**Airbnb Analytical Report**

**Introduction and Business Question**

This report poses the business question: what is the biggest determinant of price for listings? This will be answered through an exploratory data analysis of price in relation to other variables, followed by utilisation of the machine learning algorithms Random Forest Regressor and XGBoost Regressor. These will identify features that determine price and thus predict price. Having a model that can predict price can then be used in the app. For example, for each new listing, the model can provide an estimated price for the listing based on its features.

**Exploratory Data Analysis**

**Data Preprocessing**

During the data preprocessing step, null values have been filled and some of the unnecessary columns such as ‘id’, ‘name’, ‘host\_name’, and ‘last\_review’ have been removed as there is no need to analyse these, especially details of the host as this is personal information. Further, for the Machine Learning models, ‘host\_id’ and ‘reviews\_per\_month’ were dropped as they were found not to have feature importance.

We can see from the correlation graph in figure 1, there are no strong correlations between variables other than ‘reviews’ and ‘reviews\_per\_month’ which is to be expected as they are both counting reviews.



Figure 1 - Correlation Graph

**Variation and Covariation**

First, the categorical variables are considered and then their relationship with price and location. For the purposes of this report and based on the results of the machine learning models discussed below, ‘neighbourhood group’, location (‘latitude’ and ‘longitude’ and ‘room\_type’ are the most likely to contain features that impact price and so shall be examined below.

**Neighbourhood Group and Price**

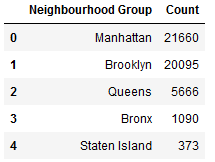


Figure 2 – Neighbourhood Count charts

Manhattan has the most listings, followed closely by Brooklyn and then there is a large drop for the other neighbourhoods (see Figure 2). This distribution could be due to the Queens and Bronx areas being primarily residential and the population of Staten Island being small.

When looking at price, the data has been limited to anything below the 95th percentile to remove extreme values. The histogram in figure 3 displays a positive skewness with most listings around 100 and below. Looking at figure 4, the same observation can be found for each feature, except for Manhattan having a more even distribution of price and thus may have a bigger impact on price.

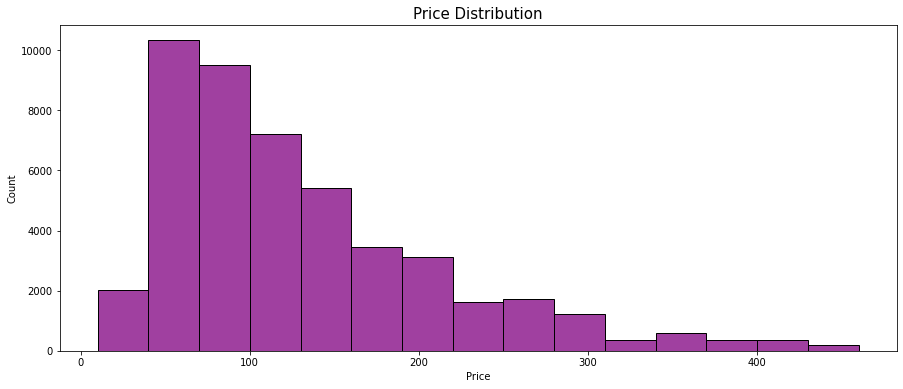


Figure 3 - Price histogram

**Location, Neighbourhood Group, and Price**

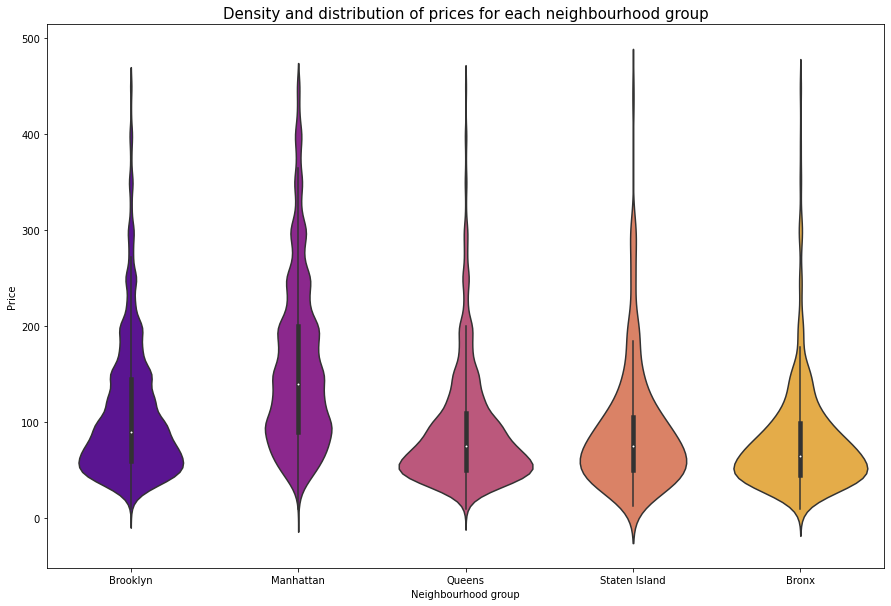


Figure 4 - Violin chart for price and neighbourhood group

When looking at location in *figure 5*, this conforms to neighbourhood groups as they are simply groups of locations. The scatter shows the distribution of price by location, and as corroborated above, there is a concentration of high price listings in Manhattan.



