Desafio Cientista de Dados INDICIUM - Davi Ribeiro

Price Prediction

In this dataset we have the variables: *'id', 'nome', 'host_id', 'host_name', 'bairro_group', 'bairro', 'latitude', 'longitude', 'room_type', 'price', 'minimo_noites', 'numero_de_reviews', 'ultima_review', 'reviews_por_mes', 'calculado_host_listings_count', 'disponibilidade_365'*, but only some of them can be usefull for predicting the price.

Firstly, eliminating those that cannot help and may even confuse the model, they are:

- ***'id'*** (is used only to represent the user, but as it is a number, can affect the model);
- ***'nome'*** (the name of the place, it doesn't affect the price of the place);
- ***'host_id'*** (similar to the id of the rent ad, is used only to represent the place's owner, but can affect the model);
- ***'host_name'*** (the name of the owner's place doesn't affect the price of the place);
- ***'bairro_group'*** (the '*bairro*' variable is enough to represent the place's location);
- '***latitude***' and '***longitude***' (both are coordinates, but as a number can affect the model, and we already have the '*bairro*' and '*bairro_group*');
- '***ultima_review***' (is only the date of the last review, it doesn't affect the price of the place),
- '***reviews_por_mes***' (how many reviews per month, it doesn't affect the price of the place);
- '***calculado_host_listings_count***' (the number of places that the owner has doens't affect the price of a place).

With this, we have the following variables to predict the price:

- ***'bairro'***;
- ***'room_type'***;
- ***'price'*** (the variable we want to predict);
- ***'minimo_noites'***
- ***'numero_de_reviews'***
- ***'disponibilidade_365'***

Since in this problem we are not predicting categories, we are predicting the price which is a continuous variable, we will use some regression model.

Importing Libraries

```
import warnings
warnings.filterwarnings("ignore")

import joblib
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
from sklearn.metrics import (mean_squared_error, r2_score, mean_absolute_error, root_mean_squared_error)
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
```

```
df = pd.read_csv("teste_indicium_precificacao.csv")
df.head()
```

	id	nome	host_id	host_name	bairro_group	bairro	latitude	longitude	room_type	price	minimo_noites
0	2595	Skylit Midtown Castle	2845	Jennifer	Manhattan	Midtown	40.75362	-73.98377	Entire home/apt	225	1
1	3647	THE VILLAGE OF HARLEM....NEW YORK !	4632	Elisabeth	Manhattan	Harlem	40.80902	-73.94190	Private room	150	3
2	3831	Cozy Entire Floor of Brownstone	4869	LisaRoxanne	Brooklyn	Clinton Hill	40.68514	-73.95976	Entire home/apt	89	1
3	5022	Entire Apt: Spacious Studio/Loft by central park	7192	Laura	Manhattan	East Harlem	40.79851	-73.94399	Entire home/apt	80	10
4	5099	Large Cozy 1 BR Apartment In Midtown East	7322	Chris	Manhattan	Murray Hill	40.74767	-73.97500	Entire home/apt	200	3

```python
# First removing the unnecessary columns
df.drop(
    columns=['id', 'nome', 'host_id', 'host_name', 'bairro_group', 'latitude',
             'longitude', 'ultima_review', 'reviews_por_mes', 'calculado_host_listings_count'],
    inplace=True
    )
```

```python
df.isna().sum()
```

```
bairro               0
room_type            0
price                0
minimo_noites        0
numero_de_reviews    0
disponibilidade_365  0
dtype: int64
```

First of all, we have some categorical variables, so we have to transform them into numeric variables, but we have a difficult situation on our hands. The variable *'bairro'* has 221 different values, but on this problem I think that the small granularity of the *'bairro'* is a big problem talking about dimensionality, , but it will probably be a good resource for predicting the price, bacause if I restrict it to *'bairro_group'*, I will lose a lot of information, for example two *'bairros'* on Brooklyn, but one has more violent crimes than the other, using Brooklyn instead of using the *'bairros'* will make the model miss this information. So, I'm going to use the *'bairro'* as a feature to predict the price. I will use the ***One Hot Encoding*** to transform the categorical variable into numerical variable, this may increase the cost of the model, but I think that it will be worth the pity.

As I know that dimensionality is a problem, I will change the type of columns to make better use of memory.

```python
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 48894 entries, 0 to 48893
Data columns (total 6 columns):
 #   Column               Non-Null Count  Dtype 
---  ------               --------------  ----- 
 0   bairro               48894 non-null  object
 1   room_type            48894 non-null  object
 2   price                48894 non-null  int64 
 3   minimo_noites        48894 non-null  int64 
 4   numero_de_reviews    48894 non-null  int64 
 5   disponibilidade_365  48894 non-null  int64 
dtypes: int64(4), object(2)
memory usage: 2.2+ MB
```

```python
for col in df.columns[2:]:
    print(f'{col} - Min: {df[col].min()} - Max: {df[col].max()}')
```

```
price - Min: 0 - Max: 10000
minimo_noites - Min: 1 - Max: 1250
numero_de_reviews - Min: 0 - Max: 629
disponibilidade_365 - Min: 0 - Max: 365
```

