Short Term Rental Analytics: Airbnb Berlin, Germany

Introduction to Data Science Term Project

1st Alper Yusuf Dursun
1600001752
Department of Computer Science Engineering
Istanbul Kultur University

2nd Ela Nur Zamur
1800002622
Department of Computer Science Engineering
Istanbul Kultur University

***Abstract*—In this research, we used the data from "Insideairbnb" website which is scraped as of September 15, 2022. We used the listings.csv and reviews.csv files for Berlin from this website to create the data set needed. Review.csv is used as basis data for our visualization and statistical analyses and is merged with listings file by taking only some columns that we might use on our analysis from the second file. First we have done some data cleaning and preprocessing so that our analyses made more sense. After that, starting with visualization, we have visualized some important points to answer some questions that might be critical for a customer who wishes to book a hotel room or apartment, etc. After that we have prepared a statistical analysis section and gone into details with testing part. To conclude, we answered the questions that we came up with to satisfy a customer's needs. Some of our questions are listed below:**

- **What are the top districts for review value score?**
- **Does host's verification status effect the number of reviews that hosts get?**
- **Does listing being instant bookable or not effect the price?**
- **How is the price-wise distribution of listings across the entire city?**
- **How does distance to popular touristic places effect the prices?**

List of the questions and analyses we made goes on and on.

I. Introduction

Berlin, the capital of Germany and the country's largest city, is also a major center of politics, culture, media, and science. Noted for its cultural flair, Berlin is home to the world famous Reichstag building and German Historical Museum, while its diverse art scene encompasses hundreds of galleries, events, and museums, including those on Museum Island, a UNESCO World Heritage Site. Between 2015 and 2019, Berlin was in the league of metropolises with over 30 million overnight stays in hotels and guest houses. During these visits, people tend to prefer Airbnb accommodations over hotels mostly because of practical benefits (low cost, convenient location and household amenities). In this research, we tried to analyze Airbnb Berlin data set and answer some questions about the data.

II. Data Preprocessing

A. Merging Datasets

Initially, we gathered 2 columns from the "listings.csv" to our main data set "review.csv". Those columns were "host_identity_verified" and "instant_bookable". The process was made as seen in the figure below,

```
inst_book = berlin_detailed['instant_bookable']
host_verify = berlin_detailed['host_identity_verified']
berlin = berlin.join(inst_book)
berlin = berlin.join(host_verify)
```

Fig. 1. Merging 2 datasets

B. Data Cleaning

After the merge, we wanted to check the shape of our data set with

```
berlin.shape
```

function. The output we got was as below.

```
(16680, 20)
```

Fig. 2. Shape method for data set

According to the output above, we have 16680 listings with 20 different attributes. With the use of

```
berlin.dtypes
```

function we wanted to dive into our columns and get some knowledge about their data types. The output we got can be seen at the figure below,

After this stage, we have decided that we should drop some columns like host_name, id, last_review and license. Since these data are irrelevant or unethical for our analysis we dropped them via,

After we dropped those columns, we checked if any columns has null values with,

```
berlin.isnull().sum()
```

and the output was like below:
According to the output above, we discovered that our data set had some missing values. We treated them as insignificant data so we filled them all with 0 via,

```
berlin.fillna(0,inplace=True)
```

```
id                                 int64
name                              object
host_id                            int64
host_name                         object
neighbourhood_group               object
neighbourhood                     object
latitude                         float64
longitude                        float64
room_type                         object
price                              int64
minimum_nights                     int64
number_of_reviews                  int64
last_review                       object
reviews_per_month                float64
calculated_host_listings_count     int64
availability_365                   int64
number_of_reviews_ltm              int64
license                           object
instant_bookable                  object
host_identity_verified            object
```

Fig. 3. Data types of columns

```
berlin.drop(['id','host_name','last_review', 'license'],axis=1,inplace=True)
```

Fig. 4. Dropping irrelevant columns

So since our data set had no missing(NaN) values anymore, our analysis would make more sense.

III. Exploratory and Statistical Analysis

Before we get our questions answered, we had to do some exploring to know what category of data we could've worked with. So by starting with neighbourhood groups we wanted to know how many regions we have in the data set via,

```
berlin['neighbourhood_group'].unique()
```

As we can see from the output, we discovered that there are 12 unique neighbourhood groups(regions). As the next step, we decided to check the room types to have some knowledge about them. So we executed the following code,

```
berlin['room_type'].unique()
```

```
name                              20
host_id                            0
neighbourhood_group                0
neighbourhood                      0
latitude                           0
longitude                          0
room_type                          0
price                              0
minimum_nights                     0
number_of_reviews                  0
reviews_per_month               2868
calculated_host_listings_count     0
availability_365                   0
number_of_reviews_ltm              0
instant_bookable                   0
host_identity_verified            16
```

Fig. 5. Checking for null values

```
array(['Tempelhof - Schöneberg', 'Marzahn - Hellersdorf',
       'Steglitz - Zehlendorf', 'Pankow', 'Treptow - Köpenick',
       'Charlottenburg-Wilm.', 'Friedrichshain-Kreuzberg', 'Neukölln',
       'Mitte', 'Lichtenberg', 'Spandau', 'Reinickendorf'], dtype=object)
```

Fig. 6. Neighbourhood Groups (Regions)

And the output was,

```
array(['Entire home/apt', 'Private room', 'Shared room', 'Hotel room'],
      dtype=object)
```

Fig. 7. Neighbourhood Groups (Regions)

We discovered that there are 4 different room types, Entire home/apt, Private room, Shared room, Hotel room.

A. Analysis with Visuals

To enrich our project, we used visuals like graphs, plots, charts so that our analysis would make more sense and certain information would be deduced by analyzing the visuals we created.

We started our visualization with top 10 hosts with most listings.

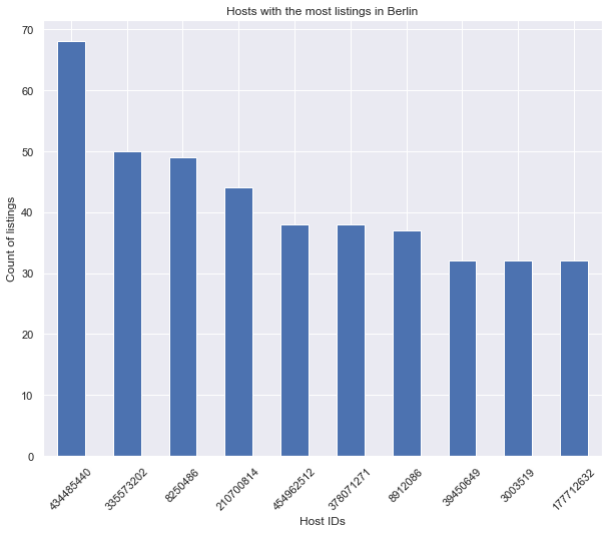


Fig. 8. Top 10 Hosts' Listings Numbers Groups (Regions)

From the chart above, we can see the total listings of top 10 hosts is only 2,67% (445 listings) of the whole data set (16680 listings).