Figure 5 - Scatter Plot showing Location and Neighbourhood Group and Price

**Room Type and Price**

*Figure 6* shows that shared room listings are very low in comparison to entire homes/apartments and private rooms, with entire homes/apartments being the highest.



Figure 6 - Room price count plot

In *figure 7*, it is observable that entire homes/apt has a higher median than the other room types, with its first quartile higher than the third quartile of both shared room and private room. This shows that this feature may be a strong determiner of price.

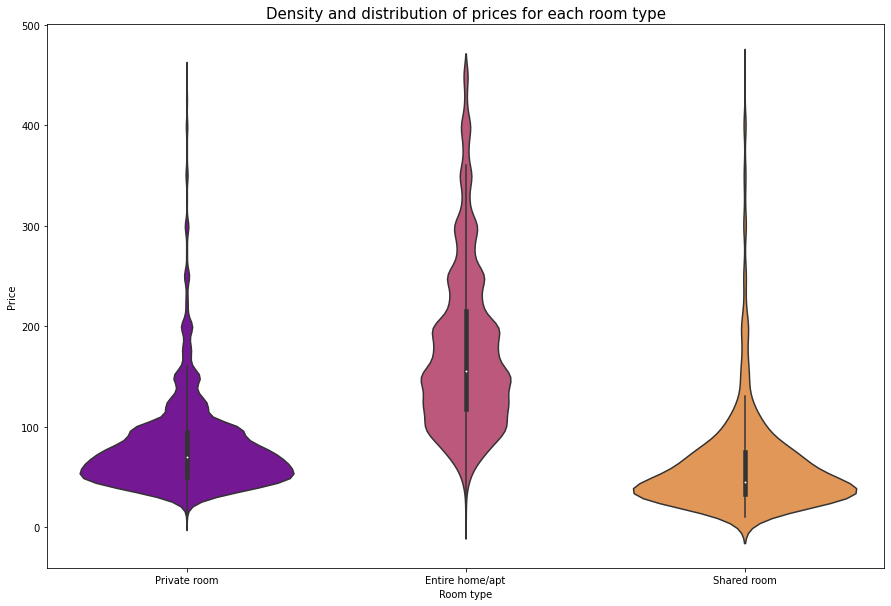


Figure 7 - violin chart of price and room type

**Machine Learning**

**Purpose of the Models**

Of the models that were developed, the ones that showed the highest accuracy were XGBoost Regressor and Random Forest.

The intended purpose of these models is to understand the most important features from the dataset that affect price and then predicting price determined by the other variables in the dataset.

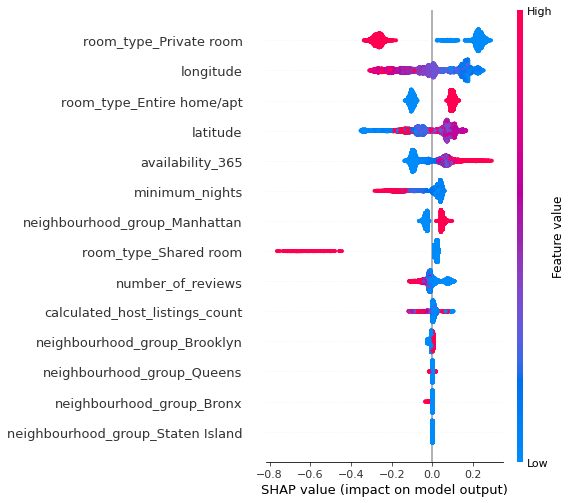
**Evaluating XGBoost Regressor Model**

The initial model has an R2 score of 0.635 on the test data and thus predicts 64% of the relationship of price and the other variables.

The SHAP summary chart in *Figure 8* lists the most important features in descending order determined by the model. Thus, it shows that room type and location are the most important variables in determining listing price since features from these variables are the highest.

The next steps to iterate on this model is to keep the most important features and remove unimportant features. Additional variables are likely to further enhance the model and increase accuracy. For example, a variable for amenities the listing provides, nearby landmarks, and how many people the listing can accommodate.

Figure 8 – XGBoost Regressor SHAP Summary Chart



**Evaluating Random** **Forest Regressor Model**

This model has a slightly better R2 than the XGBoost model with a score of 0.640.

The SHAP summary chart in *figure 9* shows a similar outcome to the XGBoost model. The model ranks features in a different order but the conclusion remains the same: room type and location are the most important features in determining price.

