CSCI 4146/6409 - Process of Data Science (Summer 2023)

Assignment 2

Tasneem Hoque
B00841761

Karan Aggarwal
B00912580

1. [2.5] Predictive Modelling

In [1]:
```
import pandas as pd
data = pd.read_csv('AB_US_2023.csv', low_memory=False)
data.drop(['name', 'host_id', 'host_name', 'number_of_reviews_ltm', 'reviews_per_month
           'last_review', 'calculated_host_listings_count', 'latitude',
           'availability_365', 'longitude'], axis=1, inplace=True)
data.isnull().sum() * 100 / len(data)
data.drop(['neighbourhood_group'], axis=1, inplace=True)
city_map = dict(enumerate(data['city'].astype('category').cat.categories))
data['city'] = data['city'].astype('category').cat.codes
data['neighbourhood'] = data['neighbourhood'].astype('category').cat.codes
data['room_type'] = data['room_type'].astype('category').cat.codes
data.drop(data[data['price'] > 20000].index, inplace = True)
```

In [2]:
```
data.head()
```

Out[2]:

	id	neighbourhood	room_type	price	minimum_nights	number_of_reviews	city
0	958	1375	0	202	2	383	20
1	5858	151	0	235	30	111	20
2	8142	572	2	56	32	9	20
3	8339	1375	0	575	9	28	20
4	8739	821	2	110	1	770	20

a. Select an appropriate predictive model for your problem. Explain your choice. [0.5]

For our Airbnb price prediction task, we have carefully considered the suitability of various predictive models and concluded that XGBoost is the most appropriate choice. We have excluded linear regression due to the highly complex and fluctuating nature of Airbnb pricing, which is influenced by factors such as location, seasons, and amenities. Categorizing the data is also not feasible since the prices depend on a multitude of features. Additionally, other predictive models lack the capability to handle missing values, which further supports our decision to choose XGBoost or Random Forest.

We have selected XGBoost over Random Forest based on its superior predictive performance and speed. XGBoost excels at capturing complex relationships and non-linear patterns in the data, making it well-suited for our task. Furthermore, its sequential tree-building approach allows for dependencies among features to be considered. For instance, prices can vary not only across cities but also within cities based on room types, and XGBoost's ability to model these interactions is advantageous.

In conclusion, after careful consideration of the specific requirements and characteristics of our Airbnb price prediction task, we have determined that XGBoost is the most suitable predictive model, offering both excellent predictive performance and efficient computation.

b. Partition the dataset into a training set and a test set. Describe the method you used for this. [0.5]

To partition the dataset into a training set and test set, I used the popular train_test_split function from the sklearn library.

This function allowed me to split the data into four parts: x_train as the training set, x_test as a subset of the training set for testing, y_train representing the remaining data for prediction, and y_test used to evaluate the accuracy of the predictions.

I divided the data in a 70:30 ratio, with 70% of the data allocated for training and the remaining 30% for testing. To ensure randomness in the splitting process, I included a random state of 40. This allowed for consistent random sampling even when the code is executed multiple times.

By using the train_test_split function, I successfully partitioned the dataset into distinct training and test sets, enabling accurate model training and evaluation.

In [3]:
```
from sklearn.model_selection import train_test_split
```

In [4]:
```
X=data.drop('price',axis=1)
y = data.price
```

In [5]:
```
x_train, x_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=
```

In [6]:
```
x_train.head()
```

Out[6]:

	id	neighbourhood	room_type	minimum_nights	number_of_reviews	city
20513	41384631	870	0	30	34	12
85593	38850758	806	0	2	8	13
179134	11852407	1052	0	30	433	16
46172	39030147	1105	2	1	5	10
50097	32919453	1287	0	31	13	10

In [7]:
```
x_test.head()
```

Out[7]:

	id	neighbourhood	room_type	minimum_nights	number_of_reviews	city
141809	21655498	382	0	1	36	11
184383	17483323	562	0	1	411	24
16930	756069325935741528	1319	0	29	1	9
139808	26201541	936	0	5	0	19
193543	742045082673815994	1031	0	2	8	25

In [8]:
```
y_train.head()
```

Out[8]:
```
20513     227
85593     299
179134     45
46172     429
50097     121
Name: price, dtype: int64
```

In [9]:
```
y_test.head()
```

Out[9]:
```
141809    155
184383    132
16930     219
139808    750
193543    171
Name: price, dtype: int64
```

c. Train the model on the training set. Discuss the parameters you used and why. [0.5]

In the Airbnb dataset, our objective is to predict the continuous prices of Airbnb houses. To accomplish this, we require a regression class from the XGBoost library. Hence, we utilize the XGBRegressor class provided by XGBoost.