With this minimum and maximum information for each column, we can use the correct type to save memory space:

- uint8 is from 0 to 255;
- uint16 is from 0 to 65535;

So we can use uint8 for the columns that have a maximum of 255 and uint16 for the columns that have a maximum of 65535

```python
# Using price as integer only because the price on this dataset doensn't have float values
df['price'] = df['price'].astype('uint16')
df['minimo_noites'] = df['minimo_noites'].astype('uint16')
df['numero_de_reviews'] = df['numero_de_reviews'].astype('uint16')
df['disponibilidade_365'] = df['disponibilidade_365'].astype('uint16')
```

```python
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 48894 entries, 0 to 48893
Data columns (total 6 columns):
 #   Column              Non-Null Count  Dtype 
---  ------              --------------  ----- 
 0   bairro              48894 non-null  object
 1   room_type           48894 non-null  object
 2   price               48894 non-null  uint16
 3   minimo_noites       48894 non-null  uint16
 4   numero_de_reviews   48894 non-null  uint16
 5   disponibilidade_365  48894 non-null  uint16
dtypes: object(2), uint16(4)
memory usage: 1.1+ MB
```

We can see that only changing types, our memory usage is 50% smaller.

```python
# Applying the One-Hot-Encoding on the columns room_type and bairro
df = pd.concat((df, pd.get_dummies(df['room_type'], dtype=np.uint8)), axis=1).drop(columns=['room_type'])
df = pd.concat((df, pd.get_dummies(df['bairro'], dtype=np.uint8)), axis=1).drop(columns=['bairro']).reset_index(drop=True)
```

```python
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 48894 entries, 0 to 48893
Columns: 228 entries, price to Woodside
dtypes: uint16(4), uint8(224)
memory usage: 10.8 MB
```

Doing One-Hot Encoding, we can see that we went from 6 columns to 228, and memory usage increased almost 10x, but as I said before, this is expected, and I think it will be worth it.

```python
from tabulate import tabulate

# Function to print the metrics for the regression models
def regression_report(y_true, y_pred):
    mae = mean_absolute_error(y_true, y_pred).round(2)
    rmse = root_mean_squared_error(y_true, y_pred).round(2)
    r2 = r2_score(y_true, y_pred).round(2)

    metrics_data = [
        ['Root Mean Squared Error', rmse],
        ['Mean Absolute Error', mae],
        ['R-squared', r2],
    ]

    table = tabulate(metrics_data, headers=['Metric', 'Score'], tablefmt='pretty')

    print("Regression Report:")
    print(table)

    return rmse, mae, r2
```

For metrics, I will use 3 to look at, but only 2 for real evaluation, they are Root Mean Squared(RMSE) and R-Squared(r2). I'm using RMSE because it's a good metric that measures mean squared errors, but when the root is applied the error measure is on the same scale as its variable, so if my RMSE is 50, it means I'm predicting wrong on average. 50+ or 50-. And I like to use R2 because it's a good statistical measure to see if the percentage of actual variation in the data is similar to the prediction, so closer to 1 is better.

```python
# Splitting the data between features and target
X = df.drop(columns=['price']).values
y = df['price'].values
```

```python
# Splitting the data between train and test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

```python
# Let's normalize the data
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

```
# Using joblib to save the scaler
joblib.dump(scaler, 'scaler.pkl')
```

```
['scaler.pkl']
```

To load the scaler is very simple:

`scaler = joblib.load('scaler.pkl')`

Using the models

Gradient Boosting Regressor

```
from sklearn.ensemble import GradientBoostingRegressor

gbr = GradientBoostingRegressor()
gbr.fit(X_train, y_train)
```

```
▾ GradientBoostingRegressor
GradientBoostingRegressor()
```

```
rmse_gbr, mae_gbr, r2_gbr = regression_report(y_test, gbr.predict(X_test))
```

```
Regression Report:
+-------------------------+--------+
|         Metric          | Score  |
+-------------------------+--------+
| Root Mean Squared Error | 213.47 |
|   Mean Absolute Error   | 66.01  |
|        R-squared        |  0.09  |
+-------------------------+--------+
```

Support Vector Regressor

```
from sklearn.svm import SVR

svr = SVR()
svr.fit(X_train, y_train)
```

```
▾ SVR
SVR()
```

```
rmse_svr, mae_svr, r2_svr = regression_report(y_test, svr.predict(X_test))
```

```
Regression Report:
+-------------------------+--------+
|         Metric          | Score  |
+-------------------------+--------+
| Root Mean Squared Error | 216.01 |
|   Mean Absolute Error   | 58.82  |
|        R-squared        |  0.07  |
+-------------------------+--------+
```

Random Forest Regressor

```
from sklearn.ensemble import RandomForestRegressor

random_forest = RandomForestRegressor()
random_forest.fit(X_train, y_train)
```

```
▾ RandomForestRegressor
RandomForestRegressor()
```

```
rmse_random_forest, mae_random_forest, r2_random_forest = regression_report(y_test, random_forest.predict(X_test))
```

```
Regression Report:
+-------------------------+--------+
|         Metric          | Score  |
+-------------------------+--------+
| Root Mean Squared Error | 233.96 |
|   Mean Absolute Error   | 68.89  |
|        R-squared        |  -0.1  |
+-------------------------+--------+
```