As above, this model would also benefit from the additional variables suggested for the XGBoost model.

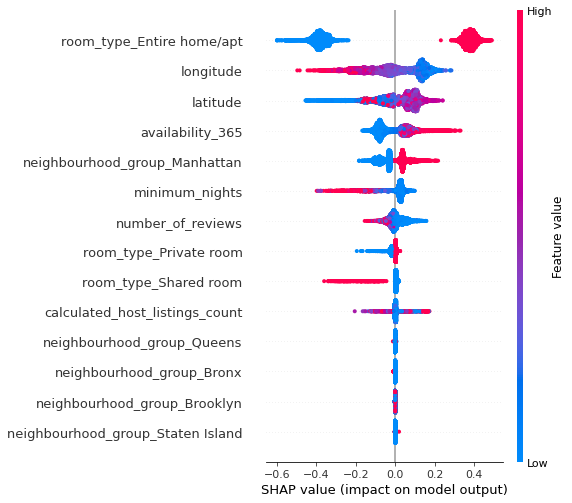


Figure 9 - Random Forest Regressor SHAP Summary Chart

**Conclusion**

This report has analysed and determined the variables that contain the most important features (room type and location) and provided a model to predict the price of listings which should be iterated on with new variables to improve accuracy.  Additionally, by examining the distributions of the most important variables from the exploratory analysis combined with the findings of the models, Airbnb can look to encourage listings from hosts with these features to increase profit.

**Appendices**

**APPENDIX A**: EDA and ML for determining most important features and predicting price.

# # EDA

### Package Import

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import matplotlib.image as mpimg

get\_ipython().run\_line\_magic('matplotlib', 'inline')

import seaborn as sns

import warnings

import xgboost

import scipy.stats as stats

import pylab

warnings.filterwarnings('ignore')

# ## Importing Dataset

#importing dataset

airbnb=pd.read\_csv("AB\_NYC\_2019.csv")

#looking at head

airbnb.head(5)

#looking at number of rise to guage size of dataset - notably some values missing

airbnb.shape

#data types

airbnb.info

# ## Data Preprocessing

#removing duplicates

airbnb.duplicated().sum()

airbnb.drop\_duplicates(inplace=True)

#checking how many null values in dataset for each column

airbnb.isnull().sum()

#drop columns such as id, host name, and last review as there is no need to analyse the names of hosts. The last\_review column is a date. This won't exist if thee number of reviews is 0 and will not be relevant to analysis. The name of the listing is also to be dropped as we will not be conducting NLP.

airbnb.drop(['id','name','host\_name','last\_review'], axis=1, inplace=True)

airbnb.head(5)

#replace null values with 0

airbnb.fillna({'reviews\_per\_month':0}, inplace=True)

airbnb.reviews\_per\_month.isnull().sum()

airbnb.isnull().sum()

airbnb.dropna(how='any',inplace=True)

#remove any listings with price = 0

airbnb = airbnb[airbnb.price>0]

airbnb.describe()

# ## Visualisations for Variation and Covariation

# ### Correlation between variables

#correlation between variables

corr = airbnb.corr(method='kendall')

plt.figure(figsize=(15,8))

sns.heatmap(corr, annot=True)

airbnb.columns

# ### Neighbourhood Group and Price

#looking at unique values for neighbourhood group

airbnb['neighbourhood\_group'].unique()

#looking at distribution of neighbourhood group

df=pd.DataFrame(airbnb['neighbourhood\_group'].value\_counts()).reset\_index().rename(columns={'index': 'Neighbourhood Group','neighbourhood\_group':'Count'})

df

#visualising distribution

plt.figure(figsize=(15,6))

sns.countplot(data=airbnb, x='neighbourhood\_group', palette='plasma')

plt.title('Count of Listings by Neighbourhood Group', fontsize=15)

plt.xlabel('Neighbourhood Group')

plt.ylabel("Count")

#looking at stats for price by neighbourhood group

airbnb\_neighbourprice = airbnb[['host\_id','neighbourhood\_group','price']]

airbnb\_neighbourprice = pd.pivot\_table(airbnb\_neighbourprice,index=['host\_id'],columns='neighbourhood\_group',values="price")

airbnb\_neighbourprice = airbnb\_neighbourprice[['Brooklyn', 'Manhattan', 'Queens', 'Staten Island', 'Bronx']]

airbnb\_neighbourprice.describe()

#Price histogram

plt.figure(figsize=(15,6))

sns.histplot(data=airbnb, x='price', binwidth = 200, color='purple')

plt.title('Price Distribution', fontsize=15)

plt.xlabel('Price')

plt.ylabel("Count")

#finding 95th percentile for each neighbourhood group

airbnb\_neighbourprice.quantile(0.95)

#limiting data to remove outliers above 95th percentile to get rid of outliers

airbnb = airbnb[airbnb.price < 451]