Host with ID 434485440 has the most listings between top 10 hosts with 68 listings.

Next, we visualize the proportion of the listing count on each area using the 'neighbourhood_group' column.

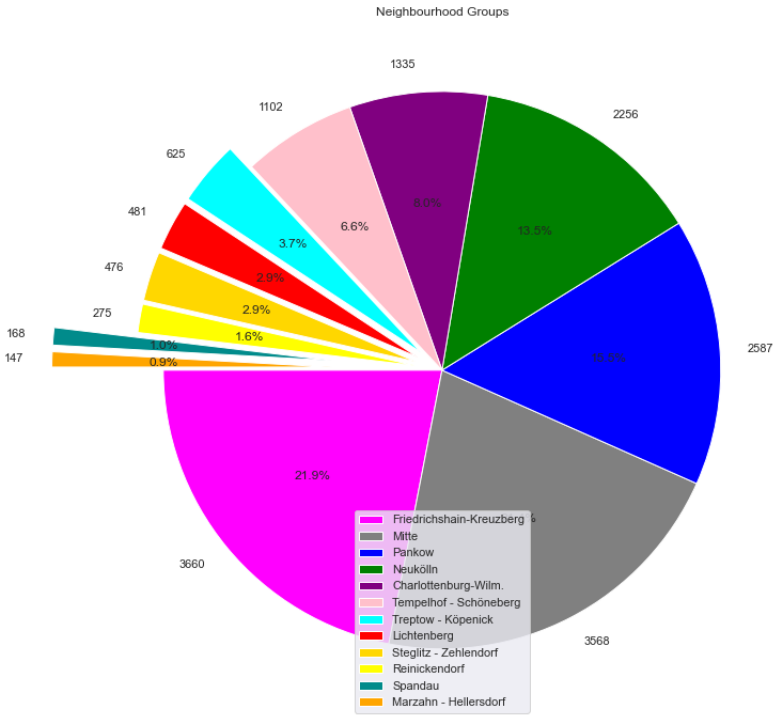


Fig. 9. Listings' Distribution among Neighbourhood Groups (Regions)

From the pie chart above, we found out that the Friedrichshain-Kreuzberg region has the most listings with 3660 listings number, covering 21.9% of the total listings.
Marzahn-Hellersdorf and Spandau Regions have the least amount of listings with 0.9%(147) and 1.0%(168), respectfully.
Next, we looked up the top 10 neighbourhood areas that have the highest number of listings.

```
Alexanderplatz              937
Frankfurter Allee Süd FK    906
Tempelhofer Vorstadt        813
Reuterstraße                651
Brunnenstr. Süd             635
Rixdorf                     515
Neuköllner Mitte/Zentrum    491
südliche Luisenstadt        473
Prenzlauer Berg Südwest     427
Schillerpromenade           424
Name: neighbourhood, dtype: int64
```

Fig. 10. Top 10 neighbourhoods on listing counts

As we can see, Alexanderplatz has the highest number of listings with 937 listings. Since there are so many neighbourhoods, we did not deal with them and used neighbourhood groups instead.
On next step, to create a map of the listing location, we used the 'longitude' and 'latitude' column. But first, we needed to check the values within the column with,

```
coordinates = berlin.loc[:,['longitude',
'latitude']]
coordinates.describe()
```

And the output was as shown as below,

	longitude	latitude
count	16680.000000	16680.000000
mean	13.404151	52.509265
std	0.071314	0.036103
min	13.072850	52.332300
25%	13.365737	52.489050
50%	13.414495	52.509575
75%	13.439420	52.532913
max	13.769390	52.673960

Fig. 11. Description of listings' coordinates

From the above figure, we had the information of the center of the listings(mean). That was useful for our mapping visuals.

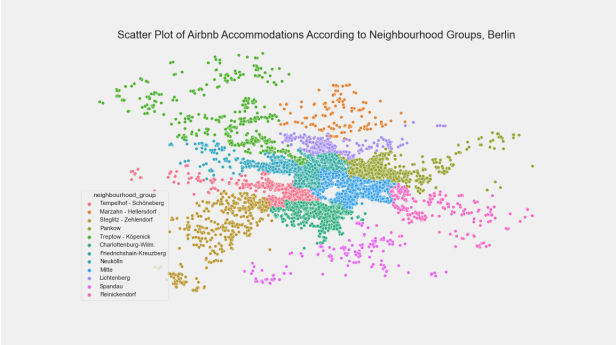


Fig. 12. Scatter Plot of Airbnb Accommodations According to Neighbourhood Groups, Berlin

Now we can see how the listings are plotted into a map. For a better understanding of the listings density, we can use the folium heat map.

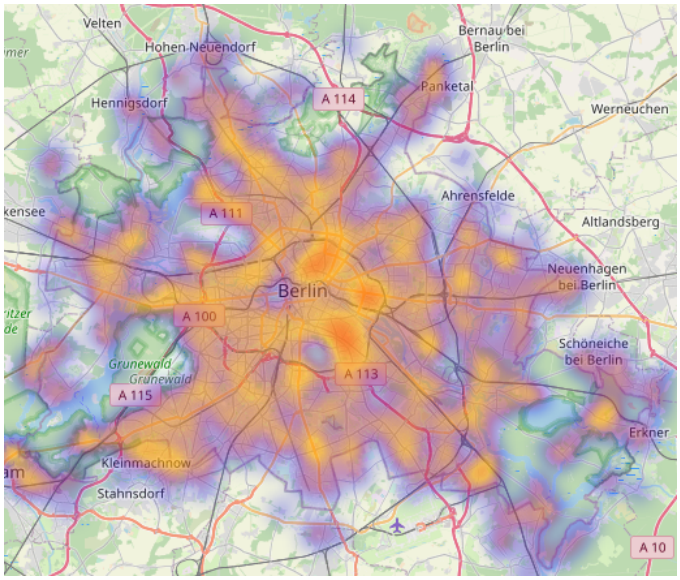


Fig. 13. Scatter Plot of Airbnb Accommodations According to Neighbourhood Groups, Berlin

From the map above, we can see clearly that density of the listings are almost equally distributed, shown by the redish color in the heat map. The listing density increasingly declining the more it's farther away from the center of the city, with some exceptions at certain areas.
To clear data, we removed some extra outliers, we got rid of price values which above €300.

```
berlin_1 = berlin[(berlin.price <= 300) &
(berlin.price >=1)]
```

To have some knowledge about the distribution of prices around the city, we can create a color map.