XGBoost Regressor

```python
from xgboost import XGBRegressor

xgb = XGBRegressor(n_estimators=1000, max_depth=7, random_state=42, learning_rate=0.001, n_jobs = -1)
xgb.fit(X_train, y_train)
```

```
XGBRegressor
XGBRegressor(base_score=None, booster=None, callbacks=None,
             colsample_bylevel=None, colsample_bynode=None,
             colsample_bytree=None, device=None, early_stopping_rounds=None,
             enable_categorical=False, eval_metric=None, feature_types=None,
             gamma=None, grow_policy=None, importance_type=None,
             interaction_constraints=None, learning_rate=0.001, max_bin=None,
             max_cat_threshold=None, max_cat_to_onehot=None,
             max_delta_step=None, max_depth=7, max_leaves=None,
             min_child_weight=None, missing=nan, monotone_constraints=None,
             multi_strategy=None, n_estimators=1000, n_jobs=-1,
```

```python
rmse_xgb, mae_xgb, r2_xgb = regression_report(y_test, xgb.predict(X_test))
```

```
Regression Report:
+-------------------------+--------------------+
|         Metric          |       Score        |
+-------------------------+--------------------+
| Root Mean Squared Error | 214.82000732421875 |
|   Mean Absolute Error   | 71.51000213623047  |
|        R-squared        |        0.08        |
+-------------------------+--------------------+
```

We can see that these regressors are not good, with R-Squared close to zero and RMSE greater than 200, let's try to make a Neural Network to predict the values.

Neural Network

```python
import tensorflow as tf
from tensorflow.keras.layers import Dense, Dropout, Input
from tensorflow.keras.callbacks import Callback
from tensorflow.keras.models import Sequential
from tqdm import tqdm
```

```
WARNING:tensorflow:From s:\PYTHON\Virtual_Environments\indicium_cd\Lib\site-packages\keras\src\losses.py:2976: The name tf.losses.sparse_softmax
```

```python
# Callback function to save the best model based best RMSE

class SaveBestModel(Callback):
    def __init__(self,filepath_weights):
        super(SaveBestModel, self).__init__()
        self.filepath_weights = filepath_weights
        self.best_metric = 1e3

    def on_epoch_end(self, epoch, logs=None):
        val_metric = logs['val_root_mean_squared_error']
        if val_metric < self.best_metric:
            self.best_metric = val_metric
            self.model.save_weights(f"{self.filepath_weights}_{(self.best_metric) :.2f}.h5", overwrite=True)
            print(f"\nModel saved with validation accuracy: {val_metric:.4f}")
```

```
# Creating the Neural Network

model = Sequential()
model.add(Input(shape=(X_train.shape[1],)))
model.add(Dense(64, activation="relu",))
model.add(Dropout(0.5))
model.add(Dense(32, activation="relu"))
model.add(Dropout(0.5))
model.add(Dense(1, activation="relu"))

opt = tf.optimizers.Adam(learning_rate=5e-3)
loss_fn = tf.keras.losses.Huber() # Less sensitive to outliers than MSE
rmse_metric = tf.keras.metrics.RootMeanSquaredError()
r2_metric = tf.keras.metrics.MeanSquaredError()

model.compile(optimizer=opt, loss=loss_fn, metrics=[rmse_metric, r2_metric])

model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 64)                14592

 dropout (Dropout)           (None, 64)                0

 dense_1 (Dense)             (None, 32)                2080

 dropout_1 (Dropout)         (None, 32)                0

 dense_2 (Dense)             (None, 1)                 33

=================================================================
Total params: 16705 (65.25 KB)
Trainable params: 16705 (65.25 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Simple NN, but I believe that for this problem, is enough.

```
epochs = 1000
hist = model.fit(X_train, y_train, epochs=epochs,
                 validation_data=(X_test, y_test),
                 callbacks=[SaveBestModel(filepath_weights='model_weights')],
                 batch_size=128,
                 verbose=1)
```

```
Epoch 1/1000
WARNING:tensorflow:From s:\PYTHON\Virtual_Environments\indicium_cd\Lib\site-packages\keras\src\utils\tf_utils.py:492: The name tf.ragged.RaggedT