#Looking at price distribution after transformation

plt.figure(figsize=(15,6))

sns.histplot(data=airbnb, x='price', binwidth = 30, color='purple')

plt.title('Price Distribution', fontsize=15)

plt.xlabel('Price')

plt.ylabel("Count")

#looking at covariation of price and neighbourhood group

plt.figure(figsize=(15,10))

sns.violinplot(data=airbnb, x='neighbourhood\_group', y='price', palette='plasma')

plt.title('Density and distribution of prices for each neighbourhood group', fontsize=15)

plt.xlabel('Neighbourhood group')

plt.ylabel("Price")

# ### Room Type and Price

#room type distribution

plt.figure(figsize=(15,6))

sns.countplot(data=airbnb, x='room\_type', palette='plasma')

plt.title('Count of Listings by Room Type', fontsize=15)

plt.xlabel('Room Type')

plt.ylabel("Count")

#covariation room type and price

plt.figure(figsize=(15,10))

sns.violinplot(data=airbnb, x='room\_type', y='price', palette='plasma')

plt.title('Density and distribution of prices for each room type', fontsize=15)

plt.xlabel('Room type')

plt.ylabel("Price")

plt.figure(figsize=(15,10))

ax = sns.boxplot(data=airbnb, x='neighbourhood\_group', y='availability\_365', palette='plasma')

plt.title('Relationship between Neighbourhood Group and Availability', fontsize=15)

plt.xlabel('Neighbourhood Group')

plt.ylabel("Availability")

plt.figure(figsize=(15,10))

sns.scatterplot(data=airbnb, x='latitude', y = 'longitude', palette='plasma', hue = "price", style = "neighbourhood\_group")

plt.title('Price Distribution', fontsize=15)

plt.xlabel('Latitude')

plt.ylabel("Longitude")

# ## Preparing Data for Machine Learning

#function to return plots for the feature

def normality(data,feature):

plt.figure(figsize=(10,5))

plt.subplot(1,2,1)

sns.kdeplot(data[feature])

plt.subplot(1,2,2)

stats.probplot(data[feature],plot=pylab)

plt.show()

#transforming features into normal distribution

airbnb['price\_log'] = np.log(airbnb.price)

normality(airbnb,'price\_log')

# # Machine Learning

y = airbnb['price\_log']

x = airbnb[['neighbourhood\_group','latitude','longitude','room\_type','minimum\_nights','number\_of\_reviews','calculated\_host\_listings\_count','availability\_365']]

x = pd.get\_dummies(x, columns=['room\_type','neighbourhood\_group'])

import sklearn

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import r2\_score

from sklearn.inspection import permutation\_importance

#Split into train and test

x\_train,x\_test,y\_train,y\_test=train\_test\_split(x,y,test\_size=0.25,random\_state=212)

# ## XGBoost

# ### Hyperparameter Tuning

#Hyperparameter tuning to reduce overfitting

import xgboost as xgb

from hyperopt import STATUS\_OK, Trials, fmin, hp, tpe

space={'max\_depth': hp.quniform("max\_depth", 3, 18, 1),

'gamma': hp.uniform ('gamma', 1,9),

'reg\_alpha' : hp.quniform('reg\_alpha', 40,180,1),

'reg\_lambda' : hp.uniform('reg\_lambda', 0,1),

'colsample\_bytree' : hp.uniform('colsample\_bytree', 0.5,1),

'min\_child\_weight' : hp.quniform('min\_child\_weight', 0, 10, 1),

'n\_estimators': 180,

'seed': 0

}

def objective(space):

clf=xgb.XGBRegressor(

n\_estimators =space['n\_estimators'], max\_depth = int(space['max\_depth']), gamma = space['gamma'],

reg\_alpha = int(space['reg\_alpha']),min\_child\_weight=int(space['min\_child\_weight']),

colsample\_bytree=int(space['colsample\_bytree']))

evaluation = [( x\_train, y\_train), ( x\_test, y\_test)]

clf.fit(x\_train, y\_train)

pred = clf.predict(x\_test)

accuracy = r2\_score(y\_test, pred)

print ("SCORE:", accuracy)

return {'loss': -accuracy, 'status': STATUS\_OK }

trials = Trials()

best\_hyperparams = fmin(fn = objective,

space = space,

algo = tpe.suggest,

max\_evals = 100,

trials = trials)

#Get best hyperparameters

print(best\_hyperparams)

# ### XGBoost Model

#XGBoost Regressor model

xgb\_model = xgb.XGBRegressor(colsample\_bytree = 0.5, gamma = 1, max\_depth = 15, min\_child\_weight = 7, reg\_alpha = 42, reg\_lambda = 0.7936246860452796)

xgb\_model.fit(x\_train, y\_train)

#training data score

y\_pred=xgb\_model.predict(x\_train)

r2\_score(y\_train,y\_pred)