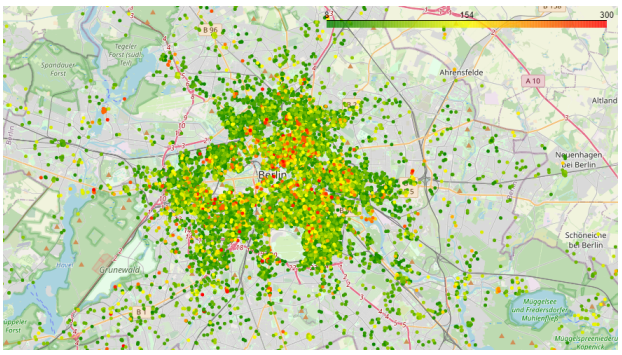


Fig. 14. Color Map of Listings' Prices in Berlin

According to the map above, we found out that most of the expensive listings are located around the center of the city.

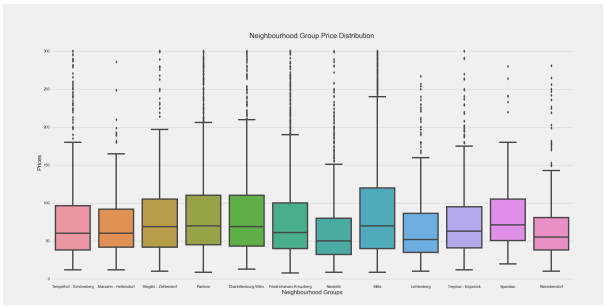


Fig. 15. Boxplot of Price Distribution Among Neighbourhood Groups

As we can see, we still have some outliers. But the amount is less than before. At the same time, we observed that Mitte region has the highest price and highest Q3 value with a median around €70.
Next, we explored deeper on the property detail by finding out what the most used word in the listing name. The most used word could represent the selling value of their property for the prospective guests. First, we created a function to collect the words. Then with the help of Counter library, we listed the word counts into variables and plotted them using Bar plot.

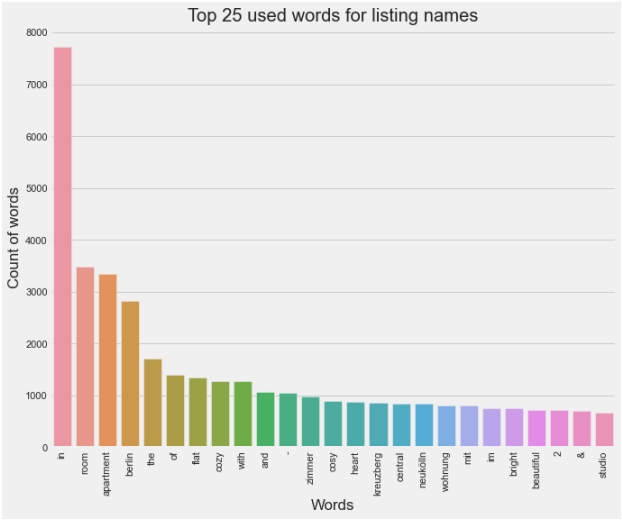


Fig. 16. Top 25 Words Used in Listings

We found out that every area has at least 50% Entire home/apt listings. Next, we figured out the top 10 listings based on their number of reviews to know the most popular Airbnb listings in Berlin.

```
top10_listings = berlin.nlargest(10,
'number_of_reviews')
top10_listings
```

We found out that "Nena Apartments Metropolpark Studio" has the highest number of reviews with 1483, and we also discovered that the majority of the most reviewed listings are located in the Pankow Neighbourhood, with 4 out of 10 listings.

B. Statistical Analysis

We started out with average price between top 10 listings, which is €152.20. Also 6 of them are "Entire home/apt" room type. The most popular listing is also one of the most expensive listings with price €219. This makes sense since it's a "Entire home/apt" room type rather than Private Room.

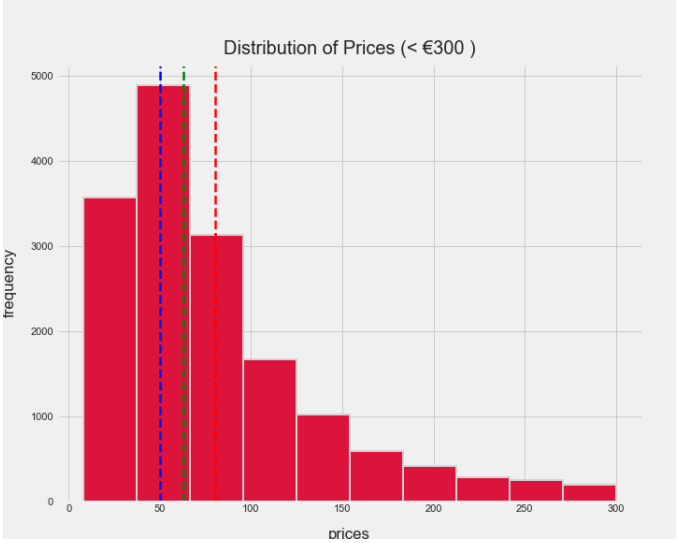
```
average_price = top10_listings.price.mean()
print('Average price per night: € {}'.format(average_price))
```

```
Average price per night: € 152.2
```

Next, we started out with finding the mean, median and mode of the prices of the listings. Since we want it to make some sense, we used the set where we removed some outliers(Price¡300).

```
a = berlin_1['price'].mean()
b = berlin_1['price'].median()
c = float(berlin_1['price'].mode())
print(a)
print(b)
print(c)
80.33897247019165
63.0
50.0
```

With these values, we created a histogram to see how the prices were distributed.



From the look of our graph, we realized that it's a right skewed (positive skewness) graph. It means the more the prices go, the less listings exist.
Next, we analyzed the number of reviews that listings have. Since there might be listings with 0 reviews, we decided to remove listings with 0 reviews and checked their statistics with,

```
temp = berlin[(berlin['number_of_reviews'] > 0)]
print(temp2['number_of_reviews'].mean())
print(temp2['number_of_reviews'].median())
print(float(temp2['number_of_reviews'].mode()))
33.53373877787431
9.0
1.0
```

We found out that average number of reviews that listings have is 33.53, the middle listing has 9 reviews and majority of listings have 1 review.
Next we decided to separate listings depending on their room types and minimum night conditions to analyze their statistics.