261/306 [========================>.....] - ETA: 0s - loss: 82.7742 - root_mean_squared_error: 248.0725 - mean_squared_error: 61539.9883
Model saved with validation accuracy: 213.3608
306/306 [==============================] - 1s 1ms/step - loss: 80.9530 - root_mean_squared_error: 244.9126 - mean_squared_error: 59982.1836 - va
Epoch 2/1000
306/306 [==============================] - 0s 1ms/step - loss: 69.4610 - root_mean_squared_error: 236.9053 - mean_squared_error: 56124.1133 - va
Epoch 3/1000
306/306 [==============================] - 0s 1ms/step - loss: 68.9154 - root_mean_squared_error: 237.0186 - mean_squared_error: 56177.8125 - va
Epoch 4/1000
306/306 [==============================] - 0s 1ms/step - loss: 68.2624 - root_mean_squared_error: 236.7554 - mean_squared_error: 56053.1211 - va
Epoch 5/1000
306/306 [==============================] - 0s 1ms/step - loss: 67.7142 - root_mean_squared_error: 236.3932 - mean_squared_error: 55881.7539 - va
Epoch 6/1000
274/306 [=========================>....] - ETA: 0s - loss: 66.7456 - root_mean_squared_error: 234.8468 - mean_squared_error: 55153.0430
Model saved with validation accuracy: 213.2673
306/306 [==============================] - 0s 1ms/step - loss: 67.1534 - root_mean_squared_error: 235.9739 - mean_squared_error: 55683.6875 - va
Epoch 7/1000
306/306 [==============================] - 0s 1ms/step - loss: 66.8301 - root_mean_squared_error: 235.9576 - mean_squared_error: 55676.0117 - va
Epoch 8/1000
281/306 [==========================>...] - ETA: 0s - loss: 66.7011 - root_mean_squared_error: 238.1345 - mean_squared_error: 56708.0508
Model saved with validation accuracy: 212.7265
306/306 [==============================] - 0s 1ms/step - loss: 66.5454 - root_mean_squared_error: 236.0232 - mean_squared_error: 55706.9531 - va
Epoch 9/1000
306/306 [==============================] - 0s 1ms/step - loss: 66.3786 - root_mean_squared_error: 235.9190 - mean_squared_error: 55657.7891 - va
Epoch 10/1000
261/306 [========================>.....] - ETA: 0s - loss: 66.0705 - root_mean_squared_error: 234.7184 - mean_squared_error: 55092.7070
Model saved with validation accuracy: 212.2134
306/306 [==============================] - 0s 1ms/step - loss: 66.2581 - root_mean_squared_error: 235.7557 - mean_squared_error: 55580.7461 - va
Epoch 11/1000
306/306 [==============================] - 0s 1ms/step - loss: 66.0092 - root_mean_squared_error: 235.6487 - mean_squared_error: 55530.2969 - va
Epoch 12/1000
306/306 [==============================] - 0s 1ms/step - loss: 65.6502 - root_mean_squared_error: 234.9762 - mean_squared_error: 55213.8359 - va
Epoch 13/1000
306/306 [==============================] - 0s 1ms/step - loss: 65.4379 - root_mean_squared_error: 235.1376 - mean_squared_error: 55289.6836 - va
Epoch 14/1000
306/306 [==============================] - 0s 1ms/step - loss: 65.6528 - root_mean_squared_error: 235.6071 - mean_squared_error: 55510.6875 - va
Epoch 15/1000
278/306 [==========================>...] - ETA: 0s - loss: 65.7564 - root_mean_squared_error: 241.1168 - mean_squared_error: 58137.3203
Model saved with validation accuracy: 212.0788
306/306 [==============================] - 0s 1ms/step - loss: 65.4421 - root_mean_squared_error: 235.1805 - mean_squared_error: 55309.8555 - va
Epoch 16/1000
306/306 [==============================] - 0s 1ms/step - loss: 65.3171 - root_mean_squared_error: 235.0222 - mean_squared_error: 55235.4414 - va
Epoch 17/1000
306/306 [==============================] - 0s 1ms/step - loss: 65.0541 - root_mean_squared_error: 234.8270 - mean_squared_error: 55143.7031 - va
Epoch 18/1000
265/306 [========================>.....] - ETA: 0s - loss: 65.2084 - root_mean_squared_error: 239.7555 - mean_squared_error: 57482.6953
Model saved with validation accuracy: 211.9338
306/306 [==============================] - 0s 1ms/step - loss: 65.0172 - root_mean_squared_error: 234.6612 - mean_squared_error: 55065.8906 - va
Epoch 19/1000