#test data score

y\_pred=xgb\_model.predict(x\_test)

r2\_score(y\_test,y\_pred)

# ## Descision Tree

# ### Hyperparameter Tuning

#hyperparameter tuning for regurlarisation to avoid overfitting

from sklearn.tree import DecisionTreeRegressor

from matplotlib import pyplot

train\_scores, test\_scores = list(), list()

# define the tree depths to evaluate

values = [i for i in range(1, 21)]

# evaluate a decision tree for each depth

for i in values:

model = DecisionTreeRegressor(max\_depth=i)

# fit model on the training dataset

model.fit(x\_train, y\_train)

# evaluate on the train dataset

train\_yhat = model.predict(x\_train)

train\_acc = r2\_score(y\_train, train\_yhat)

train\_scores.append(train\_acc)

# evaluate on the test dataset

test\_yhat = model.predict(x\_test)

test\_acc = r2\_score(y\_test, test\_yhat)

test\_scores.append(test\_acc)

# summarize progress

print('>%d, train: %.3f, test: %.3f' % (i, train\_acc, test\_acc))

# plot of train and test scores vs tree depth

pyplot.plot(values, train\_scores, '-o', label='Train')

pyplot.plot(values, test\_scores, '-o', label='Test')

pyplot.legend()

pyplot.show()

# ### Decision Tree Model

#best max\_depth is 8

dt\_model = DecisionTreeRegressor(max\_depth = 8)

dt\_model.fit(x\_train, y\_train)

y\_pred=dt\_model.predict(x\_train)

r2\_score(y\_train,y\_pred)

y\_pred=dt\_model.predict(x\_test)

r2\_score(y\_test,y\_pred)

# ## Random Forest

#hyperparameter tuning for regurlarisation to avoid overfitting

from sklearn.ensemble import RandomForestRegressor

from matplotlib import pyplot

train\_scores, test\_scores = list(), list()

#define max depths to evaluate

values = [i for i in range(1, 11)]

#evaluate a random forest for each depth

for i in values:

model = RandomForestRegressor(max\_depth=i)

#fit model on the training dataset

model.fit(x\_train, y\_train)

#evaluate on the train dataset

train\_yhat = model.predict(x\_train)

train\_acc = r2\_score(y\_train, train\_yhat)

train\_scores.append(train\_acc)

#evaluate on the test dataset

test\_yhat = model.predict(x\_test)

test\_acc = r2\_score(y\_test, test\_yhat)

test\_scores.append(test\_acc)

#summarize progress

print('>%d, train: %.3f, test: %.3f' % (i, train\_acc, test\_acc))

#plot of train and test scores vs tree depth

pyplot.plot(values, train\_scores, '-o', label='Train')

pyplot.plot(values, test\_scores, '-o', label='Test')

pyplot.legend()

pyplot.show()

#max depth of 9 is best

rf\_model = RandomForestRegressor(max\_depth = 9)

rf\_model.fit(x\_train, y\_train)

y\_pred=rf\_model.predict(x\_train)

r2\_score(y\_train,y\_pred)

y\_pred=rf\_model.predict(x\_test)

r2\_score(y\_test,y\_pred)

## Shap

import shap

shap.initjs()

x\_train,x\_test,y\_train,y\_test=train\_test\_split(x,y,test\_size=0.25,random\_state=212)

x\_train\_summary = shap.kmeans(x\_train, 10)

# ## Shap - XGBoost

shap\_xgb\_model = xgb.XGBRegressor(colsample\_bytree = 0.5, gamma = 1, max\_depth = 15, min\_child\_weight = 7, reg\_alpha = 42, reg\_lambda = 0.7936246860452796)

shap\_xgb\_model.fit(x\_train, y\_train)

explainer = shap.TreeExplainer(shap\_xgb\_model)

shap\_values = explainer.shap\_values(x\_test)

shap.summary\_plot(shap\_values, x\_test)

# ## SHAP Decision Tree

shap\_dt\_model = DecisionTreeRegressor(max\_depth = 8)

shap\_dt\_model.fit(x\_train, y\_train)

explainer = shap.TreeExplainer(shap\_dt\_model)

shap\_values = explainer.shap\_values(x\_test)

shap.summary\_plot(shap\_values, x\_test)

# ## SHAP Random Forest

shap\_rf\_model = RandomForestRegressor(max\_depth = 9)

shap\_rf\_model.fit(x\_train, y\_train)

explainer = shap.TreeExplainer(shap\_rf\_model)

shap\_values = explainer.shap\_values(x\_test)

shap.summary\_plot(shap\_values, x\_test)

**APPENDIX B:** Linear Regression for Availability and Price

%matplotlib inline

#imports

from numpy import \*

import matplotlib.pyplot as plt

import pandas as pd

import sklearn

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import r2\_score

from sklearn.inspection import permutation\_importance

df=pd.read\_csv("AB\_NYC\_2019.csv")

print(df.head(10))

x=df.price

y=df.availability\_365

plt.scatter(x, y)

plt.xlabel('price')

plt.ylabel("availability")

plt.title('price x availability')

plt.show()