By using the XGBRegressor class, we can train a regression model that learns the relationship between the input features and the corresponding continuous price values. This enables us to make accurate predictions on data.

Parameters Used:

In the XGBRegressor, I utilized several important parameters including n_estimators, max_depth, learning_rate, and subsample. By manually setting these parameters, I aimed to prevent the model from overfitting or underfitting.

The parameter n_estimators determines the number of trees to be built in the model. By carefully selecting this value, I ensured that the model achieves an appropriate level of complexity without overfitting the training data or sacrificing its ability to capture important patterns.

Another crucial parameter, max_depth, defines the maximum depth of each decision tree in the ensemble. By limiting the depth, I controlled the level of complexity and prevented the trees from becoming excessively deep. This helped avoid overfitting and encouraged the model to generalize well to unseen data.

The learning_rate parameter played a significant role in the training process. By setting a lower learning rate, I aimed to achieve better generalization and enhance the model's ability to capture underlying patterns in the data. It effectively controlled the step size at each boosting iteration, allowing the model to gradually learn from the data without making drastic adjustments.

The subsample parameter determined the fraction of samples used for training each individual tree. By keeping this value less than 1, I introduced randomness into the training process, which can help prevent overfitting. This ensured that the model learned from a diverse set of samples and improved its ability to generalize to new instances.

In [10]:
```
import xgboost
from sklearn.metrics import mean_squared_error, mean_absolute_error
```

In [11]:
```
regressor = xgboost.XGBRegressor(n_estimators=200, max_depth=6, learning_rate = 0.20,
```

In [12]:
```
regressor.fit(x_train, y_train, eval_set=[(x_test, y_test)], verbose=False)
```

Out[12]:
```
XGBRegressor(base_score=None, booster=None, callbacks=None,
             colsample_bylevel=None, colsample_bynode=None,
             colsample_bytree=None, early_stopping_rounds=None,
             enable_categorical=False, eval_metric=['rmse', 'mae'],
             feature_types=None, gamma=None, gpu_id=None, grow_policy=None,
             importance_type=None, interaction_constraints=None,
             learning_rate=0.2, max_bin=None, max_cat_threshold=None,
             max_cat_to_onehot=None, max_delta_step=None, max_depth=6,
             max_leaves=None, min_child_weight=None, missing=nan,
             monotone_constraints=None, n_estimators=200, n_jobs=None,
             num_parallel_tree=None, predictor=None, random_state=None, ...)
```

In [13]:
```
y_pred = regressor.predict(x_test)
```

In [14]:
```
pd.Series(y_pred).head()
```

Out[14]:
```
0    319.132538
1     94.744141
2    174.690323
3    637.360718
4    140.110977
dtype: float32
```

In [15]:
```
y_test.head()
```

Out[15]:
```
141809    155
184383    132
16930     219
139808    750
193543    171
Name: price, dtype: int64
```

d. Evaluate the model's performance on the test set. Discuss the evaluation metrics used and the results. [0.5]

To Evaluate the model's performance on the test set, We will be using certain evaluations metrics In our case we will be using Root Mean Squared Error and Mean Absolute Error.

Root Mean Square Error defines the magnitude of error made by the model. Lower the value better the prediction model is.

Mean Absolute Error is the absolute difference between predicted and actual values. Similarly lower it is better our model are.

In [16]:
```
eval_results = regressor.evals_result()
rmse = eval_results['validation_0']['rmse']
mae = eval_results['validation_0']['mae']
```

In [17]:
```
rmse[-1]
```

Out[17]:
```
473.0411374920516
```

In [18]:
```
mae[-1]
```

Out[18]:
```
148.95670522276103
```

The value of Root mean Square Error we got is 473.04

The value of Mean Absolute Error we got is 148.95

e. Interpret the model. Explain what it reveals about the prediction subject and its domainconcepts. [0.5]

Our XGBoost model establishes various relationships between the features and learns from them. It examines the connections between factors such as city, room types, neighborhood, and reviews, and utilizes these relationships to predict prices.