306/306 [==============================] - 0s 1ms/step - loss: 64.9040 - root_mean_squared_error: 234.8868 - mean_squared_error: 55171.8008 - va
Epoch 20/1000
306/306 [==============================] - 0s 1ms/step - loss: 64.8456 - root_mean_squared_error: 234.6083 - mean_squared_error: 55041.0391 - va
Epoch 21/1000
306/306 [==============================] - 0s 1ms/step - loss: 64.8751 - root_mean_squared_error: 235.1275 - mean_squared_error: 55284.9180 - va
Epoch 22/1000
306/306 [==============================] - 0s 1ms/step - loss: 64.7039 - root_mean_squared_error: 234.4812 - mean_squared_error: 54981.4375 - va
Epoch 23/1000
306/306 [==============================] - 0s 1ms/step - loss: 64.2566 - root_mean_squared_error: 234.4812 - mean_squared_error: 54981.4375 - va
Epoch 24/1000
306/306 [==============================] - 0s 1ms/step - loss: 64.5426 - root_mean_squared_error: 234.4741 - mean_squared_error: 54978.0938 - va
Epoch 25/1000
306/306 [==============================] - 0s 1ms/step - loss: 64.4320 - root_mean_squared_error: 234.5621 - mean_squared_error: 55019.4023 - va
Epoch 26/1000
306/306 [==============================] - 0s 1ms/step - loss: 64.2639 - root_mean_squared_error: 234.4264 - mean_squared_error: 54955.7305 - va
Epoch 27/1000
306/306 [==============================] - 0s 1ms/step - loss: 64.0564 - root_mean_squared_error: 234.2740 - mean_squared_error: 54884.2930 - va
Epoch 28/1000
306/306 [==============================] - 0s 1ms/step - loss: 64.1334 - root_mean_squared_error: 234.3664 - mean_squared_error: 54927.5938 - va
Epoch 29/1000
306/306 [==============================] - 0s 1ms/step - loss: 64.2229 - root_mean_squared_error: 234.4568 - mean_squared_error: 54970.0117 - va
Epoch 30/1000
306/306 [==============================] - 0s 1ms/step - loss: 63.9338 - root_mean_squared_error: 233.7717 - mean_squared_error: 54649.2305 - va
Epoch 31/1000
264/306 [========================>.....] - ETA: 0s - loss: 63.6199 - root_mean_squared_error: 237.9627 - mean_squared_error: 56626.2617
Model saved with validation accuracy: 211.5236
306/306 [==============================] - 0s 1ms/step - loss: 63.6784 - root_mean_squared_error: 233.6579 - mean_squared_error: 54596.0156 - va
Epoch 32/1000
306/306 [==============================] - 0s 1ms/step - loss: 64.0733 - root_mean_squared_error: 234.0521 - mean_squared_error: 54780.3789 - va
Epoch 33/1000
306/306 [==============================] - 0s 1ms/step - loss: 63.9163 - root_mean_squared_error: 234.3564 - mean_squared_error: 54922.9141 - va
```

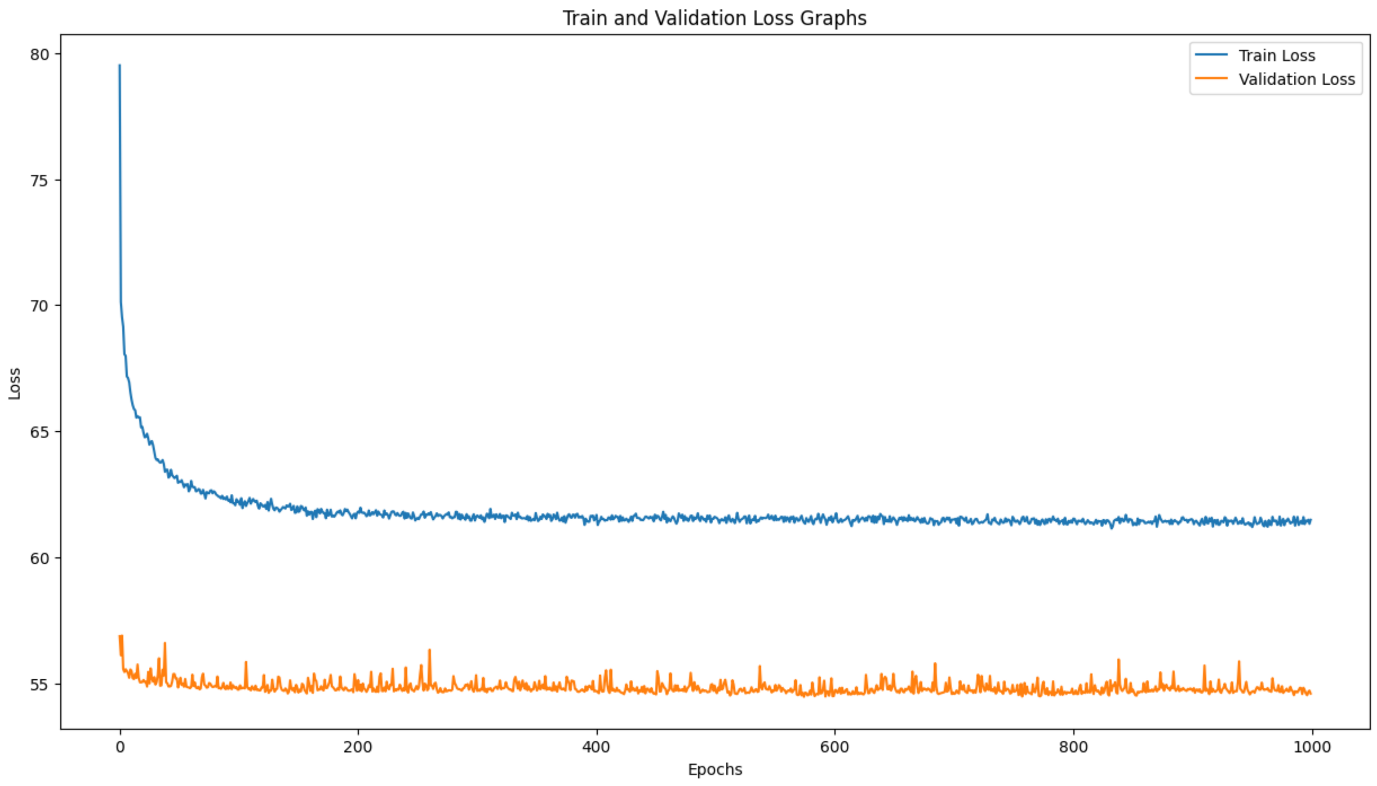
```
Epoch 968/1000
306/306 [==============================] - 0s 1ms/step - loss: 61.3830 - root_mean_squared_error: 232.1928 - mean_squared_error: 53913.4922 - va
Epoch 969/1000
306/306 [==============================] - 0s 1ms/step - loss: 61.3196 - root_mean_squared_error: 232.0930 - mean_squared_error: 53867.1562 - va
Epoch 970/1000
306/306 [==============================] - 0s 1ms/step - loss: 61.5443 - root_mean_squared_error: 232.6420 - mean_squared_error: 54122.3125 - va
Epoch 971/1000
306/306 [==============================] - 0s 1ms/step - loss: 61.4478 - root_mean_squared_error: 231.9478 - mean_squared_error: 53799.7969 - va
Epoch 972/1000
306/306 [==============================] - 0s 1ms/step - loss: 61.4547 - root_mean_squared_error: 232.3271 - mean_squared_error: 53975.8633 - va
Epoch 973/1000
306/306 [==============================] - 0s 1ms/step - loss: 61.4112 - root_mean_squared_error: 231.8355 - mean_squared_error: 53747.6797 - va
Epoch 974/1000
306/306 [==============================] - 0s 1ms/step - loss: 61.4258 - root_mean_squared_error: 232.0911 - mean_squared_error: 53866.2891 - va
Epoch 975/1000
306/306 [==============================] - 0s 1ms/step - loss: 61.4559 - root_mean_squared_error: 232.4944 - mean_squared_error: 54053.6641 - va
Epoch 976/1000
