoundly affected by the pandemic. As a result, correlations and behaviours learned from the NYC dataset do not generalize to the 2023 US market.
- **Loss of locality-based meaning for engineered features.** Our engineered features such as `distance_center` and `neighbourhood_rank` were specifically designed for New York City. When applied at a national scale, these features lose relevance because the concept of a "city centre" or "borough price hierarchy" does not translate across states.
- **Increased noise despite more data.** More data does not always imply higher quality. The additional listings introduce substantial noise and variability unrelated to the core price formation mechanisms we aim to model. This ultimately harms the model more than it helps it.

In all experiments, the models trained on `AB_US_2023.csv` produced lower validation scores and higher residual variance than the original NYC-only dataset. Because of this, we decided to revert to the cleaner, more homogeneous, and conceptually consistent `AB_NYC_2019.csv` dataset for the final analysis.

This experience highlights a crucial lesson in machine learning: **more data is not always better**. Data relevance, consistency, and structure often matter more than sheer volume.

6 Discussion and conclusion

This project demonstrates how classical machine learning techniques can be used to predict Airbnb prices from a real-world dataset. By combining data cleaning, exploratory analysis, feature engineering and several regression algorithms, we obtain a model that explains a substantial proportion of the variance in log-prices and produces reasonably accurate predictions for unseen listings.

From a business perspective, the analysis confirms that:

- **Location** is a key driver of price: listings in Manhattan and central areas of Brooklyn command higher prices.
- **Room type** matters: entire homes and apartments are significantly more expensive than private or shared rooms.

- **Activity indicators** such as the number of reviews and availability also provide useful signals about price.
- **Engineered features** like distance to the city centre, borough rank and an explicit "entire home" indicator help capture spatial and structural effects in a way that is both predictive and interpretable.

The tuned XGBoost and Random Forest models can serve as decision-support tools for hosts and platform managers. Given the characteristics of a listing, they return a recommended price that is consistent with market patterns observed in 2019 New York data.

However, several limitations should be stressed:

- The model is trained on a single city and year; its predictions may not generalize to other cities or to later periods, especially after regulatory changes or market shocks.
- Some important determinants of price (quality of pictures, description text, host behavior, seasonality) are not included in the dataset.
- The frequency encoding of categorical variables, while simple, might not be the most expressive representation, and it may cause a loss of interpretability.

Code and resources. All experiments presented in this report are implemented in the files `Projet_MachineLearning_Vfinal(2).ipynb` and `Projet_Machine_Learning.py`, using the dependencies listed in `requirements.txt`.