Certain domain concepts have a significant impact on prices. For example, urban cities generally have higher prices compared to rural areas, and the specific neighborhood

within a city can drastically affect prices.

To prevent overfitting, we incorporated different parameters into our model, including estimators, learning rate, and max depth. These parameters help ensure that the model generalizes well to new data and avoids being too closely fitted to the training set.

To assess the model's performance, we employed root mean square error (RMSE) and mean absolute error (MAE) as evaluation metrics. These metrics allow us to measure the accuracy of the model's predictions and assess its effectiveness in capturing the underlying patterns in the data.

In conclusion, the XGBoost model establishes relationships between various subject areas and domain concepts, and its interpretation helps validate the model's relevance and reliability in predicting Airbnb prices.

2. [1.5] Model Tuning

a. Explain how you can tune the model. What parameters can you adjust? [0.5]

Theere are lots of parameters for this library as can be seen from their documentation: https://xgboost.readthedocs.io/en/stable/python/python_api.html

However, the parameters that we applied were n_estimators, max_depth, learning_rate, sub_sample, and the eval_metric.

The parameters we can tune to improve the model:

n_estimators: Increasing the number of estimators can improve the model's ability to capture complex relationships in the data. However, there is a trade-off as a larger number of estimators can increase training time.

max_depth: Increasing the max_depth value can potentially lead to a more complex model that captures intricate patterns in the data. However, setting it too high can result in overfitting, where the model learns noise or irrelevant details in the training data.

leraning_rate: A lower learning rate allows for more cautious learning, potentially improving the model's generalization. However, setting it too low can increase the number of iterations needed to converge.

subsample: By setting it to a value less than 1.0, will introduce stochasticity and reduce overfitting. A smaller subsample value can improve the model's ability to generalize by training on different subsets of the data in each iteration. However, setting it too low may result in underfitting.

eval_metric: 'RMSE' (root mean squared error) and 'MAE' (mean absolute error) are commonly used for regression tasks. Adding more evaluation metrics, such as 'R-squared'

or 'mean squared logarithmic error,' can provide a more comprehensive assessment of the model's performance.

b. Tune the model. Explain the method you used and why. [0.5]

In [19]:
```
regressor = xgboost.XGBRegressor(
    n_estimators=185,
    max_depth=6,
    learning_rate = 0.20,
    subsample=0.8,
    reg_alpha = 0.7,
    reg_lambda = 0.8,
    num_parallel_tree = 10,
    eval_metric=['rmse', 'mae'])

regressor.fit(x_train, y_train, eval_set=[(x_test, y_test)], verbose=False)
y_pred = regressor.predict(x_test)
```

The method used to tune the model includes mostly trial and error.

Tuning any existing variables causes a trade-off between the RMSE and MAE evaluation methods. This trade-off is caused because of overfitting and underfitting, so for example if the number of max_depth is increased from 6 to 10, RMSE increased from 473.04 to 494.32 but RMAE decreases from 148.95 to 147.34, a similar patten is seen with the learning_rate and subsample.

However, some minor improvements could be made by tuning existing parameters and introducing some new parameters:

1) RMSE decreased from 473.04 to 472.63 and MAE decreases from 148.95 to 148.65 when n_estimators is reduced to 185, any other values (increased or decreased) cause greater variance in the results.

2) RMSE decreased from 472.63 to 468.05 and MAE decreased from 148.65 to 148.62 with the added parameter reg_alpha set to 0.7 and reg_lambda set to 0.8.

3) RMSE decreased from 468.05 to 467.92 and MAE decreased from 148.62 to 145.66 with the added parameter num_parallel_tree set to 10.

c. Compare the performance of the tuned model with the initial model. Explain any changes. [0.5]

In [21]:
```
new_eval_results = regressor.evals_result()
new_rmse = new_eval_results['validation_0']['rmse']
new_mae = new_eval_results['validation_0']['mae']
```

In [22]:
```
rmse[-1]
```

Out[22]:
```
473.0411374920516
```

In [23]:
```
new_rmse[-1]
```

Out[23]: 467.92190130183184

In [24]:
```
mae[-1]
```

Out[24]: 148.95670522276103

In [25]:
```
new_mae[1]
```

Out[25]: 167.81247317469973

The model was not significantly improved, as seen by the final results RMSE decreased from 473.04 to 467.92 and MAE decreased from 148.95 to 145.66 by changing one existing parameter introducting some new parameters.