306/306 [==============================] - 0s 1ms/step - loss: 61.5234 - root_mean_squared_error: 231.9709 - mean_squared_error: 53810.5078 - va
Epoch 977/1000
306/306 [==============================] - 0s 1ms/step - loss: 61.3631 - root_mean_squared_error: 232.3037 - mean_squared_error: 53965.0039 - va
Epoch 978/1000
306/306 [==============================] - 0s 1ms/step - loss: 61.4831 - root_mean_squared_error: 232.3877 - mean_squared_error: 54004.0547 - va
Epoch 979/1000
306/306 [==============================] - 0s 1ms/step - loss: 61.5080 - root_mean_squared_error: 231.9613 - mean_squared_error: 53806.0586 - va
Epoch 980/1000
306/306 [==============================] - 0s 1ms/step - loss: 61.4794 - root_mean_squared_error: 231.9798 - mean_squared_error: 53814.6406 - va
Epoch 981/1000
306/306 [==============================] - 0s 1ms/step - loss: 61.5302 - root_mean_squared_error: 232.1350 - mean_squared_error: 53886.6758 - va
Epoch 982/1000
306/306 [==============================] - 0s 1ms/step - loss: 61.4600 - root_mean_squared_error: 232.3764 - mean_squared_error: 53998.8047 - va
Epoch 983/1000
306/306 [==============================] - 0s 1ms/step - loss: 61.4316 - root_mean_squared_error: 232.5308 - mean_squared_error: 54070.5859 - va
Epoch 984/1000
306/306 [==============================] - 0s 1ms/step - loss: 61.5367 - root_mean_squared_error: 232.0372 - mean_squared_error: 53841.2656 - va
Epoch 985/1000
306/306 [==============================] - 0s 1ms/step - loss: 61.2440 - root_mean_squared_error: 232.3119 - mean_squared_error: 53968.8281 - va
Epoch 986/1000
306/306 [==============================] - 0s 1ms/step - loss: 61.3059 - root_mean_squared_error: 232.2468 - mean_squared_error: 53938.5625 - va
Epoch 987/1000
306/306 [==============================] - 0s 1ms/step - loss: 61.4952 - root_mean_squared_error: 232.4841 - mean_squared_error: 54048.8398 - va
Epoch 988/1000
306/306 [==============================] - 0s 1ms/step - loss: 61.5488 - root_mean_squared_error: 232.0297 - mean_squared_error: 53837.7695 - va
Epoch 989/1000
306/306 [==============================] - 0s 1ms/step - loss: 61.3086 - root_mean_squared_error: 231.8817 - mean_squared_error: 53769.1172 - va
Epoch 990/1000
306/306 [==============================] - 0s 1ms/step - loss: 61.4018 - root_mean_squared_error: 232.1692 - mean_squared_error: 53902.5234 - va
Epoch 991/1000
306/306 [==============================] - 0s 1ms/step - loss: 61.3792 - root_mean_squared_error: 232.4464 - mean_squared_error: 54031.3516 - va
Epoch 992/1000
306/306 [==============================] - 0s 1ms/step - loss: 61.5942 - root_mean_squared_error: 232.1385 - mean_squared_error: 53888.2617 - va
Epoch 993/1000
306/306 [==============================] - 0s 1ms/step - loss: 61.4817 - root_mean_squared_error: 232.4555 - mean_squared_error: 54035.5703 - va
Epoch 994/1000
306/306 [==============================] - 0s 1ms/step - loss: 61.5916 - root_mean_squared_error: 232.4007 - mean_squared_error: 54010.0625 - va
Epoch 995/1000
306/306 [==============================] - 0s 1ms/step - loss: 61.3106 - root_mean_squared_error: 231.7976 - mean_squared_error: 53730.1484 - va
Epoch 996/1000
306/306 [==============================] - 0s 1ms/step - loss: 61.4176 - root_mean_squared_error: 232.1120 - mean_squared_error: 53875.9805 - va
Epoch 997/1000
306/306 [==============================] - 0s 1ms/step - loss: 61.7034 - root_mean_squared_error: 232.2990 - mean_squared_error: 53962.8398 - va
Epoch 998/1000
306/306 [==============================] - 0s 1ms/step - loss: 61.5183 - root_mean_squared_error: 232.1273 - mean_squared_error: 53883.1055 - va
Epoch 999/1000
306/306 [==============================] - 0s 1ms/step - loss: 61.5721 - root_mean_squared_error: 232.5710 - mean_squared_error: 54089.2539 - va
Epoch 1000/1000
306/306 [==============================] - 0s 1ms/step - loss: 61.3418 - root_mean_squared_error: 231.8330 - mean_squared_error: 53746.5586 - va
```