- We found out that average price for Entire/home apartment type listings with 5 minimum nights booking is $127.60 per day.
- For long-term stays, we limited the data to more than 30 nights booking with the same room type. And we found out that the average price is $90, which is less than before. So we can say that long-term bookings are cheaper with Entire/home apartment type.
- We kept going with hotel rooms. We found out that hotel room prices for 5 days or less bookings are high in average which is $331. We also discovered that there are no hotel room bookings for 30 days or more.
- For private rooms, the average price for 5 days or less is $60, for 30 days or more it's $53. So if a customer can't decide their booking duration. It's safe to say that it will be cheaper to book for 30 days or more.
- If we go deeper on shared rooms, the average price for 5 days or less is $78, for 30 days or more it's $66.

IV. Sampling Methods

In this step, we performed some sampling methods and decide which one represents the data best. To test which one fits better, we take the mean of the price of all data set first. Then we take the mean of price again in different sampling methods.

```
print(berlin_1['price'].mean())
80.33897247019165
```

As we can see from above, our total mean price is $80.33. There are several sampling methods.

- First one we used is random sampling. It's one of the easier ones. With the help of the "random" library, we imported "sample". After we defined our sampling lists and mean function, we found out with random sampling we get mean price as $79.30. But since it's random sampling it's not reliable.
- Next sampling method we used is systematic sampling, which is taking samples according to a variable "step" which we defined as 5.

```
# Define systematic sampling function
def systematic_sampling(df,step):
  indexes=np.arange(0,len(df),step=step)
  systematic_sample=df.iloc[indexes]
  return(systematic_sample)

# Obtain a systematic sample and save it in a new variable
systematic_sample=systematic_sampling(berlin_1,5)
```

```
print(systematic_sample['price'].mean())
```

```
79.66822721598002
```

As you can see from above output, with systematic sampling we got $79.66 as our mean price. It's closer to our real mean price than random sampling.

- Next we got cluster sampling, we divided the data set into subgroups and name these groups by numbers and we selected a subgroup every T time. We got $88.60 as our mean price which is the worst method for our case until now.

V. Testings

For our testings, we came up with some hypothesises to test our data. Next we dived into those hypothesises.

- #H0 (Null Hypothesis) : Distribution of the prices of listings with "Hotel room" type are normal.
  #H1 (Alternate Hypothesis) : Distribution of the prices of listings with "Hotel room" type are not normal.

```
st.shapiro(berlin_hotel.price)
```

```
ShapiroResult(statistic=0.9496127963066101, pvalue=0.011832049116492271)
```

As we can see from the figure above, since p-value is less than 0.05, we can say that distribution of the prices of listings with "Hotel room" type are not normal.

- #H0 (Null Hypothesis) : Distribution of the prices of listings with "Shared room" type are normal.
  #H1 (Alternate Hypothesis) : Distribution of the prices of listings with "Shared room" type are not normal.

```
Out[120]: ShapiroResult(statistic=0.8379092216491699, pvalue=4.487642896178201e-13)
```

As we can see from the figure above, p value is almost 0, we can say that distribution of the prices of listings with "Shared room" type are not normal.

- #H0 (Null Hypothesis) : Distribution of the prices of listings with "Private room" type are normal.
  #H1 (Alternate Hypothesis) : Distribution of the prices of listings with "Private room" type are not normal.

```
C:\Users\Alper\anaconda3\lib\site-packages\scipy\stats\morestats.py:1760: UserWarning:
p-value may not be accurate for N > 5000.
ShapiroResult(statistic=0.7026438117027283, pvalue=0.0)
```

For tests that include over 5000 values, Shapiro test is not accurate.*Buraya ekleme yapılacak*. So we moved on with Levene test between Shared room, Entire home/apt and Private room types .

- #H0 (Null Hypothesis) : The variances between 3 room types are not significantly different from each other.
  #H1 (Alternate Hypothesis) : The variances between 3 room types are significantly different from each other.

```
st.levene(berlin_prv.price, berlin_shared.price, berlin_apt.price)
```

```
LeveneResult(statistic=500.6969800292686, pvalue=1.2650162321393243e-211)
```

As we can see from the code above, p value is less than 0.05 so we rejected the null hypothesis and accepted the alternate hypothesis which means the variances between 3 room types are significantly different from each other.

Next we did an ANOVA test to see if there's a difference amongst the average prices of top 5 regions. We knew that from our previous analysis, top 5 regions are Friedrichshain-Kreuzberg, Mitte, Pankow, Neukölln and Charlottenburg-Wilm. So we gathered data which belongs to these regions via,

```
friedkreuz = berlin_1[berlin_1['neighbourhood_group'] == 'Friedrichshain-Kreuzberg']
mitte = berlin_1[berlin_1['neighbourhood_group'] == 'Mitte']
pankow = berlin_1[berlin_1['neighbourhood_group'] == 'Pankow']
neukolln = berlin_1[berlin_1['neighbourhood_group'] == 'Neukölln']
charlotwilm = berlin_1[berlin_1['neighbourhood_group'] == 'Charlottenburg-Wilm.']
```

After we created 5 sub data frame for our testings, we came up with a null and alternate hypothesis.

- H0 (Null Hypothesis) : The average prices of 5 top regions are not significantly different from each other.
  H1 (Alternate Hypothesis) : The average prices of 5 top regions are significantly different from each other.

After we set our hypothesis, we ran our code,

```
st.f_oneway(friedkreuz.price,mitte.price,pankow.price,neukolln.price,charlotwilm.price)
```

```
F_onewayResult(statistic=95.87308364445825, pvalue=1.643820607595249e-80)
```

According to the results above, we stated that top 5 regions' average prices are significantly different from each other.
We wanted to analyse and test if a host being verified or not has any effect on the prices or on that listing's number of reviews. We also wanted to test out if a listing being instantly bookable or not has any effect on it's price. So we created 4 different sub data frame for our testings via,

```
verified = berlin_1[berlin_1['host_identity_verified'] == 't']
not_verified = berlin_1[berlin_1['host_identity_verified'] == 'f']
instant_bookable = berlin_1[berlin_1['instant_bookable'] == 't']
not_instant_bookable = berlin_1[berlin_1['instant_bookable'] == 'f']
```

After we created our data frames, we did our testings with following hypothesises.

- H0 (Null Hypothesis) : Host's verification status has no effect on their total number of reviews.
  H1 (Alternate Hypothesis) : Host's verification status has effect on their total number of reviews.

```
st.f_oneway(not_verified.number_of_reviews,verified.number_of_reviews)
```

```
F_onewayResult(statistic=0.5134127244137988, pvalue=0.4736756916296123)
```

According to the output above, we stated that since p-value is more than 0.05, host's verification status has no effect on their total number of reviews.

Next with the same data frames we tested the prices.