#hyperparamters

learning\_rate = 0.0001

initial\_b = 0

initial\_m = 0

num\_iterations = 10

def compute\_cost(b, m):

total\_cost = 0

N = float(len(df))

#Compute sum of squared errors

for i in range(0, len(df)):

x = (df['price'].values[i])

y = (df['availability\_365'].values[i])

total\_cost += (y - (m \* x + b)) \*\* 24

#Return average of squared error

return total\_cost/N

def gradient\_descent\_runner(starting\_b, starting\_m, learning\_rate, num\_iterations):

b = starting\_b

m = starting\_m

cost\_graph = []

#For every iteration, optimize b, m and compute its cost

for i in range(num\_iterations):

cost\_graph.append(compute\_cost(b, m))

b, m = step\_gradient(b, m, learning\_rate)

return [b, m, cost\_graph]

def step\_gradient(b\_current, m\_current, learning\_rate):

m\_gradient = 0

b\_gradient = 0

N = float(len(df))

#Calculate Gradient

for i in range(0, len(df)):

x = (df['price'].values[i])

y = (df['availability\_365'].values[i])

m\_gradient += - (2/N) \* x \* (y - (m\_current \* x + b\_current))

b\_gradient += - (2/N) \* (y - (m\_current \* x + b\_current))

#Update current m and b

m\_updated = m\_current - learning\_rate \* m\_gradient

b\_updated = b\_current - learning\_rate \* b\_gradient

#Return updated parameters

return b\_updated, m\_updated

b, m, cost\_graph = gradient\_descent\_runner(initial\_b, initial\_m, learning\_rate, num\_iterations)

#Print optimized parameters

print ('Optimized b:', b)

print ('Optimized m:', m)

#Print error with optimized parameters

print ('Minimized cost:', compute\_cost(b, m))

plt.plot(cost\_graph)

plt.xlabel('No. of iterations')

plt.ylabel('Cost')

plt.title('Cost per iteration')

plt.show()

#Plot dataset

print(x)

print(y)

plt.scatter(x, y)

#Predict y values

pred = m \* x + b

#Plot predictions as line of best fit

plt.plot(x, pred, c='r')

plt.xlabel('price')

plt.ylabel('availability')

plt.title('price x availability')

plt.show()

**APPENDIX C:** Clustering Neighbourhood and Price

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

df = pd.read\_csv("AB\_NYC\_2019.csv")

df.head()

columns = ['neighbourhood','price']

df = df[columns]

df.head()

df.isna().sum()

df.neighbourhood.value\_counts()

len(df.neighbourhood.unique())

plt.figure(figsize = (10,5))

plt.hist(df.price,bins = 600)

plt.show()

other\_than\_0\_mask = df.price != 0

df = df[other\_than\_0\_mask]

original\_price = df['price']

df['price'] = np.log(df['price'])

plt.figure(figsize = (10,5))

plt.hist(df.price,bins = 50)

plt.show()

Like many Unsupervised Learning Algorithms, it's essential to ensure that our input data is entirely numeric. In our dataset, the only categorical data pertains to neighborhoods. To address this, we will employ a technique known as target encoding. Target encoding strives to transform categorical data by incorporating information from the target variable it aims to predict. Although our primary goal may not be predicting price based on neighborhood, this method can effectively encode the data while still capturing any existing relationship between the two variables.

data\_encoded = pd.DataFrame()

data\_encoded['price'] = df['price']

means = df.groupby('neighbourhood')['price'].mean().to\_dict()

data\_encoded['neighbourhood\_encoded'] = df['neighbourhood'].map(means)

from sklearn.covariance import EllipticEnvelope

pred\_eliptic = EllipticEnvelope(contamination=0.1).fit(data\_encoded).predict(data\_encoded)

outlier\_mask = pred\_eliptic == -1

data\_encoded = data\_encoded[~outlier\_mask]

len(data\_encoded)

Standardization is a data preprocessing technique in statistics and machine learning that transforms numerical data to have a mean of zero and a standard deviation of one. It involves subtracting the mean of the data from each data point and then dividing by the standard deviation.

from sklearn.preprocessing import StandardScaler

# Standardize the 'price' column

scaler = StandardScaler()

data\_standarized = scaler.fit\_transform(data\_encoded)

def plot\_data(data,labels):

# Plot the clusters in two dimensions

x1 = data[:,0]

x2 = data[:,1]

plt.figure(figsize = (7,7))

plt.scatter(x1, x2,

c = labels,

s = 1)

plt.title('Pre-processed data and clusters')

plt.xlabel("x\_1",fontsize = 13)

plt.ylabel("x\_2",fontsize = 13)

plt.show()

plot\_data(data\_standarized,None)