```python
# Save the model on a .pkl file
joblib.dump(model, 'model_neural_network.pkl')
```

```
['model_neural_network.pkl']
```

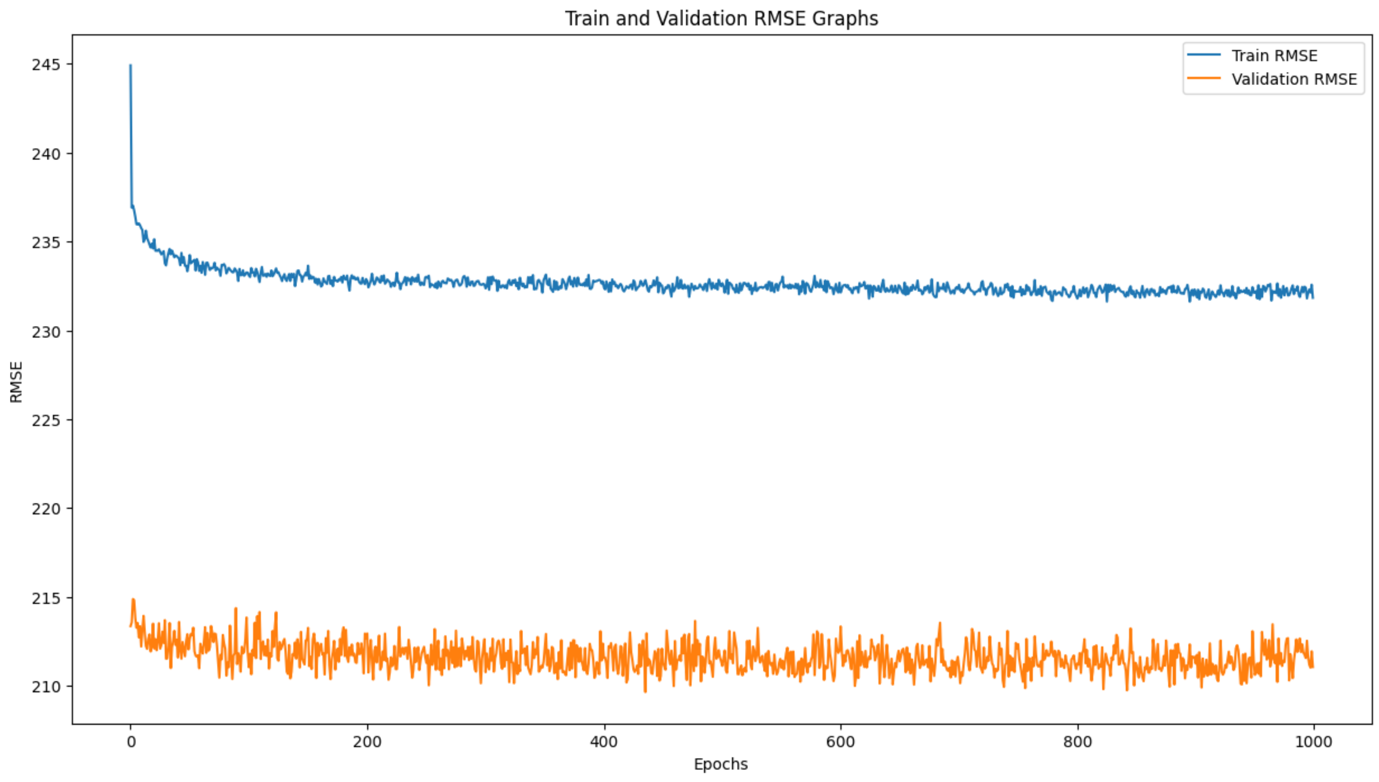
```
import matplotlib.pyplot as plt
plt.figure(figsize=(15, 8))
plt.plot(hist.history['loss'], label='Train Loss')
plt.plot(hist.history['val_loss'], label='Validation Loss')
plt.title('Train and Validation Loss Graphs')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
```