- H0 (Null Hypothesis) : Host's verification status has no effect on their prices.
  H1 (Alternate Hypothesis) : Host's verification status has effect on their prices.

```
st.f_oneway(verified.price,not_verified.price)
```

```
F_onewayResult(statistic=57.597348567821165, pvalue=3.39333179168695e-14)
```

Since p value is very less than alpha=0.05 we concluded that, host's verification status has no significant effect on prices.
Next we tested the instant bookable data to check whether a listing being instant bookable or not has significant effect on it's price.

- H0 (Null Hypothesis) : Listing being instant bookable has no significant effect on prices.
  H1 (Alternate Hypothesis) : Listing being instant bookable has significant effect on prices.

And we did our test via, From the result above, since the

```
st.f_oneway(instant_bookable.price, not_instant_bookable.price)
```

```
F_onewayResult(statistic=65.81151266526545, pvalue=5.318619348837782e-16)
```

p value is less than 0.05, we rejected the null hypothesis and stated that a listing being instant bookable has significant effect on it's price.
Next up we wanted to test out if a listing's distance to touristic places has any any effect on it's price. We did some research and gathered 5 popular touristic places and used their longitude and latitude to create a dictionary like,

```
touristic_places_coord = {"Brandenburg Gate":[(52.516266,13.377775)],
                          "Reichstag":[(52.518623,13.376198)],
                          "Museum Island":[(52.516640,13.402318)],
                          "Berlin Wall Memorial":[(52.535152, 13.390206)],
                          "German Historical Museum":[(52.517664,13.390998)]}
```

With the help of pygeodesic library in python, we created 5 different columns that tells us the listings' distance to those 5 touristic places in kilometers.
After that we created another column called "minimum_distance" that selects the minimum distance to one of those 5 touristic places for every listing in the data set.
Next up, we divided minimum distance column into 5 different categories like, "very close, close, middle, far, very far". And replaced the results into a new column called "distance_category".
Lastly, we created 5 different sub data frames for our testing purposes like,

```
very_close = berlin_1[berlin_1['distance_category'] == 'very close']
close = berlin_1[berlin_1['distance_category'] == 'close']
middle = berlin_1[berlin_1['distance_category'] == 'middle']
far = berlin_1[berlin_1['distance_category'] == 'far']
very_far = berlin_1[berlin_1['distance_category'] == 'very far']
```

So we made an ANOVA test to see if there's any significant difference between average prices in categories based on distance metrics we calculated with the help of pygeodesic library.

- H0 (Null Hypothesis) : Average distance to most popular 5 touristic places has no effect on the average price.
  H1 (Alternate Hypothesis) : Average distance to most popular 5 touristic places has effect on the average price.

```
st.f_oneway(very_close.price, close.price, middle.price, far.price,very_far.price)
```

```
F_onewayResult(statistic=174.87007206313336, pvalue=7.243498956942235e-147)
```

According to the above ANOVA test, we found out that average distance to most popular 5 touristic places has significant effect on the average prices.

VI. Machine Learning Models : Regression Models

To start of the modeling part of our project, we determined some regression and classification models that we could work with. These models are:

- Linear Regression
- Ordinary Least Squares Regression
- Ridge Regression
- ElasticNet Regression
- Random Forest Regression
- XGBoost Regressor
- Support Vector Machine
- Decision Tree
- The K-Nearest Neighbors
- Random Forest Classification

A. A small preparation for modelling

We first started off with preparing for our regression models. We added 3 features from the detailed csv file, which are "beds", "bedrooms" and "accommodates". We dropped some unnecessary columns that we created or used in our analysis. We also replaced the true false indicators in the instant_bookable and host_identity_verified columns with 1 and 0, respectfully via;

```
# We dropped columns like "name", "host_id"
berlin_1.drop(['name', 'host_id'], axis=1, inplace=True)
berlin_1.drop(['distance_category'], axis=1, inplace=True)
berlin_1.replace({'f': 0, 't': 1}, inplace=True)
berlin_1
```

Next, we selected the most effective features for our price prediction, and placed them on X data frame with y data frame being the price column.

```
features = ['minimum_nights', 'number_of_reviews', 'calculated_host_listings_count',
            'host_identity_verified', 'instant_bookable',
            'availability_365', 'minimum_distance']
X = berlin_1[features]
y = berlin_1['price']
```

Next part was to splitting the data into train and test set. We used a 7/3 ratio meaning 30% for test size.

We also wanted to use Ordinary Least Squares(OLS), since OLS results summary provides a brief description on how each feature measures with it's p-value and other test scores, which would be very useful in selecting our features for the model. But it did not help much. So we decided to keep it out.
We kept going with normalizing our data. We used Standard Scaler from the sklearn.preprocessing library. What Standard Scaler actually does is standardizing features by removing the mean and scaling to unit variance to that the data will be scaled down to smaller numbers.

$$z = \frac{x - \mu}{\sigma}$$

After initiating the standard scaler, we fit the train and test values, fit transform and transform, respectfully.

```
# We standardized data
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test  = scaler.transform(X_test)
```

B. Linear Regression

After fitting the data, we initiated the Linear Regression function in the library and fit our train sets to the linear regressor. Next up we created variables such as pred and test pred to see our R2 score and Mean Squared Error values.

```
y_pred = lr.predict(X_train)
y_test_pred = lr.predict(X_test)

print(f"Training Mean Squared Error: {np.sqrt(mean_squared_error(y_train, y_pred))}")
print(f"Train R2 score: {r2_score(y_train, y_pred)}")

print(f"Testing Mean Squared Error: {np.sqrt(mean_squared_error(y_test, y_test_pred))}")
print(f"Test R2 score: {r2_score(y_test, y_test_pred)}")
```

```
Training Mean Squared Error: 95.71384416236948
Train R2 score: -1.7341827042667068
Testing Mean Squared Error: 93.84855921967461
Test R2 score: -1.8083862024255546
```

From the above model, we realized that Linear Regression model did not perform well at all. In fact it had a negative R2 score which could mean that this model is performing worst. Next we tried regularized models and see if they would perform any better, because regularization can prevent over fitting and reduce the model complexity.

C. Ridge Regression

Ridge regression could shrink the coefficients and helps to reduce the model complexity and multicollinearity which we have seen from our OLS works. We also used RidgeCV cross validation, so that we could choose the best regularized alpha value.