1) In this particular case, having n_estimators at 200 was causing some amount of overfitting, by decreasing the value to 185, the model was a bit more generalized which yeilded lower loss values.

2) reg_alpha encourages the model to reduce the impact of less important features by adding a penalty proportional to the absolute value of the coefficients. A higher value of reg_alpha will increase the regularization strength and push more coefficients towards zero, effectively performing feature selection and reducing model complexity.reg_lambda adds a penalty proportional to the square of the coefficients, encouraging them to be small but not necessarily zero. It helps to prevent collinearity and stabilize the model by shrinking the impact of all features simultaneously. Increasing reg_lambda value strengthens the regularization and reduces model complexity.

3) Increasing the value of num_parallel_tree adds more trees to the ensemble, which can enhance the diversity of the model. Ensemble diversity helps to capture different patterns and improve the model's ability to generalize to unseen data. However, larger values significantly improve training time.

3. [0.5] Model Deployment

a. What are the potential issues that might arise during deployment. Moreover, how would you address such deployment issues? [0.5]

Environment Compatibility: The deployment environment may have different versions of libraries or dependencies compared to the development environment. To address this, virtual environments, containerization tools like Docker, or package managers to ensure consistency between the training and deployment environments can be used.

Data Drift: The model's performance can degrade over time if the distribution of the input data changes. Regular monitoring and retraining on fresh data can help address data drift. Implementing monitoring systems and data quality checks can help detect shifts in the data distribution and trigger model retraining or adaptation.

Monitoring and Performance Metrics: Monitoring the deployed model's performance and collecting relevant performance metrics is crucial. Implementing logging and monitoring systems to track prediction outcomes, detecting anomalies, and gathering user feedback can help identify issues and improve the model over time.

Error Handling and Robustness: The model should be able to handle unexpected inputs and gracefully handle errors. Implementing proper error handling mechanisms, input validation, and fallback strategies can help improve the robustness and reliability of the deployed model.

Overfitting and Generalization: If the model's weights continue to be updated during deployment, there is a risk of overfitting the model to the specific deployment data. Freezing the weights after training helps prevent overfitting and promotes better generalization to new data.

4. [0.5] Conclusion

a. Summarize your findings and the value of your solution to the business problem. [0.25]

The analysis helps hosts determine the most suitable nightly rates for their listings based on various factors, including neighborhood, room type, minimum nights, and number of reviews. This enables hosts to set competitive prices that attract guests while maximizing revenue.

Hosts who are new to the platform or unfamiliar with the local market can benefit from the analysis of historical booking data. They can gain a better understanding of pricing trends specific to their location, allowing them to make informed pricing decisions and stay competitive.

Providing competitive rates based on the analysis can improve the overall guest experience. Guests are more likely to choose listings that offer reasonable prices, leading to higher guest satisfaction and potential positive reviews.

b. Identify potential areas for further analysis or improvements. [0.25]

Model Improvement: To further evaluate and validate the model's performance, k-fold cross-validation could be used. This technique involves splitting the training data into k equal-sized folds. The model is trained on k-1 folds and validated on the remaining fold. This process is repeated k times, with each fold acting as the validation set once. Average the evaluation metrics across the k iterations to obtain a more robust estimation of the model's performance.

Seasonal Variation: Analyzing the impact of seasonal variations on pricing. Explore how prices fluctuate during peak seasons versus off-peak seasons. This analysis can help hosts identify optimal pricing strategies for different times of the year.

Amenities: Explore the relationship between listing ammenities (e.g. hair dryer, refrigerator, swimming pool, parking, etc.) and pricing. Analyzing how specific features affect pricing can help hosts understand the value of different amenities and optimize their listing descriptions to justify their rates.

Dynamic Pricing Strategies: limplementing dynamic pricing strategies that take into account real-time factors such as local events, holidays, and demand fluctuations. This can involve analyzing external data sources and integrating them into the pricing model to optimize rates dynamically.

References:

1. **https://medium.com/ibm-data-science-experience/markdown-for-jupyter-notebooks-cheatsheet-386c05aeebed**