```
<matplotlib.legend.Legend at 0x2a415674ed0>
```



```
plt.figure(figsize=(15, 8))
plt.plot(hist.history['root_mean_squared_error'], label='Train RMSE')
plt.plot(hist.history['val_root_mean_squared_error'], label='Validation RMSE')
plt.title('Train and Validation RMSE Graphs')
plt.xlabel('Epochs')
plt.ylabel('RMSE')
plt.legend()
```

```
<matplotlib.legend.Legend at 0x2702ab239d0>
```



```
rmse_nn, mae_nn, r2_nn = regression_report(y_test, model.predict(X_test, verbose=0))
```

```
Regression Report:
+------------------------+-------------------+
|         Metric         |       Score       |
+------------------------+-------------------+
| Root Mean Squared Error | 208.8699951171875  |
|   Mean Absolute Error   | 60.959999084472656 |
|        R-squared        |        0.13        |
+------------------------+-------------------+
```

Testing the model on this single case

```
test = pd.read_csv('test.csv')
test.drop(
    columns=['id', 'nome', 'host_id', 'host_name', 'bairro_group', 'latitude',
             'longitude', 'ultima_review', 'reviews_por_mes', 'calculado_host_listings_count'],
    inplace=True
    )
test
```

	bairro	room_type	price	minimo_noites	numero_de_reviews	disponibilidade_365
0	Midtown	Entire home/apt	225	1	45	355

```
# These variables are necessary for the One-Hot-Encoding part
unique_room_type = df['room_type'].unique()
unique_bairro = df['bairro'].unique()

test = pd.concat((test, pd.get_dummies(test['room_type'], dtype=np.uint8)), axis=1).drop(columns=['room_type'])
test = pd.concat((test, pd.get_dummies(test['bairro'], dtype=np.uint8)), axis=1).drop(columns=['bairro'])
```

```
# This is for get the columns that are missing
missing_columns = set(np.concatenate((unique_room_type, unique_bairro))) - set(test.columns)
for column in missing_columns:
    test[column] = np.uint8(0)
```

```
# Loading the scaler saved on the beginning of the notebook
scaler_ = joblib.load('models-scalers/scaler_default.pkl')
```

```
X_ = test.drop(columns=['price']).values
X_ = scaler_.transform(X_)
```

```
# Loading the model saved above
model = joblib.load('models-scalers/model_neural_network_1.pkl')
```

Important point, for the model and the scaler to work well, the data must be in the same structure as the training here, which is why all the above treatment was necessary.

```
real = test['price'][0]
predicted = model.predict(X_, verbose=0)[0][0]

print(f"Real: {real:.2f}, Predicted: {predicted:.2f}")
print(f"Difference: {abs(real - predicted):.2f}")
```

```
Real: 225.00, Predicted: 286.93
Difference: 61.93
```

The performance of the model in general is not very good, but on this case, the performance of the model wasn't bad, the price was predicted with an error of 61.93, which is not bad, but it's not good either.

Performance Summary

```
model_summary = pd.DataFrame(columns=['model', 'RMSE', 'R²'])
model_summary.loc[0] = ['Gradient Boosting', rmse_gbr, r2_gbr]
model_summary.loc[1] = ['Support Vector Machine', rmse_svr, r2_svr]
model_summary.loc[2] = ['Random Forest', rmse_random_forest, r2_random_forest]
model_summary.loc[3] = ['XGBoost', rmse_xgb, r2_xgb]
model_summary.loc[4] = ['Neural Network', rmse_nn, r2_nn]
model_summary.sort_values(by='RMSE')
```

	model	RMSE	R²
4	Neural Network	208.869995	0.13
0	Gradient Boosting	213.470000	0.09
3	XGBoost	214.820007	0.08
1	Support Vector Machine	216.010000	0.07
2	Random Forest	233.960000	-0.10

The pros and cons of using a neural network here are:

- **Pros**:
  - Non-linearity (but as you can see I only used non-linear regressors);
  - Flexibility (I used a simple architecture, because that was enough, but I could make a bigger and more complex model);
- **Cons**:
  - High computational cost (Training can be slow and require significant computational resources);
  - If you are not careful, it is very easy to overfit;
  - Sometimes requires a lot of data to work well.

As we can see, the Neural Network is better than the other regressors, and this is expected, because the Neural Network is a very flexible model, and it can learn the data better than the other models but it is still not good enough. I believe that as we saw in the EDA notebook, this data set has a large number of outliers, so the variance of the data is large and this is affecting the accuracy of the models. So, trying to solve this, I made another notebook with exactly the same layout as this one, but removing the outliers.

One important thing to say is that the outliers that we saw in the boxplots are above the maximum fence, which means they were the high-value locations, which means that probably by removing the outliers, I'm removing some luxury rental locations, but I'm sure that by removing them the variance will decrease and the accuracy of the models will be better.