After the process, we realized that best alpha value was 10. So we used 10 as our alpha value. Next up we imported Ridge library, fit train sets and predict our y sets. After that,

```
from sklearn.linear_model import RidgeCV

modelCV = RidgeCV(alphas = [0.1, 0.01, 0.05, 0.001,0.0001, 1, 2, 5, 10], store_cv_values = True)
modelCV.fit(X_train, y_train)
alpha = modelCV.alpha_
alpha
```

we printed out our Mean Squared Error values and R2 scores for both training and test sets via,

```
from sklearn.linear_model import Ridge
reg = Ridge(alpha = alpha)
reg.fit(X_train, y_train)

y_reg_pred = reg.predict(X_train)
y_reg_test_pred = reg.predict(X_test)

print(f"Training Mean Squared Error: {np.sqrt(mean_squared_error(y_reg_pred,y_train))}")
print(f"Train R2 score: {r2_score(y_reg_pred,y_train)}")

print(f"Testing Mean Squared Error: {np.sqrt(mean_squared_error(y_reg_test_pred,y_test))}")
print(f"Test R2 score: {r2_score(y_reg_test_pred, y_test)}")
```

and our results was:

```
Training Mean Squared Error: 51.07619162460479
Train R2 score: -2.5225890477767736
Testing Mean Squared Error: 50.673340807140285
Test R2 score: -2.5019129466329693
```

From the results above, even though it did a bit better than Linear Regression model in MSE, it was still bad compared to the results we were expecting for a model. Next, we worked through another model which combines both L1 and L2 regularizations in one model called ElasticNet.

D. ElasticNet Regression

ElasticNet Model is useful when we have more correlated variables(deals with multicollinearity issues). It sets the proper alpha value between 0 and 1. Next, we imported ElasticNet model to train the data with Cross Validation. We wanted to see what the best alpha value would be and what intercept value we would get. So we did the work,

```
from sklearn.linear_model import ElasticNetCV, ElasticNet

alphas = [0.0001, 0.001, 0.01, 0.1, 0.3, 0.5, 0.7, 1]
elastic_cv=ElasticNetCV(alphas=alphas, cv=5)
model = elastic_cv.fit(X_train, y_train)
print(model.alpha_)
print(model.intercept_)
```

```
0.01
80.94666904485865
```

After that with 0.01 as alpha value, we fit our model and printed our MSE and R2 scores for both train and test sets.

As we saw from the R2 scores and Mean squared error, Elastic Net was also not very useful in determining proper price as a dependant variable. So, this model also performed badly. Below figure shows the original scatter points of the testing data set:

Since, the above linear models which we have trained the data on, did not perform well, which comprised of lower accuracy scores and R2 scores on training and testing set, we

```
elastic=ElasticNet(alpha=0.01)
elastic.fit(X_train, y_train)

print(f"Training Mean Squared Error: {np.sqrt(mean_squared_error(y_train,elastic.predict(X_train)))}")
print(f"Train R2 score: {r2_score(elastic.predict(X_train), y_train)}")

print(f"Testing Mean Squared Error: {np.sqrt(mean_squared_error(y_test,elastic.predict(X_test)))}")
print(f"Test R2 score: {r2_score(elastic.predict(X_test), y_test)}")
```

```
Training Mean Squared Error: 51.076269560330836
Train R2 score: -2.5194132932782374
Testing Mean Squared Error: 50.67545670272102
Test R2 score: -2.498825849867456
```

tried to move to non Linear models and check if they would perform better than what was achieved from the comprising linear models.

E. Random Forest Regression

Since the output of the linear model like Linear and Elastic Regression performed not so well on the training and test data that we had (they had a negative R2 score), we decided to use Random Forest ensemble method which is a non linear model. In this, we used a Regressor, since our output contains a Regression task. It uses multiple decision trees and a technique called as bagging. This combines multiple decision trees in determining the final output, rather than relying on individual decision trees.

We also splitted our data into train and test sets, fit X sets with features and y with price, just like the previous models with a 70/30 ratio.

After the split, we called Random Forest Regressor function with random state 42.

```
param_grid = {
    'n_estimators': [500, 700, 1000],
    'max_features': ['auto'],
    'max_depth' : [8, 10, 12],
    'min_samples_split': [5],
    'bootstrap': [True]
}
```

Here we set the parameter grid for selecting the best parameters using GridSearchCV for our Random Forest Regression. It also contains cross validation within the GridSearch on the dataset. This method is called hyperparameter tuning, where optimization is the key factor for selecting the best parameters.

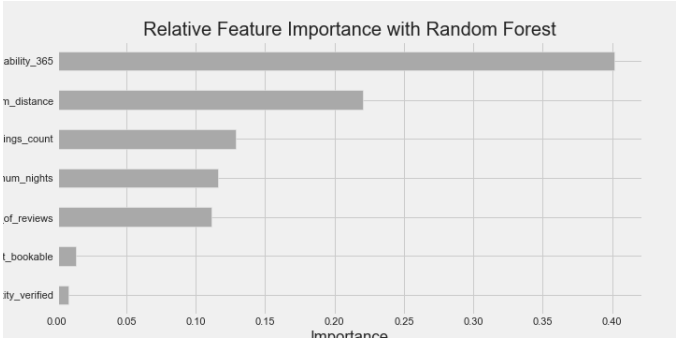
Next up, we initiated the Random Forest Regression with the best parameters obtained from GridSearchCV. We fit the data tho the RFR model and ran the prediction code, and the results were like below:

```
print(f"Training RMSE: {np.sqrt(mean_squared_error(y_train, y_pred))}")
print(f"Train R2 score: {r2_score(y_train, y_pred)}")

print(f"Testing RMSE: {np.sqrt(mean_squared_error(y_test, y_test_pred))}")
print(f"Test R2 score: {r2_score(y_test, y_test_pred)}")
```

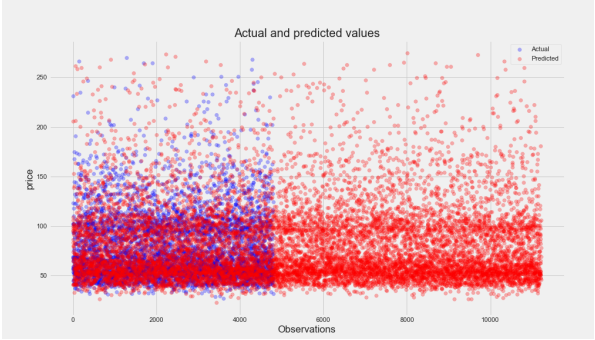
```
Training RMSE: 35.340685314597934
Train R2 score: 0.6272411636951449
Testing RMSE: 45.255044703515416
Test R2 score: 0.34696627877086894
```

From the above scores, we saw that the training score is almost 63% which the model fits very well on seen data. Our testing score is around 34% which does not do a pretty decent job on unseen data, and still there was some room for improvement like selecting some more features(feature engineering), tweaking a few parameters to obtain some more best results(even though not likely every time it succeeds).

After the necessary work, we came up with a feature importance plot:



Then we created a plot that shows the actual and predicted values for price:



It can be seen from the plot above that our model did a pretty good job at detecting the output of the price with regards to the features included in our model.