The silhouette score is a metric used to evaluate the performance of clustering algorithms. It measures how well each data point fits into its assigned cluster, based on both the distance between the point and other points in the same cluster (intra-cluster distance) and the distance between the point and points in other clusters (inter-cluster distance)

from sklearn.metrics import silhouette\_score, calinski\_harabasz\_score, davies\_bouldin\_score

def labels\_prop(labels):

return pd.Series(labels).value\_counts()

def score(data,labels):

# Calculate silhouette score

silhouette\_avg = silhouette\_score(data, labels)

print("Silhouette Score: {:.4f}".format(silhouette\_avg))

print(labels\_prop(labels))

K-Means is a popular unsupervised machine learning algorithm used for clustering similar data points into groups or clusters. Here's a brief overview:

Objective: The main goal of K-Means is to partition a dataset into K distinct, non-overlapping clusters, where each data point belongs to the cluster with the nearest mean (centroid).

from sklearn.cluster import KMeans

# Initialize the k-means model with the desired number of clusters (k)

kmeans = KMeans(n\_clusters = 3)

# Fit the model to the data in the dataframe

kmeans.fit(data\_standarized)

# Get the labels for each row in the dataframe

labels = kmeans.predict(data\_standarized)

score(data\_standarized,labels)

plot\_data(data\_standarized,labels)

from sklearn.cluster import DBSCAN

# Assuming you have your data standardized and combined as X\_combined

# If not, please refer to the previous responses for data preprocessing

# Create a DBSCAN model

dbscan = DBSCAN(eps=0.1, min\_samples=5) # You can adjust eps and min\_samples as needed

# Fit the model to your data

clusters = dbscan.fit\_predict(data\_standarized)

# Check the number of clusters found by DBSCAN

n\_clusters = len(set(clusters)) - (1 if -1 in clusters else 0)

n\_noise = list(clusters).count(-1)

print(f'Number of clusters found by DBSCAN: {n\_clusters}')

print(f'Number of noise points: {n\_noise}')

score(data\_standarized,clusters)

from sklearn.neighbors import NearestNeighbors

import numpy as np

import matplotlib.pyplot as plt

from tqdm import tqdm

# Fit a k-nearest neighbors model to your data

k\_values = range(1, 100) # You can adjust the range as needed

distances = []

for k in tqdm(k\_values):

neighbors\_model = NearestNeighbors(n\_neighbors=k)

neighbors\_model.fit(data\_standarized)

avg\_distances, \_ = neighbors\_model.kneighbors()

distances.append(np.mean(avg\_distances[:, -1]))

# Plot the average distances for different k values

plt.figure(figsize=(10, 6))

plt.plot(k\_values, distances, marker='o', linestyle='-', color='b')

plt.xlabel('Number of Neighbors (k)')

plt.ylabel('Average Distance')

plt.title('Average Distance to k-Nearest Neighbors')

plt.grid(True)

plt.show()

plot\_data(data\_standarized,clusters)

A Gaussian Mixture Model (GMM) is a probabilistic model used in statistics and machine learning. It represents a dataset as a combination of multiple Gaussian distributions, each associated with a specific cluster or component. GMMs are employed for tasks such as clustering and density estimation. They capture complex data patterns by modeling the underlying probability distribution as a mixture of simpler Gaussian distributions, allowing them to handle datasets with non-uniform shapes and varying densities effectively. GMMs are characterized by parameters like means, variances, and mixing coefficients, which are estimated from the data through methods like the Expectation-Maximization (EM) algorithm.

import pandas as pd

import numpy as np

from sklearn.mixture import GaussianMixture

# Assuming 'neighbourhood' and 'price' are your feature columns

# Initialize and fit the GMM model

n\_clusters = 7 # You can specify the number of clusters you want

gmm = GaussianMixture(n\_components=n\_clusters)

gmm.fit(data\_standarized)

# Predict cluster assignments

cluster\_assignments = gmm.predict(data\_standarized)

# You can access the means and covariances of each cluster like this:

cluster\_means = gmm.means\_

cluster\_covariances = gmm.covariances\_

# You can explore the results or visualize the clusters as needed

score(data\_standarized,cluster\_assignments)

plot\_data(data\_standarized,cluster\_assignments)

To determine the optimum number of clusters a good technique is to test for many clusters and storet a score. In this specific case, the BIC score is more appropriate.