After this step, we decided we need one hot encoding for our categorical features, so with the help of get_dummies function, we created a transformed data frame, to be used in XGBoost Regression model. After the one hot encoding, we checked on a multi collinearity heat map. And decided to drop every column that would not help our model improve.

After dropping the unnecessary columns in our one hot encoded new data frame, there were still some fairly strong correlations between highly rated properties of different reviews categories. However, these would be left in for now and can be experimented with later to see if removing them improves the model.
Next, the same process for scaling the data and spliting it for x and y steps were done again. After the preps, we created our XGBoost model via the code,

```
# Another Model: Gradient boosted decision trees

xgb_reg = xgb.XGBRegressor()
xgb_reg.fit(X_train, y_train)
training_preds_xgb_reg = xgb_reg.predict(X_train)
val_preds_xgb_reg = xgb_reg.predict(X_test)
print("\nTraining r2:", round(r2_score(y_train, training_preds_xgb_reg),4))
print("Validation r2:", round(r2_score(y_test, val_preds_xgb_reg),4))
```

```
Training r2: 0.7652
Validation r2: 0.5303
```

With XGBoost, our features explained approximately 76% of the variance in our target variable. Overall, we can say that our data was not very compatible with regression models.

VII. Classification Models

To continue of the modeling part of our project, below we stated our classification models as a reminder:

- Support Vector Machine
- Decision Tree
- The K-Nearest Neighbors
- Random Forest Classification

For our classification models, we re-read the airbnb data file, and did some cleaning, such as unnecessary columns, filling NaN values with average value. We also added the same 3 features we used from more detailed csv file.
After preparing the data same as regression, standardizing and other things as previous. We performed binary encoding on the neighbourhood columns. And kept going with Principal Component Analysis(PCA).
PCA is an unsupervised technique used for dimensionality reduction for preprocessing before supervised methods.

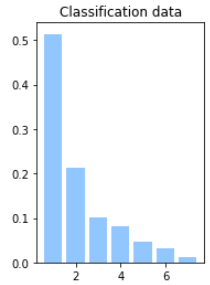
The first 3 components of data set explain about the 80% of the total variance so we used 3 PCs for the PCA. After the PCA process, we turned it back into a data frame.
Next, we defined a dictionary named ”accuracies” that would contain all the accuracies produced by the classifiers we used, and defined the functions used to plot the learning curve. After we got ready, we started off with Support Vector Machine.

A. Support Vector Machine (SVM)

The goal of the support vector classifier (SVC) is to find the hyperplane that separates the classes in feature space. We defined 2 function in Python for both SVM and SVM with GridSearchCV, and in our SVM model we used both linear and rbf kernel. After that, we ran our SVM with data set composed by dummy variables.

```
### Proportion of Variance Explained (PVE) FOR CLA

pca_class = PCA()
x_class_pca = pca_class.fit_transform(x_class)

cvar_arr = pca_class.explained_variance_ratio_
plt.subplot(1, 2, 1)
plt.bar(np.arange(1, len(cvar_arr)+1), cvar_arr)
plt.title('Classification data')
```

```
Text(0.5, 1.0, 'Classification data')
```




```
# Perform the svm, with linear kernel
accuracies["SVM_linear"] = our_SVM_func(dummy_class, y_class, 'linear')

# Perform the svm, with rbf kernel
accuracies["SVM_rbf"] = our_SVM_func_with_GridSearch(dummy_class, y_class, 'rbf')
```

After the process, with linear kernel, our test and train accuracy were both 71%. However, with rbf kernel and GridSearchCV, train accuracy was 78% and test accuracy was 73%. So we can say that, for our data, rbf kernel with GridSearchCV was the better choice for SVM.

B. Decision Tree

The decision tree is a supervised learning algorithm used both for regression and classification tasks. It is based on a simple idea, for each feature in the data set, it forms a node, using the most important attribute in the root node. So, it consists to recursive binary splitting, having a ’condition’ or ’decision’ that impose us if follow the left branch or the right one. The splitting continues until a leaf node is reached, which contains the outcome of the decision tree.
Next up, we defined our decision tree function. We performed the decision tree with dummy variables and max depth equal to 5. Too low value of max depth produces underfitting, while too high produces overfitting. Then we ran our function with using the ’accuracies’ dictionary that we created.

```
accuracies["DecisionTree"] = our_dec_tree_func(dummy_class, y_class, 5)
```

```
Training with maximum depth= 5
        |->End training
Testing with test set:
        |->End testing - Accuracy= 70.91%
Confusion matrix:
[[2536  434]
 [ 678 1355]]
Classification report
              precision    recall  f1-score   support

           0       0.79      0.85      0.82      2970
           1       0.76      0.67      0.71      2033

    accuracy                           0.78      5003
   macro avg       0.77      0.76      0.76      5003
weighted avg       0.78      0.78      0.78      5003
```

Fig. 17. Decision Tree With Dummy Variables

As we can see from the figure above, decision tree with max depth = 5, had around 71% accuracy for our test set.
Next up, we ran our decision tree function, but this time with the principal components found by PCA.

```
accuracies["DecisionTree_PCA"] = our_dec_tree_func(x_class_pca, y_class, 5)
```

```
Training with maximum depth= 5
        |->End training
Testing with test set:
        |->End testing - Accuracy= 45.33%
Confusion matrix:
[[2386  549]
 [1301  767]]
Classification report
              precision    recall  f1-score   support

           0       0.65      0.81      0.72      2935
           1       0.58      0.37      0.45      2068

    accuracy                           0.63      5003
   macro avg       0.61      0.59      0.59      5003
weighted avg       0.62      0.63      0.61      5003
```

Fig. 18. Decision Tree With PCA

This time, the algorithm resulted in 45% accuracy on test set. Which means using PCA with decision tree algorithm did not help.

C. Random Forest Classifier

The random forest algorithm works building a set of decision tree and training them on a sub-set of records of data set randomly picked. In the case of classification, the test sample is evaluated by each tree, so it is assigned to the category that wins the majority vote, between the outcomes produced by each 'internal' classifier. As in all of our classification algorithms, we defined another function for random forest aswell.
First we ran the random forest function with the entire data set, composed by dummy variables.

As we can see from the figure above, random forest classification with dummy variables got us 74% accuracy. After that, we executed the random forest classifier with the principal components found by the PCA.

As we can see from the figure above, Principal Component Analysis once again was not helpful. The model accuracy decreased to 48%.

D. The K-Nearest Neighbors Algorithm (KNN)

The K-Nearest Neighbors algorithm (KNN) is a non-parametric method, which considers the K closest training examples to the point of interest for predicting its class. This is done by a simple majority vote over the K closest points.
First, we defined a function to use for KNN models, that receives the dataset, samples and labels. This function splits the dataset in train and test sets. It also receives a third parameter, for the K value: if it is 0, the function searches for the best k, that provides the lowest mean error, and finally train and test again the classifier, with the best K value found, evaluating the performances. Otherwise, train and test the

```
accuracies["RandomForest"] = our_rf_func(dummy_class, y_class, 0)
```

```
Training random forest classifier with n_estimators= 10
        |->End training
Testing with validation set:
        |->End testing - Accuracy= 70.91%

Training random forest classifier with n_estimators= 20
        |->End training
Testing with validation set:
        |->End testing - Accuracy= 72.25%

Training random forest classifier with n_estimators= 50
        |->End training
Testing with validation set:
        |->End testing - Accuracy= 72.95%

Training random forest classifier with n_estimators= 100
        |->End training
Testing with validation set:
        |->End testing - Accuracy= 73.19%

Training random forest classifier with n_estimators= 1000
        |->End training
Testing with validation set:
        |->End testing - Accuracy= 73.94%

Best number of trees evaluated: 1000
```

Fig. 19. Random Forest Classification With Dummy Variables

```
accuracies["RandomForest_PCA"] = our_rf_func(x_class_pca, y_class, 100)
```

```
Training random forest classifier with n_estimators= 100
        |->End training
Testing with test set:
        |->End testing - Accuracy= 48.47%
Confusion matrix:
[[2261  710]
 [1155  877]]
Classification report
              precision    recall  f1-score   support

           0       0.66      0.76      0.71      2971
           1       0.55      0.43      0.48      2032

    accuracy                           0.63      5003
   macro avg       0.61      0.60      0.60      5003
weighted avg       0.62      0.63      0.62      5003
```

Fig. 20. Random Forest Classification With PCA

classifier with the received K value.
After we defined the function, we initiated it with data set composed by dummy variables:

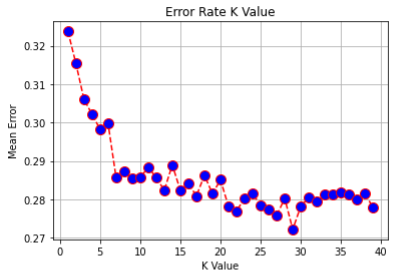
As we can see from the figure above, KNN with dummy variables had 63% accuracy. We continued with principal components:

Once again, PCA was not helpful. Accuracy decreased to 64%.
Here, we are done with our classification algorithms. Next, in Python, we printed out all the accuracy values obtained by the classifiers we used:

As we can see, the accuracies obtained by the algorithms trained with only the principal components, are much more lower with respect to the ones with the entire dataset. This probably might be affected by the fact that there are few features, so not enough to adopt the feature reduction with the PCA. Moreover, also the accuracies with the entire dataset were not too high. Overall, the best classifier, for our dataset

```
accuracies["KNN"] = our_KNN_func(dummy_class, y_class, 0)
```

```
Evaluate K in [1,40]... please wait!
Best K= 29
```

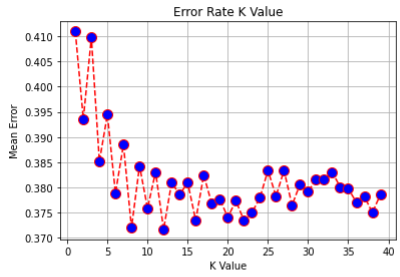



```
Train the model with K= 29
Evaluate the model:
Confusion matrix:
[[1647  328]
 [ 580  781]]
Classification report
              precision    recall  f1-score   support

           0       0.74      0.83      0.78      1975
           1       0.70      0.57      0.63      1361

    accuracy                           0.73      3336
   macro avg       0.72      0.70      0.71      3336
weighted avg       0.73      0.73      0.72      3336

Accuracy: 63.24%
```

Fig. 21. KNN with Dummy Variables

```
accuracies["KNN_PCA"] = our_KNN_func(x_class_pca, y_class, 0)
```

```
Evaluate K in [1,40]... please wait!
Best K= 12
```




```
Train the model with K= 12
Evaluate the model:
Confusion matrix:
[[1636  336]
 [ 904  460]]
Classification report
              precision    recall  f1-score   support

           0       0.64      0.83      0.73      1972
           1       0.58      0.34      0.43      1364

    accuracy                           0.63      3336
   macro avg       0.61      0.58      0.58      3336
weighted avg       0.62      0.63      0.60      3336

Accuracy: 42.59%
```

Fig. 22. KNN with PCA

```
# print the accuracies obtained by the classifiers tested
for key, val in accuracies.items():
  print("%s: %.2f%%" % (key, val))
```

```
SVM_linear: 69.46%
SVM_rbf: 73.03%
DecisionTree: 71.73%
DecisionTree_PCA: 45.33%
RandomForest: 73.00%
RandomForest_PCA: 48.47%
KNN: 63.24%
KNN_PCA: 42.75%
```

Fig. 23. Accuracy Values of Every Classification Model We Used

was Support Vector Machine (SVM) with RBF kernel with 75% accuracy. Second best was Random Forest Classifier with 74% accuracy.

Conclusion

In the first part of our project, we simply performed some EDA and statistical analysis. We have identified various new insight on how the Airbnb listings distributed on Berlin, we found out where are the listings located, mostly in the top 5 regions. We also questioned the relationship between various features like instant bookable and price; host's verification status and number of reviews. We created heat map, scatter map to dive into more details, etc.

In the second part of our project, we applied various regression and classification models to AirBnb Berlin data. We used various models, tried different techniques. We also used Principal Component Analysis even though it was not very helpful. After our work, we found out that best regression model was XGBoost for our data at hand. And the best classification models were Support Vector Machine and Random Forest Clasification.

References

[1] Inside Airbnb Website, http://insideairbnb.com/get-the-data/, scraped at September, 2022.

[2] Folium and Python, https://towardsdatascience.com/creating-a-simple-map-with-folium-and-python-4c083abfff94

[3] Pygeodesic Library Documentation, https://pypi.org/project/pygeodesic/

[4] A similar project for EDA, https://mohamedirfansh.github.io/Airbnb-Data-Science-Project/

[5] PDFs of Lecture 4,5,6,7 and 8 for our classes created by Fatma Patlar Akbulut, https://cats.iku.edu.tr

[6] Scikit-learn website, https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html

[7] Overview of Classification Algorithms, https://stackabuse.com/overview-of-classification-methods-in-python-with-scikit-learn/

[8] Most Popular Regression Algorithms, https://towardsdatascience.com/five-regression-python-modules-that-every-data-scientist-must-know-a4e03a886853