The Bayesian Information Criterion (BIC) is a measure used in Gaussian Mixture Models (GMMs) to find the best number of clusters. It balances model fit and complexity by penalizing complex models. In GMMs, lower BIC scores indicate a better trade-off between model fit and simplicity, helping select the right number of clusters for the data.

import numpy as np

import matplotlib.pyplot as plt

from sklearn.mixture import GaussianMixture

from tqdm import tqdm

# Assuming you have your data in 'features'

# Fit GMM with a range of cluster numbers

num\_clusters = range(1, 11)

bic\_scores = []

for n in tqdm(num\_clusters):

gmm = GaussianMixture(n\_components=n)

gmm.fit(data\_standarized)

bic\_scores.append(gmm.bic(data\_standarized))

# Plot the BIC scores

plt.figure(figsize=(8, 4))

plt.plot(num\_clusters, bic\_scores, marker='o', linestyle='-', color='b')

plt.xlabel('Number of Clusters')

plt.ylabel('BIC Score')

plt.title('BIC Score vs. Number of Clusters')

plt.grid(True)

plt.show()

The sudden drop in the BIC score at 7 clusters suggests that the model fits the data much better at 7 clusters than at 6 or 8 clusters . This could be due to the data having a natural clustering at 7 clusters, or it could be due to the model overfitting the data at 6 or 8 clusters .

While the Gaussian Mixture Model may not exhibit a particularly high silhouette score, it stands out as the most intriguing option among all the models. This is primarily due to its unique ability to demonstrate a substantial distinction when varying the number of clusters. This suggests that the model possesses the capability to identify inherent clusters within the data, an assertion supported by the relatively low BIC score associated with seven clusters when compared to other cluster numbers.

Based on these observations, we can confidently conclude that the GMM model, configured with seven clusters, yields the most favorable results among all the models considered.

plot\_data(data\_standarized,cluster\_assignments)

import pandas as pd

original\_df = pd.read\_csv('/content/drive/MyDrive/Fiverr/Client 7 - Lisa/AB\_NYC\_2019.csv')

# Create a new column 'Cluster' in the original dataset

original\_df['Cluster'] = -1 # Initialize all rows with -1

# Assign clusters to the rows present in the final dataset

original\_df.loc[data\_encoded.index, 'Cluster'] = cluster\_assignments

# Now, your original DataFrame has 'Cluster' values assigned, with -1 for deleted lines

original\_df

original\_df.to\_csv('original\_dataset\_with\_clusters.csv')

**APPENDIX D:** Kprototype Neighbourhoods and Availability & Neighbourhoods and Price

import numpy as np

import pandas as pd

import sklearn.cluster

from sklearn import preprocessing

from sklearn.preprocessing import LabelEncoder

import matplotlib.cm as cm

from kmodes.kmodes import KModes

from kmodes.kprototypes import KPrototypes

import matplotlib.pyplot as plt

from sklearn.preprocessing import StandardScaler

df = pd.read\_csv("AB\_NYC\_2019.csv")

prices = df["price"].to\_numpy()

nhbs = df["neighbourhood"].to\_numpy()

df2 = df[["price", "neighbourhood"]].to\_numpy() #selecting parameters of interest: modular

"""

kproto = KPrototypes(n\_clusters = 5, init = 'Cao', n\_jobs = -1)

clusters = kproto.fit\_predict(df2, categorical = 1)

pd.series(clusters).value\_counts()

"""

costs = []

n\_clusters = []

clusters\_assigned = []

for i in range(1,10): #running for optimal k

try:

kproto = KPrototypes(n\_clusters = i, init = 'Cao', verbose = 2)

clusters = kproto.fit\_predict(df2, categorical = 1)

costs.append(kproto.cost\_)

n\_clusters.append(i)

clusters\_assigned.append(clusters)

except:

print("Invalid with {i} clusters")

#optimal k determined through elbow method

fig, ax = plt.subplots()

clt = ax.plot(n\_clusters, costs, marker = 'o')

plt.show()

import numpy as np

import pandas as pd

import sklearn.cluster

from sklearn import preprocessing

from sklearn.preprocessing import LabelEncoder

import matplotlib.cm as cm

from kmodes.kmodes import KModes

from kmodes.kprototypes import KPrototypes

import matplotlib.pyplot as plt

from sklearn.preprocessing import StandardScaler

df = pd.read\_csv("AB\_NYC\_2019.csv")

prices = df["price"].to\_numpy()

nhbs = df["neighbourhood"].to\_numpy()

df2 = df[["price", "neighbourhood"]].to\_numpy() #selecting parameters of interest: modular

"""

kproto = KPrototypes(n\_clusters = 5, init = 'Cao', n\_jobs = -1)

clusters = kproto.fit\_predict(df2, categorical = 1)

pd.series(clusters).value\_counts()

"""

costs = []

n\_clusters = []

clusters\_assigned = []

for i in range(1,10): #running for optimal k

try:

kproto = KPrototypes(n\_clusters = i, init = 'Cao', verbose = 2)

clusters = kproto.fit\_predict(df2, categorical = 1)

costs.append(kproto.cost\_)

n\_clusters.append(i)

clusters\_assigned.append(clusters)

except:

print("Invalid with {i} clusters")

#optimal k determined through elbow method

fig, ax = plt.subplots()

clt = ax.plot(n\_clusters, costs, marker = 'o')

plt.show()